A Theorem Prover for Subregular Systems

The Language Toolkit and its Interpreter, Plebby

Dakotah Lambert

FLOPS 2024
15–17 May 2024

What is it?

What it is NOT:

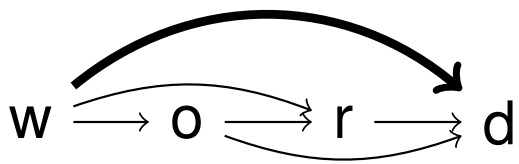
- A regular expression engine:
  there are better systems for that
- A good general-purpose DFA library:
  automata are the internal representations, not the focus

Finite model theory

$$w \longrightarrow o \longrightarrow r \longrightarrow d$$

$$\langle o\ r\ d \rangle$$

Finite model theory



$\langle w, d \rangle$

Finite model theory

$$\langle x_1 \oplus_1 x_2 \oplus_2 \ldots \oplus_n x_n \rangle$$

$$(\exists p_1 \ldots p_n)[x_i(p_i) \wedge R_{\oplus_i}(x_i, x_{i+1})],$$

where $R_{\langle \text{whitespace} \rangle}$ is successor and $R_{\langle \text{comma} \rangle}$ is precedence

$\rtimes$: first character is start of word

$\ltimes$: last character is end of word

Finite model theory

$\rtimes \ltimes \langle \text{w o r d} \rangle$ asserts that the word is ‘word’

$\rtimes \langle \text{p r e} \rangle$ asserts that words begin with ‘pre’

$\langle x, y \rangle$ asserts that ‘x’ and ‘y’ appear, in that order

Finite model theory

Build up systems of constraints with Boolean operations:

- $\bigwedge\{e_1, \ldots, e_n\}$: intersection / AND
- $\bigvee\{e_1, \ldots, e_n\}$: union / OR
- $\neg e_1$: complement / NOT

Plenty of other operations:
Kleene star, reversal, concatenation, shuffle product, infiltration product, quotients, ...

Finite model theory

‘x does not immediately follow y’

$\neg\langle \mathrm{y}\ \mathrm{x}\rangle$

‘x does not follow y at any distance’

$\neg\langle \mathrm{y}, \mathrm{x}\rangle$

Identical languages with alphabet $A = \{\mathrm{x}, \mathrm{y}\}$.
but not logically equivalent: yzx satisfies only one.

Finite model theory

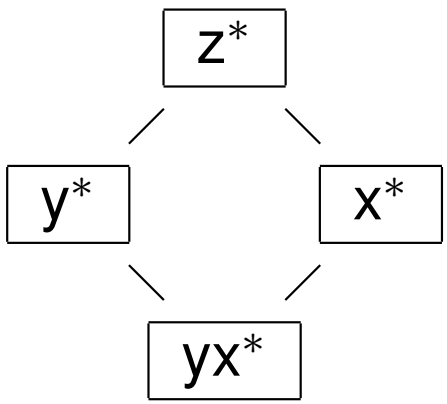
```
plebby, version 1.2:
https://hackage.haskell.org/package/language-toolkit
:help for help
> =x /x =y /y
> :equal ¬⟨y x⟩ ¬⟨y, x⟩
True
> :cequal ¬⟨y x⟩ ¬⟨y, x⟩
False
```

(ASCII syntax is also available)

Algebra

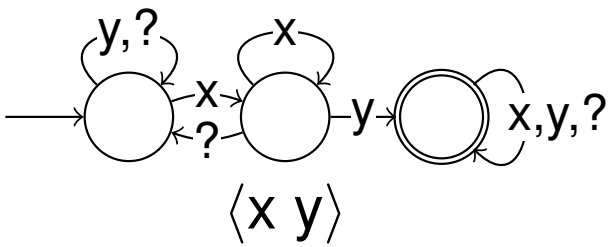
Most classification based on syntactic semigroup properties

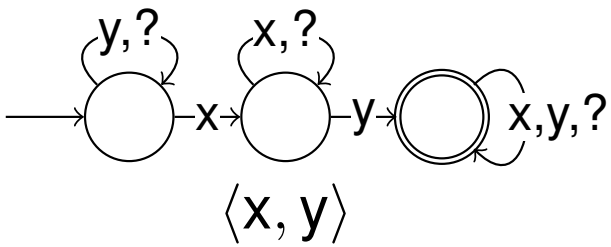
```
> =x /x =y /y =z /z
> ⋀{⟨x⟩,⟨y⟩}
> :isVarietyS [ab=ba] it
True
> :eggbox it
```



How is it implemented?

Finite automata





Finite automata

```
data FSA n e
    = FSA
      { sigma           :: Set e
      , transitions     :: Set (Transition n e)
      , initials        :: Set (State n)
      , finals          :: Set (State n)
      , isDeterministic :: Bool
      }

newtype State n = State {nodeLabel :: n}

data Symbol e = Epsilon
              | Symbol e

data Transition n e
    = Transition
      { edgeLabel   :: Symbol e
      , source      :: State n
      , destination :: State n
      }
```

- Easy to use, represents everything
- `isDeterministic` flag speeds up some operations
- Redundancy is a larger bug surface
- Generality means more cases to handle
- Deterministic-only interface coming soon!

Finite semigroup

Semigroups are already deterministic-only:

```
data GeneratedAction
   = GeneratedAction { bases      :: [[Int]]
                     , completion :: [[Int]]
                     }
```

Upcoming determinstic interface:

```
data DFA n e = DFA {
     overlays    :: Map e (Seq Int) -- corresponds to ‘bases’
   , finalStates :: IntSet
   , stateNames  :: IntMap n
   }
```

The interpreter

`Applicative` and `Alternative` make parsing simple:
easy to read and easy to extend

```
parseNAryExpr
   = makeLifter
     [ (["⋀", "⋂", "∧", "∩", "/\\"], Conjunction)
     , (["⋁", "⋃", "∨", "∪", "\\/"], Disjunction)
     , (["\\\\"],                    QuotientL)
     , (["//"],                      QuotientR)
     , ([".●.", ".@."],              StrictOrder)
     , (["●●", "@@"],                Domination)
     , (["●", "@"],                  Concatenation)
     , (["ш", "|_|_|"],              Shuffle)
     , (["⇑", ".^."],                Infiltration)
     ] <*> (parseEmpty <|> parseDelimited ['(', '{'] (parseSeparated "," parseExpr))
```

Interpreter

With combinatory parsers,
error messages can be tricky:

```
> =x /x =y /y
> :display ⟨a⟩ # syntactically correct
undefined variable "a"
> :display ⟨a # invalid syntax, failed parse
not a variable:  TSymbol '⟨'
sought '[' but instead found '⟨'
sought '|' but instead found '⟨'
not a variable:  TSymbol '⟨'
```

What next?

Future work

- Performance
  - Started with splitting off `finite-semigroups` package
  - Moving to simpler, determinism-enforcing types
  - New internals won't change interpreter's interface
- Different parser style for better errors?
- Process analysis (transducers)
- Feature bundles over symbols?
- Creating a thorough test suite

Thank you!

`https://hackage.haskell.org/package/language-toolkit`