

Pattern Match on LLVM IR

1 Mechanism

The pattern match header in LLVM project seems easy to use. Considering the 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();
}
```

We need to figure out what match, m_Add, and m_Value do.

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.

Note that every LLVM Value has a unique enum number that can be used to determine its concrete class, which is defined in `llvm/include/llvm/IR/Value.def`. Through it, `BinaryOp_match` can directly tell whether the input value is a add instruction.

What is the type of L and R in the code? According to their usage in the code, they must be some classes that have method `match` too.

1.3 m_Value

```

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

```

Like `m_Add`, `m_Value` is a wrapper function that returns a structure which has 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 keeps a reference to a pointer after construction. Populates that pointer when given `Class` is matched.