Pattern Match on LLVM IR

1 Mechanism

The pattern matching header in the LLVM project appears to be user-friendly. Consider the following code:

```
llvm::PreservedAnalyses run(llvm::Function &F,
                            llvm::FunctionAnalysisManager &FAM) {
  using namespace llvm::PatternMatch;
  llvm::outs();
  for (auto &BB : F) {
    for (auto &I : BB) {
      llvm::Value *X, *Y;
      if (match(&I, m_Add(m_Value(X), m_Value(Y)))) {
        // I is an add instruction, and X and Y are its operands
        llvm::outs() << "Found␣an␣add␣instruction:␣" << I << "\n";
        llvm::outs() << "Operand␣1:␣" << *X << "\n";
        llvm::outs() << "Operand␣2:␣" << *Y << "\n";
      }
    }
  }
  return llvm::PreservedAnalyses::all();
}
```

In this code, we need to understand the roles of the `match`, `m_Add`, and `m_Value` functions.

1.1 `match`

`match` has two variants, our code uses this one:

```
template <typename Val, typename Pattern> bool match(Val *V, const Pattern &P)
{
  return const_cast<Pattern &>(P).match(V);
}
```

It just calls method `match` of a `Pattern` object.

1.2 `m_Add`

```
template <typename LHS, typename RHS>
inline BinaryOp_match<LHS, RHS, Instruction::Add> m_Add(const LHS &L,
                                                        const RHS &R) {
  return BinaryOp_match<LHS, RHS, Instruction::Add>(L, R);
}
```

`m_Add` relates symbol “Add” (in function name) to LLVM class `Instruction::Add`.

1.2.1 `BinaryOp_match`

```
template <typename LHS_t, typename RHS_t, unsigned Opcode,
          bool Commutable = false>
struct BinaryOp_match {
  LHS_t L;
  RHS_t R;
```

```
  // The evaluation order is always stable, regardless of Commutability.
  // The LHS is always matched first.
  BinaryOp_match(const LHS_t &LHS, const RHS_t &RHS) : L(LHS), R(RHS) {}

  template <typename OpTy> inline bool match(unsigned Opc, OpTy *V) {
    if (V->getValueID() == Value::InstructionVal + Opc) {
      auto *I = cast<BinaryOperator>(V);
      return (L.match(I->getOperand(0)) && R.match(I->getOperand(1))) ||
             (Commutable && L.match(I->getOperand(1)) &&
              R.match(I->getOperand(0)));
    }
    return false;
  }

  template <typename OpTy> bool match(OpTy *V) { return match(Opcode, V); }
};
```

This is what top level match function eventually calls.

It is important to note that every LLVM Value has a unique enum number that can be used to determine its concrete class. This enum is defined in the header file `llvm/include/llvm/IR/Value.def`. By using this enum, the `BinaryOp_match` function can directly determine whether the input value is an add instruction.

What are the types of L and R in the code? Based on their usage in the code, it can be inferred that they must be classes that also have a `match` method.

1.3 m_Value

```
inline bind_ty<Value> m_Value(Value *&V) { return V; }
```

Similar to `m_Add`, `m_Value` is a wrapper function that returns a structure with a match method.

1.3.1 bind_ty

```
template <typename Class> struct bind_ty {
  Class *&VR;

  bind_ty(Class *&V) : VR(V) {}

  template <typename ITy> bool match(ITy *V) {
    if (auto *CV = dyn_cast<Class>(V)) {
      VR = CV;
      return true;
    }
    return false;
  }
};
```

It maintains a reference to a pointer after construction and populates that pointer when the given class is matched.