Towards efficient, typed LR parsers

François Pottier and Yann Régis-Gianas

September 2005

In short

This talk is meant to illustrate how an expressive type system allows guaranteeing the safety of complex programs.

The programs considered here are *LR parsers* and the type system is *an extension of ML with generalized algebraic data types (GADTs)*.

In short

There are tools that *generate*, out of an LR grammar, a program that simulates execution of the corresponding pushdown deterministic automaton (PDA).

Can one guarantee the *safety* of the generated program without requiring *trust* in the tool's correctness ?

What do existing tools produce ?

Yacc, Bison, OCamlYacc etc. produce C programs, with *no safety guarantee.* They use a *untagged union* to represent semantic values, and do not protect against stack underflow.

ML-Yacc or Happy produce ML or Haskell programs, which are typed. They implement semantic values using a *tagged union*. Yet, *runtime exceptions still arise* when pattern matching fails, so safety isn't quite guaranteed.

A simple grammar

Here is a very simple LR grammar, drawn from the "Dragon Book" :

$$
\begin{array}{lrcl}
(1) & E\{x\} + T\{y\} & \rightarrow & E\{x+y\} \\
(2) & T\{x\} & \rightarrow & E\{x\} \\
(3) & T\{x\} * F\{y\} & \rightarrow & T\{x \times y\} \\
(4) & F\{x\} & \rightarrow & T\{x\} \\
(5) & (E\{x\}) & \rightarrow & F\{x\} \\
(6) & \textbf{int}\{x\} & \rightarrow & F\{x\}
\end{array}
$$

The *terminals* or *tokens* are +, *, (,), and **int**. The *non-terminals* are E, T, and F. The first four have no semantic value ; the last four have an integer semantic value.

Lexer interface

Tokens are made up of a tag and possibly of a semantic value :

type *token* = *KPlus* | *KStar* | *KLeft* | *KRight* | *KEnd* | *KInt* of *int*

The lexer provides two functions for looking up and for discarding the current token :

val *peek* : *unit* → *token*
val *discard* : *unit* → *unit*

Data structures

The type of states is easily defined :

type *state* = *S0* | *S1* | ... | *S11*

Data structures (cont'd)

$$
\begin{array}{l}
\text{type } \mathit{stack} = \\
\quad \mid \mathit{SEmpty} \\
\quad \mid \mathit{SP} \text{ of } \mathit{stack} \times \mathit{state} \\
\quad \mid \mathit{SS} \text{ of } \mathit{stack} \times \mathit{state} \\
\quad \mid \mathit{SL} \text{ of } \mathit{stack} \times \mathit{state} \\
\quad \mid \mathit{SR} \text{ of } \mathit{stack} \times \mathit{state} \\
\quad \mid \mathit{SI} \text{ of } \mathit{stack} \times \mathit{state} \times \mathit{int} \\
\quad \mid \mathit{SE} \text{ of } \mathit{stack} \times \mathit{state} \times \mathit{int} \\
\quad \mid \mathit{ST} \text{ of } \mathit{stack} \times \mathit{state} \times \mathit{int} \\
\quad \mid \mathit{SF} \text{ of } \mathit{stack} \times \mathit{state} \times \mathit{int}
\end{array}
$$

(P, S, L, R, I are shorthands for *Plus, Star, Left, Right* and *Int)*

Implementation (general structure)

The automaton is simulated by *run*. Out of the current state, stack, and (implicitly) token stream, this function either produces a semantic value for the entire parse or fails.

```
let rec run (s : state) (stack : stack) : int =
  match s, peek() with
  | ... (* shift or reduce transitions *)
  | _, _ →
      raise SyntaxError
```

Implementation (shift)

A *shift* transition pushes the current state and the semantic value for the current token onto the stack, discards the current token, and changes the current state :

```
let rec run (s : state) (stack : stack) : int =
  match s, peek() with
  | ...
  | S9, KStar → (* shift S7 *)
      discard ();
      run S7 (SS (stack, S9))
  | ...
```

Implementation (reduce)

A *reduce* transition pops a number of semantic values off the stack and exploits them to compute a new one, which is pushed back onto the stack.

```
let rec run (s : state) (stack : stack) : int =
  match s, peek() with
  | ...
  | S9, KPlus → (* reduce E{x} + T{y} → E{x + y} *)
      let ST (SP (SE (stack', s', x), _), _, y) = stack in
      let stack'' = SE (stack', s, x + y) in
      gotoE s' stack'' (* goto E *)
  | ...
```

Observe that *pattern matching is nonexhaustive*.

Implementation (end)

A *goto* transition examines the state that was popped off the stack during reduction and changes the current state.

```
and gotoE (s : state) : stack → int =
  match s with
  | S0 →
      run S1
  | S4 →
      run S8
```

Again, *pattern matching is nonexhaustive*.

Why are these tests redundant ?

The dynamic tests performed during the previous *reduce* transition are redundant because, when the automaton is in state S_9, the stack must be of the form

$$\ldots \quad S_? \quad E \quad S_? \quad + \quad S_? \quad T$$

The dynamic tests performed during the previous *goto* E transition are redundant because, when the automaton is in state S_9, the stack must be of the form

$$\ldots \quad (S_0 \mid S_4) \quad ? \quad S_? \quad ? \quad S_? \quad ?$$

The invariant (fragment)

In fact, one can prove that, when the automaton is in state S_9, the stack must be of the form

$$\ldots \quad (S_0 \mid S_4) \quad E \quad (S_1 \mid S_8) \quad + \quad S_6 \quad T$$

The stack's shape can be known similarly for every state.

This *invariant* is proved by induction during the execution of the automaton. We want the typechecker to do this proof ...

The idea

One must tell the compiler about the *correlation* between the current state and the structure of the stack.

To this end, one parameterizes the type *state* with a type variable α. The idea is, *if the current state has type α state, then the current stack has type α*.

The structure of stacks

The type *stack* *disappears*. The structure of stacks is defined by a family of parameterized types, which are *independent* one of another :

type *empty* = *SEmpty*
type α *cP* = *SP* of $\alpha \times \alpha$ *state*
type α *cS* = *SS* of $\alpha \times \alpha$ *state*
type α *cL* = *SL* of $\alpha \times \alpha$ *state*
type α *cR* = *SR* of $\alpha \times \alpha$ *state*
type α *cI* = *SI* of $\alpha \times \alpha$ *state* $\times$ *int*
type α *cE* = *SE* of $\alpha \times \alpha$ *state* $\times$ *int*
type α *cT* = *ST* of $\alpha \times \alpha$ *state* $\times$ *int*
type α *cF* = *SF* of $\alpha \times \alpha$ *state* $\times$ *int*

Encoding the invariant (fragment)

The fact that, when the automaton is in state S_9, the stack must be of the form

$$\ldots \quad S_? \quad E \quad S_? \quad + \quad S_? \quad T,$$

is encoded by assigning the type

$$\forall\alpha.\alpha\ \mathit{cE}\ \mathit{cP}\ \mathit{cT}\ \mathit{state}$$

to the data constructor *S9* and similarly for other states.

Such a declaration is impossible in ML ! The type *state* is a *generalized algebraic data type* (GADT).

The structure of states

type $state : * \longrightarrow *$ where
| *S0 : empty state*
| *S1 : empty cE state*
| *S2* : $\forall\alpha.\alpha$ *cT state*
| *S3* : $\forall\alpha.\alpha$ *cF state*
| *S4* : $\forall\alpha.\alpha$ *cL state*
| *S5* : $\forall\alpha.\alpha$ *cI state*
| *S6* : $\forall\alpha.\alpha$ *cE cP state*
| *S7* : $\forall\alpha.\alpha$ *cT cS state*
| *S8* : $\forall\alpha.\alpha$ *cL cE state*
| *S9* : $\forall\alpha.\alpha$ *cE cP cT state*
| *S10* : $\forall\alpha.\alpha$ *cT cS cF state*
| *S11* : $\forall\alpha.\alpha$ *cL cE cR state*

Implementation (general structure)

The type of *run* changes : it now accepts an arbitrary state and a stack *whose structure is consistent with respect to that state.*

```
let rec run : ∀α.α state → α → int =
  fun s stack →
    match s, peek() with
    | ...
    | _, _ →
        raise SyntaxError
```

Implementation (reduce)

The code for *reduce* transitions is also unchanged, but *pattern matching is now exhaustive*.

```
let rec run : ∀α.α state → α → int =
  fun s stack →
    match s, peek() with
    | S9, KPlus →
        (* There exists an unknown β such that: *)
        (* S9 : β cE cP cT state *)
        (* s : β cE cP cT state *)
        (* α = β cE cP cT} *)
        (* Thus stack : β cE cP cT *)
        let ST (SP (SE (stack', s', x), _), _, y) = stack in
        ...
```

Results

We have encoded part of the invariant into data type declarations and into the types ascribed to *run* and *goto*. In fact, the whole invariant can be encoded (see the paper).

Ideas to take home

Typechecking *involves proving* the invariant. Trusting the compiler remains necessary, unless of course a certifying compiler is used.

Pattern matching provides *type equations with local scope*. Shared type variables allow *coordinating (physically disjoint) data structures*.

Future work and References

The integration of GADTs into O'Caml is work in progress.

Slides, draft paper, and prototype implementations of the typechecker and parser generator are available online :

http://cristal.inria.fr/~regisgia/