Parsing CSP-CASL with Parsec

Andy Gimblett

Department of Computer Science
University of Wales Swansea

2006.11.20

Outline of talk

- Parsec
- CSP-CASL grammar
  - In particular, PROCESS grammar
- Parser organisation & implementation
  - Encoding of precedence
  - Left recursion/grammar transformation
  - The `chainl1` combinator
- Testing strategies & techniques
- Future work

Introducing Parsec

- A monadic parser combinator library for Haskell
- *Predictive*, *backtracking*, *infinite-lookahead*
- Builds on work of Wadler/Hutton/etc as described by Liam
- `GenParser tok st a`
  - Data type representing a parser
  - Parses tokens of type `tok`
  - User-supplied state `st`
  - Returns a value of type `a` on success
  - A `Monad` (and a `Functor` & `MonadPlus`)
- `Parser a` — synonym for `GenParser Char () a`
- `parse :: Parser a -> FilePath -> String -> Either ParseError a`
  - `data Either a b = Left a | Right b`

`GenParser` is a `Monad`

- `return :: a -> GenParser tok st a`
  - Always succeeds with value `x` without consuming any input
- `(>>=) :: GenParser tok st a -> (a -> GenParser tok st b) -> GenParser tok st b`
  - The **bind** operator
  - Allows us to sequence parsers using Haskell's `do`
  - Examples later...
- `fail :: String -> GenParser tok st a`
  - `(fail m)` always fails with error message `m`

Some basic parsers

- `char :: Char -> CharParser st Char`
  `(char c)` parses a single character `c` and returns it
- `string :: String -> CharParser st String`
  `(string s)` parses sequence of characters `s`
- `oneOf :: [Char] -> CharParser st Char`
  `(oneOf c)` succeeds if char in list; returns it
- `space :: CharParser st Char`
  Parses a whitespace character, returns it.
- `satisfy :: (Char -> Bool) -> CharParser st Char`
  Does what you'd expect :-)

Some simple (?) combinators

- `many :: GenParser tok st a -> GenParser tok st [a]`
  - `(many p)` — 0+ `p`s
- `sepBy :: GenParser tok st a -> GenParser tok st sep -> GenParser tok st [a]`
  - `(sepBy p sep)` — 0+ `p`s separated by `sep`
- `chainl1 :: GenParser tok st a -> GenParser tok st (a->a->a) -> GenParser tok st a`
  - `(chainl1 p op x)` — 1+ `p`s, separated by `op`.
  - Returns: left-assoc application of all functions returned by `op` to the values returned by `p`.
  - 0 occurrences? Returns `x`
  - Real example later

Two Combinators for choice & lookahead

- `(<|>) :: GenParser tok st a -> GenParser tok st a -> GenParser tok st a`
- `try :: GenParser tok st a -> GenParser tok st a`
- `(p <|> q)` — acts like `p` but if fails, acts like `q`
  - But only if `p` fails without consuming input!
  - *Predictive* — no backtracking
- `(try p)` — behaves like `p` unless there's an error
  - Error? No input consumed!
  - *Back-tracking*
  - Arbitrary lookahead
- `(try p) <|> q`

CSP-CASL without channels

Defined in *The CSP-CASL Summary* (WIP)

```
CSP-CASL-SPEC ::= data DATA-DEFN
                  process PROCESS-DEFN
                  end/

DATA-DEFN ::=    SPEC
               | SPEC-DEFN

PROCESS-DEFN ::=   PROCESS
  | REC-PROCESS
  | var/vars VAR-DECL; ...; VAR-DECL;/ . PROCESS
  | var/vars VAR-DECL; ...; VAR-DECL;/ . REC-PROCESS
```

- `SPEC` / `SPEC-DEFN` — CASL entities
- `PROCESS` — most of my work so far

PROCESS grammar

```
PROCESS ::= (PROCESS)
  | Skip | Stop
  | EVENT -> PROCESS
  | [] VAR: EVENT-SET -> PROCESS
  | PROCESS ; PROCESS
  | PROCESS [] PROCESS
  | PROCESS |~| PROCESS
  | PROCESS [| EVENT-SET |] PROCESS
  | PROCESS [ EVENT-SET || EVENT-SET ] PROCESS
  | PROCESS || PROCESS
  | PROCESS ||| PROCESS
  | PROCESS \ EVENT-SET
  | PROCESS [[CSP-RENAMING]]
  | if FORMULA then PROCESS else PROCESS
```

PROCESS grammar

- Operators from CSP
- Again, also several entities from CASL
- eg: EVENT-SET is CASL SORT; EVENT is CASL TERM
- Precedence rules in line with CSP
  - Renaming, hiding — highest
  - Prefix, multiple prefix
  - Sequence
  - External, internal choice
  - Parallel operators
  - Conditional — lowest
- (My early work: restricted subset, flexing Parsec muscles)

Organising a Parsec parser the HETS way

- `AS_CspCASL.hs` — abstract syntax
  - Data types for results of parsing
- `Parse_CspCASL.hs` — parsers
  - Functions for actually parsing
  - Transform text into entities from `AS_CspCASL.hs`
- `ccparse.hs` — wrapper
  - `main()`, essentially
- `CspCASL_Keywords.hs` — keyword definitions
  - Just factored out into a common place

Exceprts from AS_CspCASL.hs

```
data EVENT_SET = EventSet SORT deriving (Show,Eq)

data PROCESS = Skip | Stop
  | PrefixProcess EVENT PROCESS
  | InternalPrefixProcess VAR EVENT_SET PROCESS
  | ExternalPrefixProcess VAR EVENT_SET PROCESS
  | Sequential PROCESS PROCESS
  | ExternalChoice PROCESS PROCESS
  | InternalChoice PROCESS PROCESS
  | Interleaving PROCESS PROCESS
  | SynchronousParallel PROCESS PROCESS
  | GeneralParallel PROCESS EVENT_SET PROCESS
  | AlphaParallel PROCESS EVENT_SET EVENT_SET PROCESS
  | Hiding PROCESS EVENT_SET
  | Renaming PROCESS PRIMITIVE_RENAMING
  | ConditionalProcess FORMULA PROCESS PROCESS
```

The naïve approach to parsing

```
process :: AParser st PROCESS  -- (AParser from HETS)
process = (try parenthesised)
  <|> (try conditional)
  <|> (try synchronous) <|> (try parallel)
  <|> (try internal_choice) <|> (try external_choice)
  <|> (try sequence_process)
  <|> (try prefix) <|> (try multiple_prefix)
  <|> (try hiding) <|> (try renaming)
  <|> (try skip) <|> (try stop)
...
synchronous = do p <- process
                 token "||"
                 q <- process
                 return (SynchronousParallel p q)
```

Problems with the naïve parser

- (At least) two big ones. . .
- One: No actual encoding of precedence rules (although some attempt has been made)
- Strictly left-to-right
- `P ||| Q ; S` is `(P ||| Q) ; S`
  Should be `P ||| (Q ; S)`
- Two (worse): left-recursion
- How does `synchronous` ever fail?
- Fix with *grammar transformations* & thoughtful ordering
- (Though it turns out Parsec helps us a lot here)

Encoding priority

```
PROCESS ::= if FORMULA then PROCESS else PROCESS
  | PAR_PROCESS

PAR_PROCESS ::= CHOICE_PROCESS
  | PAR_PROCESS || CHOICE_PROCESS
  | PAR_PROCESS ||| CHOICE_PROCESS

CHOICE_PROCESS ::= SEQUENCE_PROCESS
  | CHOICE_PROCESS [] SEQUENCE_PROCESS
  | CHOICE_PROCESS |~| SEQUENCE_PROCESS

SEQUENCE_PROCESS ::= PREFIX_PROCESS
  | SEQUENCE_PROCESS ; PREFIX_PROCESS
...
PRIMITIVE_PROCESS ::= (PROCESS) | SKIP | STOP
```

Removing left recursion

- Suppose grammar contains something like
  `A -> Ap | BqA | Ar | C`
  (Two left-recursive productions)
- Separate productions into left-recursive & non-:
  `A -> BqA | C`
  `A -> Ap | Ar`
- Then add new non-terminal Z and rewrite as:
  `A -> BqA | BqAZ | C | CZ`
  `Z -> p | pZ | r | rZ`
- This grammar is equivalent but non-left-recursive
- Good news: `chainl1` does this for us

Using `chainl1`

```
-- SEQUENCE_PROCESS ::= PREFIX_PROCESS
--    | SEQUENCE_PROCESS ; PREFIX_PROCESS
seq_process :: AParser st PROCESS
seq_process = prefix_process `chainl1` seq_op

seq_op :: AParser st (PROCESS -> PROCESS -> PROCESS)
seq_op = try (do asKey semicolonS
                 return sequencing)

sequencing :: PROCESS -> PROCESS -> PROCESS
sequencing left right = Sequential left right
```

- Note `try` in `seq_op`
- Similarly in parallel ops: first try `|||`, then `||`
- Judicious use of `try` for 3-char lookahead

`PROCESS` parser — eventual structure

- Process parser ends up fairly simple
- 'Straightforward' translation of prioritised grammar
- `chainl1` to remove left-recursion
  - So a number of auxilliary functions
- No explicit grammar transformation for left-recursion
- No explicit 'left-factoring' of grammar (none necessary)
  - Another common transformation
  - `A -> Bq | Br | C` becomes

    ```
    A -> B | C
    B -> q | r
    ```
  - (Was necessary if doing explicit LR-removal)

Demo?

Maybe it's time for a demo?
Or even a look at the code...

Automating the testing

- Early days: write test text into `tests/amg.csp-casl`
- Edit that file every time you want to change what's tested
- Gets tedious very quickly — and do old tests still pass?
- Obvious idea: *automated testing*
- I know how to do this in python (`unittest.py`)
- 'Rolled my own' in Haskell for CSP-CASL
- `Testbed.hs` — test $>$ 50 parses

```
tests :: [(String, Process)]
tests = [("STOP", Stop),
         ...
```

The trouble with `Testbed.hs`

'What we expect' gets ++long ++quickly!

```
("((a -> STOP)[[b]] ||| STOP\c ; SKIP [] SKIP)[[d]]",
  (Renaming
  (Interleaving
  (Renaming (PrefixProcess "a" Stop) "b")
  (ExternalChoice (Sequential (Hiding Stop "c") Skip) Sk
  ) "d")),
```

- Testing non-trivial specs tedious/brittle
- Also, `"c"` not actually an `EVENT-SET` (for example)
- No ability to perform negative tests

Negative tests

- All tests so far have been ‘positive’
- “With input X we expect parse tree Y”
- What if X isn’t a valid text?
- Need to test:
  - That invalidity is recognised
  - That a ‘good’ (ie helpful!) error is raised
  - Fairly unsophisticated at the moment
- So, test strategy/suite needs **negative tests**

‘Next generation’ testing framework

- Positive tests based on round-trip parse/unparse
- Unparse: turn syntax tree back into text
  - AKA pretty-printing
  - Outputs: LaTeX and ASCII
  - Good support in Hets
- t_1 to tree to t_2, then ask: does $t_1 = t_2$?
- Doesn’t catch everything, but much easier
- Negative tests: need a way to specify desired error output
- Either case: test files in a test suite directory

Future work

- Complete current integration with HETS
- Process part: more than one process
- Unparsing/pretty-printing
- More on the testing, esp. negative tests
- Explicitly disallow ‘implicit parentheses’ in some cases
  - eg `P || Q ||| R`
- Channels — essentially a syntactic sugar
  - Not the only one, in fact
  - ‘Core’ parser — output amenable to reasoning about
  - ‘Sugared’ parser — input amenable to use for specification