MATCHING MATHEMATICA PATTERNS

DANIEL SCHULTZ

Abstract. A process to construct machines recognizing mathematica patterns is discussed. This makes the pattern matching process completely transparent, and we attempt to either match some of the behaviour exhibited by WRI's implementation or improve on it.

1. Introduction

1.1. **orderless.** When matching `f[a,b,c,d,e]` against `f[x__,y__]` with an orderless `f`, one will find that WRI's matcher does not even consider the possibility `x=b,a`, `y=c,d,e`. This is apparently because `b,a` is itself out of order and the pattern matcher does not change the order in sequences. While not completely ideal, we reproduce this behaviour simply because it cuts down on the number of possiblity to be tried while still retaining some useful functionality: 2^5 is much smaller than 5!.

1.2. **flat.** When matching `f[a,b,a,b,c,d,e]` against `f[(x_)..,y__,z__]`, we allow `x` to match `f[a,b]`.

2. Architecture of the compiler

Besides the pattern that is to be constructed, the function for converting a pattern to a machine takes the following arguments:

- A stack of symbols called `vars` which consists of the variables currently being recorded (from `Pattern`). If the the symbol x is uninitialized, the instruction $\mathit{start}(x)$ will at run time cause the instruction $\mathit{capture}(x)$ to append the local variable cur to x. The instruction $\mathit{stop}(x)$ will stop such appends and put the symbol x in the initialized state. If the symbol x is initialized, the instruction $\mathit{start}(x)$ will at run time cause the instruction $\mathit{capture}(x)$ to start comparing with the currently stored sequence for x. Then, the instruction $\mathit{stop}(x)$ will check that the number of expressions checked by the captures matches the original number of values captured in the initial $\mathit{start}(x)$ - $\mathit{stop}(x)$ pair.
- A stack of expressions called `tests`. The instruction $\mathit{test}(f)$ will apply f to the value of the local variable cur and fail if the return is not exactly `True`. This is used to implement `PaternTest`.
- A boole `ishead` indicating if we are in the head of a pattern.
- A boole `isFlat` indicating if we are in the arguments of a flat head.
- A boole `isOrderless` indicating if we are in the arguments of an orderless head.

We have the following runtime local variables of the pattern matcher machine:

- An integer i which starts at $i = 0$. This is the index of the current child we are looking at ($i = 0$ correspondes to the head).

- An expression *cur* which is the most recent child looked at in the current subexpression.

The instructions *down* (resp. *up*) are responsible for moving *down* from parent to child (resp. up from child to parent) and constructing a new stack frame for i and *cur* (resp. destroying it) These are all operation done at runtime when the pattern matcher is trying to match, and not done in the process of compiling a pattern. The most important routine with respect to the variables i and *cur* is the *next* routine, which is responsible for choosing a new argument from the arguments of the expressions passed to the matcher.

3. Basic routines

3.1. **bootstrapping.** To compile the pattern p at the top level we use

$$\begin{tikzcd}
\text{start} \arrow[d] \\
\text{set } i = 0 \arrow[d] \\
\boxed{p} \arrow[d] \\
\text{fail if } i \neq 1 \arrow[d] \\
\text{pass}
\end{tikzcd}$$

The box around p means that it should be compiled recursively according to the rules given below and its *start* and *end* nodes should be spliced in. p should be compiled with `vars` and `tests` both empty and `ishead=True`. This bootstrapping step is a little awkward because it means that the final pattern matcher must be passed $e[]$ when matching e. However, this is the simplest way to ensure that `BlankSequence` and `BlankNullSequence` do not get erroneously matched at the top level.

3.2. **Routine for patternless expressions.** If p is a pattern free from expressions related to pattern matching, it may be compiled as

$$\begin{tikzcd}
\text{start} \arrow[d] \\
\text{next} \arrow[d] \\
\text{fail if } !SameQ[p, cur] \arrow[d] \\
\forall_{f \in tests} \text{ fail if } eval(f[cur]) \neq True \arrow[d] \\
\forall_{x \in vars} capture(x) \arrow[d] \\
\text{end}
\end{tikzcd}$$

The routine *next* depends on the context supplied by the context in 3.3 and is described in Section 3.4.

3.3. **Routine for nodes without pattern-related heads.** The pattern $p_0[p_1, \ldots, p_n]$ where p_0 has no meaning related to patterns (basically it's not in Section 4) may be compiled as

$$\begin{tikzcd}
\text{start} \arrow[d] \\
\text{next} \arrow[d] \\
\text{down} \arrow[d] \\
\text{set } i = 0 \arrow[d] \\
\boxed{p_0} \arrow[d] \\
\text{fail if } i \neq 1 \arrow[d] \\
\text{initialize an array of arguments to be matched} \arrow[d] \\
\boxed{p_1} \arrow[d] \\
\text{set } i \text{ to index of first available argument} \arrow[d] \\
\boxed{p_2} \arrow[d] \\
\text{set } i \text{ to index of first available argument} \arrow[d] \\
\boxed{p_n} \arrow[d] \\
\text{fail if any arguments are still available} \arrow[d] \\
\text{end}
\end{tikzcd}$$

All of the p_i should be compiled with empty `vars` and empty `tests`. p_0 should be compiled with `ishead=True` and $p_1, \ldots, p_n$ should be compiled with `ishead=False` and `isFlat`, `isOrderless` set according to p_0. The instruction "initialize an array of arguments to be matched" needs to mark each of the children of the expression e to be matched as available. When the instruction *next* in Section 3.4 consumes a child via e_{i++} this child of index i needs be marked as unavailable. When there are no more available children, the instruction "set i to index of first available argument" should set i to an integer greater than the length of the expression to be matched. This will ensure that all calls to $i++$ or e_{i++} fail.

3.4. **Routine** ***next*** **for heads and childs.** Incrementing i when it is already past the length of the expression e to be matched should be considered a fail. In all of these routines, the increment on i $(i++)$ will be a simple "add one" when the corresponding head is not orderless. However, when the head is orderless, it should move i to the next available child. Similarly, e_{i++} should mark the ith child as unavailable before incrementing i. The following one routine is for head mode (`ishead=True`). The values of `isFlat` and `isOrderless` do not matter in this case. Routine *next* with `ishead=True`:

$$\begin{tikzcd}
\text{start} \arrow[d] \\
\text{fail if } i \neq 0 \arrow[d] \\
cur = e_{i++} \arrow[d] \\
\text{end}
\end{tikzcd}$$

The following are all for child mode (`ishead=False`). Routine *next* with `isFlat=False` and `Orderless=False`:

$$\begin{tikzcd}
\text{start} \arrow[d] \\
cur = e_{i++} \arrow[d] \\
\text{end}
\end{tikzcd}$$

Routine *next* with `isFlat=False` and `isOrderless=True`:

$$\begin{tikzcd}
\text{start} \arrow[d] \arrow[r, bend right] & i++ \arrow[l, bend right] \\
cur = e_{i++} \arrow[d] & \\
\text{end} &
\end{tikzcd}$$

Routine *next* with `isFlat=True` (head f) and `isOrderless=False`:

$$\begin{tikzcd}
\text{start} \arrow[d] & & \\
cur = e_{i++} \arrow[dd] \arrow[dr] & & \\
& cur = f[cur] \arrow[r] \arrow[dl] & AppendTo[cur, e_{i++}] \arrow[loop above] \arrow[dll] \\
\text{end} & &
\end{tikzcd}$$

Routine *next* with `isFlat=True` (head f) and `isOrderless=True`:

$$\begin{tikzcd}
\text{start} \arrow[d] \arrow[r, bend right] & i++ \arrow[l, bend right] & & & \\
cur = e_{i++} \arrow[dd] \arrow[dr] & & & i++ \arrow[d, bend left] & \\
& cur = f[cur] \arrow[r] \arrow[dl] & {} & \cdot \arrow[u, bend left] \arrow[r] \arrow[dlll] & AppendTo[cur, e_{i++}] \arrow[l, bend left] \\
\text{end} & & & &
\end{tikzcd}$$

Note that WRI's implementation only gives *next* such a special behaviour with respect to `Flat` when it is in a `Blank[]` at the top level of a child (possibliy only having a `Pattern` object as a parent).

4. PATTERN HEADS

4.1. **`Blank`.** $Blank[]$ may be compiled as

$$\begin{tikzcd}
\text{start} \arrow[d] \\
\text{next} \arrow[d] \\
\forall_{f\in tests} \text{ fail if } eval(f[cur]) \neq True \arrow[d] \\
\forall_{x\in vars} capture(x) \arrow[d] \\
\text{end}
\end{tikzcd}$$

$Blank[h]$ may be compiled by inserting an instruction testing the head of *cur* after the *next* instruction.

4.2. **`BlankSequence`.** $BlankSequence[]$ may be compiled as

$$\begin{tikzcd}
\text{start} \arrow[d] \\
\text{next} \arrow[d] \\
\forall_{f\in tests} \text{ fail if } eval(f[cur]) \neq True \arrow[d] \arrow[u, bend left] \\
\forall_{x\in vars} capture(x) \arrow[d] \\
\text{end}
\end{tikzcd}$$

The *next* routine here should ignore the `isFlat` flag, i.e. set it to false.

4.3. **`BlankNullSequence`.** $BlankNullSequence[]$ may be compiled as

$$\begin{tikzcd}
\text{start} \arrow[d] \arrow[dddd, bend left] \\
\text{next} \arrow[d] \\
\forall_{f\in tests} \text{ fail if } eval(f[cur]) \neq True \arrow[d] \arrow[u, bend left] \\
\forall_{x\in vars} capture(x) \arrow[d] \\
\text{end}
\end{tikzcd}$$

The *next* routine here should ignore the `isFlat` flag, i.e. set it to false.

4.4. `Pattern.` $Pattern[s, p]$ for a symbol s may be compiled as

$$\begin{tikzcd}
\text{start} \arrow[d] \\
start(s) \arrow[d] \\
\boxed{p} \arrow[d] \\
stop(s) \arrow[d] \\
\text{end}
\end{tikzcd}$$

Before p is compiled, s should be pushed onto `vars`. Then, p should be compiled with parameters matching those of the calling environment. After s is compiled, `vars` should be popped.

4.5. `PatternTest.` $PatternTest[p, f]$ may be compiled as

$$\begin{tikzcd}
\text{start} \arrow[d] \\
\boxed{p} \arrow[d] \\
\text{end}
\end{tikzcd}$$

Before p is compiled, f should be pushed onto `tests`. Then, p should be compiled with parameters matching those of the calling environment. After p is compiled, `tests` should be popped.

4.6. `PatternSequence.` $PatternSequence[p_1, \ldots, p_n]$ may be compiled as

$$\begin{tikzcd}
\text{start} \arrow[d] \\
\boxed{p_1} \arrow[d] \\
\boxed{p_2} \arrow[d] \\
\boxed{p_n} \arrow[d] \\
\text{end}
\end{tikzcd}$$

All p_i should be compiled with parameters matching those of the calling environment.

4.7. `Alternatives.` $Alternatives[p_1, \ldots, p_n]$ may be compiled as

$$\begin{tikzcd}
 & \text{start} \arrow[dl] \arrow[d] \arrow[dr] & \\
\boxed{p_1} \arrow[dr] & \boxed{p_2} \arrow[d] & \boxed{p_n} \arrow[dl] \\
 & \text{end} &
\end{tikzcd}$$

All p_i should be compiled with parameters matching those of the calling environment.

4.8. `Repeated.` $Repeated[p]$ may be compiled as

$$\begin{tikzcd} \text{start} \arrow[d] \\ \boxed{p} \arrow[d] \arrow[loop left] \\ \text{end} \end{tikzcd}$$

p should be compiled with parameters matching those of the calling environment.

4.9. `RepeatedNull.` $RepeatedNull[p]$ may be compiled as

$$\begin{tikzcd} \text{start} \arrow[d] \arrow[dd, bend left] \\ \boxed{p} \arrow[d] \arrow[loop left] \\ \text{end} \end{tikzcd}$$

p should be compiled with parameters matching those of the calling environment.

4.10. `Condition.` $Condition[p, f]$ may be compiled as

$$\begin{tikzcd} \text{start} \arrow[d] \\ \boxed{p} \arrow[d] \\ \text{fail if } eval(f) \neq True \arrow[d] \\ \text{end} \end{tikzcd}$$

p should be compiled with parameters matching those of the calling environment.

4.11. `Optional.` $Optional[Pattern[s, p], v]$ for a symbol s may be compiled as

$$\begin{tikzcd} \text{start} \arrow[d] & \\ start(s) \arrow[d] \arrow[dr] & \\ \boxed{p} \arrow[d] & capture(s, v) \arrow[dl] \\ stop(s) \arrow[d] & \\ \text{end} & \end{tikzcd}$$

Of course `vars` should be pushed with s and popped around the compilation of p. The special two argument form of *capture* captures into s not the value of *cur* but the constant value v.

TODO: see if this matches WRI implementation, in particular, do the `tests` from *PatternTest* affect the capture of v?

5. implementation issues

It seems very possible to match mathematica patterns by machine. Many nodes in the resulting machine have more than one outgoing path: the pattern is said to match if there is some path from the *start* node (of the bootstrapping step) to the *pass* node without hitting a *fail*. In practice this need to be implemented via backtracking by saving the state every time a decision is made.

5.1. **substitution of captures.** There is an issue of how the captured variables should be substituted into expressions when they require evaluation (for example, in `Condition`). It seems reasonable to leave an uninitialized variable as itself and splice in initialized variables. It seems that WRI's implementation sometimes treats uninitialized variables as `Sequence[]`.

5.2. **side effects.** If the main evaluations implied by `Condition` or `PatternTest` involve computation with side effects, we should consider the behaviour of the pattern matcher to be undefined, especially if these side effects include changing the attributes of the heads involved in matching!

5.3. **Repeated[BlankNullSequence[]].** The savvy reader will have noticed that `x___..` results in a machine with an infinite loop. Such infinite loops only result from the back edges of `Repeated` and `RepeatedNull`, and this problem can be resolved by failing if the value of i is the same as it was the last time the back edge was taken.