Fling – A Fluent API Generator

Yossi Gil
Technion I.I.T Computer Science Dept., Haifa, Israel
yogi@cs.technion.ac.il

Ori Roth
Technion I.I.T Computer Science Dept., Haifa, Israel
ori.rothh@gmail.com

Abstract

We present the first general and practical solution of the fluent API problem – an algorithm, that given a deterministic language (equivalently, LR(k), $k \geq 0$ language) encodes it in an unbounded parametric polymorphism type system employing only a polynomial number of types. The theoretical result is accompanied by an actual tool Fling– a fluent API compiler-compiler in the venue of YACC, tailored for embedding DSLs in Java.

2012 ACM Subject Classification Software and its engineering → General programming languages; Software and its engineering → Domain specific languages

Keywords and phrases fluent API, type system, compilation, code generation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.13

Supplement Material ECOOP 2019 Artifact Evaluation approved artifact available at
https://dx.doi.org/10.4230/DARTS.5.2.12

Funding *Yossi Gil*: Technion I.I.T
Ori Roth: Technion I.I.T

1 Introduction

Given a formal language ℓ over some finite alphabet Σ, i.e., $\ell \subseteq \Sigma^*$ and a host programming language, e.g., Java [1], the *fluent API problem* is to generate $E = E(\ell)$, a set of definitions in the host language that encode ℓ such that membership in ℓ is equivalent to type-checking against E: Concretely, for a word $w = \sigma_1\sigma_2 \cdots \sigma_n$, $\sigma_i \in \Sigma$, $i = 1, \ldots, n$, the chain of method calls

$$\sigma_0.\sigma_1().\sigma_2().\cdots.\sigma_n().\sigma_\$() \tag{1}$$

type checks (in the host language) against $E(\ell)$ if and only if $w \in \ell$. (Here variable σ_0 and method $\sigma_\$$ are specified by E.) The fluent API problem is parameterized by ℓ's ranking in the Chomsky hierarchy and the capabilities of the host language.

Attention was drawn to the problem since fluent APIs are a valuable software engineering technique, useful, e.g., when ℓ specifies an object protocol, or for embedding a domain specific language (DSL) in the host language. (See, e.g., [5] for motivation and applications.)

A straightforward solution for the limited set of regular languages has been known and used for years; yet, interesting DSLs and protocols are often not regular.

Previous *theoretical* advances at the problem remain unpractical due to algorithm complexity, size of $E(\ell)$, or, inherent time complexity of the type checking. Prior attempts to make these theoretical results *practical* remain ad hoc, with no clear specification of the class of formal languages that can be processed.

© Yossi Gil and Ori Roth;
licensed under Creative Commons License CC-BY

1.1 Contribution

This work presents the first general and practical solution of the fluent API problem. To this end, we prove that any **D**eterministic **C**ontext **F**ree **L**anguage (DCFL) can be encoded in a type system that supports unbounded parametric polymorphism, while employing only a polynomial (in the size of the language specification) number of types.

Recall that DCFL is the class of formal languages that are recognizable by a **D**eterministic **P**ush**D**own **A**utomaton (DPDA). Our proof is supported by an *automata compiler* which converts a DPDA into JAVA definitions so that (1) holds.

The *generality* of the result is in two respects:

- Most programming languages, and hence most DSLs, are designed with practical parsing in mind. The DCFL class includes all languages for which an LL or LR grammar exists.
- The requirements from the type system of the host language are minimal. In particular, we assume the type parameterization:
  - does not allow the generic invoke functions found in the in its type parameter, nor expose any other property of this type
  - does not allow generic class make any assumption, nor place constraints on its type parameter, inherit from it, or supply a default value for it.
  - does not support partial specialization of generic classes (with the help of these, it is possible emulate a two-stack machine, and henceforth a Turing machine [9]),

  In fact, the generics we assume are as weak as found in ML. The only thing a generic can do with its parameter is to pass it as a parameter to another generic.

Conversely, the *practicality* of the result is in two respects:

- The time to type checking an expression such as (1) is linear in its size.
- The length of $E(\ell)$ is polynomial in the size of the DPDA that defines ℓ.

We further demonstrate the theoretical result in an actual tool FLING– a fluent API compiler-compiler in the style of YACC, tailored for embedding DSLs in JAVA. Given an LL(1) grammar $\mathcal{G}$, FLING generates appropriate JAVA definitions of modest size by which (1) type-checks, if and only if $\mathcal{G}$ derives w.

FLING's restriction to LL(1) is not inherent – extending FLING to support LR(1) languages (and hence all deterministic languages) is technical (though laborious). Further, FLING generates JAVA class definitions of the abstract syntax tree (AST) implicit by $\mathcal{G}$. Even further, FLING generates bodies for methods σ_i that generate the AST: When executed, the sequence (1) returns an instance of this AST that describes w, to be used by clients for further processing.

To demonstrate, we use FLING to define a fluent API language for writing regular expressions. In this API the regular expression $(ab?)^* \mid (0-9)^+$ is written as:

```
re.noneOrMore(exactly("a").and().option(exactly("b"))).or().oneOrMore(anyDigit()).$()
```

Composing the API begins with a grammar for the language ℓ_R of regular expressions:

$$
\begin{aligned}
\langle Expression\rangle &::= \texttt{re}\ \langle RE\rangle \\
\langle RE\rangle &::= \texttt{exactly}\ \langle Tail\rangle \\
&\mid \texttt{option}\ \langle Tail\rangle \\
&\mid \texttt{noneOrMore}\ \langle Tail\rangle \\
&\mid \texttt{oneOrMore}\ \langle Tail\rangle \\
&\mid \texttt{either}\ \langle Tail\rangle \\
&\mid \texttt{anyChar}\ \langle Tail\rangle \\
&\mid \texttt{anyDigit}\ \langle Tail\rangle \\
\langle Tail\rangle &::= \texttt{and}\ \langle RE\rangle \mid \texttt{or}\ \langle RE\rangle \mid \varepsilon.
\end{aligned}
\tag{2}
$$

This grammar is then supplied into FLING; incidentally, the grammar specification for FLING uses the fluent API style:

Listing 1 FLING fluent API specifying the regex grammar depicted in (2).

```
1  start(Expression). // ⟨Expression⟩ is the start symbol
2  derive(Expression).to(re, RE). // ⟨Expression⟩ ::= re⟨RE⟩
3  derive(RE).to(exactly.with(String.class), Tail) // ⟨RE⟩ ::= exactly⟨Tail⟩
4    or(option.with(RE), Tail). // | option⟨Tail⟩
5    or(noneOrMore.with(RE), Tail). // | noneOrMore⟨Tail⟩
6    or(oneOrMore.with(RE), Tail). // | oneOrMore⟨Tail⟩
7    or(either.with(RE, RE), Tail). // | either⟨Tail⟩
8    or(anyChar, Tail). // | anyChar⟨Tail⟩
9    or(anyDigit, Tail). // | anyDigit⟨Tail⟩
10 derive(Tail).to(and, RE).or(or, RE).orNone(); // ⟨Tail⟩ ::= and⟨RE⟩| or⟨RE⟩| ε
```

The above chain of method calls produces, at run time, a FLING object of type BNF. This object can then be requested to emit the JAVA (indeed, also C++ [19]) code that implements the fluent API. To use this API, one must compile and link again this emitted code.

One may disregard the regular-expression example and others as being singular, asking whether it would be better to manually compose a fluent API for regular expressions and other examples, rather than developing a general purpose machinery. An answer would be in considering the problem of embedding XML literals and expressions as fluent code, by writing, e.g.,

```
XMLData t = data().th().td("Customer").td("Number").th(end).tr().td("A").td(3).tr(end).data(end);
```

to create an XML data object. To maintain correctness of the generated object, the fluent API that begins with `data()` must support the DTD (XML schema) of an `XMLData` object. Alas, this schema could be very different in different applications. A tool such as FLING is handy in creating a fluent API to support different XML schemas and their evolution through the software's lifetime.

It is interesting that previous efforts to introduce XML literals and expressions into programming languages fail to do the DTD specific type checking. Such is the case with XJ [10], an XML extension to JAVA in which all XML instances belong to the same type. Similarly, XML literals of Visual Basic do not distinguish between various DTDs. Also, attempts to introduce XML into C# [11] [16, 15] ignored the issue of static type checking of distinct XML objects. To our knowledge, the only language supporting typed XML objects is HASKELL [20]. With the current contribution, XML objects with DTD static typing can be easily introduced into JAVA, C# and C++, for all DTDs that can be written as a deterministic language. (Note however that a DTD requirement that all rows in the table have the same number of columns, is not context free, for the same reason that the language $a^n b^n c^n$ is not context free.)

A concern often raised against fluent API carry is that the syntactic baggage of parenthesis and a '.' dot operator in each call obscures the syntax of the implemented language. This is true for JAVA in the implementation we offer. We believe it might be possible to tune the implementation to use fields rather than methods whereby minimizing the baggage to the '.' dot operator. The same is possible in languages such as EIFFEL [12], in which it possible to omit the empty parenthesis in calls to methods that do not take parameters. A workaround to omit the parentheses is possible in C# with getter methods.

We further note that one of the most successful examples of embedding a DSL in a host language, namely of SQL in C# with LINQ, relies on a fluent API. SQL code excerpts found in C# are free of any syntactic baggage and use the SQL syntax. An SQL excerpt is converted in a technical iteratively process (which involves little parsing if any) into a fluent API call chain. The definitions beyond this fluent API were handcrafted for the particular syntax. We believe it should be possible to extend this mechanism to support user defined DSLs.

1.2 Previous work

Gil and Levy [5] described the first non-trivial solution of the fluent API problem, recognizing the same class of languages as we do here, and making the same requirements from the type system of the host language as we do. However, in their construction $E(\ell)$ is exponential in the specification of ℓ. Moreover, since their result relies on on a very complex theoretical construction [3], no implementation is provided nor one does seem feasible.

Grigore [8] noticed that bounded below parametric polymorphism, e.g., JAVA's `super` constraints on generics, makes it possible to coerce the type checker into non-constant computation, going up and down the inheritance tree. With this observation, he was able to show that JAVA generics are Turing-complete and solve the fluent API problem for any recursive language ℓ. For a context free grammar language ℓ, his construction produces a polyomially sized $E(\ell)$. However type checking an expression of size n requires $O(n^9)$ time. In Grigore's words,

> *"...the degree of the polynomial is not exactly encouraging. There is room for improvement, and work to be done to achieve a practical parser generator for fluent interfaces."*

Indeed, practical parser generators for fluent APIs restrict themselves to using unbounded parametric polymorphism as we do here.

Fajita [14] is a tool for generating a fluent API definition from a given grammar specification. The class of grammars accepted by Fajita is contained in LL(1), but otherwise unspecified. Also, unlike FLING which generates an AST, Fajita can only be used for language recognition but not for language processing.

In contrast, *Silverchain* by Nakamaru et al. [18] is a real compiler-compiler. However, the class of languages it can process has not been defined. Unlike FLING, *Silverchain* fails on the many common languages that require an unbounded number of ε-transitions (see below Sect. 2).

Another related contribution is by Xu's [22] compiler-compiler for LL(1) languages provided they are given in Greibach Normal Form.

Outline. *Sect. 2 defines deterministic pushdown automata, establishes vocabulary and gives some intuition on the difficulty posed by ε-transitions. With these, we proceed in Sect. 3 to describe how deterministic pushdown automata can be emulated by a realtime device that uses tree encoding data structure, rather than a stack. Sect. 4 demonstrates (using* JAVA*) how the emulation can be compiled to any unbounded parametric type. The* FLING *tool and its implementation are the subject of Sect. 5. Sect. 6 concludes, mentioning directions for further research.*

2 Pushdown Automata

Intuitively, a pushdown automaton (PDA) is a finite state automaton (FSA) additionally equipped with a stack whose values are drawn from some finite set of stack symbols. A PDA has two kinds of transitions:

1. In a *consuming transition*, the PDA consumes an input letter and pops a symbol from the top of stack. Then, depending on the popped symbol, the input letter, and its current state, the PDA moves into another state, and pushes onto the stack a (possibly empty) sequence of stack symbols.
2. *An ε-transition* is similar to a consuming transition, except that no input letter is consumed: The selection of the next state and the symbols to push in the transition therefore depends solely on the stack's top and the current state of the automaton.

The PDA process the input letter by letter: For each letter, the automaton carries out a single consuming transition, followed by zero or more ε-transitions. There is no upper bound on the number of ε-transitions carried out for a single input letter. As our next example shows, this number may be linear in the input size.

2.1 Example

Consider a simple (balanced) parentheses language defined by the grammar of Fig. 1. In this language each opening parenthesis, ‘`(`’, must be balanced with a closing ‘`)`’, except that a closing square bracket, ‘`]`’, closes *all* previous opening parentheses.

$$
\begin{aligned}
\langle \mathit{Word} \rangle &::= \langle \mathit{Word} \rangle\ \langle \mathit{Word} \rangle \\
&\mid \langle \mathit{Balanced} \rangle \\
&\mid \langle \mathit{Squared} \rangle \\
&\mid \varepsilon
\end{aligned}
\qquad
\begin{aligned}
\langle \mathit{Balanced} \rangle &::= \langle \mathit{Balanced} \rangle\ \langle \mathit{Balanced} \rangle \\
&\mid \texttt{(}\ \langle \mathit{Balanced} \rangle\ \texttt{)} \\
&\mid \varepsilon \\
\langle \mathit{Squared} \rangle &::= \texttt{(}^{+}\ \langle \mathit{Balanced} \rangle\ \texttt{]}
\end{aligned}
$$

Figure 1 Grammar for a parentheses language in which ‘`(`’ and ‘`)`’ are balanced except that ‘`]`’ closes all currently open parentheses.

To recognize this language a PDA should maintain a stack to count all unclosed opening parentheses. The stack should be cleared when a ‘`]`’ is encountered. One such PDA is depicted in Fig. 2.

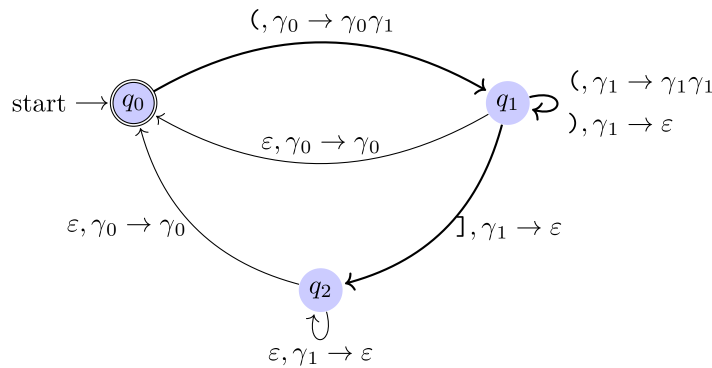


Figure 2 A deterministic pushdown automaton recognizing the balanced parentheses language of Fig. 1.

The figure uses the usual representation of automata as a multi-graph whose nodes are the states of the *inner* FSA of the PDA, while multi-edges emanating from a node describe the transitions taken by the PDA when being in this state. Edge labels describe the dependency of the transition on both the current input letter and the top stack, *and* the sequence of stack symbols to be pushed in response to these. The figure also employs the convention that consuming transitions are depicted as thicker edges than ε-transitions.

The automaton in the figure can be in one of three inner states: q_0, q_1, and, q_2. It uses of stack symbols, γ_0, marking the bottom of the stack, and γ_1. The number of occurrences of γ_1 on the stack is count of the yet unmatched opening parentheses.

Focus on the self edge of q_1 and the two labels that annotate it.

The *first label* `(`$,\gamma_1 \to \gamma_1\gamma_1$ means that if the automaton is in this state, and if the current input letter is '`(`', and if the stack's top is γ_1, then the automaton carries out a consuming transition, in which the stack top is replaced by $\gamma_1\gamma_1$, hence maintaining the unary representation of n.

The *second label* `)`$,\gamma_1 \to \varepsilon$ over this edge means that if the current input letter is '`)`' then a γ_1 at the top of the stack is popped without being replaced.

Notice that if after popping, the symbol at the top of the stack is also γ_1, then no further processing of this input letter occurs. If however after popping, γ_0 is revealed, then the transition consuming '`)`' is followed by precisely one ε-transition, in which the automaton traverses the edge from q_1 to q_0. The label $\varepsilon, \gamma_0 \to \gamma_0$ on this edge demonstrates the notation for ε-transition: in this ε-transition the popped γ_0 is pushed back to the stack.

Any number of ε-transitions may occur if the input letter is '`]`': state q_1 is the only inner state in which this letter is legal, and the stack top must be γ_1: if this is not the case, the automaton aborts; Otherwise, it moves to inner state q_2, and then follows the self edge of this state in a sequence of n ε-transitions, popping all occurrences of γ_1 from the stack. Finally, the automaton follows another ε-transition which brings the automaton back into its initial internal state q_0.

2.2 Deterministic pushdown automata

We distinguish between deterministic and non-deterministic PDAs: At each point during its computation a *non-deterministic* PDA (NDPDA) may have several legal transitions (be they consuming or ε). For example, the automaton depicted in Fig. 2 is deterministic. An NDPDA chooses among these in the usual non-deterministic fashion: It accepts when there is a non-deterministic run which leads to an accepting state.

▶ **Definition 1** (DPDA). *A* deterministic pushdown automaton *(DPDA) M is a septuple $M = \langle \Sigma, Q, q_0, F, \Gamma, \gamma_0, \delta \rangle$ where Q is a finite set of* automaton states*, $q_0 \in Q$ is the* initial state*, $F \subseteq Q$ is the set of* accepting states*, Γ is a finite set of* stack symbols*, $\gamma_0 \in \Gamma$ is a designated symbol marking the initial* bottom of the stack*, and δ is the (partial)* transition function*:*

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to Q \times \Gamma^*; \tag{3}$$

A configuration *of a DPDA is a pair $\langle q, s \rangle \in Q \times \Gamma^*$ where q is M's inner FSA state and s is the stack contents.*

A DPDA gives an operational definition of a some (typically infinite) language $\ell \subseteq \Sigma^*$: Given a word $w \in \Sigma^*$, a DPDA M starts consuming input, left to right, at the configuration $\langle q_0, \gamma_0 \rangle$. For each input letter it encounters, automaton M carries out a consuming transition and followed by a (possibly empty) sequence of ε-transitions. Automaton M aborts, rejecting the input and announcing $w \notin \ell$, if it either *(i)* tries to consume a letter when the stack is empty, or if *(ii)* the transition function is undefined for the current combination of input letter and top of stack symbol.

Suppose that M is in configuration $\langle q, s \rangle$ and that the stack top is symbol $\gamma \in \Gamma$, i.e., $s = \gamma s'$ where $s' \in \Gamma^*$. Then, if $\delta(q, \sigma, \gamma) = \langle q', \alpha \rangle$ the automaton moves through a consuming transition to configuration $\langle q', \alpha s' \rangle$. Alternatively, the automaton also moves to configuration $\langle q', \alpha s' \rangle$ in through an ε-transition, if $\delta(q, \varepsilon, \gamma) = \langle q', \alpha \rangle$.

A configuration is called *consuming* if M is about to consume an input letter. *Intermediate* configurations are the configurations in which M attempts to carry out an ε transition. Thus, a consuming configuration is one in which the automaton cannot make any ε transitions; an intermediate configuration is one in which such transitions are possible.

It follows from the determinism requirement that we can assume the initial configuration $\langle q_0, \gamma_0 \rangle$ is consuming: If this is not the case, then initially the automaton must choose between a possible ε-transition and a consuming transition.

The automaton accepts, announcing $w \in \ell$, and terminates if after w is consumed in full, it reaches a consuming configuration $\langle q, s \rangle$, where $q \in F$.

Notice that it is technically possible for a DPDA to loop indefinitely via pushing ε-transition, e.g., by repeatedly pushing symbols into the stack, or, by repeatedly replacing the stack's top. This singularity is easy to detect and can be ignored. We tacitly assume that a DPDA always halts.

2.3 Deterministic languages

Unlike FSAs, the expressive power of PDAs depends on whether they are deterministic or not: The set of languages accepted by NDPDAs is exactly the set of languages that can be specified by a context free grammar (CFG). In contrast, the set of languages accepted by DPDAs, also called the set of *deterministic languages*, is exactly the set of languages recognizable by an LR(1) parser [13]. Another important property of deterministic languages is that they are guaranteed to have an unambiguous grammar. (In contrast, some non-deterministic languages are inherently ambiguous, i.e., all CFGs for such a language are ambiguous.)

Parsing algorithms used in practice, e.g., LL(k) and LR(k) employ a deterministic automaton. In fact, the computation in all of the classical LL(1)-, SLR-, LALR-, and LR(1)-parsers is essentially that of a DPDA.

Deterministic languages are all context free, but not all context free languages are deterministic. For example, the language specified by the condition

$$\{\ ww^r \mid w \in \Sigma^+ \wedge w^r \text{ is } w \text{ in reverse order}\ \}, \tag{4}$$

or, equivalently, the context free grammar

$$\begin{aligned} \langle S \rangle\ &::= a\ \langle S \rangle\ \ a \\ &\ \mid\ b\ \langle S \rangle\ \ b \\ &\ \mid\ \varepsilon, \end{aligned} \tag{5}$$

requires a PDA to "guess" when w ends and w^r begins. If the guess is correct, then the automaton must pop symbols of the stack. If it is not, more symbols must be pushed. A non-deterministic automaton can explore both options for each letter in the input. Such guessing cannot be done in a deterministic fashion.

2.4 Simplification of DPDAs

A *real-time* automaton is an automaton that carries out precisely one transition for each input symbol. The challenge in producing a real-time pushdown automaton is in the consolidation of an unbounded number of (ε-) transitions into one. Next we do the first step towards this consolidation:

We say that the transition is *popping* if α is empty, and that it is *pushing* otherwise. We show how a given DPDA can be transformed into an equivalent one, in which pushing transitions are never followed by ε-transitions.

▶ **Lemma 2.** *For every DPDA M, there is a DPDA M' recognizing the same language as M, such that every pushing transition of M' leads to a consuming configuration, in which no further ε-transitions are possible.*

Proof. Consider any particular pushing transition. If no ε-transition follows it, we are done. Otherwise, we consolidate the transition with the subsequent one. We need to consider four cases, as depicted in Fig. 3.

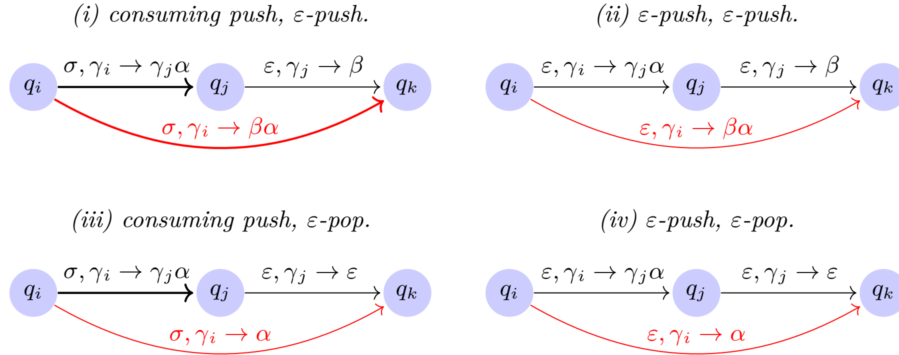


Figure 3 Four cases of consolidating a pushing transition with a subsequent ε-transition.

The top left of the figure shows case *(i)*, in which the first transition is consuming push and the second transition is an ε-push: A DPDA at state q_i and top stack symbol γ_i consumes letter σ removes γ_i, pushes back a sequence $\gamma_j\alpha$, and moves to state q_j. In this state, it carries out an ε-push in which γ_j is removed and replaced with a sequence β of stack symbols, and moves to state q_k.

The cumulative effect of the two transitions is then: consuming σ, moving from q_i to q_k and pushing the sequence $\beta\alpha$. As shown in the figure, the automaton can be transformed without perturbing its semantics, by replacing the edge $\langle q_i, q_j\rangle$ labeled $\sigma, \gamma_i \to \gamma_j\alpha$ (rendered in black in the figure) with an edge $\langle q_j, q_k\rangle$ with labeled $\sigma, \gamma_i \to \beta\alpha$ (rendered in red).

Notice that edge $\langle q_j, q_k\rangle$ with label $\varepsilon, \gamma_j \to \beta$ is not eliminated in the transformation: In the case that there are other edges leading to q_j, elimination of the edge would change the semantics of the automaton.

Case *(ii)* is similar, except that the first transition is not consuming. In this case, the transformation replaces edge $\langle q_i, q_j\rangle$ with label $\varepsilon, \gamma_i \to \gamma_j\alpha$ with a $\langle q_j, q_k\rangle$ edge with label $\varepsilon, \gamma_i \to \beta\alpha$.

These two cases assume that the second transition pushes a non-empty sequence β of stack symbols. In both cases, the consolidating transition is a pushing consuming transition, which could be consolidated further.

Cases *(iii)* and *(iv)* in Fig. 3 pertain to the situation in which the sequence β is empty, i.e., the second transition is an ε-pop. As shown in the figure, the transformations correspond respectively to cases *(i)* and *(ii)*, with the assumption $|\beta| = 0$. In these cases, the consolidated transition pushes α, which may, or may not be empty. If α is non-empty, then the consolidated transition is a push, which may be consolidated further with a succeeding ε-transition (if present).

We repeatedly apply the transformations depicted in Fig. 3 as long as they are applicable: If $\langle q_i, q_k\rangle$ is an edge that the transformation yields (marked in red in the figure) is pushing, then it must be further consolidated with any ε-edge outgoing from q_k.

Clearly, the process must stop, and when it does, no ε-transitions can follow a pushing transition. ◀

What impact does the transformation described in Lemma 2 have on the size of the automaton? Each step of the transformation consolidates two edges into one. Revisiting

Fig. 3 we see it is not always safe to remove edge $\langle q_j, q_k\rangle$: There might be yet another state $q_{i'}$ (not depicted in the figure) with an edge $\langle q_{i'}, q_j\rangle$. If $\langle q_j, q_k\rangle$ is eliminated then the four case analysis of q_i, q_j and q_k and the edges that connect them cannot be repeated for states $q_{i'}$, q_j and q_k and their edges.

The number of edges in the automaton may sometimes be reduced in the course of the transformation. However, the encoding of the automaton typically increases in size. In cases *(i)* and *(ii)* edge $\langle q_i, q_j\rangle$ with string α is replaced by edge $\langle q_i, q_k\rangle$ with string $\alpha\beta$. If $|\beta| > 1$, the length of the label increases. We argue that this increase is polynomial in the size of the specification of the automaton: The assumption that the automaton halts is tantamount to the claim that no ε-transition can occur twice in the processing of a single input letter, and hence can add $|\beta|$ symbols to each other edge at most once.

3 Realtime emulation of DPDAs with tree encoding

There are three steps in our algorithm for converting a given DPDA M into to a fluent API encoding of $L(M)$, the language recognized by M: *First* we convert M to an equivalent automaton where no pushing transition can be followed by an ε-transition, relying on Lemma 2 from the previous section (Sect. 2). *Second*, in this section we show how M can be encoded in a tree data structure that makes it possible to process an input letter in constant time. Effectively, we emulate the computation of M using this data structure. *Finally*, the following section (Sect. 4) will hows how this data structure and the constant time processing can be compiled to Java's type system, using only unbounded parametric polymorphism.

3.1 Encoding configurations as trees

Let $M = \langle\Sigma, Q, q_0, F, \Gamma, \gamma_0, \delta\rangle$ be implicit henceforth, and suppose that $Q = \{q_0, \ldots, q_h\}$ for some $h \geq 0$. Recall that configurations of M are pairs $\langle q, s\rangle \in Q \times \Gamma^*$, storing the inner state q and a string $s \in \Gamma^*$ that represents the entire contents of the stack.

For an intermediate configuration c define c^ε, the *ε-closure* of c, as the consuming configuration obtained from c after carrying out all possible ε-transitions. If c is consuming then $c^\varepsilon = c$. Observe that c^ε depends only on c, and not on the input.

The *encoding of a configuration* $c = \langle q, s\rangle$, written $e = e(c)$, is a complete tree of degree $h+1$: The root node of e carries labels q and α, where α is a prefix of s, i.e., $s = \alpha r$, $r \in \Gamma^*$, and where $|\alpha| > 0$ is bounded by some constant that depends on M, but not on the stack's depth.

More generally, any non-leaf sub-tree e' of e (including e itself) is an encoding of some configuration $c' = \langle q', s'\rangle$, where s' is a suffix of r, and has two labels associated with it:

1. a *state label*, which is the state q',
2. a *stack prefix label* α', a non-empty string of stack symbols whose length does not depend on the input.

We will use the field notation $e.q$ for the state label; $e.\alpha$ for the stack prefix label.

Consider, for example, the DPDA of Fig. 2 recognizing the parentheses language specified by Fig. 1. The configuration $\langle q_1, \gamma_1\gamma_1\gamma_1\gamma_0\rangle$ of this automaton (obtained, e.g., after reading the input prefix "`(((`") is encoded by our algorithm as depicted in Fig. 4.

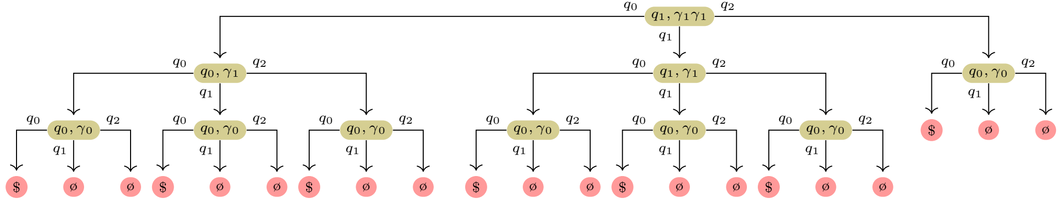


Figure 4 The tree encoding of configuration $\langle q_1, \gamma_1\gamma_1\gamma_1\gamma_0\rangle$ of the automaton of Fig. 2.

(We are oblivious to the question of whether encoding, as defined above, of a configuration is unique; incidentally, our algorithm always creates the same encoding for the same configuration.)

Since the automaton has three inner states q_0, q_1, and, q_2, the tree is, as can be seen in the figure, of degree 3. Examine now the path that starts at the root, and choose the edge labeled q_0 until a leaf is encountered. Collecting the stack prefix labels along this path, yields first the $\gamma_1\gamma_1$, then γ_1 and then γ_0, whereby reconstructing the full stack of the configuration $\langle q_1, \gamma_1\gamma_1\gamma_1\gamma_0\rangle$. Following the q_2 edge will only yield fragments of this stack contents. However, this curious property is not true for all encodings – the consolidation of pushing transitions may make changes to the stack contents, so that s' is not necessarily a suffix of s, even if e' occurs in e.

Notice that the same sub-trees occur several times in the figure. We will see below that even though tree encoding may be exponential in size, it has an efficient, linearly sized, representation in memory.

The emulation does not examine s' directly: Instead, it stores in the ith child of e, written either as $e[i]$ or $e[q_i]$, the encoding of configuration $\langle q_i, s'\rangle^{\varepsilon}$. Intuitively, the emulation does not know to which state q_i automaton M will move when α is removed from the stack, so it stores the result of the computation for all possible values of q_i. More precisely, we have the recursive encoding property

▶ **Property 1** (Recursive encoding property). *If e is the encoding of configuration $c = \langle q, s\rangle$ and $e.\alpha = \alpha$, then $e[i]$ is the encoding of the ε-closure of the configuration of $\langle q_i, s'\rangle$, i.e.,*

$$\forall i = 0, \dots, h \bullet e[i] = e\left(\langle q_i, s'\rangle^{\varepsilon}\right). \tag{6}$$

In general, configuration $\langle q_i, s'\rangle^{\varepsilon}$ depends on the stack contents s' and therefore computing its encoding may require an unbounded time. The emulation however these said values incrementally: whenever an encoding of a new configuration is generated, the values of its $h+1$ children are computed from the encoding of the previous configuration and its immediate children, as explained below in Sect. 3.5.

Also notice that the length of the encoded stack decreases as one goes down the tree. Leaves thus encode configurations in which the stack is empty. We allow two kinds of leaves: the special node $\bot$, denoting the (pseudo) configuration of M rejecting the input, and, $\top$, denoting the (pseudo) configuration of accepting it. It will become evident that these special nodes represent acceptance or rejection even in the case that the stack is not emptied.

3.2 Emulating a DPDA with tree encoding

There is only one valid encoding of the initial configuration $\langle q_0, \gamma_0\rangle$, since there is only one way of selecting α. All children of this configuration are leaves. Leaf i is $\top$ if $q_i \in F$ and $\bot$ otherwise. The emulation of M iterates from this initial encoding following Alg. 1.

Algorithm 1 Function emulate(w) – emulate the computation of DPDA M on $w \in \Sigma^*$; returns $\top$ if M accepts w, and $\bot$ otherwise.

```
1:  Function emulate(w):
2:  Let e ← newNode(q_0, γ_0)                    // encoding of initial configuration ⟨q_0, γ_0⟩
3:  For i ← 0, ..., h do                          // fill in the i^th child of e
4:     If q_i ∈ F then                            // M accepting the input
5:        e[i] ← ⊤                                // an accepting leaf
6:     else                                       // M rejecting it
7:        e[i] ← ⊥                                // a rejecting leaf
8:  For σ ∈ w do                                  // iterate on input letters, left to right
9:      e ← next(e, σ)                            // encoding of next consuming configuration
10:    If e = ⊥ then                              // M was unable to proceed
11:       Return ⊥                                // halt emulation rejecting the input
12: If e.q ∈ F then                               // M terminates in an accepting state
13:    Return ⊤                                   // accept the input
14: Return ⊥        // else, M terminated in a non-accepting state, reject the input
```

The emulation in the algorithm has three parts: In lines 2–7 the encoding of the initial configuration is computed. Input processing is in lines 8–11 – for each input letter, the emulator calls function next to compute the encoding of the next consuming configuration. If it is determined during the iteration that M does not have a valid transition, then the emulator aborts, rejecting the input. Finally, in lines 12–14, the emulator decides on accepting or rejecting the input, depending on whether M ended in an accepting state.

3.3 Computing the next encoding

The gist of the emulation is in function next(e, σ) (Alg. 2): given e, an encoding of a consuming configuration, and input letter σ, next(e, σ) returns the encoding of the consuming configuration obtained after consuming the input letter σ. This function computes the cumulative effect on the stack and the inner state of the consuming transition from e on σ and all ε-transitions that might follow.

Algorithm 2 Function next(e, σ) – given e, an encoding of a consuming configuration, and input letter σ, returns the encoding of the consuming configuration obtained after consuming σ.

```
1: Function next(e, σ):
2: Let q, α ← label of e                   // e is encoding of ⟨q, s⟩, s = αs', s' is unknown
3: Let γ ← first(α)                         // Pop is possible since α ∈ Γ^+
4: Let β ← rest(α)                         // e is encoding of ⟨q, s⟩, s = γβs', s' is unknown
5: If δ(q, σ, γ) is undefined then          // σ-consuming transition is undefined
6:    Return ⊥                              // The automaton rejects the input
7: Let q', α' ← δ(q, σ, γ)                  // Compute the consuming transition
8: continue as in Alg. 4
```

We see that next first (lines 2–4) determines that e is encoding of a configuration $\langle q, \gamma\beta s'\rangle$, where $\gamma \in \Gamma$ is the head of the stack, $\beta \in \Gamma^*$ is the known stack prefix under it, and $s' \in \Gamma^*$

is the unknown remainder of the stack. In principle, next can examine the values of $e[i]$ to determine s', but doing so would lead to a computation that depends on the stack size, and hence on the input, which we would like to avoid.

In lines 5–7, next examines the consuming transition of M. If this transition is undefined, then the input is rejected. Otherwise, next determines that M moves to state q' and replaces γ with the string α in the consuming transition, i.e., M moves to intermediate configuration $c = \langle q', \alpha'\beta s'\rangle$.

Function next uses an auxiliary recursive function, consolidate(e) (Alg. 3), which recursively computes the effect on the stack and the inner state of all ε-transitions that might follow: given e, an encoding of an intermediate configuration c, the function returns the encoding of c^ε.

Algorithm 3 Recursive function consolidate(e) – given e, an encoding of an intermediate configuration, returns the encoding of the consuming configuration obtained from e after all ε-transitions were carried out.

```
1: Function consolidate(e):
2: Let q, α ← label of e                     // e encodes ⟨q, s⟩, s = αs′, s′ is unknown
3: Let γ ← first(α)                          // Pop is possible since α ∈ Γ⁺
4: Let β ← rest(α)                           // e encodes ⟨q, s⟩, s = γβs′, s′ is unknown
5: If δ(q, ε, γ) is undefined then           // No further ε-transitions are possible
6:    Return e                               // Automaton is ready to consume
7: Let (q′, α′) ← δ(q, ε, γ)                 // Compute the ε transition
8: continue as in Alg. 4
```

We see that function consolidate is quite similar to next: It starts (lines 2–4) by determining that e is an encoding of a configuration $\langle q, \gamma\beta s'\rangle$, $\gamma \in \Gamma$, $\beta \in \Gamma^*$ and unknown remainder of the stack $s' \in \Gamma^*$. In lines 5–7, consolidate examines the forthcoming ε-transition of M. However, if this transition is undefined, it is determined no more ε-transitions are possible, and that e is in fact an encoding of a consuming transition. Function consolidate then simply returns e.

The similarity of next and consolidate goes further: after initial processing, they proceed identically: The common part of these two functions is described in Alg. 4, which, given e, an encoding of a configuration $\langle q, \gamma\beta s'\rangle$, and a transition from this configuration to new configuration $\langle q', \alpha'\beta s'\rangle$, returns the encoding of $\langle q', \alpha'\beta s'\rangle^\varepsilon$.

3.4 Correctness of the emulation

▶ **Lemma 3.** *If e is the encoding of an intermediate configuration c, then* consolidate(e) *returns the encoding of c^ε*

Proof. Let $c = \langle q, \alpha s'\rangle$, where $e.q = q$ and $e.\alpha = \alpha$. Recall that $|\alpha| \geq 1$ and the decomposition $\alpha = \gamma\beta$, $\gamma \in \Gamma$. By the lemma's assumption $e[i]$ is the encoding of $\langle q_i, s'\rangle^\varepsilon$ for $i = 0, \ldots, h$.

First notice that if $\delta(q, \varepsilon, \gamma)$ is undefined, then there is no ε-transition from c, c is a consuming configuration, and $c = c^\varepsilon$. In this case no further processing is required and consolidate returns e (line 6 in Alg. 3).

Otherwise the function focuses on the ε-transition leading from q to q' with label $\varepsilon, \gamma \to \alpha'$. In this transition M moves from c to configuration $c' = \langle q', \alpha'\beta s'\rangle$.

Algorithm 4 Common code of functions next (Alg. 2) and consolidate (Alg. 3) – given e, an encoding of an intermediate configuration $c = \langle q, \gamma\beta s' \rangle$, a state $q' \in Q$, and stack prefix α', the code considers an ε-transition from c to configuration $c' = \langle q', \alpha'\beta s' \rangle$, and returns e', the encoding of $(c')^\varepsilon$, the ε-closure of c'.

```
1:  If |β| = 0 then                                                        // |α| = 1
2:    If |α'| = 0 then                  // This is a popping transition. We encode ⟨q', s'⟩^ε
3:      Return  e[q']                   // Configuration ⟨q', s'⟩^ε is encoded by e[q'] by (6)
4:    else                              // |β| = 0, pushing ε-transition: encode ⟨q', α's'⟩
5:      Let e' ← newNode(q', α')        // e' is encoding of consuming configuration ⟨q', α's'⟩
6:      For i ← 0, . . . , h do                               // Copy the i^th child of e' from e
7:        Let e'[i] ← e[i]   // By definition, configuration ⟨q_i, s'⟩^ε is encoded by e[i] (= e[q_i])
8:      Return  e'                                          // Encoding of consuming configuration
9:  else                                                                    // |β| > 0, |α| > 1
10:   If |α'| = 0 then                  // This is a popping transition. We encode ⟨q', βs'⟩^ε
11:     Let e' ← newNode(q', β)        // e' is encoding of intermediate configuration ⟨q', βs'⟩
12:     For i ← 0, . . . , h do                                          // Create the i^th child of e'
13:       Let e'[i] ← e[i]                                     // Other than label e' is the same as e
14:     Return  consolidate(e')                // Then, continue recursively to yield ⟨q', βs'⟩^ε
15:   else                              // |β| > 0, pushing ε-transition: encode ⟨q', α'βs'⟩
16:     Let e' ← newNode(q', α')        // e' is encoding of consuming configuration ⟨q', α'βs'⟩
17:     For i ← 0, . . . , h do                 // Create the i^th child of e'. We encode ⟨q_i, βs'⟩^ε
18:       Let e'[i] ← newNode(q_i, β)    // Temporarily store the encoding of ⟨q_i, βs'⟩ in e'[i]
19:       For j ← 0, . . . , h do            // Create the j^th child of e'[i]. We encode ⟨q_j, s'⟩^ε
20:         Let e'[i][j] ← e[j]                                 // Encoding of ⟨q_j, s'⟩^ε is e[j] by (6)
21:       Let e'[i] ← consolidate(e'[i])       // Recursive call to compute encoding of ⟨q_i, βs'⟩^ε
22:     Return  e'                                  // Return consuming configuration ⟨q', α'βs'⟩
```

Function consolidate then carries on in Alg. 4 to compute and return c'^ε. If the transition is popping, consolidate calls itself recursively, to compute the effect of further ε-transitions. In the case it is pushing, no recursion is required thanks to the transformation of Lemma 2.

We complete the proof by induction on the length of α.

Inductive base. The case $|\alpha| = 1$, i.e., $|\beta| = 0$ is managed in lines 1–8 of Alg. 4. There are two sub-cases to consider:

Popping transition. If α' is empty, (line 2) then the transition is popping and $c' = \langle q', s' \rangle$. By Property 1 the encoding of c'^ε is stored in $e[q']$ which consolidate returns (line 3).

Pushing transition. If α' is not empty (line 4) the transition is pushing, $c' = \langle q', \alpha' s' \rangle$ is consuming and cannot be followed by ε-transitions, i.e., $c'^\varepsilon = c'$ (Lemma 2). In choosing the prefix α' for the encoding e' of c', we have that the stack remainder of e and e' are the same, i.e., $s = s'$. Encoding e' is therefore created with label q', α' (line 5) and reusing the children of e (lines 6–7). It is then returned by the function without any further processing (line 8).

Inductive step. If $|\alpha| > 1$ then string β, which is one character shorter than α, is not empty. Function consolidate then proceeds in line 15 with the same two sub-cases:

Popping transition. If α' is empty (line 10), then this is a popping ε transition and $c' = \langle q', \beta s' \rangle$. Function consolidate then defines a new encoding e' (line 11) with the non-empty prefix β. With this selection of stack prefix, the stack remainder of e' is the same as that of e, and e' reuses the children of e (lines 12–13).

The recursive call (line 14) then deals with the subsequent ε-transitions. It returns the correct result by the inductive hypothesis (recall that $|\beta| = |\alpha| - 1$).

Pushing transition. If α' is not empty (line 15), then the ε-transition under consideration is pushing. The automaton reaches configuration $c' = \langle q', \alpha'\beta s'\rangle$, and since no ε-transitions follow a pushing transition, we know that c' is consuming and $c' = c'^{\varepsilon}$.
Function consolidate then generates a new encoding e' with labels q' and α' (line 16). The remainder that follows the stack prefix α' in this case is not s', but rather $\beta s'$. The stack remainder s' is obtained with the aid of an extra level in the tree.
Indeed, the children of e' are labeled with β as a stack prefix (line 18). The stack remainder of each of these h children is s'. We can therefore reuse the children of e in populating the h^2 grandchildren of e' (lines 19–20).
Revisiting line 18, we see that we set the ith child of e' to the encoding of configuration $\langle q_i, \beta s'\rangle$. However, by Property 1, the ith child of e' should contain not the encoding of this configuration, but of its ε-closure.
The recursive call to consolidate in line 21 fixes the children of e': After this call, each of its children stores, as required by Property 1, the encoding of $\langle q_i, \beta s'\rangle^{\varepsilon}$.
As before, the correctness of the recursive call on the children is guaranteed by the inductive hypothesis and the fact that β, the stack prefix of each of these children, is one character short of α. ◀

▶ **Lemma 4.** *Suppose consuming configuration c is encoded by the tree e, and c yields the intermediate configuration c' upon consuming the input letter σ. Then configuration c'^{ε} is encoded by next(e, σ).*

Proof. If $\delta(q, \sigma, \gamma)$ is undefined, then there is no suitable σ-consuming transition from c, hence $c = \bot$. In this case next returns $\bot$ (line 6 in Alg. 2).

Otherwise there is a σ-consuming transition leading to state q' and replacing γ with α'. This case is managed in Alg. 4: Observe that this algorithm does not use the input letter σ. The rest of the proof is identical to the inductive part in the proof of Lemma 3. ◀

▶ **Lemma 5.** *Suppose configuration $c = \langle q, s\rangle$ is encoded by e with label q, α. Then the computation time of consolidate(e) is bounded by $O(|Q|^{1+|\alpha|})$ and does not depend on $|s|$, the stack's depth.*

Proof. Let us examine the body of function consolidate: It calls itself recursively in lines 14 and 21. In the first case the function calls itself once, and in the second it calls itself $|Q|$ times.

In both cases the function is called on a configuration with label containing the string β which is shorter than α, $|\beta| = |\alpha| - 1$. Therefore the depth of the call tree is α, and at the worst case $|Q|$ recursive calls are made in each non-leaf invocation. The number of recursive calls is at most $|Q|^{|\alpha|}$.

The proof is completed by noticing that the amount of work in the body of the function is $O(|Q|)$. ◀

3.5 Use of memory

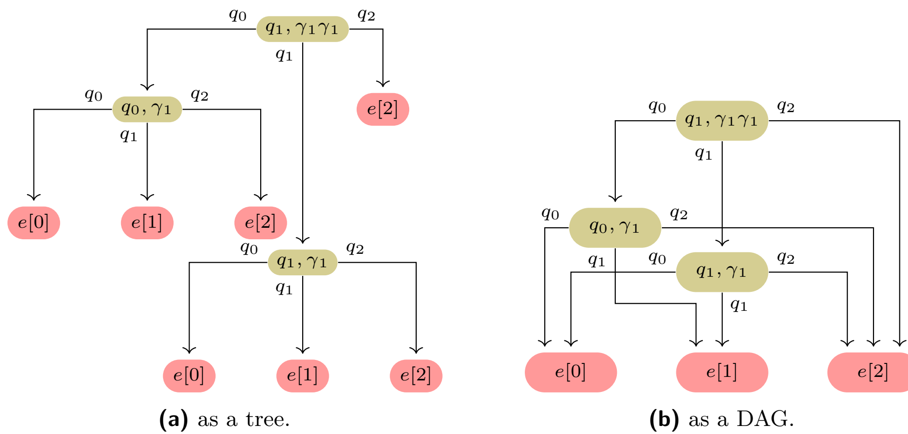
An encoding is represented by a tree data structure. Functions next and consolidate receive such a tree, and return another tree of this sort. The depth of these trees is linear in the stack size, and since their degree is $h + 1$, their total size may be exponential in the length of the input.

Examining the body of Alg. 4, we see that this exponential blowup is not an issue:

- The code receives as parameters references to children $e[0], \ldots, e[h]$ of encoding e: these references are copied in lines 7, 13, and 20, but never de-referenced.

- The body of `next` creates new tree nodes in lines 5, 11, 16, and 18 of Alg. 4.
  Examining these lines, we see that the number of these nodes is maximized if the algorithm executes lines 16, and 18. In this case, precisely $|Q|+1$ nodes are created before a recursive call is made. In total, `next` creates $O(|Q|^{|\alpha|})$ nodes.

We therefore represent encoding trees in memory as a compact DAG. As illustrated in Fig. 5, this encoding is natural: instead of copying children and grandchildren in creating a new encoding, one stores references to these.



(a) as a tree. **(b)** as a DAG.

Figure 5 Representation of a certain encoding as a tree (5a) and as a DAG (5b).

Since the number of primary calls to `next` is $|w|$, the number of letters in the input (Alg. 1), we have that the memory required for emulating the working of M is linear $|w|$. Fig. 6, showing the compact DAG encoding of in Fig. 4, demonstrates.

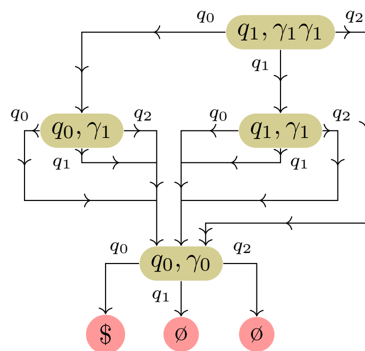


Figure 6 A DAG representation of tree encoding (Fig. 4) of configuration $\langle q_1, \gamma_1\gamma_1\gamma_1\gamma_0\rangle$ of the automaton of Fig. 2.

Comparing the figure to the DAG representation of the same configuration (Fig. 4 above), we see that only the nodes explicitly created by the algorithm are present in the DAG. Indeed, each of these nodes has three children, but as made clear by the figure, the sharing of children makes a significant saving in the size of the representation.

4 Compiling a Tree Encoding to Java

This section shows how to construct a JAVA fluent API definitions $E(\ell)$ for the language $\ell = \ell(M)$ recognized by a given DPDA M. These definitions should be such that the predicate $w \in \ell$ is equivalent to type checking the fluent API call chain expression $x = x(w)$ (see (1) above). JAVA is used for the sake of exposition: In essence, we show how to *compile* a DPDA specification into an *abstract* declaration in an unbounded parametric polymorphism type system (UPPTS): Correctness of compilation means that every run of the input specification (some programming language in the general compilation process, but DPDAs here), has an equivalent run in the output specification (written as machine code instructions in general, but as type declaration here), and vice versa.

An implementation of the construction is offered as part of the contribution[1] in the form of an automata compiler which translates a given DPDA to JAVA definitions. For example, to generate a fluent API for the balanced parentheses language of Fig. 1, a specification of the automaton (Fig. 2) that recognizes it is supplied to the compiler. The specification begins with three **enum** definitions, describing the finite sets of symbols Q, Σ and Γ[2]:

```
enum Q { q0, q1, q2 }
enum Σ { c, ɔ, Ɔ }
enum Γ { γ0, γ1 }
```

Observe the use of letter '`c`' instead of '`(`' (which is not a valid method name in JAVA). Also, letter ɔ (inverted lower case 'c') replaces '`)`' and '`Ɔ`' replace '`]`'. With these definitions, a JAVA model of the DPDA is constructed using the BUILDER design pattern (List. 2).

Listing 2 Supplying to the automata compiler the specification of the DPDA of Fig. 2.

```
1  DPDA<Q, Σ, Γ> M = new DPDA.Builder<>(Q.class, Σ.class, Γ.class).
2    q0(q0). // Starting in state q0
3    F(q0). // q0 is an accepting state
4    γ0(γ0). // γ0 is the bottom of stack marker
5    δ(q0, c, γ0, q1, γ0, γ1). // pushing consuming transition δ(q0,γ0,c) = ⟨q1,γ1γ0⟩
6    δ(q1, c, γ1, q1, γ1, γ1). // pushing consuming transition δ(q1,γ1,c) = ⟨q1,γ1γ1⟩
7    δ(q1, ɔ, γ1, q1). // popping consuming transition δ(q1,γ1,ɔ) = ⟨q1,ε⟩
8    δ(q1, null, γ0, q0, γ0). // pushing ε−transition δ(q1,γ0,ε) = ⟨q0,γ0⟩
9    δ(q1, Ɔ, γ1, q2). // popping consuming transition δ(q1,γ0,Ɔ) = ⟨q2,ε⟩
10   δ(q2, null, γ1, q2). // popping ε−transition δ(q2,γ1,ε) = ⟨q2,ε⟩
11   δ(q2, null, γ0, q0, γ0). // pushing ε−transition δ(q2,γ0,ε) = ⟨q0,γ0⟩
12   go(); // having accumulated the specification of M, build its model
```

The code in the listing builds in a fluent API fashion: the first three calls in the chain define the initial state, the accepting state, and the initial stack symbol. Then follows a series of calls to the JAVA function $\delta(\ldots)$. These are a piecemeal specification of the transition function of the modeled automaton: the call `δ(q0,c,γ0,q1,γ0,γ1)` is to say that the automaton in state q_0 and γ_0 at the stack top, moves to state q_1 and pushes back onto the stack γ_0 and then γ_1.

The final call in the above chain to `go()` concludes the construction, and the above code stores the DPDA model in variable `M` of generic class `DPDA`.

We stress that the JAVA code produced by the automata compiler is not meant to be run: the definitions in $J_{L(M)}$ are incomplete, and even if completed, evaluation of the fluent API expression does not produce any useful value, and will probably fail with run-time error. The generated code is used solely for type checking. Sect. 5 describes a more practical tool in which the evaluation of $x(w)$ returns the AST of word w.

[1] `https://github.com/OriRoth/jdpda`
[2] This, just as other code excerpts in the paper, is drawn from the implementation. Note that JAVA supports UTF characters

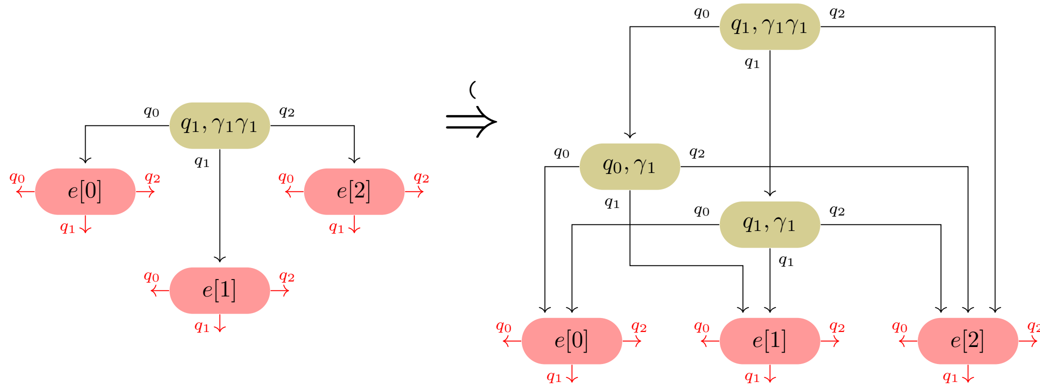


Figure 7 Partial evaluation of $\mathsf{next}(e(\langle q_1, \gamma_1\gamma_1 s'\rangle),$‘(’) presented as a DAG whose leaves are the implicit parameters (sub-trees) $e[0]$, $e[1]$ and $e[2]$.

4.1 Intuition

Compilation is based on the emulation algorithm described in the previous section. The main idea is partial evaluation: Instead of compiling to the type system the many details in Alg. 2, Alg. 3, and Alg. 4, the compiler relies on partial evaluation and caching. The JAVA type system is only used on the partially evaluation form.

Let $q \in Q$, $\alpha \in \Gamma^+$ and σ be fixed, and let e be defined by q and α and some stack remainder s'. Examining the three algorithms that make function $\mathsf{next}(e, \sigma)$ we see that for any such fixture the function constructs the same tree from the sub-trees $e[0], \ldots, e[h]$.

Consider for example the language of Fig. 1 and its DPDA (Fig. 2.) Suppose that M is in the consuming configuration $\langle q_1, \gamma_1\gamma_1 s'\rangle$, where $s' \in \{\gamma_0\gamma_1\}^*$ and that e is an encoding of this configuration with stack prefix $\alpha = \gamma_1, \gamma_1$. Indeed, trees $e[0]$, $e[1]$, and $e[2]$ depend on the actual contents of s, but next never inspects the contents of these trees.

For these values of q, α and for $\sigma =$‘(’, the call $\mathsf{next}(e, c)$ involves a call to consolidate, which may even call itself recursively. Working out the details, one finds that $\mathsf{next}(e, c)$ is in effect the transformation depicted in Fig. 7.

As shown in the figure, the net effect of $\mathsf{next}(e(\langle q_1, \gamma_1\gamma_1 s'\rangle),$‘(’ is to transform the tree encoding on the left to the tree encoding in the right. This transformation does not examine nodes $e[0]$, $e[1]$, and $e[2]$ (marked in red in the figure). Partial evaluation of $\mathsf{next}(e(\langle q_1, \gamma_1\gamma_1 s'\rangle),$‘(’) is therefore an oblivious function of $e[0]$, $e[1]$ and $e[2]$.

More generally, we have that for every q, α and σ the partial evaluation of next has a simple DAG structure that represents an oblivious function of $e[0], \ldots, e[h]$. Our DPDA to JAVA compiler computes and caches these DAGs for each combination of q, α, and σ it encounters.

Let $D(q, \alpha, \sigma)$ denote the DAG defined by q, α, and σ. Note that function $D(q, \alpha, \sigma)$ may also assume the special value $\bot$ in the case that $\delta(q, \gamma, \sigma)$ (γ being the first symbol in α) is undefined.

We argue that D has a finite representation: q and σ are drawn from finite sets. To see that the number of distinct stack prefixes α is finite, examine again lines 5, 11, 16, and 18 of Alg. 4 in which newNode is invoked: In all of these the stack prefix label attached to the newly created node is either a label α' of a certain transition of M, or β, a suffix of an existing label.

4.2 Structure of the encoding

The Java code emitted by the compiler contains these definitions:

1. A designated type that represents the (leaf) encoding $\bot$ – the encoding of a rejecting automaton.
```
interface ∅ { }
```
Notice that this designated type is not parametric.
2. A designated type that represents the (leaf) encoding $\top$ – the encoding of an accepting automaton.
```
interface $ { }
```
Again, this designated type is not parametric.
3. Parametric state types, each designating an encoding label $\langle q, \alpha \rangle$, $q \in Q$ and $\alpha \in \Gamma^+$, and each taking $|Q|$ unbounded type parameters. As a matter of convention, the name of this type is the string of symbols $q\alpha$ separated by underscores: For example, for the encoding label $\langle q_1, \gamma_1\gamma_1 \rangle$ (described in Fig. 7) the compiler generates the parametric type
```
interface q1_γ1_γ1<e0, e1, e2> {...}
```
4. A start variable (named `__` in the implementation), from which fluent API chains start. This variable is one of the generated parametric state types: specifically, the state type that bears the label of the initial configuration $\langle q_0, \gamma_0 \rangle$.
The values of the parameters are either the rejection or the accepting designated types, depending on whether the state corresponding to the parameter is accepting or not (as in Alg. 1). In our example,
```
q0_γ0<$, ∅, ∅>__= ...;
```

With these we have a representation of the tree encodings of configurations of M as a type obtained from the instantiation of of an appropriate Java generics: An encoding e with state label q, state prefix label $\alpha = \gamma_1\gamma_2 \cdots \gamma_k$ and children $e[0], \ldots, e[h]$ is represented by the following instantiation of the parametric state type
```
q_γ1_γ2···_γk<τ0, τ1, ..., τh>
```
with actual type parameters $\tau_0, \tau_1, \ldots, \tau_h$ being the type representation of child encodings $e[0], \ldots, e[h]$.

Alg. 1, the emulation of M with tree encoding is done step by step by the fluent API call chain: If M is in a configuration c after reading the input $\sigma_1\sigma_2 \cdots \sigma_i$, and e is the encoding of c. Then, the type of the fluent API call chain

$$\texttt{__.}\sigma_1\texttt{().}\sigma_2\texttt{().}\cdots\texttt{.}\sigma_i\texttt{()} \tag{7}$$

is precisely the type representation of e: A call to a method named σ`()` in the chain represents the consumption of input letter σ; the type that the method returns is the type encoding of the subsequent consuming configuration. It is the chief duty of the compiler to correctly generate this type. For each generic class t with certain q and α, the compiler examines every $\sigma \in \Sigma$:

- If $D(q, \alpha, \sigma) \neq \bot$, the compiler generates a method σ`()` in t, and uses $D(q, \alpha, \sigma)$ to specify the return type of σ in terms of the type parameters of t. In the example of Fig. 7, the compiler generates in `q1_γ1_γ1<e0, e1, e2>` a method `c()` whose return type is the Java representation of the DAG in Fig. 7:
```
q1_γ1_γ1<q0_γ1<e0, e1, e2>, q1_γ1<e0, e1, e2>, e2> c();
```
Examine, e.g., the first type argument of the return type of `c()`: The value of this type argument is `q0_γ1<e0, e1, e2>` which is exactly the q_0 child of the root of the DAG of the figure.

- Conversely, if $D(q, \alpha, \sigma) = \bot$, the compiler sets the return type of $\sigma()$ to the rejection type ø.

After constructing the variable `M` in List. 2 the call `M.compile()` returns a text including JAVA definitions for the fluent API of `M`. The code is shown in List. 3.

Listing 3 Output of the automata compiler for the DPDA of Fig. 2.

```
1  interface ø { } // designated type denoting rejection
2  interface $ { } // designated type denoting acceptance
3
4  q0_γ0<$, ø, ø> __ = null; // Initial configuration
5
6  interface q0_γ0<e0, e1, e2> { // (1) configurations ⟨q0, γ0 s'⟩
7    q1_γ1_γ0<e0, e1, e2> c(); // 'c' is the only input letter allowed in this state
8   $ $(); // Input may end in this configuration
9  }
10 interface q1_γ1_γ0<e0, e1, e2> { // (2) configurations ⟨q1, γ1γ1 s'⟩
11   q1_γ1_γ1<q0_γ0<e0, e1, e2>, q0_γ0<e0, e1, e2>, q0_γ0<e0, e1, e2>> c();
12   q0_γ0<e0, e1, e2> ɔ();
13   q0_γ0<e0, e1, e2> Ɔ();
14 }
15 interface q1_γ1_γ1<e0, e1, e2> { // (3) configurations ⟨q1, γ1γ1 s'⟩
16   q1_γ1_γ1<q0_γ1<e0, e1, e2>, q1_γ1<e0, e1, e2>, e2> c();
17   q1_γ1<e0, e1, e2> ɔ();
18   e2 Ɔ();
19 }
20 interface q0_γ1<e0, e1, e2> extends $ { // (4) Configurations ⟨q0, γ1 s'⟩
21   $ $(); // Input may end in this configuration
22   // No other input letter is legal here
23 }
24 interface q1_γ1<e0, e1, e2> { // (5) configurations ⟨q1, γ1 s'⟩
25   q1_γ1_γ1<e0, e1, e2> c();
26   e1 ɔ();
27   e2 Ɔ();
28 }
```

Observe in the code the two designated types for acceptance and rejection, and the start variable whose type is the initial configuration; then follow five parametric state types (interface `q0_γ<e0, e1, e2>` through `q1_γ1<e0, e1, e2>`). Also notice that classes of configurations with state $q_0 \in F$ offer a function `$()` returning the special type \$.

4.3 Correctness

With this construction we can argue

Lemma 6. *The JAVA expression* `__`$.\sigma_1$`()`$.\sigma_2$`()`$.\cdots.\sigma_i$`()`

1. *does not compile, producing a missing method error message, if M aborts on $\sigma_1\sigma_2\cdots\sigma_j$, $1 <= j <= i$, or,*
2. *is of a type that represents the encoding of the configuration of M after reading $\sigma_1\sigma_2\cdots\sigma)_i$, including the special type* `interface ø` *denoting rejection.*

Proof. Mundane, by induction on i: The type of `__` represents the encoding of the initial configuration. The return type of the call to method σ_i is, by construction, the type representation of the encoding of M. ◀

In addition, if $q \in F$, the code generator adds to type t a method `$()` whose return type is `interface $`, the special type denoting acceptance. This method marks the end of input in the emulation.

With this addition, we have that the fluent API chain

$$__.\sigma_1().\sigma_2().\cdots\sigma_n().\$()$$

type checks, if and only if, word $\sigma_1\sigma_2\cdots\sigma_n \in L(M)$.

5 Fluent-API Generation in Fling

The contributions of tree encoding (Sect. 3) and automata compilation strategy (Sect. 4) made it possible to develop FLING– a compiler-compiler in the vein of e.g., YACC and ANTLR: FLING receives its input in a form suitable for clients – an EBNF grammar rather than DPDA specification. It converts the grammar into an automaton, and generates the fluent API type definitions for the language specified by the grammar.

FLING improves on the automata type compiler: Having recognized the fluent API chain as a valid word, FLING also generates code to construct, at runtime, the AST of the chain, and provides clients with means for traversing this tree.

FLING is more limited than the automata compiler, since it can only process LL(1) grammars. The class of languages that can be expressed by such grammars is strictly contained in the class of deterministic languages, which can all be recognized by the automata compiler. However, the restriction to LL(1) grammars is not inherent – extending FLING to support LR(1) grammars (and hence all deterministic languages) is technical (though laborious): One needs to re-implement the classical LR(1) parser generator to produce its output in the format expected by a DPDA compiler. The compiler-compiler features of FLING that we describe here are applicable regardless of the parsing engine.

5.1 Embedding Datalog in Java using Fling

Our open source implementation of FLING[3] includes examples of a dozen or so (small) languages. Here we describe in brief the embedding of DATALOG [2] in JAVA.

Recall that DATALOG is a simpler version of PROLOG [4] in that predicates cannot be nested. List. 4 is a reminder of the syntax of DATALOG, depicting a simple program to manage the ancestral relation, including two facts, three rules, and one query.

Listing 4 A DATALOG program managing an ancestral relation.

```
1  parent(john,bob).
2  parent(bob,donald).
3  ancestor(adam,X).
4  ancestor(A,B) :- parent(A,B).
5  ancestor(A,B) :- parent(A,C), ancestor(C,B).
6  ancestor(john,X)?
```

List. 5 is the JAVA fluent API equivalent of the DATALOG program in List. 4. Code comment show the DATALOG equivalent of fragments of the call chain.

Listing 5 A fluent API specification of the DATALOG program of List. 4.

```
1  Datalog program = datalog.
2    fact("parent").of("john", "bob"). // fluent API of parent(john,bob).
3    fact("parent").of("bob", "donald"). // fluent API of parent(bob,donald).
4    always("ancestor").of(l("adam"), v("X")). // fluent API of ancestor(adam,X)
5    infer("ancestor").of(v("A"), v("B")). // fluent API of ancestor(A,B)
6      when("parent").of(v("A"), v("B")). // fluent API of :- parent(A,B).
7    infer("ancestor").of(v("A"), v("B")). // fluent API of ancestor(A,B)
8      when("parent").of(v("A"), v("C")). // fluent API of :- parent(A,C)
9      and("ancestor").of(v("C"), v("B")). // fluent API of , ancestor(C,B).
10   query("ancestor").of(l("john"), v("X")); // fluent API of ancestor(john,X)?
```

To create the fluent API demonstrated in List. 5, we start with the names of the methods involved in the chain. These are precisely the terminals (tokens) of the grammar that

[3] https://github.com/OriRoth/fling

generated the fluent API. We therefore write an **enum** definition that enumerates all these methods:

```
enum Terminals implements Fling.Terminals { infer, fact, query, of, and, when, always, v, l }
```

Notice that the methods in fluent API for writing embedded Datalog code take parameters, `infer("ancestor")` and `of("bob", "donald")`. To understand why, recall that grammars of programming languages such as Pascal [21] and C++ use two kinds of non-terminals:

- Keywords such as **begin**, **var**, punctuation such as '(', ';', ':', and operators such as ';' can appear in only one form in the program code. We call these *vacuous tokens*. Vacuous tokens can appear in programs in only one way; they serve as parsing aide in the concrete grammar, but are omitted from the abstract syntax tree.
- In contrast, tokens such as *StringLiteral* and *Identifier* carry additional information: A string literal token carries its content, and an identifier literal token carries its name. We call tokens of this sort *informational tokens*.

Grammars of fluent APIs tend to use informational tokens more than vacuous tokens. Compiler compilers such as YACC use a distinct lexical analyzer to specify the many shapes any informational token may take. Fling has no accompanying lexical analyzer. However, since terminals of fluent API are method names, the contents of an informational token is supplied as argument to the method. For example, an identifier token in a fluent API is typically written as `id("fubar")`.

Fig. 8 is the EBNF grammar of the fluent API used for embedding Datalog in Java (e.g., List. 5). The grammar makes frequent use of informational tokens: writing `l("thingy")` is to say that the string parameter is a literal, while `v("thingy")` is to say that it is a literal. Also, the ⟨*Fact*⟩ symbol is composed of two informational tokens: method `fact` in which the Datalog predicate name is supplied as argument, and method `of` in which the literal parameters are supplied.

⟨*Program*⟩ ::=⟨*Statement*⟩$^+$
⟨*Statement*⟩ ::=⟨*Fact*⟩ | ⟨*Query*⟩ | ⟨*Rule*⟩
⟨*Fact*⟩ ::=`fact`(⟨*String*⟩) `of`(⟨*Literal*⟩*)
⟨*Query*⟩ ::=`query`(⟨*String*⟩) `of`(⟨*Term*⟩*)
⟨*Rule*⟩ ::=⟨*Bodyless*⟩ | ⟨*WithBody*⟩
⟨*Bodyless*⟩ ::=`always`(⟨*String*⟩) `of`(⟨*Term*⟩*)
⟨*WithBody*⟩ ::=⟨*RuleHead*⟩ ⟨*RuleBody*⟩
⟨*RuleHead*⟩ ::=`infer`(⟨*String*⟩) `of`(⟨*Term*⟩*)
⟨*RuleBody*⟩ ::=⟨*FirstClause*⟩ ⟨*AdditionalClause*⟩*
⟨*FirstClause*⟩ ::=`when`(⟨*String*⟩) `of`(⟨*Term*⟩*)
⟨*AdditionalClause*⟩ ::=`and`(⟨*String*⟩) `of`(⟨*Term*⟩*)
⟨*Term*⟩ ::=`v`(⟨*String*⟩) | `l`(⟨*String*⟩)

Figure 8 EBNF grammar of embedding Datalog in Java.

Observe that the same token may take different arguments in different contexts, e.g., token `of` any number of plain strings when it is part of a ⟨*Fact*⟩ , and any number of ⟨*Term*⟩ values when it appears as part of a ⟨*Rulehead*⟩ . Notice that symbol ⟨*Term*⟩ is also defined by the grammar in Fig. 8: In general, the information that accompanies a token is not limited to lexical values, and may be defined by its own grammar.

The specification of the grammar of Fig. 8 requires an **enum** definition of the list of symbols

```
enum Symbols implements Fling.Symbols {
  Program, Statement, Rule, Query, Fact, Bodyless, WithBody,
  RuleHead, RuleBody, FirstClause, AdditionalClause, Term }
```

List. 6 uses **enum** `Tokens` and **enum** `Symbols` in a fluent API chain of methods used to describe to Fling the grammar in the Fig. 8.

Listing 6 A fluent API specifying the DATALOG grammar of Fig. 8.

```
1  BNF bnf = bnf(Terminals.class, Symbols.class). // Create a BNF with these terminals and symbols
2    start(Program). // ⟨Program⟩ is the start symbol
3    derive(Program).to(oneOrMore(Statement)). // ⟨Program⟩ ::= ⟨Statement⟩+
4    specialize(Statement).into(Fact, Query, Rule). // ⟨Statement⟩ ::= ⟨Fact⟩ | ⟨Query⟩ | ⟨Rule⟩
5    derive(Fact).to(fact.with(String)).and(of.many(String)).
6      // ⟨Fact⟩ ::= fact(⟨String⟩) of(⟨Literal⟩*)
7    derive(Query).to(query.with(String)).and(of.many(Term)).
8    // ⟨Query⟩ ::= query(⟨String⟩)of(⟨Term⟩*)
9    specialize(Rule).into(Bodyless, WithBody). // ⟨Rule⟩ ::= ⟨Bodyless⟩ | ⟨WithBody⟩
10   derive(Bodyless).to(always.with(String)).and(of.many(Term)).
11     // ⟨Bodyless⟩ ::= always(⟨String⟩) of(⟨Term⟩*)
12   derive(WithBody).to(RuleHead).and(RuleBody).
13     // ⟨WithBody⟩ ::= ⟨RuleHead⟩ ⟨RuleBody⟩
14   derive(RuleHead).to(infer.with(String)).and(of.many(Term)).
15     // ⟨RuleHead⟩ ::= infer(⟨String⟩) of(⟨Term⟩*)
16   derive(RuleBody).to(FirstClause).and(oneOrMore(AdditionalClause)).
17     // ⟨RuleBody⟩ ::= ⟨FirstClause⟩ ⟨AdditionalClause⟩*
18   derive(FirstClause).to(when.with(String)).and(of.many(Term)).
19     // ⟨FirstClause⟩ ::= when(⟨String⟩) of(⟨Term⟩*)
20   derive(AdditionalClause).to(and.with(String)).and(of.many(Term)).
21     // ⟨AdditionalClause⟩ ::= and(⟨String⟩) of(⟨Term⟩*)
22   derive(Term).to(and.with(String)).and(of.many(Term));
23     // ⟨Term⟩ ::= l(⟨String⟩)| v(⟨String⟩)
```

FLING offers its own DSL, written in fluent API style, for grammar specification. The following notes, along with code comments in the listing should make this language clear: Token **`derive`** in FLING's DSL denotes a grammar derivation; the information it carries is the left hand side of a derivation rule. Information attached to token **`to`** (that follows **`derives`**) is the first element in the right hand side of the rule. In writing **`to(infer.with(String))`** we use informational token **`to`** to specify that the grammar admits an **`infer`** token with information of type **`String`**. Similarly, writing **`and(of.many(Term)`** we use informational token **`and`** to specify that the grammar admits an **`of`** token with information that consists of any number of occurrences of symbol **`Term`**.

5.2 Code generation

Having created a BNF description of DATALOG as in List. 6, the execution of **`bnf.generate()`** will make FLING generate all the definitions required for the code in List. 5 to compile correctly.

Moreover, FLING will generate definitions that make it possible to analyze DATALOG programs produced by the fluent API. Variable **`program`** of type **`Datalog`** produced in List. 5 is an abstract syntax tree (AST) of the program.

As part of the code generation process, similarly to JAMOOS [7] and SOOP [6] FLING implements an AST class for every symbol in the grammar specification. These classes uses lists for repetitions, omit vacuous tokens, and may stand in inheritance relation: The sub-expression

```
specialize(Statement).into(Fact, Query, Rule)
```

in List. 6 specifies not only grammatical alternation but also inheritance, i.e., class **`Fact`** inherits from abstract class **`Statement`**.

Along with these AST classes, FLING generates a template of an AST visitor that clients may use for processing a DSL program supplied in fluent API form. List. 7 demonstrates the use of this visitor to actually run from within JAVA the DATALOG program of List. 4.

Listing 7 `ASTVisitor` running DATALOG programs created via fluent API using Jatalog.

```
1   ASTVisitor runner() {
2     Jatalog j = new Jatalog();
3     return new ASTVisitor(DatalogAST.class) {
4       public boolean visit(Fact f) throws DatalogException {
5         j.fact(f.fact, f.of);
6         print(f);
7         return true;
8       }
9       public boolean visit(Bodyless r) throws DatalogException {
10        j.rule(Expr.expr(r.always, r.of1));
11        print(r);
12        return true;
13      }
14      public boolean visit(WithBody r) throws DatalogException {
15        j.rule(Expr.expr(r.infer, r.of1), getExprRightHandSide(r));
16        print(r);
17        return true;
18      }
19      public boolean visit(Query q) throws DatalogException {
20        print(q);
21        print(j.query(Expr.expr(q.query, q.of)));
22        return true;
23      }
24    };
25  }
```

Our demonstration of DATALOG employs Jatalog[4], an open source DATALOG engine to make the DATALOG embedding complete. The first three methods in the visitor object returned by List. 7 store (and print) facts and rules in the Jatalog engine. The last method asks the engine to compute the query, and then prints the result.

When this visitor is applied to variable **`program`** we obtain the output of shown in List. 8.

Listing 8 Output of List. 7 when run on the embedded DATALOG program produced by the fluent API of List. 5.

```
1   Jatalog:Fling$ parent(john,bob).
2   Jatalog:Fling$ parent(bob,donald).
3   Jatalog:Fling$ ancestor(A,B) :- parent(A,B).
4   Jatalog:Fling$ ancestor(A,B) :- parent(A,C), ancestor(C,B).
5   Jatalog:Fling$ ancestor(john,X)?
6   ---[{X=bob}, {X=donald}]
```

6 Discussion and Further Work

We showed that any deterministic pushdown automaton can be emulated by a realtime device using the tree encoding data structure. The tree encoding is exponential in the input size, but has a compact DAG representation. An interesting topic in theory of automata is investigation of this data structure, comparing its computational power with that of a stack. For example, one can ask whether PDAs, deterministic or not, can recognize more languages if the pushdown structure is replaced with that of a tree or a DAG.

A crucial observation in our construction is that the tree data structure can be written as a multi-level instantiation of a generic datatype, and that the kind of tree transformations applied in the emulator can also be written as the return type of methods in this data type. This observation was demonstrated in the automata compiler, which in turn made possible the theoretical result of encoding DCFLs in an unbounded polymorphic type system.

The automata compiler was employed in FLING– the first general and practical compiler-compiler of fluent APIs. FLING can be easily extended to support other programming languages, including ML [17], C#, and C++ . With some engineering effort FLING can be extended to LR(1) grammars as supported by YACC does.

[4] `https://github.com/wernsey/Jatalog`

FLING is weaker in its expressive power than the DPDA compiler, supporting only LL(1) grammars. However, we observed the construction in FLING is slightly more useful, in that auto-completion is more accurate. The fluent API problem, as defined here, does not concern itself with auto completion. It would be interesting to study a version of the in which an auto completion feature is required.

In a case study, we showed how FLING is used for embedding DATALOG in JAVA, and how such programs can be written and run from within JAVA.

Our description of the embedding, FLING itself, and of the automata compiler also demonstrate the usability of fluent API and the specific syntax flavor they induce. See listings 1, 2, 5 and 6. In the context of this style, we coined the term " informational tokens". Unlike traditional lexical analyzers, in FLING information that these tokens carry is not restricted to plain literals and may be defined by their own grammar. In this respect, FLING supports nested and even recursive grammars: The information that a token carries may be defined by the grammar in which the token participates.

We believe that the theoretical result may be also useful in designing extensible languages and languages whose syntax may be extended by their programs. Designers of such languages may choose to use the type system instead of a dedicated parsing engine for the unknown portion of their grammar.

In our implementation we encountered a weakness of the implementation of the to the discussion type system in the `javac` compiler. As noticed previously by Gil and Levy [5], this compiler represents instantiated generics by value, and admits no sharing.

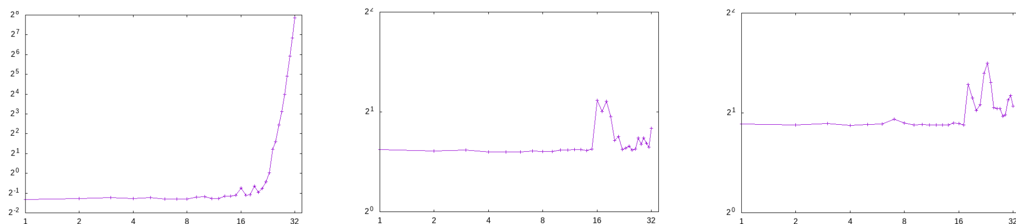
Consider for example the test program in List. 9.

Listing 9 Fluent interfaces exponential type complexity test case.

```
1 public class BinaryTreeTypeTest {
2   interface Binary<Left,Right> {
3     Binary<Binary<Left,Right>,Binary<Left,Right>> b();
4   }
5   public static void main(String[] args) {
6     System.out.println(((Binary<?,?>)null).b().b().b()....b()); // Use chain of length n
7   }
8 }
```

Type **`interace Binary`** is a generic type. Method **`b()`** in this type is such that in a given instantiation **`Binary`**, it returns an instantiation of this type which is twice the size. Thus, a sequence of n fluent calls to **`b()`**, as in function **`main`** in the figure, will return an instantiation of **`Binary`** of size 2^n.

Fig. 9 shows the compile time of List. 9 as a function n on various compilers. Measurements were conducted on JAVA 1.8.0_131, ECJ 3.11.1 and GCJ 6.3.0.



(a) javac compiler (Oracle). (b) Eclipse Java compiler (ECJ). (c) Gnu JAVA compiler GCJ.

Figure 9 Compilation time in seconds of **`BinaryTreeTypeTest`** List. 9 vs. length of fluent API call chain on various compilers.

As can bee seen in the figure, compilation time grows exponentially on `javac`. With ECJ and GCJ, these remain constant. Perhaps this work would encourage the makers of `javac` to use the compact DAG representation of types demonstrated above in Fig. 5.

References

1. Ken Arnold and James Gosling. *The* JAVA *Programming Language*. Addison Wesley, 1996.
2. Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic programming and databases*. SVNY, 1990.
3. Bruno Courcelle. On jump-deterministic pushdown automata. *Theory of Computing Systems*, 11(1):87–109, 1977.
4. Pierre Deransart, Laurent Cervoni, and AbdelAli Ed-Dbali. *Prolog: The Standard: reference manual*. Springer-Verlag, 1996.
5. Yossi Gil and Tomer Levy. Formal language recognition with the Java type checker. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.
6. Yossi Gil and David H Lorenz. SOOP – A Synthesizer of an Object-Oriented Parser. *TOOLS'95 Europe*, pages 81–96, 1995.
7. Yossi Gil and Yuri Tsoglin. JAMOOS - A Domain-Specific Language for Language Processing. *Journal of Computing and Information Technology*, 9(4):305–321, 2001.
8. Radu Grigore. Java generics are Turing complete. In Andrew D. Gordon, editor, *(POPL'17)*, pages 73–85, 2017.
9. Zvi Gutterman. Symbolic Pre-computation for Numerical Applications. Master's thesis, Technion, 2004.
10. Matthew Harren, Mukund Raghavachari, Oded Shmueli, Michael G Burke, Rajesh Bordawekar, Igor Pechtchanski, and Vivek Sarkar. XJ: facilitating XML processing in Java. In *Proceedings of the 14th international conference on World Wide Web*, pages 278–287. ACM, 2005.
11. Anders Hejlsberg, Scott Wiltamuth, Peter Golde, and Mads Torgersenby. *The C# Programming Language*. Addison-Wesley Publishing Company, 2nd edition, 2003-10.
12. ISE. *ISE EIFFEL The Language Reference*. ISE, 1997.
13. Donald E Knuth. On the translation of languages from left to right. *Information and control*, 8(6):607–639, 1965.
14. Tomer Levy. A Fluent API for Automatic Generation of Fluent APIs in Java. Master's thesis, Technion, 2016.
15. Erik Meijer and Brian Beckman. Xlinq: Xml programming refactored (the return of the monoids). In *Proceedings of XML*, volume 5, 2005.
16. Erik Meijer, Wolfram Schulte, and Gavin Bierman. Unifying tables, objects, and documents. In *Workshop on Declarative Programming in the Context of Object-Oriented Languages*, pages 145–166. Citeseer, 2003.
17. R. Milner and M. TofteD. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
18. Tomoki Nakamaru, Kazuhiro Ichikawa, Tetsuro Yamazaki, and Shigeru Chiba. Silverchain: a fluent API generator. *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences - GPCE 2017*, pages 199–211, 2017. URL: `http://dl.acm.org/citation.cfm?doid=3136040.3136041`.
19. Bjarne Stroustrup. *The C++ programming language*. Pearson Education India, 2000.
20. Peter Thiemann. A typed representation for HTML and XML documents in Haskell. *Journal of functional programming*, 12(4-5):435–468, 2002.
21. N. Wirth. The programming language Pascal. *Acta informatica*, 1:35–63, 1971.
22. Hao Xu. EriLex: an embedded domain specific language generator. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 192–212. Springer, 2010.