Seaching through a Trace with Patterns

September 15, 2016

The principal method by which a bug will be located in our system is by the (semi-)automatic edition of the program trace. This can happen as the program is evaluated, or by interactive searching afterward. But we need a way to express the searches.

Text-based approaches (e.g regular expressions) are not much use. Fine for searching for a function name, but no good for tree-based data like functional programs. We want to be able to say, for example:

1. "Find any function application in the trace taking an empty tree as input."
2. "Show me all calls to function `f`"
3. "Show me any time a list begins with a negative number"

Patterns for 1, 2, 3 might be "`_ Lf`", "f _", and "[-_; ...]".

1 Patterns

For example...

`_`	wildcard
`()`	parentheses
`[p; p]`	lists (similarly for records, tuples etc)
`...`	to indicate the tail of a list
`text`	literal text
`remove*`	wildcards as part of text

- We use the same lexical conventions as OCaml, so we can reuse the OCaml lexer. The parser should not be too difficult, just must have a subset of the associativities / precedences of the OCaml one.
- The parser should accept *any* string, just counts as text to match.

2 TODO

1. Define a small example pattern language
2. Choose a subset of `Tinyocaml.t`
3. Find Example program, trace, and patterns
4. Write the pattern parser
5. Write the matcher, which sees if a pattern matches a line of the trace, by matching the tree of the pattern against the tree of that line, rather than text against text. Need a way to indicate the matched part, say by underlining.

3 Difficult examples

How do we match pattern `1 + 2 + 3`. Does the tree `1 + (2 + 3)` match? Could we just remove all parentheses? Too many false matches?

4 The text approach

Advantages:

- Probably simpler to implement
- Regular expressions are familiar to the user - no need for our own pattern language, or at least we could base it on something.

Limitations:

- No clean support for dealing with parentheses, associativity, precedence.
- We want wildcards which correspond to nodes of a tree, not text items!

5 The tree approach

Advantages:

- Mirrors the structure of the language - feels a bit like pattern matching – OCaml users already familiar – in fact, will be a strict superset of OCaml pattern matching.

Limitations:

- Dealing with precedence and associativity means the tree shapes don't necessarily match.

6 References

1. "Pattern Matching in Trees" (Hoffmann, O'Donnell, 1982). This early paper considers the pattern matching of trees in terms of interpreters for languages whose evaluation is defined by term rewriting rules. This is relevant to our wider work, despite finding it in this context.

2. "Pattern matching syntax trees" `http://checkmyworking.com/2014/06/pattern-matching-syntax-trees/` This is interesting work in the context of matching up childrens' answers to computer algebra questions with the expected answers. It uses a special kind of pattern matching which knows about nesting of parentheses, and commutivity (for example, $x + 1$ and $1 + x$ are considered equal).

3. There may be good research in this area for XML processing systems – it's all trees. XPath. See `https://theantlrguy.atlassian.net/wiki/display/~admin/2013/09/01/Matching+parse+tree+patterns,+paths`

4. *Darren C. Atkinson and William G. Griswold, "Effective pattern matching of source code using abstract syntax patterns* look this one up.

5. Darren C. Atkinson and William Griswold and Collin McCurdy *Fast, Flexible Syntactic Pattern Matching and Processing*.

6. A toolset for FORTRAN programs – contains pattern matching over fortran statements `https://www.irisa.fr/caps/projects/TSF/demo/doc/draft.html`

7. Roger F. Crew, *ASTLOG: A Language for Examining Abstract Syntax Trees*.

7 Notes on First Implementation of on-line search

An initial implementation, just with regular expressions for now, has been written to explore what kind of tools are needed for searching program traces.

1. The option `-search <searchterm>`, given a regular expression, shows only those lines matching the search term.
2. The option `-n <n>` prints only up to n matches.
3. The option `-invert-search` which shows those lines not matching the search term.
4. The option `-after <searchterm>` which further restricts the output to lines occurring after and including the first to match the given searchterm, which must match something which matches the `-search` searchterm.
5. The option `-until <searchterm>` which further restricts the output to lines occurring after and including the first to match the given searchterm, which must match something which matches the `-search` searchterm.
6. Options `-after-any <searchterm>` and `-until-any <searchterm>` which are like `-after` and `-until` but which match on any line, not just those matching the main `-search` search term. Alternative to not in addition to `-until/-after`.
7. Options `-invert-after` and `-invert-until` which invert the meanings of `-after`, `-after-any`, `-until` and `-until-any` just like `-invert-search`.
8. Option `-repeat` to not stop printing after the `-after` or `-after-any` condition. Instead, wait for the next `-until` or `-until-any`, and keep going.
9. The option `-upto <n>` which prints n lines (not n search results) before each matching one, for context.
10. An option `-stop` to stop computation after search result(s) printed if `-n` has been supplied. Defaults is to carry on to the end.
11. An option `-highlight` to highlight, in reverse video, the part of the step which matched the search expression.

8 Offline vs Online search: Pros and cons

Offline search (we make the log of the whole computation and then search it):

1. What we do doesn't make side effects or computation happen. It's cleaner.
2. Producing the original trace is much slower
3. But searching it repeatedly with different search terms is much faster (almost instant, except for very large traces).
4. Trace may be huge (gigabytes)

Online search (we provide the search parameters, and the search is run, printing only matching steps, as the computation proceeds):

1. Only what needs to be printed is printed (printing dominates time, so this is good). (n.b. this depends on whether we match on trees or lexmes, or printed output. For printed output no difference)
2. Re-running the search may be done on cached log, so should be as quick.
3. Fits in well with other interactive features.

Conclusion: **need both!**

9 Our own search syntax

1. The old, regular expression method can be used by adding `-regex` to the command line.
2. Our new syntax is converted internally to a regular expression, so the search mechanism is the same for both search syntaxes.
3. We follow the same lexical conventions as OCaml, by using the Genlex module, although it's not quite clear how to get it exhaustive – the documentation is not up to scratch.
4. It is worth investigating if there is a general method to get all the lexemes from a ocamllex / menhir parser without it having to parse. Who can we ask about this? The mailing list?

Conversion to Regular expression:

1. Lex with genlex.
2. Add allowable whitespace between each and every genlex lexeme
3. Replace any use of the underscore with the regexp `.*` to allow anything.
4. Quote all the parts of the regular expression with `Str.quote`. Put together.
5. The option `-no-parens` removes all uses of `( ) begin end` from the lexeme list.