Approaches to pattern matching in compilers

Git revision #34fd6adb[1]
Modified at 19. August 2025 09:55

Written by alex_s168[2]

Introduction

Compilers often have to deal with pattern matching and rewriting (find-and-replace) inside the compiler IR (intermediate representation).

Common use cases for pattern matching in compilers:

- “peephole optimizations”: the most common kind of optimization in compilers. They find a short sequence of code and replace it with some other code. For example replacing `x & (1 << b)` with a bit test operation.
- finding a sequence of operations for complex optimization passes to operate on: advanced compilers have complex optimizations that can’t really be performed with simple IR operation replacements, and instead require complex logic. Patterns are used here to find operation sequences where those optimizations are applicable, and also to extract details inside that sequence.
- code generation: converting the IR to machine code / VM bytecode. A compiler needs to find operations (or sequences of operations) inside the IR, and “replace” them with machine code.

Simplest Approach

Currently, most compilers mostly do this inside their source code. For example, in MLIR, most (but not all) pattern matches are performed in C++ code.

The only advantage to this approach is that it doesn’t require a complex pattern matching system.

I only recommend doing this for small compiler toy projects.

[1] https://github.com/alex-s168/website/tree/34fd6adb3c89bef3f2d18b06b24533b52641bf4a
[2] https://alex.vxcc.dev

Disadvantages

Doing pattern matching this way has many disadvantages.

Some (but not all):

- debugging pattern match rules can be hard
- IR rewrites need to be tracked manually (for debugging)
- source locations and debug information also need to be tracked manually, which often isn't implemented very well.
- verbose and barely readable pattern matching code
- overall error-prone

I myself did pattern matching this way in my old compiler backend, and I speak from experience when I say that this approach **sucks** (in most cases).

Pattern Matching DSLs

A custom language for describing IR patterns and IR transformations (aka rewrites).

I will put this into the category of "structured pattern matching".

An example is Cranelift's ISLE DSL:

```
;; x ^ x == 0.
(rule (simplify (bxor (ty_int ty) x x))
      (subsume (iconst_u ty 0)))
```

Another example is tinygrad's pattern system:

```
(UPat(Ops.AND, src=(
   UPat.var("x"),
   UPat(Ops.SHL, src=(
     UPat.const(1),
     UPat.var("b")))),
 lambda x,b: UOp(Ops.BIT_TEST, src=(x, b)))
```

Fun fact: tinygrad actually decompiles the python code inside the second element of the pair, and runs multiple optimization passes on that.

This approach is used by many popular compilers such as LLVM, GCC, and Cranelift for peephole optimizations and code generation.

Advantages

- **debugging and tracking of rewrites, source locations, and debug information can be done properly**
- patterns themselves can be inspected and modified programmatically.
- they are easier to use and read than manual pattern matching in the source code.

There is however an even better alternative:

Pattern Matching Dialects

I will also put this method into the category of “structured pattern matching”.

The main example of this is MLIR, with the `pdl` and the `transform` dialects. Sadly few projects/people use these dialects, and instead do pattern matching in C++ code. Probably because the dialects aren’t documented very well.

What are compiler dialects?

Modern compilers, especially multi-level compilers, such as MLIR, have their operations grouped in “dialects”.

Each dialect either represents specific kinds of operations, like arithmetic operations, or a specific backend’s/frontend’s operations, such as the `llvm`, `emitc`, and the `spirv` dialects in MLIR.

Dialects commonly contain operations, data types, as well as optimization and dialect conversion passes.

Core Concept

The IR patterns and transformations are represented using the compiler’s IR. This is mostly done in a separate dialect, with dedicated operations for operating on IR.

Examples

MLIR's `pdl` dialect can be used to replace `arith.addi` with `my.add` like this:

```
pdl.pattern @replace_addi_with_my_add : benefit(1) {
  %arg0 = pdl.operand
  %arg1 = pdl.operand
  %op = pdl.operation "arith.addi"(%arg0, %arg1)

  pdl.rewrite %op {
    %new_op = pdl.operation "my.add"(%arg0, %arg1) -> (%op)
    pdl.replace %op with %new_op
  }
}
```

Advantages

- the pattern matching infrastructure can optimize it's own patterns: The compiler can operate on patterns and rewrite rules like they are normal operations. This removes the need for special infrastructure regarding pattern matching DSLs.
- the compiler could AOT compile patterns
- the compiler could optimize, analyze, and combine patterns to reduce compile time.
- IR (de-)serialization infrastructure in the compiler can also be used to exchange peephole optimizations.
- bragging rights: your compiler represents its patterns in it's own IR

Combining with a DSL

I recommend having a pattern matching / rewrite DSL, that transpiles to pattern matching / rewrite dialect operations.

The advantage of this over just having a rewrite dialect is that it makes patterns even more readable (and maintainable!)

E-Graphs

E-Graphs[3]

are magical datastructures that can be used to efficiently encode all possible transformations, and then select the best transformation.

An example implementation is egg[4]

Even though E-Graphs solve most problems, I still recommend using a pattern matching dialect, especially in multi-level compilers, to be more flexible, and have more future-proof pattern matching, or you decide that you want to match some complex patterns manually.

[3]https://en.wikipedia.org/wiki/E-graph

More Advantages of Structured Pattern Matching

Smart Pattern Matchers

Instead of brute-forcing all peephole optimizations (of which there can be a LOT in advanced compilers), the compiler can organize all the patterns to provide more efficient matching. I didn't yet investigate how to do this. If you have any ideas regarding this, please contact me.[5]

There are other ways to speed up the pattern matching and rewrite process using this too.

Reversible Transformations

I don't think that there currently is any compiler that does this. If you do know one, again, please contact me.

Optimizing compilers typically deal with code (mostly written by people) that is on a lower level than the compiler theoretically supports. For example, humans tend to write code like this for extracting a bit: `x & (1 << b)`, but compilers tend to have a high-level bit test operation (with exceptions). A reason for having higher-level primitives is that it allows the compiler to do more high-level optimizations, but also some target architectures have a bit test operation, that is more optimal.

This is not just the case for "low-level" things like bit tests, but also high level concepts, like a reduction over an array, or even the implementation of a whole algorithm. For example LLVM, since recently, can detect implementations of CRC.[6]

LLVM actually doesn't have many dedicated operations like a bit-test operation, and instead canonicalizes all bit-test patterns to `x & (1 << b) != 0`, and matches for that in compiler passes that expect bit test operations.

Now let's go back to the `x & (1 << b)` (bit test) example. Optimizing compilers should be able to detect that, and other bit test patterns (like `x & (1 << b) > 0`), and then replace those with a bit-test operation. But they also have to be able to convert bit-test operations back to their implementation for compilation targets that don't have a bit-test instruction. Currently, compiler backends do this by having separate patterns for converting bit-test to it's dedicated operation, and back.

[4] https://egraphs-good.github.io/
[5] https://alex.vxcc.dev
[6] https://en.wikipedia.org/wiki/Cyclic_redundancy_check

A better solution is to associate a set of implementations with the bit test operation, and make the compiler **automatically reverse** those to generate the best implementation (in the instruction selector for example).

Implementing pattern/transformation reversion can be challenging however, but it provides many benefits, and all “big” compilers should definitely do this, in my opinion.

Runtime Library

Compilers typically come with a runtime library that implement more complex operations that aren’t supported by most processors or architectures.

The implementation of those functions should also use that pattern matching dialect. This allows your backend to detect code written by users with a similar implementation as in the runtime library, giving you some additional optimizations for free.

I don’t think any compiler currently does this either.

Problems with Pattern Matching

The main problem is ordering the patterns.

As an example, consider these three patterns:

```
;; A
(add x:Const y) => (add y x)

;; B
(sub (add x y:Const) z:Const) => (lea x y (const_neg z))

;; C
(add x 1) => (inc x)
```

Now what should the compiler do when it sees this:

```
(sub (add 5 1) 2)
```

All three patterns would match:

```
;; apply A
(sub (add 5 1) 2) => (sub (add 1 5) 2)
;; only B applies now
(sub (add 1 5) 2) => (lea 1 5 (const_neg 2))
;; nothing applies anymore

;; alternatively apply B
(sub (add 5 1) 2) => (lea 5 1 (const_neg 2))
;; nothing applies anymore

;; atlernatively apply C
(sub (add 5 1) 2) => (sub (inc 5) 2)
;; nothing applies anymore
```

Now which of those transformations should be performed?

This is not as easy to solve as it seems, especially in the context of instruction selection (specifically scheduling), where the performance on processors depends on a sequence of instructions, instead of just a single instruction.

Superscalar CPUs

Modern processor architecture features like superscalar execution make this even more complicated.

As a simple, **unrealistic** example, let’s imagine a CPU (core) that has one bit operations execution unit, and two ALU execution units / ports.
This means that the CPU can execute two instructions in the ALU unit and one instruction in the bit ops unit at the same time.

One might think that always optimizing `a & (1 << b)` to a bit test operation is good for performance. But in this example, that is not the case.

If we have a function that does a lot of bitwise operations next to each other, and the compiler replaces all bit tests with bit test operations, suddenly all operations depend on the bit ops unit, which means that instead of executing 3 instructions at a time (ignoring pipelining), the CPU can only execute one instruction at a time.

This shows that we won’t know if an optimization is actually good, until we are at a late point in the compilation process where we can simulate the CPU’s instruction scheduling.

This does not only apply to instruction selection, but also to more higher-level optimizations, such as loop and control flow related optimizations.

Conclusion

One can see how pattern matching dialects are the best option to approach pattern matching.

Someone wanted me to insert a takeaway here, but I won’t.

PS: I’ll hunt down everyone who still decides to do pattern matching in their compiler source after reading this article.