Towards efficient, typed LR parsers

François Pottier and Yann Régis-Gianas

June 2005

INRIA

Introduction

An automaton

An ML implementation

Beyond ML

Conclusion

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)*.

LR parsers

People like to *specify* a parser as a context-free grammar, typically in BNF format, decorated with semantic actions.

People like to *implement* a parser as a deterministic pushdown automaton (DPDA).

A grammar is LR if such an implementation is possible.

LR parser generators

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

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

What do existing tools produce?

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

ML-Yacc or Happy produce ML or Haskell programs, which are typed. Yet, *runtime exceptions still arise* when pattern matching fails, so safety isn't quite guaranteed. Furthermore, *redundant dynamic tests* incur a runtime penalty.

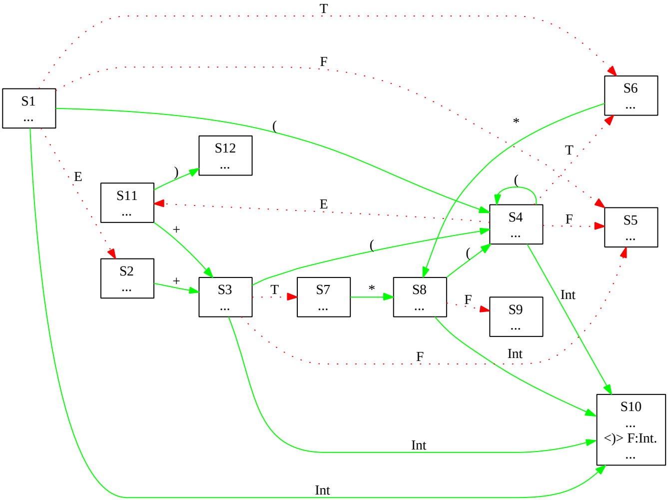
Before showing any *code*, let's have a look at a sample grammar and automaton.

A simple grammar

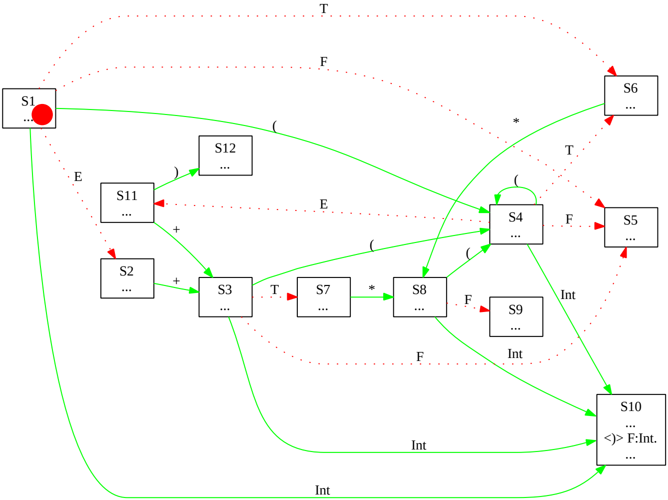
Here is a very simple LR grammar, drawn from the "Dragon Book:"

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

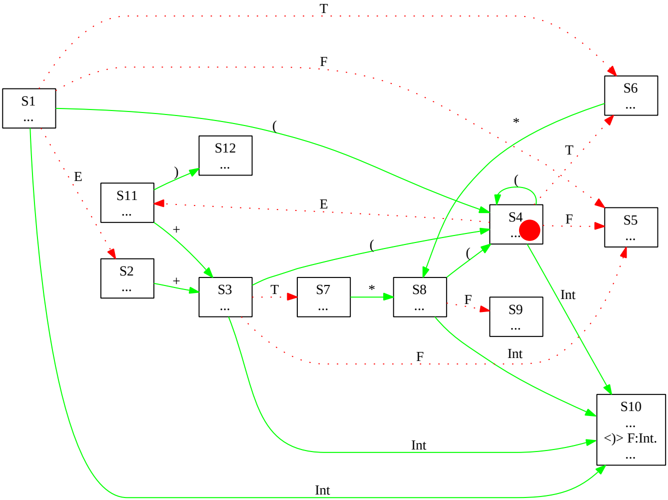
The *terminals* or *tokens* are **+**, *****, **(**, **)**, and **int**. The *non-terminals* are E, T, and F. The first four have no semantic value; the last four have an integer semantic value.



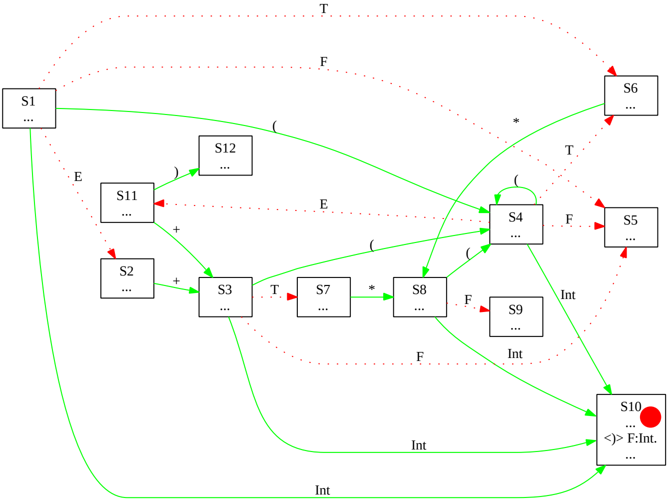
Here is a pushdown automaton that accepts this grammar.



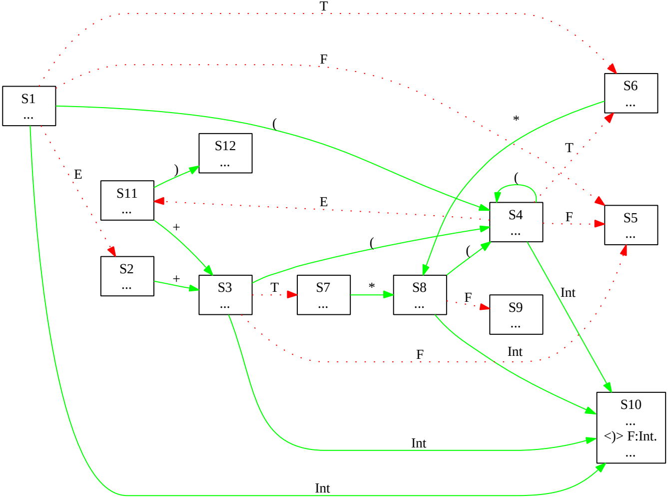
Input	Stack	State	Next action
(**int**) $	ϵ	S_1	shift S_4



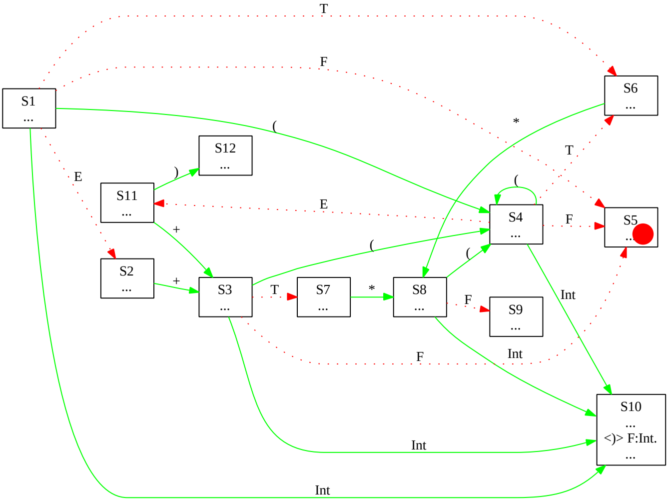
Input	Stack	State	Next action
int) $	S_1 (	S_4	shift S_{10}



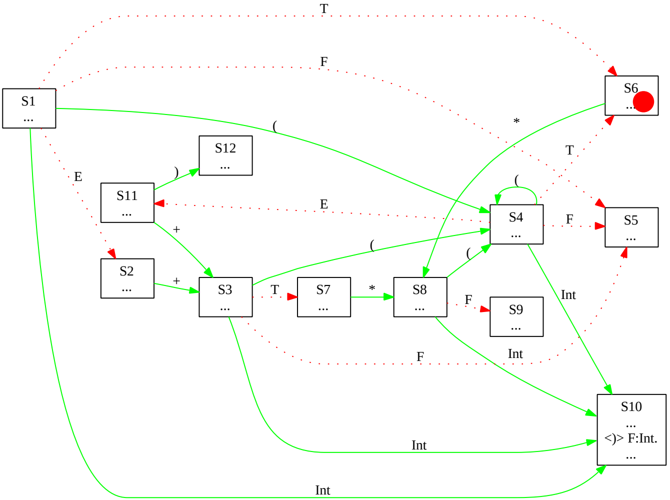
Input	Stack	State	Next action
) $	S_1 (S_4 **int**	S_{10}	reduce **int** $\rightarrow F$, goto F



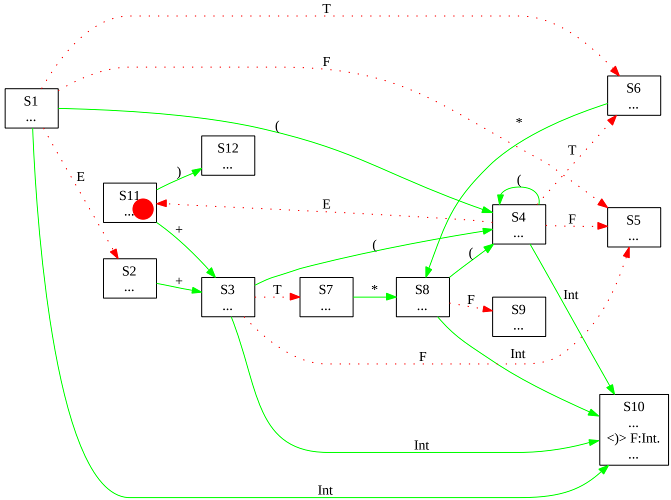
Input	Stack	State	Next action
) $	S_1 (S_4 *F*		*goto F*



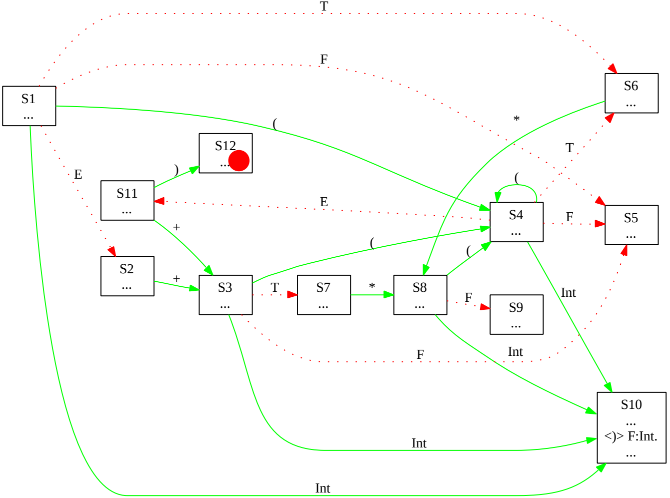
Input	Stack	State	Next action
) \$	S_1 (S_4 F	S_5	reduce $F \longrightarrow T$, goto T



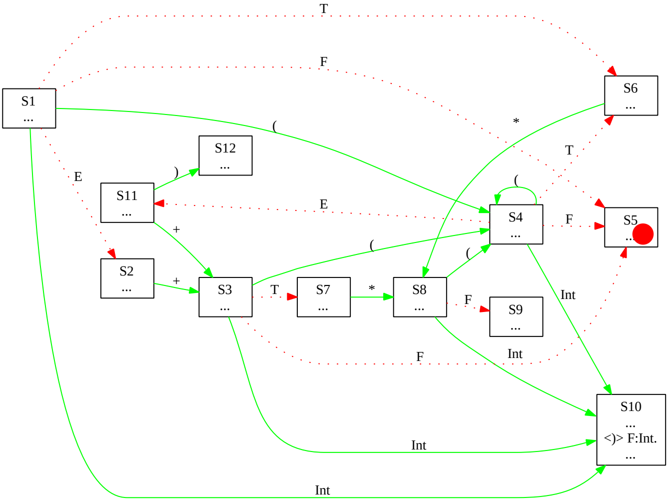
Input	Stack	State	Next action
) $	S_1 (S_4 T	S_6	reduce $T \longrightarrow E$, goto E



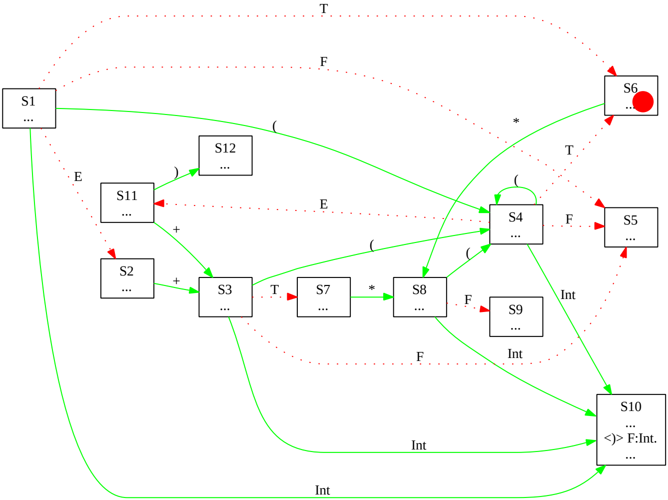
Input	Stack	State	Next action
) $	S_1 (S_4 E	S_{11}	shift S_{12}



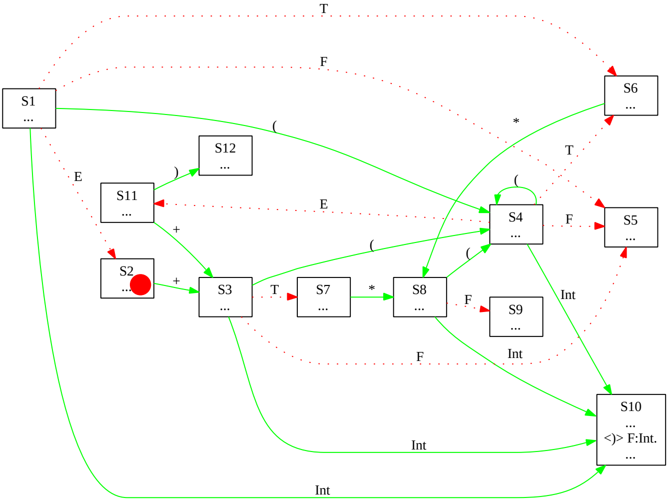
Input	Stack	State	Next action
\$	S_1 (S_4 E S_{11})	S_{12}	reduce $(E) \longrightarrow F$, goto F



Input	Stack	State	Next action
\$	S_1 F	S_5	reduce $F \longrightarrow T$, goto T



Input	Stack	State	Next action
\$	$S_1\ T$	S_6	reduce $T \longrightarrow E$, goto E



Input	Stack	State	Next action
$	S_1 E	S_2	accept

Introduction

An automaton

An ML implementation

Beyond ML

Conclusion

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* = $S0 \mid S1 \mid \ldots \mid S11$

Data structures (cont'd)

The stack is made up of pairs of a state and a semantic value whose type depends on the non-terminal with which it is associated. This is a linked list of *tagged* cells.

```
type stack =
  | SEmpty
  | SPlus of stack × state
  | SStar of stack × state
  | SLeft of stack × state
  | SRight of stack × state
  | SInt of stack × state × int
  | SE of stack × state × int
  | ST of stack × state × int
  | SF of stack × state × 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 (SStar (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 (SPlus (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*.

In short

This program is considered well-typed by an ML compiler. Yet, the compiler warns about nonexhaustive pattern matching, which means that *the absence of runtime failures is not guaranteed.*

The problem is to modify the program so that every pattern matching becomes exhaustive. Suppressing redundant dynamic tests will lead to a *safety guarantee* as well as *better efficiency*.

Introduction

An automaton

An ML implementation

Beyond ML

Conclusion

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

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

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

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

The invariant (fragment)

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

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

More generally, knowledge of the current state determines a *suffix* of the stack...

The full invariant

Stack							State
ϵ							S_0
ϵ	S_0	E					S_1
$\ldots$	$(S_0 \mid S_4)$	T					S_2
$\ldots$	$(S_0 \mid S_4 \mid S_6)$	F					S_3
$\ldots$	$(S_0 \mid S_4 \mid S_6 \mid S_7)$	(					S_4
$\ldots$	$(S_0 \mid S_4 \mid S_6 \mid S_7)$	**int**					S_5
$\ldots$	$(S_0 \mid S_4)$	E	$(S_1 \mid S_8)$	+			S_6
$\ldots$	$(S_0 \mid S_4 \mid S_6)$	T	$(S_2 \mid S_9)$	*			S_7
$\ldots$	$(S_0 \mid S_4 \mid S_6 \mid S_7)$	(	S_4	E			S_8
$\ldots$	$(S_0 \mid S_4)$	E	$(S_1 \mid S_8)$	+	S_6	T	S_9
$\ldots$	$(S_0 \mid S_4 \mid S_6)$	T	$(S_2 \mid S_9)$	*	S_7	F	S_{10}
$\ldots$	$(S_0 \mid S_4 \mid S_6 \mid S_7)$	(	S_4	E	S_8	)	S_{11}

Towards more precise types

It is easy to *manually* prove, by structural induction over a run of the automaton, that the invariant is sound.

For this invariant to be exploited by the compiler, it has to be *explicitly* provided and *mechanically* verified.

The programming language must come with a type system that is sufficiently expressive to allow encoding the invariant.

The idea

On 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* of one another:

type *empty* = *SEmpty*
type a *cellPlus* = *SPlus* of $a \times a$ *state*
type a *cellStar* = *SStar* of $a \times a$ *state*
type a *cellLeft* = *SLeft* of $a \times a$ *state*
type a *cellRight* = *SRight* of $a \times a$ *state*
type a *cellInt* = *SInt* of $a \times a$ *state* $\times$ *int*
type a *cellE* = *SE* of $a \times a$ *state* $\times$ *int*
type a *cellT* = *ST* of $a \times a$ *state* $\times$ *int*
type a *cellF* = *SF* of $a \times a$ *state* $\times$ *int*

(Compare to the original definition.)

Encoding the invariant (fragment)

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

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

is encoded by assigning the data constructor *S9* the type

$$\forall \alpha.\alpha \; cE \; cP \; cT \; state$$

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*
| $S2$: $\forall a.a$ *cT state*
| $S3$: $\forall a.a$ *cF state*
| $S4$: $\forall a.a$ *cL state*
| $S5$: $\forall a.a$ *cI state*
| $S6$: $\forall a.a$ *cE cP state*
| $S7$: $\forall a.a$ *cT cS state*
| $S8$: $\forall a.a$ *cL cE state*
| $S9$: $\forall a.a$ *cE cP cT state*
| $S10$: $\forall a.a$ *cT cS cF state*
| $S11$: $\forall a.a$ *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* : $\forall a. a\ state \rightarrow a \rightarrow int$ =
  fun *s* *stack* →
    match *s*, *peek*() with
    | ...
    | _, _ →
      raise SyntaxError

(Compare to the original type.)

Implementation (shift)

The code for *shift* transitions is unchanged, but typechecking becomes more subtle.

```
let rec run : ∀α.α state → α → int =
  fun s stack →
    match s, peek() with
    | S9, KStar →
        (* SStar (stack, S9) has type α cS *)
        (* run S7 has type ∀γ.γ cT cS → int *)
        (* Furthermore, α = β cE cP cT, for an unknown β *)
        (* Thus α cS = γ cT cS, where γ = β cE cP *)
        discard ();
        run S7 (SStar (stack, S9))
```

(Consult the definition of the type of states.)

Implementation (reduce)

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

```
let rec run : ∀a.a state → a → int =
  fun s stack →
    match s, peek() with
    | S9, KPlus →
        (* a = β cE cP cT, for an unknown β *)
        (* Thus stack : β cE cP cT *)
        let ST (SPlus (SE (stack, s, x), _), _, y) = stack in
        (* stack : β, s : β state, x : int, y : int *)
        let stack = SE (stack, s, x + y) in
        (* stack : β cE *)
        gotoE s stack
```

Implementation (end)

The type ascribed to *gotoE* states that at the top of the stack is a cell associated with the non-terminal *E* and that the remainder of the stack must be consistent with state *s*.

$$
\begin{array}{l}
\text{and } \mathit{gotoE} : \forall a. a\ \mathit{state} \rightarrow a\ \mathit{cE} \rightarrow \mathit{int} = \\
\quad \text{fun } s \rightarrow \\
\quad\quad \text{match } s \text{ with} \\
\quad\quad \mid S0 \rightarrow \\
\quad\quad\quad \mathit{run}\ S1 \\
\quad\quad \mid S4 \rightarrow \\
\quad\quad\quad (*\ \mathit{run}\ S8\ \mathit{has\ type}\ \beta\ \mathit{cL}\ \mathit{cE} \rightarrow \mathit{int},\ \mathit{for\ every}\ \beta\ *) \\
\quad\quad\quad (*\ \mathit{Furthermore},\ a = \beta\ \mathit{cL},\ \mathit{for\ an\ unknown}\ \beta\ *) \\
\quad\quad\quad \mathit{run}\ S8
\end{array}
$$

(Here, pattern matching remains nonexhaustive.)

In short

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.

Then, typechecking *involves proving* the invariant.

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

All this is typical of GADTs.

Results

We have obtained a *safety guarantee* about the generated parser, without requiring *trust* in the generator.

The tool that produces the automaton *knows the invariant, or thinks it knows,* and produces appropriate data type declarations without difficulty.

If the tool produces an incorrect program, the latter is rejected by the compiler.

Trusting the compiler remains necessary, unless of course a certifying compiler is used.

Towards more proofs in programs

We have exploited a very expressive type system to prove the *safety* of a program.

Proof assistants have allowed this, and more, for a long time. Here, however, we have remained within the framework of a *programming language* equipped, in particular, with a powerful type inference mechanism and with an extremely efficient compilation scheme.

Narrowing the gap between programming and proving is probably a worthy (long-term ?) research goal.

References

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

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