Towards efficient, typed LR parsers

François Pottier [1] and Yann Régis-Gianas [2]

INRIA

Abstract

The LR parser generators that are bundled with many functional programming language implementations produce code that is untyped, needlessly inefficient, or both. We show that, using generalized algebraic data types, it is possible to produce parsers that are well-typed (so they cannot unexpectedly crash or fail) and nevertheless efficient. This is a pleasing result as well as an illustration of the new expressiveness offered by generalized algebraic data types.

Key words: parsing, type safety, generalized algebraic data types

1 Introduction

It is well understood how to automatically transform certain classes of context-free grammars into fast, executable parsers [1]. For instance, every LALR(1) grammar can be turned into a compact deterministic pushdown automaton that recognizes the language generated by the grammar. This parser construction technique has been made available to users of many mainstream programming languages. As an example, `yacc` [7] turns LALR(1) grammars, decorated with pieces of C code known as *semantic actions*, into executable C parsers. In the functional programming realm, each of Standard ML [10], Objective Caml [8], and Haskell [12] comes with an adaptation of `yacc`. These tools are respectively known as `ML-Yacc` [23], `ocamlyacc` [8], and `happy` [9].

The parsers generated by `yacc` are fast, but are written in C, an unsafe language. Likewise, the automata produced by `ocamlyacc` are encoded as tables of integers and of Objective Caml function closures. They are interpreted, at runtime, by a piece of C code. It is not obvious, when examining the code for such parsers, why the automaton's stack cannot underflow, or why the numerous type casts used to store and retrieve semantic values into and out of the stack are safe. Thus, a bug in `yacc` or `ocamlyacc` could, in principle, cause a generated parser to crash at runtime. In practice, *users* of these tools

[1] Email: `Francois.Pottier@inria.fr`

[2] Email: `Yann.Regis-Gianas@inria.fr`

are not concerned with this issue, because they trust the tool to be correct. The *maintainers* of `yacc` or `ocamlyacc`, however, must be careful to preserve this property.

`ML-Yacc`, on the other hand, generates valid Standard ML code. Since Standard ML's type system is sound, a parser produced by `ML-Yacc` cannot crash at runtime. Is that more satisfactory? In fact, not much. Although a Standard ML program cannot *crash*, it may *fail* at runtime, due to either a nonexhaustive case analysis or an uncaught exception. Although such failures are usually easier to debug than arbitrary memory faults, they do abruptly terminate the program, so they are still a serious issue. Thus, the maintainers of `ML-Yacc` must again be careful to guarantee that a generated parser cannot fail. This fact is not obvious: indeed, `ML-Yacc` attaches *tags* to stack cells, to semantic values, and to automaton states, and uses nonexhaustive case analyses to examine these tags. Thus, a Standard ML parser only offers a limited robustness guarantee, and is less efficient than a C parser, because of the extra boxing and unboxing operations and extra dynamic checks that are required in order to please the typechecker.

`happy` lets users choose between these two evils. When the `-c` flag is supplied, unsafe Haskell code is produced, which in principle could crash at runtime. When it is not, valid Haskell code is produced, which cannot crash, but could still fail, and is less efficient.

To remedy this situation, we suggest writing parsers in a version of ML [3] equipped with a slightly more complex, but vastly more expressive, type system. The key extra feature that we require is known (among other names) as *generalized algebraic data types*. This notion, due to Xi, Chen, and Chen [26], was recently explored by a number of authors [4,14,21]. We show that appropriate use of generalized algebraic data types allows making a great amount of information about the pushdown automaton known to the typechecker. This, in turn, allows the typechecker to *automatically check that the parser cannot crash or fail*, even though *we no longer attach tags* to stack cells or semantic values. In short, we recover true type safety, while eliminating much of the runtime overhead imposed by current versions of ML.

Generalized algebraic data types are available today in version 6.4 of the Glasgow Haskell compiler [13,24]. We are studying their introduction into the Objective Caml compiler, and have used a separate prototype implementation of ML with generalized algebraic data types [15,19] to check that our prototype parser generator [18] indeed produces well-typed parsers.

Our result is interesting on several grounds. First, it is original and can be used to modify `ML-Yacc`, `ocamlyacc`, or `happy` so that they produce well-

[3] In the following, "ML" collectively refers to Standard ML, Objective Caml, Haskell, or any other programming language whose type system follows Hindley and Milner's discipline. Indeed, because we are interested in type-theoretic issues, the differences between these programming languages are irrelevant. We provide code fragments in a somewhat Objective Caml-like syntax.

typed, efficient parsers, with no need for unsafe type casts or pieces of C code. This allows moving the parser generator out of the *trusted computing base*, that is, of the software that must be trusted to be correct for the final executable program to be safe. Of course, the compiler that is used to turn the generated parser into executable code remains part of the trusted computing base, unless other techniques, such as type-preserving compilation or certification of the compiler itself, are applied.

Second, and perhaps more importantly, we believe that this is a *representative* application of generalized algebraic data types. In short, implementing a pushdown automaton in terms of ordinary algebraic data types requires the tags that describe the structure of the automaton's stack to be *physically* part of the stack, which is redundant, because this information is already encoded in the automaton's state. Generalized algebraic data types, on the other hand, allow the typechecker to accept the fact that the tags that describe the shape of a data structure can be stored *outside* of this data structure, in a physically separate place. In other words, our work exploits, and emphasizes, the fact that generalized algebraic data types provide a simple and elegant solution to the problem of *coordinating data structures*, that is, the problem of expressing and exploiting the existence of *correlations* between physically separate data structures. This fact has been noted by Ringenburg and Grossman [20], who apparently consider it an accidental feature of generalized algebraic data types. We believe, on the contrary, that this is intrinsically what generalized algebraic data types are about.

Although we have tested our ideas by writing a prototype parser generator [18] and a prototype typechecker [15,19], we do not report any performance figures. Indeed, our focus is on safety at least as much as on performance. Furthermore, we cannot meaningfully measure our prototype parser generator against `ocamlyacc` until the Objective Caml compiler supports generalized algebraic data types.

The paper is laid out as follows. We first introduce a sample grammar, which forms our running example, and a pushdown automaton that recognizes its language (§2 and §3). We present a straightforward ML implementation of the automaton (§4) and discuss its flaws. Then, we take a closer look at the automaton's invariant (§5), and explain how to take advantage of it using generalized algebraic data types (§6 and §7). Last, we suggest a few optimizations (§8) and conclude.

2 A sample grammar

Throughout the paper, we focus on a simple grammar for arithmetic expressions [1] whose definition appears in Figure 1. We construct a parser for this specific grammar, instead of building a more versatile parser generator, because this simple example is sufficient to convey our ideas.

The grammar's *terminal symbols*, or *tokens*, are **int**, `+`, `*`, `(`, and `)`. We

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

Fig. 1. A simple grammar with semantic actions

assume that the underlying lexical analyzer associates semantic values of type *int* with the token **int**, and semantic values of type *unit* with the tokens `+`, `*`, `(`, and `)`. The grammar's *nonterminal symbols* are E, T, and F, which respectively stand for *expression*, *term*, and *factor*. The grammar's *start symbol* is E.

There are six productions, numbered (1) to (6). Roughly speaking, each production is of the form $S_1 \ldots S_n \to S$, where $S_1, \ldots, S_n$ are (terminal or nonterminal) symbols and S is a nonterminal symbol. However, we are interested not only in determining whether some input string belongs to the language defined by this grammar, but also in exploiting this fact to convert the input string into a new form, called a *semantic value*. Thus, each production is decorated with a *semantic action*, that is, an ML expression, which specifies how to compute a semantic value. More precisely, every S_i must be followed by a distinct variable x_i, while S must be followed by an ML expression e. The variables x_i and the expression e are surrounded with braces. We allow S_i alone as syntactic sugar for $S_i\{x_i\}$, where x_i does not occur elsewhere in the production. The variables $x_1, \ldots, x_n$ are bound within e.

Often, semantic values are abstract syntax trees. Here, for the sake of simplicity, we prefer to associate semantic values of type *int* with the symbols E, T, and F. As a result, the decorated grammar in Figure 1 specifies a simple evaluator for arithmetic expressions. For instance, its first production specifies that an expression E that evaluates to x, followed by the token `+`, followed by a term T that evaluates to y, together form an expression E that evaluates to $x + y$.

3 An LR parser for the sample grammar

We now describe an LR parser for the sample grammar. This parser, also taken from Aho *et al.* [1], is presented as a finite deterministic pushdown automaton. The automaton consumes an *input stream* consisting of the tokens **int**, `+`, `*`, `(`, and `)`, and of the pseudo-token `$`, which signals the end of the stream. It maintains a current *state*. States are integers in the range $\{0, \ldots, 11\}$. It also maintains a current *stack*. Stacks are of the form $\sigma ::= \epsilon \mid \sigma s v$, where

STATE	*action*						*goto*		
	int	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				**acc**			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Fig. 2. Analysis tables for the expression grammar

s ranges over states and v ranges over semantic values. ϵ denotes the empty stack, while $\sigma s v$ denotes the stack obtained by pushing s and v on top of the stack σ.

Initially, the input stream consists of the tokens that must be parsed, followed by the pseudo-token $; the current state is 0; and the stack is empty.

The automaton's transitions are defined by two tables, *action* and *goto*, which appear in Figure 2. At every step, the automaton consults the current state, as well as the current input token, that is, the first token in the current input stream. Together, they determine an entry in the two-dimensional *action* table, which is interpreted as follows.

(i) If the entry reads "shift s" (written "ss" in Figure 2), where s is a state, then the current state and the current input token's semantic value are pushed onto the stack; one input token is discarded; and s becomes the current state.

(ii) If the entry reads "reduce k" ("rk" in Figure 2), where the grammar's k-th production is $S_1\{x_1\} \ldots S_n\{x_n\} \to S\{e\}$, then the current stack *must* be of the form $\sigma s_1 v_1 \ldots s_n v_n$. The ML expression $e[v_1/x_1, \ldots, v_n/x_n]$ is evaluated, which *must* succeed and yield a new semantic value v. Together, the state s_1 and the nonterminal symbol S determine an entry in the two-dimensional *goto* table, which *must* contain a state s. Then, $\sigma s_1 v$ becomes the current stack, and s becomes the current state. No input token is discarded.

(iii) If the entry reads "accept" ("**acc**" in Figure 2), then the current stack *must* be of the form $\sigma s v$. The automaton successfully stops and returns v.

(iv) If the entry is undefined, then the automaton stops and reports that the input string does not conform to the grammar.

How the *action* and *goto* tables are constructed is irrelevant here. A key point is that these tables are set up in such a way that the conditions which we expect *must* hold in items ii and iii above *do* indeed always hold, so that *no runtime checks are required*. The key difficulty that we are now about to confront is to implement a *typed* parser that does *not* perform these superfluous runtime checks.

4 A simple ML implementation

We now describe a simple ML implementation of the automaton, which is typed, but *does* perform the superfluous runtime checks mentioned above.

To begin, we must specify how the parser interacts with the lexical analyzer, which encapsulates the input stream. We let tokens consist of a tag and an optional semantic value:

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

We assume that the lexical analyzer provides two functions that allow retrieving and discarding the current input token:

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

This interface remains unchanged throughout the paper.

We now attack the design of the parser itself. To begin, let us define the type of states as an enumeration, that is, an algebraic data type whose data constructors are nullary:

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

Next, we must come up with a type definition for stacks. A natural first approach would be to mirror the formal definition of stacks ($\sigma ::= \epsilon \mid \sigma s v$, see §3) as an algebraic data type definition:

type *stack* =
  SEmpty | *SCons* of *stack* × *state* × *semantic_value*
and *semantic_value* = ...

This declaration states that a stack is a list of pairs of a state and a semantic value. This sounds good, but how should we define the type of semantic values? Semantic values associated with distinct symbols may have distinct types. Here, for instance, the symbols `+`, `*`, `(`, `)`, and `$` have semantic values of type *unit*, while **int**, E, T, and F have semantic values of type *int*. Therefore, it seems that we should define the type *semantic_value* as a tagged union, that is, as *another* algebraic data type. But doing so would introduce a redundancy, as both stack cells and semantic values would carry tags.

To avoid this redundancy, we follow a slightly more elaborate approach. We merge the proposed definitions for *stack* and *semantic_value* into a single definition, so that only stack cells are tagged:

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

(In the names chosen for the data constructors, P, S, L, R, and I are short for *Plus*, *Star*, *Left*, *Right*, and *Int*.) By examining the tag carried by a value of type *stack*, we can now tell not only whether it represents an empty or nonempty stack, but also, in the latter case, what symbol its top stack cell is associated with. This, in turn, allows us to tell what type of semantic value that cell contains. As a slight optimization, we choose not to represent semantic values of type *unit*. Thus, the stack cells associated with the symbols `+`, `*`, `(`, and `)` contain only a state, as opposed to a pair of a state and the unit semantic value. No stack cells are ever associated with the token `$`, because, by construction, the automaton never takes a shift transition upon encountering this token.

To sum up, every value of type *stack* carries a tag, which must be examined before the actual contents of the stack can be accessed. If, thanks to external reasoning, the tag is known beforehand, then this dynamic check is redundant. This approach, where stacks and/or semantic values are tagged, is adopted by `ML-Yacc` and by `happy` (without the `-c` flag).

The parser's central function, *run*, implements the pushdown automaton. It is parameterized by the automaton's current state and stack. It may terminate either by raising the exception *SyntaxError*, which means that the input stream does not conform to the grammar, or by returning an integer semantic value for the arithmetic expression E that was parsed. The side-effecting functions *peek* and *discard* are used to manipulate the input stream, but the code is otherwise purely functional. This turns out to be important in §6 and §7, where the types of the current state and stack evolve over time.

The definition of *run* appears in Figure 3. The function examines the current state s as well as the current input token *peek()* (line 5), and determines which action should be taken. There are many cases, two of which are shown.

When the current state is 9 and the next input token is `*` (line 7), the

```
1  exception SyntaxError
2
3  let rec run : state → stack → int =
4    fun s stack →
5      match s, peek() with
6      | ...
7      | S9, KStar →
8          (* Shift to state 7. *)
9          discard ();
10         run S7 (SS (stack, S9))
11     | S9, KPlus →
12         (* Reduce E{x} + T{y} → E{x + y}. *)
13         (* Pop three stack cells . *)
14         let ST (SP (SE (stack, s, x), _), _, y) =
15           stack in
16         (* Push the current state and the new
17            semantic value onto the stack. *)
18         let stack = SE (stack, s, x + y) in
19         (* Choose a successor state based on
20            the column E in the goto table. *)
21         gotoE s stack
22     | ...
23     | _, _ →
24         raise SyntaxError
25
26 and gotoE : state → stack → int =
27   fun s →
28     match s with
29     | S0 →
30         run S1
31     | S4 →
32         run S8
```

Fig. 3. A simple ML implementation

action table in Figure 2 specifies that the automaton should shift to state 7. Thus, the token `*` is discarded (line 9); state 9 is pushed onto the stack; and the current state is changed to 7 (line 10). In this particular case, no semantic value is pushed onto the stack. Indeed, no semantic value is associated with the token `*`, and, accordingly, the data constructor *SS* does not expect a third argument. The current state and stack are changed, in a purely functional style, by supplying appropriate parameters to the tail recursive call to *run*.

When the current state is 9 and the current input token is `+` (line 11), the *action* table specifies that the automaton should reduce production 1, that is, $E\{x\}$ + $T\{y\} \rightarrow E\{x + y\}$. As previously explained, the structure of the topmost three elements of the stack is known at that point. The first stack cell must carry the tag *ST* and contain a semantic value *y* associated with a term T. The second cell must be associated with the token `+`, that is, it must carry the tag *SP*. The third cell must carry the tag *SE* and contain a semantic value *x* associated with an expression E. Via pattern matching, *x* and *y* are extracted out of the stack (lines 14–15). At the same time, the variables *stack* and *s* are bound to new values, masking their previous values; this amounts to popping three stack cells. The semantic action *x+y* is evaluated, producing a new semantic value, and a new stack cell, holding *s* and the new semantic value, is allocated (line 18). Last, the auxiliary function *gotoE* is invoked (line 21). (There is one such auxiliary function per nonterminal symbol. The functions *gotoT* and *gotoF* are not shown.)

The job of *gotoE* (line 26) is to determine the automaton's new state after reducing a production whose head is the nonterminal symbol E, such as production 1. This is done by looking up the *goto* table at column E and row *s*. This column only has two entries (see Figure 2), so *s* must be one of *S0* and *S4*. If the former, then the new state is *S1* (line 30), otherwise it is *S8* (line 32). The current state is again changed by supplying an appropriate parameter to *run*.

Note that *gotoE* does not consult or modify the stack—in fact, it does not have access to it. One could also write *gotoE* as a function of type *state* → *state*, which simply returns *S1* or *S8*, and invoke *run* outside *gotoE* by replacing line 21 with *run (gotoE s) stack*. However, doing so would break the property that *run* is always applied to a constant state, which we exploit later on (§8).

This ML implementation appears reasonable, but performs a number of superfluous runtime checks. Indeed, the construct "let *ST (...) = stack*" (lines 14–15) dynamically checks whether the stack contains at least three cells, and whether these stack cells are associated with the symbols T, `+`, and E, as expected. Furthermore, the construct "match *s* with" (line 28) checks whether *s* is indeed one of *S0* and *S4*. Both pattern matching constructs are *nonexhaustive*, so the compiler must generate code that raises an exception when they fail. This is a waste of time and code: indeed, assuming that the parser generator is correct, then, by design of the automaton, these checks cannot fail. This overhead is present in parsers produced by `ML-Yacc` and by `happy` (without the `-c` flag). Parsers produced by `happy -gc` or by `ocamlyacc` avoid some or all of it, but are untyped—they involve unsafe type casts and, in the case of `ocamlyacc`, a piece of C code. The point of this paper is to show how an advanced type system allows eliminating these overheads while guaranteeing safety.

Before carrying on, let us end this section with a couple of remarks.

First, in this implementation, the *action* and *goto* tables are compiled into *code*, as in parsers produced by `happy` (without the `-a` flag), as opposed to encoded as *data* (say, as arrays of integers) and interpreted by code, as

Stack shape							State
ϵ							0
ϵ	$\{0\}$	E					1
σ	$\{0,4\}$	T					2
σ	$\{0,4,6\}$	F					3
σ	$\{0,4,6,7\}$	`(`					4
σ	$\{0,4,6,7\}$	**int**					5
σ	$\{0,4\}$	E	$\{1,8\}$	`+`			6
σ	$\{0,4,6\}$	T	$\{2,9\}$	`*`			7
σ	$\{0,4,6,7\}$	`(`	$\{4\}$	E			8
σ	$\{0,4\}$	E	$\{1,8\}$	`+`	$\{6\}$	T	9
σ	$\{0,4,6\}$	T	$\{2,9\}$	`*`	$\{7\}$	F	10
σ	$\{0,4,6,7\}$	`(`	$\{4\}$	E	$\{8\}$	`)`	11

Fig. 4. The automaton's invariant

in parsers produced by `happy -a`, `ML-Yacc`, or `ocamlyacc`. This approach, studied in a number of earlier works [11,5,2], has the disadvantage of leading to greater code size. Its key advantage, as far as we are concerned, is to make the code more amenable to analysis by a general-purpose type system. A secondary advantage is to remove the interpretation overhead and to enable extra optimizations based on code specialization (§8).

Second, we represent the stack as a purely functional data structure, that is, a linked list of immutable, heap-allocated cells. This is somewhat inefficient, since a pair of mutable, extensible arrays would do—one for states, one for semantic values. However, an array of semantic values would form a *mutable, heterogeneous* data structure, whose entries can change type over time. ML's type system does not support such data structures. At the very least, a notion of *linearity* would be required in order to guarantee that no pointers to deallocated stack cells are kept around and dereferenced. Thus, our choice of an immutable data structure is imposed by our somewhat naïve type discipline. Designing type systems that support mutable stacks is an active area of research; see, for example, Jia *et al.*'s recent work [6].

5 Understanding the automaton's invariant

We asserted earlier that, by design of the *action* and *goto* tables, when a "reduce" action is taken, the contents of the top few stack cells are known and may be accessed without a dynamic check. Before modifying the code to take advantage of this fact, we must understand why this is so.

The reason is simple. Although stacks were defined as arbitrary sequences of pairs of a state and a semantic value, the stacks that do arise at runtime are not arbitrary: they range over a strict subset of that space, which is given in Figure 4. The left-hand column specifies the shape of a stack σ, while the right-hand column specifies a state s. In the left-hand column, ϵ represents the empty stack, while σ represents an unknown stack. Integers denote states, so sets of integers denote sets of possible states. A terminal or nonterminal symbol denotes a semantic value associated with that symbol. For instance, the table's sixth line states that, when the automaton is in state 5, the stack is nonempty, and its top cell holds a state in the subset $\{0, 4, 6, 7\}$ and a semantic value for the token **int**. This implies, in particular, that the top stack cell carries the tag *SI*, so the information contained in this tag is redundant.

Definition A stack σ and a state s are *consistent* if and only if (i) σ and s match one of the shapes in Figure 4 and (ii) if σ is of the form $\sigma' s' v'$, then σ' and s' are consistent. This is an inductive definition.

Then, the following invariant holds:

Invariant *At every time, the automaton's current stack σ and current state s are consistent.*

The proof is by induction over runs of the automaton, as defined in §3. The automaton's initial stack and state are ϵ and 0, which are consistent because they appear in the first line of Figure 4. There remains to prove that every possible transition preserves this invariant. We omit the proof, which is straightforward. We will in fact go through a few proof cases when explaining why the modified parser in §6 is well-typed.

Here, it looks as if we discovered the invariant *after* building the automaton, through a careful (and perhaps painful) analysis of its transitions. In fact, when an automaton is produced, out of an arbitrary grammar, by a parser generator, the automaton's invariant is very easily constructed at the same time, so no "invariant discovery" phase is needed.

In §6 and §7, we explain how to make the type system aware of this invariant and how this allows getting rid of the superfluous runtime checks. We proceed in two steps. In §6, we only exploit knowledge about the height of the stack and the type of the semantic values that it contains. Then, in §7, we show how to also exploit knowledge about the identity of the states contained in the stack.

6 Keeping track of the stack's structure

Thanks to the automaton's invariant, examining the current state is enough to acquire some information about the structure of the stack. For instance, if the current state is 9, then the stack ends with three cells tagged *SE*, *SP*, and *ST*. This is exactly the information needed to prove that the construct "let *ST (SP (SE (...))) = stack*" (Figure 3) cannot fail. But how can we persuade the compiler of such a fact? We must make it aware of the correlation between states and stack shapes.

To do so, we parameterize the type of states with a type variable α. The idea is, *if the current state has type α state, then the current stack has type α*. The runtime representation of states does not change, so α does not describe the *structure* of states, and could be referred to as a *phantom* type parameter [4].

In the following, we first assume that the type *state* is given a definition that satisfies this intuition, and explain how this allows altering our view of the stack (§6.1). Then, we actually define *state* as a generalized algebraic data type (§6.2). Last, we discuss how these changes in our type definitions affect the code for the automaton (§6.3).

6.1 Types for stack cells

In order to convince the typechecker that accesses to the stack cannot fail, we must no longer define the type *stack* as a tagged union. Instead, our vocabulary must be sufficiently precise to express, say, "the type of all stacks that end with three cells associated with the symbols E, +, and T." For this reason, we define not a single type *stack*, but a *family* of types (Figure 5, lines 2–10).

The type *empty* (Figure 5, line 2) is the type of the empty stack. Its only value is *SEmpty*, so it is isomorphic to *unit*.

The type α *cP* (Figure 5, line 3) is the type of a nonempty stack whose top cell is associated with symbol + and whose remainder has type α. A key point is that α occurs twice in this definition, once as the type of the remainder of the stack and once as the parameter to *state*. This encodes item (ii) in the definition of consistency (§5) and tells the type system that, in every stack cell, there is a relationship between the state that is held in the cell and the structure of the remainder of the stack. The definitions on lines 4–10 are analogous.

Thus, a stack that consists of a single cell associated with the symbol E, has type *empty cE*. A stack that consists of two cells, respectively associated with the symbols E and +, has type *empty cE cP*; and so on. It might seem that we now need types of unbounded size in order to describe all possible stacks. Fortunately, we are happy with incomplete information about stacks. For instance, every nonempty stack whose top cell is associated with the symbol E must have a type of the form τ *cE* for some type τ. As a result, every such stack is a valid argument to a function of type $\forall\alpha.\alpha\ cE \rightarrow \ldots$. In other words, thanks to type variables and type abstraction, a single type can describe an infinite collection of stacks.

Although, for syntactic convenience, we have kept the tags *SEmpty*, *SP*, etc., none of the types defined in lines 2–10 of Figure 5 is a tagged union. In fact, they are *tuple* types with zero (*empty*), two (*cP*, etc.), or three (*cI*, etc.) components. In other words, according to these new type definitions,

```
1  (* Types for stack cells . *)
2  type empty = SEmpty
3  type α cP = SP of α × α state
4  type α cS = SS of α × α state
5  type α cL = SL of α × α state
6  type α cR = SR of α × α state
7  type α cI = SI of α × α state × int
8  type α cE = SE of α × α state × int
9  type α cT = ST of α × α state × int
10 type α cF = SF of α × α state × int
11
12 (* The type of states . *)
13 type state : ⋆ → ⋆ where
14 | S0 : empty state
15 | S1 : empty cE state
16 | S2 : ∀α.α cT state
17 | S3 : ∀α.α cF state
18 | S4 : ∀α.α cL state
19 | S5 : ∀α.α cI state
20 | S6 : ∀α.α cE cP state
21 | S7 : ∀α.α cT cS state
22 | S8 : ∀α.α cL cE state
23 | S9 : ∀α.α cE cP cT state
24 | S10 : ∀α.α cT cS cF state
25 | S11 : ∀α.α cL cE cR state
```

Fig. 5. Encoding part of the invariant into types

stack cells are no longer tagged. Stacks are still linked sequences of cells, just like standard linked lists. Each cell can be a tuple of zero, two, or three components, yet *no tag* is stored inside the cell to distinguish between these cases. Instead, *the automaton's current state* will be used, when necessary, to predict the shape of the top stack cells. Thus, the automaton's state and stack are now *coordinated data structures* in the sense of Ringenburg and Grossman [20].

6.2 Types for states

We now come to the definition of the parameterized type α *state* (Figure 5, lines 13–25). The definition is in pseudo-Objective Caml syntax, since Objective Caml does not yet offer generalized algebraic data types. Line 13 states that *state* has kind $\star \to \star$, that is, *state* is now a unary algebraic data type constructor. Lines 14–25 specify its data constructors, all of which remain nullary, together with their type scheme.

The novelty lies in the way the new type parameter is constrained so as to reflect knowledge about the structure of the stack. Consider state 0. According to Figure 4, when the automaton is in state 0, the stack is empty. Thus, we want *S0* to have type *empty state* (line 14). Here, the type parameter is constrained to be *empty*. Similarly, when the automaton is in state 1, the stack consists of a single cell, containing a semantic value for symbol E. Thus, we want *S1* to have type *empty cE state* (line 15). In the case of state 2, matters are slightly more complex: according to Figure 4, when the automaton is in state 2, the stack ends with a cell associated with symbol T, but the remainder of the stack is unknown. Thus, we let *S2* have type $\forall\alpha.\alpha\ cT\ state$ (line 16). The type variable α, which represents the remainder of the stack, is universally quantified, so that *every* value of α is compatible with state *S2*. As a result, determining that the automaton's current state is *S2* only allows concluding that the current stack has type $\tau\ cT$ for *some* type τ. The declarations for *S3* to *S11* (lines 17–25) are obtained in a similar manner. This algebraic data type declaration encodes item (i) in the definition of consistency (§5).

The definitions of *cP*, *cS*, etc. and of *state* are mutually recursive. *state* is a *generalized* algebraic data type constructor [26]. Indeed, if it were an ordinary one, then each of *S0*, *S1*, etc. would necessarily be assigned type $\forall\alpha.\alpha\ state$, preventing us from encoding the automaton's invariant. In other words, the key opportunity offered by generalized algebraic data types is to allow intentionally assigning more specific types to states. This pays off when doing case analysis over a state, as we are now about to explain.

6.3 Implementation

Let us now study the new definition of *run*, which appears in Figure 6. We have arranged everything so that *only the type annotations carried by* run *and gotoE change*; the program itself is identical.

The type of *run* changes from *state* $\rightarrow$ *stack* $\rightarrow$ *int* to $\forall\alpha.\alpha\ state \rightarrow \alpha \rightarrow int$ (line 1). In other words, the structure of the stack, represented by α, may be arbitrary, provided it is consistent with the current state. Letting the type of the state and the type of the stack *share* a type variable α allows us to reflect the correlation (some may say the *dependency*) between the current state and the current stack.

By hypothesis, s has type α *state*. Then, thanks to our new definition of *state* as a generalized algebraic data type constructor, examining s (line 3) yields information about α. For instance, let us see what happens when s is matched against *S9*. By definition of the type *state* (Figure 5, line 23), *S9* has all types of the form $\tau'\ cE\ cP\ cT\ state$, and only those. As a result, if s is found to be equal to *S9*, then the type denoted by α must be of the form $\tau'\ cE\ cP\ cT$, for some type τ'. In other words, it is safe to enrich the typechecking context with the equation

$$\alpha = \alpha'\ cE\ cP\ cT, \qquad (\mathbf{1})$$

```
1  let rec run : ∀α.α state → α → int =
2    fun s stack →
3      match s, peek() with
4      | ...
5      | S9, KStar →
6          discard ();
7          run S7 (SS (stack, S9))
8      | S9, KPlus →
9          let ST (SP (SE (stack, s, x), _), _, y) =
10           stack in
11         let stack = SE (stack, s, x + y) in
12         gotoE s stack
13     | ...
14     | _, _ →
15         raise SyntaxError
16
17 and gotoE : ∀α.α state → α cE → int =
18   fun s →
19     match s with
20     | S0 →
21         run S1
22     | S4 →
23         run S8
```

Fig. 6. A refined implementation

where α' is a fresh type variable. Typecheckers that support generalized algebraic data types are able to make such deductions [26].

Shift transitions Let us now check why the code for a "shift" transition (lines 6–7) is well-typed. Because *stack* and *s* have types α and α *state*, the cell *SS (stack, S9)* on line 7 has type α *cS*.

Now, let τ stand for α' *cE cP*, where α' is the abstract type variable introduced in the previous paragraph. Then, equation (1) can be written $\alpha = \tau\ cT$, which, by congruence, implies

$$\alpha\ cS = \tau\ cT\ cS.$$

This lets the typechecker deduce that the stack cell *SS (stack, S9)*, which is known to have type α *cS*, also has type τ *cT cS*.

Finally, instantiating α to τ in the definition of *S7* (Figure 5, line 21) shows that τ *cT cS state* is a valid type for *S7*. Then, instantiating α to τ *cT cS* in the type of *run* shows that the call *run S7 (SS (stack, S9))* on line 7 is well-typed.

In short, we have checked that the top two stack cells exist and are associated with the symbols T and `*`. This is sufficient to guarantee that the new stack is consistent with state 7. The fact that α was instantiated to τ in the definition of *S7*, where τ stands for $\alpha'\ cE\ cP$, means that, when performing this “shift” transition, the automaton *forgets* about the existence of the next two stack cells, which are associated with the symbols E and `+`.

Reduce transitions Let us now check a “reduce” transition (lines 9–12). The variable *stack* has type α, so, by equation (1), also has type $\alpha'\ cE\ cP\ cT$. Therefore, it is legal to match *stack* against the pattern *ST (SP (SE (stack, s, x), _), _, y)*, and this binds *stack*, *s*, *x*, and *y* to values of types α', α' *state*, *int*, and *int*, respectively. Furthermore, this pattern matching construct *cannot fail*, since only tuple patterns are involved. Since *x* and *y* both have type *int*, *x+y* (line 11) is well-typed, and the new *stack* (line 11) has type $\alpha'\ cE$. Thus, the call to *gotoE* (line 12) is valid.

Goto functions The task of *gotoE* (lines 17–23) is to *recover* the information that was lost during “shift” transitions. When *gotoE* is called, it is known that the top cell of the stack is associated with symbol E. Indeed, we are in the process of reducing a production whose head symbol is E, so we just created that cell, inside *run*, before invoking *gotoE*. Yet, nothing more is known about the stack. To recover information about the remainder of the stack, we must examine the state that is held inside its top cell. This state is passed to *gotoE* under the name *s* (line 18).

Imagine *s* is *S4* (line 22). According to the type ascribed to *gotoE* (line 17), *s* has type α *state*. Matching this information against the definition of *S4* (Figure 5, line 18), we find that it is safe to enrich the typechecking context with the equation

$$\alpha = \alpha'\ cL, \tag{2}$$

where α' is a fresh type variable. Thus, we recover the information that the next cell down in the stack exists and is associated with the symbol `(`. Here, the success of a *dynamic* test, namely the case analysis on *s*, yields *static* information about the shape of the stack. This feature is characteristic of generalized algebraic data types.

Finally, by instantiating α to α' in the types of *run* and *S8*, we find that *run S8* has type $\alpha'\ cL\ cE \rightarrow int$, which, thanks to equation (2), can be written $\alpha\ cE \rightarrow int$. Thus, the value returned on line 23 satisfies the type ascribed to *gotoE* (line 17).

The case where *s* is *S0* (lines 20–21) is similar.

6.4 Summary and remarks

We have encoded part of the invariant in Figure 4 into the type definitions of Figure 5. This allows us to remove the tags carried by stack cells, yielding better efficiency and, more importantly, a stronger correctness guarantee. The code for "reduce" actions inside *run* now performs *no* runtime check. Yet, the parser is well-typed.

One could be puzzled by our claim that a runtime check has been eliminated, since the source code hasn't changed, except in type annotations. The point is that, thanks to the new type information, a formerly nonexhaustive pattern matching construct has become exhaustive. This allows the compiler to produce better machine code, without a runtime check, out of the same source code.

The function *gotoE* still performs a *nonexhaustive* case analysis (line 19), which translates down to a dynamic check. In other words, the typechecker has no way of proving that *s* must be either *S0* or *S4*. As a result, the compiler must emit a compile-time warning and generate code that causes a runtime failure in the event that *s* is some other state. So, although the parser cannot crash, it can in principle still fail unexpectedly. We attack this issue in §7.

The code on line 9 of Figure 6 must be able to access the third stack cell, which holds *stack*, *s*, and *x*, without examining the states stored in the top two stack cells, which are here discarded using wildcard patterns _. This requirement appears to preclude a representation of stacks as ordinary algebraic data types where the state held in each cell serves as a tag that *must* be examined before the remainder of the stack can be accessed.

7 Keeping track of states inside the stack

In order to eliminate the dynamic check that remains inside *gotoE*, it is necessary to prove that the parameter *s* must be either *S0* or *S4*. Since *s* is originally found on the stack during a "reduce" transition, we must keep track of the *identity* of the states found inside the stack.

To do so, we add a new parameter, ρ, to the type constructor *state*. Informally speaking, the idea is to set things up so that ρ ranges over sets of states, and so that a state has type (τ, ρ) *state* only if it is a member of the set ρ. For instance, if some state has type $(\alpha, \{0, 4\})$ *state*, then it should be one of *S0* and *S4*.

Technically speaking, however, sets of integers are not types (at least, not in ML), so ρ cannot range over such sets. Instead, we must encode sets of states into types. We first discuss two ways of doing so (§7.1). Then, we explain how to define the new binary *state* type constructor (§7.2), and how the code for the automaton is affected (§7.3).

7.1 Encoding sets into types

Let *pre* and *abs*, standing for *present* and *absent*, be two distinct abstract types. (They can be defined as algebraic data types with no data constructors.) Then, sets of states can be encoded as 12-tuple types whose components are *pre*, *abs*, or type variables.

For instance, the *constant* set $\{0, 4\}$ can be encoded as the type $pre \times abs \times abs \times abs \times pre \times abs \times abs \times abs \times abs \times abs \times abs \times abs$. For the sake of conciseness, we write $\{0, 4\}$ for this type. More generally, if S is an arbitrary set of states, we write $\{S\}$ for the product type whose i-th component is pre (resp. abs) if and only if $i \in S$ (resp. $i \notin S$) holds.

An arbitrary *subset* of $\{0, 4\}$ can be encoded as the type $\gamma_0 \times abs \times abs \times abs \times \gamma_4 \times abs \times abs \times abs \times abs \times abs \times abs \times abs$, where γ_0 and γ_4 are fresh type variables. We write $\langle 0, 4\rangle$ for such a type. More generally, if S is an arbitrary set of states, we write $\langle S\rangle$ for the product type whose i-th component is a fresh type variable γ_i (resp. abs) if and only if $i \in S$ (resp. $i \notin S$) holds. This notation is concise, but informal, since it does not specify how the names γ_i are chosen. We view this as tolerable for the purposes of this exposition.

Two key properties are that, if S and S' are sets of states, then (i) the type $\{S\}$ is an instance of the type $\langle S'\rangle$ if and only if the subset relationship $S \subseteq S'$ holds and (ii) similarly, the type $\langle S\rangle$ is an instance of the type $\langle S'\rangle$ if and only if the subset relationship $S \subseteq S'$ holds. Thus, we are able to encode subset relationships in ML's type system, even though it is based on unification and lacks a notion of subtyping. This trick was inspired to us by Rémy's treatment of records [16]. It was independently discovered and studied by Fluet and Pucella [3].

This encoding works, but is extremely verbose when the automaton has many states. If the type system has rows [16], another, more economical encoding is available. Objective Caml, for instance, has rows, which it uses to form object types [17]. Our prototype implementation of ML with generalized algebraic data types [19] also has full support for rows.

The idea is to encode sets of states as rows whose labels are states and whose components are pre, abs, or type variables. We write $\{0, 4\}$ for the row $(0 : pre; 4 : pre; \partial abs)$, which maps 0 and 4 to pre and all other states to abs. We write $\langle 0, 4\rangle$ for the row $(0 : \gamma_0; 4 : \gamma_4; \partial abs)$. As usual, row equality is defined modulo the following commutation and expansion laws:

$$(s_1 : \tau_1; s_2 : \tau_2; \tau) = (s_2 : \tau_2; s_1 : \tau_1; \tau)$$
$$(s : \tau; \partial\tau) = \partial\tau$$

As a result, this alternate encoding also satisfies properties (i) and (ii) above. Of course, these laws must be taken into account by the typechecker when deciding whether one set of equations entails another—a check that becomes necessary in the presence of generalized algebraic data types.

One should acknowledge that neither of these two encodings is extremely natural. One might even argue that their very existence is somewhat accidental. Indeed, Hindley and Milner's type system was certainly not designed to allow reasoning about subset relationships. Instead, it would be worth designing a new type system that facilitates this kind of reasoning. Nevertheless, since these encodings do exist, let us exploit them. In the following, we assume

1 type $\mathit{empty} = \mathit{SEmpty}$
2 type $(\alpha, \rho)\ \mathit{cP} = \mathit{SP}$ of $\alpha \times (\alpha, \rho)\ \mathit{state}$
3 type $(\alpha, \rho)\ \mathit{cS} = \mathit{SS}$ of $\alpha \times (\alpha, \rho)\ \mathit{state}$
4 type $(\alpha, \rho)\ \mathit{cL} = \mathit{SL}$ of $\alpha \times (\alpha, \rho)\ \mathit{state}$
5 type $(\alpha, \rho)\ \mathit{cR} = \mathit{SR}$ of $\alpha \times (\alpha, \rho)\ \mathit{state}$
6 type $(\alpha, \rho)\ \mathit{cI} = \mathit{SI}$ of $\alpha \times (\alpha, \rho)\ \mathit{state} \times \mathit{int}$
7 type $(\alpha, \rho)\ \mathit{cE} = \mathit{SE}$ of $\alpha \times (\alpha, \rho)\ \mathit{state} \times \mathit{int}$
8 type $(\alpha, \rho)\ \mathit{cT} = \mathit{ST}$ of $\alpha \times (\alpha, \rho)\ \mathit{state} \times \mathit{int}$
9 type $(\alpha, \rho)\ \mathit{cF} = \mathit{SF}$ of $\alpha \times (\alpha, \rho)\ \mathit{state} \times \mathit{int}$
10
11 type $\mathit{state} : (\star, \mathit{row}) \to \star$ where
12 $\mid \mathit{S0} : (\mathit{empty}, \{0\})\ \mathit{state}$
13 $\mid \mathit{S1} : \forall\bar{\gamma}.((\mathit{empty}, \langle 0 \rangle)\ \mathit{cE}, \{1\})\ \mathit{state}$
14 $\mid \mathit{S2} : \forall\alpha\bar{\gamma}.((\alpha, \langle 0, 4 \rangle)\ \mathit{cT}, \{2\})\ \mathit{state}$
15 $\mid \mathit{S3} : \forall\alpha\bar{\gamma}.((\alpha, \langle 0, 4, 6 \rangle)\ \mathit{cF}, \{3\})\ \mathit{state}$
16 $\mid \mathit{S4} : \forall\alpha\bar{\gamma}.((\alpha, \langle 0, 4, 6, 7 \rangle)\ \mathit{cL}, \{4\})\ \mathit{state}$
17 $\mid \mathit{S5} : \forall\alpha\bar{\gamma}.((\alpha, \langle 0, 4, 6, 7 \rangle)\ \mathit{cI}, \{5\})\ \mathit{state}$
18 $\mid \mathit{S6} : \forall\alpha\bar{\gamma}.(((\alpha, \langle 0, 4 \rangle)\ \mathit{cE}, \langle 1, 8 \rangle)\ \mathit{cP}, \{6\})\ \mathit{state}$
19 $\mid \mathit{S7} : \forall\alpha\bar{\gamma}.(((\alpha, \langle 0, 4, 6 \rangle)\ \mathit{cT}, \langle 2, 9 \rangle)\ \mathit{cS}, \{7\})\ \mathit{state}$
20 $\mid \mathit{S8} : \forall\alpha\bar{\gamma}.(((\alpha, \langle 0, 4, 6, 7 \rangle)\ \mathit{cL}, \langle 4 \rangle)\ \mathit{cE}, \{8\})\ \mathit{state}$
21 $\mid \mathit{S9} : \forall\alpha\bar{\gamma}.((((\alpha, \langle 0, 4 \rangle)\ \mathit{cE}, \langle 1, 8 \rangle)\ \mathit{cP}, \langle 6 \rangle)\ \mathit{cT}, \{9\})\ \mathit{state}$
22 $\mid \mathit{S10} : \forall\alpha\bar{\gamma}.((((\alpha, \langle 0, 4, 6 \rangle)\ \mathit{cT}, \langle 2, 9 \rangle)\ \mathit{cS}, \langle 7 \rangle)\ \mathit{cF}, \{10\})\ \mathit{state}$
23 $\mid \mathit{S11} : \forall\alpha\bar{\gamma}.((((\alpha, \langle 0, 4, 6, 7 \rangle)\ \mathit{cL}, \langle 4 \rangle)\ \mathit{cE}, \langle 8 \rangle)\ \mathit{cR}, \{11\})\ \mathit{state}$
24
25 val $\mathit{run} : \forall\alpha\rho.(\alpha, \rho)\ \mathit{state} \to \alpha \to \mathit{int}$
26 val $\mathit{gotoE} : \forall\alpha\bar{\gamma}.(\alpha, \rho)\ \mathit{state} \to (\alpha, \rho)\ \mathit{cE} \to \mathit{int}$
27 where $\rho = \langle 0, 4 \rangle$

Fig. 7. Encoding the entire invariant into types

that the row encoding is used, but our results are equally valid with the more naïve product encoding.

7.2 Types for states

Equipped with notation for encoding sets as types, we can now provide a new definition of the type *state* (Figure 7). There are two changes with respect to the previous definition (Figure 5).

First, in every line, the second parameter to *state* is constrained in a way that reflects the state's identity in an exact manner. For instance, *S0* is given a type of the form $(\dots, \{0\})\ \mathit{state}$ (line 12); *S1* is given a type of the form $(\dots, \{1\})\ \mathit{state}$ (line 13); and so on. As a result, a type of the form $(\tau, \{0\})\ \mathit{state}$ can be inhabited only by *S0*; a type of the form $(\tau, \{1\})\ \mathit{state}$ can be inhabited only by *S1*; and so on. This technique is related to *singleton types*.

Second, in every line, the first parameter to *state*, which reflects the structure of the stack, is modified so as to keep track of the identity of the states contained in the stack. This is done via an additional parameter to the family of cell type constructors. For every cell, an upper bound on the identity of the state that is held inside the cell is specified, using a type of the form $\langle S \rangle$. For instance, the type ascribed to *S2* is

$$\forall \alpha\bar{\gamma}.((\alpha, \langle 0, 4\rangle)\ cT, \{2\})\ \mathit{state}$$

(line 14), which reflects the fact that, whenever the current state is 2, the topmost stack cell contains a semantic value for symbol T and a state in the set $\{0, 4\}$. By convention, on every line, $\bar{\gamma}$ stands for all of the type variables implicitly introduced in types of the form $\langle S \rangle$.

7.3 Implementation

The last changes are in the types of *run* and *gotoE* (lines 25–27).

The first parameter to *run* now has type (α, ρ) *state*, instead of α *state*. The variable ρ is unconstrained, because *run* accepts an arbitrary current state.

The first parameter to *gotoE* now has type (α, ρ) *state*, where ρ is an alias for $\langle 0, 4\rangle$, that is, for $(0 : \gamma_0; 4 : \gamma_4; \partial \mathit{abs})$. We purposely use an alias, instead of simply writing $\langle 0, 4\rangle$ twice, because that would give rise to *four* fresh type variables γ_0, γ_4, γ'_0, and γ'_4, which is not what we intend. The type variables γ_0 and γ_4 are universally quantified: indeed, here, $\bar{\gamma}$ stands for $\gamma_0\gamma_4$.

Because γ_0 and γ_4 are universally quantified, they can be instantiated at will with *pre* or *abs*. As a result, the application *gotoE S0* is well-typed: indeed, by property (i), the type $\{0\}$ is an instance of the type $\langle 0, 4\rangle$. Similarly, *gotoE S4* is well-typed. However, no other state can be passed to *gotoE*. For instance, the application *gotoE S1* is ill-typed, because the type $\{1\}$ is not an instance of the type $\langle 0, 4\rangle$.

A typechecker for generalized algebraic data types can take advantage of this fact to recognize that the case analysis inside *gotoE* is exhaustive and cannot fail. Consider, for instance, the case of *S1*, which syntactically appears to be missing. If a branch for *S1* was present in *gotoE*, then it would be typechecked under the equation

$$\{1\} = \langle 0, 4\rangle,$$

that is,

$$(1 : \mathit{pre}; \partial \mathit{abs}) = (0 : \gamma_0; 4 : \gamma_4; \partial \mathit{abs}),$$

where γ_0 and γ_4 are fresh. By property (i), this equation is unsatisfiable, which proves that such a branch would be dead.

The same reasoning can be conducted for every state other than 0 and 4, which allows concluding that the case analysis really is exhaustive, even though it explicitly deals with only two cases. No compile-time warning is emitted, and no runtime check is required. This feature, referred to as *dead code elimination* by Xi [25], is also described by Simonet and Pottier [22]. Our prototype typechecker [19] implements it.

We let the reader check that the code for "shift" and "reduce" transitions remains well-typed after these changes. Property (ii) is used when typechecking "shift" transitions. Again, in moving from §6 to §7, we have modified a few type declarations and type annotations, but the code itself is unchanged. The only effect of the extra type information is to allow the compiler to better deal with the case analysis inside *gotoE*. No compile-time warning is emitted, which means that the compiler now guarantees that the program won't crash *or fail*.

All of the information in Figure 4 is now encoded in the definition of the type *state*. In fact, when the typechecker analyzes the program, it automatically verifies that the automaton's invariant holds.

8 Optimizations

We conclude with a list of optional optimizations, which our prototype implements. They are straightforward: the point is that our aggressive use of types does not get in the way.

Some of the states that are pushed onto the stack are never used. A look at the *goto* table shows that the only states that are ever consulted during a "reduce" operation are 0, 4, 6, and 7. Indeed, all other states have empty rows in the *goto* table. In other words, there is no point in pushing the states 1, 2, 8, and 9 on the stack. (The states 3, 5, 10, and 11 are never pushed on the stack anyway, because they have no outgoing "shift" transition.) So, when we perform a "shift" transition out of state 1, 2, 8, or 9, we can allocate a stack cell that does not contain a state. Horspool and Whitney [5] refer to this idea as the "minimal push" optimization.

This optimization, together with our earlier decision of not storing semantic values of type *unit* into the stack, means that some "shift" transitions require no allocation at all. Here, the "shift" transitions that leave states 1, 2, 8, and 9 are associated with tokens whose semantic values have type *unit*. When one of these transitions is taken, there is no need to modify the stack: only the current state changes.

Another optimization consists in defining one specialized version of *run* for every state: *run0*, *run1*, and so on. These specialized functions are assigned types that reflect knowledge of the shape of the stack: for instance, *run9* is assigned type $\forall \alpha.\alpha\ cE\ cP\ cT \rightarrow int$. The parameter s disappears: calls to *run S9* are replaced with calls to *run9*. This optimization is made possible by the fact that *run* is always applied to a constant state.

After these optimizations are performed, the runtime representations of all states other than 0, 4, 6, and 7 are no longer used, so the corresponding data

```
type empty = SEmpty
type (α, ρ) cL = SL of α × (α, ρ) state
type (α, ρ) cE = SE of α × (α, ρ) state × int
type (α, ρ) cT = ST of α × (α, ρ) state × int

type state : (⋆, row) → ⋆ where
| S0 : (empty, {0}) state
| S4 : ∀αγ̄.((α, ⟨0, 4, 6, 7⟩) cL, {4}) state
| S6 : ∀αγ̄.((α, ⟨0, 4⟩) cE, {6}) state
| S7 : ∀αγ̄.((α, ⟨0, 4, 6⟩) cT, {7}) state

let rec gotoT : ∀αγ̄.(α, ρ) state → (α, ρ) cT → int
where ρ = ⟨0, 4, 6⟩ =
  fun s →
    match s with
    | S0 → ...
    | S4 → ...
    | S6 →
        (* Inlined version of run9. *)
        fun stack →
          match peek() with
          | KStar →
              discard ();
              run7 stack
          | KPlus →
              let ST (SE (stack, s, x), _, y) =
                stack in
              let stack = SE (stack, s, x + y) in
              gotoE s stack
          | ...
          | _ →
              raise SyntaxError

and gotoE : ∀αγ̄.(α, ρ) state → (α, ρ) cE → int where ρ = ⟨0, 4⟩ =
  fun s →
    match s with
    | S0 →
        run1
    | S4 →
        (* Inlined version of run8. *)
        ...
```

Fig. 8. An optimized implementation

constructors need no longer be defined; only the corresponding *run* functions remain. In fact, some of these functions only have one call site, and can be eliminated altogether via inlining. This is the case of *run8*, *run9*, *run10*, and *run11*.

Our final type definitions appear in Figure 8 (lines 1–10). All data constructors but *S0*, *S4*, *S6*, and *S7* have disappeared. Furthermore, the stack shapes associated with these four states have been simplified. The cells that did not contain a semantic value and held a state in the set $\{1, 2, 8, 9\}$ have disappeared altogether.

A fragment of the final code is also shown in Figure 8 (lines 13–41). The definition of *run9* (lines 19–32) is inlined at its unique call site inside *gotoT*. The “shift” transition to state 7 (line 24) is performed by invoking *run7*. No new stack cell is allocated, because neither the state 9 nor the semantic value *()* are useful. In the “reduce” transition, only two stack cells are popped (lines 26–27), because the intermediate cell, which was associated with the symbol + and did not contain any useful information, has been eliminated. The auxiliary function *gotoE* is unchanged, except the call *run S1* is replaced with *run1* (line 38) and the call *run S8* is replaced with an inlined version of *run8* (line 40), again because *run8* has no other call sites.

Horspool and Whitney’s “direct goto” optimization [5] comes for free: when a *goto* function contains a match statement with only one branch, the compiler naturally produces code that involves no runtime check.

9 Conclusion

We have explained how ML, extended with generalized algebraic data types, is able to express *efficient* and *safe* LR parsers. Here, we understand “efficiency” as the absence of redundant dynamic checks, and “safety” as the existence of a compiler-verifiable proof that the program cannot crash or fail at runtime. This is a pleasing result as well as an illustration of the new expressiveness offered by generalized algebraic data types.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] Achyutram Bhamidipaty and Todd A. Proebsting. Very fast YACC-compatible parsers (for very little effort). *Software – Practice & Experience*, 28(2):181–190, February 1998.

[3] Matthew Fluet and Riccardo Pucella. Phantom types and subtyping. In *IFIP International Conference on Theoretical Computer Science (TCS)*, volume 223 of *IFIP Conference Proceedings*, pages 448–460. Kluwer, August 2002.

[4] Ralf Hinze. Fun with phantom types. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, pages 245–262. Palgrave Macmillan, March 2003.

[5] R. Nigel Horspool and Michael Whitney. Even faster LR parsing. *Software – Practice & Experience*, 20(6):515–535, June 1990.

[6] Limin Jia, Frances Spalding, David Walker, and Neal Glew. Certifying compilation for a language with stack allocation. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 407–416, June 2005.

[7] Stephen C. Johnson. Yacc: Yet another compiler-compiler. Computing Science Technical Report 32, Bell Laboratories, 1975.

[8] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system*, July 2004.

[9] Simon Marlow and Andy Gill. *Happy: the parser generator for Haskell*, April 2004.

[10] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML – Revised*. MIT Press, May 1997.

[11] Thomas J. Pennello. Very fast LR parsing. In *Symposium on Compiler Construction*, pages 145–151, 1986.

[12] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, April 2003.

[13] Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: type inference for generalised algebraic data types. Manuscript, July 2004.

[14] François Pottier and Nadji Gauthier. Polymorphic typed defunctionalization and concretization. *Higher-Order and Symbolic Computation*, May 2005. To appear.

[15] François Pottier and Yann Régis-Gianas. Stratified type inference for generalized algebraic data types. Submitted, July 2005.

[16] Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1994.

[17] Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1):27–50, 1998.

[18] Yann Régis-Gianas. A prototype parser generator for ML with generalized algebraic data types. `http://cristal.inria.fr/~regisgia/software/`, September 2004.

[19] Yann Régis-Gianas. A prototype typechecker for ML with generalized algebraic data types. `http://cristal.inria.fr/~regisgia/software/`, July 2005.

[20] Michael F. Ringenburg and Dan Grossman. Types for describing coordinated data structures. In *Workshop on Types in Language Design and Implementation (TLDI)*, pages 25–36, January 2005.

[21] Tim Sheard. Languages of the future. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 116–119, October 2004.

[22] Vincent Simonet and François Pottier. Constraint-based type inference for guarded algebraic data types. Research Report 5462, INRIA, January 2005.

[23] David R. Tarditi and Andrew W. Appel. *ML-Yacc User's Manual*, April 2000.

[24] The GHC team. *The Glasgow Haskell compiler*, March 2005.

[25] Hongwei Xi. Dead code elimination through dependent types. In *International Workshop on Practical Aspects of Declarative Languages (PADL)*, volume 1551 of *Lecture Notes in Computer Science*, pages 228–242. Springer Verlag, January 1999.

[26] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 224–235, January 2003.