Visibly Powerful Parsing

Philippa Cowderoy

flippa@flippac.org

Outline

- Regular Languages, Applicatives and Visibly Pushdown Languages
- Transformations
- Layout
- State Hacks
- Incremental Parsing
- Beyond Parsing
- Summary, Questions

Regular Languages

Regular languages are defineable using regular expressions:

e	::=	A, B, C	Terminals / tokens
	|	$e.e$	Sequencing
	|	$e \mid e$	Choice
	|	e^*	Iteration

Context-Free Languages

Context-Free Languages are defineable using EBNF. EBNF is essentially a recursive let block around regular expressions:

l	::=	let	$nt_0 = e_0$	Non-terminal bindings
			$nt_1 = e_1$	
			...	
		in	e	Starting production
e	::=	...		Per regular expressions
	|	nt		Non-terminals

Applicatives and Kleene Algebras

Applicative parsers in Haskell correspond to CFLs - we turn a finite recursive specification into an infinite grammar.

Finite applicative parsers correspond to regular languages. In fact, they almost form a kleene algebra, with `pure` as the empty word and `<*>` as .

Exception:

```
pure Var <*> identifier
≠ identifier
≠ identifier <*> pure Var
```

Visibly Pushdown Languages

Visibly Pushdown Languages are defineable with a variant of EBNF – all recursion is bracketed.

l	::=	let	nt_0 = e_0	Non-terminal bindings
			nt_1 = e_1	
			...	
		in	e	Starting production
e	::=	...		Per regular expressions
	|	$O\ nt\ C$		Bracketed non-terminals

O, C and A,B,C are disjoint sets of tokens

Applicative VP Parsers

Take a finite applicative parser, add a ‘`bracket`’ operation for recursion and we can parse VPLs. For a finite representation use a tagging monad around the applicative like so:

```
do rec digit <- tag $ foldr1 (<|>)
                                 (map token ['0'..'9'])
       number <- tag $ Val . read <$> some digit
       fac <- tag $ number
                    <|> bracket lparen rparen expr
       expr <- tag $ fac `chainl1`
                     (token '+' *> pure Add)
   return expr
```

Pretty-printed Applicative

```
#0 = '0' <|> '1' <|> '2' <|> '3' <|> '4'
     <|> '5' <|> '6' <|> '7' <|> '8' <|> '9'
#1 = <PURE0> <*> (<PURE1> <*> #0)
     <*> Many #0
#2 = #1 <|> '(' #3 ')'
#3 = (<PURE2> <*> #2) <*>
     Many ((<PURE3> <*> (<PURE4> <*> '+')
            <*> <PURE5>)
           <*> #2)

In #3
```

Transformations and Analysis

Lots of operations on VPLs are closed!
Union, intersection, difference, negation...

If you can do it to a regex, you can lift it to VPLs with a bit of book-keeping. Locally, VPLs *are* regexes – and bracketing syncs recursion.

Visibly Pushdown Automata can thus be determinised.

(see Alur & Madhusudan, Visibly Pushdown Languages (2005))

Backtracking and Non-determinism

You can't quite determinise the pure aspects of an applicative VP parser, but consumption can be determinised.

This means you know exactly where the remaining non-determinism is, however you choose to handle it.

LL(1), LL(ω), we can figure it out every time and for each production individually.

No more try, no more commit!

Layout

You can do Haskell-style layout for VPLs, or anything you can erase to them.

No need for a parse-error rule:

Identify tokens that can’t appear in a given production (e.g. commas)

Use them to start popping the layout stack in emulation of parse-error

State Hacks

Imperative parser generators and monadic parsing combinators support state during parsing.

Common use cases: Symbol tables, position counting, context sensitivity hacks

Applicative VPL parsing can do something similar using operations similar to the ArrowChoice class – context-sensitive but statically-structured choice.

This follows similar laws to $<|>$, leaving all our transformations and analyses intact!

Even Bigger Stack Hacks

We don't have to use full state however, we can use a Reader-like stack discipline.

Even better, we can use state to work around the Visibly Pushdown limitation! Using a lexer with state access, we can decide whether < is $<_{operator}$ or $<_{bracket}$ by checking whether we're currently parsing a term or a type.

Incremental Parsing

Using insight and technique from Edward Kmett, we can do incremental parsing for VPLs as well.

The key elements of our grammars are monoidal - sequencing, choice, contextual choice, parsing results, state.

We can store the parse tree in a finger tree. We can resume part way through and thanks to the visibly pushdown property we also know when we can reuse the rest of the previous parse – or when they can't be compatible due to bracket imbalance.

Beyond Parsing

VPLs aren’t just used for parsing - in fact, parsing is a minority application.

Much research has been put into VPLs for program analysis and XML processing. Can we combine this with static typing?

Is there a use for a BracketedApplicative class, and if so what is its most general form?

Exaggerated Claims?

Visibly Pushdown Applicative parsers can express a wide range of syntax using the state hack and occasionally staging (Haskell-style operators, anyone?)

They also offer a number of attractive technical possibilities, ranging from easy efficient implementations to incremental parsing. They play well with regex-based technology common in text editors and IDEs.

Exaggerated Claims?

All good programming language syntax can be expressed neatly in terms of Visibly Pushdown Applicative parsing.

Bibliography

- Alur & Madhusudan, Visibly Pushdown Languages (2005)
- `http://www.cs.uiuc.edu/~madhu/vpa/` – the Visibly Pushdown Languages page
- Hinze & Paterson, Finger Trees: A Simple General-purpose Data Structure
- Edward Kmett on monoidal parsing, `http://comonad.com/reader/category/parsing/` :
  - Slides from Hac Phi: All About Monoids
  - Iteratees, Parsec and Monoids (Slides)