Love in the Time of Compiler Construction

Andrea Shepard <andrea@persephoneslair.org>

2017-05-28

Hammer

Parser combinator library in C

Many of you will already be familiar, but...

- Parse input strings according to a tree of parser combinators, created using library API
- For example, this recognizes {'ab', 'ac'}:

```
HParser *p =
  h_sequence(
    h_ch('a'),
    h_choice(
      h_ch('b'),
      h_ch('c'),
      NULL),
  NULL);
```

Hammer backends

Such as...

- Packrat parser (default)
- GLR and LALR backends
- LL(k) backend

Problem

The packrat backend is slow. Parsing the preceding example involves calling `h_do_parse()`, pulling a function pointer from the `h_sequence` vtable, calling it, reading out a list of parsers in the `HParser` structure, calling `h_do_parse()` on them in turn, and so on.

Toward parser JITs

We actually know a lot more about what we're expecting to see next when parsing some concrete `HParser` than when writing a generic parse function for any `HParser` of that type, though. A custom parse function for that specific parser could cut out a lot of the pointer-chasing, and open the path to many more potential optimizations.

If only there were some way of constructing such functions on the fly...

Introducing the LLVM backend

The new LLVM backend I've been writing does exactly this - constructs and compiles functions to recognize HParsers on the fly, and then runs them to parse input. It thus reduces the amount of indirection and number of memory accesses needed for many operations, and sometimes gains a significant edge from knowing the parser's arguments at compile time.

A simple example, pt. 1

Consider the parser `h_not_in({'a', 'b', 'c'}, 3)`. This leads to LLVM IR like this (some preamble/postamble omitted):

```
l1: %cmp1 = icmp ule i8 %2, 99
    br i1 %cmp1, label %l2, label %l3

l2: %cmp2 = icmp ule i8 %2, 96
    br i1 %cmp2, label %l4, label %l5

l3: br label %success

l4: br label %success

l5: br label %fail
```

A simple example, pt. 2

Compiling this IR, in turn, generates this (again, some preamble/ postamble is omitted):

```
mov     %rax,%rbx
movzbl  %bl,%eax
cmp     $0x64,%eax
jae     0x7ffff7fed033 <success>
movzbl  %bl,%ecx
xor     %eax,%eax
cmp     $0x61,%ecx
jae     0x7ffff7fed076 <fail>
```

Much nicer than index calculations and loading something from a bitmap!

Generating optimal charsets

The `h_not_in()` parser is implemented as a charset, which in the non-LLVM packrat backend is just done by indexing into a 32-byte bitmap. For sufficiently rough character sets this may be the most efficient method, but for many cases in practice, comparisons as in the preceding are better.

To generate charsets, we express them as a tree of operations such as splitting at a particular point (lt/gt comparison instruction), complementation (a free swap of success and failure branch targets), sequential equality comparisons (efficient for very sparse sets) or a bitmap. The optimal tree is searched for once during `h_llvm_compile()`.

Charset optimization

You can see here how we build up the membership tests out of elementary actions (the full dump also shows a bitmap and range brackets at each node, omitted here for space reasons):

```
CHARSET_ACTION_SPLIT
|   idx_start = 0, split_point = 99, idx_end = 255
|   cost = 2, depth = 0
+--CHARSET_ACTION_SPLIT
|   |   idx_start = 0, split_point = 96, idx_end =
|   |   cost = 1, depth = 1
|   +--CHARSET_ACTION_ACCEPT
|   |       idx_start = 0, idx_end = 96
|   |       cost = 0, depth = 2
|   +--CHARSET_ACTION_COMPLEMENT
...
```

Other leaf nodes

Most of the other leaf `HParsers` are straightforward; tokens are a little complex - very short ones are sequential equality tests, longer ones are comparison loops against a string constant stored in memory.

To minimize memory and cache use, and since some, such as charsets, can be complex to generate, we will memoize these parsers and recognize when the same one occurs more than once.

Higher-order nodes (not yet implemented)

Higher-order parsers like `h_sequence()` are implemented with indirect calls to their children in the pre-LLVM backend. The requirement to correctly implement this recursion is to keep a stack of (parser, position within parser) pairs; an explicit stack was considered, but return addresses are an almost-as-concise representation as any other likely to be feasible, and LLVM is pleasantly flexible about calling conventions, so the overhead of making each memoized `HParser` its own LLVM function seems acceptable.

Future Directions?

- We could JIT functions based on pattern-matching parser tree fragments - e.g. `h_choice(h_ch('a'), h_ch('b'), ..., NULL)` replaced with a single optimal charset recognizer.
- We could also recognize parsers stacked with `h_action()`, and avoid allocating and constructing intermediate parse results in cases when the action rather than a parse tree is the intended result and we could just invoke it directly from the JITted parser.

Remaining possible optimizations

- The explicit calls out of LLVM to `h_read_bits()` one char at a time or to `make_result()` may not be the most efficient thing to do; this should be investigated further and profiled.
- It probably isn't useful to try to use this approach for any of the other backends (e.g. LALR, LL(k)), since they're table-driven and have most of their complexity at table generation time, with much tighter and simpler parse-time code.
- We should try to empirically determine optimal values of the cost parameters in some of the leaf node parser codegen. E.g. bitmap lookups are currently somewhat arbitrarily treated as equal to 5 compare-and-branch ops.

URLs

Main Hammer repository

`https://github.com/upstandinghackers/hammer`

LLVM backend WIP repository

`https://github.com/puellavulnerata/hammer/tree/llvm-backend`