Efficient, typed LR parsers

Émile Trotignon

INRIA - ENS Paris Saclay

2021

Intership goals

I interned under the guidance of François Pottier as part of team Cambium of INRIA Paris. I worked on Menhir, the LR(1) parser generator for OCaml. My goal was to improve the code generated by Menhir in two ways :

- Increasing safety by producing well-typed code.
- Introducing new optimisations for better performance.

Producing well-typed code was suggested in 2006 by François Pottier and Yann Régis-Gianas in *Towards efficient, typed LR parsers*.

Menhir

Menhir is a LR(1) parser generator for OCaml. LR(1) parsers generators :

- Take a grammar as input
- Construct a stack-automaton that recognises the grammar
- Generate code that executes the automaton on arbitrary inputs

LR(1) Parsers

Here is the definition of a grammar called *rcalc*, for *restricted calculus*, in menhir syntax:

```
%token <int> INT
%token PLUS LPAREN RPAREN EOL
%left PLUS
%start <int> main
%%
main:
| e=expr EOL { e }
expr:
| i = INT { i }
| LPAREN e=expr RPAREN { e }
| e1=expr PLUS e2=expr
    { e1 + e2 }
```

LR(1) Parsers

Here is the definition of a grammar called *rcalc*, for *restricted calculus*, in menhir syntax:

```
%token <int> INT
%token PLUS LPAREN RPAREN EOL
%left PLUS
%start <int> main
%%
main:
| e=expr EOL { e }
expr:
| i = INT { i }
| LPAREN e=expr RPAREN { e }
| e1=expr PLUS e2=expr
    { e1 + e2 }
```

- *Symbols* : `LPAREN`, expr
- *Tokens / terminal symbols* : `%token xxx`
- *Non-terminal symbols* : defined by a rule

LR(1) Parsers
Information

Two sequences of symbols : the *stack* and the *input*/*lexer*.
They form an interpretation of the original text: the stack could be `expr PLUS expr` and the input `PLUS INT EOL`.
The stack annotates symbols with automaton states.

LR(1) Parsers

Actions

The automaton has two kinds of actions : shifting and reducing. When parsing the previous exemple according to rcalc, the (stack, entry) pair would go through the following changes :

$$\begin{aligned}
(\text{expr PLUS expr}, \text{PLUS INT EOL}) &\xrightarrow[\text{reduce}]{} \\
(\text{expr}, \text{PLUS INT EOL}) &\xrightarrow[\text{shift}]{} \\
(\text{expr PLUS}, \ \text{INT EOL})&
\end{aligned}$$

LR(1) Parsers

Routines

The automaton is implemented with three kind of *routines* :

- run
- reduce
- goto

The old code backend

```
type token = RPAREN | PLUS | LPAREN | INT of int | EOL

let rec (...) and _run_01:
  _env -> 'ttv_tail -> _state -> 'ttv_return =
 fun _env _stack _s ->
  (* State 1: *)
  let _stack = (_stack, _s) in
  let _env = _discard _env in
  let _tok = _env._token in
  match _tok with
  (* Shifting (INT) to state 2 *)
  | INT _v -> _run2 _env (Obj.magic _stack) S1 _v
  (* Shifting (LPAREN) to state 1 *)
  | LPAREN -> _run1 _env (Obj.magic _stack) S1
  (* Error handling ommited *)
  | _ -> assert false
```

Why an unsafe type cast is required

It may not be apparent yet why Obj.magic is required.

```
and _goto_expr:
 _env -> 'ttv_tail -> _state -> 'tv_expr -> 'ttv_return =
 fun _env _stack _s _v ->
  let _stack = (_stack, _s, _v) in
  match _s with
  | S1 -> _run3 _env (Obj.magic _stack)
  | S5 -> _run6 _env (Obj.magic _stack)
  | S0 -> _run8 _env (Obj.magic _stack)
```

The runtime analysis of `_s` discover information about the type of `_stack`.

Generalised algebraic datatypes

GADTS

- GADTs are a generalisation of regular ADT.
- Lets look at regular ADTs as a starting point.

ADTs

```
type direction = Left | Right
```

Or with a payload :

```
type nullable_int = Null | NonNull of int
```

Generalised algebraic datatypes

GADTs add the possibility of having different constructors that have different types :

```
type 'a my_gadt =
  ConsChar: char my_gadt | ConsFloat: float my_gadt
```

The expression `ConsInt` has type `int my_gadt`, and the expression `ConsFloat` has type `float my_gadt`.
We can also write a function as such:

```
let f: type t. t my_gadt -> t =
  function ConsChar -> 'c' | ConsFloat -> 0.
```

Generalised algebraic datatypes

Say that we want to write a function that takes one argument.
That argument is either a float or an int.
The function returns the argument as such if it is an int, or calls `int_of_float` on it if it is a float.

Generalised algebraic datatypes

Say that we want to write a function that takes one argument.
That argument is either a float or an int.
The function returns the argument as such if it is an int, or calls `int_of_float` on it if it is a float.

With ADTs

```
type number = TagInt of int | TagFloat of float
let f_adt: number -> int = function
    TagInt i -> i | TagFloat f -> int_of_float f
```

Generalised algebraic datatypes

Say that we want to write a function that takes one argument.
That argument is either a float or an int.
The function returns the argument as such if it is an int, or calls `int_of_float` on it if it is a float.

With ADTs

```
type number = TagInt of int | TagFloat of float
let f_adt: number -> int = function
    TagInt i -> i | TagFloat f -> int_of_float f
```

With magic

```
type tag = TagInt | TagFloat
let f_unsafe: tag -> 'a -> int =
  fun tag number ->
  match tag with
  | TagInt -> (Obj.magic number: int)
  | TagFloat -> int_of_float (Obj.magic number: float)
```

Generalised algebraic datatypes

We get the best of both worlds with GADTs :

```
type 'number tag = TagInt: int tag | TagFloat: float tag
let f_gadt: type number. number tag -> number -> int = fun tag n
  match tag with
   | ConsInt -> number
   | ConsFloat -> int_of_float number
```

`f_unsafe` is analogous to the code produced by old code backend, and we want to transform it to look like `f_gadt`.

Generalised algebraic datatypes

Regarding our example _goto_expr, the desired code may look like something like this:

```
and _goto_expr:
 type tail. _env -> tail -> tail _state -> _ -> int =
 fun _env _stack _s _v ->
  let _stack = (_stack, _s, _v) in
  match _s with
   | S1 -> _run_03 _env _stack
   | S5 -> _run_06 _env _stack
   | S0 -> _run_08 _env _stack
```

Here the _state type is a GADT.

Generalised algebraic datatypes

The _state type

```
type ('tail, 'semantic) _cell_1111 =
  'tail * 'tail _state * 'semantic * position * position

and 't_tail _state =
  | S5: (('t_tail, int) _cell_1100) _state
  | S1: ('t_tail _cell_1000) _state
  | S0: 't_tail _state
```

Code generation
Stacklang

Issue

The old code backend : automaton $\rightarrow$ OCaml
This is not very practical :

- the automaton representation is too stiff.
- the representation of OCaml code is too low-level.

Solution

An intermediate language called *Stacklang*, that need the four following properties :

1. Easy to compile a stack automaton to it.
2. Easy to manipulate and transform for optimisations.
3. Easy to compile to OCaml.
4. Possible to compile it to OCaml without compromising type safety or efficiency.

Stacklang

Here is an exemple of a run routine associated to *rcalc* in Stacklang :

```
routine _run_01 ( _lexbuf _lexer _s ) :
  { cells : [], final-type : int } =
  _startpos <- startpos _lexbuf ;
  _endpos <- endpos _lexbuf ;
  push _s ; /* pushing LPAREN0100 */
  _s <- 1 ;
  _tok <- next-token _lexbuf _lexer ;
  match _tok with
  /* Shifting (LPAREN) to state 1 */
  | ( LPAREN of _v )  -> jump _run_01
  /* Shifting int to state 2 */
  | ( INT of _v ) -> jump _run_02
  | _ -> die
```

Stacklang → OCaml

```
and _run_01 :
  type t_tail.
    t_tail -> _ -> _ -> (t_tail, int) _state -> int =
  fun _stack _lexbuf _lexer _s ->
    let _startpos = _lexbuf.lex_start_p in
    let _endpos = _lexbuf.lex_curr_p in
    let _stack = (_stack, _s) in
    let _s = S1 in
    let _tok = _discard _lexer _lexbuf in
    match _tok with
    | LPAREN ->
        let _v = () in
        _run_01 _stack _lexbuf _lexer _s
    | INT _v -> _run_02 _stack _lexbuf _lexer _s _v
    | _ -> raise _eRR
```

OCaml typing

Let's look more closely at the type annotation of the run routine :

```
type t_tail.
  t_tail -> _ -> _ -> (t_tail, int) _state -> int
```

What this type annotation tells us is :

1. There are no known cells on top of the stack.
2. The `_s` variable/last argument describes that stack correctly.

Goto routine

in Stacklang

```
routine _goto_expr
  ( _lexbuf _lexer _s _tok _v ) :
  { cells : [], final-type : int } =
  match _s with
  | 0 -> jump _run_08
  | 5 -> jump _run_06
  | 1 -> jump _run_03
```

Goto routine
in OCaml

```
let rec _goto_expr :
  type t_tail.
  t_tail -> _ -> _ ->
  (t_tail, int) _state ->
  _ -> _ -> int =
  fun _stack _lexbuf _lexer _s _tok _v ->
    match _s with
    | S0 -> _run_08 _stack _lexbuf _lexer _s _tok _v
    | S5 -> _run_06 _stack _lexbuf _lexer _s _tok _v
    | S1 -> _run_03 _stack _lexbuf _lexer _s _tok _v
```

Reduce routine

In Stacklang

```
_reduce_3 (_lexbuf, _lexer, _tok):
  { cells: [expr1100; expr1100], final-type: int } =
  pop (_, _3) ;
  pop (_s, _1) ;
  /* Reducing production expr -> expr PLUS expr */
  _v <- prim < _1 + _3 > ;
  jump _goto_expr
```

Reduce routine

In OCaml

```
and _reduce_3:
  type t_tail.
    ( (t_tail, int, int) _cell_1100
    , int
    , int ) _cell_1100 ->
    _ -> _ -> _ -> int =
  fun _stack _lexbuf _lexer _tok ->
    let (_stack, _, _3) = _stack in
    let (_stack, _s, _1) = _stack in
    (* Reducing production expr -> expr PLUS expr *)
    let _v = ( _1 + _3 ) in
    _goto_expr _stack _lexbuf _lexer _s _tok _v
```

Final types

Menhir allows you to have multipe entry points with arbitrary types.
Multiple exposed functions with different return types but same run routines.
Without the `final-type` argument : an error.
The final type is known in the routine and states that are accessible only by one entry point.

For instance if the following rule to `rcalc`, and make it an entry point :

```
main2:
| v=separated_list(COMMA, expr) EOL
    { v }
```

Then the states are as such :

```
states=
{ 16: { cells: [expr1100] ; final-type: int list }
; 10: { cells: [] ; final-type: int list }
; 5: { cells: [expr1100] }
; 1: { cells: [LPAREN0100] }
; 0: { cells: [] ; final-type: int } }
```

Optimisations

A few optimisation were implemented :

- Push-pop cancelling
- Inlining
- Single-branch match deletion

Push-pop cancelling

```
push (s, v) ;
pop (s', v') ;
```

Push-pop cancelling

```
push (s, v) ;
pop (s', v') ;
```

Can be simplified as :

```
(s', v') <- (s, v) ;
```

Push-pop cancelling

The previous situation never happens.

```
push (s, v) ;
a <- b ;
pop (s', v') ;
```

Is is more likely.
It can be simplified as :

```
a <- b ;
(s', v') <- (s, v) ;
```

We call this tranformation *push commutation*.

Push commutation

Conflicts

Some domain conflicts may arise :

```
push (s, v) ;
s <- b ;
pop (s', v') ;
```

Is equivalent to :

```
(s, s', v') <- (b, s, v) ;
```

but not to :

```
s <- b ;
(s', v') <- (s, v) ;
```

Push commutation
Conflicts

```
push (s, v) ;
s' <- a ;
pop (s', v') ;
```

is equivalent to :

```
(s', v') <- (s, v) ;
```

not

```
(s', v', v') <- (s, v, a) ;
```

To keep track of that, we need to commute the writes in front of the pushes.

Inlining

We need inlining to provide opportunity for push commutation.

What do we inline ?

We use a ad-hock method :
For every routine, we compute its degree, that is the number of jump instruction to it present in the program.
We then inline every routine that has degree $\leq n$. (n is 2 by default)

Single branch match removal

A match on a state does not bind anything.
This means that a single branch match could be optimized away.
The OCaml compiler cannot do it because it adds a catch-all clause.
Careful about not losing typing information :

```
routine
  _goto_example (_s, v) :
  {cells:[]} =
  match _s with
  | 5 -> jump _run_02
```

```
routine
  _goto_example (_s, v) :
  {cells:[]} =
  jump _run_02
```

The routine on the right will be translated to ill-typed code if `_run_02` expects cells to be present on the stack.

Typing Stacklang

The optimisation above requires to track typing information. This is not the only one that does so, inlining and push commutation bring their typing issues that I deemed to complex to explain here. This is very easy to get wrong, and if you do get it wrong, understanding what happened with the resulting OCaml errors will be very hard to do.
Solution : an OCaml function that type-checks a Stacklang program.

Benchmark

Three grammars tested : Calc, JSON and Houblix.
Multiple measurements :

- Time in seconds per billion tokens.
- Code size of the `parser.o` in kilobytes.
- Words per token allocated on the minor heap.
- Words per token allocated on the major heap.

Multiple backends :

- *Code* : the new code backend, with default optimisations.
- *Old code* : the old one, that still uses `Obj.magic`.
- *Code no commute* : the new code backend, with push commutation disabled.
- *Code no inlining* : the new code backend with no inlining.
- *Code high inlining* : max inlining degree of 5 (default of 2)

Benchmark

JSON

Backend				
Metric	new	old	no commute	high inlining
Code size (ko)	32	29	30	89
Time (s/Gtoks)	59.1	74.5	66.0	58.8
Alloc minor (words/toks)	3.79	11.60	9.00	3.79
Alloc major (words/toks)	1.86	2.19	1.96	1.86

Average AST size : 2.2 words per token.

Benchmark

The new code backend is very efficient

Allocations divided by a value between 3 and 6.
Exectution faster by a coefficient between 1.1 and 1.3 faster.

The commutation of the pushes has a big effect.
Inlining more aggressively has an effect, but the drawback in code size is huge.

Conclusion

Results

- Increase in safety
- Increase in efficiency

Further work

- Smart inlining
- Remove dead catch-all branches.