Programming Research Group

SAFETY IS NOT A RESTRICTION AT LEVEL 2 FOR STRING LANGUAGES

K. Aehlig
J. G. de Miranda
C.-H. L. Ong

PRG-RR-04-23

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

Abstract

Recent work by Knapik, Niwinski and Urzyczyn [KNU02] has revived interest in the connexions between higher-order grammars and higher-order pushdown automata. Both devices can be viewed as definitions for term trees as well as string languages. In the latter setting we recall the extensive study by Damm [Dam82], and Damm and Goerdt [DG86]. There it was shown that the language of a level-n higher-order grammar is accepted by a level-n higher-order pushdown automaton subject to the restriction of *derived types*, more recently rebranded as *safety*. We show that at level 2, if a string language is generated by an unsafe grammar, there is a (level-2, non-deterministic) safe grammar that generates the same language. Thus safety is not a restriction for level-2 string languages.

Contents

1 Introduction

The result we present in this paper concerns safety – a syntactic restriction introduced by Knapik *et al.* [KNU01, KNU02]. They define higher-order grammars as generators of (possibly infinite) term trees. They show that if a grammar is safe then the term tree produced by the grammar enjoys a decidable monadic second order (MSO) theory. Moreover, safety ensures that the term tree produced can also be accepted by a higher-order pushdown automaton of the same level, and vice versa. By providing two characterisations of a new infinite hierarchy of infinite trees with decidable MSO theories, their result pushes the frontier of decidability to new boundaries. (This hierarchy actually coincides with a Caucal Hierarchy of deterministic term trees [Cau02].) It is intriguing that safety *appears* to be key to such good algorithmic behaviour and desirable properties. In particular it leaves us with two natural open questions. First, is the restriction of safety essential for MSO decidability? Secondly, does safety really reduce the generating power of a grammar?

As indicated by Knapik *et al.*, it is interesting to note that the restriction of safety has previously appeared in the literature in another setting and under a different name. In the earlier papers of [Dam82, DG86] higher-order grammars and higher-order pushdown automata were introduced not as definitional devices for term trees, but as generators, respectively acceptors, of string languages. Thus Damm defined an infinite hierarchy of languages, called the OI-hierarchy, which some consider a more natural, infinite alternative to the Chomsky Hierarchy [Cho59]. Damm's grammars are rewrite relations over expressions that are required to be objects of "derived types". An analysis of his definition reveals that the constraint of "derived types" is equivalent to the requirement that all types be *homogeneous* and the grammar be *safe*, both in the sense of Knapik *et al.* Assuming the grammar makes use of only homogeneous types (which all definitions in the literature do), it follows that safety and derived types are equivalent. From now on, we will only use the term "safe".

Damm and Goerdt went on to show that if a language is generated by a safe grammar, then it can be accepted by a higher-order pushdown automaton of the same level (and vice versa). Another "good" property Damm illustrates of his safe grammars is that each level of the hierarchy is a full AFL. However, to date, no results exist for unsafe grammars. This is surprising – but perhaps less so as Damm's definition of a higher-order grammar was such that the safety restriction was "built-in" to it. However, in this paper our definition of a string-language generating grammar will follow Knapik *et al.* in that we will allow for both safe and unsafe grammars.

This paper is concerned with the second of the open problems mentioned at the end of the opening paragraph. We show that we can effectively transform a level-2 grammar (whether safe or not) into a (non-deterministic) level-2 pushdown automaton that accepts the same language. Thus it follows from [DG86] that every level-2 string language that is generated by an unsafe grammar can also by generated by a (level-2, non-deterministic) safe grammar. In this sense, we say that safety is not a restriction at level 2. As a bonus, we then apply our result to extract decidability results for level-2 unsafe term-tree generating grammars.

Related work

In a recent preprint [Blu04], Blumensath has given the first "pumping lemma" for languages accepted by higher-order pushdown automata. Following intricate surgeries on runs on an automaton, his work gives conditions under which they can be "pumped". These conditions are expressed in terms of the length of a given run and the size of the stacks of each configuration.

2 Higher-order grammars, safety and higher-order PDAs

In this section we introduce higher-order grammars (HOGs) and higher-order pushdown automata (HOPDAs). Both devices have appeared previously in the literature [Dam82, DG86, KNU02, Eng91], however, under two sets of definitions.

In [Dam82, DG86, Eng91], HOGs and HOPDAs are introduced as generators and acceptors of *string languages* respectively. Thus each HOG (HOPDA) generates (accepts) a language of strings. In contrast, Knapik *et al.* [KNU01, KNU02] introduce HOGs and HOPDAS as generators and acceptors of *term trees* respectively. In particular, each HOG (HOPDA) generates (accepts) exactly one term tree.

As stated in the introduction, our result concerns only the string-language setting. Therefore, unless otherwise stated, we will understand higher-order grammars and higher-order pushdown automata to be definitional devices for string languages. However, in Section 7 we will relate our result to the term-tree setting of [KNU01, KNU02].

2.1 Safe λ-Calculus

We shall consider *simple types* (or just *types* for short) as defined by the grammar $A ::= o \mid A \to A$. Each type A, other than the ground type o, can be uniquely written as $(A_1, \cdots, A_n, o)$ for some $n \geq 1$, which is a shorthand for $A_1 \to \cdots \to A_n \to o$ (by convention, $\to$ associates to the right). We define the *level*[1] of a type by $\mathsf{level}(o) = 0$ and $\mathsf{level}(A \to B) = \max(\mathsf{level}(A) + 1, \mathsf{level}(B))$. In the following we shall consider terms-in-context $\Gamma \vdash M : A$ of the simply-typed λ-calculus. Let Δ be a simply-typed alphabet i.e., each symbol in Δ has a simple type. We write $\mathcal{T}^A(\Delta)$ for the set of terms of type A built up from the set Δ understood as constant symbols, *without* using λ-abstraction.

Following [KNU02], we say that o is ***homogeneous***, and for $n \geq 1$, $A = (A_1, \cdots, A_n, o)$ is ***homogeneous*** just if $\mathsf{level}(A_1) \geq \mathsf{level}(A_2) \geq \cdots \geq \mathsf{level}(A_n)$, and each A_i is homogeneous. Assuming $A = (\underbrace{A_{11}, \cdots, A_{1l_1}}_{\overline{A_1}}, \cdots, \underbrace{A_{r1}, \cdots, A_{rl_r}}_{\overline{A_r}}, o)$ is homogeneous, we write

$$A \;=\; (\overline{A_1} \mid \cdots \mid \overline{A_r} \mid o)$$

to mean: all types in each partition (or sequence) $\overline{A_i} = A_{i1}, \cdots, A_{il_i}$ have the same level, and $i < j \iff \mathsf{level}(A_{ia}) > \mathsf{level}(A_{jb})$. Thus the notation organises the A_{ij}s into partitions according to their levels. Suppose $B = (\overline{B_1} \mid \cdots \mid \overline{B_m} \mid o)$. We write $(\overline{A_1} \mid \cdots \mid \overline{A_n} \mid B)$ to mean

$$(\overline{A_1} \mid \cdots \mid \overline{A_n} \mid \overline{B_1} \mid \cdots \mid \overline{B_m} \mid o).$$

The ***Safe λ-Calculus*** is a sub-system of the simply-typed λ-calculus. Typing judgements (or terms-in-context) are of the form

$$\overline{x_1} : \overline{A_1} \mid \cdots \mid \overline{x_n} : \overline{A_n} \vdash M : B$$

which is shorthand for $x_{11} : A_{11}, \cdots, x_{1r} : A_{1r}, \cdots \vdash M : B$. (We shall see shortly that for valid such judgements, $(\overline{A_1} \mid \cdots \mid \overline{A_n} \mid B)$ is a homogeneous type.) *Valid typing judgements* of the system are defined by induction over the following rules, where Δ is a given homogeneously-typed alphabet:

$$\frac{(\overline{A_1} \mid \cdots \mid \overline{A_n} \mid B) \text{ homogeneous} \qquad b : B \in \Delta}{\overline{x_1} : \overline{A_1} \mid \cdots \mid \overline{x_n} : \overline{A_n} \vdash b : B}$$

[1] This is sometimes referred to as *order* in the literature.

$$\frac{(\overline{A_1} \,|\, \cdots \,|\, \overline{A_n} \,|\, A_{ni}) \text{ homogeneous}}{\overline{x_1} : \overline{A_1} \,|\, \cdots \,|\, \overline{x_n} : \overline{A_n} \vdash x_{ni} : A_{ni}}$$

$$\frac{\overline{x_1} : \overline{A_1} \,|\, \cdots \,|\, \overline{x_{n+1}} : \overline{A_{n+1}} \vdash M : B}{\overline{x_1} : \overline{A_1} \,|\, \cdots \,|\, \overline{x_n} : \overline{A_n} \vdash \boldsymbol{\lambda}\overline{x_{n+1}} : \overline{A_{n+1}}.M : (\overline{A_{n+1}} \,|\, B)}$$

$$\frac{\Gamma \vdash M : (\overline{B_1} \,|\, \cdots \,|\, \overline{B_m} \,|\, o) \qquad \Gamma \vdash N_1 : B_{11} \quad \cdots \quad \Gamma \vdash N_{l_1} : B_{1l_1}}{\Gamma \vdash M N_1 \cdots N_{l_1} : (\overline{B_2} \,|\, \cdots \,|\, \overline{B_m} \,|\, o)}$$

In the safe λ-calculus, when constructing λ-abstractions, *all variables* of the relevant type-partition must be abstracted; when constructing applications, the operator-term must be applied to *all operand-terms* (one for each type) of the relevant type-partition. For example $F : ((o,o),o,o,o), \varphi : (o,o), x : o \vdash F\varphi x : (o,o)$ is not safe; it follows that

$$F : ((o,o),o,o,o), \varphi : (o,o), x : o, y : o \vdash F(F\varphi x)xy : o$$

is not safe. But $F : ((o,o),o,o,o), \varphi : (o,o) \vdash F\varphi a : (o,o)$ is safe for some constant a, and so is $F : ((o,o),o,o,o), \varphi : (o,o), x : o, y : o \vdash F\varphi xy : o$.

In the following, whenever it is clear from the context what the type of a term M is, we shall write $\mathsf{level}(M)$ to mean the level of that type.

Lemma 2.1. *Suppose* $\overline{x_1} : \overline{A_1} \,|\, \cdots \,|\, \overline{x_n} : \overline{A_n} \vdash M : B$ *is valid, where* $B = (\overline{B_1} \,|\, \cdots \,|\, \overline{B_m} \,|\, o)$.

(i) $(\overline{A_1} \,|\, \cdots \,|\, \overline{A_n} \,|\, \overline{B_1} \,|\, \cdots \,|\, \overline{B_m} \,|\, o)$ *is homogeneous.*

(ii) Any free variable of M *has level at least* $\mathsf{level}(M)$.

(iii) For any subterm $\boldsymbol{\lambda}\Phi.L$ *of* M*, if the variable* φ *occurs in* L *and* $\mathsf{level}(\varphi) < \mathsf{level}(\Phi)$ *then* φ *is bound in* L. □

We omit the straightforward proofs.

What does "safe" mean?

Substitution is a fundamental operation in the λ-calculus. In the key clause of the definition

$$(\boldsymbol{\lambda}x.M)[N/y] \stackrel{\text{def}}{=} \boldsymbol{\lambda}z.((M[z/x])[N/y]) \quad \text{where "}z\text{ is fresh"}$$

bound variables are renamed afresh to prevent variable capture. In the safe λ-calculus, one can get away without any renaming.

Lemma 2.2. *In the safe λ-calculus, there is no need to rename bound variables afresh when performing substitution*

$$M[N_1/x_1, \cdots, N_n/x_n]$$

provided the substitution is performed simultaneously on all *free variables of the same level in* M *i.e.* $\{x_1, \cdots, x_n\}$ *is the set variables of the same level as* x_1 *that occur free in* M.

Proof. Suppose φ occurs free in M, and bound variables in M are not renamed in the substitution $M[N/\varphi]$. Further suppose x, a variable occurring free in N, is captured as a result of the substitution. I.e. there is a subterm $\boldsymbol{\lambda}x.L$ of M such that φ occurs free in L. We compare $\mathsf{level}(x)$ with $\mathsf{level}(\varphi)$:

- Case 1: $\mathsf{level}(x) > \mathsf{level}(\varphi)$.

  This is impossible: Since $\boldsymbol{\lambda} x.L$ is safe, by Lemma 2.1(iii), L can have no free variables of level less than $\mathsf{level}(x)$.

- Case 2: $\mathsf{level}(x) < \mathsf{level}(\varphi)$.

  This is impossible: Since N is safe and of level $\mathsf{level}(\varphi)$, by Lemma 2.1(ii), it can have no free variable of level less than $\mathsf{level}(\varphi)$.

- Case 3: $\mathsf{level}(x) = \mathsf{level}(\varphi)$.

  If follows from the formation rule for λ-abstraction that φ cannot occur free in M since the subterm $\boldsymbol{\lambda} x.L$ must be in the scope of some subterm $\boldsymbol{\lambda}\varphi.\cdots$ of M, so that φ does not occur free in M. Thus this case cannot arise either.

$\square$

2.2 Higher-order grammars and the OI-hierarchy

A ***higher-order grammar*** (or HOG, for short) is a five-tuple $G = \langle\, N, V, \Sigma, \mathcal{R}, S, e\,\rangle$ such that

(i) N is a finite set of homogeneously-typed non-terminals, and S, the *start symbol*, is a distinguished element of N of type o

(ii) V is a finite set of typed variables

(iii) Σ is a finite alphabet

(iv) $\mathcal{R}$ is a finite set of triples, called *rewrite rules* (also referred to as production rules), of the form

$$F x_1 \cdots x_n \ \xrightarrow{\alpha}\ E$$

where $\alpha \in (\Sigma \cup \{\, \epsilon\,\})$, $F : (A_1, \cdots, A_n, o) \in N$, each $x_i : A_i \in V$, and E is either a term in $\mathcal{T}^o(N \cup \{x_1, \cdots, x_m\})$ or is $e : o$. We say that F has ***formal parameters*** $x_1, \cdots, x_m$. In the case where there grammar has two or more rules with the non-terminal F on the lefthand side, then we assume both rules have the same formal parameters in the same order. Following [KNU01] we make the assumption that if $F \in N$ has type $(A_1, \cdots, A_n, o)$ and $n \geq 1$, then $A_n = o$. Thus, each non-terminal has at least one level-0 variable. Note that this is not really a restriction – as this variable need not occur on the righthand side. We also assume that S never occurs on the righthand side of a rewrite rule. We say that the rewrite rule has *name* F and *label* α; it has level n just in case the type of its name has level n.

We say that G is of level n just in case n is the level of the rewrite rule that has the highest-level. We say that G is ***deterministic*** if for every $F x_1 \cdots x_n \xrightarrow{\alpha} E$ and $F x_1 \cdots x_n \xrightarrow{\alpha'} E'$ in $\mathcal{R}$

- If $\alpha = \alpha'$ then $E = E'$.
- If $\alpha = \epsilon$ and $E \neq e$, then $\alpha = \alpha'$

We say that a deterministic HOG is *real-time* if no rule has an ϵ label.

The language of a HOG

We extend $\mathcal{R}$ to a family of binary relations $\xrightarrow{\alpha}$ over $\mathcal{T}^o(N) \cup \{ e \}$, where α ranges over $\Sigma \cup \{ \epsilon \}$, by the rule: if $F x_1 \cdots x_n \xrightarrow{\alpha} E$ is a rule in $\mathcal{R}$ where $x_i : A_i$ then for each $M_i \in \mathcal{T}^{A_i}(N)$ we have

$$F M_1 \cdots M_n \xrightarrow{\alpha} E[\overline{M_i / x_i}].$$

A *derivation* of $w \in \Sigma^*$ is a sequence $P_1, P_2, \cdots, P_n$ of terms in $\mathcal{T}^o(N)$, and a corresponding sequence $\alpha_1, \cdots, \alpha_n$ of elements in $\Sigma \cup \{ \epsilon \}$ such that

$$S = P_1 \xrightarrow{\alpha_1} P_2 \xrightarrow{\alpha_2} P_3 \xrightarrow{\alpha_3} \quad \cdots \quad \xrightarrow{\alpha_{n-1}} P_n \xrightarrow{\alpha_n} e$$

and $w = \alpha_1 \cdots \alpha_n$. The *language* generated by G, written $L(G)$, is the set of words over Σ that have derivations in G. We say that two grammars are *equivalent* if they generate the same language.

Safe Grammars

A grammar is ***safe*** if for each rewrite rule $F x_1 \cdots x_n \xrightarrow{\alpha} E$ we have that

$$x_1 : A_1, \cdots, x_n : A_n \vdash E : o$$

is a valid typing judgement of the safe λ-calculus, where E is constructed possibly using symbols from N as constants. Otherwise, the grammar is ***unsafe***.

The OI-hierarchy

In [Dam82], Damm introduced the OI-hierarchy. The nth level of the hierarchy is generated by level-n grammars (defined differently from our grammars). Furthermore, each level is strictly contained in the one above it. The first three levels correspond to the regular, the context-free, and the indexed languages [Aho68]. Damm's definition of a level-n grammar, although packaged somewhat differently from ours, can be thought of as a special case of our definition. In particular, it is routine to show that a level-n grammar using his definition corresponds to a *safe* level-n grammar in our definition (and the converse holds too). Thus, Damm's definition is such that the safety restriction – referred to as derived types in his paper – is always "built-in". For a note comparing the two definitions (ours and Damm's) we point the reader to [dMO04]. This note also motivates our preference for our definition.

The reader familiar with the OI-hierarchy should note that our derivation relation is constructed so that all derivations are outside-in (which, in the monadic case, corresponds to a leftmost derivation). Bearing this in mind, the fact that our definition of a safe level-n grammar coincides with Damm's definition of a level-n grammar should be almost immediate.

Remark 2.3. We note there is nothing to be gained by allowing the rhs of rules to be λ-terms, since any such rule can be transformed to an equivalent system of rules, all of whose rhs are applicative terms. E.g. the level-3 rule

$$F \varphi \xrightarrow{\alpha} \varphi(\boldsymbol{\lambda} x.\varphi(\boldsymbol{\lambda} y.x))$$

is equivalent to the following system of rules

$$\begin{array}{rcl} F\,\varphi & \xrightarrow{\alpha} & \varphi\,(G\varphi) \\ G\,\varphi\,x & \xrightarrow{\epsilon} & \varphi\,(Hx) \\ H\,x\,y & \xrightarrow{\epsilon} & x \end{array}$$

Example 2.4. Consider the following *deterministic* grammar, where

$$\Sigma = \{ h_1, h_2, h_3, f_1, f_2, g_1, a, b \},$$

the typed non-terminals are

$$D : ((o, o), o, o, o), \quad H : ((o, o), o, o), \quad F : (o, o, o), \quad G : (o, o), \quad A, B : o$$

with rules:

$$\begin{array}{rclrcl} S & \xrightarrow{\epsilon} & DGAB & Gx & \xrightarrow{g_1} & x \\ D\varphi xy & \xrightarrow{h_1} & D(\underline{D\varphi x})y(\varphi y) & Fxy & \xrightarrow{f_1} & x \\ D\varphi xy & \xrightarrow{h_2} & H(\underline{Fy})x & Fxy & \xrightarrow{f_2} & y \\ D\varphi xy & \xrightarrow{h_3} & \varphi B & A & \xrightarrow{a} & e \\ H\varphi x & \xrightarrow{\epsilon} & \varphi x & B & \xrightarrow{b} & e \end{array}$$

This is an unsafe grammar because of the underlined expressions: both of which are level-1 terms, but contain level-0 variables.

As this grammar is deterministic [dMO04] this means that each word in the language is generated in a unique way. Hence, it should be easy for the reader to check by hand that the words $h_1h_3h_2f_1b$ and $h_1h_3h_2f_2a$ are part of the language, whereas $h_1h_3h_2f_1a$ is not.

2.3 Pointer machines

Pointer machines are a model of computation for string languages generated by higher-order grammars. The ***pointer machine*** for an n-grammar $G = \langle N, V, \Sigma, \mathcal{R}, S, e \rangle$ has a (pushdown) stack. A stack β is a non-empty sequence $[a_1, a_2, \cdots, a_n]$ where each $a_i \in \Gamma$, a set determined by G. The following operations are defined on a stack: for $a \in \Gamma$

$$\begin{array}{rcl} \mathsf{head}\,[a_1, a_2, \cdots, a_n] & = & a_1 \\ \mathsf{tail}\,[a_1, a_2, \cdots, a_n] & = & [a_2, \cdots, a_n] \\ a : [a_1, a_2, \cdots, a_n] & = & [a, a_1, a_2, \cdots, a_n]. \end{array}$$

The stack alphabet Γ is a certain finite subset of $\mathcal{T}^o(N \cup V) \cup \{e\}$, which will be defined shortly. Each a_i, which we shall refer to as an ***item***, will have exactly one pointer for each variable (from V) that occurs in it. Thus an item in the stack may have several pointers emanating from it (apart from a_n which has no pointers). Intuitively, a pointer is a physical link connecting an item a_i to some other item a_j for $j > i$.

We define the ***stack transition relation***. The behaviour of a pointer machine is given in terms of a labelled transition relation between stacks. A computation of a pointer machine begins with the stack containing a single item, which is the start symbol S. Thereafter, let β be the stack. We use meta-variables F and x to range over N (non-terminals) and V (variables) respectively.

1. Assume $\mathsf{head}\,\beta = Ft_1 \cdots t_n$. If $Fx_1 \cdots x_n \xrightarrow{\alpha} E$ is a G-rule then

$$\beta \xrightarrow{\alpha} E : \beta$$

   and each occurrence of a variable x_j in E points to $\mathsf{head}\,\beta$, in other words $Ft_1 \cdots t_n$. Further, the pointer structure of β is preserved.

2. Assume $\mathsf{head}\,\beta = x t_1 \cdots t_n$ and x points to an item $a = D s_1 \cdots s_m$ in β. Furthermore, suppose that x is the ith formal parameter of D. Then

$$\beta \xrightarrow{\epsilon} s_i t_1 \cdots t_n : \mathsf{tail}\,\beta$$

all pointers in $\mathsf{tail}\,\beta$ are preserved and all pointers from $t_1, \cdots, t_n$, as well as those from s_i, are preserved.

3. Assume $\mathsf{head}\,\beta = e$, then the pointer machine halts.

Let $[S] \xrightarrow{\alpha_1} P_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_{n-1}} P_n$ be a transition sequence of pointer machine stacks such that $\mathsf{head}\,P_n = e$. Then, we say that the word, $\alpha_1\alpha_2 \cdots \alpha_{n-1}$ (over Σ), is ***accepted*** by the pointer machine. We say that a word belongs to the language of a given pointer machine if and only if there exists a transition sequence of the pointer machine starting from $[S]$ that results in the word being accepted.

The language accepted by a pointer machine can easily be seen to be exactly the language of the higher-order grammar. In particular, it corresponds to linear head reduction in the λ-calculus.

Remark 2.5. (i) Pointer machines are due to C. Stirling. The second author first learnt of the idea in [Sti02].

(ii) The reader acquainted with [KNU01, KNU02] should be able to easily modify the definition of a pointer machine to serve as a model of computation for term-trees generated by higher-order grammars.

Stack alphabet

We write R for the set consisting of the rhs of each rule from $\mathcal{R}$. Let $\xi E_1 \cdots E_m$ be a ground-type term, where ξ is either a variable or a non-terminal. We define

$$Args\,(\xi E_1 \cdots E_m) \;=\; \{\, E_1, \cdots, E_m \,\}.$$

Now define two sets, Γ (which is a set of ground-type terms) and Exp, by mutual induction over the following rules:

$$\frac{}{R \subseteq \Gamma}$$

$$\frac{}{Args\,(r) \subseteq Exp}\; r \in R$$

$$\frac{E^A \in Exp \quad x^A E_1 \cdots E_m \in \Gamma}{E\, E_1 \cdots E_m \in \Gamma}$$

$$\frac{\xi E_1 \cdots E_m \in \Gamma}{Args\,(\xi E_1 \cdots E_m) \subseteq Exp}$$

We can take Γ to be the stack alphabet. (Γ is a superset of what we actually need).

Lemma 2.6. *Γ is finite.*

Proof. We define a new set Γ' (of ground-type terms) by induction over the rules: $R \subseteq \Gamma'$ and

$$\frac{E^A \in Exp' \quad x^A E_1 \cdots E_m \in \Gamma'}{E^A E_1 \cdots E_m \in \Gamma'}$$

where $Exp' = \{ N : N \text{ is a subterm of some } r \in R \}$, which is clearly finite. It is straightforward to see that $Exp \subseteq Exp'$ and $\Gamma \subseteq \Gamma'$. Therefore it suffices to prove that Γ' is finite. We define a partial order over the set $\{ A : E^A \in Exp' \}$ of types by:

$$A > B \iff A = A_1 \longrightarrow \cdots \longrightarrow A_n \longrightarrow B$$

where $n > 0$. Now define a function $H : \mathcal{T}^o(N \cup V) \longrightarrow \mathcal{T}^o(N \cup V)$ as follows:

$$H(G) \;=\; R \cup G \cup \{ \xi^A U_1 \cdots U_n E_1 \cdots E_m : x^B E_1 \cdots E_m \in G, \xi^A U_1 \cdots U_n : B \in Exp' \}$$

We observe that Γ' is the fixpoint of H, which is reached after at most j iterations of H, where j is the length of the longest chain in the poset $\{ A : E^A \in Exp' \}$. $\square$

Useful properties

We consider the operation of a pointer machine. The stack items of the pointer machine are of two types: *incomplete* and *complete*. A complete item is one that is headed by a non-terminal, so for example $F\overline{s}$ is a complete item. An incomplete item is headed by a variable. We call it incomplete, because it is, in a sense, work in progress: we will remain at this stack item (replacing the head variable) until it is finally headed by a non-terminal, in which case it will become complete.

Lemma 2.7. *All items of any reachable pointer machine stack β are of level 0; further,* $\mathsf{tail}\,\beta$ *consists only of complete items.*

Proof. Induction on the number of stack transitions. $\square$

Lemma 2.8. *Let $s_0 s_1 \cdots s_k$ be an item in a reachable pointer machine stack β. Then the following hold:*

(i) If x, a variable occurring in $s_0 s_1 \cdots s_k$, points to a stack item $D t_1 \cdots t_m$ then x is a formal parameter of D.

(ii) For each i, the variables in s_i all point to the same stack item.

(iii) Suppose s_i has a variable with pointer to an item a, and s_j has a variable with pointer to an item a'. If $i < j$ then either a and a' are the same item or a occurs deeper in the stack than a'.

Proof. Induction on the number of stack transitions.

$\square$

Bearing these two observations in mind, we modify the definition of a pointer machine slightly to obtain a ***compacting pointer machine*** for a HOG $G = \langle N, V, \Sigma, \mathcal{R}, S, e \rangle$. As before, it has a pushdown stack and a computation of a pointer machine begins with the stack containing only the start symbol S. Afterwards, the stack transition relation is defined as follows below. Let β be the current stack:

1. Assume $\mathsf{head}\,\beta = F t_1 \cdots t_n$. If $F x_1 \cdots x_n \stackrel{\alpha}{\longrightarrow} E$ is a G-rule then

$$\beta \stackrel{\alpha}{\longrightarrow} E : \beta$$

and each occurrence of a variable x_j in E points to $\mathsf{head}\,\beta$, in other words $F t_1 \cdots t_n$. Further the pointer structure of β is preserved.

2. a. Assume $\mathsf{head}\,\beta = xt_1 \cdots t_n$ and suppose that x is a variable of level at least 1, and points to an item $a = Ds_1 \cdots s_m$ in β. Furthermore, suppose that x is the ith formal parameter of D. Then

$$\beta \xrightarrow{\epsilon} s_i t_1 \cdots t_n : \mathsf{tail}\,\beta$$

all pointers in $\mathsf{tail}\,\beta$ are preserved and all pointers from $t_1, \cdots, t_n$, as well as those from s_i, are preserved.

b. Assume $\mathsf{head}\,\beta = x$, x is of level 0, and points to an item $\mathsf{head}\,\beta' = Ds_1 \cdots s_m$ where β' is a suffix of β. Furthermore, suppose that x is the ith formal parameter for D. By Lemma 2.8 we know that all the variables in s_i (if any) point to the *same* item in β', (say) $\mathsf{head}\,\beta''$ where β'' is a suffix of β'. Then

$$\beta \xrightarrow{\epsilon} s_i : \beta''$$

all pointers in β'' are preserved as well as those from s_i[2]. In the case where s_i contains no variables β'' may be chosen arbitrarily.

3. Assume $\mathsf{head}\,\beta = e$, then the pointer machine halts.

The key to understanding the compacting pointer machine is that it maintains the following invariant. If β is the pointer stack, with $\mathsf{head}\,\beta = s_0 s_1 \cdots s_n$, then either 1) s_n contains at least one variable with a pointer to $\mathsf{tail}\,\beta$ or 2) s_n contains only non-terminals. As an example, suppose that

$$[\varphi x_1 x_2, A, B, C, D, E]$$

is a pointer stack of the original pointer machine, where φ is a variable of level 1 and x_1, x_2 are variables of level 0. Here, $A, B, C, \cdots$ are meta-variables for terms in $\mathcal{T}^o(N \cup V)$. Only one pointer has been shown: from variable x_2 in the topmost item to C. Other pointers exist, but have not be indicated. By Lemma 2.8(iii) we know that x_2 and φ must point to item C or items deeper in the stack, namely, D or E. In this instance the items A and B have now become redundant: for any stack configuration reachable from here, the topmost item will never contain a pointer to A or B. Thus, for all intents and purposes, they may as well be cut out, and removing such redundancies is what the compacting pointer machine strives to achieve. So the above configuration would become:

$$[\varphi x_1 x_2, C, D, E]$$

and all other pointers for remaining items are preserved. Hence the name compacting[3].

Lemma 2.9. *A compacting pointer machine is equivalent to a pointer machine.* □

Representation

This section concerns notation for pointer machine stacks. Recall that each item in the stack may have several pointers emanating from it. In particular, suppose we have the following PM stack:

$$[\varphi y z,\ FGxAB,\ DA,\ S]$$

[2]In particular, all pointers from variables in s_i will be to $\mathsf{head}\,\beta''$.

[3]Note that although the compacting pointer machine does remove such redundant segments, it is not optimal; in the sense that some such redundant segments may still exist. This, however, is deliberate.

Note that φ points to DA, both y and z point to $FGxAB$, and x points to DA. Rather than resorting to having to draw arrows we can convey this information by subscripting each variable with a PM stack. We annotate each variable with a suffix s of the PM stack, such that the pointer points to $\mathsf{head}\, s$. Thus, for the above example we would have:

$$[x_{p_\varphi} y_{p_y} z_{p_z}, FGx_{p_x}AB, DA, S] \text{ where}$$

$$\begin{aligned} p_\varphi &= [DA, S] \\ p_y &= [FGx_{p_x}AB, DA, S] \\ p_z &= [FGx_{p_x}AB, DA, S] \\ p_x &= [DA, S] \end{aligned}$$

Recall from Lemma 2.8 that if $s_0 s_1 \cdots s_n$ is an item of a PM stack, and if x, x' are variables occurring in s_i for some i, then the pointer of x is the same as the pointer of x'. Hence, to avoid having to subscript *every* single variable with a PM stack, we adopt following convention: For p, q pointer stacks and $s, t \in \mathcal{T}(N \cup V) \cup \{e\}$, we have:

- $(st)_p = (s_p)(t_p)$ for (st) an application
- $(s_p)_q = s_p$ for all s
- $F_p = F$ for F a non-terminal
- $e_p = e$

Thus, adopting the above convention, our earlier example could be rewritten as:

$$[(\varphi_{p_\varphi} yz)_{p_z}, (FGxAB)_{p_x}, DA, S] \text{ where } p_\varphi, p_z, p_x \text{ are as before}$$

Note the yz.

An example

Example 2.10. We give an example computation of the (non-compacting) pointer machine for the grammar in 2.4.

$$\begin{aligned}
& [S] \\
\xrightarrow{\epsilon}\; & [DGAB, S] \\
\xrightarrow{h_1}\; & [(D(D\varphi x)y(\varphi y))_{p_1}, DGAB, S] \\
\xrightarrow{h_3}\; & [(\varphi x)_{p_2}, (D(D\varphi x)y(\varphi y))_{p_1}, DGAB, S] \\
\xrightarrow{\epsilon}\; & [(D\varphi x)_{p_1}B, (D(D\varphi x)y(\varphi y))_{p_1}, DGAB, S] \\
\xrightarrow{h_2}\; & [(H(Fy)x)_{p_3}, (D\varphi x)_{p_1}B, (D(D\varphi x)y(\varphi y))_{p_1}, DGAB, S] \\
\xrightarrow{\epsilon}\; & [(\varphi x)_{p_4}, (H(Fy)x)_{p_3}, (D\varphi x)_{p_1}B, (D(D\varphi x)y(\varphi y))_{p_1}, DGAB, S] \\
\xrightarrow{\epsilon}\; & [(Fy)_{p_3}x_{p_4}, (H(Fy)x)_{p_3}, (D\varphi x)_{p_1}B, (D(D\varphi x)y(\varphi y))_{p_1}, DGAB, S] \\
\xrightarrow{f_1}\; & [x_{p_5}, (Fy)_{p_3}x_{p_4}, (H(Fy)x)_{p_3}, (D\varphi x)_{p_1}B, (D(D\varphi x)y(\varphi y))_{p_1}, DGAB, S] \\
\xrightarrow{\epsilon}\; & [y_{p_3}, (Fy)_{p_3}x_{p_4}, (H(Fy)x)_{p_3}, (D\varphi x)_{p_1}B, (D(D\varphi x)y(\varphi y))_{p_1}, DGAB, S] \\
\xrightarrow{\epsilon}\; & [B, (Fy)_{p_3}x_{p_4}, (H(Fy)x)_{p_3}, (D\varphi x)_{p_1}B, (D(D\varphi x)y(\varphi y))_{p_1}, DGAB, S] \\
\xrightarrow{b}\; & [e, B, (Fy)_{p_3}x_{p_4}, (H(Fy)x)_{p_3}, (D\varphi x)_{p_1}B, (D(D\varphi x)y(\varphi y))_{p_1}, DGAB, S]
\end{aligned}$$

where

$$\begin{array}{lcl}
p_1 & = & [DGAB, S] \\
p_2 & = & [D(D\varphi x)y(\varphi y))_{p_1}, DGAB, S] \\
p_3 & = & [(D\varphi x)_{p_1}B, (D(D\varphi x)y(\varphi y))_{p_1}, DGAB, S] \\
p_4 & = & [H(Fy)x)_{p_3}, (D\varphi x)_{p_1}B, (D(D\varphi x)y(\varphi y))_{p_1}, DGAB, S] \\
p_5 & = & [(Fy)_{p_3}x_{p_4}, (H(Fy)x)_{p_3}, (D\varphi x)_{p_1}B, (D(D\varphi x)y(\varphi y))_{p_1}, DGAB, S]
\end{array}$$

Note that the pointer machine accepts the word $h_1h_3h_2f_1b$. We now demonstrate how the compacting version of the pointer machine is more economical. In particular, the computation proceeds as above, up to the point:

$$\xrightarrow{f_1} \; [x_{p_5}, (Fy)_{p_3}x_{p_4}, (H(Fy)x)_{p_3}, (D\varphi x)_{p_1}B, (D(D\varphi x)y(\varphi y))_{p_1}, DGAB, S]$$

we then proceed as follows:

$$\begin{array}{ll}
\xrightarrow{\epsilon} & [y_{p_3}, (D\varphi x)_{p_1}B, (D(D\varphi x)y(\varphi y))_{p_1}, DGAB, S] \\
\xrightarrow{\epsilon} & [B, (D(D\varphi x)y(\varphi y))_{p_1}, DGAB, S] \\
\xrightarrow{b} & [e, B, (D(D\varphi x)y(\varphi y))_{p_1}, DGAB, S]
\end{array}$$

where $p_1, \cdots, p_5$ are as before.

2.4 Higher-order pushdown automata

Before we can define a higher-order pushdown automaton, we need to define a ***level-n store*** or simply *n-store*. Fix a finite set Γ of *store symbols*, including a distinguished bottom-of-store symbol $\bot$. A *1-store* is a finite non-empty sequence $[a_1, \cdots, a_m]$ of Γ-symbols such that $a_i = \bot$ iff $i = m$. For $n \geq 1$, an *$(n+1)$-store* is a non-empty sequence of n-stores. Inductively we define the *empty $(n+1)$-store* $\bot_{n+1}$ to be $[\bot_n]$ where we set $\bot_0 = \bot$. Recall the following standard operations on 1-stores: for $a \in (\Gamma \setminus \{\bot\})$

$$\begin{array}{rcl}
\mathsf{push}_1(a)\,[a_1, \cdots, a_m] & = & [a, a_1, \cdots, a_m] \\
\mathsf{pop}_1\,[a_1, a_2, \cdots, a_m] & = & [a_2, \cdots, a_m]
\end{array}$$

For $n \geq 2$, the following set Op_n of *level-n operations* are defined over n-stores:

$$\left\{\begin{array}{rcll}
\mathsf{push}_n\,[s_1, \cdots, s_l] & = & [s_1, s_1, \cdots, s_l] & \\
\mathsf{push}_k\,[s_1, \cdots, s_l] & = & [\mathsf{push}_k\,s_1, s_2, \cdots, s_l], & 2 \leq k < n \\
\mathsf{push}_1(a)\,[s_1, \cdots, s_l] & = & [\mathsf{push}_1(a)\,s_1, s_2, \cdots, s_l] & \\
\mathsf{pop}_n\,[s_1, \cdots, s_l] & = & [s_2, \cdots, s_l] & \\
\mathsf{pop}_k\,[s_1, \cdots, s_l] & = & [\mathsf{pop}_k\,s_1, s_2, \cdots, s_l], & 1 \leq k < n \\
\mathsf{id}\,[s_1, \cdots, s_l] & = & [s_1, \cdots, s_l] &
\end{array}\right.$$

In addition we define

$$\begin{array}{rcl}
\mathsf{top}_n\,[s_1, \cdots, s_l] & = & s_1 \\
\mathsf{top}_k\,[s_1, \cdots, s_l] & = & \mathsf{top}_k\,s_1, \quad 1 \leq k < n
\end{array}$$

Note that $\mathsf{pop}_k\,s$ is undefined if $\mathsf{top}_k\,s = \bot_{k-1}$, for $k \geq 1$.

A ***level-n pushdown automaton*** (or nPDA for short) is a 6-tuple $\langle Q, \Sigma, \Gamma, \delta, q_0, F \rangle$ where:

(i) Q is a finite set of states, $q_0 \in Q$ is the start state, and $F \subseteq Q$ is a set of accepting states

(ii) Σ the finite input alphabet

(iii) Γ the finite store alphabet (which is assumed to contain $\perp$)

(iv) $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \times Q \times Op_n$ the transition relation.

A ***configuration***[4] of an nPDA is given by a triple (q, w, s) where q is the current state, w is the remaining input, and s is the is an n-store over Γ.

Given a configuration (q, aw, s) (where $a \in \Sigma$ or $a = \epsilon$, and $w \in \Sigma^*$) where $\mathsf{top}_1(s) = Z$, we define the following relation $\to$:

$$(q, aw, s) \to (p, w, s')$$

if $(q, a, Z, p, \theta) \in \delta$ and $s' = \theta(s)$. Intuitively, this says that if we are in state q reading input symbol a and the topmost store symbol is Z, then we may change the state to p, consume a, and perform the operation θ to the current n-store. The transitive closure of $\to$ is denoted by $\to^+$, whereas the reflexive and transitive closure is denoted by $\to^*$. We say that the input w is ***accepted*** by the above nPDA if $(q_0, w, \perp_n) \to^* (q_f, \epsilon, s)$ for some pushdown store s and some $q_f \in F$.

Remark 2.11. We define 0PDAs to be finite automata. Note that 1PDAs are just the standard pushdown automata. For an example of a 2PDA we refer the reader to the following section.

The above defined nPDA can be thought of as being non-deterministic. However, we say that an nPDA is ***deterministic*** if the following hold:

(i) If $\delta(q, a, Z) \neq \emptyset$ for some $a \in \Sigma$ then $\delta(q, \epsilon, Z) = \emptyset$

(ii) $|\delta(q, a, Z)| \leq 1$ for all $a \in \Sigma \cup \{\epsilon\}$, $q \in Q$ and $Z \in \Gamma$.

Remark 2.12. In [dMO04] it is shown that a language L is generated by a level-n safe deterministic grammar if and only if it is accepted by a level-n deterministic pushdown automaton. This is an easy result to show – but has not appeared previously in the literature.

3 Relating nPDAs and n-grammars

3.1 Known results

Historically, the notion of safety has been key to relating nPDAs and level-n grammars. In particular, the following results are due to Damm and Goerdt[5]:

Theorem 3.1. *(Damm and Goerdt [DG86]) If G is a safe grammar of level n, then $L(G)$ is accepted by an nPDA.*

Theorem 3.2. *(Damm and Goerdt [DG86]) If L is the language of an nPDA then it is generated by a level-n safe grammar.*

[4]This is sometimes called *total configuration* in the literature.

[5]In their paper, safety is referred to as the restriction of derived types.

However, to date, no results exist for *unsafe* level-n grammars. In particular, if G is an unsafe level-n grammar, it is not known whether $L(G)$ is accepted by an nPDA, or perhaps a PDA of another level. Thus, we believe that our result, which we present in the following section, is a first step towards solving this problem. We will show that at level 2, every unsafe grammar can be converted into a safe one that generates the same language.

Before we give our result, however, we present a case study. Below we introduce a deterministic but unsafe level-2 grammar that generates a language U. We will show, via a "bespoke" proof, that U can be accepted by 2PDA. This effort may seem redundant, given that U is exactly the type of language that is amenable to our result. However, we believe there is much to be gained from our case study:

- It is a non-trivial example of an unsafe grammar at the lowest possible level (where safety becomes an issue), hence it is interesting in its own right.
- It provides us with a good idea as to the capabilities of a 2PDA.
- Finally, and perhaps most importantly, it lays the foundation for a conjecture we will make in Section 7.

3.2 An example: Urzyczyn's language

The language U consists of words of the form $w *^n$ where w is a proper prefix of a well-bracketed word such that no prefix of w is a well-bracketed expression; each parenthesis in w is implicitly labelled with a number, and n is the label of the last parenthesis. The two labelling rules are:

I. The label of the opening **(** is one; the label of any subsequent **(** is that of the preceding **(** plus one.

II. The label of **)** is the label of the parenthesis that precedes the matching **(**.

For example, the following are words in U (together with their respective sequences of labels):

(i)

$$\begin{array}{cccccccccccccccccccc}
(& (& (& (&) &) & (& (&) & (& (&) &) &) & (& (&) &) & * & * \\
1 & 2 & 3 & 4 & 3 & 2 & 5 & 6 & 5 & 7 & 8 & 7 & 5 & 2 & 9 & 10 & 9 & 2 & &
\end{array}$$

(ii)

$$\begin{array}{cccccccccccccccccc}
(& (& (& (&) &) & (& (&) & (& (&) &) & * & * & * & * & * \\
1 & 2 & 3 & 4 & 3 & 2 & 5 & 6 & 5 & 7 & 8 & 7 & 5 & & & & &
\end{array}$$

We first learnt of the language from [Urz03], wherein it was conjectured that U is inherently unsafe i.e. not acceptable by any higher-order PDA. This is in fact not the case. We shall first give an unsafe 2-grammar that generates the language, and then show that it is accepted by a safe (non-deterministic) 2PDA.

A way to compute the labels is to maintain a *configuration*, which is either a triple $\langle \gamma, y, z \rangle$ such that

- γ is a stack of future **)**-labels (written as a list $x : \phi$ whose head x is the top of the stack)
- y is the number of **(** read thus far
- z is the label of the last parenthesis read,

or a number, which is the number of remaining $*$ to be read. Note that the length of γ is equal to the number of as yet unmatched **(**'s at that point. The transitions are as follows:

$$\begin{array}{rcl} \langle\, x : \phi, y, z \,\rangle & \xrightarrow{(} & \langle\, z : x : \phi, y+1, y+1 \,\rangle \\ \langle\, x : \phi, y, z \,\rangle & \xrightarrow{)} & \langle\, \phi, y, x \,\rangle \\ \langle\, x : \phi, y, z \,\rangle & \xrightarrow{*} & z \\ z+1 & \xrightarrow{*} & z \end{array}$$

By mimicking the transition system, we can define a *deterministic* (and unsafe) 2-grammar that generates U. We set

$$\begin{array}{rcl} \Sigma & = & \{\, (,), * \,\} \\ N & = & \{\, S : o, D : ((o,o,o), o, o, o, o), G : (o,o,o), F : (o,o), E : o \,\} \end{array}$$

with production rules as follows:

$$\begin{cases} S & \xrightarrow{(} & D\, G\, E\, E\, E \\ D\, \phi\, x\, y\, z & \xrightarrow{(} & D\, (D\, \phi\, x)\, z\, (F\, y)\, (F\, y) \\ D\, \phi\, x\, y\, z & \xrightarrow{)} & \phi\, y\, x \\ D\, \phi\, x\, y\, z & \xrightarrow{*} & z \\ F\, x & \xrightarrow{*} & x \\ E & \xrightarrow{\epsilon} & e \end{cases}$$

Note that we have simply encoded the configuration $\langle\, x : \phi, y, z \,\rangle$ as the term $D\, \phi\, x\, y\, z$, and $*^{n+1}$ as $\underbrace{F(\cdots(F}_{n}\, E)$.

Unique decomposition of U-words

Consider words over the alphabet $\{\, (,), * \,\}$ composed of three parts as follows

$$\underbrace{(\cdots(\cdots(}_{(1)}\ \underbrace{(\cdots)\cdots(\cdots)}_{(2)}\ \underbrace{*\cdots*}_{(3)}$$

(1) is a prefix of a well-bracketed word such that no prefix of it (including itself) is a well-bracketed word

(2) is a well-bracketed word

(3) has length equal to the number of **(** in (1).

Call the collection of such words V. We claim that $U = V$. In the preceding example, the first two parts of the 3-partition for (i) are $(($ and $(())(()(()))(())$; for (ii) they are $(((())($ and $()(())$.

Proposition 3.3. *Let $y \in \{(,),*\}^*$. Then $y \in U$ if an only if it has a unique decomposition into $wx*^n$ where w, x, n satisfy conditions (1),(2),(3) above respectively.*

Proof. For convenience, given a word $w \in \Sigma^*$ and $a \in \Sigma$, we denote by $|w|_a$ the number of occurrences of a in w.

$\Rightarrow$: First let us show *existence* of such a decomposition for $y \in U$. We perform a case analysis.

(i). Suppose $y = z(*^k$. Then, we simply take $w = z($, and clearly $|w|_(= k$ as required.

(ii). Suppose instead that $y = z)*^k$. We give an algorithm which will decompose y correctly. Clearly, the final $)$ must have a matching $($, say it is the following: $y = z_1(z_2)*^k$ where (z_2) must be a well-bracketed expression. We know that k is the label of the final $)$. We also know that the label of the final $)$ is the label of the parenthesis which precedes the matching $($. Now, if z_1 has final parenthesis $($, then we are done as we set $w = z_1$ and $x = (z_2)$.

If, on the other hand, z_1 has final parenthesis $)$, then we repeat this process again as $z_1 = z_3(z_4)$. It is obvious that this process terminates. Note that if this process requires n iterations, then $x = (z_{2n}) \cdots (z_4)(z_2)$ and $w = z_{2n-1}$.

To show *uniqueness*, suppose that $y \in U$ has two decompositions: $wx*^k$ and $w'x'*^k$. It suffices to observe that these decompositions are distinct if and only if $w \neq w'$. However, note that $|w|_(= k = |w'|_($. Thus, it must be the case that either w or w' violates the condition that they must end in a $($.

$\Leftarrow$: Again, we perform a case analysis.

(i). Suppose that $m = 0$. This is obviously true.

(ii). Suppose that $m > 0$ and let $y = wx*^n$. Clearly wx is the proper prefix of a well-bracketed expression.

Note that $x = x_1 \cdots x_m$ for some $m \geq 0$, where each $x_i = (z_i)$ and each z_i is well-bracketed. Thus we have $y = w(_1 z_1)_1 \cdots (_m z_m)_m *^n$. Parentheses have been labelled for demonstrative purposes. For y to be in U the number of stars, n, should be the the label of the parenthesis $)_m$. We show this is indeed the case. The label of $)_m$ should be equal to the label of the parenthesis preceding $(_m$, which in this case would be $)_{m-1}$. Continuing in this way, we see that the label of $)_m$ is the label of the last parenthesis of w, which is always a $($, hence, it is the number $|w|_($.

$\square$

Constructing a 2PDA that accepts U

So, how do we construct a 2PDA that accepts U? Let y be the input. Let us reuse the notation from Proposition 3.3, so each word in U has a unique decomposition into the form $wx*^n$. The 2PDA guesses what prefix of y constitutes w - say it is the first k characters. As it reads the first k symbols it acts like a 1PDA checking that w is indeed a prefix of a well-bracketed word such that no prefix of it is well-bracketed. However, at the same time we need it to keep track of the number of ('s read. In order to do this dual job we need the power of the 2PDA. Essentially, the topmost 1-store behaves like the stack of a normal PDA which checks for a (proper prefix of a) well-bracketed word. However, each time we find a (we do both a push_1 and a push_2. Each time we find a), we do only a pop_1. Thus, the number of 1-stores is the number of ('s read in w. If it turns out that the first k characters of the input y do not satisfy the criteria for what w should be, we immediately reject. Otherwise, we enter phase 2 of the operation. In phase 2 we check that the next portion of the string, x, is well-bracketed (for this we only need the power of a 1PDA), until we come across the first $*$. If x was indeed well-bracketed we proceed to phase 3, otherwise we abort. In phase 3, we perform a pop_2 for each $*$ we meet. If we end up with an empty 2-store after reading the all the $*$'s, we accept. Otherwise, reject.

4 The Theorem: motivation and explanation

We now present our main result.

Theorem 4.1. *If G is a higher-order grammar at level 2 that is not assumed to be safe, there exists a non-deterministic 2PDA that accepts the same language. Moreover, the conversion is effective.*

Our proof is split into two transformations. Given a level-2 grammar G, we show:

1. The (compacting) pointer machine for G is simulated by a 2PDAL, where 2PDAL is a machine that has yet to be introduced. (Section 5)
2. We then show that a 2PDAL for G can be simulated by a nondeterministic 2PDA. (Section 6)

Combining our result with that of [DG86], we have:

Corollary 4.2. *Every string language that is generated by an unsafe 2-grammar can also be generated by some safe (non-deterministic) 2-grammar.*

Thus safety is not really a restriction at level 2 for string languages.

5 Simulating 2PMs by 2PDALs

5.1 Understanding KNU's proof

In [KNU02] it was shown that a term tree generated by a safe grammar of level n is accepted by a pushdown automata of the same level. Their proof, which given a grammar G constructs a corresponding pushdown automaton, can easily be adapted to work in the string-language setting: all one needs to do is incorporate the input alphabet. Recall, however, that this result for the string-language setting was proved earlier by Damm and Goerdt [DG86]. Here we present an adaption of the proof in [KNU02], specialised to (level 2) string languages.

Theorem 5.1. *Let G be a 2-grammar that generates a string language. Then $L(G)$ is accepted by some 2PDA.*

Proof. We use the same setup as in Section 5.2 of [KNU02], but now we incorporate an input string over the alphabet Σ. Recall that there are 6 cases for the transition function; they are given in Fig. 1. $\square$

$$
\begin{array}{rcll}
(q_0, a, Dt_1 \cdots t_n) & \to & (q_0, \mathsf{push}_1(E)) \text{ if } Dx_1 \cdots x_n \xrightarrow{a} E & (1) \\
(q_0, \epsilon, e) & \to & \mathit{accept} & (2) \\
(q_0, \epsilon, x_j) & \to & (q_j, \mathsf{pop}_1) \text{ if } x_j : o & (3) \\
(q_0, \epsilon, x_j t_1 \cdots t_n) & \to & (q_j, \mathsf{push}_2 \,;\, \mathsf{pop}_1) \text{ if } x_j \text{ has level} > 0 & (4) \\
1 < j \leq n, (q_j, \epsilon, \$t_1 \cdots t_n) & \to & (q_0, \mathsf{pop}_1 \,;\, \mathsf{push}_1(t_j)) & (5) \\
j > n, (q_j, \epsilon, \$t_1 \cdots t_n) & \to & (q_{j-n}, \mathsf{pop}_2) & (6)
\end{array}
$$

Figure 1: Adapted transition rules from [KNU02]

Convention. In the Figure x_j means the j-th formal parameter of the relevant non-terminal.

Let us examine why their proof fails if we attempt to apply it (blindly) to a level-2 unsafe grammar. As an example, we consider the grammar given in Example 2.4. Recall that the word $h_1 h_3 h_2 f_1 a$ is *not* in the language.

The automaton starts off in the configuration $(q_0, h_1 h_3 h_2 f_1 a, [[S]])$, after a few steps we reach the following configuration:

$$(q_0, \quad h_2 f_1 a, \quad [[\varphi B, D(D\varphi x)y(\varphi y), DGAB, S]])$$

As the topmost item, φB, is headed by an level-1 variable, we need to find out what φ is in order to proceed. Note that φ is the 1st formal parameter of the preceding item: $D(D\varphi x)y(\varphi y)$, i.e., it refers to $D\varphi x$. To this end, we perform a push_2 and then perform a pop_1, and replace the topmost item with $D\varphi x$. In other words, we have applied rule 4 followed by rule 5 to arrive at:

$$\begin{array}{lll}(q_0, & h_2 f_1 a, & [[(D\varphi x)^{\langle 1- \rangle}, DGAB, S], \\ & & [\varphi B^{\langle 1+ \rangle}, D(D\varphi x)y(\varphi y), DGAB, S]])\end{array}$$

Here we have labelled two store items, one with a $1-$ and the other with a $1+$. These labels are not part of the store alphabet, they have been added for our benefit: so that we may identify these two store items later on.

The crux behind their construction is the following. Suppose we meet the item $D\varphi x^{\langle 1- \rangle}$ later on in the computation, and suppose that we would like to request its third argument, meaning we would be in state q_3. Note, however, that $D\varphi x^{\langle 1- \rangle}$ has only 2 arguments. The missing argument can be found by visiting the item $\varphi B^{\langle 1+ \rangle}$. Hence the labelling. We need to ensure that there is a systematic way to get from $D\varphi x^{\langle 1- \rangle}$ to $\varphi B^{\langle 1+ \rangle}$ whenever we are in a state q_n for $n > 2$ and we have $D\varphi x^{\langle 1- \rangle}$ as our topmost symbol. This systematic way suggested by [KNU02] is embodied by rule 6 of Fig. 1. In particular, it says that all we need to do is perform a pop_2, followed by a change in state to q_{n-2}, and to repeat if necessary. Note that if we applied rule 6 to the current configuration, we would indeed be brought to the right place, $\varphi B^{\langle 1+ \rangle}$. We will see, however, that with an unsafe grammar, this invariant may be violated.

After a few more steps of the 2PDA we will arrive at another configuration where the topmost symbol is headed by a level-1 variable:

$$\begin{array}{lll}(q_0, & f_1 a, & [[\varphi x, H(Fy)x, (D\varphi x)^{\langle 1- \rangle}, DGAB, S], \\ & & [\varphi B^{\langle 1+ \rangle}, D(D\varphi x)y(\varphi y), DGAB, S]])\end{array}$$

Therefore, we next get:

$$\begin{array}{lll}(q_0, & f_1a, & \texttt{[[}Fy^{\langle 2-\rangle}, (D\varphi x)^{\langle 1-\rangle}, DGAB, S\texttt{]},\\ & & \texttt{[}\varphi x^{\langle 2+\rangle}, H(Fy)x, (D\varphi x)^{\langle 1-\rangle}, DGAB, S\texttt{]},\\ & & \texttt{[}\varphi B^{\langle 1+\rangle}, D(D\varphi x)y(\varphi y), DGAB, S\texttt{]]})\end{array}$$

Again we have labelled a new pair of store items, so that the same principle applies: if we want the missing argument of $Fy^{\langle 2-\rangle}$, then we will be able to find it at $\varphi x^{\langle 2+\rangle}$. The next configuration is now:

$$\begin{array}{rlll}& (q_0, & a, & \texttt{[[}x, Fy^{\langle 2-\rangle}, (D\varphi x)^{\langle 1-\rangle}, DGAB, S\texttt{]},\\ & & & \texttt{[}\varphi x^{\langle 2+\rangle}, H(Fy)x, (D\varphi x)^{\langle 1-\rangle}, DGAB, S\texttt{]},\\ & & & \texttt{[}\varphi B^{\langle 1+\rangle}, D(D\varphi x)y(\varphi y), DGAB, S\texttt{]]})\\ \to & (q_1, & a, & \texttt{[[}Fy^{\langle 2-\rangle}, (D\varphi x)^{\langle 1-\rangle}, DGAB, S\texttt{]},\\ & & & \texttt{[}\varphi x^{\langle 2+\rangle}, H(Fy)x, (D\varphi x)^{\langle 1-\rangle}, DGAB, S\texttt{]},\\ & & & \texttt{[}\varphi B^{\langle 1+\rangle}, D(D\varphi x)y(\varphi y), DGAB, S\texttt{]]})\\ \to & (q_0, & a, & \texttt{[[}y, (D\varphi x)^{\langle 1-\rangle}, DGAB, S\texttt{]},\\ & & & \texttt{[}\varphi x^{\langle 2+\rangle}, H(Fy)x, (D\varphi x)^{\langle 1-\rangle}, DGAB, S\texttt{]},\\ & & & \texttt{[}\varphi B^{\langle 1+\rangle}, D(D\varphi x)y(\varphi y), DGAB, S\texttt{]]})\end{array}$$

However, at this point y is the 3rd argument of the preceding item $(D\varphi x)^{\langle 1-\rangle}$, therefore, we have:

$$\begin{array}{lll}(q_3, & a, & \texttt{[[}(D\varphi x)^{\langle 1-\rangle}, DGAB, S\texttt{]},\\ & & \texttt{[}\varphi x^{\langle 2+\rangle}, H(Fy)x, (D\varphi x)^{\langle 1-\rangle}, DGAB, S\texttt{]},\\ & & \texttt{[}\varphi B^{\langle 1+\rangle}, D(D\varphi x)y(\varphi y), DGAB, S\texttt{]]})\end{array}$$

By rule 6 we arrive at: (in the following $\to_n$ means n steps of $\to$)

$$\begin{array}{rlll}& (q_1, & a, & \texttt{[[}\varphi x^{\langle 2+\rangle}, H(Fy)x, (D\varphi x)^{\langle 1-\rangle}, DGAB, S\texttt{]},\\ & & & \texttt{[}\varphi B^{\langle 1+\rangle}, D(D\varphi x)y(\varphi y), DGAB, S\texttt{]]})\\ \to_1 & (q_0, & a, & \texttt{[[}x, H(Fy)x, (D\varphi x)^{\langle 1-\rangle}, DGAB, S\texttt{]},\\ & & & \texttt{[}\varphi B^{\langle 1+\rangle}, D(D\varphi x)y(\varphi y), DGAB, S\texttt{]]})\\ \to_1 & (q_2, & a, & \texttt{[[}H(Fy)x, (D\varphi x)^{\langle 1-\rangle}, DGAB, S\texttt{]},\\ & & & \texttt{[}\varphi B^{\langle 1+\rangle}, D(D\varphi x)y(\varphi y), DGAB, S\texttt{]]})\\ \to_2 & (q_2, & a, & \texttt{[[}(D\varphi x)^{\langle 1-\rangle}, DGAB, S\texttt{]},\\ & & & \texttt{[}\varphi B^{\langle 1+\rangle}, D(D\varphi x)y(\varphi y), DGAB, S\texttt{]]})\\ \to_2 & (q_2, & a, & \texttt{[[}DGAB, S\texttt{]},\\ & & & \texttt{[}\varphi B^{\langle 1+\rangle}, D(D\varphi x)y(\varphi y), DGAB, S\texttt{]]})\\ \to_2 & (q_0, & \epsilon, & \texttt{[[}e, S\texttt{]},\\ & & & \texttt{[}\varphi B^{\langle 1+\rangle}, D(D\varphi x)y(\varphi y), DGAB, S\texttt{]]})\end{array}$$

Note that we have accepted $h_1 h_3 h_2 f_1 a$ which is incorrect! Their construction only works under the assumption that the grammar is safe. However, the labels we have used lead us to the construction of a machine which can remedy this problem.

5.2 2PDAL: 2PDA with links

Let us suppose that the labels used in the above example $1+, 1-, 2+, \cdots$ were actually part of the alphabet. This would, in general, lead to an infinite alphabet, but let us ignore the finiteness requirement for now. Provided that each time we create a new pair of labels (the $+$ and $-$ part), we ensure they are unique, then these labels provide a way of always jumping to the correct 1-store when we are looking for missing arguments. Why? Because each time we want the missing argument of an item labelled with $n-$, we would simply perform as many pop_2's as necessary until our topmost symbol was labelled with the corresponding $n+$! To see how this would work, let us backtrack to the following configuration in the above example:

$$\begin{array}{lll}(q_3, & a, & \mathtt{[[}(D\varphi x)^{\langle 1-\rangle}, DGAB, S\mathtt{]}, \\ & & \mathtt{[}\varphi x^{\langle 2+\rangle}, H(Fy)x, (D\varphi x)^{\langle 1-\rangle}, DGAB, S\mathtt{]}, \\ & & \mathtt{[}\varphi B^{\langle 1+\rangle}, D(D\varphi x)y(\varphi y), DGAB, S\mathtt{]]})\end{array}$$

Applying the rule we have just said, we will keep performing a pop_2 until we find a corresponding $1+$. This brings us to:

$$\begin{array}{llll} & (q_1, & a, & \mathtt{[[}\varphi B^{\langle 1+\rangle}, D(D\varphi x)y(\varphi y), DGAB\mathtt{]]}) \\ \rightarrow_1 & (q_0, & a, & \mathtt{[[}B, D(D\varphi x)y(\varphi y), DGAB\mathtt{]]})\end{array}$$

which is indeed what we wanted, and the word is rejected, as the only rule for B is $B \xrightarrow{b} e$. In fact, using the same computation, but on the word $h_1 h_3 h_2 f_1 b$ we end up in an accepting state.

Thus, for the remainder of this section, we afford ourselves the luxury of this embellished 2PDA, which we call 2PDA with links, or simply 2PDAL. It is a 2PDA as defined earlier, but we allow ourselves to adorn items with matching labels, as we have done in the previous example. We work under the assumption that each time we create a new pair of labels they are "fresh" and unique.

5.3 Formal definition of 2PDAL

Formally each item of a 2PDAL will now be embellished with labels from the set

$$\{n+ : n \geq 1\} \cup \{n- : n \geq 1\}$$

It is possible for an item to have zero, one or two labels – no other possibilities exist. We write labels as superscripts, as in $a^{\langle\rangle}$ (or simply a), $a^{\langle 3+\rangle}$ and $a^{\langle 3+,4-\rangle}$. These superscripts are sets of at most two elements, ranged over by λ; thus we have $\langle 3+\rangle \cup \langle 4-\rangle = \langle 3+, 4-\rangle = \langle 4-, 3+\rangle$. We shall see that in the case where an item has two labels, one of these will always be a $+$ and the other a $-$.

These labels come in matching pairs; we will see that if there is an item labelled by $m-$ then there will be one labelled by $m+$ (although the converse is not necessarily true). Thus, if one item is labelled by $m-$ and another is labelled by $m+$, this pair will be referred to an ***instance***

$$
\begin{array}{rcll}
(q_0, a, Dt_1 \cdots t_n{}^{\lambda}) & \to & (q_0, \mathsf{push}_1(E)) \text{ if } Dx_1 \cdots x_n \xrightarrow{a} E & (1) \\
(q_0, \epsilon, e) & \to & \mathit{accept} & (2) \\
(q_0, \epsilon, x_j) & \to & (q_j, \mathsf{pop}_1) & (3) \\
(q_0, \epsilon, \varphi_j t_1 \cdots t_n{}^{\lambda}) & \to & (q_0, \mathsf{repl}_1(\varphi_j t_1 \cdots t_n{}^{\lambda \cup \langle m+ \rangle}) \mathbin{;} \mathsf{push}_2 \mathbin{;} \mathsf{pop}_1 \mathbin{;} \mathsf{repl}_1(s_j{}^{\langle m- \rangle})) & \\
 & & \text{where } m \text{ is fresh and } Ds_1 \cdots s_{n'}{}^{\lambda'} \text{ precedes } \varphi_j t_1 \cdots t_n{}^{\lambda}. & (4) \\
1 \le j \le n, (q_j, \epsilon, \$t_1 \cdots t_n{}^{\lambda}) & \to & (q_0, \mathsf{repl}_1(t_j)) & (5) \\
j > n, (q_j, \epsilon, \$t_1 \cdots t_n{}^{\lambda}) & \to & (q_{j-n}, \mathsf{pop}_2(m)) \text{ if } m- \in \lambda & (6)
\end{array}
$$

Figure 2: Transition rules of the 2PDAL, $2PDAL_G$.

of the ***link*** m. Each link m may have several instances, i.e. several such pairs. For such a given pair, the item which gains the $-$ part will be called the ***start point***, whereas that which receives the corresponding $+$ part, the ***end point***. This terminology should coincide with intuition.

In addition to the usual operations of a 2PDA, a 2PDAL has an iterated form of pop_2, parameterised over links m, defined as follows: for s ranging over 2-stores

$$
\mathsf{pop}_2(m)\, s \quad = \quad \begin{cases} s & \text{if } \mathsf{top}_1(s) \text{ has label } m+ \\ \mathsf{pop}_2(m)(\mathsf{pop}_2(s)) & \text{otherwise} \end{cases}
$$

For convenience we write $\mathsf{repl}_1(a)$ as a shorthand for $\mathsf{pop}_1 \mathbin{;} \mathsf{push}_1(a)$ i.e. replacing the top_1 item by a.

Given a 2-grammar G, which is not assumed to be safe, transitions of the corresponding 2PDAL, written $2PDAL_G$, are defined by induction over the set of rules in Fig. 2. The store alphabet is a subset of the (finite) set of all subexpressions of the right hand sides of the productions in G. We assume that each production rule of the grammar assumes the following format:

$$
F\varphi_1 \cdots \varphi_m x_{m+1} \cdots x_{m+n} \xrightarrow{a} E \tag{7}
$$

where the φ's are used for level-1 parameters, and the x's are used for level-0 parameters. Also note that, in Fig. 2, \$ is a place holder for either a non-terminal or variable. As in [KNU02], the automaton works in phases beginning and ending in distinguished states q_i, with some auxiliary states in between. We assume, for the sake of clarity, that these auxiliary states are disjoint from $\{q_i : 0 \le i \le m\}$, where m is the maximum of the arities of any non-terminal or variable occurring in the grammar. If a non-terminal or a variable has type $A_1 \to \cdots \to A_n \to o$, then it is said to have arity n. Thus, when we refer to a state q_i, it is *not* an auxiliary state.

5.4 Proof of correctness

Theorem 5.2. *The language generated by a (possibly unsafe) 2-grammar is accepted by $2PDAL_G$.*

Remark 5.3. We believe this theorem can be extended to unsafe grammars of all levels, but we do not prove this here.

First, some notation. Let s be a 2-store. We define

$$
\begin{array}{rcl}
s' \sqsubseteq s & \iff & s' \text{ is obtained from } s \text{ by performing zero or more } \mathsf{pop}_1\text{'s and } \mathsf{pop}_2\text{'s} \\
s' \sqsubseteq_i s & \iff & s' \sqsubseteq s \text{ and the final action is a } \mathsf{pop}_1 \text{ (hence the ``}i\text{'' for internal)} \\
s' \sqsubseteq_1 s & \iff & s' = \mathsf{pop}_1^n(s) \text{ for some } n > 0 \\
s' \sqsubseteq_2 s & \iff & s' = \mathsf{pop}_2^n(s) \text{ for some } n > 0.
\end{array}
$$

We define a function pm that transforms a 2-store of $2PDAL_G$ to an equivalent stack of a 2PM as follows:

$$
pm(s) \quad = \quad \begin{cases} [S] & \text{if } \mathsf{top}_1(s) = S \\ \mathsf{top}_1(s)_{pm(\mathsf{pop}_1(s))} : pm(\mathsf{pop}_1(s)) & \text{elseif } \mathsf{top}_1(s) : o \\ join(\mathsf{top}_1(s)_{pm(\mathsf{pop}_1(s))}, split(pm(\mathsf{pop}_2(m)s))) & \text{elseif } \mathsf{top}_1(s) \text{ has label } m- \end{cases}
$$

where the auxiliary functions $split$ and $join$ are defined as follows: where t ranges over 2PM stacks

$$
\begin{array}{rcl}
split(t) & = & (\langle u_1, \cdots, u_r \rangle, \mathsf{tail}(t)) \quad \text{where } \mathsf{head}(t) = \varphi u_1 \cdots u_r \\
join(\theta, (\langle u_1, \cdots, u_r \rangle, t)) & = & (\theta.u_1 \cdots u_r) \ : \ t
\end{array}
$$

(Note: The case of $r = 0$ never arises.)

Lemma 5.4. *If (q_i, s) is a reachable configuration of $2PDAL_G$, then the following hold, where $s = [s_1, s_2, \cdots, s_n]$:*

(i) If $s' \sqsubseteq s$ and $\mathsf{top}_1(s') = u^\lambda$ such that $m- \in \lambda$, and $u : \tau$, then there exists unique $s'' \sqsubseteq_2 s'$ such that $\mathsf{top}_1(s'') = \varphi \cdots^{\lambda'}$ and $m+ \in \lambda$ and $\varphi : \tau$.

(ii) If $s' \sqsubseteq_i s$ then $\mathsf{top}_1(s')$ is headed by a non-terminal.

(iii) If $s' \sqsubseteq s$, and $\mathsf{top}_1(s') = D \cdots^\lambda$ then all level-1 arguments of D are present. Furthermore, λ is either $\langle m- \rangle$ for some m (in the case that $D \cdots$ is of type level 1) or it is empty (in the case that $D \cdots : o$).

(iv) If an item t occurs in a 1-store directly atop another of the form $D \cdots^\lambda$, where D is a non-terminal (by the above), then all the variables occurring in t are formal parameters of D.

(v) If $i > 0$ and $\mathsf{top}_1(s) = \$ \cdots^\lambda$, then $\$: (A_1, \cdots, A_n, o)$ for some $n \geq i$.

(vi) If $s' \sqsubseteq s$ and $\mathsf{top}_1(s') = \varphi \cdots^\lambda$ has type level 1, then $m- \in \lambda$ for some m.

Proof. By induction on the number of transitions, where a transition constitutes the application of one rule in Fig. 2. $\square$

Lemma 5.5. *Let (q_i, s) be a reachable configuration of $2PDAL_G$, and let*

$$[\cdots, a^{\lambda \cup \langle m- \rangle}, \cdots, b^{\langle k- \rangle}, \cdots]$$

be a 1-store in s. The end point of the link m is a 1-stack strictly above the end point of the link k.

Proof. Routine induction on the number of transitions. □

Lemma 5.6. *If (q_i, s) is a reachable configuration of $2PDAL_G$ and $s' \sqsubseteq s$ and $s'' = \mathsf{pop}_1(s')$, then $pm(s') = (\mathsf{top}_1(s')_{pm(s'')} \cdot \cdots) : \cdots : pm(s'')$.*

Proof. By induction on the number of transitions, where a transition constitutes the application of one rule. For the inductive step one performs a case analysis on $\mathsf{top}_1(s)$. The only case which requires some care is when we are in state q_0 with $\mathsf{top}_1(s) = \varphi_i u_1 \cdots u_n{}^{\lambda}$; we do this here. If $\mathsf{top}_1(s) = \varphi_i u_1 \cdots u_n{}^{\lambda}$, then by Lemma 5.4(iv) we must have that $s = [s_1, s_2, \cdots, s_k]$ where $s_1 = [\varphi_i u_1 \cdots u_n{}^{\lambda}, D v_1 \cdots v_{n'}{}^{\lambda'}, s_{11}, \cdots, s_{1N}]$. Now, the next transition results in (q_0, t) where $t = [s_0, s_1', s_2, \cdots s_k]$, where

$$\begin{array}{rcl} s_0 & = & [v_i{}^{\langle m- \rangle}, s_{11}, \cdots, s_{1N}] \\ s_1' & = & [\varphi_i u_1 \cdots u_n{}^{\lambda \cup \langle m+ \rangle}, D v_1 \cdots v_{n'}{}^{\lambda'}, s_{11}, \cdots, s_{1N}] \end{array}$$

where m is a fresh label. We now check that the induction hypothesis holds. Certainly, it holds for any $s' \sqsubseteq \mathsf{pop}_2(t)$ (as these have not been affected by the transition). However, we do need to check the cases where:

1. $s' = t$ and the case where
2. $s' \sqsubseteq_1 t$

For (1) we note that:

$$pm(t) = join((v_i)_{pm(\mathsf{pop}_1(t))}, split(pm(\mathsf{pop}_2(m)(t))))$$

However, $pm(\mathsf{pop}_2(m)(t))$ is equivalent to $pm(s)$, as the only difference between $\mathsf{pop}_2(m)(t)$ and s is that the topmost symbol of the former has the extra label $m+$, but it is obvious from the definition of pm, that the extra $m+$ makes no difference to the output. Hence we have:

$$\begin{array}{rcl} pm(t) & = & join((v_i)_{pm(\mathsf{pop}_1(t))}, split(pm(s))) \\ pm(t) & = & ((v_i)_{pm(\mathsf{pop}_1(t))} \cdot \cdots) : (\mathsf{tail}\; pm(s)) \end{array}$$

Now, by the induction hypothesis we have:

$$\begin{array}{rcl} pm(t) & = & ((v_i)_{pm(\mathsf{pop}_1(t))} \cdot \cdots) : (\mathsf{tail}\; ((\varphi_i u_1 \cdots u_{n\, pm(\mathsf{pop}_1(s))} \cdot \cdots) : \cdots : pm(\mathsf{pop}_1(s)))) \\ pm(t) & = & ((v_i)_{pm(\mathsf{pop}_1(t))} \cdot \cdots) : \cdots : pm(\mathsf{pop}_1(s)) \end{array}$$

It is easy to check, with the aid of Lemma 5.5 that $pm(\mathsf{pop}_1(t)) = pm(\mathsf{pop}_1(\mathsf{pop}_1(s)))$. Furthermore, by the induction hypothesis, $pm(\mathsf{pop}_1(s)) = \cdots : pm(\mathsf{pop}_1(\mathsf{pop}_1(s)))$. Hence we have:

$$pm(t) = ((v_i)_{pm(\mathsf{pop}_1(t))} \cdot \cdots) : \cdots : pm(\mathsf{pop}_1(t))$$

As for case (2), this follows easily with the aid of Lemma 5.5.

□

Lemma 5.7. *Let (q_j, s) be a reachable configuration of $2PDAL_G$ for $j > 0$ such that $pm(s) = ((s_0)_{p_0} (s_1)_{p_1} \cdots (s_j)_{p_j} \cdots (s_n)_{p_n}) : \cdots : p_j$. If s_j After a finite number of transitions we will reach a configuration (q_0, s') such that $pm(s') = (s_j)_{p_j} : p_j$. In the case that s_j contains no variables, p_j is dictated by the function pm.*

Proof. Induction with respect to s. For the inductive step we perform a case analysis on $\mathsf{top}_1(s)$. Suppose that $\mathsf{top}_1(s) = \$t_1 \cdots t_n{}^\lambda$ and that $j \leq n$, then the hypothesis is immediate. If, on the other hand we have $j > n$, then it must be the case that that $m- \in \lambda$ for some m, and the next configuration is (q_{j-n}, s') where $s' = \mathsf{pop}_2(m)(s)$. The result now follows by the induction hypothesis and by noting that the $j-n$th argument of $\mathsf{head}\,pm(s')$ equals the jth argument in $\mathsf{head}\,pm(s)$. □

We say a configuration is ***stable*** if it is of the form (q_0, s). A stable transition is one from one stable configuration (q_0, s) to another (q_0, s') such that $(q_0, s) \to \cdots \to (q_0, s')$ and there are no stable configurations in the $\cdots$'s.

Lemma 5.8. *Let G be a level 2 (possibly unsafe) grammar. If (q_0, s) is a reachable configuration of the $2PDAL_G$, then $pm(s)$ is a reachable configuration of the corresponding compacting pointer machine for G.*

Proof. Induction on the number of stable transition. With the aid of Lemma 5.6 and Lemma 5.7, it follows almost immediately. □

6 Simulating 2PDALs by non-deterministic 2PDAs

We have just seen that the incorporation of labels (as names of links) into the stack alphabet will, in general, lead to an infinite alphabet. In this section, we show how these links and the way in which they are manipulated can be simulated by a *non-deterministic* 2PDA.

Let G be a possibly unsafe level-2 grammar and let the corresponding 2PDAL be denoted by $2PDAL_G$. Furthermore, let Γ be the stack alphabet described in the preceding section. Our non-deterministic 2PDA will have stack alphabet $\Gamma \cup \{w^+ : w \in \Gamma\} \cup \{w^- : w \in \Gamma\} \cup \{w^{+/-} : w \in \Gamma\}$. The $+$ and $-$ take on the same role as before, i.e. the $-$ marks the start point of an instance of a link, whereas the $+$ the end point. Note however that they are *anonymous* - they are no longer prefixed with natural numbers. We will explain how to use them shortly.

The crux of this proof lies in understanding how links are manipulated in $2PDAL_G$. We dedicate the following subsection to exactly this.

6.1 The use of links in $2PDAL_G$

Below we list some observations on the way links are manipulated. Recall that labels are preserved with push_2 operations. For example, consider the following 2-store of $2PDAL_G$

$$\begin{bmatrix} [a, & b^{\langle m- \rangle}, & c] \\ [d^{\langle m+ \rangle}, & e, & f] \end{bmatrix}$$

If we perform a push_2, we will have

$$\begin{bmatrix} [a, & b^{\langle m- \rangle}, & c] \\ [a, & b^{\langle m- \rangle}, & c] \\ [d^{\langle m+ \rangle}, & e, & f] \end{bmatrix}$$

We now have two instances of the link m. One connects the upper b to d, and the other connects the lower b to d. By Lemma 5.4(i), if there are two or more instances of a label m, then although each may have a different start point, they all share the same end point, as illustrated by the above example.

Lemma 6.1. *In any run of $2PDAL_G$, if (q_j, s) is a reachable configuration where $top_1(s) = \$t_1 \cdots t_n{}^{\langle m-\rangle \cup \lambda}$ and $j > 0$, then, for all (q_k, s') such that $(q_j, s) \to_+ (q_k, s')$, $top_1(s')$ is not labelled by $m-$.*

Proof. The key to this proof is recognising that pop_2 is never used, only the parameterised form $\mathsf{pop}_2(m)$ is used. There are two cases. If $j \leq n$, then the result follows from Lemma 5.5. However, if $j > n$, note that the next configuration after (q_j, s) is (q_{j-n}, s'') where $\mathsf{top}_1(s'') = \varphi u_1 \cdots u_{n'}{}^{\langle m+\rangle \cup \lambda'}$. By Lemma 5.4(i), no item in s'' can be labelled by $m-$. Furthermore, any new labels introduced will be fresh. □

We say that a link m is ***followed*** (at a configuration (q_j, s)) if $\mathsf{top}_1(s) = \$t_1 \cdots t_n{}^{\langle m-\rangle \cup \lambda}$ for some λ and some $n \geq 0$ such that $j > n$. A corollary of the above lemma is that any given link m is followed at most once.

6.2 Intuition

The important thing to note about $2PDAL_G$ is that not all links are followed. In the example illustrated in Section 5.2, no link apart from 1 was followed. In fact, we may as well not have bothered to label the remaining links! This observation is the key to our construction.

The simulating non-deterministic 2PDA will follow the rules in Fig. 2 almost exactly. The difference is that each time we are about to generate a link (i.e. Case (4) of Fig. 2), we guess whether it will ever be followed in the future or not, and we label the start and end points of the link if and only if we guess that it will be followed. Furthermore, instead of a fresh label m, we simply mark the start point with a $-$ and the end point with a $+$.

A controlled form of guessing

Now this presents a problem of ambiguity. Suppose we find ourselves in a configuration (q, s) where $\mathsf{top}_1(s)$ is labelled by $-$, how can we tell which of the stack items labelled by a $+$ is the *true* end point of this link? (True in the sense that if we did have the ability to name our links as with $2PDAL_G$, the topmost item would have label $m-$ for some m, and the *real* end point would have label $m+$ for the same m.) The answer lies in the use of a *controlled* form of guessing: when guessing whether a link will be followed in the future, the machine will not always be allowed to guess whichever way it pleases; instead we require the guess to be subject to some constraints. We shall see that as a consequence the following invariant can be maintained:

> *Assume that the topmost 1-store has at least one item labelled by $-$. For the leftmost (closest to the top) of these, the corresponding end point can be found in the first 1-store beneath it whose topmost item is marked with a $+$.*[6]

Before formalising the controlled form of guessing, we introduce a definition. Let (q_0, s) be a reachable configuration of $2PDAL_G$ such that

$$\mathsf{top}_2(s) = [\varphi_{j_1} t_1 \cdots t_n^{\lambda}, A_1, \cdots, A_k, \cdots, A_N]$$

where $N \geq 2$. We say that φ_{j_1} ***ultimately refers to*** A_k just if:

(i) For $i = 1, \cdots, k-1$, the j_ith level-1 argument of A_i is a variable $\varphi_{j_{i+1}}$. We remind the reader of the notational convention for level-1 parameters set out in (7).

[6]The invariant is actually stronger than this, but this is sufficient to ensure that the simulation works correctly.

(ii) The j_kth argument of A_k is an application or a non-terminal.

So, for example, using the grammar in Example 2.4, the following is a reachable 2-store for $2PDAL_G$ on input $h_1 h_3 h_3$:

$$\begin{array}{ll}(q_0, & [[\varphi B, (D\varphi x)^{\langle 1-\rangle}, DGAB, S],\\ & [\varphi B^{\langle 1+\rangle}, D(D\varphi x)y(\varphi y), DGAB, S]])\end{array}$$

Here the topmost φ ultimately refers to (the G in) $DGAB$ (in the topmost 1-store).

Suppose that we are in a configuration (q_0, s) of the non-deterministic 2PDA where

$$\mathsf{top}_2(s) = [\varphi t_1 \cdots t_n^?, A_1, \cdots, A_j, \cdots, A_n]$$

where ? may either denote a $-$ or no label at all. Furthermore, suppose that φ ultimately refers to A_j. There are two possibilities:

A. None of the stack items $\varphi t_1 \cdots t_n^?, A_1, \cdots, A_j$ are labelled by a $-$; or

B. There exists a stack item in $\varphi t_1 \cdots t_n^?, A_1, \cdots, A_j$ labelled by a $-$.

In the first case we leave it up to the 2PDA to non-deterministically label $\varphi t_1 \cdots t_n$ (with $+$) in the configuration (q_0, s) as well as its matching partner (with $-$). However, in the second case, we *insist* that the 2PDA label $\varphi t_1 \cdots t_n$ in the configuration (q_0, s) as well as its matching partner, as given in Case (4) of Fig. 2.

Let us illustrate why this maintains the above invariant with an example. Suppose we have the following configuration:

$$\left[\begin{array}{l}[\varphi x_1 x_2, D\varphi x^-, F(F\varphi x)y, G\varphi x^-, E, S]\\ [A^+, \cdots]\\ [B^+, \cdots]\end{array}\right]$$

Note that the topmost stack has two items labelled with a $-$, $D\varphi x$ and $G\varphi x$. By our invariant we know that $D\varphi x$ has end point A^+. And let us suppose that $G\varphi x$ points to B^+. Suppose that the φ of the topmost item ultimately refers to (the subterm $F\varphi x$) in $F(F\varphi x)y$. Furthermore, suppose we go against our controlled form of guessing and allow the machine *not* to label $\varphi x_1 x_2$ and its matching partner. Thus we arrive at

$$\left[\begin{array}{l}[\varphi, F(F\varphi x)y, G\varphi x^-, E, S]\\ [\varphi x_1 x_2, D\varphi x^-, F(F\varphi x)y, G\varphi x^-, E, S]\\ [A^+, \cdots]\\ [B^+, \cdots]\end{array}\right]$$

Note that now $G\varphi x$ is the leftmost item labelled with a $-$. Our invariant has been violated as the real end point of $G\varphi x$ is not A^+.

$$
\begin{array}{rcl}
(q_0, a, Dt_1 \cdots t_n^\lambda) & \to & (q_0, \mathsf{push}_1(E)) \text{ if } Dx_1 \cdots x_n \xrightarrow{a} E \\
(q_0, \epsilon, e) & \to & \text{accept} \\
(q_0, \epsilon, x_j) & \to & (q_j, \mathsf{pop}_1) \\
\\
(q_0, \epsilon, \varphi_j t_1 \cdots t_n) & \to & \left\{ \begin{array}{l} (q_0, \mathsf{repl}_1(\varphi_j t_1 \cdots {t_n}^+) \,;\, \mathsf{push}_2 \,;\, \mathsf{pop}_1 \,;\, \mathsf{repl}_1({s_j}^-)) \\ (q_0, \mathsf{push}_2 \,;\, \mathsf{pop}_1 \,;\, \mathsf{repl}_1(s_j)) \end{array} \right. \\
 & & \text{where Situation A holds and } Ds_1 \cdots {s_{n'}}^{\lambda'} \text{ precedes } \varphi_j t_1 \cdots t_n \\
(q_0, \epsilon, \varphi_j t_1 \cdots {t_n}^\lambda) & \to & (q_0, \mathsf{repl}_1(\varphi_j t_1 \cdots {t_n}^{+\cup\lambda}) \,;\, \mathsf{push}_2 \,;\, \mathsf{pop}_1 \,;\, \mathsf{repl}_1({s_j}^-)) \\
 & & \text{where Situation B holds and } Ds_1 \cdots {s_{n'}}^{\lambda'} \text{ precedes } \varphi_j t_1 \cdots {t_n}^\lambda \\
\\
1 \le j \le n, (q_j, \epsilon, \$t_1 \cdots t_n^\lambda) & \to & \left\{ \begin{array}{ll} (q_0, \mathsf{repl}_1(t_j)) & \text{if } - \notin \lambda \\ \text{abort} & \text{if } - \in \lambda \end{array} \right. \\
j > n, (q_j, \epsilon, \$t_1 \cdots t_n^\lambda) & \to & \left\{ \begin{array}{ll} \text{abort} & \text{if } - \notin \lambda \\ (q_{j-n}, \mathsf{pop}_2^+) & \text{if } - \in \lambda \end{array} \right.
\end{array}
$$

Figure 3: Transition rules of the non-deterministic 2PDA, $2PDA_G$

Penalty for guessing wrongly

The cost of using non-determinism (regardless of how controlled it may be) is that we commit ourselves to following our guesses. When we find out that we have guessed wrongly, we shall have to abort the run. There are two cases. Suppose we find ourselves in a configuration (q_j, s) where $\mathsf{top}_1(s) = \$x_1 \cdots {x_n}^-$ and $j \le n$. The fact that the topmost item is labelled by $-$ means that at some point in the past, we guessed that we would follow this link. But, by Lemma 6.1 we have not yet followed the link nor will we ever follow it in the future. The machine has guessed wrongly, and we abort immediately. Symmetrically if we find ourselves in a configuration (q_j, s) where $\mathsf{top}_1(s) = \$x_1 \cdots x_n$ and $j > n$, then we also abort. Why? The absence of a $-$ label means that at some point in the past we guessed that we would *not* follow this link, but we are now about to turn against our original guess.

6.3 Definition of the non-deterministic 2PDA, $2PDA_G$

Let G be a (possibly unsafe) 2-grammar that generates a string language. The transition rules of the non-deterministic 2PDA, $2PDA_G$, are given in Fig. 3. Note that, again, we assume that production rules of the grammar assume the format given in rule (7). The function pop_2^+ performs a pop_2 and then repeats until the topmost symbol it reads is marked with a $+$. Formally, let s range over 2-stores, we define $\mathsf{pop}_2^+(s) = p(\mathsf{pop}_2(s))$ where

$$
p(s) = \left\{ \begin{array}{ll} s & \text{if } \mathsf{top}_1(s) \text{ has label } + \\ p(\mathsf{pop}_2(s)) & \text{otherwise} \end{array} \right.
$$

Observe that the rules of the simulating non-deterministic 2PDA (Fig. 3) are almost identical to those of the 2PDAL (Fig. 2).

Remark 6.2. In the definition of the transition rules (Fig. 3), in case the top_1 item of the 2-store is headed by a level-1 variable, the 2PDA has to work out whether it is situation A or B. This can be achieved by a little scratch work on the side: do a push_2, inspect the topmost 1-store for as deep as necessary, followed by a pop_2. Alternatively we could ask the oracle to tell us whether it is A or B, taking care to ensure that a wrong pronouncement will lead to an abort.

6.4 Proof of correctness

First some notations. Items labelled with superscripts $-, +$ or $+/-$, will be referred to as ***marked*** items. In particular, if an item is marked with a $-$ or a $+/-$, then we say it is $-$***marked***, indicating that one of its labels is indeed a $-$. Similarly, if an item is marked with a $+$ or a $+/-$, then we say it is $+$***marked***. If the topmost 1-store of s has at least one $-$marked item, then we say that the leftmost of these (closest to the top) items is the ***foremost***. (If the topmost 1-store of s has no $-$marked items, then s has no foremost item.) Let θ be a $-$marked item in a 2-store s. We say that θ is $-$***reachable*** in s just if θ is foremost in s, or if s has a foremost item and θ is $-$reachable in $\mathsf{pop}_2^+(s)$. For example consider the following 2-store:

$$s = \begin{bmatrix} [a, & b^-, & c, & d^-, & e] \\ [f, & g, & h, & i, & j] \\ [k^{+/-}, & l, & m, & n, & o] \\ [p^+, & q, & r^-, & s, & t] \\ [u^+, & v, & w, & x, & y] \end{bmatrix}$$

The foremost item is b. The items b, k and r are the only $-$reachable ones. Note that if we perform a *single* pop_2, then there will be no foremost item. However, with another pop_2, the foremost item will be k.

Finally, we fix a possibly unsafe 2-grammar G that generates a string language. Let (q, s) and (q', s') be reachable configurations of the $2PDA_G$ and $2PDAL_G$ respectively. Intuitively we write $s \precsim s'$ to mean that s simulates s'. Precisely $s \precsim s'$ holds if either both s and s' are the empty 2-store, or the following hold

(i) If $\mathsf{top}_1(s) = \mathsf{top}_1(s') = \bot$, then $pop_2(s) \precsim s'$, and

(ii) If $\mathsf{top}_1(s) = a^\lambda$ and $\mathsf{top}_1(s') = a'^{\lambda'}$ then

a. $a = a'$ and

b. If λ has label $-$ then λ' has label $m-$ for some m. Similarly, if λ has label $+$ then λ' has label $n+$ for some n; and

c. $pop_1(s) \precsim pop_1(s')$

Lemma 6.3. *Let* $(q_0, [[S]]) = (p_0, s_0) \to \cdots \to (p_n, s_n)$ *be a transition sequence of* $2PDA_G$*. Then there exists a unique (modulo renaming of labels) transition sequence* $(p_0, s'_0) \to \cdots \to (p_n, s'_n)$ *of* $2PDAL_G$ *such that for all* $i = 0, \cdots, n$*:*

(i) $s_i \precsim s'_i$

(ii) *If $s \sqsubseteq s_i$ such that $\mathsf{top}_1(s)$ is $-$reachable in s_i, and if $s' \sqsubseteq s'_i$ such that $s \lessapprox s'$, then $\mathsf{pop}_2^+(s) \lessapprox \mathsf{pop}_2(m)(s')$ where $\mathsf{top}_1(s')$ has label $m-$.*

Proof. Induction on the number of transitions. ☐

Proposition 6.4. *If a string is accepted by $2PDA_G$, it is also accepted by $2PDAL_G$.*

Proof. This follows from Lemma 6.3 and the correctness of the 2PDAL. ☐

Proposition 6.5. *If a string is accepted by $2PDAL_G$, it is also accepted by $2PDA_G$.*

Proof. Suppose there is an all-knowing oracle that always tells us *correctly* whether or not we will ever follow a link. We need to check that the controlled guessing does not restrict the choices of the oracle. Recall that when Situation B arises we do not allow our 2PDA a choice: we *always* guess that the link we are about to create will be followed in the future. We must check that the oracle agrees with this decision.

Thus suppose that the current configuration is (q_0, s) and $\mathsf{top}_2(s)$ is the following:

$$[A_1^{\lambda_1}, A_2^{\lambda_2}, \cdots, A_k^-, \cdots, A_{n-1}^{\lambda^{n-1}}, A_n^{\lambda_n}, \cdots]$$

where $A_1 = \varphi_j t_1 \cdots t_m$, and $1 \leq k < n$. Furthermore, suppose that φ_j in A_1 ultimately refers to A_n. According to our rules, after a finite number of transition steps we reach the following configuration:

$$\begin{array}{l}
[t_{A_n}^-, \cdots] \\
[t_{A_n-1}^{+/-}, A_n^{\lambda_n}, \cdots] \\
\vdots \\
[t_{A_k}^{+/-}, A_{k+1}^{\lambda_{k+1}}, \cdots, A_{n-1}^{\lambda_{n-1}}, A_n^{\lambda_n}, \cdots] \\
\vdots \\
[t_{A_2}^{+/-}, A_3^{\lambda_3}, \cdots, A_k^-, \cdots, A_{n-1}^{\lambda^{n-1}}, A_n^{\lambda_n}, \cdots] \\
[\varphi_j t_1 \cdots t_n^{+\cup\lambda}, A_2^{\lambda_2}, \cdots, A_k^-, \cdots, A_{n-1}^{\lambda^{n-1}}, A_n^{\lambda_n}, \cdots] \\
\vdots
\end{array}$$

where t_{A_i} is a subterm of A_i. We must check that the above "forced" choices would tally with those of the oracle. Suppose, for a contradiction, that it does not, and that t_{A_i} for some i is not $-$marked. There are two possibilities:

(1) t_{A_n} is not $-$marked.

(2) t_{A_n} is $-$marked, but there exists an i such that t_{A_i} is not $-$marked for some $i < n$.

For case (1), the proof of Lemma 6.1 says that the link represented by A_k^- has not yet been followed. However, it should not be difficult to see that by not marking t_{A_n} we have eliminated the possibility of ever returning to A_k^-. In particular, this contradicts that the oracle's previous information that we would perform a pop_2^+ at A_k^- (or a duplicate) in the future. For case (2), let $I = \max\{i : A_i \text{ is not } -\text{marked}\}$. Then, according to the oracle, we will perform a pop_2^+ at $t_{A_{I+1}}$, however, the next configuration will be (q_k, s') for $k > 0$ and $\mathsf{top}_1(s') = t_{A_I}$. This is untenable, as by assumption $t_{A_I} = \varphi$ for some level-1 variable φ, thus we must follow a link here unconditionally or we will be stuck. Again, this contradicts the oracle's information that we would not need to follow a link at t_{A_I}. ☐

Combining Propositions 6.4 and 6.5 we can conclude:

Theorem 6.6. *There is an effective transformation of any (possibly unsafe) string-language generating 2-grammar to a non-deterministic 2PDA that accepts the same language.* □

6.5 Some examples

Example 6.7. We demonstrate the non-deterministic 2PDA for the grammar in Example 2.4 on input $h_1h_3h_2f_1a$. Recall that this word does not belong in the language. As before, the automaton starts in the configuration $(q_0, h_1h_3h_2f_1a, [[S]])$. After a few steps the configuration is:

$$(q_0, \quad h_2f_1a, \quad [[\varphi B, D(D\varphi x)y(\varphi y), DGAB, S]])$$

It is now up to the automaton to choose whether or not to label the start and end points of this link. Suppose it chooses to label:

$$\begin{aligned}(q_0, \quad h_2f_1a, \quad & [[(D\varphi x)^-, DGAB, S],\\ & [\varphi B^+, D(D\varphi x)y(\varphi y), DGAB, S]])\end{aligned}$$

After a few more steps the automaton arrives at another configuration where the topmost symbol is headed by a level-1 variable.

$$\begin{aligned}(q_0, \quad f_1a, \quad & [[\varphi x, H(Fy)x, (D\varphi x)^-, DGAB, S],\\ & [\varphi B^+, D(D\varphi x)y(\varphi y), DGAB, S]])\end{aligned}$$

This time suppose it chooses not to label:

$$\begin{aligned}
\to_+ (q_0, \quad a, \quad & [[x, (Fy), (D\varphi x)^-, DGAB, S]\\
& [\varphi x, H(Fy)x, (D\varphi x)^-, DGAB, S],\\
& [\varphi B^+, D(D\varphi x)y(\varphi y), DGAB, S]])\\
\to_+ (q_0, \quad a, \quad & [[y, (D\varphi x)^-, DGAB, S]\\
& [\varphi x, H(Fy)x, (D\varphi x)^-, DGAB, S],\\
& [\varphi B^+, D(D\varphi x)y(\varphi y), DGAB, S]])\\
\to_+ (q_3, \quad a, \quad & [[(D\varphi x)^-, DGAB, S]\\
& [\varphi x, H(Fy)x, (D\varphi x)^-, DGAB, S],\\
& [\varphi B^+, D(D\varphi x)y(\varphi y), DGAB, S]])\\
\to_+ (q_1, \quad a, \quad & [[\varphi B^+, D(D\varphi x)y(\varphi y), DGAB, S]])\\
\to_+ (q_0, \quad a, \quad & [[B, D(D\varphi x)y(\varphi y), DGAB, S]])
\end{aligned}$$

and, as the word $h_1h_3h_2f_1a$ is rejected, as required.

Example 6.8. As another example using the same grammar, let us attempt a computation on the same word $h_1h_3h_2f_1a$ but this time using a different set of guesses.

$$
\begin{array}{lccl}
& (q_0, & h_1h_3h_2f_1a, & [[S]]) \\
\to_+ & (q_0, & h_2f_1a, & [[(D\varphi x), DGAB, S], \\
& & & [\varphi B, D(D\varphi x)y(\varphi y), DGAB, S]]) \\
\to_+ & (q_0, & f_1a, & [[\varphi x, H(Fy)x, (D\varphi x), DGAB, S], \\
& & & [\varphi B, D(D\varphi x)y(\varphi y), DGAB, S]]) \\
\to_+ & (q_0, & f_1a, & [[(Fy)^-, (D\varphi x), DGAB, S] \\
& & & [\varphi x^+, H(Fy)x, (D\varphi x), DGAB, S], \\
& & & [\varphi B, D(D\varphi x)y(\varphi y), DGAB, S]]) \\
\to_+ & (q_0, & a, & [[x, (Fy)^-, (D\varphi x), DGAB, S] \\
& & & [\varphi x^+, H(Fy)x, (D\varphi x), DGAB, S], \\
& & & [\varphi B, D(D\varphi x)y(\varphi y), DGAB, S]]) \\
\to_+ & (q_1, & a, & [[(Fy)^-, (D\varphi x), DGAB, S] \\
& & & [\varphi x^+, H(Fy)x, (D\varphi x), DGAB, S], \\
& & & [\varphi B, D(D\varphi x)y(\varphi y), DGAB, S]])
\end{array}
$$

But now the automaton aborts prematurely. The fact the the topmost symbol is labelled with a $-$ indicates that earlier the automaton guessed it would follow the link here. However, it is now in state q_1 and the first argument *is* present, hence, the automaton guessed incorrectly. There are two other combinations of guesses that can consume the prefix $h_1h_3h_2f_1$; we leave it to the reader to check that neither results in acceptance of $h_1h_3h_2f_1a$.

Example 6.9. Finally, as a last example, we use the same grammar, but this time our input word is $h_1h_3h_3b$. It can easily be checked that this is indeed a word in the language.

$$
\begin{array}{lccl}
& (q_0, & h_1h_3h_3b, & [[S]]) \\
\to_+ & (q_0, & h_3b, & [[\varphi B, D(D\varphi x)y(\varphi y), DGAB, S]]) \\
\to_+ & (q_0, & h_3b, & [[(D\varphi x)^-, DGAB, S] \\
& & & [\varphi B^+, D(D\varphi x)y(\varphi y), DGAB, S]]) \\
\to_+ & (q_0, & b, & [[\varphi B, (D\varphi x)^-, DGAB, S] \\
& & & [\varphi B^+, D(D\varphi x)y(\varphi y), DGAB, S]])
\end{array}
$$

At this point, situation B occurs: φ in the topmost item ultimately refers to (G in) $DGAB$, therefore, our controlled form of guessing insists that the automaton label the link:

$$
\begin{array}{llll}
\to_+ (q_0, & b, & [[\varphi^-, DGAB, S] \\
& & [\varphi B^+, (D\varphi x), DGAB, S] \\
& & [\varphi B^+, D(D\varphi x)y(\varphi y), DGAB, S]]) \\
\to_+ (q_0, & b, & [[G^-, S] \\
& & [\varphi^{+/-}, DGAB, S] \\
& & [\varphi B^+, (D\varphi x)^-, DGAB, S] \\
& & [\varphi B^+, D(D\varphi x)y(\varphi y), DGAB, S]]) \\
\to_+ (q_0, & b, & [[x, G^-, S] \\
& & [\varphi^{+/-}, DGAB, S] \\
& & [\varphi B^+, (D\varphi x)^-, DGAB, S] \\
& & [\varphi B^+, D(D\varphi x)y(\varphi y), DGAB, S]]) \\
\to_+ (q_1, & b, & [[G^-, S] \\
& & [\varphi^{+/-}, DGAB, S] \\
& & [\varphi B^+, (D\varphi x)^-, DGAB, S] \\
& & [\varphi B^+, D(D\varphi x)y(\varphi y), DGAB, S]]) \\
\to_+ (q_1, & b, & [[\varphi^{+/-}, DGAB, S] \\
& & [\varphi B^+, (D\varphi x)^-, DGAB, S] \\
& & [\varphi B^+, D(D\varphi x)y(\varphi y), DGAB, S]]) \\
\to_+ (q_1, & b, & [[\varphi B^+, (D\varphi x)^-, DGAB, S] \\
& & [\varphi B^+, D(D\varphi x)y(\varphi y), DGAB, S]]) \\
\to_+ (q_1, & b, & [[B, (D\varphi x)^-, DGAB, S] \\
& & [\varphi B^+, D(D\varphi x)y(\varphi y), DGAB, S]]) \\
\to_+ (q_1, & \epsilon, & [[\epsilon, B, (D\varphi x)^-, DGAB, S] \\
& & [\varphi B^+, D(D\varphi x)y(\varphi y), DGAB, S]])
\end{array}
$$

And the word is accepted as required.

7 Unrestricted term trees

As mentioned earlier, grammars can be used for generating term trees as well as string languages. We now give the definition of a term-tree generating grammar as introduced by [KNU01, KNU02]. We then examine the relationship between string languages and term trees. We conclude with an application of our level-2 result.

7.1 Definitions

We shall use higher-order grammars as generators of possibly infinite terms, viewed as trees. As in Section 2.1 we consider simple types. Fix a typed alphabet Σ of symbols of type level at most 1 (this is often referred to as a ***signature***). Thus we can think of elements of Σ as function symbols $f : \underbrace{o \to \cdots \to o}_{r \geq 0} \to o$ of arity $\mathsf{ar}(f) = r$.

A Σ-***tree*** is a map $t : T \longrightarrow \Sigma$ where T is a prefix-closed subset of $\{1, 2, 3, \cdots\}^*$, and for $k \geq 0$ whenever $t(w)$ has arity k then w has exactly k-successors in T, which are $w1, \cdots, wk$. We write $\mathcal{T}^\infty(\Sigma)$ for the set of Σ-trees. Thus a Σ-tree is just a possibly-infinite "applicative term" constructed using symbols from Σ. Henceforth we shall identify *finite* Σ-trees with elements of $\mathcal{T}^o(\Sigma)$, and use them interchangeably.

Definition 7.1. A (term-tree generating) ***higher-order grammar*** is a 5-tuple $G = \langle N, V, \Sigma, \mathcal{R}, S \rangle$ such that N is a finite set of *homogeneously* typed non-terminals, with start symbol $S : o \in N$; V is a set of typed variables; Σ is as above; and $\mathcal{R}$ is a finite set of rules of the form:

$$F x_1 \cdots x_m \ \rightarrow \ E$$

where $F : (A_1, \cdots, A_m, o) \in N$, each $x_i : A_i \in V$, and $E \in T^o(N \cup \Sigma \cup \{x_1, \cdots, x_m\})$. As with string-language generating grammars, the notions of the level of a rule and the level of a grammar apply. Furthermore, we say the grammar is ***safe*** if and only if the righthand side of each rule is a safe term-in-context. Also, as we did for the string-language setting we make a proviso that if $F \in N$ has type $(A_1, \cdots, A_m, o)$ and $m \geq 1$, then $A_m = o$. Following the setting suggested [KNU02] we also make the further assumption that for each non-terminal $F \in N$ there exists *exactly one* production rule with F on the left hand side.

Infinite term tree or $\Sigma^\perp$-tree generated by a grammar

We write $\Sigma^\perp$ to mean $\Sigma \cup \{\perp\}$ for a distinguished symbol $\perp : o$. For any $t \in \mathcal{T}^A(N \cup \Sigma)$, we define $t^\perp$, a term in $\mathcal{T}^A(\Sigma^\perp)$, by induction over the following rules:

- $f^\perp = f$ for $f \in \Sigma$
- $F^\perp = \perp$ for $F \in N$
- $(st)^\perp = (s^\perp t^\perp)$ if $s^\perp \neq \perp$, otherwise $(st)^\perp = \perp$

We define $\rightarrow_G$, a binary relation over $\mathcal{T}(N \cup \Sigma)$, by recursion over the structure of t:

- $F t_1 \cdots t_n \rightarrow_G r[t_1/x_1, \cdots, t_n/x_n]$ if there is a rule $F x_1 \cdots x_n \rightarrow r$, with $x_i : A_i$ and $t_i \in \mathcal{T}^{A_i}(N \cup \Sigma)$ for $i = 1, \cdots, n$.
- If $t \rightarrow_G t'$ then $(st) \rightarrow_G (st')$ and $(tr) \rightarrow_G (t'r)$, for applicative terms st and tr of the appropriate types.

For $t \in \mathcal{T}^o(N \cup \Sigma)$ and $s \in \mathcal{T}^\infty(\Sigma^\perp)$, we say that $t \downarrow_G s$ if:

- s is finite and there is a finite reduction sequence $t = t_0 \rightarrow_G \cdots \rightarrow_G t_n = s$
- s is infinite and there is an infinite reduction sequence $t = t_0 \rightarrow_G t_1 \rightarrow_G \cdots$ such that $s = \lim t_n^\perp$

Given a grammar G, with start symbol S, we define the $\Sigma^\perp$-***tree generated by*** G, written $[G]$, by

$$[G] \ = \ \sup\{t \in \mathcal{T}^\infty(\Sigma^\perp) : S \downarrow_G t\}$$

Intuitively, the tree $[G]$ represents all possible computation paths taken.

Remark 7.2. Note that the setting outlined above for a term-tree generating grammar differs from our definition of a string-language generating grammar in the following ways:

1. Rules are no longer labelled.
2. Signature symbols in Σ may now appear on the righthand side of a production rule.
3. We insist that for each non-terminal F there is exactly one production rule with F on the lefthand side.

These are, in fact, not substantial changes. We could easily have defined our string-language generating grammars to conform to items (1) and (2) by insisting that $\Sigma = \Sigma' \cup \{e\}$ where e is the end of word marker, and all symbols in Σ' are of type (o, o). We refer the reader to [dMO04] for a discussion of the possible (but equivalent) definitions of string-language generating higher-order grammars.

However, restriction (3) is necessary in the term-tree setting to ensure that we produce exactly *one* term tree and not a *language* of term trees.

Higher-order pushdown tree automata

We also introduce higher-order pushdown tree automata. The definition we present here is taken from [KNU02].

A ***level-n pushdown tree automaton*** (abbreviated to nPDTA) is a tuple $\langle Q, \Sigma, \Gamma, q_0, \delta \rangle$ where Q, Σ, Γ and q_0 are as in Section 2; however, the transition function is now defined as:

$$\delta : Q \times \Gamma \to ((Q \times Op_n) \cup TreeOp_n)$$

where $TreeOp_n = \{f(p_1, \cdots, p_{\mathsf{ar}(f)}) : f \in \Sigma \wedge p_1, \cdots, p_{\mathsf{ar}(f)} \in Q\}$.

Remark 7.3. Note that the above insists on an injective transition function: for each $(q, Z) \in Q \times \Gamma$ there is exactly one operation to be performed. Hence, a level-n pushdown tree automaton is always, in a sense, deterministic.

A configuration is denoted by a pair (q, s) where $q \in Q$ and s is an n-store, with initial configuration given by $(q, \perp_n)$. We denote by C the set of all configurations. Given a configuration (q, s) we say that $(q, s) \to (p, s')$ if $(q, \mathsf{top}_1(s), (p, \theta)) \in \delta$ where $(p, \theta) \in Q \times Op_n$ and $\theta(s) = s'$. As before, we denote by $\to^*$ the reflexive and transitive closure of $\to$.

Let $t : T \to \Sigma$ be a Σ-tree. A partial function $\rho : T \to C$ defined on an initial fragment of T is referred to as ***partial run*** of the automaton on t if and only the following hold:

- $(q_0, \perp_n) \to^* \rho(\epsilon)$
- If $w \in T$ and $\rho(w) = (q, s)$ then $\delta(q, \mathsf{top}_1(s)) = f(p_1, \cdots, p_{\mathsf{ar}(f)})$ where $t(w) = f$ and $p_1, \cdots, p_{\mathsf{ar}(f)} \in Q$. Furthermore, $(p_i, s) \to^* \rho(wi)$ for $1 \leq i \leq \mathsf{ar}(f)$ when $\rho(wi)$ is defined.

In the case that ρ is total, then it is called simply a ***run***. The tree t is accepted by an automaton if there is a run on t. Note that any given automaton can accept at most one tree.

In the following two sections we consider conversions from string languages to term trees and vice versa. However, in preparation for this, we give a representation of trees by languages.

Branch languages

Let $t : T \to \Delta$ be a Δ-tree. We say that a finite word $a_1 d_1 a_2 d_2 \cdots d_{n-1} a_n$ where each $a_i \in \Delta$ and each $d_i \in \{1, 2, \cdots\}$ is a ***finite branch*** in t if and only if:

1. $d_1 d_2 \cdots d_{n-1} \in T$
2. $a_i = t(d_1 \cdots d_{i-1})$ for $i = 1, 2, \cdots, n$
3. a_n has arity 0.

We say that an infinite word $a_1 d_1 a_2 d_2 \cdots$ is an ***infinite branch*** in t if and only if:

1. $d_1 d_2 \cdots d_n \in T$ for each n
2. $a_i = t(d_1 \cdots d_{i-1})$

Given a branch $a_1 d_1 a_2 d_2 \cdots$ we say that $a_1 a_2 \cdots$ is the ***label*** of the branch.

Following Courcelle [Cou83], given a signature Δ we form a new alphabet:

$$\overline{\Delta} = \{[f, i] : f \in \Delta \wedge 1 \leq i \leq \mathsf{ar}(f)\} \cup \{f : f \in \Delta \wedge \mathsf{ar}(f) = 0\}$$

and we say that $w \in \overline{\Delta}^* \cup \overline{\Delta}^w$ is in the ***branch language*** of $t : T \to \Sigma$ if and only if $w = [a_1, d_1][a_2, d_2] \cdots$ and $a_1 d_1 a_2 d_2 \cdots$ is a (finite or infinite) branch of t. The branch language of a tree t will be denoted by $Branch^{\infty}(t)$. For notational convenience, we will often write f_i instead of $[f, i]$. Thus if $[h, 1][h, 3][h, 2][f, 1]b$ is a word in the branch language, we will write this as $h_1 h_3 h_2 f_1 b$.

Lemma 7.4. *[Cou83] A Δ-tree is completely specified by its branch language.*

7.2 From string languages to term trees

Let G be a (possibly non-deterministic, possibly unsafe) grammar that generates a string language $L(G)$ over the signature Σ. We will convert it into a term-tree generating grammar G_t over the signature $\Sigma \cup \{2, 3, \cdots n\} \cup \{e, r\}$, where n is the maximum level of branching (to be defined later). Intuitively, the tree generated by G_t will represent *all* possible words generated by G, and the nodes labelled by the new signature symbols $\{2, 3, \cdots, n\}$ will represent the points at which choices are allowed to be made.

For example, consider the following grammar:

$$\begin{array}{rcl} S & \xrightarrow{a} & FHAB \\ F\varphi xy & \xrightarrow{b} & E_1 \\ F\varphi xy & \xrightarrow{c} & E_2 \\ F\varphi xy & \xrightarrow{d} & E_3 \end{array} \qquad \begin{array}{rcl} G\varphi x & \xrightarrow{j} & D_1 \\ G\varphi x & \xrightarrow{k} & D_2 \\ Hx & \xrightarrow{g} & C \\ A & \xrightarrow{h} & e \\ B & \xrightarrow{i} & e \end{array}$$

Note there are 3 production rules with the non-terminal F on the left hand side, and 2 for G. Thus whenever we have $Fs_1s_2s_3$ as a term we can *choose* which production rule to apply. Therefore we say that the level of branching for F is 3, (it is 2 for G, and 1 for H, A, B and S). We say that the level of branching for a grammar is n just in case n is the maximum of the levels of branching for each non-terminal. In order to convert the example grammar above into a term-tree generating one, in the sense of [KNU01, KNU02], G_t, we use the signature $\Sigma \cup \{2,3\} \cup \{e,r\}$, where $2 : o^2 \to o$ and $3 : o^3 \to o$, each symbol in Σ is of type $o \to o$ and $e, r : o$ and we have the following production rules:

$$\begin{array}{rcl} S & \to & a(FHAB) \\ F\phi xy & \to & 3(bE_1)(cE_2)(dE_3) \\ G\phi x & \to & 2(jD_1)(kD_2) \\ Hx & \to & gC \\ A & \to & he \\ B & \to & ie \end{array}$$

Finally, in the case where there exists one or more non-terminals M such that M does not occur on the lefthand side of any production rule, we add the rule $M\overrightarrow{x} \to r$, for $\overrightarrow{x}$ of the appropriate type.

It should not be too difficult to see that each "properly ending" branch in $[G_t]$ corresponds to a word in $L(G)$. A ***properly ending*** branch is one that is finite and whose final node is labelled by e. In fact, $[G_t]$ captures exactly those words in $L(G)$:

Lemma 7.5. *$w \in L(G)$ if and only if there exists a finite branch labelled by $w_0a_1w_1a_2 \cdots a_nw_ne$ in $[G_t]$ such that each $a_i \in \Sigma$, $w_i \in \{2,3,\cdots m\}^*$ (where m is the branching level of G) and $w = a_1a_2\cdots a_n$.*

Proof. Obvious. □

Remark 7.6. It should be clear that if the original grammar G is safe, then so is the resulting one, G_t.

Applying KNU's decidability result

By [KNU02] the term tree generated by a safe term-tree generating grammar has a decidable monadic second order (MSO) theory (see [Tho97] for an introduction to MSO logic).

This has obvious repercussions for a *safe* string-language generating grammar G. Because of the above conversion, we obtain a tree $[G_t]$, such that each properly ending branch corresponds to a word in $L(G)$ and vice versa. Therefore, we can obtain several decidability results about the language $L(G)$ by virtue of the MSO decidability of $[G_t]$.

Proposition 7.7. *Some of the decidability results we can obtain are (1) non-emptiness, (2) membership, and, if G is deterministic (3) finiteness.*

Proof. We give only the proof for non-emptiness; the rest can be similarly argued. We make the conversion from G to G_t as outlined above and combine it with the following MSO sentence:

$$\exists y.p_e(y)$$

where p_a is the predicate "is labelled by a" for $a \in \Sigma \cup \{2,3\cdots m\} \cup \{e,r\}$ where m is the level of branching. □

Thus, the above conversion from a grammar G that generates a string language into a grammar G_t that generates a term tree is quite a useful one. In particular, for languages in the OI-hierarchy (see Section 2), we have a new and simple way to prove decidability results.

7.3 From term trees to string language

Let us first introduce the definition of the ∞***-language*** of a string-language generating grammar G, $L^{\infty}(G)$.

Let $G = \langle N, V, \Sigma, \mathcal{R}, S, e \rangle$ be level-n grammar that generates a string language. We say that a derivation sequence $S = P_1 \xrightarrow{a_1} P_2 \xrightarrow{a_2} \cdots$ is ***valid*** if

- it is finite and ends in e; or
- it is infinite.

If $S = P_1 \xrightarrow{a_1} P_2 \xrightarrow{a_2} \cdots$ is a valid derivation sequence, then we say that the word $a_1 a_2 \cdots$ is in the ∞-language of G.

Remark 7.8. Thus, we now allow for a string-language generating grammar to generate ω-words *in addition to* finite words.

In the literature, ω-languages (consisting only of infinite words) are more commonly defined in terms of a machine model with an associated acceptance condition; such as Buchi, Muller, and so on (see e.g. [GTW02]). Thus, although our definition seems somewhat of a departure from this more standard notion, we will see that it is sufficient for our purposes.

Definition 7.9. Let G be a term-tree generating grammar, and let $[G]$ be the resulting term tree over the signature $\Sigma^{\perp}$. We say that a word w, over the alphabet $\overline{\Sigma}$, is in the ***bottomless branch language*** of $[G]$ if and only if:

- w is an infinite word in $Branch^{\infty}([G])$, or
- $w\perp$ is a finite word in $Branch^{\infty}([G])$, or
- w is a finite word in $Branch^{\infty}([G])$ that *does not* end in $\perp$.

Lemma 7.10. *If t is a tree over $\Sigma^{\perp}$ (as above), then t is completely defined by its bottomless branch language.*

Proof. Obvious. $\square$

Lemma 7.11. *Given a term-generating grammar $G = \langle N, V, \Sigma, \mathcal{R}, S \rangle$, we can convert it into a string-language generating grammar G_s such that $w \in L^{\infty}(G_s)$ if and only if w is in the bottomless branch language of $[G]$*

Proof. Let $G_s = \langle N_s, V_s, \overline{\Sigma}, \mathcal{R}_s, S, e \rangle$, where $N_s = N \cup \{S_f : f \in \Sigma\}$ and the set of rewrite rules, $\mathcal{R}_s$, is constructed as follows. For each existing production rule, $F\overrightarrow{x} \to E$, in $\mathcal{R}$ we simply rewrite this as:

$$F\overrightarrow{x} \xrightarrow{\epsilon} E'$$

where E' is E with each occurrence of a signature symbol f replaced by S_f. Furthermore, for each signature constant f of arity $\mathsf{ar}(f) > 0$ we add the production below for each $i = 1, \cdots, \mathsf{ar}(f)$:

$$S_f x_1 \cdots x_{\mathsf{ar}(f)} \quad \xrightarrow{f_i} \quad x_i$$

In case $\mathsf{ar}(f) = 0$, then we simply add

$$S_f \quad \xrightarrow{f} \quad e$$

Finally V_s can easily be inferred from the above. $\square$

Note that for a given term-tree generating grammar G, the resulting grammar G_s will always be deterministic. This is because of the restriction on G that for each non-terminal N there is exactly one production rule with N on the lefthand side.

Example 7.12. As an example, consider the following term-tree generating grammar, with $\Sigma = \{h, g, f, a, b\}$ where $h : (o, o, o, o)$, $f : (o, o, o)$, $g : (o, o)$ and $a, b : o$.

$$\begin{array}{rcl} S & \to & Dgab \\ D\varphi xy & \to & h(D(D\varphi x)y(\varphi y))(H(fy)x)(\varphi b) \\ H\varphi x & \to & \varphi x \end{array}$$

Applying the above conversion gives us the grammar given in Example 2.4.

Applying our result

Unfortunately, our level-2 result offers little in the way of solving either of the key open problems regarding the safety restriction for term-tree generating grammars. In particular, they are, as stated by Knapik *et al.*:

1. Can every term tree generated by an unsafe grammar of level-n be generated by a safe grammar of the same level? Or of another level?
2. Is safety a necessary requirement to ensure MSO-decidability?

The reason we cannot apply our result is due to the introduced non-determinism. As indicated above, we *can* convert a term-tree generating grammar G into a grammar G_s such that $L^\infty(G_s)$ is the bottomless branch language of G. At level 2, by our result, we can even go on to construct a 2PDA such that it accepts $L^\infty(G_s)$. However, the 2PDA will, in general, be non-deterministic and therein lies the problem.

However, as a small consolation we mention two possible avenues that we believe are worthy of investigation. The first is a conjecture, which, if proven correct will show that safety *is* a restriction in terms of generating power for the term-tree setting. The second introduces the temporal logic, existential LTL, and shows that it is decidable for term trees generated by unsafe level-2 grammars by way of our result.

7.4 Urzyczyn's Language: a conjecture

We have shown that the language U is accepted by a non-deterministic 2PDA. We conjecture that it cannot be accepted by a deterministic 2PDA. If our conjecture is true then we will have an example of an inherently unsafe term tree, i.e. an unsafe level-2 grammar whose term tree cannot be generated by a safe level-2 grammar.

Let us apply the conversion from Section 7.2 to the unsafe string-language generating grammar for U in Section 3, to give us a new grammar, G_U. In particular, our signature is $\Sigma = \{\textbf{(} : (o, o),\ \textbf{)} : (o, o),\ * : (o, o),\ 3 : (o, o, o, o),\ e : o,\ r : o\}$ and we have the following production rules:

$$\begin{array}{rcl} S & \to & \textbf{(}DGEEE \\ D\varphi xyz & \to & 3(\textbf{(}(D(D\varphi x)z(Fy)(Fy))\textbf{()}\varphi yx)(*z) \\ Fx & \to & *x \\ E & \to & e \\ G & \to & r \end{array}$$

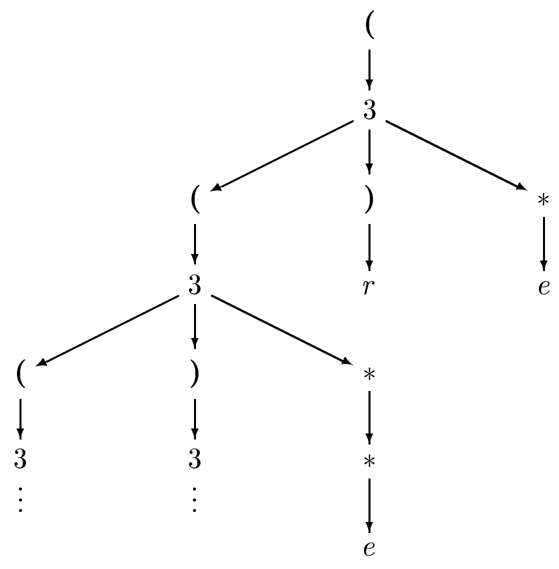

Figure 4: The term tree $[G_U]$

The term tree generated by G_U is shown in Figure 4.

Proposition 7.13. *Suppose that $[G_U]$ can be generated by a safe level-2 (term tree) grammar. Then it must be the case that the language U can be accepted by a deterministic 2PDA.*

Proof. By [KNU02], if $[G_U]$ is indeed generated by a safe level-2 grammar then there exists a 2PDTA $A = \langle Q, \Sigma, \Gamma, q_0, \delta \rangle$ that accepts $[G_U]$.

Analysing $[G_U]$ we can easily see that whenever $\delta(q, a) = 3(p_1, p_2, p_3)$ it must be the case that $p_i \neq p_j$ for $1 \leq i < j \leq 3$ as each immediate successor of 3-node is distinct. It is this property that allows us to convert A into a deterministic pushdown automaton A_s that accepts the language U. Let $A_s = \langle Q_s, \Sigma - \{3, e, r\}, \Gamma, q_0, \delta_s, F_s \rangle$ where let $Q_s = Q \cup \{q_r, q_e\} \cup \{q^a : q \in Q \wedge a \in \Sigma\}$. Furthermore, δ_s is constructed from δ as follows:

- for each $(q, Z, (p, op)) \in \delta$ we add the rule (q, ϵ, Z, p, op) to δ_s.
- if $f \in \Sigma^{(o,o)}$, then for each $(q, Z, f(p_1)) \in \delta$ we add the rule $(q, f, Z, p_1, \mathsf{id})$ to δ_s.
- For every $(q, Z, 3(p_1, p_2, p_n)) \in \delta$, we add the following the family of rules to δ_s:

$$\begin{aligned} \delta(q, \mathbf{(}, Z) &= (p_1^{(}, \mathsf{id}) \\ \delta(q, \mathbf{)}, Z) &= (p_2^{)}, \mathsf{id}) \\ \delta(q, *, Z) &= (p_3^{*}, \mathsf{id}) \end{aligned}$$

  where, for each $a \in \{\mathbf{(}, \mathbf{)}, *\}$, $(p^a, \epsilon, Z, q^a, op) \in \delta_s$ if and only if $(p, Z, (q, op)) \in \delta$ and $(p^a, \epsilon, Z, q, id) \in \delta_s$ if and only if $(p, Z, a(q)) \in \delta$.
- For each $(q, Z, r) \in \delta$ we have $(q, \epsilon, Z, q_r, \mathsf{id}) \in \delta_s$ where q_r is a new state not in F. In particular, $F = \{q_e\}$ where q_e is another new state, and we have $(q, \epsilon, Z, q_e, \mathsf{id}) \in \delta_s$ if and only if $(q, Z, e) \in \delta$.

The key to the construction lies in examining what happens in the 2PDTA when we are in a configuration (q, Z) such that $\delta(q, Z) = 3(p_1, p_2, p_3)$. As mentioned, the p_i's are distinct, thus, in the translation to a deterministic 2PDA, we will ignore the 3, and depending on the current input symbol we change the state to $p_1^{(}, p_2^{)}$ or p_3^*. Note the superscripted states: these are required because we must take care of the fact that we have consumed input "prematurely". In particular, the set of states $Q^{(} = \{q^{(} : q \in Q\}$ are such that the transitions between them match exactly the ϵ-transitions of δ. But we are not allowed to leave $Q^{(}$ until we reach a configuration $(q^{(}, w, s)$ where $(q, top_1(s), ((p_1)) \in \delta$, and, in this case, we leave $Q^{(}$ via an ϵ-transition (p_1, w, s). $\square$

Conjecture 7.14. *U cannot be accepted by a deterministic 2PDA, and hence, there exists a term-tree generated by an unsafe level-2 grammar that cannot be generated by any safe level-2 grammar.*

7.5 Existential LTL

Let $G = \langle N, V, \Sigma, \mathcal{R}, S \rangle$ be a level-2 unsafe term-tree generating grammar. By Section 7.3 we can convert this into a level-2 unsafe string-language generating grammar G_s, such that $L^\infty(G_s)$ is the bottomless branch language of G.

Now, by our level-2 result, we can convert G_s into a safe grammar G_{safe} that generates this same bottomless branch language. In general, however, G_{safe} will contain "dead" non-terminals, such that for each "dead" non-terminal F, there will be no production rule that contains F on the left hand side. These non-terminals arises solely because of abortive computations of the non-deterministic 2PDA caused by incorrect guesses.

Finally, by the conversion[7] in Section 7.2 we obtain a term-tree generating grammar G_t with the following property. Every infinite or properly ending branch in $[G_t]$ corresponds to a word in $L^\infty(G_{safe})$. In particular, each branch in $[G_t]$ that does not end in r[8] corresponds to a branch in $[G]$! Summing up (works only for level 2):

G	unsafe term-tree generating grammar
$\downarrow$	(Section 7.3)
G_s	unsafe string-language generating grammar
$\downarrow$	(Theorem 4.1)
G_{safe}	safe string-language generating grammar
$\downarrow$	(Section 7.2)
G_t	safe term-tree generating grammar

As an example, suppose that $f1f2g1a$ is a branch in $[G]$, this will manifest itself as a branch $w_0 f_1 w_1 f_2 w_3 g_1 w_4 a w_5 e$ in $[G_t]$, where each $w_i \in \{2, 3, \cdots m\}^*$ for some fixed m.

Note that $[G_t]$ *does* possess a decidable MSO theory as it is generated by a safe grammar. Thus, we can certainly model-check certain path properties of $[G]$ via $[G_t]$. As an example, we consider *existential LTL*. In fact, we can probably model-check more properties than these, but these are sufficient to show that some useful model-checking is viable.

[7]It should be clear that the conversion in Section 7.2 also works when we consider ∞-languages instead of languages of finite words.

[8]We intend the r to mean "reject".

The formulae of existential LTL are the following. If Σ is the set of signature symbols of G, then we have the following formulae:

$$\phi := p_f \mid \texttt{true} \mid \texttt{false} \mid \phi \wedge \phi \mid \neg\phi \mid X\phi \mid \phi U \phi$$

where $f \in \Sigma^{\perp}$. X means "next" and U means "until" in the usual sense (see [Var95]). In particular, we say that $[G] \models_{\exists} \phi$ if there exists a path π in $[G]$ such that ϕ holds. Thus, we can state properties such as, "There exists a path such that f never holds," or "There exists a path such that whenever g holds, eventually h holds." Hence the name *existential* LTL. Note that this is useful for model checking safety[9] properties.

To show that such a property ϕ is indeed decidable for $[G]$ all one needs to do is give a corresponding formulae ϕ_t and show that it holds for $[G_t]$.

For example, suppose that we wish to establish that $[G] \models_{\exists} X\phi$. All we need do is decide whether the following MSO formula holds true for $[G_t]$: $\exists X.path(X) \wedge \exists z.z \neq root(X) \wedge z \in X \wedge \phi(z) \wedge (\bigvee_{f\in\Sigma, i\leq \mathsf{ar}(f)} p_{f_i}(z)) \wedge \forall y.(y \in X \wedge (root(X) < y < z) \to \vee_{i\in\{2,3,\cdots m\}} p_i(y))$, where the predicate $path(X)$ means "X is a prefix-closed maximal path that does not end in the signature constant r" and $root(X)$ is the least node in X labelled with a signature constant.

Remark 7.15. If π is a branch in $[G]$ labelled by $a_1 \cdots a_n \perp$ then either

- There exists a branch labelled by $w_0 a_1 w_1 \cdots w_{n-1} a_n w_n \perp$ in $[G_t]$ where $w_0, \cdots, w_n$ are finite words over $\{2, 3, \cdots, m\}$; or
- There exists a branch labelled by $w_0 a_1 w_1 \cdots w_{n-1} a_n w_n$ in $[G_t]$ where $w_0, \cdots, w_{n-1}$ are finite words over $\{2, 3, \cdots, m\}$ and w_n is an infinite word over $\{2, 3, \cdots, m\}$.

Thus, it is possible (and easy) to have a predicate $p_{\perp}$.

8 Further directions

Let us recall our main result. We have shown that the string language of every level-2 grammar (whether safe of unsafe) can be accepted by a 2PDA. Combining this with [DG86] we have proved that every string language that is generated by an unsafe 2-grammar can also be generated by a safe (non-deterministic) 2-grammar. Thus, there are no *inherently* unsafe string languages at level 2. Hence we arrive at the title of our paper: safety is not a restriction at level 2 (at least for string languages). We have also given a small application of our result in the term-tree setting. However, our result leaves many questions unanswered. Some of the most obvious are listed here.

- Does our result extend to levels 3 and beyond?
- What is the relationship between deterministic unsafe grammars and deterministic safe grammars? In particular, can U – which is generated by a deterministic unsafe level-2 grammar – be generated by a deterministic safe grammar and hence accepted by a deterministic 2PDA?
- Is homogeneity a necessary restriction as well?
- Can we extract more decidability results for the term-tree setting? More specifically, is safety a requirement for MSO decidability?

[9] This has nothing to do with the syntactic restriction of safety.

References

[Aho68] A. Aho. Indexed grammars - an extension of context-free grammars. *J. ACM*, 15:647–671, 1968.

[Blu04] A. Blumensath. A pumping lemma for higher-order pushdown automata. preprint, 36 pages, 2004.

[Cau02] D. Caucal. On infinite terms having a decidable monadic theory. In *Proceedings 27th MFCS, Warszawa, 2002*. Springer, 2002.

[Cho59] N. Chomsky. On certain formal properties of grammars. *Information and Control*, 2:137–167, 1959.

[Cou83] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

[Dam82] W. Damm. The IO- and OI-hierarchy. *Theoretical Computer Science*, 20:95–207, 1982.

[DG86] W. Damm and A. Goerdt. An automata-theoretical characterization of the OI-hierarchy. *Information and Control*, 71:1–32, 1986.

[dMO04] J. G. de Miranda and C. H. L. Ong. A note on deterministic pushdown languages. Available at `http://web.comlab.ox.ac.uk/oucl/work/jolie.de.miranda`, 2004.

[Eng91] J. Engelfriet. Interated stack automata and complexity classes. *Information and Computation*, pages 21–75, 1991.

[GTW02] E. Graedel, W. Thomas, and T. Wilke. *Auotmata, Logics, and Infinite Games.* LNCS 2500. Springer-Verlag, 2002.

[KNU01] T. Knapik, D. Niwiński, and P. Urzyczyn. Deciding monadic theories of hyperalgebraic trees. In *TLCA'01*, pages 253–267. Springer, 2001. LNCS Vol. 2044.

[KNU02] T. Knapik, D. Niwiński, and P. Urzyczyn. Higher-order pushdown trees are easy. In *FOSSACS'02*, pages 205–222. Springer, 2002. LNCS Vol. 2303.

[Sti02] C. Stirling. Personal email communication. 15 October, 2002.

[Tho97] W. Thomas. Languages, automata and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3. Springer-Verlag, 1997.

[Urz03] P. Urzyczyn. Personal email communication. 26 July, 2003.

[Var95] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Banff Higher Order Workshop*, pages 238–266. Springer-Verlag, 1995.