Monads and C++ Template Metaprogramming

Bartosz Milewski

Motivation

- Eric Niebler's (Joel Falcou's) Proto
- Example: Lambda DSL
  - Expressions turned into types
  - Types manipulated at compile time
  - The result: a runtime function object
- Mixing compile-time TMP with runtime execution

Plan

- Convert Haskell Expression monad to C++
  - Expression tree (compile-time)
  - State (runtime)
  - Action (constructed at compile-time, executed at runtime)
  - Metafunctions Bind/Return (compile-time)
  - “Compile” metafunction (compile-time)
- 2-argument lambda EDSL

Teaser

```
template<class L, class R>
struct Compile<Plus<L, R>> : Prog {
    int operator()(Args args) {
        return Bind<Compile<L>, Bind<Compile<R>, Return>> (
            Compile<L>(),
            [](int left) -> Bind<Compile<R>, Return> {
                return Bind<Compile<R>, Return>(
                Compile<R>(),
                    [left](int right) -> Return {
                        return Return(left + right);
                    }
                );
            }
        )(args);
    }
};
```

Domain Specific Languages

- Write a C++ expression which, by
- Abusing C++ operator overloading
- Is interpreted as something completely different:
- An expression tree of very specific type
- The type is processed at compile time resulting in an object that can be
- Executed at runtime to produce a desired result
- What does it have to do with Haskell?
  - Hint: the expression monad

Expression Tree (compile time)

```
data Exp = Const Int
         | Plus Exp Exp
         | Times Exp Exp
         | Arg1
         | Arg2
```

```
template<int c> struct Const {};

template<class E1, class E2>
struct Plus {};

template<class E1, class E2>
struct Times {};

struct Arg1 {};
struct Arg2 {};
```

State (runtime)

```
type Args = [Int]
```

```
struct Args
{
    Args(int i, int j) {
        _a[0] = i;
        _a[1] = j;
    }
    int operator[](int n) { return _a[n]; }
    int _a[2];
};
```

Type Constructor

```
Args -> a
```

```
newtype Prog a = PR (Args -> a)
```

- Prog created at compile time using a metafunction
- Action executed at runtime
- PR is really a concept with operator() as associated function

```
struct PR {
      // int operator()(Args args);
};
```

Monadic Metafunction

- Metafunctions may "return"
  - Values
  - Types
  - Other metafunctions
- New type of metafunction "returning" a function
  - (Further generalization: metafunction returning template function)

```
getArg :: Int -> Prog Int
getArg n = PR (λ args -> args !! n)
```

```
template<int n> // compile-time metafunction argument
struct GetArg : PR { // "returns" a Prog
    // runtime action
    int operator()(Args args) {
        return args[n];
    }
};
```

Bind: Construction

```
bind (PR prog) cont =
    PR (λ args ->
        let v = prog args
            (PR prog') = cont v
        in
            prog' args)
```

```
template<class P1, class P2> // compile-time type parameters
struct Bind : PR {           // “returns” a Prog
    // Bind object constructed at runtime
    Bind(P1 prog, std::function<P2(int)> cont)
        : _prog(prog), _cont(cont)
    {}
    …
    P1 _prog;
    std::function<P2(int)> _cont;
};
```

Bind: Action

```
bind (PR prog) cont =
    PR (λ args ->
        let v = prog args
            (PR prog') = cont v
        in
            prog' args)
```

```
template<class P1, class P2>
struct Bind : PR {
    …
    int operator()(Args args) {
        int v = _prog(args);
        P2 prog2 = _cont(v);
        return prog2(args);
    }
    …
};
```

Return

```
return :: a -> Prog a
return v = PR (λ args -> v)
```

```
struct Return : PR
{
    Return(int v) : _v(v) {}
    int operator()(Args args)
    {
        return _v;
    }
    int _v;
};
```

The Compile Metafunction

```
compile :: Exp -> Prog Int
```

- Metafunction Compile
  - Takes compile-time Exp
  - Returns a Prog
- Every specialization will define its own operator()

```
template<class Exp>
struct Compile {};
```

Simple Specializations

```
compile (Const c) = return c
```

```
template<int c>
struct Compile<Const<c>> : Return
{
    Compile() : Return(c) {}
};
```

```
compile Arg1 =
    getArg 0
```

```
template<>
struct Compile<Arg1> : GetArg<0> {};
```

Composite Specializations

```
template<class L, class R>
struct Compile<Plus<L, R>> {
  int operator()(Args args)
  {
    return Bind<…> (
      Compile<L>(),
      [](int left) {
        return Bind<…>(
          Compile<R>(),
          [left](int right) {
            return Return(left + right);
          }
        );
      }
    )(args);
  }
};
```

```
compile (Plus exL exR) =
  bind compile exL
       λ left ->
          bind compile exR
               λ right ->
                  return (left+right)
```

The Plus Node: Types

```
template<class L, class R>
struct Compile<Plus<L, R>> {
  int operator()(Args args)
  {
    return Bind<Compile<L>, Bind<Compile<R>, Return>> (
      Compile<L>(),
      [](int left) -> Bind<Compile<R>, Return> {
        return Bind<Compile<R>, Return>(
          Compile<R>(),
          [left](int right) -> Return {
            return Return(left + right);
          }
        );
      }
    )(args);
  }
};
```

```
compile (Plus exL exR) =
  bind compile exL
       λ left ->
          bind compile exR
               λ right ->
                   return (left+right)
```

Test: Arg1 * Arg2 + 13

```
testExp =
    let exp = Plus (Times Arg1 Arg2) (Const 13)
    compile exp
```

```
test =
    let args = [3, 4]
        act = testExp
    in
        run act args
```

```
void main () {
    Args args(3, 4);
    Compile<Plus<Times<Arg1, Arg2>, Const<13>>> act;
    int v = act(args);
    std::cout << v << std::endl;
}
```

2-Arg Lambda EDSL

```
int x = (arg1 + arg2 * arg2)(3, 4);
```

- Trick C++ into converting this expression into a tree
- arg1 and arg2: objects of types for which overloaded operators + and * exist
- Their return types correspond to expression trees
- Expression trees are (2-argument) function objects

Expression Wrapper

- For any expression E
  - Compile it to an action
  - Run the action and return the result

```
template<class E>
struct Lambda {
    int operator()(int x, int y) {
        Args args(x, y);
        Compile<E> prog;
        return prog(args);
    }
};
```

From Expression to Lambda

```
int x = (arg1 + arg2 * arg2)(3, 4);
```

- Special Lambda objects for Arg1 and Arg2 Expressions

```
const Lambda<Arg1> arg1;
const Lambda<Arg2> arg2;
```

- Overloaded operators
  - Generate types at compile time
  - Generate function objects at runtime

```
template<class E1, class E2>
Lambda<Plus<E1, E2>> operator+ (Lambda<E1> e1, Lambda<E2> e2)
{
    return Lambda<Plus<E1, E2>>();
}
```

Compile-Time vs. Runtime

```
(arg1   + arg2   *  arg2)     (3, 4)

// Compile time
Lambda<Arg1>  Lambda<Arg2> Lambda<Arg2> // original types
              Lambda<Times<Arg2, Arg2>> // type returned by *
Lambda<Plus<Arg1, Times<Arg2, Arg2>>    // type returned by +

// runtime, after template expansion
int Lambda<Plus<Arg1, Times<Arg2, Arg2>>::operator()(int x, int y)
{
     Args args(x, y);
     Compile<Plus<Arg1, Times<Arg2, Arg2>> prog;
     return prog(args);
}
```

Haskell Lambda EDSL

```
newtype Lambda = L Exp

toFun (L ex) =
  \ x y ->
     run (compile ex) [x, y]

instance Num Lambda where
    (L e1) + (L e2) = L (Plus e1 e2)
    (L e1) * (L e2) = L (Times e1 e2)
    fromInteger n = L (Const n)

test =
    let arg1 = L Arg1
        arg2 = L Arg2
    in
        (toFun (arg1 + 2 * arg2 * arg2)) 2 3
```

Conclusion

- Almost mechanical translation from Haskell state monad to C++ EDSL
- Haskell code easy to understand (after some initial pains) and test
- What seemed to be a bunch of template hacks gains strong theoretical foundations
  - Makes possible reasoning and proofs of correctness
- Reusable abstraction with unexplored potential
  - C++ state monad orthogonal to the construction of EDSL
  - Bind and Return used in defining monadic metafunction "Compile"
  - Monadic metafunction plugged into EDSL

Future Directions

- Explore metafunctions that “return” template functions
  - This is what Proto Transform does
- Better story on Const
  - Const values are part of state (even though they’re known at compile time)
- Factor out Transform and Domain of Proto
  - It’s really Transform that is a monad?

Bibliography

- Mike Vanier's blog (Haskell monads): http://mvanier.livejournal.com/3917.html
- Eric Niebler's blog (C++ Proto): http://cpp-next.com/archive/2010/08/expressive-c-introduction/
- My blog (extended treatment of current presentation): http://bartoszmilewski.wordpress.com/2011/01/09/monads-for-the-curious-programmer-part-1/
- Brian McNamara, Yannis Smaragdakis, Syntax sugar for FC++: lambda, infix, monads, and more.