Visibly pushdown automata

Sebastian Muskalla

June 27, 2017

This document is based on the paper:

Visibly pushdown languages
R. Alur and P. Madhusudan
In: *Proceedings of STOC 2004*

It is available here:

- Conference version:
  madhu.cs.illinois.edu/stoc04.pdf
- Full version:
  dept-info.labri.fr/ fleury/Formation_Doctorale_2006/papers/AM04.pdf

TU Braunschweig

Summer term 2017

1. Motivation

So far, we have mostly restricted ourselves to finite automata (for finite and infinite words and for finite trees) in this lecture. These automata have no storage besides their finitely many control states.

This is of course a heavy restriction on the computational power of these devices. They are not even able to do unbounded counting (e.g. to recognize $\{a^n b^n \mid n \in \mathbb{N}\}$).

Therefore, we are interested in a more powerful model of computation. **Pushdown automata (PDA)** use a stack as storage. The stack is unbounded, but it can only be used in a FIFO manner. This allows PDAs to recognize the language $a^n b^n$ mentioned above. Furthermore, PDAs are also able to deal with nesting. For example, the language of all well-bracketed expressions over (,), [and] is recognizable by a PDA.

In verification, automata are used to describe the behavior of programs: each possible execution should correspond to one word in the language of the automaton. PDAs can be used to model the behavior of recursive programs. The problem with recursive programs is that after the call of a function is processed, the program has to return to the point from which the call was mode. PDAs can implement this using their stack as **call stack**: When a function call is made, the address of the callee is pushed onto the stack. After the call, this address can be popped and used to return to the correct part of the code.

It is well-known that PDAs accept exactly the **context-free languages (CFL)**, the languages definable by context-free grammars. The context-free languages share some nice algebraic properties with the regular languages: They are a so-called full trio, meaning they are closed under regular intersection, homomorphisms and inverse homomorphisms. They are also closed under union, concatenation, and Kleene star.

Unfortunately, they are not closed under complement and intersection. Furthermore, they don't enjoy good algorithmic properties: While emptiness and membership are decidable, intersection-emptiness, universality, inclusion and equivalence are undecidable.

In contrast to finite automata, where determinism and non-determinism are equally powerful, **deterministic pushdown automata (DPDA)** are strictly less powerful than non-deterministic PDAs. They define the so-called **deterministic context-free languages (DCFL)**. The set of all even-length palindromes

$$\{ww^{\text{reverse}} \mid w \in \{a,b\}^*\}$$

is context-free, but not deterministic context-free: A PDA needs to guess the middle of the input to be able to guess the correct point for switching from pushing symbols onto the set to popping them.

Deterministic context-free languages are closed under complement (by swapping the final states of a DPDA), but neither under union nor intersection. In 2001, Sénizergues showed that language equivalence (and thus universality) for DPDAs is decidable, for which he was awarded the Gödel Prize in 2002. Nowadays, there are several papers (by him and other researchers) that contain algorithms for language equivalence. So far, all algorithms have primitive-recursive, but non-elementary complexity. (This means that the running time of the algorithm is given by a tower of exponentials, where the height of the tower depends on the input size.) Inclusion and intersection-emptiness remain undecidable even for DPDAs.

It may seem surprising that even for two deterministic automata, it is not possible to decide whether their intersection is empty. One might think that the usual cross product construction does the job. Since the automata under consideration have stacks, we would also need to apply the cross product construction to the stack. Taking the product of the stack alphabets as the new stack alphabet is easy, but there is a big problem: For a fixed word, the two (unique) runs of the automata might produce vastly different looking stacks. If one automaton pushes while the other pops, the second automaton will have a much smaller stack. There is no way to simulate this using only one stack (e.g. if we use filler symbols, we will not be able to access the topmost symbol of the smaller stack).

One could say that the problem with (deterministic) pushdown automata is that even for a fixed word, it is not clear how the stack of the automaton will look like after processing the word without actually simulating the run. Visibly pushdown automata fix this problem by making the shape of the stack uniquely determined by the input word.

This makes them more expressive than finite automata, but less expressive than DPDAs. One the one hand, they are still powerful enough to model executions of recursive programs, since the call (push) and the return (pop) of functions can be marked with special letters. On the other hand, they enjoy good algorithmic properties that are similar to those of finite automata.

2. Visibly pushdown automata

2.1 Definition
A **visibly alphabet** Σ^V is an alphabet together with a decomposition

$$\Sigma = \Sigma_c \dot\cup \Sigma_r \dot\cup \Sigma_i$$

into

- **call-symbols** $a \in \Sigma_c$ (symbols for which the automaton should push),
- **return-symbols** $a \in \Sigma_r$ (symbols for which the automaton should pop), and
- **internal symbols** $a \in \Sigma_i$ (symbols for which the automaton should not modify the stack). They are also called local symbols in the literature.

A visibly pushdown automaton is essentially a PDA over a visibly alphabet whose transition relation respects the decomposition of the alphabet. Still, we give a precise definition to fix the syntax that we be used in the rest of the lecture.

2.2 Definition
A (non-deterministic) **visibly pushdown automaton (VPA)** A is a tuple

$$A = (\Sigma^V, Q, Q_0, Q_F, \Gamma, \delta)\,,$$

where

- Σ^V with $\Sigma = \Sigma_c \dot\cup \Sigma_r \dot\cup \Sigma_i$ is a visibly alphabet,
- Q is a finite set of control states,
- $Q_0 \subseteq Q$ is the set of initial states,
- $Q_F \subseteq Q_F$ is the set of final states,
- Γ is the stack alphabet containing the special symbol $\bot \in \Gamma$ indicating the bottom of stack, and
- $\delta = (\delta_c, \delta_r, \delta_i)$ is a tuple, where

$$\begin{aligned}\delta_c &\subseteq Q \times \Sigma_c \times (\Gamma \setminus \{\bot\}) \times Q\\ \delta_r &\subseteq Q \times \Sigma_r \times \Gamma \times Q\\ \delta_i &\subseteq Q \times \Sigma_i \times Q\,.\end{aligned}$$

We write transitions $(q, a, \gamma, q') \in \delta_c$ as $q \xrightarrow{\text{push}\,\gamma}_a q'$. Their meaning is that upon seeing call-symbol a in state q, automaton A can go to state q' while pushing γ onto the stack.

We write transitions $(q, a, \gamma, q') \in \delta_r$ as $q \xrightarrow{\text{pop}\,\gamma}_a q'$. Their meaning is that upon seeing return-symbol a in state q, automaton A can go to state q' while γ from the stack.

We write transitions $(q, a, q') \in \delta_i$ as $q \rightarrow_a q'$. Their meaning is that upon seeing internal symbol a in state q, automaton A can go to state q' (without modifying the stack).

We will now formalize this intuition by providing the semantics of VPAs.

2.3 Definition
A **configuration** of VPA A is a tuple $c = (q, \sigma)$ where

- $q \in Q$ is a control state, and
- $\Sigma \in \bot.(\Gamma \setminus \{\bot\})^*$ is a stack content ending in the bottom symbol. (We display the top-of-stack on the right-hand side end of the word.)

The transition relation δ of A induces a non-deterministic transition relation among configurations.

1. For $a \in \Sigma_c$, we have
$$(q, \sigma) \rightarrow (q', \sigma.\gamma)$$
if $q \xrightarrow{\text{push}\,\gamma}_a q'$.
2. For $a \in \Sigma_r$, we have
$$(q, \sigma.\gamma) \rightarrow (q', \sigma)$$
if $q \xrightarrow{\text{pop}\,\gamma}_a q'$.
3. For $a \in \Sigma_r$, we have
$$(q, \bot) \rightarrow (q', \bot)$$
if $q \xrightarrow{\text{pop}\,\bot}_a q'$.
4. For $a \in \Sigma_i$, we have
$$(q, \sigma) \rightarrow (q', \sigma)$$
if $q \rightarrow q' \in \delta_i$.

Sometimes we make explicit the letter that was red and write $(q, \sigma) \rightarrow_a (q', \sigma')$.

Let $w = a_1 \dots a_n \in \Sigma^*$ be a word. A **run** of A on w is a sequence of configurations

$$(q_0, \bot) \rightarrow_{a_1} (q_1, \sigma_1) \rightarrow_{a_2} \dots \rightarrow_{a_n} (q_n, \sigma_n)$$

where $q_0 \in Q_0$ is an initial state.

It is **accepting** if $q_n \in Q_F$ is a final state.

The language of A is the set of all words over Σ that have an accepting run,

$$\mathcal{L}(A) = \left\{ w \in \Sigma^* \;\middle|\; (q_0, \bot) \to_w (q_f, \sigma), q_0 \in Q_0, q_f \in Q_F \right\}.$$

2.4 Remark

- Note that VPAs accept using final states. This in particular means that the stack maybe non-empty after a word has been processed. This means there can be **unmatched calls** in a word, calls for which a symbol was pushed that was not popped later during the run.

  Making VPAs accept only on the empty stack would be a restriction on their computational power.

- Transitions $q \xrightarrow{\text{pop}\,\gamma}_a q'$ do not actually pop $\bot$ from the stack. These are transitions that can just be fired if the stack is currently empty.

  This in particular means that in a word, there can be **unmatched returns**, returns for which no corresponding call exists.

- One could allow internal ε transitions (but no ε-labeled push or pop transitions). The proof that such transitions can be eliminated in finite automata carries over to VPAs.

- According our definition, VPAs are not allowed to **peek** (read without modifying) at the the top-of-stack on internal transitions. This is not a restriction, as the top-of-stack can be encoded into the control state.

We define the class of languages recognizable by VPAs.

2.5 Definition
A language $\mathcal{L} \subseteq \Sigma^*$ is a called a **visibly pushdown language (VPL)** if there is a decomposition of Σ into a visibly alphabet Σ^V and a VPA A over Σ^V with $\mathcal{L}(A) = \mathcal{L}$.

We call a language $\mathcal{L} \subseteq \Sigma^*$ is a a **visibly pushdown language (VPL) with respect to the the visibly alphabet** Σ^V (that is a decomposition of Σ) if there is a VPA A over Σ^V with $\mathcal{L}(A) = \mathcal{L}$.

Note that in the first part of the definition, we allow an arbitrary decomposition, while we fix it in the second part. We will see in the next section why this is needed in some situations.

2.6 Remark
Since every VPA can be seen as PDA, is is clear that VPL is a subclass of CFL.

The relationship to DCFL is not yet clear.

3. Some closure properties

We now start to study the closure properties of VPA-languages.

3.1 Lemma
Let Σ^V be a visibly alphabet. VPLs with respect to Σ^V are effectively closed under union, intersection, concatenation and Klenee-star. Given two VPAs A_1, A_2 over Σ^V, one can construct VPAs for

$$\mathcal{L}(A_1) \cap \mathcal{L}(A_2) \qquad \mathcal{L}(A_1) \cup \mathcal{L}(A_2) \qquad \mathcal{L}(A_1).\mathcal{L}(A_2) \qquad \mathcal{L}(A_1)^* .$$

Proving the lemma can be done using essentially the same techniques that were used for finite automata.

3.2 Remark

- For union, intersection and concatenation, it is important that the VPAs agree on the decomposition of the alphabet.
- VPLs are also closed under homomorphisms that respect the decompositions of the alphabet.
- Effectively closed means that we cannot just prove that e.g. the union is a VPL, but we can actually construct the corresponding automaton.

  This is not clear in general. For example, consider the so-called lossy channel systems. One the one hand, one can prove that the downward closure of their languages are regular languages. One the other hand, one can prove that there can be no algorithm that actually computes an NFA representing the downward closure.

4. Determinization and complementation

It remains to proof that VPLs are effectively closed under complement. To this end, we proceed as for finite automata: We prove that VPAs are determinizable, closure under complementation is then an easy corollary.

4.1 Definition
A VPA $A = (\Sigma^v, Q, Q_0, Q_F, \Gamma, \delta)$ is **deterministic (DVPA)** if

- it has exactly one initial state, $\left|Q_0\right| = 1$, and
- for each $a \in \Sigma_i$, there is at most one transition of the form $q \rightarrow_a q' \in \delta_i$,
- for each $a \in \Sigma_c$, there is at most one transition of the form $q \xrightarrow{\text{push}\,\gamma}_a q' \in \delta_c$, and
- for each $a \in \Sigma_r$ and each $\gamma \in \Gamma$, there is at most one transition of the form $q \xrightarrow{\text{pop}\,\gamma} q' \in \delta_r$.

4.2 Remark
Note the difference between push- and pop-transitions:

For push transitions, the automaton decide the symbol to push, thus, it has to be unique in a DVPA.

For pop transitions, there is only at most one enabled pop transition, namely the transition that pops the current top-of-stack (or $\perp$ if the stack is empty). Thus, we allow one transition per stack symbol.

4.3 Theorem
Let A be a VPA over Σ^v. One can construct a DVPA A^{det} over Σ^v with $\mathcal{L}(A) = \mathcal{L}\left(A^{det}\right)$.

Before we show how to construct A^{det}, we discuss the approach.

A naive idea would be the powerset construction, as for NFAs. The problem is that there is the stack. The shape (size) of the stack in each configuration is determined by the input word, but not its content.

Assume that at some point of the computation, A could either be in $(q_1, \perp ba)$ or in $(q_2, \perp ab)$. Using a powerset construction also for the stack alphabet and representing both configurations together as $(\{q_1, q_2\}, \perp\{a, b\}\{a, b\})$ is not valid: From this representation, it is not clear that $(q_1, \perp aa)$ is not a possible configuration of A at this point.

This means the determinized automaton would need to keep track of which possible stack content belongs to which computational. Unfortunately, the number of computations grows together with the input words. This means that the bounded storage provided by the control states is insufficient. Even with the stack, it is hard to realize this in a naive way.

We now explain how the construction solves this problem.

The crucial idea is to not really execute push moves made by A when a call symbol a' occurs: As highlighted in the above example, it would be impossible to match the correct pop move later. Instead, the automaton A^{det} waits until the corresponding return symbol a occurs. It then checks which transitions $q \xrightarrow{\text{push}\,\gamma}_{a'} q_1$ the original automaton A could have made when reading a', and matches them with the transitions $q_2 \xrightarrow{\text{pop}\,\gamma}_{a} q'$ for the current symbol a that pop the *same* stack symbol γ.

There are two difficulties in implementing this:

Firstly, when reading a, the automaton needs to be able to access a' and its old state in which it was when a' was read. To this end, the automaton stores a' and the current state on the stack when a' is read. Since a is the return corresponding to the call a', this means that this stored information will be the top-of-stack when a is read and can be accessed.

Secondly, the combination of push-transition $q \xrightarrow{\text{push}\,\gamma}_{a'} q_1$ and pop-transition $q_2 \xrightarrow{\text{pop}\,\gamma}_{a} q'$ is only valid if there is a computation leading from q_1 to q_2 for the word that was read between a and a'. To be able to check this, the automaton stores a **summary relation** in its state, meaning it stores for each pair of states (q, q') whether there is a computation going from q to q'.

We are now able to formally introduce the first part of the construction.

Construction of A^{det} (States):
Let $A = (\Sigma^v, Q, Q_0, Q_F, \Gamma, \delta)$ be the given PDA. We define

$$A^{det} = (\Sigma^v, Q', Q_0', Q_F', \Gamma', \delta')$$

where

- $Q' = \mathcal{P}\big(Q \times Q\big) \times \mathcal{P}\big(Q\big)$

  This means a state $(S, R) \in Q'$ consists of a set of states $R \subseteq Q$ and a relation between states $S \subseteq Q \times Q$. The set R implements the usual powerset construction, it contains all states of A in which A could be after reading the prefix of the word so far. The set S is the summary mentioned above. A pair (q, q') is contained in S if there is a computation from q to q' for a certain part of the word. We will make this precise later.

- $Q'_0 = \{(\mathrm{Id}, Q_0)\}$ where Id denotes the relation $\mathrm{Id} = \{(q, q) \mid q \in Q\}$.

  Initially, the initial states Q_0 of A are possible. Furthermore, we start with the trivial relation in which each state is only related to itself, since no computation has happened yet.

- $Q'_F = \{(S, R) \mid R \cap Q_F \neq \varnothing\}$

  We require that at least one possible state is final. The summary is ignored.

- $\Gamma' = \mathcal{P}(Q \times Q) \times \mathcal{P}(Q) \times \Sigma_c$

  A stack symbol is of the shape (S, R, a_c), where $(S, R) \in Q'$ is a state, and $a_c \in \Sigma_c$ is a call symbol. As already explained, when a call symbol a_c occurs, A^{det} will store its current state as well as the letter a_c on the stack.

Before formally defining the transition relation, let us consider an example to get a better understanding.

4.4 Example
Let $w = w_1 a_1 w_2 a_2 w_3$ be a word where

- In w_1, all calls have a matching return. (There may be unmatched returns.)
- a_1, a_2 are call with no matching return.
- The words w_2, w_3 are well-matched, i.e. all calls have a matching return, and all returns have a matching call. (A return without matching call in w_2 would match a_1, similar for w_3 and a_2).

After reading w, assume that the state of A^{det} is (S, R) and its stack content is

$$\bot\, (S_1, R_1, a_1)\, (S_2, R_2, a_2)\,.$$

Stack symbol (S_1, R_1, a_1) was pushed when reading a_1, (S_2, R_2, a_2) was pushed when reading (S_2, R_2, a_2). All other calls in w have a matching return, thus, the stack has size 2.

As already explained, the component R of the state implements the usual powerset construction. It is the set of states in which A could be after reading the prefix of the input word that has already been processed.

Since R_1 and R_2 were (a part of) the states when a_2 resp. a_2 were, read, we have that

- R is the set of states in which A could be after reading w,
- R_1 is the set of states in which A could be after w_1, and
- R_2 is the set of states in which A could be after $w_1 a_1 w_2$.

$$R = \left\{ q' \;\middle|\; (q_0, \bot) \to_w (q', \sigma) \text{ for some } q_0 \in Q_0 \right\},$$
$$R_1 = \left\{ q' \;\middle|\; (q_0, \bot) \to_{w_1} (q', \sigma) \text{ for some } q_0 \in Q_0 \right\},$$
$$R_1 = \left\{ q' \;\middle|\; (q_0, \bot) \to_{w_1 a_1 w_2} (q', \sigma) \text{ for some } q_0 \in Q_0 \right\}.$$

It is now time to explain the purpose of the summaries S. Whenever the automaton reads a call-letter that has no matching return yet, it starts to track all possible computations from this point on. For example, a_2 is the latest call without matching return in w. This means the summary S in the current state of A^{det} is the summary of all possible computations on w_3.

$$S = \left\{ (q, q') \;\middle|\; (q, \bot) \to_{w_3} (q', \bot) \right\}$$

Actually, the stack is not empty after reading the call letter a_1. But since w_3 does not contain a return corresponding to a_1, the computation will not access the lower parts of the stack. Therefore, we may fix the stack content of configuration $(q, \bot)$ to be empty.

We will need the summary as soon as the return corresponding to a_c occurs. This means that all symbols that have been pushed onto the stack during w_3 have also been popped. Therefore, we are interested in configurations $(q', \bot)$ where the stack is also empty.

Whenever we see a newer unmatched call, we store the old summary onto the stack. This will allow us to resume them as soon as we see the corresponding return to the newer call.

In the example, S_1 is the summary for the beginning of the word until the first unmatched call a_1,

$$S_1 = \left\{ (q, q') \;\middle|\; (q, \bot) \to_{w_1} (q', \bot) \right\}.$$

When reading a_2, a_2 is now the newest unmatched call, so S_1 gets stored onto the stack. The automaton starts a new summary for the computation on w_2,

$$S_2 = \left\{ (q, q') \;\middle|\; (q, \bot) \to_{w_2} (q', \bot) \right\}.$$

When reading a_3, S_2 gets stored, and a new summary (that later will become S) is started.

Assume the automaton would now read a new call-symbol a_3. This is now the newest call with unmatched return, so the automaton would store the old state and summary onto the stack, together with the symbol a_3. This means it would push (S, R, a_3). The new state consists of R' updated using all possible transitions of A for a_3, and of the trivial summary Id. Since we have not read any letter after a_3 yet, the new summary should be the summary for the empty word ε, and indeed we can think of having transitions $(q, \sigma) \to_\varepsilon (q, \sigma)$ among configurations.

Assume we process some internal letters afterwards. The possible transitions of A for these letters would be used to update summary and possible states stored in the control state. Assume that $(\tilde{S}, \tilde{R})$ is the resulting control state of A^{det}.

When we read the return symbol b, the pop corresponding to the push for a_3 should happen. The topmost stack symbol is (S, R, a_3), i.e. the old control state together with the letter a_3 We use a_3 to determine all transitions $q \xrightarrow{\text{push}\,\gamma}_{a_3} q_1$. Here q should be a possible state of A before reading a_3, we can ensure this by requiring $q \in R$. We look the the transitions for b that pop the same symbol, i.e. $q_2 \xrightarrow{\text{pop}\,\gamma}_{b} q'$. To ensure that there is a possible computation in between leading from q_1 to q_2, we require $(q_1, q_2) \in \tilde{S}$, as S is the summary for the word that has been processed since the occurrence of a_3.

The new control state will be (S', R'), where R' is the set of all states q' as above. Summary S' is obtained by updating S (which was stored on the stack). Since a_3 now has a matching return, we can throw away $\tilde{S}$, and continue S, as a_2 is again the newest call without matching return.

Construction of A^{det} (Transition relation):

We consider each type of symbol separately.

Internal symbols:
For $a \in \Sigma_i$, we have

$$(S, R) \xrightarrow{a} (S', R') \in \delta_i'$$

where

$$S' = \left\{(q, q'') \;\middle|\; \exists q' \colon (q, q') \in S \exists q' \xrightarrow{a} q'' \in \delta_i\right\}, R' \quad = \left\{q'' \;\middle|\; \exists q' \in R \colon q' \xrightarrow{a} q'' \in \delta_i\right\}.$$

Here, we update summary and possible states as usual for the powerset construction.

Call symbols:
For $a \in \Sigma_c$, we have

$$(S, R) \xrightarrow{\text{push}(S,R,a)}_{a} (\text{Id}, R') \in \delta_c'$$

where

$$R' = \left\{q'' \;\middle|\; \exists q' \in R \exists \colon q' \xrightarrow{\text{push}\,\gamma}_{a} q'' \in \delta_c\right\}.$$

We store the old state and the letter that was just read on the stack. The set of possible states is updated using all possible transitions as usual in the power set construction. We start a new summary, since a is now the newest call without matching return.

Return symbols:
For $a \in \Sigma_r$, we have

$$(S, R) \xrightarrow{\text{pop}(S', R', a')}_a (S'', R'') \in \delta'_r .$$

Before defining (S'', R''), we define the relation

$$\text{Update} = \left\{ (q, q') \;\middle|\; \exists q_1, q_2 \in Q, \gamma \in \Gamma \colon q \xrightarrow{\text{push}\,\gamma}_{a'} q_1 \in \delta_c, (q_1, q_2) \in S, q_2 \xrightarrow{\text{pop}\,\gamma}_a q' \in \delta_r \right\}.$$

It contains all computations that consist of a push for the letter a' stored on the stack, a pop for the letter a that we read now that pops the same symbol, and a computation in between.

Computation of A:

$$q \xrightarrow{\text{push}\,\gamma}_{a'} q_1 \underbrace{\rightsquigarrow}_{\in S} q_2 \xrightarrow{\text{pop}\,\gamma}_a q'$$

Computation of A^{det}:

$$(S', R') \xrightarrow{\text{push}(S', R', a')}_{a'} (\text{Id}, \tilde{R}) \rightsquigarrow (S, R) \xrightarrow{\text{pop}(S', R', a')}_a (S'', R'')$$

We use the old set of possible states R' (stored on the stack) and Update to compute the set new possible states q' contained in R'':.

$$R'' = \left\{ q'' \;\middle|\; \exists q' \in R' \colon (q', q'') \in \text{Update} \right\}.$$

Furthermore, we continue the old summary S' stored on the stack. We update it using the relation Update.

$$S'' = \left\{ (q, q'') \;\middle|\; \exists q' \colon (q, q') \in S', (q', q'') \in \text{Update} \right\}.$$

Finally, we also need a transition for unmatched returns when we pop $\perp$.

$$(S, R) \xrightarrow{\text{pop}\,\perp}_a (S'', R'') \in \delta'_r$$

where

$$S' = \left\{ (q, q'') \;\middle|\; \exists q' \colon (q, q') \in S \exists q' \xrightarrow{\text{pop}\,\perp}_a q'' \in \delta_i \right\},$$
$$R' = \left\{ q'' \;\middle|\; \exists q' \in R \colon q' \xrightarrow{\text{pop}\,\perp}_a q'' \in \delta_i \right\}.$$

Since there is no matching needed in this case, this is again as usual for the powerset construction.

4.5 Remark
We do not prove the soundness of the our construction.

To formally do this, one would need to show that if A^{det} goes to state (S, R) while processing word w, then A can go to any $q' \in R$.

This cannot be directly proving by induction. One would need to strengthen the induction hypothesis by requiring that all states (S, R) that occur as state resp. on the stack have certain properties. These properties are as explained in the example above:

- Each stored R' is the set of possible states after reading the prefix up to the point when R was stored.
- Each stored S' is the summary for the word that is read between two unmatched calls.

One can then prove that this invariant is maintained by every transition of A^{det}.

4.6 Remark
As for NFAs, the determinization might lead to an exponential blowup. Since we store summaries, the exponent is even quadratic in the old number of states.

Assume A has n states. Then A^{det} has $2^{\mathcal{O}(n^2)}$-many states and used $2^{\mathcal{O}(n^2)}\left|\Sigma_c\right|$-many stack symbols.

4.7 Corollary
Let Σ^v be a visibly alphabet. VPLs with respect to Σ^v are effectively closed under complement: Given a VPA A over Σ^v, one can construct a VPA $\overline{A}$ over with

$$\mathcal{L}\left(\overline{A}\right) = \Sigma^* \setminus \mathcal{L}\left(A\right).$$

Proof:
For the given VPA $A = (\Sigma^v, Q, Q_0, Q_F, \Gamma, \delta)$, we can construct its determinization

$$A^{det} = (\Sigma^v, Q', Q_0', Q_F', \Gamma', \delta').$$

We define $\overline{A}$ to be

$$\overline{A} = (\Sigma^v, Q', Q_0', Q' \setminus Q_F', \Gamma', \delta'),$$

i.e. A^{det}, where the final and the non-final states have been swapped. We have

$$\mathcal{L}\left(\overline{A}\right) = \Sigma^* \setminus \mathcal{L}\left(A^{det}\right) = \Sigma^* \setminus \mathcal{L}\left(A\right).$$

□

4.8 Corollary
VPL is a strict subclass of DCFL.

Proof:
Every VPA has a language-equivalent DVPA, which in turn can be seen as DPDA. Thus, it is clear that VPL is a subclass of DCFL.

The language

$$\left\{w\#w^{reverse} \;\middle|\; w \in \left\{a,b\right\}^*\right\}$$

of even-length palindromes, where the middle is marked using a special symbol, is a DCFL, but not a VPL.

Similarly, the language

$$\left\{a^n\#a^n \;\middle|\; n \in \mathbb{N}\right\}$$

is a DCFL, but not a VPL: for the letters a occurring before #, the automaton would need to push, but for the letters a occurring after # it would need to pop. □

5. Concluding remarks

The context-free languages can be embedded as VPA-languages over an extended alphabet.

5.1 Proposition
Let Σ be an alphabet. We define

$$\begin{aligned}
\Sigma^c &= \left\{a^c \;\middle|\; a \in \Sigma\right\} \\
\Sigma^r &= \left\{a^r \;\middle|\; a \in \Sigma\right\} \\
\Sigma^i &= \left\{a^i \;\middle|\; a \in \Sigma\right\} \\
\Sigma^v &= \Sigma^c \cup \Sigma^r \cup \Sigma^i,
\end{aligned}$$

i.e. we define three variants for each letter.

Furthermore, we define the projection $\mathrm{proj}\colon \Sigma^v \to \Sigma$ by $\mathrm{proj}(a^x) = a$ for $x in \left\{c,r,i\right\}$.

For each CFL $\mathcal{L} \subseteq \Sigma$, there is a VPL $\mathcal{L}' \subseteq \Sigma^v$ such that $\mathrm{proj}(\mathcal{L}') = \mathcal{L}'$.

The decision problems for VPLs are slightly harder than for regular languages.

	NFAs	VPAs	DPDAs	PDAs
Emptiness	NL-complete	P-complete	P-complete	P-complete
Inclusion	PSPACE-complete	EXP-complete	Undecidable	Undecidable
Universality, Equivalence	PSPACE-complete	EXP-complete	Decidable	Undecidable

Note that there is no real difference in running time between NFAs and VPAs: Unless PSPACE = P, PSPACE-complete problems cannot be solved within polynomial time.

There is a difference in space consumption: Unless PSPACE = EXP, solving EXP-complete problems needs more than polynomial space.

People have extended the research on VPLs in many directions:

- Logical characterization,
- Characterization via grammars and derivation trees,
- ω-VPL,
- and many more.

One of the authors of the original paper has a collection of papers concerning VPAs: madhu.cs.illinois.edu/vpa/