On Collapsible Pushdown Automata, their Graphs and the Power of Links

Christopher H. Broadbent

Keble College

University of Oxford

A dissertation submitted for the degree of

Doctor of Philosophy in Computer Science

September 2011

Abstract

Higher-Order Pushdown Automata (HOPDA) are abstract machines equipped with a nested stacks of stacks... of stacks of stacks. Collapsible pushdown automata (CPDA) enhance these stacks with the addition of 'links' emanating from atomic elements to the higher-order stacks below. For trees CPDA are equi-expressive with recursion schemes, which can be viewed as simply-typed λY terms. With vanilla HOPDA, one can only capture schemes satisfying a syntactic constraint called *safety*.

This dissertation begins with some results concerning the significance of links in terms of recursion schemes. We introduce a fine-grained notion of safety that allows us to correlate the need for links of a given order with the imposition of safety on variables of a corresponding order. This generalises some joint work with William Blum that shows we can dispense with homogeneous types when characterising safety. We complement this result with a demonstration that homogeneity by itself does not constrain the expressivity of otherwise unrestricted recursion schemes.

The main results of the dissertation, however, concern the configuration graphs of CPDA. Whilst the configuration graphs of HOPDA are well understood and have decidable MSO theories (they coincide with the Caucal hierarchy), relatively little is known about the transition graphs of CPDA. It is known that they already have undecidable MSO theories at order-2, but Kartzow recently showed that 2-CPDA graphs are tree automatic and hence first-order logic is decidable at order-2. We provide a characterisation of the decidability of first-order logic on CPDA graphs in terms of quantifier-alternation and the order of CPDA stacks and the links contained within. Whilst this characterisation is fairly comprehensive, we do leave open the question of decidability for some sub-classes of CPDA. It turns out that decidability can be highly sensitive to the order of links in a stack relative to the order of the stack itself.

In addition to some strong and surprising undecidability results, we also develop further Kartzow's work on 2-CPDA. We introduce prefix-rewrite systems for nested-words that characterise the configuration graphs of both 2-CPDA and 2-HOPDA, capturing the power of collapse precisely in terms outside of the language of CPDA. It also formalises and demonstrates the inherent asymmetry of the collapse operation. This generalises the rational prefix-rewriting systems characterising conventional pushdown graphs and we believe establishes the 2-CPDA graphs as an interesting and robust class.

Contents

Acknowledgements

I am very grateful to my supervisor Prof. Luke Ong for all of his help and encouragement over the past few years. Many thanks also to Prof. Colin Stirling (Edinburgh) and Prof. Joel Ouaknine (Oxford) for agreeing to be the examiners for this dissertation. I also greatly appreciate the continued support given by my family and friends. I would like to thank EPSRC and the Oxford University Department of Computer Science for funding me via an EPSRC Doctoral Training Award.

Chapter 1

Introduction

> "...objectivity is the most important attribute of science...don't claim that your scientific results are useful. Don't claim that they are more useful than somebody else's scientific results. You are not the right person to judge that. Let other people who use it decide whether it's useful or not. Leave that question open!"
> — *Tony Hoare, speaking at the Marktoberdorf Summer School, 2006*

Theoretical computer science is abound with solutions looking for problems as much as it has problems looking for solutions. Whilst approaches to practical challenges are informed by abstract theory, apparatus developed with a practical end in mind can fill out to form a beautiful mathematical picture.

Recursion schemes and their associated automata are no exception to this phenomenon. Since they are essentially simply typed lambda terms enhanced with a fixed-point combinator, recursion schemes have always had a close connection to higher-order functional computation. Indeed a flurry of work over the last few years has confirmed that they show great promise as a model for approximating higher-order functional programs such as those written in the languages Haskell, OCaML and the increasingly popular .NET language F#. The most recent progress came with Kobayashi's fixed-parameter tractable algorithm for checking a simple but useful class of properties of trees generated by recursion schemes using an intersection type system [55]. Kobayashi implemented his algorithm in the tool TReCS [54] and Lester *et al.* have implemented an extension of the algorithm in their tool THORS [57], which is capable of checking a wider class of properties. Both of these demonstrate that automatically checking interesting properties of recursion schemes is feasible in practise. Another link in the puzzle was provided by Ong and Ramsay [61] in which it was observed that the only essential feature missing from recursion schemes when compared with real functional languages was pattern matching on (co)-algebraic data types. They therefore provide a translation

from real functional programs to an extension of recursion schemes with pattern matching—*pattern matching recursion schemes (PMRS)*—together with an abstraction-refinement loop in the spirit of Clarke *et al.*'s CEGAR [30]. This yields a semi-algorithm based on recursion schemes for verifying real functional programs.

1.1 Theoretical Background

Recursion Schemes and Safety

Recursion schemes also enjoy a good theoretical grounding with a rich history. Maslov [58] introduced a generalisation of Aho's indexed grammars [3]; these generate word-languages which can be arranged into a strict ω-hierarchy with the regular languages at the bottom (zeroth level); the context free languages at the first level and Aho's indexed languages at the second. Notably Maslov showed that the nth level of his hierarchy coincides precisely with the word-languages generated by order-n *nested-stack automata*, a model that is essentially the same as the n-PDA that will be discussed in this dissertation. A 1-PDA is a pushdown automaton in the conventional sense with an ordinary stack, whilst an $(n+1)$-PDA is inductively defined to be equipped with a stack of n-PDA stacks. The higher-order grammars of Damm and Goerdt [32, 33] also coincide with this same hierarchy and indeed can be viewed as similar to recursion schemes constrained by a syntactic condition called *safety*, which we also consider in this piece.

Safety was introduced by Knapik *et al.* [51, 52] in a setting where each recursion scheme is deterministic and generates a single (possibly infinite) tree rather than a word language. They show that order-n safe recursion schemes as tree generators also coincide precisely with n-PDA. The important consequence of safety is that *safe* recursion schemes, viewed as lambda terms, can be evaluated without needing to α-convert variables to avoid variable capture upon β-reduction. This allows them to represent the reduction of a recursion scheme whilst living within a universe containing only a finite number of variables, which in turn leads to the first model-checking result for safe recursion schemes of all orders. The μ-calculus theory of trees generated by order-n safe recursion schemes is decidable and the problem is n-EXPTIME complete [52]. Indeed it follows that the MSO theory of such trees must be decidable [46]. The μ-calculus and MSO are logics subsuming a large number of temporal properties and achieving decidability for them is considered an admirable goal to which a class of structures may aspire.

The Safety Conjecture

This yields the question of μ-calculus model-checking for trees generated by recursion schemes that are not constrained by safety, but there is a question that should be asked before turning our attention to this. Can unsafe recursion schemes generate anything that safe recursion schemes do not? For trees the answer to this question is currently unknown, although it is conjectured that safety does indeed constrain expressivity for tree generation. It is widely believed that there exists a tree generated by an unsafe recursion scheme that cannot be generated by a safe recursion scheme of any order, a hypothesis that has been dubbed *'The Safety Conjecture'*. Some recent progress has been made towards establishing this. Parys demonstrated that the 'Urzycyn tree', which can be generated by an order-2 recursion scheme, cannot be produced by an order-2 safe scheme [62]. This confirms Miranda's conjecture concerning this tree [35]. The analogue of Parys's result for trees does not hold for words. Aehlig *et al.* showed that every word language generated by an order-2 unsafe scheme can also be generated by an order-2 safe scheme (provided that nondeterminism is allowed) [47]. The truth of the more general conjecture for both trees and words remains an open problem.

Model-Checking Recursion Schemes and Traversals

So μ-calculus model-checking for trees generated by unsafe schemes is probably a more general problem than model-checking safe trees. Ong made a breakthrough with this problem in 2006 [60]. Building on previous work [2] at order-2 he showed that μ-calculus model-checking is n-EXPTIME complete for all order-n recursion schemes, regardless of whether they are safe. One of Ong's insights is that the essence of β-reduction can be represented without needing to actually perform these reductions. This avoids the need for an infinite supply of variables that Knapik *et al.* mitigated through the imposition of the safety constraint. Inspired by a notion of game playing on λ-terms used by Stirling when proving his higher-order matching result [67], Ong introduces the idea of the *traversal* of a(n infinite) term. This can be viewed as simulating linear head reduction [34] where one jumps about subterms in a manner that searches for the head variable, repeating the process for the term bound to that head variable. Since the μ-calculus is expressively equivalent to alternating parity automata on trees [37], Ong is able to consider the behaviour of an alternating parity automaton when jumping over traversals and simulate this behaviour using a more sophisticated parity automaton reading the term in a straightforward top-down manner.

We can also view traversals as constituting a form of Hyland-Ong game semantics [44], a type of denotational semantics for programming languages

that has a strong operational flavour. Roughly speaking, a type corresponds to a particular kind of 'game' and a term with that type corresponds to a particular 'strategy' in that game. Indeed Ong makes use of these apparatus in order to build his theory of traversals and his use of it can be viewed as extending the metaphor—a μ-calculus specification acts as a winning condition and terms satisfying the specification denote winning strategies.

Conversely the theory of traversals has fed back into the game semantics literature through the work of Blum [11, 10], in particular giving an account of the implications of safety for the game semantics of a term. It turns out that safety has a very nice characterisation—it corresponds to what Blum describes as *order-incremental* strategies. This work turns out to be highly useful in studying a class of automata corresponding to unsafe (as well as safe) recursion schemes.

Collapsible Pushdown Automata

Assuming the safety conjecture, vanilla higher-order pushdown automata are not as expressive as unrestricted recursion schemes. This motivated the design of *collapsible pushdown automata* [40] that employ higher-order pushdown stacks enriched with 'links' between atomic elements and constituent stacks. The links can be viewed as keeping track of which term is bound to what variable, something that is unnecessary in the safe case since the lack of α-conversion means that bindings, to a greater extent, are determined by the label of the variable in question, which are finite in number. The automata resulting in Knapik *et al's* translation from recursion schemes [52] can be viewed as evaluating the recursion scheme according to its operational semantics—the stack alphabet consists of subterms of the scheme. A similar approach was used by Knapik *et al*'s [53] panic automata, which are essentially the special case of collapsible pushdown automata at order-2. Whilst Carayol and Serre have recently shown that this idea can also work with CPDA more generally [26], the original translation from general recursion schemes makes use of traversals. The CPDA can be viewed as 'computing' the traversal(s) of a recursion scheme.

Against this background, Blum's work can be used to make clear the role of links in terms of game semantics. He provided most of a proof, which was completed in joint work with the author, that the traversal based translation only makes trivial use of links when the recursion scheme is safe and thus essentially yields a non-collapsible higher-order pushdown automaton [9, 18]. This reaches the same result as Knapik *et al.* but also allows for a cleaner definition of safety that depends only on the terms in the recursion scheme and does not assume homogeneous types, as was the case when the notion was introduced.

The automaton models of recursion schemes have provided another avenue

through which to approach model-checking. Deciding whether a μ-calculus property holds on the tree generated by an automaton can be reduced to deciding which of two players has a winning strategy in a *parity game* played on the automaton's configuration graph [37]. Walukiewicz [69] showed that parity games played on pushdown graphs (the configuration graphs of 1-PDA) are decidable in EXPTIME and this progressed to a generalisation for higher-order pushdown graphs [22] and eventually collapsible pushdown graphs [40].

Regular Sets of Stacks

One advantage of higher-order automata is that their stacks can be viewed as words and consequently we can consider automata that act on the stacks themselves. In particular it therefore makes sense to speak of *regular sets* of configurations. For example, Cachat and Serre independently extended Walukiewicz's result on 1-PDA to show that the configurations of the transition graph from which a given player in the parity game has a winning strategy forms a regular set [21, 66]. This is an example of a solution to the so-called *global model-checking* problem, for which one is interested in computing a useful representation of the (possibly infinite) set of all states in a system at which a particular property holds, rather than just asking whether the property holds in a particular state.

Generalisations to higher-order automata include Hague and Ong's extension [41] of Bouajjani and Meyer's [15] saturation method for 'context free processes' (1-PDA). This demonstrates that alternation free μ-calculus sentences define regular sets of configurations in higher-order automata. A generalisation to the full μ-calculus due to Carayol *et al.* [24] makes use of a notion of *abstract pushdown automata* viewing an n-PDA stack as a 1-stack with an infinite alphabet (consisting of $(n-1)$-stacks). This shows that the winning region of an arbitrary parity game on any n-PDA is regular. One pleasing consequence of this regularity is that an n-PDA is able to keep track of state in which a deterministic finite automaton would be upon reading its stack. Viewed as a tree generator, this means that it enjoys *logical reflection* with respect to the μ-calculus—it can be adapted to print out the same tree as before but with the addition of annotations indicating precisely where a particular μ-calculus property holds.

Unlike vanilla higher-order stacks, collapsible stacks cannot be viewed straightforwardly as words, since they have the additional link structure. Serre designed a natural notion of finite automaton that can act on words 'with links' and thereby provide a suitable definition of 'regular set of CPDA configurations'. It turns out that the μ-calculus definable regions of CPDA graphs are regular in this sense and that CPDA also enjoy logical reflection [19].

There are limits to what can be expressed as regular sets of configurations

in this sense. For example it is easy to exhibit an example showing that the set of reachable configurations of an n-PDA (for $n \geq 2$) is not regular. Carayol was partly motivated by this to consider regular sets of configurations where configurations are specified by a canonical sequence of stack operations generating the stack rather than the stack contents itself [23]. Indeed Carayol and Slaats also approached the global μ-calculus model-checking problem in this way [25]. With respect to this notion of regularity the reachable configurations are indeed regular, moreover regular configurations in this sense are precisely those MSO definable in a suitable 'tree like' structure. To some extent a much weaker version of the global model-checking result for CPDA by the author and Ong [20] uses a similar representation of stacks in that collapsible stacks are specified by sequences of operations that generate them. Unfortunately this work lacks the canonicity of the sequences enjoyed by the non-collapsible case treated by Carayol and Slaats. The main reason for this is that there exists a *symmetric* version of n-PDA where every push operation can be assigned a pop operation that forms precisely the reverse of the push-edges in the transition graph. The *collapse* operation handling links in CPDA can have no such symmetric dual. Indeed some results in this dissertation formally exhibit this inherent asymmetry. The utility of the results in [20] is eclipsed by [19] although the former does retain some interest in that it uses traversals and the traversal simulating techniques from Ong's seminal decidability paper [60].

The fact that finite automata can represent configurations of higher-order automata make the latter an attractive vehicle for global model-checking, although for practical applications one would often be more interested in obtaining a representation of a set of lambda terms rather than configurations of an automaton implementing those terms. Some recent work by Salvati and Walukiewicz [65] shows that recursion schemes can be implemented using a Krivine machine [31] and they use this to provide a solution to the global model-checking problem—the set of terms generating the sub-trees of the recursion scheme output that satisfy a given μ-calculus property is decidable. In some respects the Krivine machine implementation is neater than CPDA; its evaluation is more space efficient and it bears a much closer resemblance to the way that functional languages are implemented in practice. Indeed one might argue, as Salvati and Walukiewicz do, that CPDA try to do what the Krivine machine does but in a manner that disregards any kind of garbage collection. The fact that we have a solution to the global model-checking problem expressed as a set of *terms* is also potentially of more practical use.

However, there is currently no corresponding logical reflection result via the Krivine machine method since the operational semantics of a Krivine machine and the terms that it evaluates belong to a paradigm differerent from the mechanism by which sets of terms are represented. By contrast, CPDA share

traits with the devices used to represent sets of their configurations. This allows them to keep track of the way in which the stack-recogniser would behave on its stack at any given point during their execution.

Configuration Graphs

Another reason to retain interest in the automaton model is that the configuration graphs are of intrinsic interest themselves, even when disconnected from recursion schemes. For one thing MSO is strictly more expressive than the μ-calculus on graphs, in contrast to the case for trees. The transition graphs of conventional 1-PDA have decidable MSO theories [59], as do the graphs that result from glueing together nodes connected by a sequence of ϵ-transitions—the ϵ-closure of the graphs [28, 68]. The last result is interesting in that it makes use of the fact that the ϵ-closures of 1-PDA graphs correspond precisely to the graphs generated by Caucal's prefix rewrite systems [28]. This demonstrates that this class of graphs is robust in that it enjoys two natural but distinct characterisations.

It turns out that this robustness extends to all ϵ-closures of n-PDA graphs, which coincide precisely [21, 27] with Caucal's hierarchy of graphs generated by iterated unfolding and inverse rational maps [29]. Indeed Caucal showed that every graph inhabiting his hierarchy has a decidable MSO theory, a result which thereby transfers to n-PDA graphs.

Much less is known about CPDA graphs, and one of the main purposes of this dissertation is to contribute towards filling this gap in understanding. Given the strong properties of n-PDA graphs and the robustness of the hierarchy, this is a very natural line of inquiry. Unfortunately Hague *et al.* noted that there is a CPDA graph even at order-2 that has undecidable MSO theory [40]. Nevertheless, there was hope for some respite with the much weaker first-order logic.

Automatic Structures

Kartzow was the first to make progress on this question. He showed that the ϵ-closures of 2-CPDA graphs are *tree automatic* and consequently have decidable first-order theory [48]. Khoussainov was one of the first to propose the concept of automaticity [50] with other researchers such as Grädel, Blumensath and Rubin generalising the idea [13, 14, 64]. An *automatic structure* is one whose domain can be represented as a set generated by an automaton belonging to a class with good closure properties—such as a finite tree or word automaton. The relations are recognised by automata acting on tuples of the objects (such as trees or words) in a synchronous manner. The good closure properties allow a relation defined by any first-order formula to be represented by such an

	2-CPDA /w ϵ-clos.	3_2-CPDA	3_2-CPDA /w ϵ-clos.	n_n-CPDA ($n \geq 3$)	n_n-CPDA /w ϵ-clos. ($n \geq 3$)	$n_{n,(n-1)}$-CPDA $n \geq 4$	$n_{n,(n-1)}$-CPDA /w ϵ-clos. (all n)	n_m-CPDA ($n \geq 4$, $m \leq n-2$)	n-PDA /w ϵ-clos. (all n)
Σ_1	Dec	**Dec**	**Dec**	**Dec**	**Dec**	?	?	**Und**	Dec
Π_2	Dec	**Dec**	**Und**	**Und**	**Und**	**Und**	**Und**	**Und**	Dec
FO	Dec	**Dec**	**Und**	**Und**	**Und**	**Und**	**Und**	**Und**	Dec
MSO	Und	Und	Und	Und	Und	Und	Und	Und	Dec
μ-calculus	Dec	Dec	Dec	Dec	Dec	Dec	Dec	Dec	Dec

Table 1.1: Summary of the main (un)decidability results known to date. Those in bold are new results of this dissertation, although some others (such as ***FO*** decidability and MSO undecidability for 2-CPDA) have been reproved using the new tools that we develop. The notation $n_{m,m'}$-CPDA means 'n-CPDA with order-m and order-m' links'. Note that the table covers all possible CPDA with and without ϵ-closure.

automaton. Checking a sentence of first order logic on the automatic structure can thus be reduced to the emptiness problem for the class of automata.

One further attraction of an automatic presentation of CPDA is that pumping lemmas on the automata representing relations can be used to construct a form of pumping lemma on paths in the underlying CPDA graph, as Kartzow has demonstrated [49]. He used this technique to reduce the threshold for pumping being applicable compared to previous pumping results for 2-PDA (which are applicable to 2-CPDA due to the equivalence for word languages) due to Hayashi [42] and Gilman [39].

Much of this dissertation continues the inquiry into first-order logic for CPDA as well as providing a refined notion of automaticity demonstrating that the class of 2-CPDA graphs is robust.

1.2 The Contributions of this Dissertation

In Chapter 4 we will exhibit some remarkably strong limits on the decidability of first-order logic for CPDA. Given the weakness of first-order logic, including its inability to recursively define properties (in contrast to the μ-calculus and MSO), this is quite surprising. Undecidability already rears its head with 3-CPDA graphs and for 4-CPDA graphs even the class of sentences with no quantifier alternation whatsoever is undecidable, even though this is one of the least expressive fragments of first-order logic. We believe that the undecidability proofs in the chapter give a good insight into what links and the associated *collapse* operation mean in terms of graph generation.

In Chapter 5 we introduce new techniques for manipulating CPDA graph generators. These include the notion of *derivative CPDA* allowing us to represent configurations of an $(n+1)$-CPDA by runs of an n-CPDA. This makes heavy use of the notion of *monotonic n-CPDA*, which can construct all of its

reachable configurations without performing any operations that destroy an $(n-1)$-stack. This is made possible due to the fact that CPDA enjoy logical reflection, which allows them to know the eventual outcome of destructive order-n operations without having to perform them. In some respects this is analogous to the way in which 'regular tests' are used by Carayol as part of his derivation of a canonical sequence of operations to construct a non-collapsible stack [23]. In particular, we make use of a presentation of the logical reflection result in terms of a new notion of μCPDA, which we introduce as part of the preliminaries in Chapter 2. Chapter 5 uses these techniques to establish some limited decidability results for Σ_1 first-order sentences (without any quantifier alternation) on n-CPDA with restricted links but of arbitrary order (specifically n_n-CPDA and 3_2-CPDA). Whilst these decidability results may appear weak, they should be seen against the background of very strong undecidability elsewhere on the landscape; they make a significant contribution towards reducing the number of fragments of the decision problem with unknown decidability status. The derivative construction is also of intrinsic interest in that it relates automata at one level of the hierarchy to those in the level below. It thus finds a further application in Chapter 7 when we come to describe a correspondence in the progression from dendrisophilic to tree-nondisophilic structures with the construction of 3_2-CPDA from its 2-CPDA derivative.

In Chapter 6 we introduce a novel notion of automaticity based on Alur *et al.*'s *nested-words* and *nested-trees* [7, 4]. Such a notion of automaticity has been considered by Arenas *et al.* [8] but their notion is actually more encompassing whilst our's is intentionally more restrictive. We can vary one of our restrictions in a number of natural ways to obtain multiple classes of automatic structures which we call *isophilic*, *dendrisophilic*, *symmetric dendrisophilic* and *nondisophilic*[1]. All of these classes enjoy decidable first-order theories, a fact which we prove from first principles. Whilst they are subsumed by tree-automatic structures, they remain an interesting subclass in that a reachability relation can always be defined in isophilic and dendrisophilic graphs, which is not true of tree automatic graphs in general.

In Chapter 7 we give what is arguably a more intuitive characterisation of the isophilic and dendrisophilic graphs in terms of prefix rewriting on nested-words. This can be viewed as a natural generalisation of the prefix rewrite systems on vanilla words that correspond to the ϵ-cloures of 1-PDA graphs [28, 68]. Indeed the distinction between dendrisophilic and isophilic can be given a very neat explanation in terms of prefix rewriting. Moreover we show

[1] As the author has been asked on a number of occasions, it is worth mentioning the etymology of these terms! The *isophilic* structures are defined in a manner that has connotations of 'sameness loving' ($\iota\sigma o$); the *dendrisophilic* structures are generated in a manner that additionally has awareness of branching or 'treeness' ($\delta\epsilon\nu\delta\rho o\varsigma$) and the generators of *nondisophilic* structures have more liberal access to *non*determinism.

that the isophilic graphs are *precisely* the ϵ-closures of 2-PDA graphs and that the dendrisophilic graphs are *precisely* the ϵ-closures of 2-CPDA graphs. This gives an account of the power of the *collapse* operation that is very different to that obtained by contrasting the definition of 2-CPDA and 2-PDA and shows that 2-CPDA graphs are a robust class admitting multiple different characterisations. The results also enable one to exhibit the inherent asymmetry of the *collapse* operation, showing the 'symmetric closure' of 2-CPDA graphs to be a strictly larger class than those without symmetric closure being applied. Note that Carayol [23] has a notion of 'generalised prefix rewriting' capturing all of the n-PDA graphs (without *collapse*). Whilst some parallels can be drawn, and indeed Carayol's work suggests ways to further develop our idea (discussed briefly in Chapter 8), we feel that our approach is of a different nature and serves another purpose. Carayol considers 'canonical sequences of stack operations' that in the order-1 case coincide with specifications of a prefix rewrite systems on vanilla words. Our notion of prefix rewriting on nested-words, capturing order-2, can exist naturally without reference to stack operations and also offers a natural variation which just happens to capture the *collapse* operation. To generalise beyond order-2, however, techniques similar to Carayol's generalisation beyond order-1 for non-collapsible automata could be useful, although we do not consider them here.

Our work bears a closer resemblance to that of Kartzow's tree automaticity result for 2-CPDA graphs [48]. Our contribution does not provide any decidability results at order-2 that could not already be derived from those of Kartzow, but rather gives a notion of automaticity that *precisely* captures the class of 2-CPDA and 2-PDA graphs and the difference between them, whereas tree automaticity is an over-approximation for both classes. The finer grained notion of automaticity also has advantages in providing a simpler and more general proof of the automaticity of the reachability predicates and an account of a progression of classes of automatic structures that correspond to a progression up the first 3-levels of the (collapsible) higher-order pushdown hierarchies.

Indeed a final contribution of Chapter 7 consists of some previously unknown decidability results for first-order logic on 3-CPDA graphs. Whilst in full generality the problem is undecidable, there is a natural restriction on 3-CPDA for which the resulting class consists of graphs with decidable first-order theory.

Given that the 'natural restrictions' we place on CPDA usually concern the order of link that are admitted, it is fitting to consider the semantic significance of links in terms of the original motivation for CPDA—generating the same trees as recursion schemes. This is the role of Chapter 3. It can be viewed as a generalisation of the work of Blum, part of which was joint with the author, showing that the traversal based translation from recursion schemes to

n-CPDA yields an n-PDA when the source recursion scheme satisfies the safety constraint. In Chapter 3 we further show that there is an exact correspondence between the orders of links required and the orders of variables with unsafe occurrences. In this way it gives a particularly fine grained analysis of safety in terms of the corresponding automata.

Since Blum's original work also shows that type homogeneity is not required for a definition of safety, it is natural to ask whether type homogeneity constrains expressivity for general (possibly unsafe) recursion schemes. This is not a completely trivial question for recursion schemes as traditionally presented in the literature, since they are applicative. Their terms consequently do not respect σ-equivalence[2] in the way that terms with λ-abstractions can (with λ-abstractions one can always arbitrarily reorder the arguments). At the end of Chapter 3, however, we show that type homogeneity is in fact not a constraint, although our proof goes via CPDA rather than acting on terms directly.

[2] σ-equivalence is a standard relationship between λ-terms capturing the fact that both terms represent essentially the same function but might take their arguments in different orders.

Chapter 2

Preliminaries

The reader may only need to skim most of this chapter in order to fix notation. In particular, (s)he is likely to be familiar with MSO, the modal μ-calculus L_μ, first-order logic $\boldsymbol{FO}$ (and the extension $\boldsymbol{FO}^\infty$ with $\exists^\infty$) with the Σ_i, Π_i and Δ_i quantifier alternation depth hierarchies, and transitive closure logic. In this case it is safe to skip the second section noting that $\boldsymbol{FO(TC)}$ denotes transitive closure logic and $\boldsymbol{FO(TC[\Delta_0])}$ denotes transitive closure logic with transitive closure restricted to transitive closure free Δ_0 formulae $\phi(x, y)$.

Familiarity with higher-order automata and CPDA makes a close reading of the first section unnecessary, although it would be wise to take a look due to our slightly atypical presentation geared towards the graph bias of this dissertation. The reader should, however, make sure (s)he takes note of the definition of μCPDA in the *third* section, which is our new presentation of logical reflection [19].

The third section concerns Ong's traversals for recursion schemes and Blum's work on traversals under the safety constraint. The reader unfamiliar with these ideas might find it instructive to additionally refer to the original sources [60, 40, 11], although for convenience we have provided all necessary definitions here.

2.1 Collapsible Pushdown Automata and their Structures

Collapsible Pushdown Automata (CPDA) were defined in full generality by Hague *et al.* [40], although *panic automata*, which amount to the special case of 2-CPDA were introduced by Knapik *et al.* [53]. A vanilla higher-order pushdown automaton simply contains [58, 52] nested-stacks of stacks. As with a conventional pushdown automaton, atomic elements may be pushed and popped from a stack and the device can only examine the top atomic element. Additionally higher-order push and pop operations are defined which

respectively clone and discard the top stack of a given order. A collapsible pushdown automaton extends this by allowing atomic elements to emanate pointers called *links* that associate themselves with a stack below known as its *target*. The atomic element from which the link originates is described as the link's *source*. A *collapse* operation discards the stack contents between the top atomic element and the target of the link sourced therefrom.

Note that we follow Kartzow [48] in specifying the target of a link as a position relative to the *bottom* of the stack. This allows neater wording of the definition and offers some other advantages in the sequel. On other occasions it is convenient to specify targets relative to the *top* of the stack, as was the case in the original paper [40]. We thus have notation for both.

Higher-Order Stacks

Let us fix a stack-alphabet Γ. For higher-order stacks used by higher-order automata this alphabet must be finite, but it will be convenient in defining higher-order (and indeed collapsible) stacks to consider the more general case where it may be infinite. An *order*-1 stack over Γ is just a string of the form $[\ \gamma\ ]$ where $\gamma \in \Gamma^*$. Let us refer to the set of order-1 stacks over Γ as $stack_1(\Gamma)$. For $n \in \mathbb{N}$ an *order*-$(n+1)$-stack is recursively defined to be a stack of order-n stacks—that is the set of order-$(n+1)$ stacks $stack_{n+1}(\Gamma)$ is recursively defined by:

$$stack_{n+1}(\Gamma) \ := \ stack_1(stack_n(\Gamma))$$

We sometimes write $\bot_1$ to denote the empty 1-stack $[\ ]$ and write $\bot_{n+1}$ to denote the empty $(n+1)$-stack $[\ \bot_n\ ]$. When talking about the 'bottom' and 'top' of a stack we are respectively referring to the left and right of the strings as described. In a (higher-order) stack $[\ s_1 s_2 \cdots s_m\ ]$ we refer to s_i as the *ith component of the stack*.

We allow the following operations on an order-1 stack s for every $a \in \Gamma$:

$$\begin{aligned} push_1^a([\ a_1 \cdots a_m\ ]) &:= [\ a_1 \cdots a_m a\ ] \\ pop_1([\ a_1 \cdots a_m a_{m+1}\ ]) &:= [\ a_1 \cdots a_m\ ] \\ nop(s) &:= s \end{aligned}$$

where *nop* stands for 'no operation'.

We allow the following operations on an order-$(n+1)$ stack s where θ is any operation that may be performed on an order-n stack:

$$\begin{aligned} push_{n+1}([\ s_1 \cdots s_m\ ]) &:= [\ s_1 \cdots s_m s_m\ ] \\ pop_{n+1}([\ s_1 \cdots s_m s_{m+1}\ ]) &:= [\ s_1 \cdots s_m\ ] \\ \theta([\ s_1 \cdots s_m\ ]) &:= [\ s_1 \cdots \theta(s_m)\ ] \end{aligned}$$

We sometimes sequence operations: $\theta_1;\theta_2;\cdots;\theta_k$ meaning that the operations should be executed from left to right. Thus:

$$(\theta_1;\theta_2;\cdots;\theta_k)(s) \;:=\; \theta_k(\cdots(\theta_2(\theta_1(s)))\cdots)$$

for every stack s. We write θ^k for $k \in \mathbb{N}$ to denote the compound operation resulting from applying θ k-times. That is:

$$\theta^k \;:=\; \underbrace{\theta;\theta;\cdots;\theta}_{k \text{ times}}$$

Operations that we may perform on an order-n stack are collectively referred to as *order-n operations*. We also use the notation $top_{k+1}(s)$ to denote the top-most order-k stack in an n-stack s for $0 \leq k < n$ and for these purposes view the atomic elements of 1-stacks as '0-stacks'. We abuse notation and define $top_{n+1}(s) := s$.

We also recursively define the *k-height* of stacks as follows:

Definition 2.1. Let $s := [s_1 s_2 \cdots s_m]$ be an n-stack. Then we define the *n-height* $|s|_n$ of s by $|s|_n := m$. If $1 \leq k < n$, then the *k-height* $|s|_k$ of s is recursively defined by:

$$|s|_k \;:=\; \sum_{i=1}^{m} |s_i|_{k-1}$$

We also define the notion of being a k-prefix of an n-stack (with $1 \leq k \leq n$). Intuitively the k specifies the level of granularity used to determine whether one stack is a prefix of the other.

Definition 2.2. Let $s := [s_1 s_2 \cdots s_m]$ and $t := [t_1 t_2 \cdots t_{m'}]$ be two n-stacks. We say that s is *an n-prefix of* t written $s \sqsubseteq_n t$ just in case $m \leq m'$ and $s_i = t_i$ for $1 \leq i \leq m$. We recursively say that $s \sqsubseteq_k t$ for $k < n$ just in case $m \leq m'$ and $s_i = t_i$ for $1 \leq i < m$ and $s_m \sqsubseteq_k t_m$.

We write $s \sqsubset_k t$ to mean that $s \sqsubseteq_k t$ and $s \neq t$.

Example 2.3. Consider 2-stacks $s := [[ababa][bababa]]$ and $t := [[ababa][bab]]$. Then we have $t \sqsubseteq_1 s$ but we do not have $t \sqsubseteq_2 s$. We also have $|s|_2 = |t|_2 = 2$ but $|s|_1 = 11$ and $|t|_1 = 8$.

In a similar vein we define the *restriction* of a stack. Informally speaking if t is a constituent *occurrence of a stack* in s (and in particular maybe an order-0 stack—*i.e.* an atomic element), then we write $s_{\leq t}$ to mean s where everything above t is deleted.

Definition 2.4. Let $s = [s_1 \cdots s_m]$ be a higher-order stack. Then $s_{\leq s_i} := s = [s_1 \cdots s_i]$ for $1 \leq i \leq m$. If t is an occurrence of a stack in s_i, then $s_{\leq t} := [s_1 \cdots s_{i \leq t}]$. We also have a strict version where $s_{<s_i} := s = [s_1 \cdots s_{i-1}]$ and $s_{<t} := s = [s_1 \cdots s_{i<t}]$.

Collapsible Pushdown Stacks

For collapsible stacks there are additional parameters to play with. In addition to the order of the stack we offer fine control over the orders of links that the stack may contain—an order-$(n+1)$ link is one that targets a component of an order-$(n+1)$ stack, which must be an order-n stack. When describing the order of a stack we often include all of this information by describing it as being order-n_S where $S \subseteq \mathbb{N}$. Here n is the order of the stack, which corresponds to those described in the previous subsection, and S specifies the orders of the links that the stack may (but not necessarily) contain. Collapsible stacks do not have a truly inductive structure in the sense that S may contain an m such that $m > n$. Such links can be described as *dangling* since their targets are external to the order-n stack containing their source. Such objects are not complete collapsible stacks and only make sense when viewed as a constituent component of an order-k stack with $k > m$.

More formally, the *S-collapsible pushdown alphabet* (for $S \subseteq \mathbb{N}$) $\Gamma^{[S]}$ induced by an alphabet Γ is the set $\Gamma \times S \times \mathbb{N}$. The set of order-$n_S$ *open* collapsible stacks $stack^{\mathcal{C}}_{n_S}(\Gamma)$ is defined by:

$$stack^{\mathcal{C}}_{n_S}(\Gamma) \;:=\; stack_n(\Gamma^{[S]})$$

The order of a link of an atomic element $(a, l, p) \in \Gamma^{[S]}$ is given by the number in its second component. If $l \leq n$ the target of a link is the pth component of the l-stack in which the element resides. If $l > n$ we say that the link is 'dangling'.

We say that a stack is a *closed* collapsible stack just in case it belongs to $stack^{\mathcal{C}}_{n_S}(\Gamma)$ where $S \subseteq [1..n]$. As a convention we implicitly assume that $1 \in S$ always holds. Having an order-1 link offers no extra operational power beyond having no link and so having an order-1 link is sometimes described as having *no* link at all. When talking about collapsible stacks we always mean closed collapsible stacks unless talking about a constituent stack of a collapsible stack (*i.e.* an order-k stack contained within an order-n stack with $k < n$) in which case dangling links may be involved.

When we write $top_1(s)$, where s is a collapsible stack with top atomic element (a, l, p), by abuse of notation we usually mean $top_1(s) := a$. If (a, l, p) is intended, it is usually clear from the context. However, we have additional notation to explicitly refer to l.

Definition 2.5. Let s be a collapsible stack with top atomic element (a, l, p), which by abuse of notation we denote a. We then define $\boldsymbol{l_o(a)} := l$ (the order of the link) and $\boldsymbol{l_a(a)} := p$ (the absolute target of the link relative to the bottom of the l-stack in which it resides). It is also useful to describe the target of the link in terms of an offset from the top: $\boldsymbol{l_r(a)} := |t|_l - p$ where t is the l-stack within s in which the occurrence a resides.

The pop_k operations for each $1 \leq k \leq n$; the $push_k$ operations for each $2 \leq k \leq n$ and the *nop* operation are all defined for $stack^{\mathcal{C}}_{n_S}(\Gamma)$ exactly as they are on $stack_{n_S}(\Gamma^{[S]})$. Note in particular that a $push_k$ operation will preserve the absolute targets of links when copying the $k-1$ stack. We replace the $push_1$ operation to allow the attachment of links:

$$push_1^{a,k}(s) \;:=\; push_1^{(a,k,|top_{k+1}(s)|_k-1)}(s)$$

In other words the $push_1^{a,k}(s)$ operation places an atom a on top of s with a link pointing to the $(k-1)$-stack directly below the top $(k-1)$-stack in which a resides. We do not consider the original $push_1$ operation as being a valid operation on collapsible stacks and use it above only to formulate the definition. The *collapse* operation discards everything above the target of a pointer. This can neatly be described in terms of link offset.

$$collapse(s) \;:=\; pop_{\boldsymbol{lo}(top_1(s))}{}^{\boldsymbol{lr}(top_1(s))}$$

So the order-n_S collapsible pushdown stack operations (where $S \subseteq [1..n]$) are $pop_k, push_{k'}, push_1^{a,l}$ and *collapse* for every $1 \leq k \leq n$, $2 \leq k' \leq n$, $l \in S \cup \{1\}$ and $a \in \Gamma$, where Γ is the stack alphabet. We sometimes write $push_1^a$ to mean $push_1^{a,1}$ as attaching a 1-link is operationally analogous to having no link at all (which is not technically admitted by the definition).

Definition 2.6. We write Θ_n to denote the set of order-n collapsible stack operations.

It is sometimes convenient, however, to consider a slightly different model of collapsible stacks where we have just a single $push_1^a$ operation for each $a \in \Gamma$ (as with higher-order pushdown stacks) that creates an element simultaneously emanating links of *all* orders. The order of the link used can then be expressed in the *collapse* operation by having a family of collapse operations: $collapse_1, collapse_2, \ldots, collapse_n$ where the subscript indicates the link on which *collapse* should be performed. When we consider tree generation (but *not* graph generation) both models can easily simulate each other and so we pick whichever is most convenient. The original model are known as *single-link* CPDA and the modified version as *multi-link* CPDA.

For tree generation, single-link CPDA can simulate multi-link CPDA by creating a series of n-atoms for each $push_1^a$, each creating a link of a different order. Conversely multi-link CPDA can simulate single-link CPDA by annotating atomic elements emanating links of all orders with the order of the link that would have been created in the original. With this alternative model it is is useful to define $\boldsymbol{l_{r_k}}(a)$ to be the link offset with respect to the order-k link (so that $collapse_k(s) := pop_k{}^{\boldsymbol{lr_k}(top_1(s))}$).

Another operation that we sometimes add for convenience (but which can be easily simulated in the ϵ-closure) is a $rewrite^a$ operation for each stack symbol

a, which simply rewrites the top-most stack-symbol to a (whilst preserving links). This operation is also added in [40].

One final piece of terminology is the notion of *constructible n-stack* which is an n-stack s for which there exists a sequence of stack operations $\vec{\theta}$ such that $\vec{\theta}(\bot_n) = s$.

The Automata and their Structures

We diverge slightly from the definition of CPDA used elsewhere in the literature, but it amounts to essentially the same thing. Whilst traditionally unary predicates of the graphs generated by CPDA are equated with individual control-states, it will prove useful to have a more abstract notion of predicate as this will allow for isomorphisms between graphs that 'morally exist' to be made explicit.

Definition 2.7. Let $n \in \mathbb{N}$ and let $S \subseteq [1..n]$. An n_S-CPDA (order-n_S collapsible pushdown automaton) $\mathcal{A}$ is a tuple:

$$\left\langle\, \Sigma, \Pi, Q, q_0, \Gamma, R_{a_1}, R_{a_2}, \ldots, R_{a_r}, P_{b_1}, P_{b_2}, \ldots, P_{b_{r'}} \,\right\rangle$$

where Σ is a finite set of transition labels $\{a_1, a_2, \ldots, a_r\}$; Π is a finite set of configuration labels $\{b_1, b_2, \ldots, b_{r'}\}$; Q is a finite set of control-states; $q_0 \in Q$ is an initial control-state; Γ is a finite stack alphabet; each R_{a_i} is the a_i-labelled transition relation with $R_{a_i} \subseteq Q \times \Gamma \times \Theta_{n_S} \times Q$; each P_{b_i} is the b_i-labelled unary predicate specified by $P_{b_i} \subseteq Q \times \Gamma$.

We define n-CPDA and n-PDA in a manner consistent with the standard definitions in the literature:

Definition 2.8. An n-CPDA is an $n_{[n..2]}$-CPDA (recall that 1-links are always allowed) and an n-PDA is an $n_\emptyset$-CPDA.

Graphs

Definition 2.9. A *configuration* of an n_S-CPDA $\mathcal{A}$ is a pair (q, s) where $q \in Q$ and $s \in \mathit{stack}^{\mathcal{C}}_{n_S}(\Gamma)$. We say that *there is an a_i-transition* from a configuration (q, s) to a configuration (q', s') just in case there exists $\theta \in \Theta_{n_S}$ such that $s' = \theta(s)$ and $(q, \mathit{top}_1(s), \theta, q') \in R_{a_i}$. We write this $(q, s)\boldsymbol{a_i}(q', s')$. We say that a configuration (q, s) *satisfies a unary predicate* b_i just in case $(q, \mathit{top}_1(s)) \in P_{b_i}$. We write $\boldsymbol{b_i}(q, s)$ to indicate this.

We are particularly interested in the *reachable* configurations of an n_S-CPDA $\mathcal{A}$. This is the significance of the initial control-state. The *initial configuration* is $(q_0, \bot_n)$ and the reachable configurations $\boldsymbol{R}(\mathcal{A})$ of $\mathcal{A}$ are those (q, s) for which some sequence of transitions can take one from $(q_0, \bot_n)$ to (q, s).

There is also a more general notion of reachability between two configurations along a particular *transition path* in the automaton.

Definition 2.10. Let (q, s) and (q', s') be configurations of an n_S-CPDA $\mathcal{A}$. We say that (q', s') *can be reached from* (q, s) in $\mathcal{A}$ *with path labeled in* $\mathcal{L}$ for some $\mathcal{L} \subseteq \Sigma^*$ just in case:

$$(q, s)\boldsymbol{a_{i_1}}(q_1, s_1)\boldsymbol{a_{i_2}}(q_2, s_2)\boldsymbol{a_{i_3}} \cdots (q_{m-1}, s_{m-1})\boldsymbol{a_{i_m}}(q', s')$$

for some configurations $(q_1, s_1), \ldots, (q_{m-1}, s_{m-1})$ where $a_{i_1}a_{i_2}a_{i_3} \cdots a_{i_m} \in \mathcal{L}$. We write $(q, s)\boldsymbol{r}_{\mathcal{L}}(q', s')$ to mean this. We write $(q, s)\boldsymbol{r}(q', s')$ to mean $(q, s)\boldsymbol{r}_{\Sigma^*}(q', s')$.

The set of *reachable configurations* of $\mathcal{A}$ is given by:

$$\boldsymbol{R}(\mathcal{A}) := \{\ (q, s)\ :\ (q_0, \bot_n)\boldsymbol{r}(q, s)\ \}$$

We are now in a position to define the *configuration graph* of an n_S-CPDA, which we can view as the graph that it *generates*. The nodes of the graph are reachable configurations and the edges transitions with node-labels being provided by the unary predicates.

Definition 2.11. Let $\mathcal{A}$ be an n_S-CPDA with transition-labels Σ and configuration-labels Π. The configuration graph of (graph generated by) $\mathcal{A}$ has domain (set of nodes) $\boldsymbol{R}(\mathcal{A})$, unary predicates Π and directed edges Σ between configurations. We write $\mathcal{G}(\mathcal{A})$ to denote this graph.

Remark 2.12. A configuration graph $\mathcal{G}(\mathcal{A})$ must be connected.

Since we will be concerned with first-order logic over such graphs, which is inherently local, it is interesting to consider a variation of these graphs where edges can be generated by an unbounded number of transitions. For this we give a special status to transitions labelled with an ϵ symbol and consider the notion of ϵ-closure, which 'glues together' $\epsilon^*.a$-labelled paths into a single a-labelled edge. This idea is standard in the literature (indeed is used in the seminal [40]). One important technicality that should be emphasised is that the *initial configuration* of the CPDA does *not* necessarily belong to the ϵ-closure but rather serves to initialise a 'starting node' for each of the connected subcomponents of the possibly disconnected ϵ-closure graph. In [40] this issue was less important as the question concerned playing parity games starting from a particular node in the graph, but for ***FO*** model-checking and providing alternative characterisations of the class of graphs it is crucial. (For example, when Carayol and Wöhrle [27] prove the coincidence of the Caucal hierarchy with the ϵ-closures of higher-order PDA graphs their definition of ϵ-closure deletes the initial configuration if it has only outgoing ϵ-transitions. Our definition is essentially equivalent to their definition in terms of the graphs that can be generated.)

As a convention we write Σ to denote the edge labels without ϵ and Σ^ϵ to denote the set with the ϵ included.

Definition 2.13. The ϵ-closure $\mathcal{G}^\epsilon(\mathcal{A})$ of the configuration graph of an n_S-CPDA $\mathcal{A}$ has domain:

$$\{\ (q,s)\ :\ (q_0,\bot_n)\boldsymbol{r}_{(\Sigma^\epsilon)^*.\Sigma}(q,s)\ \}$$

and we have an a-labelled edge between (q,s) and (q',s') whenever $(q,s)\boldsymbol{r}_{\epsilon^*.a}(q',s')$. Unary predicates apply to configurations in the domain using the same criteria as the configuration graph.

Remark 2.14. Again, unlike $\mathcal{G}(\mathcal{A})$ the ϵ-closure $\mathcal{G}^\epsilon(\mathcal{A})$ might *not* be connected. For example suppose that $(q_0,\bot_n)\boldsymbol{r}_{\epsilon^*.\rho}(q,s)$ and $(q_0,\bot_n)\boldsymbol{r}_{\epsilon^*\rho}(q',s')$ for some edge label ρ but that there are no other transitions. Then the domain of $\mathcal{G}^\epsilon(\mathcal{A})$ will consist of precisely (q,s) and (q',s') and the graph will have no edges whatsoever. In particular the transitions labelled ρ are discarded and any choice of $\rho \neq \epsilon$ would suffice to do the job of populating the domain.

Some further more technical terminology includes the notion of *a slow n_S-CPDA* for which *collapse* operations on n-links, pop_n operations and $push_n$ operations are never associated with an ϵ-transition. It is slow in the sense that the outer-most stack will vary its height by at most one $(n-1)$-stack for every edge in the ϵ-closure (although the top $(n-1)$-stack may still vary by an unbounded amount).

We also refer to *collapse* on an n-link and a pop_n operation in an n_S-CPDA as *destructive* operations. Note that pop_k and *collapse* on k-links for $k < n$ are not covered by this term.

Trees

Traditionally [40] CPDA are defined to explicitly emit nodes making up a tree rather than generating a graph and we now consider this variant.

Definition 2.15. A *raw tree* D is a prefix closed (possibly infinite) subset of $\mathbf{dir}^*$, where $\mathbf{dir} \subseteq \mathbb{N}$ is a set of *directions*. A Σ-labelled tree (where Σ is a finite set of labels) is a map $T\ :\ D\ \longrightarrow\ \Sigma$ where D is a raw tree. We write $\mathbf{dom}(T)$ to denote D, the set of *nodes* of the tree, and $\mathbf{img}(T)$ to denote Σ. The node ϵ is the *root* of the tree and each node $u \in \mathbf{dom}(T)$ of the form $v.i$ with $i \in \mathbf{dir}$ is the ith *child* of the node v. Prefix-maximal elements of $\mathbf{dom}(T)$ are the *leaves* of the tree.

We say that Σ is a *ranked alphabet* if there is a function:

$$rank\ :\ \Sigma\ \longrightarrow\ \mathbb{N}$$

In this case we say that a Σ-labelled tree T is *ranked* just in case for every $u \in \mathbf{dom}(T)$ the node u has $rank(T(u))$ children. In particular note that $\mathbf{dir}$ can be considered finite for ranked trees (since Σ is finite). Usually we take $\mathbf{dir} := [1..k]$ for some k but sometimes we include 0 when convenient (as with computation trees). When we view the ordering of children as significant in distinguishing trees (as induced by the natural ordering on directions derived from the ordering on natural numbers) we say that the tree is *ordered*.

The tree generated by a CPDA is morally the unfolding of the ϵ-closure of its configuration graph. However, this would not be exactly (albeit almost) comparable to the trees generated by recursion schemes, which we introduce later. Restrictions would need to be placed to ensure that this tree is ranked and ordered and further modifications would be needed to ensure that labels appear on the nodes rather than the edges. We thus find it easier to define a tree generating n_S-CPDA differently to those used for graphs.

Definition 2.16. A *tree generating* n_S-CPDA $\mathcal{A}$ is a tuple:

$$\langle\, \Sigma, Q, q_0, \Gamma, \delta \,\rangle$$

where Σ is a finite ranked alphabet; Q is a finite set of control-states; $q_0 \in Q$ is an initial control-state; Γ is a finite stack alphabet and

$$\delta \,:\, Q \times \Gamma \longrightarrow Q \times \Theta_{n_S} \cup \{\, a; q_1 q_2 \cdots q_{rank(a)} \;:\; a \in \Sigma \text{ and } q_i \in Q \text{ for each } 1 \leq i \leq rank(a) \,\}$$

Intuitively the transition function behaves in a deterministic manner as expected, with $a; q_1 q_2 \cdots q_{rank(a)}$ meaning that it should print out an a node for a tree and commence the generation of its ith child in control-state q_i.

Definition 2.17. Let $\mathcal{A}$ be a tree generating n_S-CPDA. Then a *partial run* from a configuration (q_1, s_1) of $\mathcal{A}$ is a sequence of configurations $(q_1, s_1), (q_2, s_2), \ldots, (q_m, s_m)$ such that for each $1 \leq i < m$ we have $\delta(q_i, top_1(s_i)) = (q_{i+1}, \theta_i)$ where $s_{i+1} = \theta_i(s_i)$. We say that $\mathcal{A}$ *emits* $a; (p_1, t)(p_2, t), \cdots (p_{rank(a)}, t)$ from (q_1, s_1) where $a \in \Sigma$ and the (p_i, t) are configurations just in case there exists such a partial run from (q_1, s_1) such that $\delta(q_m, top_1(s_m)) = a; p_1 p_2 \cdots p_{rank(a)}$ and $t = s_m$.

We now define the tree generated by an n_S-CPDA, which is the same as that traditionally used:

Definition 2.18. Let $\mathcal{A}$ be a tree generating n_S-CPDA with initial control-state q_0. The *tree generated* by $\mathcal{A}$ from a configuration (q, s) (written $[\![\mathcal{A}]\!]_{(q,s)}$) has root labelled a where (q, s) emits $a; (p_1, t)(p_2, t) \cdots (p_{rank(a)}, t)$ and the subtree at the ith child is $[\![\mathcal{A}]\!]_{(p_i,t)}$. The tree $[\![\mathcal{A}]\!]$ generated by $\mathcal{A}$ is $[\![\mathcal{A}]\!]_{(q_0,\perp_n)}$.

Note we usually do not explicitly state whether a CPDA is a tree or graph generator as it should be obvious from the context.

Some other Notation

It will often be useful to add some kind of 'dummy element' $\perp$ to a set. For a set Q we write $Q^{\perp} := Q \uplus \{\perp\}$ where $\uplus$ is the disjoint-union operator (which takes a union and stipulates that the sets being merged have no element in common). We will also want to 'project' sequences/strings of elements and in particular stacks. Projection in this context means two things, which are performed simultaneously. Some elements in the sequence/string/stack may be deleted and others may be converted. So given $\Gamma \subseteq \Gamma'$ where $S_1 \times S_2 \times \cdots \times \Gamma \times \cdots \times S_m \subseteq \Gamma'$ and a sequence/string/higher-order stack over the alphabet Γ', we write $\pi_{\Gamma}(s)$ to denote the result of projecting tuples in the sequence containing a component in Γ onto Γ and then deleting all remaining elements without a Γ component.

More generally if s is a stack over an alphabet Γ and we have a map $L : \Gamma \longrightarrow \Gamma'$ we write $L(s)$ to denote the stack that results from replacing every atomic element $a \in \Gamma$ with $L(a)$ (whilst preserving links in the case of a collapsible stack).

Given an element of a Cartesian product $t := (q_1, q_2, \ldots, q_m)$ we also write $\pi_i(t)$ to denote q_i for $1 \leq i \leq m$.

2.2 Logics for Graphs

In this section we recall a number of logics used to make assertions about the trees and graphs generated by CPDA. The principle results presented in this dissertation are to do with first-order logic. However, the apparatus used to obtain them assert properties of CPDA graphs in a number of other logics including the modal μ-calculus à la Kozen [56], Monadic Second Order Logic (MSO) and (fragments of) Transitive Closure Logic [45].

In each case the language of the logic is defined over a *signature* of the form $\mathfrak{S} = \langle\, \boldsymbol{a_1^2}, \boldsymbol{a_2^2}, \ldots, \boldsymbol{a_k^2}, \boldsymbol{b_1^1}, \boldsymbol{b_2^1}, \ldots, \boldsymbol{b_{k'}^1} \,\rangle$ where the a_i^2 are symbols for binary relations and the b_j^1 for unary predicates. Each language can then be interpreted over a graph with edges labels a_i^2 and sets of nodes assigned to the b_j^1. For the purposes of this section, let us assume that such a signature $\mathfrak{S}$ is fixed.

The μ-Calculus

The language L_μ is defined by the following grammar:

$$\phi \ ::= \ X \mid \mu X.\phi^{Px} \mid [\boldsymbol{a}]\phi \mid \boldsymbol{b} \mid (\phi \wedge \psi) \mid \neg\phi$$

where ϕ, ψ range over L_μ formulae, X over an infinite number of set variables, $\boldsymbol{a}$ over edge labels (in the signature) and $\boldsymbol{b}$ over unary predicate symbols (in the signature) and ϕ^{Px} over formulae in which all free occurrences of X occur

positively (that is within the scope of an even number of negations $\neg$). The μ 'least fixpoint' operator is the sole variable binder and the notions of free and bound variables are as expected.

An L_μ formula $\phi(X_1, \ldots, X_k)$ with free variables $X_1, \ldots, X_k$ defines a set of nodes in a graph $\mathcal{G}$ with respect to an *environment* σ assigning a set of nodes $\sigma(X)$ to each *free* variable X (we assume w.l.o.g. that all bound variables with distinct binders have distinct names which in turn are all distinct from the names given to free variables). This is defined recursively as follows, where N is the set of nodes of $\mathcal{G}$:

$$\begin{aligned}
[\![X]\!]_{\mathcal{G},\sigma} &:= \sigma(X) \\
[\![\boldsymbol{b}]\!]_{\mathcal{G},\sigma} &:= \boldsymbol{b} \text{ as set by } \mathcal{G} \\
[\![[\boldsymbol{a}]\phi]\!]_{\mathcal{G},\sigma} &:= \{\, u \in N \;:\; \text{for every } u \text{ s.t. } u\boldsymbol{a}u' \; u' \in [\![\phi]\!]_{\mathcal{G},\sigma} \,\} \\
[\![\mu X.\phi(X)]\!]_{\mathcal{G},\sigma} &:= [\![\phi(X)]\!]_{\mathcal{G},(\sigma \,\cup\, [X \mapsto F])} \\
&\quad \text{where } F \text{ is smallest set s.t. } F = [\![\phi(X)]\!]_{\mathcal{G},(\sigma \,\cup\, [X \mapsto F])} \\
[\![(\phi \wedge \psi)]\!]_{\mathcal{G},\sigma} &:= [\![\phi]\!]_{\mathcal{G},\sigma} \cap [\![\psi]\!]_{\mathcal{G},\sigma} \\
[\![\neg\phi]\!]_{\mathcal{G},\sigma} &:= N - [\![\phi]\!]_{\mathcal{G},\sigma}
\end{aligned}$$

where $(\sigma \;\cup\; [X \mapsto F])$ denotes the valuation formed from σ by adding a mapping for the variable X to F (which by assumption does not already occur in σ). As is standard we additionally define a modality $< \boldsymbol{a} > \phi := \neg[\boldsymbol{a}]\neg\phi$ and a connective $(\phi \vee \psi) := \neg(\neg\phi \wedge \neg\psi)$.

A *sentence* is a formula with no free variables. For $u \in N$ we then write $\mathcal{G}, u \models \phi$ to mean that $u \in [\![\phi]\!]_{\mathcal{G}}$ where ϕ is a sentence (and so the environment is not necessary/ is empty). We pronounce this 'ϕ is true at the node u in $\mathcal{G}$'. We write L^0_μ for the set of μ-calculus sentences.

The purpose of the fixpoint binder μ is to allow recursive definitions. The reader unfamiliar therewith might find helpful a survey such as Bradfield and Stirling's [16].

Monadic Second Order Logic

Monadic Second Order Logic (MSO) is a predicate logic with quantifiers ranging over elements of the domain N of a graph as well as quantifiers ranging over sets of nodes from its domain. It is defined by the following gammar with respect to the signature $\mathfrak{S}$:

$$\phi \;::=\; x\boldsymbol{a}y \mid x = y \mid \boldsymbol{b}x \mid \exists x.\phi \mid x \in X \mid \exists X.\phi \mid \neg\phi \mid (\phi \wedge \psi)$$

where x, y range over first-order variables (bound to elements in the domain) and X ranges over monadic second-order variables (bound to sets of elements from the domain), $\boldsymbol{a}$ ranges over binary relations (edge labels) and $\boldsymbol{b}$ over

unary predicates. The semantics are as expected. If we have an environment σ assigning elements of N to first-order variables and elements of 2^N to second-order variables (with the same generality preserving assumptions as with the μ-calculus) we define:

$$\begin{aligned}
\mathcal{G}, \sigma \models x\boldsymbol{a}y &\Leftrightarrow \sigma(x)\boldsymbol{a}\sigma(y) \text{ in } \mathcal{G} \\
\mathcal{G}, \sigma \models x = y &\Leftrightarrow \sigma(x) = \sigma(y) \\
\mathcal{G}, \sigma \models \boldsymbol{b}x &\Leftrightarrow \boldsymbol{b}\sigma(x) \text{ in } \mathcal{G} \\
\mathcal{G}, \sigma \models x \in X &\Leftrightarrow \sigma(x) \in \sigma(X) \\
\mathcal{G}, \sigma \models \exists x.\phi &\Leftrightarrow \text{exists } u \in N \text{ s.t. } \mathcal{G}, \sigma \cup [x \mapsto u] \models \phi \\
\mathcal{G}, \sigma \models \exists X.\phi &\Leftrightarrow \text{exists } S \in 2^N \text{ s.t. } \mathcal{G}, \sigma \cup [X \mapsto S] \models \phi \\
\mathcal{G}, \sigma \models (\phi \wedge \psi) &\Leftrightarrow \mathcal{G}, \sigma \models \phi \text{ and } \mathcal{G}, \sigma \models \psi \\
\mathcal{G}, \sigma \models \neg\phi &\Leftrightarrow \mathcal{G}, \sigma \not\models \phi
\end{aligned}$$

Again an MSO sentence is a formula with no free variables and for sentences σ is unnecessary. In practise for a formula $\phi(x_1, \ldots, x_m, X_1, \ldots, X_{m'})$ with free variables $x_1, \ldots, x_m, X_1, \ldots, X_{m'}$ we write $\mathcal{G} \models \phi(u_1, \ldots, u_m, S_1, \ldots, S_{m'})$ to mean:

$$\mathcal{G}, [x_1 \mapsto u_1, \ldots, x_m \mapsto u_m, X_1 \mapsto S_1, \ldots, X_{m'} \mapsto S_{m'}] \models \phi$$

The MSO *theory* of a graph $\mathcal{G}$ is the set of MSO *sentences* ϕ such that $\mathcal{G} \models \phi$. We say that the MSO theory of a class of graphs (such as the n-PDA graphs) is *decidable* just in case there exists an algorithm (Turing machine) that takes pairs of the form $(\mathcal{G}, \phi)$ as input with $\mathcal{G}$ a graph in the class and ϕ an MSO sentence, such that the algorithm is guaranteed to terminate and correctly output whether or not $\mathcal{G} \models \phi$. This decision problem is called the *(local) model-checking problem* for MSO on the class of graphs.

We say that $R \subseteq N^m$ (an m-ary relation on nodes of the graph) is *definable* by MSO in $\mathcal{G}$ just in case there exists an MSO formula with m free first-order variables $\phi(x_1, \ldots, x_m)$ such that:

$$R = \{ (u_1, \ldots, u_m) \in N^m : \mathcal{G} \models \phi(u_1, \ldots, u_m) \}$$

We use the usual abbreviations $(\phi \vee \psi) := \neg(\neg\phi \wedge \neg\psi)$ and $\forall x.\phi := \neg\exists x.\neg\phi$ together with the analogue for second-order universal quantification.

Note that whilst equality is definable in MSO, we include explicit syntax so that first-order logic can be viewed as a sublanguage.

First-Order Logic

First-Order Logic is the subset of MSO that does not contain second-order set variables. As it will be our main point of focus we write ***FO*** to denote the

language of first-order logic. We also have a variant $\boldsymbol{FO}^{\infty}$ which includes an additional quantifier $\exists^{\infty}$ that may bind (first-order) variables. This is defined by:

$$\mathcal{G}, \sigma \models \exists^{\infty} x.\phi \Leftrightarrow \mathcal{G}, \sigma \cup [x \mapsto u] \models \phi \text{ for infinitely many } u$$

Decidability of model-checking $\boldsymbol{FO}^{\infty}$ is one of the hallmarks of automatic structures [14], which we will consider in a later chapter.

One way of measuring the expressive strength of first-order formulae is by the depth of its *quantifier alternation*. If one has a formula of the form $\exists \vec{x}.\phi(\vec{x})$ where we existentially quantify over multiple variables contained in the vector $\vec{x}$ then it does not matter how we order the variable bindings from amongst the elements of $\vec{x}$. In contrast, if we alternate universal and existential quantification as in $\forall \vec{y}.\exists \vec{x}.\phi(\vec{x}, \vec{y})$, then the choice of witnesses for the existentially quantified variables must *depend* on the current value taken for those that are universally quantified. Thus increasing the number of alternations between existential and universal quantification increases the number of dependencies and hence, in some sense, the 'expressive power' of the formula.

To formalise this, recall the well-known fact that all first-order formulae are logically equivalent to one in *prenex normal form* which has the form:

$$\forall \vec{x}_1 \exists \vec{x}_2 \cdots \forall \vec{x}_{m-1} \exists \vec{x}_m . \phi(\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_{m-1}, \vec{x}_m)$$

where ϕ contains no quantifiers. We make use of the well-known hierarchy defined as follows:

$$\begin{aligned} \Sigma_0 := \Pi_0 &:= \{ \phi \in \boldsymbol{FO} \ : \ \phi \text{ contains no quantifiers } \} \\ \Sigma_{i+1} &:= \{ \phi \in \boldsymbol{FO} \ : \ \phi = \exists \vec{x}.\psi \text{ s.t. } \psi \in \Pi_i \} \\ \Pi_{i+1} &:= \{ \phi \in \boldsymbol{FO} \ : \ \phi = \forall \vec{x}.\psi \text{ s.t. } \psi \in \Sigma_i \} \end{aligned}$$

More generally we say that a formula is Σ_i or Π_i if it is logically equivalent to a formula in Σ_i or Π_i respectively as defined above. We describe a property of a *class* of graphs (such as the class of n-CPDA graphs) as Δ_i if it is *both* Σ_i and Π_i—so in particular $\Delta_0 = \Sigma_0 = \Pi_0$.

The concepts of $\boldsymbol{FO}$ and $\boldsymbol{FO}^{\infty}$ theory and model-checking problem are defined analogously as for MSO. When we wish to restrict the theory to sentences of a particular alternation complexity we will specify the logical complexity class—for example we will speak of the Σ_1-theory of a class of graphs to talk about the theory restricted to Σ_1 sentences (which thus makes the decision problem less demanding as its inputs are restricted).

Transitive Closure Logic

One final logic that we will employ is *transitive closure logic* $\boldsymbol{FO(TC)}$. This is first-order logic extended with a binary predicate symbol $\overline{\phi(x,y)}$ for every

$\phi(x,y) \in \boldsymbol{FO}$ with precisely two free variables, which represents the transitive closure of the relation defined by $\phi(x,y)$. Note that here we do not allow nesting of transitive closure. More formally, in a graph $\mathcal{G}$ with domain N the predicate $\overline{\phi(x,y)}$ is interpreted as the relation:

$$\{\ (u,v) \in N^2\ :\ \mathcal{G} \vDash \phi(u,w_1),\ \mathcal{G} \vDash \phi(w_i,w_{i+1}),\ \mathcal{G} \vDash \phi(w_m,v)\ \text{ for every } 1 \leq i \leq m \text{ for some } w_1,w_2,\ldots,w_m \in N \text{ where possibly } m=0\ \}$$

We will also make use of the fragment of $\boldsymbol{FO(TC)}$ for which transitive closure may only be applied to Δ_0 (quantifier free) formulae, which we write $\boldsymbol{FO(TC[\Delta_0])}$. Note that for both $\boldsymbol{FO(TC)}$ and $\boldsymbol{FO(TC[\Delta_0])}$ we may describe quantifier alternation exactly as with $\boldsymbol{FO}$, noting that $\overline{\phi(x,y)}$ should be viewed as an atomic relation. Thus we have Σ_i-$\boldsymbol{FO(TC)}$ and Π_i-$\boldsymbol{FO(TC)}$ (and indeed Δ_i-$\boldsymbol{FO(TC)}$) and Σ_i-$\boldsymbol{FO(TC[\Delta_0])}$ and Π_i-$\boldsymbol{FO(TC[\Delta_0])}$ and Δ_i-$\boldsymbol{FO(TC[\Delta_0])}$.

2.3 μ-calculus sensitive CPDA

μ-calculus Model-Checking

The *global model-checking problem* for the μ-calculus on CPDA asks one to compute a representation of *all* nodes in the graph where the sentence holds. A detailed study of Global Model-Checking results for CPDA is beyond the scope of this dissertation. That said, we will make very heavy use of the strongest global CPDA model-checking result to date [19]. One of the consequences detailed in *op. cit.* tells us a CPDA generating a tree can be modified so as to annotate the tree-nodes satisfying a given μ-calculus sentence.

The μ-calculus is bisimulation invariant, which roughly speaking means it cannot tell the difference between a graph at a particular node and the unfolding of the graph at that node; indeed the μ-calculus is precisely the bisimulation invariant fragment of MSO [46]. This fact tends to blur the distinction between model-checking the tree generated by a CPDA and model-checking its configuration graph. As it happens, the logical reflection transformation preserves the stack structure of the CPDA at any given point in its run and the sequences of stack operations that it must perform in any given position also take the same form. We introduce μCPDA to describe this same result in terms with a 'more intensional' focus than those used in the original paper.

The Extended Model

An n_S-μCPDA is a device that has an n_S-CPDA at its disposal but may intervene and manipulate it beyond its normal course depending on whether the

original n_S-CPDA satisfies various μ-calculus sentences in its current configuration.

Definition 2.19. An n_S-μCPDA $\mathcal{B}$ is a tuple:

$$\left\langle\ \Sigma, \Pi, Q, q_0, \Gamma, R'_{a_1}, R'_{a_2}, \ldots, R'_{a_r}, P'_{b_1}, P'_{b_2}, \ldots, P'_{b_{r'}}, R_{a_1}, R_{a_2}, \ldots, R_{a_r}, P_{b_1}, P_{b_2}, \ldots, P_{b_{r'}}\ \right\rangle$$

where $\left\langle\ \Sigma, \Pi, Q, q_0, \Gamma, R_{a_1}, R_{a_2}, \ldots, R_{a_r}, P_{b_1}, P_{b_2}, \ldots, P_{b_{r'}}\ \right\rangle$ is an n_S-CPDA called *the underlying n_S-CPDA*; $R'_{a_i} \subseteq L^0_\mu \times \Theta_{n_S} \times Q$ and $P'_{b_j} \in L^0_\mu$ for each $1 \leq i \leq r$ and $1 \leq j \leq r'$.

As with CPDA configurations are elements of $Q \times \mathit{stack}^{\mathcal{C}}_{n_S}(\Gamma)$ but the only transitions allowed are specified by the R'_{b_i} with reference to the R_{b_j} rather than by the R_{b_j} themselves. Likewise P'_{b_i} are the only unary predicates it has.

Definition 2.20. Let $\mathcal{B}$ be an n_S-μCPDA with underlying n_S-CPDA $\mathcal{A}$. Let $(q, s), (q', s')$ be configurations of $\mathcal{B}$ (and hence also of $\mathcal{A}$). There is an a_i labelled transition from (q, s) to (q', s') in $\mathcal{B}$ just in case $\mathcal{G}(\mathcal{A}), (q, s) \models \phi$ where $(\phi, \theta, q') \in R'_{a_i}$ and $s' = \theta(s)$. Likewise we have (q, s) satisfying the predicate b_i just in case $\mathcal{G}(\mathcal{A}), (q, s) \models P'_{b_i}$.

Given these transition edges and predicates of $\mathcal{B}$ the graphs $\mathcal{G}(\mathcal{B})$ and $\mathcal{G}^\epsilon(\mathcal{B})$ are defined in the same way as with conventional CPDA.

Example 2.21. Consider a standard order-1 pushdown automaton that has control-states $\{\ q_0, q_1\ \}$ and stack alphabet $\{\ a, b\ \}$. Give it has a transition relation $R_c := \{\ (q_0, \mathit{pop}_1, q_0)\ \}$ and predicates $P_a := \{\ (q_0, a)\ \}$, $P_b := \{\ (q_0, b)\ \}$.

Suppose that we extend this to a 1-μPDA with a sole μPDA transition relation $R'_c := \{\ ((\mu X.(\boldsymbol{a} \vee [\boldsymbol{c}]X) \wedge \boldsymbol{b}), \mathit{push}^a_1, q_1)\ \}$. Then this μPDA will have a c-labelled transition from the configuration $(q_0, [bbaaabbbbbb])$ to the configuration $(q_1, [bbaaabbbbbba])$ but *no other* transitions from this configuration.

The μ-calculus sentence asserts that the current configuration has b on top of the stack but that repeated popping will yield a on top.

Strong Isomorphisms

Two graphs are said to be isomorphic if *qua* graphs they are essentially the same. As expected the formal definition is as follows:

Definition 2.22. Let $\mathcal{G}$ and $\mathcal{G}'$ be graphs sharing a signature $\mathfrak{S}$ with respective node sets N and N'. We say that $\mathcal{G}$ and $\mathcal{G}'$ are *isomorphic*, written $\mathcal{G} \cong \mathcal{G}'$ just in case there is a bijection $f : N \longrightarrow N'$ (called an isomorphism) such that for every $u \in N$ and unary predicate $\boldsymbol{b}$ of $\mathfrak{S}$ interpreted as $\boldsymbol{b}_{\mathcal{G}}$ in $\mathcal{G}$ and $\boldsymbol{b}_{\mathcal{G}'}$ in $\mathcal{G}'$ we have $u \in \boldsymbol{b}_{\mathcal{G}}$ iff $f(u) \in \boldsymbol{b}_{\mathcal{G}'}$ and for every edge $\boldsymbol{a}$ we have $u\boldsymbol{a}u'$ in $\mathcal{G}$ iff $f(u)\boldsymbol{a}f(u')$ in $\mathcal{G}'$.

It is well known that the theories in all logics we have introduced are invariant under isomorphism—a sentence will hold in a graph $\mathcal{G}$ just in case it holds in all isomorphic graphs $\mathcal{G}'$.

Note that every CPDA $\mathcal{A}$ can be viewed as a μCPDA $\mathcal{B}$. We simply take $\mathcal{B}$ to have underlying CPDA $\mathcal{A}$ and give it a predicate for every control-state/stack-alphabet pair in $Q \times \Gamma$ to facilitate a μ-calculus sentence asserting that we are currently in a particular control-state with a particular symbol on top of the stack. This allows us to reconstruct the original transition relation of $\mathcal{A}$ in $\mathcal{B}$. It thus follows that for every CPDA $\mathcal{A}$ there exists a μCPDA $\mathcal{B}$ such that $\mathcal{G}^\epsilon(\mathcal{B}) \cong \mathcal{G}^\epsilon(\mathcal{A})$.

For CPDA there is a stronger notion of isomorphism where stack *structure* and control-states are preserved as well. This will be the form of isomorphism to which we usually appeal. In particular the definition makes sense when comparing μCPDA and CPDA.

Definition 2.23. Let $\mathcal{A}$ and $\mathcal{A}'$ be n_S-μCPDA (and in particular either or both may be an n_S-CPDA). We say that $\mathcal{G}(\mathcal{A})$ and $\mathcal{G}(\mathcal{A}')$ (*resp.* $\mathcal{G}^\epsilon(\mathcal{A})$ and $\mathcal{G}^\epsilon(\mathcal{A}')$) are *strongly isomorphic* just in case there is an isomorphism L between the graphs where for any configuration (q, s) of $\mathcal{A}$ we can define L by an expression of the form $L(q, s) := ((L_1(q), L_2(s)), L_3(s))$, where $(L_1(q), L_2(s))$ is a control-state consisting of two components, one depending entirely on the control-state and the other entirely on the stack (with L_1 and L_2 both being injective), and $L_3(s)$ is an *injection* that preserves the *structure* of the stack (it may replace an occurrence of an atom with another, but may not delete occurrences nor may it change links).

We write $\mathcal{G}(\mathcal{A}) \mathrel{\widehat{\cong}} \mathcal{G}(\mathcal{A}')$ (*resp.* $\mathcal{G}^\epsilon(\mathcal{A}) \mathrel{\widehat{\cong}} \mathcal{G}^\epsilon(\mathcal{A}')$) to indicate this.

Note that whilst L is a bijection between the domains of each graph, in the ϵ-transition there may be intermediate control-states accessed during the course of ϵ-transitions that do not appear in nodes of the ϵ-closure of the graph. Therefore L_1 may not be a bijection between control-states. Nevertheless, for control states belonging to the ϵ-closure (on which L_1 *must be* a bijection) we adopt the convention '$L_1(q) := q$'. The correponding convention for stacks '$L_3(s) := s$' is not used as it would be highly misleading; two occurrences of a symbol a in s may map to different symbols in $L_3(s)$. However, since $L_2(s)$ and $L_3(s)$ both depend on s (and by injectivity on each other) we conflate them into one object; this is safe as each stack operation on $L_3(s)$ to form $L_3(s')$ can be viewed in this conflated object as including a transition from $L_2(s)$ to $L_2(s')$. We thus express $L(q, s)$ as $L(q, L(s))$.

Representing as Conventional CPDA

Just as every n_S-CPDA can be viewed as an n_S-μCPDA it turns out that the converse holds as well. Indeed we can view the main result of [19] as saying precisely this.

Theorem 2.24. *Given any n_S-μCPDA $\mathcal{B}$ there exists an n_S-CPDA $\mathcal{A}$ such that $\mathcal{G}(\mathcal{B}) \widehat{\cong} \mathcal{G}(\mathcal{A})$ and so also $\mathcal{G}^\epsilon(\mathcal{B}) \widehat{\cong} \mathcal{G}^\epsilon(\mathcal{A})$.*

Proof. Let $\mathcal{A}^-$ be the n_S-CPDA underlying $\mathcal{B}$ with control-states $Q_{\mathcal{A}^-}$. Extend $Q_{\mathcal{A}^-}$ with a fresh distinguished control-state $\star$. Add fresh distinguished edges $\hat{q}$ for every $q \in Q_{\mathcal{A}^-}$ from the configuration $(\star, s)$ to the configuration (q, s) for every stack s and have a transition $\boldsymbol{\theta}$ for every n_S-stack operation connecting $(\star, s)$ to $(\star, \theta(s))$. Making $\star$ the initial state call the resulting automaton $\mathcal{A}^\star$.

Now let $\phi_1, \phi_2, \ldots, \phi_m$ be a list of all of the μ-calculus sentences occurring in transition relations of $\mathcal{B}$. Let ϕ_i^q be the μ-calculus sentence $[\hat{q}]\phi_i$ for every $q \in Q_{\mathcal{A}^-}$ and $1 \leq i \leq m$. Logical reflection for CPDA, as established in [19], allows us to construct an automaton $\mathcal{A}^\star_{LR}$ such that there exists an isomorphism $f : \mathcal{G}^\epsilon(\mathcal{A}^\star) \cong \mathcal{G}^\epsilon(\mathcal{A}^\star_{LR})$ and additionally there is a set $S_{[\hat{q}]\phi_i} \subseteq Q_{\mathcal{A}^\star_{LR}} \times \Gamma_{\mathcal{A}^\star_{LR}}$ such that a configuration (p, t) of $\mathcal{A}^\star_{LR}$ satisfies $\mathcal{G}^\epsilon(\mathcal{A}^\star_{LR}), (p, t) \vDash [\hat{q}]\phi_i$ just in case $(p, top_1(t)) \in S_{[\hat{q}]\phi_i}$. That is $\mathcal{A}^\star_{LR}$ generates the same ϵ-closure as $\mathcal{A}^\star$ but is also 'aware' of what μ-calculus properties are satisfied at each configuration.

Note that we do not quite have a strong isomorphism here. Whilst [19] tells us the stacks either side of the isomorphism satisfy the required structural similarity, the control-state in the image of the isomorphism depends on the stack in the input configuration as well as the control-state without a guarantee that this dependence can be split into an independent pair of the form $(L_1(q), L_2(s))$. That said, the converse does hold: the control-state in the image determines the control-state in the input of the isomorphism. In particular it is well-defined to delete all control-states from $\mathcal{A}^\star_{LR}$ that are not associated with $\star$ in $\mathcal{A}^\star$. We also remove all edges other than those of the form $\boldsymbol{\theta}$ for $\theta \in \Theta_n$. Call the resulting automaton $\mathcal{A}^{\star\star}_{LR}$.

Now we construct the n-CPDA $\mathcal{B}$ to have the same control-states $Q_{\mathcal{A}}$ as $\mathcal{A}$ and the stack-alphabet of $\mathcal{A}^{\star\star}_{LR}$. In order to simulate $\mathcal{A}$ when in control-state q it may do the following:

- Pick a $\mathcal{A}$-transition dependent on μ-calculus sentence ϕ that performs stack-operation θ whilst moving to control-state q'.
- Check that the top element of the stack belongs to $S_{[\hat{q}]\phi}$ in which case we are indeed in a configuration corresponding to a $\mathcal{A}$-configuration in control-state q at which ϕ holds.
- Transition into control-state q' whilst performing the stack-operation dictated by the $\mathcal{A}^{\star\star}_{LR}$-transition $\boldsymbol{\theta}$.

Then $g : \mathcal{G}(\mathcal{B}) \mathrel{\widehat{\cong}} \mathcal{G}(\mathcal{A})$ with $g(q,s) := g(q,t)$ where $f(\star,s) = t$ (where t assimilates the stack of $f(\star,s)$ with the control-state that must be completely determined by s as $\star$ is fixed).

□

We will use Theorem 2.24 when constructing monotonic automata in a later chapter. These are devices that simulate a CPDA whilst skipping out large chunks of a run, using awareness of μ-calculus properties to determine what would have happened had the run really been performed.

2.4 Higher-Order Recursion Schemes

Recursion schemes can be viewed as rewrite systems employing non-terminals that can be assigned higher-order types. Alternatively we can view them as simply typed lambda terms together with a fixed-point combinator. In actual fact we will translate them to an infinite lambda term that is a regular tree, which is called the *computation tree* of the recursion scheme [60].

The definition that we give in the first instance is in the 'rewriting form' and corresponds to the modern presentation used in [52, 60].

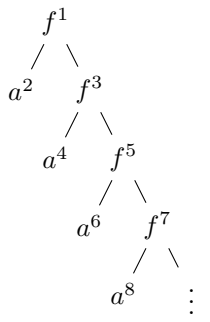


Figure 2.1: A Value Tree

Definition 2.25. The set of types **Types** is generated from a single ground type o together with the function-space constructor $\longrightarrow$. That is:

$$\textbf{Types} ::= o \mid (\textbf{Types} \longrightarrow \textbf{Types})$$

The order of the type T is defined, as usual, by:

$$\textbf{ord}(T) := \begin{cases} 0 & \text{if } T = o \\ \max(\textbf{ord}(T_1) + 1, \textbf{ord}(T_2)) & \text{if } T = (T_1 \longrightarrow T_2) \end{cases}$$

We use the standard convention of $\longrightarrow$ associating to the right, so that $T_1 \longrightarrow T_2 \longrightarrow T_3$ means $(T_1 \longrightarrow (T_2 \longrightarrow T_3))$. We also right $T^m \longrightarrow U$ to mean:

$$\underbrace{T \longrightarrow T \longrightarrow T \longrightarrow \cdots \longrightarrow T}_{m \text{ times}} \longrightarrow U$$

We work over a set of *terminal symbols* which are the elements of a finite ranked alphabet Σ; a finite set of *non-terminals* $\mathcal{N}$ and a set of *variables* $\mathcal{V}$. We will construct a space of typed terms t from these atoms, writing $\mathsf{Ty}(t)$ to denote its type. We begin by assigning types $\mathsf{Ty}(a) := o^{rank(a)} \longrightarrow o$ (so in particular terminals with rank 0 are given type o). We assign elements of $\mathcal{N}$ and $\mathcal{V}$ arbitrary but fixed types.

The set of *applicative terms* over $\Sigma, \mathcal{N}, \mathcal{V}$ is denoted $\boldsymbol{\mathcal{T}}(\Sigma, \mathcal{V}, \mathcal{N})$ and is formed from the atoms using the application rule, forming a term (uv) from terms u and v with type U where $\mathsf{Ty}(u) = V \longrightarrow U$ and $\mathsf{Ty}(v) = V$. We follow convention in associating application to the left so that tuv means $((tu)v)$.

Definition 2.26. An order-n recursion scheme is a tuple $\langle \Sigma, \mathcal{N}, \mathcal{S}, \mathcal{V}, R \rangle$ where no element of $\mathcal{N}$ has type with order greater than n, $\mathcal{S} \in \mathcal{N}$ is a distinguished initial symbol with $\mathsf{Ty}(\mathcal{S}) = o$ and R is a set of rules of the form:

$$F\phi_1\phi_2 \cdots \phi_k \longrightarrow t$$

where the ϕ_i are variables; F is a non-terminal and $t \in \boldsymbol{\mathcal{T}}(\Sigma, \mathcal{N}, \{\phi_1, \phi_2, \ldots, \phi_k\})$. There should be a unique rule for every non-terminal in $\mathcal{N}$.

Let us write $\boldsymbol{\mathcal{R}_n}$ to denote the set of order-n recursion schemes and $\boldsymbol{\mathcal{R}}$ to denote the set of recursion schemes of all orders.

The *value tree* of a recursion scheme is the unique (possibly infinite) ranked and ordered Σ-labelled tree that it generates by unfolding the rewrite rules, starting at the initial symbol, *ad infinitum*. For a recursion scheme G we denote this value tree $[\![G]\!]$.

To be more precise, we follow the definitions of Ong [60], which he establishes to be well-defined. Let us say that a *closed term* is a member of $\boldsymbol{\mathcal{T}}(\Sigma, \mathcal{N}, \emptyset)$. A *redex* in an applicative term t is any subterm of the form $Ft_1 \cdots t_k$ where F is a non-terminal and $\mathsf{Ty}(Ft_1 \cdots t_k) = o$. Thus we have $t = \mathcal{C}[Ft_1 \cdots t_k]$ for some context $\mathcal{C}$. We then say that t *reduces in one step to* $\mathcal{C}[t'[t_1/\phi_1, \ldots, t_m/\phi_k]]$ where

$$F\phi_1\phi_2 \cdots \phi_k \longrightarrow t'$$

is the rewrite rule for F. We use the phrase *reduces to* to mean the transitive reflexive closure of *reduces in one step to*.

For any closed term t, let us write $t^\perp$ to be the result of replacing every maximal subterm of t beginning with a non-terminal with a fresh symbol $\perp$. We may then view $t^\perp$ as being a $(\Sigma \cup \{\perp\})$-labelled tree (by considering its parse tree). Define the flat ordering $\sqsubseteq$ on $(\Sigma \cup \{\perp\})$ by $x \sqsubseteq y$ iff $x = y$ or $x = \perp$. This induces an ordering on $(\Sigma \cup \{\perp\})$-trees T where $T \sqsubseteq T'$ just in case $\mathsf{dom}(T) \sqsubseteq \mathsf{dom}(T')$ and for every $u \in \mathsf{dom}(T)$ we have $T(u) \sqsubseteq T'(u)$.

Example 2.27. Consider the rules:

$$\begin{array}{rcl} \mathcal{S} & \longrightarrow & Ffa \\ F\phi x & \longrightarrow & fx(F\phi a) \end{array}$$

where $rank(a) = 0$, $rank(f) = 2$ and $\mathbf{Ty}(F) = (o \longrightarrow o) \longrightarrow o$ so that $\mathbf{ord}(\mathbf{Ty}(F)) = 2$. Then the value tree generated is as shown in Figure 2.1.

Viewing terms $t^\perp$ as trees, we can then define:

$$[\![G]\!] := \bigsqcup \{\, t^\perp \,:\, \mathcal{S} \text{ reduces to } t \,\}$$

Finally we find it useful to extend applicative terms to include lambda abstractions. We write $\boldsymbol{\mathcal{T}}^{\boldsymbol{\lambda}}(\Sigma, \mathcal{V}, \mathcal{N})$ to denote the set of applicative terms formed from the atoms $\Sigma, \mathcal{N}$ and $\mathcal{V}$ with the use of application, as before, and additionally lambda abstraction: from a variable x with $\mathbf{Ty}(x) = T$ and a term u with $\mathbf{Ty}(u) = U$ we can form $\lambda x.u$ with $\mathbf{Ty}(\lambda x.u)) := (T \longrightarrow U)$. We write $\lambda x_1 x_2.u$ to mean $(\lambda x_1.(\lambda x_2.u))$. The operational semantics of such terms are as one would expect, but in fact we will only appeal to them indirectly via results established in the literature. The actual semantics we present will be in the form of Ong's traversals [60], which owe much to game semantics.

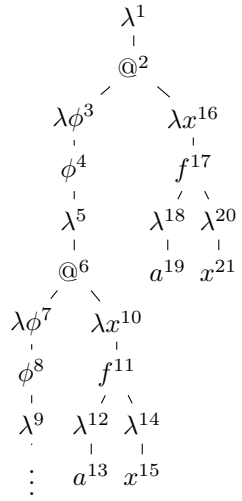


Figure 2.2: A Computation Tree

Computation Trees

The *η-long form* $t^{\boldsymbol{\eta}}$ of a term $t \in \boldsymbol{\mathcal{T}}^{\boldsymbol{\lambda}}(\Sigma, \mathcal{V}, \mathcal{N})$ (and so in particular of a term $t \in \boldsymbol{\mathcal{T}}(\Sigma, \mathcal{V}, \mathcal{N})$) is defined as follows:

Definition 2.28. If $t \equiv \lambda x.u$, then $t^{\boldsymbol{\eta}} := \lambda x.u^{\boldsymbol{\eta}}$. Otherwise $t \equiv uv_1 \cdots v_m$, where u is either a λ-abstraction or an atom ($m = 0$ when t is atomic). Suppose $\mathbf{Ty}(uv_1 \cdots v_m) = A_1 \longrightarrow A_2 \longrightarrow \cdots \longrightarrow A_k \longrightarrow o$. We pick fresh variables $x_1, \ldots, x_k$ such that $\mathbf{Ty}(x_i) = A_i$ for each $1 \leq i \leq k$ and then set $t^{\boldsymbol{\eta}} := \lambda x_1 \cdots x_k.u' v_1{}^{\boldsymbol{\eta}} \cdots v_m{}^{\boldsymbol{\eta}} x_1{}^{\boldsymbol{\eta}} \cdots x_k{}^{\boldsymbol{\eta}}$ where

$$u' = \begin{cases} u & \text{if } u \text{ is atomic} \\ u^{\boldsymbol{\eta}} & \text{if } u \text{ is a } \lambda\text{-abstraction} \end{cases}$$

When $k = 0$ we find it convenient to define $t^{\boldsymbol{\eta}} := \lambda.t$—i.e. we use a vacuous lambda-abstraction.

Remark 2.29. The inductive definition of η-long form is well-defined because both terms and types are well-founded (the latter is necessary for the termination of the definition when taking the η-long form of the added variables).

Remark 2.30. It should be clear from the construction (easy induction) that $\mathsf{Ty}(t) = \mathsf{Ty}(t^{\boldsymbol{\eta}})$. Given that we require the introduction of vacuous λ-abstractions it should also be clear that every η-long form is itself a λ-abstraction.

The *partial computation tree* is partial in the sense that it does not unwind the definitions of non-terminals, and consequently remains finite.

Definition 2.31. Let $s \in \boldsymbol{\mathcal{T}}^{\boldsymbol{\lambda}}(\Sigma, \mathcal{V}, \mathcal{N})$ and set $t : s^{\boldsymbol{\eta}}$. The *partial computation tree* $\boldsymbol{\lambda}^{\star}(t)$ of t is a finite ordered tree that is defined inductively as follows:

- If $t \equiv uv_1 \cdots v_m$ where u is either a λ-abstraction or an element of $\mathcal{N}$ (non-terminal), then $\boldsymbol{\lambda}^{\star}(t)$ has root labelled @, 0th child (numbered 0 for convenience) $\boldsymbol{\lambda}^{\star}(u)$ and ith child (numbered i) $\boldsymbol{\lambda}^{\star}(v_i)$ for each $1 \leq i \leq m$ (making for $m+1$ children altogether).

- If $t \equiv uv_1 \cdots v_m$ where u is an element of Σ or $\mathcal{V}$ (terminal or variable), then $\boldsymbol{\lambda}^{\star}(t)$ has root labelled u and ith child (numbered i) $\boldsymbol{\lambda}^{\star}(v_i)$ for each $1 \leq i \leq m$ (making for m children altogether).

- If t is atomic, then $\boldsymbol{\lambda}^{\star}(t)$ is just a single node labelled t.

- If $t \equiv \lambda\vec{x}.u$ where u is not a λ-abstraction (all λ-abstractions can be written in this form), then $\boldsymbol{\lambda}^{\star}(t)$ consists of a root labelled $\lambda\vec{x}$ and a single child $\boldsymbol{\lambda}^{\star}(u)$.

We then define $\boldsymbol{\lambda}^{\star}(s) := \boldsymbol{\lambda}^{\star}(t)$.

The following is consistent with Ong's original definition [60].

Definition 2.32. Let $G \in \boldsymbol{\mathcal{R}}$ be a recursion scheme with rules of the form $F_i x_1 \cdots x_{k_i} \longrightarrow t_i$ for $1 \leq i \leq m$ where the recursion scheme posesses m non-terminals. The scheme's *computation graph* $\boldsymbol{\lambda}^{\boldsymbol{\mathcal{G}}}(G)$ is the finite graph consisting of the disjoint union of all of the trees $T_i := \boldsymbol{\lambda}^{\star}(\lambda x_1 \cdots x_{k_i}.t_i)$ where each node labelled F_i (for some $1 \leq i \leq m$) is deleted and its parent edge instead attached to the root of T_i.

Definition 2.33. The *computation tree* $\boldsymbol{\lambda}(G)$ of G is the probably infinite regular tree formed by unfolding $\boldsymbol{\lambda}^{\boldsymbol{\mathcal{G}}}(G)$ from the root of T_j (as defined in Definition 2.32) where $F_j : o$ is the initial non-terminal of G. (This T_j will have root labelled λ).

Example 2.34. The following rules yield the computation tree in Figure 2.2.

$$\begin{aligned} \mathcal{S} &\longrightarrow FH \\ F\phi &\longrightarrow \phi\mathcal{S} \\ Hx &\longrightarrow fax \end{aligned}$$

This also generates the value tree in Figure 2.1.

The *order* $\mathbf{ord}(u)$ of node u in computation tree is the order of the type of the subterm rooted at it if it is not a variable node. The order of a variable node is the order of the type of the variable.

Example 2.35. For example, $\mathbf{ord}(u) = 0$ whenever u is an @ node and $\mathbf{ord}(u) = 2$ if u is labelled $\lambda\vec{\phi}$ where $\vec{\phi}$ is a vector of variables where the maximum order of any variable type is 1. Also $\mathbf{ord}(u) = 2$ if u is labelled with ϕ which is a variable with an order-2 type, even if the term rooted at u actually has type with order 0.

We write $\mathbf{b}(u)$ to denote the $\lambda\zeta_1\zeta_2\vec{\cdots}\zeta_m$-node v binding a variable node u and we write $\mathbf{bindpos}(u) := i$ when u is labelled with ζ_i.

Traversals

A traversal is a way of jumping about the computation tree in a manner that represents its linear head reduction—*i.e.* a search for its head variable. The set of traversals can be viewed as the 'uncovered game semantics' of the term. Traversals were defined by Ong [60] and the definition we use here is essentially equivalent. We also add the notion of *direction sequence* associated with a traversal to be able to easily phrase some relevant results from Ong.

For technical reasons, we need to number the children of @ nodes from left to right, beginning at 0, whilst all other nodes have their children numbered left to right beginning at 1. For a node u in the computation tree we write $E_i(u)$ to denote its ith child (using this numbering).

We now define traverals of the computation tree together with the associated *direction sequence* which keep track of the path in the value tree of the recursion scheme to which the traversal corresponds.

Definition 2.36. A *traversal* of $\boldsymbol{\lambda}(G)$ is a sequence σ of nodes therein equipped pointers. The root λ is a traversal with direction sequence ϵ and if $\sigma\ u$ is a traversal with direction sequence τ, then:

- If u is labelled @, $\sigma\ u\overset{0}{\leftarrow}E_0(u)$ is a traversal, where $E_0(u)$ must be a lambda node. The direction sequence remains τ.
- If u is labelled $\lambda\vec{\phi}$, $\sigma\ u\ E_1(u)$ is a traversal (where $E_1(u)$ must be labelled by one of a variable, @ or terminal symbol). The direction sequence remains τ.

- If u is labelled by a variable ϕ, and b is the most recent occurrence of $\mathfrak{b}(u)$ in σ with $\sigma\ u = \sigma_1\ c\ a\ \sigma_2\ b\ \sigma_3\ u$, then $\sigma_1\ c\ a\ \sigma_2\ b\ \sigma_3\ u\ E_j(c)$ where $j := \mathbf{bindpos}(u)$ is a traversal ($E_{\mathbf{bindpos}(u)}(c)$ must be a lambda node that is the root of the argument bound to the variable at u). The direction sequence remains τ.

- If u is labelled by a terminal symbol f, then for any $i \in [1..\mathit{rank}(f)]$ it is the case that $\sigma\ u\ E_i(u)$ is a traversal with direction sequence $\tau\ i$.

The notion of *view* is borrowed from game semantics and is the result of 'skipping over' pointers in a traversal.

Definition 2.37. Let t be a traversal over a computation tree. The *view* $\ulcorner t \urcorner$ of t is a subsequence of t defined inductively as follows:

$$\begin{aligned} \ulcorner \lambda \urcorner &:= \lambda \\ \ulcorner s\ u\ \ldots\ \lambda\overline{\zeta} \urcorner &:= \ulcorner s \urcorner\ u\ \lambda\overline{\zeta} \\ \ulcorner s\ \lambda\overline{\zeta}\ n \urcorner &:= \ulcorner s\ \lambda\overline{\zeta} \urcorner\ n \end{aligned}$$

(To see that this is exhaustive and well-defined, recall that only $\lambda\overline{\zeta}$ nodes are ever the source of a pointer.) As remarked by Ong [60], the view of a traversal is a path in the computation tree.

The bottom line concerning the significance of traversals is given by the following theorem of Ong [60] (which we paraphrase):

Theorem 2.38 (Ong 2006—Path-Traversal Correspondence)**.** *Let G be a recursion scheme. For every node u in $[\![G]\!]$ specified by a sequence of directions τ_u there is precisely one traversal of $\boldsymbol{\lambda}(G)$ with direction sequence τ_u, which when projected onto terminal symbols yields precisely the path from the root of $[\![G]\!]$ to u.*

The paths in the tree value tree and traversals are in bijective correspondence.

Example 2.39. The following is a traversal of the computation tree in Figure 2.2 with direction sequence: 2 1.

$$\lambda^1\ @^2\ \lambda\phi^3\ \phi^4\ \lambda x^{16}\ f^{17}\ \lambda^{20}\ x^{21}\ \lambda^5\ @^6\ \lambda\phi^7\ \phi^8\ \lambda x^{10}\ f^{11}\ \lambda^{12}\ a^{13}$$

The Safety Constraint

We now provide the formal definitions related to safety.

Definition 2.40. Let us say that an occurrence of a subterm u of t is a *local-head position* if there exists a v such that the occurrence u occurs in a subterm of the form (uv).

Definition 2.41. An occurrence of a variable x in a term $t \in \boldsymbol{\mathcal{T}}(\Sigma, \mathcal{V}, \mathcal{N})$ is deemed *unsafe* if it occurs in a subterm u of t with $\mathbf{ord}(x) < \mathbf{ord}(u)$ such that u is *not* in a local-head position. We say that an occurrence of a variable is *safe* if it is not unsafe.

Definition 2.42. A term $t \in \boldsymbol{\mathcal{T}}(\Sigma, \mathcal{V}, \mathcal{N})$ is deemed *safe* if it contains no unsafe occurrence of a variable. We say that a recursion scheme is *safe* if the right-hand-side of each rule is a safe term. We say that the recursion scheme is *homogeneously safe* if it is also homogeneous.

Knapik *et al.* [52] defined safety to be what we call *homogeneous safety*. Therefore the 'traditional' notion of safety differs only from our own in that the former imposes type homogeneity as an additional constraint on the recursion scheme.

The following definitions are due to Blum [11, 10, 9] and concern his 'game semantic' characterisation of safety, here presented in the context of traversals.

Definition 2.43. Consider an occurrence of a variable x in a computation tree $\boldsymbol{\lambda}(G)$ with $G \in \boldsymbol{\mathcal{R}}$ or $\boldsymbol{\lambda}^\star(t)$ for $t \in \boldsymbol{\mathcal{T}}^{\boldsymbol{\lambda}}(\Sigma, \mathcal{V}, \mathcal{N})$. We say that this occurrence is *incrementally bound* if its binder $\lambda \cdots x \cdots$ is the first λ-node in the path from x to the root of the computation tree with order strictly greater than x.

Blum's characterisation [11]:

Lemma 2.44 (Blum)**.** *If $t \in \boldsymbol{\mathcal{T}}(\Sigma, \mathcal{V}, \mathcal{N})$ is safe, then every occurrence of a variable x in $\boldsymbol{\lambda}^\star(t)$ is incrementally bound.*

Lemma 2.45. *Given a* safe *recursion scheme $G \in \boldsymbol{\mathcal{R}}^{\boldsymbol{S}}_{\boldsymbol{n}}$ every occurrence of a variable in $\boldsymbol{\lambda}(G)$ is incrementally bound.*

We are now ready to consider a finer analysis of safety, which is the subject of the next chapter.

Chapter 3

Fine Grained Safety and Homogeneity

As we have seen, safety is a syntactic restriction that can be placed on recursion schemes. It is *prima facie* necessary to restrict schemes in this manner in order to coincide with the trees generated by n-PDA [52]. Some limited progress has been made on determining the effect of safety on expressivity. Aehlig *et al.* [1, 47] showed that every *word*-language generated by an unsafe scheme can also be generated by a *non-deterministic* safe scheme. More recently Parys [62] has shown that no such results can exist for trees of order-2; there exists a tree generated by an order-2 unsafe scheme that cannot be generated by any order-2 safe scheme.

There remains plenty of scope for further research on this *extensional* question. What happens at order-3 and above? Even the weakest 'non-uniform' version of the Safety Conjecture remains at large—is there some tree generated by some unsafe scheme that cannot be generated by an order-n safe scheme for *any* n? In unpublished work the author has extended the result of Aehlig *et al.* to show that outer-most links of a CPDA can always be eliminated at the cost of non-determinism, but this does not really contribute to the root of the question, which unfortunately we are unable to pursue here.

The question addressed in the present chapter, however, is *intensional* in nature. The significance of safety from an operational standpoint is that it allows for β-reduction without incurring the need for α-conversion of variables since variable capture can never occur. William Blum has extensively studied the significance of safety from the point of view of game semantics [11, 10] and in particular in terms of traversals of computation trees. Given that the translation from (unsafe) recursion schemes to CPDA [40] yields an intimate relationship between the behaviour of the automaton and the computation of traversals, it is reasonable to ask how safety affects the CPDA constructed using this method. Given that safe recursion schemes can be translated to non-

collapsible PDA [52] it would be very nice if the traversal computing n-CPDA is in fact (morally) an n-PDA.

Blum provided the bulk of a proof for this hypothesis based on his study of traversals for the safe lambda terms. There was one gap in this proof that was filled by the author, which was in turn refined by Blum [9, 18]. This shows that the traversal computing automaton indeed does *not* require collapse when the recursion scheme is safe. Since the result depends *only* on the game semantic consequences of safety, it also allows the homogeneity requirement to be dropped for equi-expressivity with n-PDA.

In this chapter we generalise Blum's argument further to explain the exact role of CPDA-links in terms of safe occurrences of variables. We aim to characterise the expressivity of n_S-CPDA using a related notion of S'-safety which requires only order-k variables for $k \notin S'$ to occur safely in the recursion schemes. Thus $\emptyset$-safety will coincide with the original notion.

This is of interest as later chapters will consider CPDA with restricted use of links and it is nice to understand how this can be seen in terms of recursion schemes. It also enables us to say something about another question raised by the result—if homogeneity can be dropped from the definition of safety then what effect, if any, occurs when one limits recursion schemes to homogeneous types? We provide a notion of *'homogeneous CPDA'* that precisely captures the expressivity of homogeneous recursion schemes and which can in turn be encoded as a homogeneous recursion scheme. Thus 'homogeneity is not a constraint on expressivity'—something one gets for free in the presence of λ-abstractions, but not so trivially for the applicative presentation traditionally used for recursion schemes.

3.1 Relative Safety

Relative safety stipulates that only variables with types of particular orders need occur safely. Note that the orders of variables are incremented by 1 in the set S, because the minimum order of the variables' binders will be more significant than the order of the variables themselves.

Definition 3.1. Let $S \subseteq \mathbb{N}$. A term $t \in \boldsymbol{\mathcal{T}}(\Sigma, \mathcal{V}, \mathcal{N})$ is deemed *safe modulo* S (or just S-*safe*) if every occurrence of a variable x in t with $(\mathbf{ord}(x) + 1) \notin S$ is safe. A recursion scheme G is said to have *order-*n_S if $G \in \boldsymbol{\mathcal{RS}_n}$ and the right-hand-side of every rule is an S-safe term. We write $\boldsymbol{\mathcal{RS}_{n_S}}$ for the set of all such recursion schemes.

Remark 3.2. Note that by definition $\boldsymbol{\mathcal{RS}_n} = \boldsymbol{\mathcal{RS}_{n_{[1..n]}}}$ and $\boldsymbol{\mathcal{RS}_n^S} = \boldsymbol{\mathcal{RS}_{n_\emptyset}}$.

It is easy to generalise Blum's Lemma 2.45 to these *relatively safe* recursion schemes.

Lemma 3.3. *Let $t \in \boldsymbol{\mathcal{T}}(\Sigma, \mathcal{V}, \mathcal{N})$. Then every bound occurrence of a variable x in $\boldsymbol{\lambda^\star}(t)$ is incrementally bound.*

Proof. This is a straightforward induction on the structure of t with respect to the inductive definition of $t^{\boldsymbol{\eta}}$. In essence the only variables that are bound in $t^{\boldsymbol{\eta}}$ are those that are freshly introduced (i.e. no variable occurring in t is bound in $t^{\boldsymbol{\eta}}$). The manner in which corresponding abstractions are introduced ensures incremental binding. □

Definition 3.4. Given $G \in \boldsymbol{\mathcal{R}\mathcal{S}}$ let us call the variables occurring in G the *scheme-variables* and the variables added when taking the η-long form the η-variables. Thus all variables in $\boldsymbol{\lambda}(G)$ are either scheme-variables or η-variables and w.l.o.g. we may assume the two classes are disjoint.

The following states the incrementally-bound result on a *per variable* basis (rather than a statement depending on all variables being safe occurrences).

Lemma 3.5. *Every safe occurrence of a variable x in a recursion scheme $G \in \boldsymbol{\mathcal{R}\mathcal{S}}$ is incrementally bound in $\boldsymbol{\lambda}(G)$.*

Proof. By Lemma 3.3 we only need to consider scheme-variables. Consider some scheme-variable x that occurs safely in G. Suppose for contradiction that some occurrence of x in $\boldsymbol{\lambda}(G)$ *fails* to be incrementally bound.

By the definition of a computation tree, x must be bound at the root of the partial-computation tree $\boldsymbol{\lambda^\star}(\lambda x_1 \cdots x_k.t)$ for some rule $Fx_1 \cdots x_k \longrightarrow t$ in G. This node r has label $\lambda x_1 \cdots x_k$.

Since we are assuming x is not incrementally bound there must exist a λ-node u lying on the path between x and r such that $\mathbf{ord}(r) \geq \mathbf{ord}(u) > \mathbf{ord}(x)$ that does *not* bind x. Since this occurs within $\boldsymbol{\lambda^\star}(\lambda x_1 \cdots x_k.t)$ it must be the case that u abstracts only η-variables. The portion of $\boldsymbol{\lambda^\star}(\lambda x_1 \cdots x_k.t)$ rooted at u must thus correspond to a subterm $\lambda \vec{y}.v\vec{y}$ of $t^{\boldsymbol{\eta}}$ where $\vec{y}$ consists entirely of η-variables. This implies that v cannot be in a local-head position in t and that $\mathbf{ord}(v) = \mathbf{ord}(u)$. But since $\mathbf{ord}(x) < \mathbf{ord}(u)$ this would then imply that x is an unsafe occurrence, which is a contradiction. □

Lemma 3.6. *Given a recursion scheme $G \in \boldsymbol{\mathcal{R}\mathcal{S}_{n_S}}$ every occurrence of a variable x in $\boldsymbol{\lambda}(G)$ with $(\mathbf{ord}(x) + 1) \notin S$ is incrementally bound.*

Proof. This is an immediate consequence of the definition of $\boldsymbol{\mathcal{R}\mathcal{S}_{n_S}}$ and Lemma 3.5. □

3.2 An alternative Traversal Computing CPDA

Hague *et al.* specify a tree-generating n-CPDA $\mathbf{CPDA}(G)$ for every recursion scheme G for which $[\![G]\!] = [\![\mathbf{CPDA}(G)]\!]$ [40]. This operates by 'computing

traversals' of $\boldsymbol{\lambda}(G)$. The stack alphabet is taken to be nodes of $\boldsymbol{\lambda}^{\mathcal{G}}(G)$ and since nodes of $\boldsymbol{\lambda}(G)$ can be viewed as paths in $\boldsymbol{\lambda}^{\mathcal{G}}(G)$ (being its unfolding) the order-1 stacks contained within the n-stack can represent nodes of $\boldsymbol{\lambda}(G)$. Computing a traversal means that $\mathbf{CPDA}(G)$'s top_2 stack represents the current point in the traversal and the rest of its stacks contains a sufficient amount of history to proceed with computing it. Links emanate from λ-nodes and point to a context from which the argument of the corresponding λ-subterm can be recovered.

We define an n-CPDA $\mathbf{CPDA}^{+}(G)$ which differs from $\mathbf{CPDA}(G)$ in the way in which it handles links. Since a variable x can be bound by a λ node which also binds variables with order greater than $\mathbf{ord}(x)$, the original $\mathbf{CPDA}(G)$ is unable to offer the fine-grained control that we require; it attaches just a single link of a single order to each λ-node. In changing this it is easiest to present our construction as a *multi-link* CPDA for which a single atomic element may emanate links of all orders. Please see Figure 3.1 for the transition rules, noting that the *span* $\mathbf{span}(x)$ of an occurrence of a variable x is equal to $|p|$ where p is the path in the computation tree/graph from $\mathfrak{b}(x)$ to x.

We now modify the definition of computing a traversal from [40] to be appropriate for $\mathbf{CPDA}^{+}(G)$.

First let us define an *approximation* $\underline{s}$ of a stack s inspired by the corresponding definition from [40].

Definition 3.7. Let s be an n-stack reachable by $\mathbf{CPDA}^{+}(G)$. We form the sequence of stack symbols $\underline{s}$ from s in the following manner, assuming that s is represented as a pointer-word with well-bracketed [and] symbols denoting the bottom and top of a stack (with pointers targeted at the] symbols).

The stack approximation we use is going to be a set of subsequences of the stack (as opposed to a single subsequence as in [40]—however it will turn out that for the stacks of $\mathbf{CPDA}^{+}(G)$ this set is always a singleton. $\underline{s}$ is a *set* of subsequences of s that can be formed by scanning s from right to left (top to bottom) as follows:

- When a] is encountered the scan continues without recording any symbol.
- Any stack symbol read is recorded.
- If a $\lambda\vec{x}$ stack symbol is reached with $\vec{x}$ consisting of η-variables, then we record $\lambda\vec{x}$ and then skip to the target of any k-link for $k \geq n - \mathbf{ord}(\lambda\vec{x}) + 1$.

Given a traversal t the approximant $\hat{t}$ is formed by deleting all segments u of t of the form $v\overset{\frown}{u}\lambda$.

We can now define what it means for a stack s to *compute* a traversal t in a manner analogous to [40]. In what follows we disregard terminal symbols for technical convenience. Branching is the only substantial effect that terminal symbols impose and the rules of $\mathbf{CPDA}^{+}(G)$ ensure that provided we can show

Let u be the top_1 stack symbol. As with $\mathbf{CPDA}(G)$ if u is not a variable then the automaton usually just performs $push_1^{u'}$ where u' is an appropriate child of u. A difference in behaviour does occur at an @-node where we first perform some higher-order pushes. To be precise:

- $(\boldsymbol{A_1^+})$ If the label is @ and $\mathbf{ord}(E_0(u)) \geq 1$, then

  $$\delta(u) := push_n; push_{n-1}; \cdots push_{n-\min(\mathbf{ord}(E_0(u)),n-1)+1}; push_1^{E_0(u)}$$

  recalling that $E_0(u)$ must be a λ-node (abstracting scheme-variables).
- $(\boldsymbol{A_0^+})$ If the label is @ and $\mathbf{ord}(E_0(u)) = 0$ (i.e. $E_0(u)$ is labelled λ), then $\delta(u) := push_1^{E_0(u)}$.
- $(\boldsymbol{S^+})$ If the label is a terminal symbol f, then the automaton branches in direction i (where $1 \leq i \leq \mathbf{ar}(f)$) with operation $\delta(u) := push_1^{E_i(u)}$.
- $(\boldsymbol{L^+})$ If the label is a lambda, then $\delta(u) := push_1^{E_1(u)}$.

Suppose now that u is labelled with a variable x and let $b := \mathbf{bindpos}(u)$, then:

- $(\boldsymbol{V_1^+})$ If $\mathbf{ord}(x) \geq 1$, then: $\delta(u) := push_n; push_{n-1}; \cdots push_{n-\mathbf{ord}(x)+1}; pop_1^{\mathbf{span}(u)}; collapse_j; push_1^{E_b(top_1)}$ where $j := n - \mathbf{ord}(x)$.
- $(\boldsymbol{V_0^+})$ If $\mathbf{ord}(x) = 0$, then: $\delta(u) := pop_1^{\mathbf{span}(u)}; collapse_n; push_1^{E_b(top_1)}$.

where top_1 denotes the top element of the stack a that particular point in the operation (rather than at the beginning of the operation).

Figure 3.1: Rules for the n-CPDA $\mathbf{CPDA}^+(G)$ where $\mathbf{ord}(G) = n$.

correctness for 'computing traversals' without terminal symbols we can extend this trivially to take into account terminal symbols.

Definition 3.8. Let $G \in \boldsymbol{\mathcal{RS}_n}$, let s be a reachable stack of $\mathbf{CPDA}^+(G)$ and let t be a traversal over $\boldsymbol{\lambda}(G)$. We say that s *computes* t just in case the following conditions hold:

- $top_2(s) = \ulcorner t \urcorner$
- $\underline{s}$ is a singleton set whose element we will also denote $\underline{s}$ thereby overloading notation. Moreover $\underline{s} = \hat{t}$.
- Suppose that $top_2(s) = [v_1, \ldots, v_n]$. Let $v_1', \ldots, v_n'$ be the respective occurrences of $v_1, \ldots, v_n$ in t contributing to $\vec{t}$. Then for every v_i such that v_i is a λ-abstraction $\lambda\vec{x}$ we require that $collapse_j(s_{\leq v_i})$ computes $t_{<v_i'}$

for every $n - \mathbf{ord}(\lambda\vec{x}) + 1 \leq j \leq n$. Additionally $s_{\leq v_i}$ should compute $t_{\leq v'_i}$ for *every* v_i.

Lemma 3.9. *Let $G \in \mathcal{RS}_n$ be a recursion scheme featuring no terminal symbols. Let t be a traversal over $\boldsymbol{\lambda}(G)$ and suppose that we have a reachable stack s of $\mathbf{CPDA}^+(G)$ computing t. Then $s' := push_k(s)$ also computes t for any $1 \leq k \leq n$.*

Proof. The truth of this is implied by the following facts: There are no higher-order pop operations involved in the definition of computing a traversal; a $push_k$ preserves the targets of l-links for $l \geq k$; following an m-link for $m < l$ does not cause one to leave the copy of the top_k stack of s that is the top_k stack of s'. □

Lemma 3.10. *Let $G \in \mathcal{RS}_n$ be a recursion scheme featuring no terminal symbols. Let t be a traversal over $\boldsymbol{\lambda}(G)$ and suppose that we have a reachable stack s of $\mathbf{CPDA}^+(G)$ computing t. Suppose further that $t' := t\ u'$ is the unique traversal extending t with the symbol u (which may or may not source a pointer to a target in t). Then the unique operation that $\mathbf{CPDA}^+(G)$ performs from s (from Figure 3.1) results in a stack s' computing t'.*

Proof. Let u be the final node in t. We consider each case of u in turn.

Case when u bears an @ symbol: Then it must be the case that $u' = E_0(u)$ and that the automaton performs either $(\boldsymbol{A}_\mathbf{0}^+)$ or $(\boldsymbol{A}_\mathbf{1}^+)$. The former is straightforward (in this case $E_0(u)$ is just labelled λ), so just consider the latter. By Lemma 3.9 the stack after performing $push_n; \cdots push_{\min(\mathbf{ord}(E_0(u)), n-1))}$ will continue to compute the original traversal t. Note that $E_0(u)$ must have the form $\lambda\vec{\phi}$ (where $\vec{\phi}$ consists of scheme variables). When the operation is finished with $push_1^{E_0(u)}$, the preceding pushes ensure that the j-links therefrom point to a target computing t for each $n - \mathbf{ord}(\lambda\vec{\phi}) + 1 \leq j \leq n$.

Case when u bears a $\lambda\vec{x}$ symbol: Then $u' = E_1(u)$ and u' must bear either an @ or a variable symbol. Again in each case s' computing t' is straightforward (@ and variables do not emit any justification pointers and so do not affect the view).

Case when u bears a variable x with $\mathbf{ord}(x) \geq 1$: Suppose that $\mathbf{bindpos}(x) = i$. Then $u' = \lambda\vec{y}$ where $\lambda\vec{y} = E_i(v)$ is an abstraction of η-variables where v is the predecessor of $\mathfrak{b}(\lambda\vec{y})$ in t. There will also be an i-justification pointer from $\lambda\vec{y}$ to v. In particular this means that $\ulcorner t' \urcorner = \ulcorner t_{\leq v} \urcorner\ E_i(v)$. By Lemma 3.9 and the assumption that s computes t we know that $push_n; push_{n-1}; \ldots; push_{n-\mathbf{ord}(x)+1}(s)$ also computes t. Since the binder of an occurrence of a variable in a traversal always occurs in the view of the traversal up to that occurrence of a variable, it follows from $top_2(s) = \ulcorner t \urcorner$ that the binder of x occurs in

$$top_2(push_n; push_{n-1}; \ldots; push_{n-\mathbf{ord}(x)+1}(s))$$

Moreover it means that $pop_1^{\mathsf{span}(x)}$ (the first part of the operation performed by (V^+)) results in $\mathfrak{b}(x)$ coming to the top of the stack.

Since $\mathsf{ord}(x) < \mathsf{ord}(\mathfrak{b}(x))$ we must have $n - \mathsf{ord}(x) \geq n - \mathsf{ord}(\mathfrak{b}(x)) + 1$ and so a $collapse_{n-\mathsf{ord}(x)}$ operation must also result in a stack computing $t_{<\mathfrak{b}(x)} = t_{\leq v}$.

The automaton then performs $push_1^{u'}$ to produce the final stack s'. Since the traversal places an i-pointer from u' to v and since the stack prior to $push_1^{u'}$ computes $t_{\leq v}$ it must be the case that $top_2(s) = \ulcorner t' \urcorner$. For the remaining three conditions observe that the only links relevant to the definition of $\underline{s}$ and the final conditions are those with order k for $n - \mathsf{ord}(u') + 1 \leq k \leq n$. Since $\mathsf{ord}(u') = \mathsf{ord}(x)$, all such links must point to a (copy of) a stack in existence before the original $push_n; push_{n-1}; \cdots push_{n-\mathsf{ord}(x)+1}$ and so all these stacks to which a k-pointer from u' points must compute t.

Case when u bears a variable x with $\mathsf{ord}(x) = 0$: Using the same notation to the previous case we have u' bearing the label λ (i.e. $\vec{y}$ is empty). An argument similar to the previous case can be used, except at the end we invoke the fact that *no* pointers from u' are relevant and that $\hat{t'}$ skips the component of t lying between v and u'. □

Theorem 3.11. *Let $G \in \mathcal{RS}_n$. Then $[\![G]\!] = [\![\mathsf{CPDA}^+(G)]\!]$.*

Proof. For each finite initial subtree of the tree generated we can use induction together with Lemma 3.10. We then appeal to determinism to get the result for the full tree. □

3.3 Relative Safety and CPDA

We extend Blum's proof with the author's notion of stack-safety [9, 18] to the notion of relative safety introduced in the previous section. Stack-safety is a property reflecting the extent to which a *collapse* on a k-link can be replaced with a pop_k operation.

The notion of stack decomposition is due to Blum [9]:

Definition 3.12. Let $G \in \mathcal{RS}_n$ and let s be a stack for $\mathsf{CPDA}^+(G)$. Define *the l-stack decomposition of s* $\eth_l s$ to be the subsequence of $top_2(s)$ $\langle \lambda\vec{x_1}, \cdots, \lambda\vec{x_k} \rangle$ where $\lambda\vec{x_k}$ is the top-most λ-node in the stack satisfying $\mathsf{ord}(\lambda\vec{x_k}) > l$ and for each $1 \leq i < k$ $\lambda\vec{x_i}$ is the top-most lambda-node in the stack satisfying $\mathsf{ord}(\lambda\vec{x_i}) > \mathsf{ord}(\lambda\vec{x_{i+1}})$ and $\mathsf{ord}(\lambda\vec{x_{i+1}}) \in S$. The *stack-decomposition* $\eth s$ is just defined to be $\eth_0 s$.

The notion of stack safety was fully defined by the author [18] based on an initial attempt by Blum:

Definition 3.13. Let $G \in \mathcal{RS}_n$ and let s be a stack for $\mathbf{CPDA}^+(G)$. Where $\eth_{l_S} s = \langle \lambda\vec{x_1}, \ldots, \lambda\vec{x_k} \rangle$ we say that s is *l-safe* if:

- $pop_{n-j+1}(s)$ is $(j-1)$-safe for every $1 \leq j \leq n-1$
- $collapse_{n-j+1}(s_{\leq \lambda\vec{y}})$ is $(j-1)$-safe for every λ-abstraction $\lambda\vec{y}$ occurring in $top_2(s)$ and every $1 \leq j \leq \mathbf{ord}(\lambda\vec{y})$.
- $\boldsymbol{l}_{r_{n-j+1}}(\lambda\vec{x_i}) = 1$ for each $\mathbf{ord}(\lambda\vec{x_{i+1}}) < j \leq \mathbf{ord}(\lambda\vec{x_i})$ when $1 \leq i < k$.
- $\boldsymbol{l}_{r_{n-j+1}}(\lambda\vec{x_k}) = 1$ for each $l < j \leq \mathbf{ord}(\lambda\vec{x_k})$.

Note that the empty stack is trivially l-safe (as the 'for every' quantifiers are vacuous) and so the inductive content of the definition is indeed well-founded.

We say that a stack is *safe* if it is 0-safe.

Lemma 3.14. *Let $S \subseteq \mathbb{N}$ and let s be a stack for $\mathbf{CPDA}^+(G)$ where $G \in \mathcal{RS}_n$. If s is l-safe, then $push_{n-k+1}(s)$ is k-safe for every $\max(l,1) \leq k \leq n-1$.*

Proof. Let $s' := push_{n-k+1}(s)$. First recall that the following equality holds:

$$\boldsymbol{l}_{r_{n-j+1}}(s'_{\leq u'}) = \begin{cases} \boldsymbol{l}_{r_{n-j+1}}(s_{\leq u}) + 1 & \text{if } j = k \\ \boldsymbol{l}_{r_{n-j+1}}(s_{\leq u}) & \text{otherwise} \end{cases}$$

where u and u' are an arbitrary pair of corresponding elements in $top_{n-k+1}(s)$ and $top_{n-k+1}(s')$ respectively. Since $k \geq l$ this implies that the final two conditions of k-safety must hold.

Recall further that for $j \leq k$ we have $collapse_{n-j+1}(s_{\leq u}) = collapse_{n-j+1}(s'_{\leq u'})$. Also $pop_{n-j+1}(s') = pop_{n-j+1}(s)$ for $j < k$ and $pop_{n-k+1}(s') = s$.

The latter fact together with the second-condition of l-safety ensures that for $j \leq k$ $collapse_{n-j+1}(s'_{\lambda\vec{y}})$ is $(j-1)$-safe for every λ-abstraction $\lambda\vec{y}$ in $top_{n-k+1}(s')$. Together with the first-condition of l-safety this also gives us the $(j-1)$-safety of $pop_{n-j+1}(s')$ for $j \leq k$.

Suppose for contradiction that s' is not k-safe. The only possibility for this not ruled out above is that iteratively performing $collapse_{n-j+1}$ operations on λ-abstractions with order $j > k$ in the top_2-stack and pop_{n-j+1} operations for a sequence $j_1, \ldots, j_m$ of values of k must lead to a stack that fails to be $(j_m - 1)_S$-safe. Consider a maximal such sequence leading to a smallest such stack (i.e. such that extending the sequence of collapses in any way cannot lead to another violation of the first condition). This assumption means that this smallest such stack must satisfy the first condition for $(j_m - 1)$-safety. But since this stack must lie within $top_{n-k+1}(s')$ and since $j_m - 1 \geq k$, all of the nodes in the $(j_m - 1)$-stack-decomposition must have order $> k$ and so in particular $\neq k$. The final three conditions must thus be satisfied. Thus this stack cannot violate j_m-safety after all. □

Lemma 3.15. *Let s be an l-safe stack. For any node $\lambda\vec{x}$ such that $\mathbf{ord}(\lambda\vec{x}) \geq l$ it is the case that $push_1^{\lambda\vec{x}}(s)$ is safe.*

Proof. The first condition of l-safety ensures both the first and second conditions of 0-safety are met. Likewise the newly pushed abstraction must be the λx_k for the new stack referred to in the final condition, and so the final condition must be met (since all links have just been freshly created). The third and fourth conditions of 0_S-safety must be met by the fact that the original stack was l_S-safe. □

Lemmas 3.14 and 3.15 ensure that all reachable stacks of a traversal computing CPDA $\mathbf{CPDA}^+(G)$ remain safe.

Lemma 3.16. *Let $G \in \mathcal{RS}_n$. Every stack s reachable by $\mathbf{CPDA}^+(G)$ is safe.*

Proof. Argue by induction on the length of the run of $\mathbf{CPDA}^+(G)$ witnessing the reachability of s. Observe first that a $push_1^a$ does not affect safety whenever a is not a λ-abstraction. Thus the only rules we need consider are $(\boldsymbol{A_0^+})$, $(\boldsymbol{A_1^+})$, $(\boldsymbol{V_0^+})$ and $(\boldsymbol{V_1^+})$.

Consider first the rule $(\boldsymbol{A_0^+})$. In this case the symbol pushed onto the stack will just be a vacuous λ-abstraction λ with order 0. This will thus not feature in the 0-stack decomposition and the j is vacuous for this λ in the second condition of stack-safety. Thus $(\boldsymbol{A_0^+})$ preserves stack-safety.

For $(\boldsymbol{A_1^+})$ we first (repeatedly) apply Lemma 3.14 to see that after the sequence of higher-order push operations we have a $\min(\mathbf{ord}(E_0(u)), n-1)$-safe stack. Since $\mathbf{ord}(E_0(u)) \geq \min(\mathbf{ord}(E_0(u)), n-1)$ we may then apply Lemma 3.15 to see that safety is preserved over all.

It is straightforward to see that $(\boldsymbol{V_0^+})$ preserves stack-safety, since the stack following the *collapse* will be 0-safe by the second condition of safety and as with $(\boldsymbol{A_0})$ the final pushing of a λ-symbol will retain this safety.

Finally consider $(\boldsymbol{V_1^+})$ when calling a variable x. By Lemma 3.14 the initial sequence of higher-order pushes will result in an $\mathbf{ord}(x)$-safe stack. The collapse operation on the binder of x will be performed on an $n - \mathbf{ord}(x) = (n - (\mathbf{ord}(x) + 1) - 1)$-link. Again by the second condition of $\mathbf{ord}(x)$-safety this must yield a $(\mathbf{ord}(x) + 1) - 1 = \mathbf{ord}(x)$-safe stack. Since the $\lambda\vec{y}$ abstraction that will be finally pushed onto the stack must satisfy $\mathbf{ord}(\lambda\vec{y}) = \mathbf{ord}(x)$ we may invoke Lemma 3.15 to see that safety is preserved by the operation as a whole. □

In the light of this we may conclude that $\mathbf{CPDA}^+(G)$ *only* needs to use *collapse* when calling an occurrence of a scheme-variable that is not incrementally bound. In particular, *collapse* does not need to be used when calling any safe occurrence of a variable. In such cases, $collapse_k$ can be avoided by replacing it with the corresponding pop_k operation. This is captured by the

fact that the CPDA $\mathsf{CPDA}^{\mathsf{S}}(G)$ described in Figure 3.2 correctly generates the same tree as G.

Theorem 3.17. *Let $G \in \mathcal{RS}_n$. Then $[\![G]\!] = [\![\mathsf{CPDA}^{\mathsf{S}}(G)]\!]$.*

Proof. First observe that any incrementally bound variable x will only be reached by $\mathsf{CPDA}^{+}(G)$ when its binder $\lambda\vec{x}$ belongs to its stack decomposition. By Lemma 3.16 we must then have $\boldsymbol{lr}_{n-\mathsf{ord}(x)+1}(\lambda\vec{x}) = 1$ and so $collapse_{n-\mathsf{ord}(x)+1}$ on $\lambda\vec{x}$ would indeed amount to $pop_{n-\mathsf{ord}(x)+1}$.

Since all η-variables are incrementally bound it follows that $\mathsf{CPDA}^{+}(G)$ and $\mathsf{CPDA}^{\mathsf{S}}(G)$ will have precisely corresponding runs (exactly the same configuration at each point). Thus the correcness of $\mathsf{CPDA}^{\mathsf{S}}(G)$ follows from the correctness of $\mathsf{CPDA}^{+}(G)$ (Theorem 3.11). □

Corollary 3.18. *Given $S \subseteq \mathbb{N}$ and $n \in \mathbb{N}$ define the set $n - S$ to be $\{n - s : s \in S\}$. The set of trees generated by generators in $\mathcal{RS}_{n_S}$ and the set produced by generators in $\boldsymbol{CPDA}_{n_{n-S}}$ are precisely the same.*

Proof. The direction going from recursion schemes to CPDA follows from Lemma 3.6 and Theorem 3.17. The direction from CPDA to recursion schemes can be achieved by exactly the same translation as used by Hague *et al.* [40], omitting the $(n - s)$-unsafe variable encoding s-links. □

3.4 Homogeneity

For this section we return to the traditional definition of CPDA for which the order of a link is specified at the time when an element is pushed onto the stack.

The traversal computing CPDA such as $\mathsf{CPDA}(G)$ does not make use of pop_k for $k \geq 2$. Moreover whenever a k-link (for $k \geq 2$) is created, it is attached to a stack freshly created by a $push_k$ operation.

Definition 3.19. A *fresh n-CPDA* is one that never performs a pop_k operation for $k \geq 2$ and will only perform a $push_1^{a,k}$ operation for $2 \leq k < n$ when the preceding sequence of operations takes the form: $push_k; (pop_1 + collapse)^*$ where each *collapse* is on a j-link for some $j < k$ (possibly different j in each case) or else is just a *collapse* on an n-link. (Note that we make no restriction on when $push_1^{a,1}$ and $push_1^{a,n}$ can be performed.)

Lemma 3.20. *For every recursion scheme $G \in \mathcal{RS}_n$ the equivalent CPDA $\mathsf{CPDA}(G)$ is fresh.*

Proof. A simple inspection of the transition function shows this to be the case. □

Let u be the top_1 stack symbol.

- $(\boldsymbol{A_1^S})$ If the label is @ and $\mathbf{ord}(E_0(u)) \geq 1$, then

  $$\delta(u) := push_n; push_{n-1}; \cdots push_{n-\min(\mathbf{ord}(E_0(u)),n-1)+1}; push_1^{E_0(u)}$$

  recalling that $E_0(u)$ must be a λ-node (abstracting scheme-variables).

- $(\boldsymbol{A_0^S})$ If the label is @ and $\mathbf{ord}(E_0(u)) = 0$ (i.e. $E_0(u)$ is labelled λ), then $\delta(u) := push_1^{E_0(u)}$.

- $(\boldsymbol{S^S})$ If the label is a terminal symbol f, then the automaton branches in direction i (where $1 \leq i \leq \mathbf{ar}(f)$) with operation $\delta(u) := push_1^{E_i(u)}$.

- $(\boldsymbol{L^S})$ If the label is a lambda, then $\delta(u) := push_1^{E_1(u)}$.

Suppose now that u is labelled with a *scheme*-variable ϕ and that the occurrence of ϕ in u is *not incrementally bound*. Let $b := \mathbf{bindpos}(u)$, then:

- $(\boldsymbol{V_1^S})$ If $\mathbf{ord}(x) \geq 1$, then: $\delta(u) := push_n; push_{n-1}; \cdots push_{n-\mathbf{ord}(x)+1}; pop_1^{\mathbf{span}(u)}; collapse_j; push_1^{E_b(top_1)}$ where $j := n - \mathbf{ord}(x)$.

- $(\boldsymbol{V_0^S})$ If $\mathbf{ord}(x) = 0$, then: $\delta(u) := pop_1^{\mathbf{span}(u)}; collapse_n; push_1^{E_b(top_1)}$.

where top_1 denotes the top element of the stack a that particular point in the operation (rather than at the beginning of the operation).
Suppose now that u is labelled with a variable x not covered by the cases above. Then:

- $(\boldsymbol{V_{1_s}^S})$ If $\mathbf{ord}(x) \geq 1$, then: $\delta(u) := push_n; push_{n-1}; \cdots push_{n-\mathbf{ord}(x)+1}; pop_1^{\mathbf{span}(u)}; pop_j; push_1^{E_b(top_1)}$ where $j := n - \mathbf{ord}(x)$.

- $(\boldsymbol{V_{0_s}^S})$ If $\mathbf{ord}(x) = 0$, then: $\delta(u) := pop_1^{\mathbf{span}(u)}; pop_n; push_1^{E_b(top_1)}$.

Figure 3.2: Rules for the n-CPDA $\mathbf{CPDA^S}(G)$ where $\mathbf{ord}(G) = n$.

We have another technical definition describing a relationship between CPDA that are able to simulate each other when pop_k operations are disregarded.

Definition 3.21. Let $\mathcal{A}$ and $\mathcal{A}'$ be n-CPDA with each operation θ of $\mathcal{A}$ that is *not* a pop_k operation for $k \geq 2$ bearing an associated (compound) operation $\boldsymbol{\theta}$ in $\mathcal{A}'$. Call this correspondence $\boldsymbol{R}$.

Let s be a stack of $\mathcal{A}$ and t be a stack of $\mathcal{A}'$. We say that $s \cong^{\boldsymbol{R}}_{\not p'} t$ just in case:

- $top_1(s) = top_1(t)$
- $pop_1(s) \cong^{\boldsymbol{R}}_{\not p'} \boldsymbol{pop_1}(t)$
- $collapse(s) \cong^{\boldsymbol{R}}_{\not p'} \boldsymbol{collapse}(t)$

The reason using *collapse* in place of pop_k is special (apart from the fact that the latter is never used in $\mathsf{CPDA}(G)$) is that the memory provided by links makes them more robust against certain quirks in the simulating automaton—the target of a link can be described in absolute terms whilst a pop_k is always relative to the top of the stack. This enables the following:

Lemma 3.22. *Let $\mathcal{A}$ and $\mathcal{A}'$ be n-CPDA with each operation θ of $\mathcal{A}$ that is* not *a pop_k operation for $k \geq 2$ bearing a corresponding (compound) operation $\boldsymbol{\theta}$ in $\mathcal{A}'$. Call this correspondence $\boldsymbol{R}$.*

Suppose further that $\boldsymbol{collapse}$ and $\boldsymbol{pop_1}$ consist only of pop_1 and collapse operations. Let s be a stack of $\mathcal{A}$ and let t be a stack of $\mathcal{A}'$. Then $s \cong^{\boldsymbol{R}}_{\not p'} t$ implies $push_k(s) \cong^{\boldsymbol{R}}_{\not p'} push_k(t)$ for every $2 \leq k \leq n$.

Proof. This follows from the fact that $top_k(push_k(s)) = top_k(s)$ (with equality also respecting the absolute target of links) and so in particular for any atom a in $top_k(s)$ we will have $top_k(collapse(s_{\leq a})) = top_k(collapse(s_{\leq push_k(a)}))$. Thus an induction on the length of the compound operation shows that that any compound operation $\vec{\theta}$ in the set $(collapse; +pop_1;)^*$ will have $top_1(\vec{\theta}(s)) = top_1(top_k(\vec{\theta}(a_{\leq s})))$ and indeed $top_1(\vec{\boldsymbol{\theta}}(t)) = top_1(top_k(\vec{\boldsymbol{\theta}}(a_{\leq t})))$ where $\vec{\boldsymbol{\theta}}$ is the corresponding sequence of operations that by assumption must also belong to $(collapse; +pop_1;)^*$. Since $s \cong^{\boldsymbol{R}}_{\not p'} t$ we must therefore get $push_k(s) \cong^{\boldsymbol{R}}_{\not p'} push_k(t)$. □

An n-CPDA with Stepped Collapse

Given a *fresh* n-CPDA $\mathcal{A}$ we now construct an n-CPDA $\mathsf{step}(\mathcal{A})$ using the following recipe:

First add a symbol $\circ$ to the stack alphabet.

- We replace every operation $push_1^{a,k}$ for $k > 2$ in the transition function of $\mathcal{A}$ with the sequence of operations:

  $$push_1^{\circ,k}; push_{k-1}; pop_1; push_1^{\circ,k-1}; push_{k-2}; pop_1; push_1^{\circ,k-2};$$
  $$push_{k-3}; pop_1; push_1^{\circ,k-3}; \cdots ; push_3; pop_1; push_1^{\circ,3}; push_2; pop_1; push_1^{a,2}$$

  Call this compound operation $\boldsymbol{push_1^{a,k}}$.

- If $k \leq 2$, then $push_1^{a,k}$ is just mapped to itself but for notational convenience we denote this $\boldsymbol{push_1^{a,k}}$.

- We replace every operation *collapse* with a sequence of operations beginning with at least one *collapse* and continuing with *collapse* until the top element of the stack is no longer a $\circ$ symbol.

  Call this compound operation ***collapse***.

- A $push_k$ operation for $k \geq 2$ is just mapped to a $push_k$ operation, which we nevertheless write as $\boldsymbol{push_k}$ for notational convenience. Likewise every $push_1^{a,k}$ operation for $k \leq 2$ is mapped to $push_1^{a,k}$, which we write $\boldsymbol{push_1^{a,k}}$ for convenience.

- A pop_1 operation is also just mapped to a pop_1 operation, which again for notational convenience we call $\boldsymbol{pop_1}$

- Since $\mathcal{A}$ is fresh there are no pop_k operations for $k \geq 2$.

Let us name $\boldsymbol{R}$ this correspondence between an operation θ (that is not a higher order pop) of $\mathcal{A}$ to $\boldsymbol{\theta}$ of $\mathsf{step}(\mathcal{A})$.

Lemma 3.23. *Let $\mathcal{A}$ be a* fresh *n-CPDA. Consider a stack s constructible by $\mathsf{step}(\mathcal{A})$ formed from the empty stack by a sequence of $\boldsymbol{\theta}$ operations corresponding to the run constructing s: $s := (\boldsymbol{\theta}_1; \cdots ; \boldsymbol{\theta}_m)(\perp_n)$. Then for every occurence of a symbol b in $top_2(s)$ we have $b \neq \circ$ and $\boldsymbol{l_o}(b) \leq 2$.*

Proof. Argue by induction on m, augmenting the statement of the induction hypothesis with the claims:

1. No ***collapse*** operation performed on an atomic element anywhere in the stack that was initially created as a result of a $\boldsymbol{push_1^{a,k}}$ operation will perform *collapse* on a j-link with $j > k$.

2. $pop_n(s)$ satisfies the augmented induction hypothesis.

3. For any b in $top_2(s)$, $\boldsymbol{collapse}(s_{\leq b})$ also satisfies the augmented induction hypothesis.

Consider each possibility for $\boldsymbol{\theta}_m$ in turn. If $\boldsymbol{\theta}_m = \boldsymbol{push_k}$ the result is straightforward and if it is ***collapse*** then it follows from the third item augmenting the induction hypothesis. If $\boldsymbol{\theta}_m = pop_1$, the result is also an immediate consequence of the induction hypothesis.

The only remaining case is $\boldsymbol{\theta}_m = \boldsymbol{push_1^{a,l}}(\boldsymbol{\theta}_1; \cdots ; \boldsymbol{\theta}_{m-1})(\bot_n)$. If $k \leq 2$, then the result is trivial. If $2 < k < n$, then we observe that $push_k(s)$ must occur previously in the run by the definition of freshness—with intervening $(pop_1 + collapse)^*$. By the condition on the intervening *collapse* together with the first item augmenting the induction hypothesis none of these intervening *collapse* performs *collapse* on a j-link with $j \geq k$. Thus $pop_k((\boldsymbol{\theta}_1; \cdots ; \boldsymbol{\theta}_{m-1})(\bot_n))$ must have occurred previously in the run and so must satisfy the induction hypothesis. If $k = n$, then again $pop_k((\boldsymbol{\theta}_1; \cdots ; \boldsymbol{\theta}_{m-1})(\bot_n))$ satisfies the induction hypothesis by the second item augmenting it.

Let $s_0 := push_1^{\circ,k}((\boldsymbol{\theta}_1; \cdots ; \boldsymbol{\theta}_{m-1})(\bot_n))$ and recursively define

$$s_{i+1} := (push_{k-i}; pop_1; push_1^{\circ,k-i})(s_i)$$

for each $1 \leq i \leq k-3$ For each such i we can easily see by induction on i that $pop_1(s_i)$ satisfies the induction hypothesis, that s_i has a single $\circ$ on top, and that $\boldsymbol{collapse}(s_i)$ also satisfies the induction hypothesis. The base case when $i = 0$ is implied by the paragraph above.

Moreover, $s = (push_2; pop_1; push_1^{a,2})(s_{k-3})$ and so the Lemma does indeed hold for s. □

Lemma 3.24. *Let $\mathcal{A}$ be a* fresh *n-CPDA. Then $[\![\mathcal{A}]\!] = [\![\mathbf{step}(\mathcal{A})]\!]$.*

Proof. For any run of $\mathcal{A}$ applying a sequence of operations $\theta_1; \cdots \theta_m$ to the empty stack $\bot_n$ we claim that:

$$(\theta_1; \cdots ; \theta_m)(\bot_n) \cong_{\not p'}^{\boldsymbol{R}} (\boldsymbol{\theta_1}; \cdots ; \boldsymbol{\theta_m})(\bot_n)$$

We argue by induction on the length m of the run. The base case is just the fact that $\bot_n \cong_{\not p'}^{\boldsymbol{R}} \bot_n$.

- *Case $\theta_m = push_1^{a,k}$ for $k > 2$:* Since $\mathcal{A}$ is fresh, as seen in the proof of Lemma 3.23 it must be the case that $pop_k(\boldsymbol{\theta_1}; \cdots ; \boldsymbol{\theta_{m-1}})(\bot_n)$ occured previously in the run, say after performing $\boldsymbol{\theta_l}$. By the induction hypothesis we have both:

  $$(\theta_1; \cdots ; \theta_l)(\bot_n) \cong_{\not p'}^{\boldsymbol{R}} (\boldsymbol{\theta_1}; \cdots ; \boldsymbol{\theta_l})(\bot_n)$$

  and:

  $$(\theta_1; \cdots ; \theta_{m-1})(\bot_n) \cong_{\not p'}^{\boldsymbol{R}} (\boldsymbol{\theta_1}; \cdots ; \boldsymbol{\theta_{m-1}})(\bot_n)$$

  Let $s := (\theta_1; \cdots ; \theta_m)(\bot_n)$. Let $s_0 := push_1^{\circ,k}((\boldsymbol{\theta_1}; \cdots ; \boldsymbol{\theta_{m-1}})(\bot_n))$. Observe that $collapse(s) = (\theta_1; \cdots ; \theta_l)(\bot_n)$ together with $collapse(s_0) =$

$(\boldsymbol{\theta_1};\cdots;\boldsymbol{\theta_l})(\bot_n)$ and that $pop_1(s) = (\theta_1;\cdots;\theta_{m-1})(\bot_n)$ together with $pop_1(s_0) = (\boldsymbol{\theta_1};\cdots;\boldsymbol{\theta_{m-1}})(\bot_n)$. Thus in particular we get:

$$collapse(s) \cong^{\boldsymbol{R}}_{\not p} collapse(s_0) = \boldsymbol{collapse}(s_0)$$

and:

$$pop_1(s) \cong^{\boldsymbol{R}}_{\not p} pop_1(s_0)$$

Now recursively define $s_{i+1} := push_{k-i}; pop_1; push_1^{\circ,k-i}(s_i)$ for $1 \leq i \leq k-2$. Arguing by induction on i we now check that:

$$collapse(s) \cong^{\boldsymbol{R}}_{\not p} \boldsymbol{collapse}(s_i)$$

and:

$$pop_1(s) \cong^{\boldsymbol{R}}_{\not p} pop_1(s_i)$$

for $1 \leq i \leq k-2$. The first assertion comes from the fact that $collapse(s_{i+1}) = s_i$ and the induction hypothesis. The second assertion comes from the following reasoning. By the induction hypothesis $pop_1(s) \cong^{\boldsymbol{R}}_{\not p} pop_1(s_i)$ and so by Lemma 3.22 $pop_1(s) \cong^{\boldsymbol{R}}_{\not p} push_{k-i}(pop_1(s_i))$. Now observe that $top_{k-i}(pop_1(push_{k-i}(s_i))) = top_{k-i}(push_{k-i}(pop_1(s_i)))$. We may thus conclude that for any b occurring in $top_2(pop_1(push_{k-i}(s_i)))$ we have $collapse(pop_1(push_{k-i}(s_i))_{\leq b}) = collapse(push_{k-i}(pop_1(s_i))_{\leq b})$ if $\boldsymbol{l_o(b)} \geq k-i$ and if $\boldsymbol{l_o(b)} < k-i$, then $top_{k-i}(collapse(pop_1(push_{k-i}(s_i))_{\leq b})) = top_{k-i}(collapse(push_{k-i}(pop_1(s_i))_{\leq b}))$.

This is sufficient to conclude that we also have $pop_1(s) \cong^{\boldsymbol{R}}_{\not p} pop_1(push_{k-i}(s_i))$, *i.e.* that $pop_1(s) \cong^{\boldsymbol{R}}_{\not p} pop_1(s_{i+1})$.

In particular, the following two assertions thus hold:

$$collapse(s) \cong^{\boldsymbol{R}}_{\not p} \boldsymbol{collapse}(s_k) \quad \text{and} \quad pop_1(s) \cong^{\boldsymbol{R}}_{\not p} pop_1(s_k)$$

But $(\boldsymbol{\theta_1};\cdots;\boldsymbol{\theta_m})(\bot_n)$ is just s_k with the top-most element $\circ$ replaced with a. Thus $\boldsymbol{collapse}((\boldsymbol{\theta_1};\cdots;\boldsymbol{\theta_m})(\bot_n)) = \boldsymbol{collapse}(s_k)$ and $\boldsymbol{pop_1}((\boldsymbol{\theta_1};\cdots;\boldsymbol{\theta_m})(\bot_n)) = pop_1(s_k)$. Thus we get:

$$collapse(s) \cong^{\boldsymbol{R}}_{\not p} \boldsymbol{collapse}((\boldsymbol{\theta_1};\cdots;\boldsymbol{\theta_m})(\bot_n))$$

and:

$$pop_1(s) \cong^{\boldsymbol{R}}_{\not p} \boldsymbol{pop_1}((\boldsymbol{\theta_1};\cdots;\boldsymbol{\theta_m})(\bot_n))$$

thereby completing this case of the induction.

- *Case $\theta_m = collapse$ or $\theta_m = pop_1$:* This just follows from the definition of $\cong^{\boldsymbol{R}}_{\not p}$.

- *Case $\theta_m = push_1^{a,k}$ with $k \leq 2$:* In the case when $k = 2$, this is just the base case from the first item (when $\theta_m = push_1^{a,k}$ for $k > 2$). When $k = 1$ it is trivial to check that the inductive step follows from the induction hypothesis.

- *Case $\theta_m = push_k$ for $k \geq 2$:* Appeal to Lemma 3.22.

We may thus conclude that $\mathcal{A}'$ is an adequate simulation of $\mathcal{A}$. □

Lemma 3.25. *Let $\mathcal{A}$ be a fresh n-CPDA. Then the automaton $\mathbf{step}(\mathcal{A})$ has the following characteristics:*

- *It only ever performs a $push_1^{a,l}$ operation on a stack s such that $\boldsymbol{l_o}(top_1(s)) \leq 2$.*

- *It only ever performs a $push_k$ operation on a stack s when $\boldsymbol{l_o}(top_1(s)) \leq k + 1$*

Proof. Both can easily be seen by inspection of the $\boldsymbol{\theta}$ operations, with the first observation making use of Lemma 3.23 and the fact that any element created by $push_1^{a,l}$ with $l > 2$ is pop_1ed off a copy of the stack before the next order-1 push is performed. □

Lemma 3.26. *Let $\mathcal{A}$ be a* fresh *n-CPDA. Then there exists a homogeneous recursion scheme $G \in \boldsymbol{\mathcal{RS}_n}$ such that $[\![\mathbf{step}(\mathcal{A})]\!] = [\![G]\!]$.*

Proof. We adapt the construction by Hague *et al.* [40]. Let Γ be the stack-alphabet and let Q be the set of control-states $q_1, \ldots, q_{|Q|}$. Their construction made use of a set of non-terminals of the form $\mathcal{F}^{q,a,l}$ for every $q \in Q$, $a \in \Gamma$ and $1 \leq l \leq n$ with

$$\mathbf{Ty}(\mathcal{F}^{q,a,l}) := (\boldsymbol{n-l})^{\prime|Q|} \longrightarrow (\boldsymbol{n-1})^{\prime|Q|} \longrightarrow (\boldsymbol{n-2})^{\prime|Q|} \longrightarrow \cdots \longrightarrow (\boldsymbol{1})^{\prime|Q|} \longrightarrow (\boldsymbol{0})^{\prime|Q|} \longrightarrow o$$

where $(\boldsymbol{0})' := o$ and for $m \geq 1$:

$$(\boldsymbol{m})' := (\boldsymbol{m-1})^{\prime|Q|} \longrightarrow (\boldsymbol{m-2})^{\prime|Q|} \longrightarrow \cdots \longrightarrow (\boldsymbol{1})^{\prime|Q|} \longrightarrow (\boldsymbol{0})^{\prime|Q|} \longrightarrow o$$

Whenever $\delta(q, a) := (q_i, \theta)$ for some stack operation θ (where δ is the transition function of the n-CPDA) Hague *et al.* have a rule:

$$\mathcal{F}^{q,a,l}\vec{\phi}\vec{\Psi}_{n-1}\vec{\Psi}_{n-2}\cdots\vec{\Psi}_2\vec{\Psi}_1\vec{\Psi}_0 \longrightarrow \Xi_{q_i,\theta}$$

where

$$\Xi_{q_i,\theta} := \begin{cases} \mathcal{F}^{q_i,a,l}\vec{\phi}\vec{\Psi}_{n-1}\vec{\Psi}_{n-2}\cdots\vec{\Psi}_{n-k+1}\left(\mathcal{F}^{q_1,a,l}\vec{\phi}\vec{\Psi}_{n-1}\vec{\Psi}_{n-2}\cdots\vec{\Psi}_{n-k}\right) \\ \quad \cdots\left(\mathcal{F}^{q_{|Q|},a,l}\vec{\phi}\vec{\Psi}_{n-1}\vec{\Psi}_{n-2}\cdots\vec{\Psi}_{n-k}\right)\Psi_{n-k-1}\cdots\vec{\Psi}_2\vec{\Psi}_1\vec{\Psi}_0 \\ \qquad \text{if } \theta = push_k \text{ for } k \geq 2 \\ \mathcal{F}^{q_i,b,k}\vec{\Psi}_{n-l}\left(\mathcal{F}^{q_1,a,l}\vec{\phi}\vec{\Psi}_{n-1}\right)\cdots\left(\mathcal{F}^{q_{|Q|},a,l}\vec{\phi}\vec{\Psi}_{n-1}\right)\Psi_{n-2}\cdots\vec{\Psi}_2\vec{\Psi}_1\vec{\Psi}_0 \\ \qquad \text{if } \theta = push_1^{b,k} \\ \Psi^i_{n-k}\vec{\Psi}_{n-k-1}\cdots\vec{\Psi}_2\vec{\Psi}_1\vec{\Psi}_0 \qquad \text{if } \theta = pop_k \\ \phi^i\vec{\Psi}_{n-l-1}\cdots\vec{\Psi}_2\vec{\Psi}_1\vec{\Psi}_0 \qquad \text{if } \theta = collapse \end{cases}$$

where Ψ^j_m is the jth variable in the vector $\vec{\Psi}_m$.

Now let us introduce some homogeneously typed non-terminals of the form $\mathcal{H}^{q,a,l}$ for every $q \in Q$, $a \in \Gamma$ and $1 \leq l \leq n$ with:

$$\mathbf{Ty}(\mathcal{H}^{q,a,l}) := (\boldsymbol{n-1})'^{|Q|} \longrightarrow (\boldsymbol{n-2})'^{|Q|} \longrightarrow \cdots \longrightarrow (\boldsymbol{n-l+1})'^{|Q|} \longrightarrow (\boldsymbol{n-l})'^{2.|Q|} \longrightarrow (\boldsymbol{n-l-1})'^{|Q|} \longrightarrow \cdots \longrightarrow (\boldsymbol{1})' \longrightarrow (\boldsymbol{0})' \longrightarrow o$$

Note that each $(\boldsymbol{m})'$ is homogeneous and so the types of the form above must be homogeneous. The main difference to before is that the extra place holders for the encoding of the target of the link is moved in such a way that homogeneity is obtained. We now observe that Lemma 3.25 tells us that $\mathbf{step}(\mathcal{A})$ has properties enabling the $\mathcal{H}$-non-terminals to replace the $\mathcal{F}$-non-terminals in the encoding. For simplicity we assume w.l.o.g. that no 1-links are used (these can always be simulated with a pop_1 operation).

$$\mathcal{H}^{q,a,l}\vec{\Psi}_{n-1}\vec{\Psi}_{n-2}\cdots\vec{\Psi}_{n-l+1}\vec{\phi}\vec{\Psi}_{n-l}\vec{\Psi}_{n-l-1}\cdots\vec{\Psi}_2\vec{\Psi}_1\vec{\Psi}_0 \longrightarrow \Xi'_{q_i,\theta}$$

$$\Xi'_{q_i,\theta} := \begin{cases} \mathcal{H}^{q_i,a,l}\Psi_{\vec{n-1}}\Psi_{\vec{n-2}}\cdots\Psi_{\vec{n-l+1}}\vec{\phi}\cdots\Psi_{\vec{n-k+1}}\left(\mathcal{H}^{q_1,a,l}\Psi_{\vec{n-1}}\Psi_{\vec{n-2}}\cdots\Psi_{\vec{n-l+1}}\vec{\phi}\cdots\Psi_{\vec{n-k}}\right) \\ \quad\cdots\left(\mathcal{H}^{q_{|Q|},a,l}\Psi_{\vec{n-1}}\Psi_{\vec{n-2}}\cdots\Psi_{\vec{n-l+1}}\vec{\phi}\cdots\Psi_{\vec{n-k}}\right)\Psi_{\vec{n-k-1}}\cdots\vec{\Psi}_2\vec{\Psi}_1\vec{\Psi}_0 \\ \qquad \text{if } \theta = push_k \text{ for } k \geq 2 \text{ when } l < k \\ \mathcal{H}^{q_i,a,l}\Psi_{\vec{n-1}}\Psi_{\vec{n-2}}\cdots\Psi_{\vec{n-k+1}}\vec{\phi}\left(\mathcal{H}^{q_1,a,l}\Psi_{\vec{n-1}}\Psi_{\vec{n-2}}\cdots\Psi_{\vec{n-k+1}}\vec{\phi}\Psi_{\vec{n-k}}\right) \\ \quad\cdots\left(\mathcal{H}^{q_{|Q|},a,l}\Psi_{\vec{n-1}}\Psi_{\vec{n-2}}\cdots\Psi_{\vec{n-k+1}}\vec{\phi}\Psi_{\vec{n-k}}\right)\Psi_{\vec{n-k-1}}\cdots\vec{\Psi}_2\vec{\Psi}_1\vec{\Psi}_0 \\ \qquad \text{if } \theta = push_k \text{ for } k \geq 2 \text{ when } k = l \\ \mathcal{H}^{q_i,a,l}\Psi_{\vec{n-1}}\Psi_{\vec{n-2}}\cdots\Psi_{\vec{n-k+1}}\left(\mathcal{H}^{q_1,a,l}\Psi_{\vec{n-1}}\Psi_{\vec{n-2}}\cdots\Psi_{\vec{n-k}}\vec{\phi}\right) \\ \quad\cdots\left(\mathcal{H}^{q_{|Q|},a,l}\Psi_{\vec{n-1}}\Psi_{\vec{n-2}}\cdots\Psi_{\vec{n-k}}\vec{\phi}\right)\vec{\phi}\Psi_{\vec{n-k-1}}\cdots\vec{\Psi}_2\vec{\Psi}_1\vec{\Psi}_0 \\ \qquad \text{if } \theta = push_k \text{ for } k \geq 2 \text{ when } l = k+1 \\ \\ \mathcal{H}^{q_i,b,k}\left(\mathcal{H}^{q_1,a,l}\Psi_{\vec{n-1}}\vec{\phi}\right)\cdots\left(\mathcal{H}^{q_{|Q|},a,l}\Psi_{\vec{n-1}}\vec{\phi}\right)\Psi_{\vec{n-2}}\cdots\Psi_{\vec{n-k}}\Psi_{\vec{n-k}}\cdots\vec{\Psi}_2\vec{\Psi}_1\vec{\Psi}_0 \\ \qquad \text{if } \theta = push_1^{b,k} \text{ and } l = 2 \\ \\ \Psi^i_{n-k}\Psi_{\vec{n-k-1}}\cdots\vec{\Psi}_2\vec{\Psi}_1\vec{\Psi}_0 \qquad \text{if } \theta = pop_k \\ \phi^i_{n-l}\Psi_{\vec{n-l-1}}\cdots\vec{\Psi}_2\vec{\Psi}_1\vec{\Psi}_0 \qquad \text{if } \theta = collapse \end{cases}$$

The term $\Xi'_{q_i,\theta}$ is defined in sufficient cases when $\theta = push_k$ by Lemma 3.25. The same Lemma also tells us that in $l = 2$ is always a valid assumption when a $push_1^{b,k}$ operation is performed (combined with the assumption that 1-links are never used). It should also be clear that the rules for $\mathcal{H}$ behave exactly the same way as the rules for $\mathcal{F}$—the only difference is the ordering of the arguments. Thus the $\mathcal{H}$ rules inherit the correctness proof from Hague *et al.* [40]. □

As a consequence we get

Theorem 3.27. *For every recursion scheme $G \in \boldsymbol{\mathcal{R}_{\mathcal{S}_n}}$ there exists a* homogeneously typed *recursion scheme $G' \in \boldsymbol{\mathcal{R}_{\mathcal{S}_n}}$ such that $[\![G]\!] = [\![G']\!]$.*

Proof. This is a consequence of combining Lemma 3.20, Lemma 3.24 and Lemma 3.26. □

Chapter 4

The Collapse of First-Order Logic. Undecidability and CPDA Graphs

In the light of Karzow's demonstration that first-order logic is decidable on the ϵ-closures of 2-CPDA graphs [48] we now turn our attention to the model-checking problem for first-order logic and n-CPDA for $n > 2$—the problem that given an n-CPDA and sentence of first-order logic seeks to determine whether the sentence holds on the (ϵ-closure of the) CPDA graph. This chapter describes some surprisingly strong undecidability results, showing that the quest for decidability will force us to consider ways to restrict CPDA as well as first-order logic. The restrictions include turning ϵ-closure on and off; being more discriminating with the order of links that may be used and limiting quantifier alternation depth in the logic.

4.1 Post's Correspondence Problem

All of the undecidability results go via a reduction from Post's Correspondence Problem [63], which is known to be undecidable. Consider a finite-alphabet Σ with $|\Sigma| \geq 2$. An instance of the Post Correspondence Problem (PCP) consists of two finite sequences of strings over Σ: $u_1, \ldots, u_m$ and $v_1, \ldots, v_m$. The question to be decided is whether there is a finite sequence $i_1 \ldots i_k$ consisting of integers $1 \leq i_j \leq m$ such that $u_{i_1}.u_{i_2}.\cdots.u_{i_k} = v_{i_1}.v_{i_2}.\cdots.v_{i_k}$.

Example 4.1. Consider the following two sets of strings over the alphabet $\{a, b, c\}$:

$$u_1 := ab \quad u_2 := cababcabb \quad u_3 := ca$$
$$v_1 := ababc \qquad v_2 := ab \qquad v_3 := bca$$

Then the Post Correspondence Problem has answer 'yes' as witnessed by the solution: 1123. That is:

$$u_1.u_1.u_2.u_3 = ababcababcabbca = v_1.v_1.v_2.v_3$$

We can formulate the problem in terms of a pushdown stack, which will be a more useful presentation for our purposes. Given an instance of the PCP P (using the notation above) we define a pushdown automaton $\mathcal{A}_1^P$ that pushes elements of Σ onto the stack together with indices indicating a breakdown of the stack contents into strings from amongst the u_i and v_i.

Definition 4.2. Let P be an instance of Post's Correspndence Problem (using the notation above). The automaton $\mathcal{A}_1^P$ has stack alphabet:

$$\Sigma \cup [1_u, 2_u \dots m_u] \cup [1_v, 2_v \dots m_v]$$

It behaves by non-deterministically choosing one of the following options:

- Push any member of Σ onto the stack.
- If the Σ symbols in the stack since the last symbol of the form i_u (or the bottom of the stack if there is no such symbol) form the word u_j, then it may push j_u onto the stack.
- If the Σ symbols in the stack since the last symbol of the form i_v (or the bottom of the stack if there is no such symbol) form the word v_j, then it may push j_v onto the stack.

The condition on the final two options can be detected using a finite number of control-states since the lengths of the finite number of strings are bounded.

It is easy to see that P has a solution just in case $\mathcal{A}_1^P$ can generate a stack with 'matching' i_u and i_v subsequences. More precisely:

Lemma 4.3. *Let P be an instance of Post's Correspondence Problem. P has a solution just in case the automaton $\mathcal{A}_1^P$ can generate a stack s such that:*

- *$s_u = s_v$ where s_u is the subsequence of s consisting of elements of the form i_u and s_v of elements of the form i_v and equality is interpreted with respect to the indices i only.*
- *The top two elements of s form the set $\{i_u, i_v\}$ for some $1 \leq i \leq m$.*

Proof. This is a quick consequence of definitions. Suppose that there is a solution $\sigma = u_{i_1} u_{i_2} \cdots u_{i_k} = v_{i_1} v_{i_2} \cdots v_{i_k}$ to P. Then $\mathcal{A}_1^P$ may construct a stack meeting the criteria by pushing the letters in σ onto the stack whilst also pushing i_{r_u} onto the stack after completing the pushing of u_{i_r} and likewise pushing i_{r_v} after completing v_{i_r}.

Conversely suppose that there is a stack s meeting the criteria. Let $\sigma := \pi_\Sigma(s)$, which we claim is a string generated by a solution to P. Let i_{r_u} be the rth element of the form j_u in the stack. Let i_{r_v} be the rth element of the form j_v in the stack. Note that this is well defined since the first criterion requires

the subsequences of j_u and and j_v to be the same. Let k be the length of these two sequences.

Define i_{0u} and i_{0v} to be the bottom of the stack. Divide σ into segments u'_r (for $1 \leq r \leq k$) where u_{r+1} is the subsequence of letters (in Σ) from s that begins immediately after i_{ru} and ends immediately before i_{r+1_u}. Define v'_r in a similar manner using i_{rv}. The first criterion ensures that $u'_r = u_{i_r}$ and that $v'_r = v_{i_r}$ for every $1 \leq r \leq k$. Moreover the second criterion ensures that $u'_1 u'_2 \cdots u'_k = v'_1 v'_2 \cdots v'_k = s$. That is $u_{i_1} u_{i_2} \cdots u_{i_k} = v_{i_1} v_{i_2} \cdots v_{i_k} = s$ and so we do indeed have a solution to P. □

Example 4.4. To continue the running example from Example 4.1, which we call P, the solution as represented by a stack of $\mathcal{A}_1^P$ is:

$$[\ ab\mathbf{1}_{\boldsymbol{u}}ab\mathbf{1}_{\boldsymbol{u}}c\mathbf{1}_{\boldsymbol{v}}ababc\mathbf{1}_{\boldsymbol{v}}ab\mathbf{2}_{\boldsymbol{v}}b\mathbf{2}_{\boldsymbol{u}}ca\mathbf{3}_{\boldsymbol{u}}\mathbf{3}_{\boldsymbol{v}}\ ]$$

4.2 Post's Correspondence Problem and 2-CPDA

Hague *et al.* [40] showed that the model-checking problem for MSO on 2-CPDA graphs is undecidable; indeed the 2-CPDA graph that they exhibit witnesses the undecidability of transitive closure logic $\boldsymbol{FO(TC)}$[1].

In order to introduce our basic technique, we first reprove the undecidability of $\boldsymbol{FO(TC)}$ on 2-CPDA graphs by a reduction from PCP.

We first introduce a 2-CPDA $\mathcal{A}_2^P$ for each PCP instance P. This is very like $\mathcal{A}_1^P$ except that it ensures each index (the elements of the form i_u or i_v) emanate a pointer to a distinct 1-stack in the 2-stack. This will enable first-order logic to 'ascertain corresponding positions' in two instances of a 1-stack by comparing the results of collapsing.

Definition 4.5. Let P be an instance of Post's Correspndence Problem (using the notation above). The automaton $\mathcal{A}_2^P$ has stack alphabet:

$$\Sigma \cup [1_u, 2_u \ldots m_u] \cup [1_v, 2_v \ldots m_v]$$

It behaves by non-deterministically choosing one of the following options:

- Push any member of Σ onto the stack.
- If the Σ symbols in the stack since the last symbol of the form i_u in the top 1-stack (or the bottom of the 1-stack if there is no such symbol) form the word u_j, then it may perform $push_2; push_1^{j_u,2}$.
- If the Σ symbols in the stack since the last symbol of the form i_v in the top 1-stack (or the bottom of the 1-stack if there is no such symbol) form the word v_j, then it may perform $push_2; push_1^{j_v,2}$.

[1] We are grateful to the anonymous reviewer of our submission to LICS 2011 for pointing this out.

As with $\mathcal{A}_1^P$, the finite control-states can enforce the precondition on the second and third options.

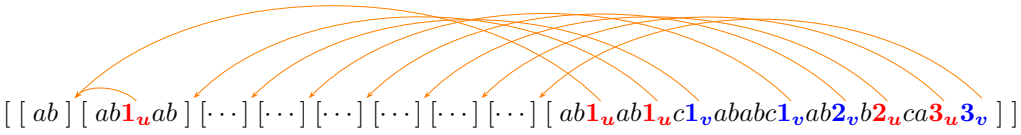
A solution to P can be formulated in terms of $\mathcal{A}_2^P$ in a very similar manner to before:

Lemma 4.6. *Let P be an instance of Post's Correspondence Problem. P has a solution just in case the automaton $\mathcal{A}_2^P$ can generate a stack s such that:*

- *$s_u = s_v$ where s_u is the subsequence of $top_2(s)$ consisting of elements of the form i_u and s_v of elements of the form i_v and equality is interpreted with respect to the indices i only.*
- *The top two elements of s form the set $\{i_u, i_v\}$ for some $1 \leq i \leq m$.*

Proof. A consequence of Lemma 4.3 given that the permissible changes to the top_2 1-stack of the 2-stack of $\mathcal{A}_2^P$ at any given point in the run is precisely the same as those that could be made to the 1-stack of $\mathcal{A}_1^P$. □

Example 4.7. To continue the running example from Example 4.1, the solution as represented by a stack of $\mathcal{A}_2^P$ is:



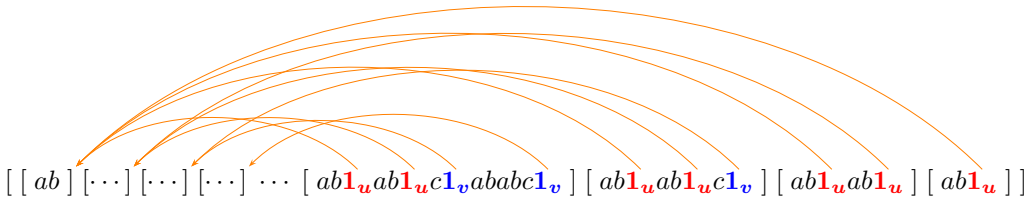
Now consider a variant of $\mathcal{A}_2^P$ which we will call $\mathcal{A}_{2^+}^P$. This behaves as follows:

- It initially behaves as $\mathcal{A}_2^P$. It may terminate this phase if its top two elements are in $\{i_u, i_v\}$ for some i. When it terminates this phase it enters a distinguished control-state ***guess***.
- It then performs a sequence of operations of the form: $push_2; pop_1^{m_1}; push_2; pop_1^{m_2}; push_2; pop_1^{m_3}; push_2; pop_1^{m_4}$ with $m_1, m_2, m_2, m_4 \in \mathbb{N}$. It should ensure that the four 1-stacks so created form a set $\{u_1, u_2, v_1, v_2\}$ (in no particular order) such that:
  - $top_1(u_1) = i_u$ and $top_1(v_1) = i_v$ for some $1 \leq i \leq m$.
  - Either $top_1(u_2) = i'_u$ and $top_1(v_2) = i'_v$ for some $1 \leq i' \leq m$ or else u_2 and v_2 are both empty.
  - Going from u_1 to u_2 can be done with the popping of precisely one symbol of the form j_u (that is the one on top of u_1) and all other symbols popped should be letters.
  - Going from v_1 to v_2 can be done with the popping of precisely one symbol of the form k_v (that is the one on top of u_1).

Compliance with these requirements can be checked using a finite number of control states. If the automaton finds that it cannot avoid violating a requirement (*e.g.* it reaches the bottom of a stack whilst iterating pop_1) then it just enters a distinguished control-state ***fail***. Otherwise it enters one of the following distinguished control-states: $\boldsymbol{u_1v_1u_2v_2}^{\boldsymbol{start}}$, $\boldsymbol{v_1u_1v_2u_2}^{\boldsymbol{start}}$ or $\boldsymbol{u_1u_2v_1v_2}$, $\boldsymbol{u_1v_1u_2v_2}$, $\boldsymbol{u_1v_1v_2u_2}$, $\boldsymbol{v_1u_1u_2v_2}$, $\boldsymbol{v_1u_1v_2u_2}$, $\boldsymbol{v_1v_2u_1u_2}$ or $\boldsymbol{u_1v_1}^{\boldsymbol{end}}$, $\boldsymbol{v_1u_1}^{\boldsymbol{end}}$ specifying the order of creation of these four stacks along with a flag indicating ***end*** if the u_2 and v_2 are empty and ***start*** if both u_1 and u_2 have the top-most i_u element of s and both v_1 and v_2 the top-most i_v element. Call these control-states *verifier states*.

- From one of the above verifier states, the automaton may perform any sequence of of pop_2 and *collapse* operations via edges labelled $\boldsymbol{pop_2}$ and ***collapse*** which all end in control-state ***test***.

Example 4.8. The automaton $\mathcal{A}^P_{2+}$ could reach the stack in Example 4.7 in control-state ***guess***. From here it could, for example, reach the configuration consisting of stack:



in control-state $\boldsymbol{v_1v_2u_1u_2}$.

Definition 4.9. Let us fix some configuration of $\mathcal{A}^P_{2+}$ with stack s and control-state ***guess***. Let us call the configurations reachable from $(\boldsymbol{guess}, s)$ associated with a verifier state the *s-verifier configurations*. Given an s-verifier configuration c (with set of top four stacks $\{u_1, u_2, v_1, v_2\}$) we define its *successor* c^+ to be the configuration (which is unique if it exists) with top four stacks $\{u_1^+, u_2^+, v_1^+, v_2^+\}$ such that $u_1^+ = u_2$ and $v_1^+ = v_2$.

Additionally the unique s-verifier bearing the ***start*** flag is dubbed 'the 0 s-verifier' 0_s. The unique s-verifier configuration whose control-state bears the ***end*** flag is dubbed *the s-terminal verifier* $\boldsymbol{end}_s$.

We can now produce a version of Lemma 4.6 in terms of s-verifier configurations.

Lemma 4.10. *Let $(\boldsymbol{guess}, s)$ be a reachable configuration of $\mathcal{A}^P_{2+}$ for some instance P of Post's Correspondence Problem. Then s represents a solution to P in the sense of Lemma 4.6 if and only if there exists a chain of stacks $s_1, s_2, \ldots, s_k$ such that $s_1 = 0_s$, $s_k = \boldsymbol{end}_s$ and $s_{i+1} = s_i^+$ for each $1 \leq i \leq k-1$ and such that for each i there exists a reachable s_i-verifier.*

Proof. From the definitions of $+$ and the relationship between u_1 and u_2 in each verifier configuration, it follows that the top element of u_2 is the element coming after the top element of u_2^+ in the subsequence of elements in $top_2(s)$ consisting of elements of the form i_u. The same holds for v_2 and v_2^+ and the subsequence of elements in $top_2(s)$ consisting of elements of the form i_v.

Also note that the top elements of u_2 and v_2 in 0_s will respectively be the last i_u and i_v in each of these subsequences and the top elements of u_1 and v_1 in $\boldsymbol{end}_s$ will be the first (since by definition of $\boldsymbol{end}_s$ there cannot be any i_u or i_v lying below the top element of u_1 and v_1 respectively).

Now we can get by an easy induction on r that if there is a chain $s_1, s_2, \ldots, s_r$ such that $s_1 = 0_s$ and $s_{i+1} = s_i^+$ for each $1 \leq i < r$, then the following three assertions are true:

- The top element of the u_2 of s_r is the rth element from the end of the subsequence of $top_2(s)$ consisting of elements of the form i_u.
- The top element of the v_2 of s_r is the rth element from the end of the subsequence of $top_2(s)$ consisting of elements of the form i_v.
- If the top element of the u_2 of s_j for $1 \leq j \leq r$ is i_{j_u}, then the top element of the v_2 of s_j is i_{j_v}.

As a consequence, we get that the existence of such a chain implies that the final r elements of the subsequence of $top_2(s)$ consisting of elements of the form i_u matches the last r elements of the subsequence of $top_2(s)$ consisting of elements of the form i_v.

So suppose there exists a chain $s_1, s_2, \ldots, s_k$ such that $s_1 = 0_s$, $s_k = \boldsymbol{end}_s$ and $s_{i+1} = s_i^+$ for each $1 \leq i \leq k-1$. The observations above ensure that s meets the conditions of Lemma 4.6 and so we may conclude that s does indeed witness a solution to P.

Conversely let us begin by assuming that s witnesses a solution to P. It must then be the case that $top_2(s)$ satisfies the properties laid out in Lemma 4.6. It is again an easy induction on r to see that under such circumstances one can generate a sequence $s_1, s_2, \ldots, s_r$ where $s_1 = 0_s$ and $s_{i+1} = s_i^+$ for each $1 \leq i < r$ so long as r does not exceed the length of the subsequence of $top_2(s)$ consisting of elements of the form i_u (which is the same as that consisting of elements of the form i_v). The induction step just needs to observe that we should order the $u_1^+, u_2^+, v_1^+, v_2^+$ with decreasing height, to enable the next to be formed from pop_1ing from the previous.

Finally observe that when r reaches the length of the subsequences, the u_2 and v_2 of s_r will have top elements corresponding to the initial i_u and i_v in the subsequences of $top_2(s)$ given by Lemma 4.6. This means that s_r^+ will have u_1 and v_1 with these same initial elements and so u_2 and v_2 must be empty. That is $s_r^+ = \boldsymbol{end}_s$. Hence we construct a chain of the required form. □

We now consider how transitive closure logic can be used to assert the existence of a $(\boldsymbol{guess}, s)$ configuration from which a chain of the form in Lemma 4.10 can be realised. In particular we need to be able to define the $^+$ operation. This can be achieved by ensuring that the element on top of u_2 in a verifier-configuration stack s is the same as the element on top of u_1 in the purported s^+. Because every element in a $\mathcal{A}^P_{2^+}$ 1-stack has a pointer to a different location, we can detect this using first-order logic by checking that the result of collapsing on either of these two elements results in the same stack (equality). Individual $^+$ steps can be extended to a chain via transitive closure.

Lemma 4.11. *There exists a Σ_1 sentence ϕ of $\boldsymbol{FO(TC)}$ containing only derived predicates $\overline{\psi}$ formed from Δ_1 formulae such that for all instances P of Post's Correspondence Problem we have:*

$$\mathcal{G}(\mathcal{A}^P_{2^+}) \models \phi \text{ iff } P \text{ has a solution.}$$

Proof. Let us first exhibit formulae witnessing the fact that the relation 'for some stack s x is an s-verifier and $y = x^+$' is Δ_1-definable. The following is a Σ_1 formula representing the relation:

$$\begin{gathered}
\exists z.\,(\exists w_1 w_2 w_3.\,(\boldsymbol{pop_2}(x, w_1) \wedge \boldsymbol{pop_2}(w_1, w_2) \wedge \boldsymbol{pop_2}(w_2, w_3) \wedge \boldsymbol{pop_2}(w_3, z)) \\
\exists w'_1 w'_2 w'_3.\,(\boldsymbol{pop_2}(y, w'_1) \wedge \boldsymbol{pop_2}(w'_1, w'_2) \wedge \boldsymbol{pop_2}(w'_2, w'_3) \wedge \boldsymbol{pop_2}(w'_3, z))) \wedge \\
\bigvee_{\substack{\{a,b,c,d\} \\ =\{u_1,u_2,v_1,v_2\}}} \boldsymbol{abcd}(x) \;\wedge\; \bigvee_{\substack{\{a,b,c,d\} \\ =\{u_1,u_2,v_1,v_2\}}} \boldsymbol{abcd}(y) \wedge \\
((\boldsymbol{u_1 u_2 v_1 v_2}(x) \wedge \boldsymbol{u_1 u_2 v_1 v_2}(y)) \rightarrow ((\exists w_1 w_2.\exists w'_1 w'_2 w'_3.\exists c_u. \\
\boldsymbol{pop_2}(x, w_1) \wedge \boldsymbol{pop_2}(w_1, w_2) \wedge \boldsymbol{collapse}(w_2, c_u) \\
\boldsymbol{pop_2}(y, w'_1) \wedge \boldsymbol{pop_2}(w'_1, w'_2) \wedge \boldsymbol{pop_2}(w'_2, w'_3) \wedge \boldsymbol{collapse}(w'_3, c_u)) \\
\wedge(\exists w'_1.\exists c_v.\boldsymbol{collapse}(x, c_v) \wedge \boldsymbol{pop_2}(y, w'_1) \wedge \boldsymbol{collapse}(w'_1, c_v)))) \wedge \cdots
\end{gathered}$$

$$\begin{gathered}
\cdots \wedge ((\boldsymbol{v_1 u_1 u_2 v_2}(x) \wedge \boldsymbol{u_1 v_1 u_2 v_2}(y)) \rightarrow ((\exists w_1.\exists w'_1 w'_2 w'_3.\exists c_u. \\
\boldsymbol{pop_2}(x, w_1) \wedge \boldsymbol{collapse}(w_1, c_u) \wedge \\
\boldsymbol{pop_2}(y, w'_1) \wedge \boldsymbol{pop_2}(w'_1, w'_2) \wedge \boldsymbol{pop_2}(w'_2, w'_3) \wedge \boldsymbol{collapse}(w'_3, c_u)) \\
\wedge (\exists w'_1 w'_2.\exists c_v.\boldsymbol{collapse}(x, c_v) \wedge \\
\boldsymbol{pop_2}(y, w'_1) \wedge \boldsymbol{pop_2}(w'_1, w'_2) \wedge \boldsymbol{collapse}(w'_2, c_v)))) \\
\\
\wedge \cdots
\end{gathered}$$

We have only mentioned two of the $6 \times 8 + 2 \times 8 = 64$ elements of the final conjunction in the formula above, but the remainder follow the same pattern. There are 6 different predicates that characteise the order of the $u_1, u_2, v_1 v_2$ in

x but 8 for y as these may enjoy the ***end*** flag and then there are additionally two with a ***start*** flag that x may exhibit. The formula correctly captures the relation $y = x^+$ since the pointer from each atomic element in the stack is assigned a different target to its 2-pointer when it is created. In the final conjunction, c_u represents the common target of the u_2 in x and u_1 in y and c_v the common target of the v_2 in x and the v_1 in y. The first clauses of the formula assert that x and y are both s-verifiers for some particular fixed stack s (embodied by the configuration bound to z).

We now exhibit a Π_1 formula defining this relation, thereby showing that the relation is Δ_1. This is effectively a rehashing of the Σ_1 formula above, exploiting the fact that all of the relations used are destructive operations on some fixed stacks and are consequently 'functional' (the result of a particular destructive operation on a fixed stack always gives a unique result):

$$\begin{aligned}
&\forall z.\forall w_1 w_2 w_3.\forall w_1' w_2' w_3' \,(\boldsymbol{pop_2}(x, w_1) \rightarrow \boldsymbol{pop_2}(w_1, w_2) \rightarrow \boldsymbol{pop_2}(w_2, w_3) \rightarrow \boldsymbol{pop_2}(w_3, z) \\
&\quad \rightarrow \boldsymbol{pop_2}(y, w_1') \rightarrow \boldsymbol{pop_2}(w_1', w_2') \rightarrow \boldsymbol{pop_2}(w_2', w_3') \rightarrow \boldsymbol{pop_2}(w_3', z)) \wedge \\
&\qquad \bigvee_{=\{u_1,u_2,v_1,v_2\}}^{\{a,b,c,d\}} \boldsymbol{abcd}(x) \;\wedge \bigvee_{=\{u_1,u_2,v_1,v_2\}}^{\{a,b,c,d\}} \boldsymbol{abcd}(y) \wedge \\
&\qquad ((\boldsymbol{u_1 u_2 v_1 v_2}(x) \wedge \boldsymbol{u_1 u_2 v_1 v_2}(y)) \rightarrow ((\forall w_1 w_2.\forall w_1' w_2' w_3'.\forall c_u. \\
&\qquad \boldsymbol{pop_2}(x, w_1) \rightarrow \boldsymbol{pop_2}(w_1, w_2) \rightarrow \boldsymbol{collapse}(w_2, c_u) \\
&\qquad \boldsymbol{pop_2}(y, w_1') \rightarrow \boldsymbol{pop_2}(w_1', w_2') \rightarrow \boldsymbol{pop_2}(w_2', w_3') \rightarrow \boldsymbol{collapse}(w_3', c_u)) \\
&\qquad \wedge(\forall w_1'.\forall c_v.\boldsymbol{collapse}(x, c_v) \rightarrow \boldsymbol{pop_2}(y, w_1') \rightarrow \boldsymbol{collapse}(w_1', c_v)))) \wedge \cdots
\end{aligned}$$

$$\begin{aligned}
&\cdots \wedge ((\boldsymbol{v_1 u_1 u_2 v_2}(x) \forall \boldsymbol{u_1 v_1 u_2 v_2}(y)) \rightarrow ((\forall w_1.\forall w_1' w_2' w_3'.\forall c_u. \\
&\qquad \boldsymbol{pop_2}(x, w_1) \rightarrow \boldsymbol{collapse}(w_1, c_u) \rightarrow \\
&\qquad \boldsymbol{pop_2}(y, w_1') \rightarrow \boldsymbol{pop_2}(w_1', w_2') \rightarrow \boldsymbol{pop_2}(w_2', w_3') \rightarrow \boldsymbol{collapse}(w_3', c_u)) \\
&\qquad \wedge (\forall w_1' w_2'.\forall c_v.\boldsymbol{collapse}(x, c_v) \rightarrow \\
&\qquad \boldsymbol{pop_2}(y, w_1') \rightarrow \boldsymbol{pop_2}(w_1', w_2') \rightarrow \boldsymbol{collapse}(w_2', c_v))))
\end{aligned}$$

$$\wedge \cdots$$

Let $\phi^+(x, y)$ be either of the formulae above.We can now define what it means to have a chain from x to y of s-verifiers (for some s) in $\boldsymbol{FO(TC)}$ using transitive closure on only a Δ_1-definable relation:

$$\overline{\phi^+}(x, y)$$

We can easily assert that y is equal to $\boldsymbol{end}_s$ for some stack s with:

$$\boldsymbol{u_1 v_1^{end}} \vee \boldsymbol{v_1 u_1^{end}}$$

Likewise we can assert that x is equal to 0_s for some stack s with:

$$\boldsymbol{u_1u_2v_1v_2}^{\boldsymbol{start}}(x) \vee \boldsymbol{v_1v_2u_1u_2}^{\boldsymbol{start}}(x)$$

By Lemma 4.10 we can construct the required Σ_1 sentence ϕ by putting all of the above together to get:

$$\exists x.\exists y.\left(\left(\boldsymbol{u_1u_2v_1v_2}^{\boldsymbol{start}}(x) \vee \boldsymbol{v_1v_2u_1u_2}^{\boldsymbol{start}}(x)\right) \wedge \bigvee_{\substack{\{a,b,c,d\} \\ =\{u_1,u_2,v_1,v_2\}}} \boldsymbol{abcd}^{\boldsymbol{end}}(y) \wedge \overline{\phi^+}(x,y)\right)$$

□

We have thus reduced PCP to $\boldsymbol{FO(TC)}$-model-checking on 2-CPDA and so this model-checking problem must be undecidable.

4.3 Marching on to 3_2-CPDA

In the previous section we saw how PCP could be reduced to model-checking $\boldsymbol{FO(TC)}$ on 2-CPDA. The key part of the reduction involved asserting the existence of a chain of 2-stacks; each successor in the chain required collapse on a certain element to yield the same result as collapse on a particular point in its predecessor. The chain could be described by taking the transitive closure of this first-order definable 'successor' relation. A 3_2-CPDA allows us to record the chain of 2-stacks directly in the 3-stack—the members of the chain are piled on top of each other. Since recording a chain can be done within the model, this removes the burden of transitive closure from the logic, although for 3_2-CPDA we require ϵ-closure. We will thus see that the ϵ-closure of 3_2-CPDA graphs have undecidable first-order theories.

Let us construct a 3_2-CPDA $\mathcal{A}^P_{3_2}$ for each instance of PCP P.

Definition 4.12. The 3_2-CPDA $\mathcal{A}^P_{3_2}$ behaves as follows:

- It begins by behaving in the same way as $\mathcal{A}^P_{2^+}$, performing only 2-stack operations until it reaches the control-state $\boldsymbol{u_1u_2v_1v_2}^{\boldsymbol{start}}$ or $\boldsymbol{v_1v_2u_1u_2}^{\boldsymbol{start}}$. If it is unable to reach such a state, it goes into a distinguished ***fail*** state and aborts.

- The automaton then pushes a record of its $\mathcal{A}^P_{3_2}$ control-state on to the stack and performs $push_3; pop_2; pop_2; pop_2; pop_2;$. This will return the stack to the original $\mathcal{A}^P_{2^+}$ ***guess***-configuration. It then behaves from here as though it were $\mathcal{A}^P_2$ in the ***guess*** control state (with a stack s) until it reaches an s-verifier-configuration (or finds it is unable to, in which case it aborts).

- The previous step is repeated until an $\boldsymbol{end}_s$ configuration is reached, at which point the CPDA pushes the $\mathcal{A}^P_{2^+}$ control-state onto the stack and enters a distinguished $\boldsymbol{guess}_{\mathbf{3_2}}$ control-state.

The following transitions are then added:

- An ϵ-transition going from any configuration to a distinguished control-state ***prototest*** via a pop_3 operation.

- The *only* transition going from a ***prototest*** control-state is allowed when the top_1 stack-symbol is one of: $\boldsymbol{u_1u_2v_1v_2}$, $\boldsymbol{u_1v_1u_2v_2}$, $\boldsymbol{u_1v_1v_2u_2}$, $\boldsymbol{v_1u_1u_2v_2}$, $\boldsymbol{v_1u_1v_2u_2}$, $\boldsymbol{v_1v_2u_1u_2}$ (in particular when the top symbol does not have a ***start*** or ***end*** flag). This transition is given the label ***toTest*** and moves to control-state ***test*** with no stack operation.

- We also allow a ***toTest***-labelled transition from a configuration with control-state $\boldsymbol{guess}_{\mathbf{3_2}}$ to control-state ***test*** whilst not performing any stack operation.

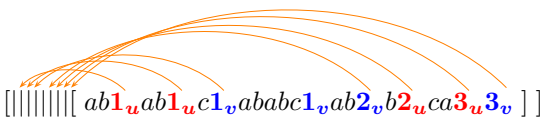
We further add these transitions:

- A transition ***to2***$^+$ performing a pop_1 and entering control-state ***abcd*** whenever in control-state ***test*** with top stack element ***abcd*** where ***abcd*** is a control-state of $\mathcal{A}^P_{2^+}$. From this control-state $\mathcal{A}^P_{3_2}$ behaves in the same way as $\mathcal{A}^P_{2^+}$, performing only order-2 operations.

- From any configuration with ***test*** control-state there exist transitions labelled $\boldsymbol{push}_3$ and $\boldsymbol{pop}_3$ performing the respective stack operations whilst remaining in control-state ***test***.

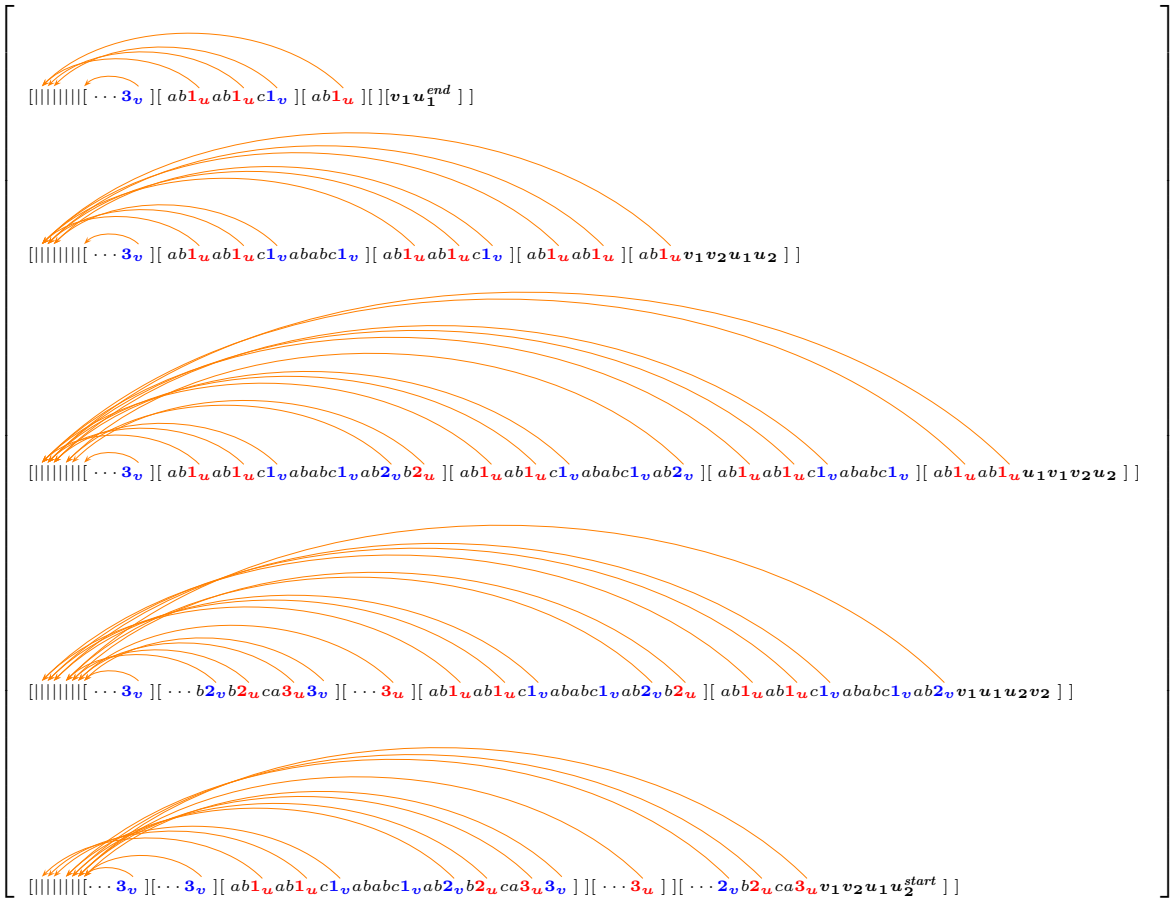
Let us continue with the running example:

Example 4.13. Recall that a solution to the PCP example P from Example 4.1 could be represented by a stack of $\mathcal{A}^P_2$ which we abbreviate to:



(as per Example 4.7). The automaton $\mathcal{A}^P_{3_2}$ extends this 2-stack upwards to form a 3-stack with contents disecting each stage in the i_u and i_v subsequences. This

yields the following 3-stack in control-state $\boldsymbol{guess_{3_2}}$:



A stack in a $\boldsymbol{guess_{3_2}}$ configuration of $\mathcal{A}^P_{3_2}$ fixes not only the 2-stack representing a guess at the solution (as is done by $\mathcal{A}^P_{2+}$) but also a sequence of 2-stacks alleged to witness the correctness of this guess. The construction of $\mathcal{A}^P_{3_2}$ means that just a single $\boldsymbol{toTest}$ transition is needed in the ϵ-closure of its configuration graph to reach any of the 2-stacks in this sequence other than the bottom-most. Thus we are able to achieve with universal quantification over $\mathcal{G}^\epsilon(\mathcal{A}^P_{3_2})$ that for which we needed transitive closure logic over $\mathcal{G}(\mathcal{A}^P_{2+})$.

Lemma 4.14. *There exists a Σ_2 sentence ϕ of **FO** such that for all instances P of Post's Correspondence Problem we have:*

$$\mathcal{G}^\epsilon(\mathcal{A}^P_{3_2}) \vDash \phi \text{ iff } P \text{ has a solution.}$$

Proof. Let us first interpret Lemma 4.10 in the context of $\mathcal{A}^P_{3_2}$. Since $\mathcal{A}^P_{3_2}$ is designed to generate arbitrary verifiers for some stack s created at the outset, beginning with 0_s and ending with $\boldsymbol{end_s}$, Lemma 4.10 implies that there is a solution to P if and only if $\mathcal{A}^P_{3_2}$ can reach a configuration of the form

$(\boldsymbol{guess_{3_2}}, s)$ such that for every pair of 2-stacks t, t' occurring in s with t' immediately above t, $t' = t^+$.

We already have a Π_1 formula $\psi(x, y)$ in first-order logic expressing that $y = x^+$ where x and y range over 2-stacks reachable by $\mathcal{A}^P_{2^+}$. This was given in the proof of Lemma 4.11. Now observe that for any 3-stack s and sequence of 2-*stack* operations $\vec{op}$ we have $\vec{\theta}(s) = \vec{\theta}(pop_3(push_3(s)))$ if and only if $\vec{\theta}(top_3(s)) = \vec{\theta}(top_3(pop_3(s)))$. We can thus exploit the $\mathcal{A}^P_{2^+}$-simulation feature of $\mathcal{A}^P_{3_2}$ to express that some variable x bound to a configuration of $\mathcal{A}_{3_2}$ of the form $(\boldsymbol{test}, s)$ has the property that $top_3(pop_3(s))^+ = top_3(s)$. This can be done by the Π_1 formula $\chi(x)$:

$$\forall y_1 y_2 y. \forall z. (\boldsymbol{pop_3}(x, y_1) \rightarrow \boldsymbol{push_3}(y_1, y_2) \rightarrow \boldsymbol{to2^+}(y_2, y) \rightarrow \boldsymbol{to2^+}(x, z) \rightarrow \psi(y, z))$$

over both the graph $\mathcal{G}(\mathcal{A}^P_{3_2})$ and $\mathcal{G}^\epsilon(\mathcal{A}^P_{3_2})$ ($\mathcal{A}^P_{3_2}$ has no ϵ-transitions reachable from a configuration with control-state $\boldsymbol{test}$).

The construction of $\mathcal{A}^P_{3_2}$ ensures that the configurations we can reach from $(\boldsymbol{guess_{3_2}}, s)$ by a (possibly empty) sequence of ϵ-transitions followed by a $\boldsymbol{toTest}$-transition are precisely those of the form $(\boldsymbol{test}, s')$ such that s' is a prefix of s whose 2-stacks all occur in s. Hence these stacks are precisely those reachable by a $\boldsymbol{toTest}$-transition in the ϵ-closure of the configuration graph. Thus combining all of the observations above the required Σ_2 sentence ϕ is:

$$\phi := \exists x. \forall y (\boldsymbol{guess_{3_2}}(x) \wedge \boldsymbol{toTest}(x, y) \wedge \chi(y) \vee (\boldsymbol{u_1 u_2 v_1 v_2^{start}}(y) \vee \boldsymbol{v_1 v_2 u_1 u_2^{start}}(y)))$$

(disregarding when y is a verifier with the $\boldsymbol{start}$ flag as this is at the bottom and so there is no stack to compare below this). □

4.4 The non-locality of 3_3-CPDA

We now consider 3_3 automata. Adapting $\mathcal{A}^P_{3_2}$ to become a 3_3-CPDA is straightforward—we can simply replace the 2-links with 3-links and replace the initial $push_2$ operations with $push_3$ operations to ensure different targets. In fact using 3-links allows us to simplify the automaton a little bit since we can compare the top elements of two 1-stacks by simply collapsing on each and seeing whether the same 3-stack is obtained from each collapse. When we were using 2-links, it was necessary to do some additional pop_3 and $push_3$ operations. These were to cope with the fact that two 'equal' elements in two different 2-stacks would still yield different results when collapsing on a 2-link since a collapse on a 2-link occurs *within* each separate 2-stack.

Moreover, the undecidability result for 3_3-CPDA is stronger; the non-locality of additional 3-links is exploited to alleviate the need for ϵ-closure. We illustrate this idea of exploiting non-localilty in Figure 4.1. An initial guess at a solution is stored in a 1-stack s with each element in s emanating a 3-link

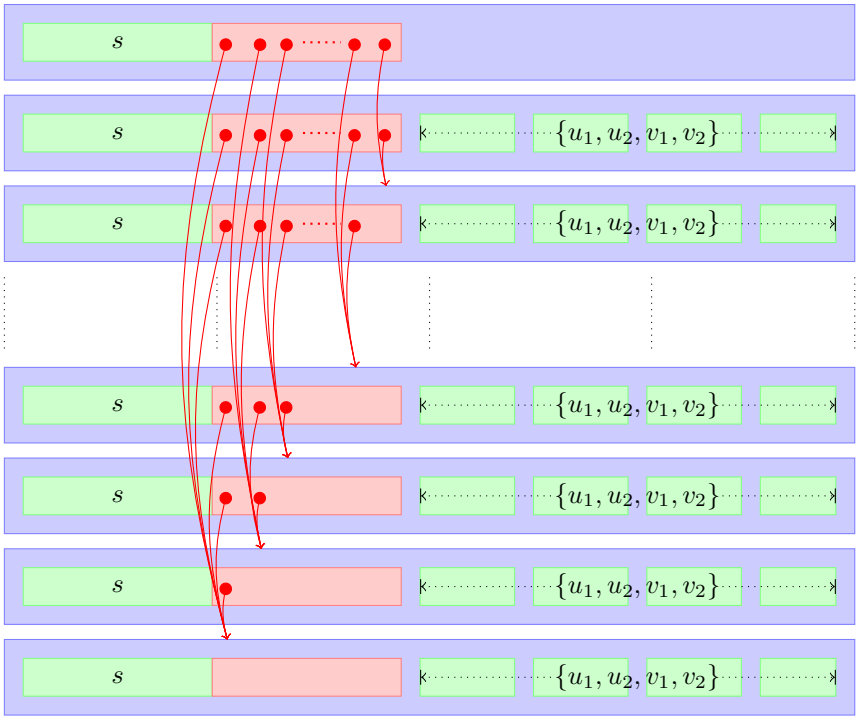


Figure 4.1: Exploiting the non-locality of 3-links

with a distinct target (these targets have been omitted from Figure 4.1). In a manner similar to $\mathcal{A}^P_{3_2}$, we then proceed to produce 2-stacks representing a chain of verifier-configurations. This time, however, we also add a series of 3-links on top of s—these can be discarded each time s is copied to begin the creation of a set of u_1, u_2, v_1, v_2. Proceeding in this way we end up with a top_3 2-stack decked with pointers to each of the 2-stacks in the verifier-chain. Naturally one needs an unbounded number of pop_1 operations in order to reach the source of each of these links, but note that the very top top_3-stack in the figure is uniquely determined by those below it. This means that we can have a universal quantifier ranging over all results of these iterated pop_1 operations by restricting the range of a such a quantifier with a common result of pop_3. This allows access to a link for each position in the chain without ϵ-closure.

Definition 4.15. Let P be an instance of Post's Correspondence Problem. The 3_3-CPDA $\mathcal{A}^P_{3_3}$ has stack alphabet:

$$\Sigma \cup [1_u, 2_u, \ldots, m_u] \cup [1_v, 2_v, \ldots, m_v] \cup \{\boldsymbol{u_1u_2v_1v_2^{start}}, \boldsymbol{v_1v_2u_1u_2^{start}}, \boldsymbol{u_1v_1^{end}}, \boldsymbol{v_1u_1^{end}}\}$$
$$\cup \{\boldsymbol{abcd} \ : \ \{a, b, c, d\} = \{u_1, u_2, v_1, v_2\}\} \cup \{\bullet\}$$

It initially behaves by non-deterministically choosing one of the following:

- Push any member of Σ onto the stack.
- If the Σ symbols in the stack since the last symbol of the form i_u in the top 1-stack (or the bottom of the 1-stack if there is no such symbol) form the word u_j, then it may perform $push_3; push_1^{j_u,3}$.
- If the Σ symbols in the stack since the last symbol of the form i_v in the top 1-stack (or the bottom of the 1-stack if there is no such symbol) form the word v_j, then it may perform $push_3; push_1^{j_v,3}$.

Finite control-states enforce the precondition on the second and third options.

Once the top two elements of the stack are of the form i_u, i_v for some i (in either order), the automaton enters the next phase and generate the first element of the verification chain:

- Perform $push_2; push_2$
- Perform $pop_1; push_2; push_2$
- Either push $\boldsymbol{u_1 u_2 v_1 v_2^{start}}$ or $\boldsymbol{v_1 v_2 u_1 u_2^{start}}$ onto the stack depending on whether the top two elements were in the order $i_u i_v$ or $i_v i_u$.

The automaton then iteratively produces further candidates for elements in the verification chain:

- Perform $push_3$.
- Perform $pop_2; pop_2 pop_2 pop_2$
- Perform $push_1^{\bullet,3}$
- Perform $push_2$ followed by iterated pop_1 until the top element of the stack is no longer a $\bullet$.
- Generate a further four 2-stacks representing a u_1, u_2, v_1, v_2 in exactly the same way as for $\mathcal{A}^P_{3_2}$, recording the ordering (possibly with an end flag) on top.
- The automaton breaks from this iteration at this point iff an element with an ***end*** flag was just deployed.

After this phase the automaton enters a distingished control-state ***candidate***. It then *leaves* this control-state to perform $push_3; pop_2; pop_2; pop_2; pop_2$ and enters control-state ***chainpos***. It then repeatedly performs the following:

- pop_1 entering a non-distinguished control-state.
- If the top element is a $\bullet$ it enters control-state ***chainpos*** and goes back to the item above.

We additionally add transitions from all configurations labelled by $\boldsymbol{pop_1}$, $\boldsymbol{pop_2}$, $\boldsymbol{pop_3}$ and $\boldsymbol{collapse}$ performing the respective stack operations whilst transitioning to a distinguished control-state $\boldsymbol{test}$.

A first-order formula very similar to that in the proof of Lemma 4.14 combined with the capacity to replace ϵ-closure with collapsing on 3-links gives:

Lemma 4.16. *Let P be an instance of Post's Correspondence Problem. Then there exists a Σ_2 sentence ϕ of **FO** such that:*

$$\mathcal{G}(\mathcal{A}^P_{3_3}) \vDash \phi$$

(note no *ϵ-closure) if and only if P has a solution.*

Proof. For the same reasons as with $\mathcal{A}^P_{3_2}$ it is the case that P has a solution if and only if $\mathcal{A}^P_{3_3}$ can reach a configuration $(\boldsymbol{candidate}, t)$ such that, for every pair of 2-stacks s, s' occurring in t with s' is the 2-stack immediately above s, it is the case that $s' = s^+$.

Assuming that the variable x is bound to a configuration $(\boldsymbol{test}, t')$ where t' is an initial segment of a stack t such that $(t, \boldsymbol{candidate})$ is reachable, we can assert that the top two 2-stacks of t' form a pair with the credentials above using the following Π_1 formula $\psi(x)$ over $\mathcal{G}(\mathcal{A}_{3_3})$:

$$\begin{aligned}\forall s'.\forall u_2 v_2 u_1 v_1.\forall w w'.\forall c_u c_v.(&\boldsymbol{pop_3}(x, s') \to \boldsymbol{u_1 u_2 v_1 v_2}(x) \to \boldsymbol{u_1 u_2 v_1 v_2}(s') \\ &\to \boldsymbol{pop_2}(x, v_1) \to \boldsymbol{pop_2}(v_1, w) \to \boldsymbol{pop_2}(w, u_1) \to \boldsymbol{pop_1}(s', v_2) \\ &\to \boldsymbol{pop_2}(s', w') \to \boldsymbol{pop_2}(w', u_2) \to \boldsymbol{collapse}(u_1, c_u) \to \boldsymbol{collapse}(v_1, c_v) \\ &\to (\boldsymbol{collapse}(u_2, c_u) \wedge \boldsymbol{collapse}(v_2, c_v))) \\ &\wedge \cdots\end{aligned}$$

where as with the proof of Lemma 4.11 we have illustrated just one of 64 conjuncts that go through all possible orderings of the u_1, u_2, v_1, v_2 in the top and penultimate 2-stacks. The correctness of this formula follows from the fact that every element constituting the guess of a solution of P is equipped with a 3-link pointing to a distinct 2-stack.

The design of $\mathcal{A}^P_{3_3}$ ensures that the elements in the verification-chain contained in t (*i.e.* all candidates for the pairing s, s' in t) are precisely those that can be reached by performing a *collapse* operation on a $(\boldsymbol{chainpos}, t')$ configuration reached from $(\boldsymbol{candidate}, t)$. Assuming that x is again bound to a $(\boldsymbol{candidate}, t)$ configuration, we can thus assert that x represents a solution to P with the following Π_1 formula $\chi(x)$ over $\mathcal{G}(\mathcal{A}^P_{3_3})$:

$$\begin{aligned}\chi(x) := \forall y.\forall z.(&\boldsymbol{chainpos}(y) \to \boldsymbol{pop_3}(y, x) \to \boldsymbol{collapse}(y, z) \\ &\to \neg(\boldsymbol{u_1 u_2 v_1 v_2}^{start}(z) \vee \boldsymbol{v_1 v_2 u_1 u_2}^{start}(z)) \to \psi(z))\end{aligned}$$

We can thus take the required Σ_2 formula ϕ (asserting the existence of such a solution witnessing configuration x) to be:

$$\phi := \exists x.(\textbf{\textit{candidate}}(x) \wedge \chi(x))$$

□

4.5 The Power of 5_2 and 5_3-CPDA

The automata $\mathcal{A}^P_{3_2}$ and $\mathcal{A}^P_{3_3}$ both work by generating a 'verification chain' of 2-stacks. First-order logic is used to express that for each member s of the chain the results of collapsing in two different positions, called A_s and B_s say, yield the same result. With an order-5 stack it is possible to do either the A_s collapse or B_s collapse at every point s in such a way that the final stack retains a record of each collapse. Moreover, we can give the CPDA two modes—one by which it generates such a stack using A collapses and the other by which it generates a stack using B collapses. There is a Σ_1 sentence asserting that it one can generate the same stack by both of these methods, which implies that the A_s collapse and B_s collapse yield the same result for *every* s.

Thus a Σ_1 sentence over the configuration graph of a 5_2-CPDA or a 5_3-CPDA is sufficient to assert the existence of a solution to an instance of Post's Correspondence Problem. This gives us an incredibly strong undecidability result—for such graphs even model-checking first-order sentence without any quantifier alternation at all is undecidable. Both these results will actually be implied by an analalgous result for 4_2-CPDA in the next section. However, the construction used there will be more fiddly whilst in the order-5 case we can resuse the same basic idea that we have used previously.

Figure 4.2 demonstrates the idea behind the 5_2-CPDA $\mathcal{A}^P_{5_2}$. The regions labelled s consist of multiple 1-stacks encoding a guess of the solution to the PCP instance P in the same manner as $\mathcal{A}^P_2$. If the 5-stack A is equal to the 5-stack B then the guessed solution is indeed correct.

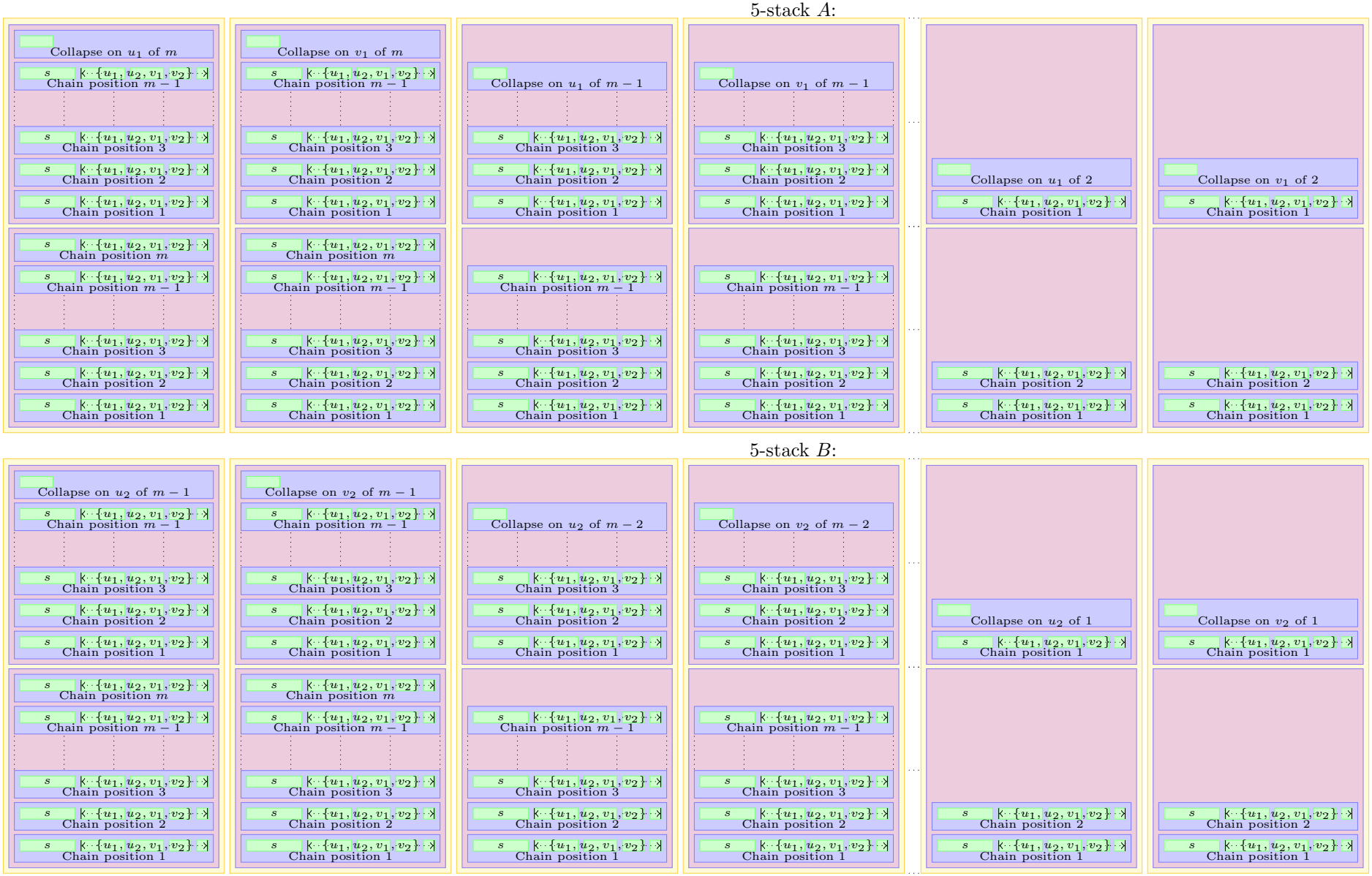


Figure 4.2: The idea behind $\mathcal{A}^P_{5_2}$

Definition 4.17. The automaton $\mathcal{A}^P_{5_2}$ is a 5_2-CPDA that initially behaves in the same manner as $\mathcal{A}^P_{3_2}$ with the same stack alphabet until it reaches the $\boldsymbol{guess_{3_2}}$ control-state. It then either adopts *mode A* or *mode B*. In mode A it behaves as follows:

- Perform a $push_4$ operation.
- Navigate to the u_1 1-stack in the top_3 stack. The position of this stack is indicated by the $\boldsymbol{abcd}$ indicator on top of the top_3 stack (where $a, b, c, d \in \{u_1, u_2, v_1v_2\}$) as is the case with $\mathcal{A}^P_{3_2}$. Navigating to the u_1 stack requires at least one and up to three pop_2 operations.
- A *collapse* operation is performed on the element on top of the u_1 1-stack in the top_3 stack.
- A $push_5$ operation is performed followed by a $pop_4; push_4$.
- Navigate to the v_1 1-stack in the top_3 stack.
- Perform *collapse* on the element on top of the v_1 1-stack in the top_3 stack.
- Perform a $push_5$ operation followed by a pop_4 operation.
- We now manipulate the next element in the verification chain by performing a pop_3 operation and going back to the top of this list of operations.
- The iteration stops as soon as the u_1 and v_1 collapses have been performed on the 3-stack second in the verification chain—*i.e.* second from the bottom of the 4-stack. When this point is reached, we enter distinguished control-state $\boldsymbol{A}$.

If the automaton adopts mode B then it instead behaves as follows:

- Perform a $push_4$ operation.
- Perform a $pop_3; push_3$ operation so now the top_3 stack is a copy of the second stack in the verification chain.
- Navigate to the u_2 1-stack in the top_3 stack. The position of this stack is indicated by the $\boldsymbol{abcd}$ indicator on top of the top_3 stack (where $a, b, c, d \in \{u_1, u_2, v_1v_2\}$) as is the case with $\mathcal{A}^P_{3_2}$. Navigating to the u_2 stack requires up to two pop_2 operations. If the u_2 stack *is* the top_2 stack then a pop_1 needs to be performed to discard the state-decoration provided by $\mathcal{A}^P_{3_2}$.
- A *collapse* operation is performed on the element on top of the u_1 1-stack in the top_3 stack.
- A $push_5$ operation is performed followed by a $pop_4; push_4$.

- Navigate to the v_1 1-stack in the top_3 stack.
- Perform *collapse* on the element on top of the v_1 1-stack in the top_3 stack.
- Perform a $push_5$ operation followed by a pop_4 operation.
- Perform a $pop_3; push_3$ operation.
- Navigate to the v_2 1-stack in the top_3 stack, performing a pop_1 operation to discard the state-decoration if it is the top_2 stack.
- Perform *collapse* on the top element of the v_2 1-stack.
- We now manipulate the next element in the verification chain by performing a pop_3 operation and going back to the top of this list of operations.
- The iteration stops as soon as the u_2 and v_2 collapses have been performed on a copy of the 3-stack bottom in the verification chain—*i.e.* bottom of the 4-stack. This will occur when the bottom 4-stack in the top_5 stack still contains two 3-stacks. When this point is reached, we enter distinguished control-state $\boldsymbol{B}$.

We additionally add a transition labelled ***toCompare*** from both $\boldsymbol{A}$ and $\boldsymbol{B}$ to a distinguished control-state ***compare*** without performing any stack operations.

Lemma 4.18. *Let P be an instance of Post's Correspondence Problem. There exists a Σ_1 sentence ϕ such that $\mathcal{G}(\mathcal{A}^P_{5_2}) \vDash \phi$ if and only if P has a solution.*

Proof. We claim that the required ϕ is simply:

$$\phi := \exists xyz.(\boldsymbol{A}(x) \wedge \boldsymbol{B}(y) \wedge \boldsymbol{toCompare}(x,z) \wedge \boldsymbol{toCompare}(y,z))$$

This is satisfied by $\mathcal{G}(\mathcal{A}^P_{5_2})$ just in case $\mathcal{A}^P_{3_2}$ can generate an identical stack in both A mode and B mode. We now argue that this coincides precisely with when P has a solution.

P has a solution iff a verification chain exists in the sense of Lemma 4.10. Referring to the proof of Lemma 4.14 such a chain exists just in case $\mathcal{A}_{3_2}$ is able to produce a 3-stack t such that for every initial segment of t consisting of an integral number of 2-stacks all belonging to t and containing at least two 2-stacks we have $top_3(t') = top_3(pop_3(t'))^+$. As observed in the proof of Lemma 4.14, this is the case iff both collapsing on u_1 of $top_3(t')$ yields the same result as collapsing on u_2 of $top_3(push_3(pop_3(t')))$ and collapsing on v_1 of $top_3(t')$ yields the same result as collapsing on v_2 of $top_3(push_3(pop_3(t')))$.

Consider a 3-stack t such that $\mathcal{A}_{3_2}$ can reach a configuration $(\boldsymbol{guess_{3_2}}, t)$. Suppose that this contains m 2-stacks. In mode A $\mathcal{A}_{5_2}$ will go on to reach a configuration $(\boldsymbol{A}, t^A)$ where t^A contains $2(m-1)$ 4-stacks $t^A_1, t^A_2, \ldots, t^A_{2(m-1)}$.

In mode B it will go on to reach a configuration $(\boldsymbol{B}, t^B)$ where t^B contains $2(m-1)$ 4-stacks $t^B_1, t^B_2, \ldots, t^B_{2(m-1)}$.

Observe that the automaton ensures that $pop_4(t^A_i) = pop_4(t^B_i)$ for every $1 \leq i \leq 2(m-1)$ since the manipulation of the bottom most 4-stack in each 5-stack is the same for both the A mode and the B mode. Let t_i be the initial segment of t consisting of the bottom-most i 2-stacks of t. The automaton also ensures that the for each $0 \leq j \leq (m-2)$ the $top_4(t^A_{2j+1})$ is the result of collapsing on the u_1 of the $top_3(t_i)$ and $top_4(t^B_{2j+1})$ is the result of collapsing on the u_2 of the $top_3(push_3(pop_3(t_i)))$. Likewise for $1 \leq j \leq (m-1)$ the $top_4(t^A_{2j})$ is the result of collapsing on the v_1 of the $top_3(t_i)$ and $top_4(t^B_{2j})$ is the result of collapsing on the v_2 of the $top_3(push_3(pop_3(t_i)))$.

We may thus conclude that $t^A = t^B$ if and only if t contains a valid verification chain witnessing a solution to P.

If P does have a solution, then we already know that $\mathcal{A}_{3_2}$ can produce such a stack t witnessing a solution to P. Hence $\mathcal{A}_{5_2}$ can reach both $(\boldsymbol{A}, t^A)$ and $(\boldsymbol{B}, t^B)$ where $t^A = t^B$, as required.

Conversely suppose that $(\boldsymbol{A}, s)$ and $(\boldsymbol{B}, s)$ are both reachable by $\mathcal{A}_{5_2}$. There must then exist stacks t and t' of $\mathcal{A}_{3_2}$ such that $s = t^A$ and $s = t'^B$. The automaton will ensure that $pop_4(t^A_1) = t$ and $pop_4(t'^A_1) = t'$. Since $t^A = t'^B$ we must thus have $t^A_1 = t'^B_1$ and so $t = t'$. Thus there does exist a reachable configuration $(\boldsymbol{guess_{3_2}}, t)$ such that $t^A = t^B (= s)$. Hence P has a solution. □

A 5_3-CPDA $\mathcal{A}^P_{5_3}$ can be constructed for each instance P of the PCP along exactly the same lines as $\mathcal{A}^P_{5_2}$. We can just employ 3-links rather than 2-links to keep track of different stack elements. The construction still works since the act of collapsing on a 3-link is still contained within a 4-stack and so the rest of the automaton can behave in the same way.

Again this result is subsumed by that in the next section, although we require a more fiddly notion of verification chain for this to work.

4.6 Fiddling with 4_2-CPDA to attack Σ_1

The need to modify the verification chain

In an earlier section we showed that solving an instance Post's Correspondence Problem P could be reduced to model-checking a Σ_2 sentence ϕ on $\mathcal{G}^\epsilon(\mathcal{A}_{3_2})$. This sentence ϕ is of the form: 'there exists a 3-stack s such that for all 2-stacks in $s\ldots$' where 'for all 2-stacks in s' is implemented by exploiting ϵ-closure in allowing ϕ to reference an arbitrary number of pop_3's as a single edge.

The same idea can be straightforwardly adapted to 4_2-CPDA removing the need for ϵ-closure. After generating the proposed 3-stack s, a 4_2-CPDA could perform a $push_4$ operation followed by an arbitrary number of pop_3 operations

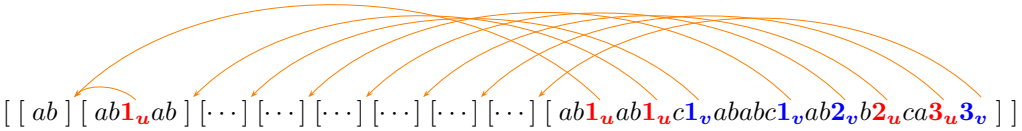
terminating in a control-state ***guess***. The assertion 'there exists a 3-stack s such that all 2-stacks in s behave like X' could then be expressed over the transition graph of this 4_2-CPDA (*without* ϵ-closure) as 'there exists a 4-stack s such that for all configurations t with control-state ***guess*** where $pop_4(t) = s$, the top 2-stack of t behaves like X'.

However, in this section we focus on a stronger result— there exists a Σ_1 sentence ϕ such that we can construct a 4_2-CPDA $\mathcal{A}^P_{4_2}$ for each instance of the PCP P $\mathcal{G}(\mathcal{A}^P_{4_2}) \vDash \phi$ if and only if P has a solution. This will use a very similar idea to the previous section and indeed will imply the results in the previous section. However, the 5_2-CPDA $\mathcal{A}_{5_2}$ very much depends on being order-5 in order to carry out its job correctly. To understand why this is the case, consider Figure 4.3 which shows initial segments of hypothetical 'A-mode' and 'B-mode' 4-stacks. Since we need to compare u_1 and v_1 of the mth element in the verification chain with u_2 and v_2 respectively of the $(m-1)$th element, we need two copies of of the mth element in the verification chain in A-mode. This could be implemented using a $push_3$ operation, as is suggested in Figure 4.3, but this presents a problem when constructing the B-mode stack on the right of Figure 4.3. The 2-stacks highlighted in red need to be a copy of the mth position in the chain to provide a chance of equality with the A-mode stack, but we also need a copy of $(m-1)$ just above it, which would require it to be the $(m-1)$th position in the chain.

The modification

We modify the verification chain so that only one pair of collapses needs to be compared for equality rather than previously where two comparisons were required (for the u and the v). This allows us to avoid the problem above.

Proceeding with our running example, a solution to a PCP instance will be represented using a 2-stack in exactly the same manner as before:



The difference is the manner in which the 'verification chain' used to check the correctness of an alleged representation is constructed and represented in a 3-stack. In this modified chain of 2-stacks it is the top three 1-stacks that are significant. Each element in the stack will posess either 'u v_1 and v_2' 1-stacks (in some order) or v, u_1 and u_2 1-stacks. This contrasts with the original verification chaion where each point posessed a u_1, u_2, v_1 and v_2 1-stack.

We refer to the u or v 1-stack in a chain position as *guaranteed* and the u_2 or v_2 stack as *tentative* with the associated u_1 or v_1 as the *condition stack*. If the condition stack represents the same position in either the u or v subsequence

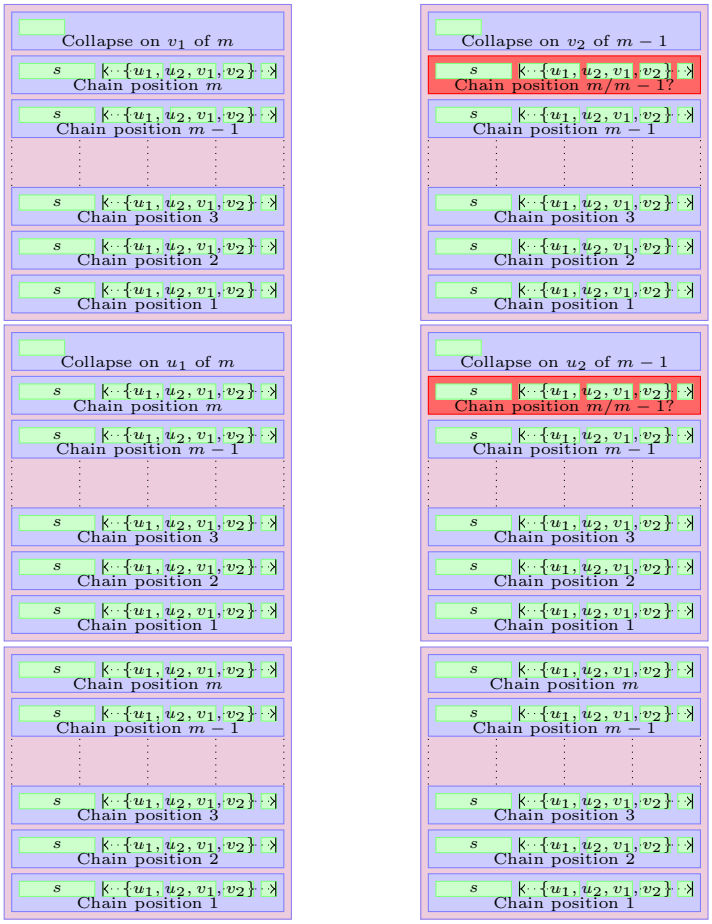
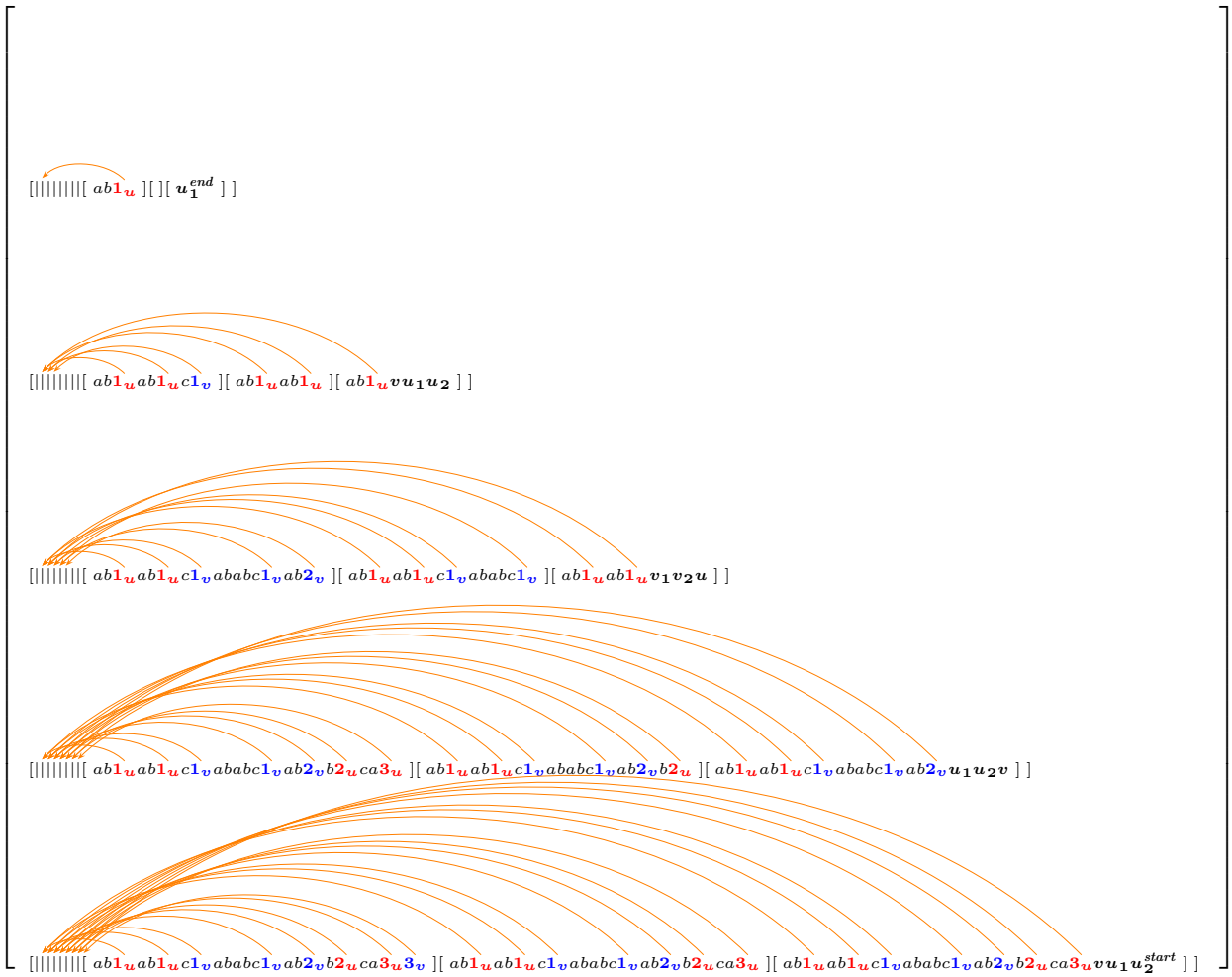


Figure 4.3: Why $\mathcal{A}^P_{5_2}$ cannot 'directly' work at order-4

from the previous member in the chain, then the tentative stack correctly represents the next position in the u or v subsequence. This is the same idea as before. The guaranteed stack, however, will always *unconditionally* represent the next position in the v or u subsequence.

The following shows this new kind of a verification chain as a 3-stack:



As before, a token describing the ordering of the stacks is added to the top of each element in the chain.

Let us formally define a 3_2-CPDA that generates verification chains in the manner illustrated in the example above.

Definition 4.19. Let P be an instance of PCP. The 3_2-CPDA $\mathcal{A}^P_{3_2^{alt}}$ shares stack alphabet with $\mathcal{A}^P_2$ but with extra symbols: $\boldsymbol{vu_1u_2}$, $\boldsymbol{u_1vu_2}$, $\boldsymbol{u_1u_2v}$, $\boldsymbol{vu_1u_2^{start}}$, $\boldsymbol{u_1^{end}}$, $\boldsymbol{uv_1v_2}$, $\boldsymbol{v_1uv_2}$, $\boldsymbol{v_1v_2u}$, $\boldsymbol{uv_1v_2^{start}}$, $\boldsymbol{v_1^{end}}$.

$\mathcal{A}^P_{3_2^{alt}}$ initially behaves the same way as $\mathcal{A}^P_2$ in order to generate a 2-stack representing a postulated solution to P. At this point the top two elements of the stack will either be i_ui_v or i_vi_u for some $i \in [1..m]$. We then perform $push_2; push_2; pop_1; push_2$ followed by $push_1^{\boldsymbol{uv_1v_2^{start}}}$ if the top two symbols are i_vi_u and $push_1^{\boldsymbol{vu_1u_2^{start}}}$ if i_ui_v. The automaton then performs $push_3$ and behaves as follows, examining the token on top of the stack to ascertain which option should be taken:

- If the current guaranteed stack of the top_3 element (which has just been copied) is below both the condition and tentative stacks, then we should perform pop_2 until the guaranteed stack is the top_2 stack—*i.e.* perform $pop_2; pop_2$. Then:
  - If the guaranteed stack is a u stack (rather than a v stack) then perform pop_1 until another j_u or j_v is found for some $j \in [1..m]$.
    - At the *first new* j_u to be discovered, we deem the resulting top_2 to be the new u stack. If v_1 and v_2 are still to be produced we perform $push_2$ and continue with the pop_1s.
    - At all *subsequent* j_u discovered, we just perform pop_1 and continue with the pop_1s.
    - If a j_v is found and we have not yet created a new v_1, then we non-deterministically choose whether to deem this stack to be the new v_1. If we choose not to we just proceed with more pop_1s. If we choose to do so, then we perform $push_2$.
    - The *first new* j_v to be found *since creating* v_1 should become the new v_2 stack. If the new u stack has not yet been created then we perform $push_2$ and continue with the popping.
    - All *subsequent* j_v found *since creating* the new v_2 should just be popped and the pop_1s continued.
    - If the empty 1-stack is produced, then abort (and fail) if either u_1 has not yet been generated or either u_2 or v have been generated. Otherwise perform $push_2; push_1^{\boldsymbol{u_1^{end}}}$.
    - If the new v, u_1 and u_2 have all been created, then cease the pop_1s and perform $push_1^{\boldsymbol{token}}$ where ***token*** is the token (without a start or end flag) denoting the order in which the stacks were produced.
  - If the guaranteed stack is a v stack (rather than a u stack) then do the same as above interchanging u and v.
- If the condition stack of the top_3 element in the chain is below both the guaranteed and tentative stacks, then we should perform pop_2 until the condition stack is the top_2 stack—*i.e.* perform $pop_2; pop_2$. Then:
  - If the condition stack is a u_1 stack (rather than a v_1 stack) then perform pop_1 until another j_u or j_v is found for some $j \in [1..m]$.
    - At the *first new* j_u to be discovered we just perform pop_1 and continue with the pop_1 operations. At the *second new* j_u to be discovered, we deem the current top_2 stack to be the new u stack. If the new v_1 and v_2 have not yet been created we perform $push_2$ and continue with the pop_1 operations.

  - At all *subsequent* j_u discovered, we just perform pop_1 and continue with the pop_1s.
  - If a j_v is discovered then we behave in the same way as in the case when the guaranteed stack u is below the condition stack v_1 (see above). This creates the v_1 and v_2 stacks.
  - If an empty 1-stack is produced or we have just finished creating all of the new u, v_1 and v_2 then again we behave in the same manner as when the guaranteed stack is below the condition stack.

- Note that the condition stack will always be below the tentative stack, so we have already exhausted the possibilities.

- If it is not the case that $top_1(u) = i_u$ and $top_1(v_2) = i_v$ or $top_1(u_2) = i_u$ and $top_1(v) = i_v$ (depending on which combination of stacks the recently produced chain element uses) for some $i \in [1..m]$ then the automaton aborts into a fail state.

- If we have deployed a token with an ***end*** marker, then we halt and move into a distinguished control-state $\boldsymbol{guess_{3_2}}$. Otherwise we perform a $push_3$ operation and repeat.

Remark 4.20. Suppose that an element in a chain generated by $\mathcal{A}^P_{3_2^{alt}}$ consists of a v and u_1, u_2 stacks. If v is the first (lowest) of these, then the next element in the chain will consist of a v, u_1 and u_2. If u_1 is the lowest, then the next element in the chain will consist of a u, v_1 and v_2.

We formalise what it means to be a correct verification chain in this new style using a revised successor operation $\oplus$ to be to $\mathcal{A}^P_{3_2^{alt}}$ what $+$ is to $\mathcal{A}^P_{3_2}$.

Definition 4.21. Let s be a 2-stack over the alphabet of $\mathcal{A}^P_2$. The *successor* $s^\oplus$ of s is the unique stack such that:

- $pop_2; pop_2; pop_2(s^\oplus) = pop_2; pop_2; pop_2(s)$
- Where a, b, c are (in order) the top three 1-stacks of $s^\oplus$: $c \sqsubseteq_1 b \sqsubseteq_1 a$.
- If the bottom of the top three 1-stacks in s is a u or u_1 stack, then $s^\oplus$ should consist of a u, v_1 and v_2 stack, if its a v or v_1 stack then, $s^\oplus$ should consist of a v, u_1 and u_2 stack.
- Let s_u be either the u_2 or u stack in s (whichever s posesses). Let s_v be either the v_2 or v stack in s (whichever s posesses).
  - If $s^\oplus$ posesses u_1, then this $u_1 = s_u$ and its u_2 has as its top_1 element the highest element of the form i_u below $top_1(s_u)$. If such an element does not exist, u_2 should be empty. Moreover the v of

$s^{\oplus}$ should have as its top_1 element the highest element of the form i_v below $top_1(s_v)$. If such an element does not exist, v should be empty.

– If $s^{\oplus}$ posesses v_1, then this $v_1 = s_v$ and its v_2 has as its top_1 element the highest element of the form i_v below $top_1(s_v)$. If such an element does not exist,v_2 should be empty. Moreover the u of $s^{\oplus}$ should have as its top_1 element the highest element of the form i_u below $top_1(s_u)$. If such an element does not exist, u should be empty.

The following Lemma is almost an immediate consequence of the definitions and Lemma 4.6.

Lemma 4.22. *Let P be an instance of Post's Correspondence Problem and let s be a 2-stack generated by $\mathcal{A}_2^P$. The 2-stack s represents a solution to P in the sense of Lemma 4.6 iff there is a 'verification chain' of 2-stacks $s_1, s_2, \ldots, s_k$ for some k such that:*

- *$s_1 = push_2; push_2; pop_1; push_2(s)$ (with top three stacks defined to be v, u_1, u_2 or u, v_1, v_2 if the top two elements of s are respectively of the form $i_v i_u$ or $i_u i_v$)*
- *s_k has empty stacks as its top two 1-stacks*
- *$s_{i+1} = s_i^{\oplus}$ for every $1 \leq i < k$.*
- *For each element s_i in the chain we have $top_1(v) = j_v$ and $top_1(u_2) = j_u$ or $top_1(u) = j_u$ and $top_1(v_2) = j_v$ for some $j \in [1..m]$ (depending on what selection of stacks s_i has).*

Proof. We establish the result by arguing that such a sequence exists iff s satisfies the conditions set out in Lemma 4.6.

Argue by induction on l that an initial segment of such a chain $s_1, s_2, \ldots, s_l$ exists with $j_1, j_2, \ldots, j_l$ the associated indices mentioned in the fourth condition iff the top-most l elements of s of the form i_u are $(j_1)_u, (j_2)_u, \ldots, (j_l)_u$, and the topmost l elements of s of the form i_v are $(j_1)_v, (j_2)_v, \ldots, (j_l)_v$ (ordered top down).

First the $\Rightarrow$) direction. The base case ($l = 1$) is immediate since all stacks generated by $\mathcal{A}_2^P$ must have top two elements of the form $j_u j_v$ or $j_v j_u$ for some j. Suppose now that an initial segment of such a chain $s_1, s_2, \ldots, s_l, s_{l+1}$ exists with fourth-condition indices $j_1, j_2, \ldots, j_l, j_{l+1}$. By the induction hypothesis, we just need to check that if the top elements of either v or v_2 in s_l and either u or u_2 in s_l are respectively the lth j_v and j_u elements from the top of s, then the top elements of either v or v_2 in $(s_l)^{\oplus}$ and either u or u_2 in $(s_l)^{\oplus}$ are respectively the $(l+1)$th elements of the form j_v and j_u from the top of s. But this is ensured directly by the fourth point in the definition of $^{\oplus}$.

Now consider the $\Leftarrow$) direction. Again the base case ($l = 1$) is straightforward since s_1 is explicitly defined to meet the criteria. Suppose that the topmost $l + 1$ elements of the form j_u of s are: $(j_1)_u, (j_2)_u, \ldots, (j_l)_u, (j_{l+1})_u$ and the topmost $l+1$ elements of the form j_v of s are: $(j_1)_v, (j_2)_v, \ldots, (j_l)_v, (j_{l+1})_v$. By the induction hypothesis we already have a chain $s_1, s_2, \ldots, s_l$ and so we just need to show that $(s_l)^{\oplus}$ both exists and satisfies the requisite criteria. If it does exist, then as above the fourth clause in the definition of $^{\oplus}$ ensures that the fourth requirement of the Lemma is satisfied, which is the only one that would need to be established (the first applies only to s_1, the second is not relevant to initial prefixes and the third is by assumption). Thus it is only existence that we need to establish. But the $^{\oplus}$ successor always exists. The top three stacks are all initial segments of s (exhaustively defined—they are defined to be the empty stack if an appropriate element in s does not exist) and so can be linearly ordered with resepct to $\sqsubseteq_1$. □

The following Lemma is critical—it tells us that $\mathcal{A}^P_{3^{alt}_2}$ is able to construct an appropriate chain of successors if one exists and moreover provides a sufficient (and necessary) condition for a stack it generates representing such a chain.

Lemma 4.23. *Let P be an instance of Post's Correspondence Problem. For any 2-stack s generated by $\mathcal{A}^P_2$ there exists a sequence of 2-stacks $s_1, s_2, \ldots, s_k$ satisfying the condition in Lemma 4.22 iff $\mathcal{A}^P_{3^{alt}_2}$ can reach a configuration $(\boldsymbol{guess}_{\mathbf{3_2}}, [s'_1, s'_2, \ldots, s'_k])$ such that:*

- *$s'_i = push_1^{\boldsymbol{token}}(s_i)$ for each $1 \leq i \leq k$ where **token** is a token indicating the ordering of the top three stacks with a **start** flag for s'_1 and a **end** flag for s'_k.*
- *For every $1 \leq i < k$:*
  - *If s'_{i+1} contains a u_1 then this is equal to u_2 or u in s'_i (depending on which one s'_i contains)*
  - *or if s'_{i+1} contains a v_1 then this is equal to v_2 or v in s'_i (depending on which one s'_i contains).*

  Note that for each i only one of the above will hold.

Proof. First let us argue in the $\Rightarrow$) direction. First argue by induction on l that if we have an initial prefix $s_1, s_2, \ldots, s_l$ of such a sequence $s_1, s_2, \ldots, s_k$, then $\mathcal{A}^P_{3^{alt}_2}$ can generate a stack $[s'_1, s'_2, \ldots, s'_l]$ without aborting, during the phase after generating the 2-stack s, such that:

- $s'_i = push_1^{\boldsymbol{token}}(s_i)$ for each $1 \leq i \leq l$ where **token** is a token indicating the ordering of the top three stacks with a **start** flag for s'_1 and a **end** flag for s'_k.

- For every $1 \leq i < l$:
  - If s'_{i+1} contains a u_1 then this is equal to u_2 or u in s'_i (depending on which one s'_i contains)
  - or if s'_{i+1} contains a v_1 then this is equal to v_2 or v in s'_i (depending on which one s'_i contains).

  Note that for each i only one of the above will hold.

For the base case ($l = 1$) the result is immediate as $\mathcal{A}^P_{3^{alt}_2}$ will construct s'_1 from s in exactly the way that s_1 is defined in Lemma 4.22 (adding a token on top).

For the inductive step, suppose that $\mathcal{A}^P_{3^{alt}_2}$ has already generated $[s'_1, s'_2, \ldots, s'_l]$ corresponding to a sequence $s_1, s_2, \ldots, s_l$. Suppose now that this sequence extends to $s_1, s_2, \ldots, s_l, s_{l+1}$. The automaton will perform $push_3; pop_2 pop_2$ in preparation to generate s'_{l+1}. We consider two cases:

Case when the guaranteed stack of s'_l (or by induction hypothesis equivalently s_l) is below both the condition and tentative stacks: W.l.o.g. assume that the guaranteed stack of s'_l is a u stack (the case when it is a v stack is similar). Due to the position of v we must have in s_l: $v \sqsupseteq_1 u_1 \sqsupseteq_1 u_2$. Thus the u_1 of $s_{l+1} = s_l^{\oplus}$, which is the u_1 of s_l can indeed be produced by performing pop_1 operations on the v of s_l (s'_l). The automaton is free to pick anything to be the u_1 of s_{l+1} and so in particular can choose the correct value. The new v of s_{l+1} is restricted to be correct with respect to the v of s_l and similarly the restriction on u_2 is precisely what is required with respect to u_1 in s_{l+1}.

Case when the condition stack of s'_l (or by induction hypothesis equivalently s_l) is below both the guaranteed and tentative stacks: Again w.l.o.g. assume that the condition stack of s'_l is a u_1 stack (the case when it is a v_1 stack is similar). Due to the position of u_1 we must have in s_l: $u_1 \sqsupseteq_1 v$ and $u_1 \sqsupseteq_1 u_2$. Since the new v_1 in s_{l+1} should be equal to v in s_l and the new u in s_{l+1} an initial prefix of u_2 in s_l, it must be possible to form both of these from performing pop_1 operations on the u_1 in s_l. The automaton is unconstrained in picking the v_1, so in particular it is able to guess the correct position—again the constraint on picking the new v_2 relative to v_1 is precisely the correct one. Note also that the constraint on picking the new u relative to the old u_1 is precisely the correct one—popping to the next j_u will yield the old u_2 and so popping from that to the second j_u will yield the correct new u.

Either way since the sequence $s_1, s_2, \ldots, s_l, s_{l+1}$ by assumption exhibits equality of the top elements of the u_2 and v or v_2 and u in each element in the chain, it will carry out the above without aborting. Furthermore the second condition on the s'_i is satisfied since the s_i form a $^{\oplus}$-successor chain.

This establishes the induction hypothesis is true for all $l \leq k$. In particular when $l = k$ the top two 1-stacks of s_k (or s'_k ignoring the token on top) will be empty and so the automaton will halt in control-state $\boldsymbol{guess}_{\mathbf{3_2}}$ as required.

Now let us consider the $\Leftarrow$) direction. We argue by induction on the converse hypothesis to what we had before. Suppose that $\mathcal{A}^P_{3_2^{alt}}$ generates a stack $[s'_1\ s'_2\ \cdots\ s'_l]$ (in the phase following the construction of the 2-stack s) that corresponds to a correct initial segment of a verification chain $s_1, s_2, \ldots, s_l$. Suppose now that $\mathcal{A}^P_{3_2^{alt}}$ proceeds to generate $[s'_1\ s'_2\ \cdots\ s'_l\ s'_{l+1}]$ that satisfies the conditions in the converse of the induction hypothesis used previously. This stack must have been produced from $[s'_1\ s'_2\ \cdots\ s'_l]$ by beginning with a $push_3; pop_2; pop_2$. Again we should consider the same two cases as before, noting that the additional assumed constraint on s'_{i+1} relative to s'_i ensures that the guessed new u_1/v_1 for s'_{l+1} must indeed be the u_1/v_1 for $s_l^{\oplus}$ (*i.e.* the u_2/v_2 of s'_l or equivalently s_l). The new u_2/v_2 and v/u will be correctly created by the automaton, as discussed before when arguing in the $\Rightarrow$ direction.

The fact that the automaton does not fail means that it must have successfully found that the u/u_2 and v_2/v stacks in s_{l+1} share top_1 elements. Thus $s_{l+1} = s_l^{\oplus}$ as required and also satisfy the top_1 equality requirement.

The automaton will only halt in control-state $\boldsymbol{guess_{3_2}}$ if it detects two empty 1-stacks (modulo the token), constituting the final s_k in the chain. □

Exploiting $\mathcal{A}^P_{3_2^{alt}}$ in a 4-CPDA

Since only one comparison needs to be made between adjacent elements, the problem illustrated in Figure 4.3 is no longer an issue. The same idea that took us from $\mathcal{A}^P_{3_2}$ to $\mathcal{A}^P_{5_2}$ can thus be used to go from $\mathcal{A}^P_{3_2^{alt}}$ to a 4_2-CPDA $\mathcal{A}^P_{4_2}$.

Definition 4.24. Let P be an instance of Post's Correspondence Problem. The 4_2-CPDA $\mathcal{A}^P_{4_2}$ shares the same stack-alphabet as $\mathcal{A}^P_{3_2^{alt}}$. It begins by behaving as $\mathcal{A}^P_{3_2^{alt}}$ until this automaton halts in control-state $\boldsymbol{guess_{3_2}}$. It then performs a $push_4$ operation and non-deterministically decides whether to operate in 'A-mode' or 'B-mode'. If it decides to operate in A-mode:

- Perform *collapse* on the conditional stack (either u_2 or v_2 depending on which it has) of the top_3 element of the verification chain.
- Perform $push_4; pop_3$—this reveals the previous member of the verification chain as the top_3 stack.
- Repeat until *collapse* has been performed on the second member of the verification chain (we do not do this to the first member). Once this stage is reached, enter distinguished control state $\boldsymbol{A}$.

If it decides to operate in B-mode, it proceeds as follows:

- First examines the token on top of the top_3 stack to determine whether the condition stack of the top_3 element in the chain is u_1 or v_1. If it is u_1 set $w := u$ and if it is v_1 set $w := v$.

- Performs $pop_3; push_3$ so that a copy of the previous element in the chain is now the top_3 stack. The automaton then performs *collapse* on either the guaranteed stack w or the tentative stack w_2 depending on which this previous element in the chain posesses.
- Perform $push_4; pop_3$.
- Repeat until *collapse* has been performed on copies of all but the last members of the verification chain (including the first represented by the bottom 3-stack). Once done enter distinguished control-state $\boldsymbol{B}$.

We add an additional transition labelled ***toCandidate*** from both $\boldsymbol{A}$ and $\boldsymbol{B}$ to a distinguished control-state ***candidate***.

Lemma 4.25. *There exists a Σ_1-sentence ϕ such that for every instance P of Post's Correspondence Problem we have $\mathcal{G}(\mathcal{A}^P_{4_2}) \vDash \phi$ iff P has a solution.*

Proof. Combine Lemmas 4.22 and 4.23. Thus P has a solution if and only if $\mathcal{A}^P_{3^{alt}_2}$ can reach a configuration $(\boldsymbol{guess_{3_2}}, [s_1, s_2, \ldots, s_k])$ such that: For every $1 \leq i < k$:

- If s_{i+1} contains a u_1 then this is equal to u_2 or u in s_i (depending on which one s_i contains)
- or if s_{i+1} contains a v_1 then this is equal to v_2 or v in s_i (depending on which one s_i contains).

We claim that this is the case iff $\mathcal{A}^P_{4_2}$ can both reach a configuration $(\boldsymbol{A}, t)$ and a configuration $(\boldsymbol{B}, t)$ for some stack t.

Suppose first that such a pair of configurations is indeed reachable for $\mathcal{A}^P_{4_2}$. Since a stack produced by either an A-mode run or a B-mode run from a $\mathcal{A}^P_{3^{alt}_2}$ stack $[s_1, s_2, \ldots, s_k]$ will have this 3-stack as its bottom most 3-stack we may conclude that the configuration $(\boldsymbol{A}, t)$ as well as the configuration $(\boldsymbol{B}, t)$ must be produced beginning with the same $\mathcal{A}^P_{3^{alt}_2}$ stack. Since a *collapse* on two elements in copies of some 2-stack s_i will yield the same result iff they are the same, the construction of the A and B modes ensures that the equalities required to relate each s_i to s_{i+1} (for $1 \leq i < k$) must hold. After all, the ith *collapse* performed in B-mode will be on the appropriate component of s_i for $1 \leq i < k$ whilst the ith *collapse* performed in A-mode will be on the correspondingly appropriate component of s_{i+1}. The results of these *collapse*s are directly compared since *collapse* is performed on a copy of the relevant 2-stack that has precisely the same set of 2-stacks below it in each case.

It follows from the above that P does indeed have a solution.

Conversely begin by assuming that P has a solution. It follows that $\mathcal{A}^P_{3^{alt}_2}$ must be able to reach a configuration $(\boldsymbol{guess_{3_2}}, [s_1\ s_2\ \cdots\ s_k])$ satisfying the

conditions above. By the converse considerations to before (in terms of comparing *collapse*s) these conditions must ensure that the A-mode and the B-mode both generate the same stack from this starting point, as required.

We can therefore take as the required Σ_1-sentence the following:

$$\exists x.\exists y.\exists z.(\boldsymbol{A}(x) \wedge \boldsymbol{B}(y) \wedge \boldsymbol{toCandidate}(x,z) \wedge \boldsymbol{toCandidate}(y,z))$$

□

4.7 Summary of Undecidability Results

We summarise the new undecidability results as the following theorem:

Theorem 4.26. *1. For every $n \geq 4$ and $2 \leq m \leq n-2$ the Σ_1-**FO** model-checking problem for n_m-CPDA graphs (even without ϵ-closure) is undecidable.*

*2. For every $n \geq 3$ and $m \geq 3$ the Π_2-**FO** model-checking problem for n_m-CPDA graphs (even without ϵ-closure) is undecidable.*

*3. For every $n \geq 3$ and $m \geq 2$ the Π_2-**FO** model-checking problem for the ϵ-closures of n_m-CPDA graphs is undecidable.*

The Π_2 undecidability results cover almost the entire hierarchy (recalling that $\Sigma_1 \subseteq \Pi_2$); the only remaining gap is the class of 3_2-CPDA graphs (without ϵ-closure). We will fill this in Chapter 7 where we show that the *whole of* **FO** is decidable on 3_2-CPDA graphs (without ϵ-closure). The next chapter, however, considers the Σ_1 question. In particular we will show that Σ_1-**FO** is decidable on the ϵ-closures of 3_2-CPDA graphs as well as the ϵ-closures of n_n-CPDA graphs for all n. Unfortunately the question of Σ_1-**FO** on the $3_{3,2}$-CPDA graphs and n_{n-1} graphs will be left open (both with and without ϵ-closure).

Chapter 5

Monotonic CPDA, Derivatives and Σ_1-Decidability

We introduce the notion of a *monotonic n-CPDA*, which is one that can witness the reachability of all configurations in its graph without destroying $(n-1)$ stacks during its run. Showing that every CPDA has a corresponding monotonic CPDA makes heavy use of logical reflection for the μ-calculus. In some respects can be viewed as the CPDA analogue of Carayol's work [23] extending n-PDA with 'regular tests' in order to derive a canonical means of constructing a stack.

Monotonicity allows us to introduce the idea of the *derivative* of an $(n+1)$-CPDA, which is an n-CPDA whose runs encode the order-$(n+1)$ configurations. This provides the basis of our decidability result for Σ_1-$\boldsymbol{FO}$ on the ϵ-closures of 3_2-CPDA. Intriguingly the barrier to extending this to $3_{3,2}$-CPDA does not arise from the operational effects of *collapse* on 3-links but rather the fact that 3-links are not explicitly represented but can be responsible for distinguishing stacks.

This problem is solved for n_n-CPDA using *trail* annotations in the stack precisely capturing the differences caused by n-links. This allows us to get Σ_1 decidability for the ϵ-closures of n_n-CPDA.

The remaining decidability result for 3_2-CPDA (*without* ϵ-closure, but for the *whole* of $\boldsymbol{FO}$) will be revealed in the final chapter. It is worth mentioning that some parts of this proof, however, will make use of monotonic CPDA and derivatives, so the reader may wish to take note of these ideas in preparation for the final chapter. We will also appeal to a result in the final chapter (the decidability of $\boldsymbol{FO}(\boldsymbol{TC}[\boldsymbol{\Delta_0}])$ on the ϵ-closure of 2-CPDA) in order to get the Σ_1-$\boldsymbol{FO}$ decidability result for Σ_1 on ϵ-closure of 3_2-CPDA.

5.1 Monotonic CPDA

We wish to decompose $\epsilon^* a$-labelled runs of an automaton between two configurations into an *ϵ-fall* and an *ϵa-climb*, which we will describe as a *bounce*. The fall is the first part of the run during which the stack will reach its lowest point, whilst the climb is the part of the run where the lowest point will be built up to the final configuration. This is illustrated in Figure 5.1.

Here 'lowest point' refers to the number of $(n-1)$-stacks constituting the n-stack, and so we find it convenient to ensure that the $|\_|_n$-height of the stack is only ever altered by ϵ-transitions. *We will without loss of generality make the assumption that $push_n$, pop_n and collapse on n-links is* only *performed during ϵ-transitions.* This avoids the need for case separation when monotising automata—we can just focus on ϵ-edges. Generality is not lost since ϵ-closure allows us to decompose an a-labelled $push_n$ edge (for example) into an ϵ-transition with $push_n$ followed by an a-edge with nop.

We will also use Σ to denote the set of *non-ϵ* transition labels and view ϵ as lying outside of Σ.

Let us begin by considering climbs.

Definition 5.1. Let $\mathcal{A}$ be an n-CPDA. An *$\epsilon^* a$-climb* of $\mathcal{A}$ from a configuration (q, s) to a configuration (q', s'), written $(q, s)\boldsymbol{r}^{\uparrow}_{\epsilon^* a}(q', s')$, is an $\epsilon^* a$-labelled run from the first configuration to the second such that each stack t occurring in the run (including s') is such that $pop_n(s) \sqsubset_n t$.

We say that an n-CPDA is *monotonic via r* just in case no r-transition performs a *collapse* on an n-link or a pop_n operation. That is, when transitioning using r-edges, the number of $(n-1)$-stacks in an n-stack increases monotonically. The following Lemma constructs an automaton monotonic via an edge r_ϵ such that ϵ^*-climbs are precisely captured by standard *reachability* using r_ϵ-edges. An n-CPDA is *monotonic* if it can construct all reachable configurations from the initial configuration only by performing monotonic transitions.

Write $\mathcal{A} \restriction_{\Pi,\Sigma}$ for the automaton formed from $\mathcal{A}$ by deleting all unary predicates not labelled in Π and deleting all transitions not labelled in $\Sigma \cup \{\epsilon\}$.

Lemma 5.2. *Let $\mathcal{A}$ be an n-CPDA with edge alphabet Σ and unary predicates Π. Then there exists an n-CPDA $\mathcal{A}^\uparrow$ such that $\mathcal{G}^\epsilon(\mathcal{A}) \mathrel{\widehat{\cong}} \mathcal{G}^\epsilon(\mathcal{A}^\uparrow \restriction_{\Sigma,\Pi})$ but whose additional distinguished edge labels include $r_\epsilon \notin \Sigma$ such that $\mathcal{A}^\uparrow$ is monotonic via r_ϵ and $(q, s)\boldsymbol{r}^{\uparrow}_{\epsilon^* a}(q', s')$ just in case $(q, s)\boldsymbol{r}_{r^*_\epsilon a}(q', s')$.*

Proof. Due to Theorem 2.24 it is sufficient to define an order-n μCPDA $\mathcal{A}^{\uparrow\mu}$ that satisfies the requirements. Conversely recall (from Chapter 2) that we can construct an order-n μCPDA $\mathcal{A}^\mu$ that shares the same configuration graph as $\mathcal{A}$. Extend this to a μCPDA $\mathcal{A}^{\uparrow\mu}$ as follows:

- Add a unary predicate $\boldsymbol{q}$ for each control-state q of $\mathcal{A}$.
- Add a marker $\textsf{marker}[\gamma]$ to the stack-alphabet for each γ in the stack alphabet of Γ. The automaton ensures that at most one of these is on the stack at any one time. Extend the Σ transitions to treat $\textsf{marker}[\gamma]$ as γ and add a single unary predicate **marker** asserting that the marker is on top of the stack.
- We add edges labelled **deployMarker** that simply rewrites the top element of the stack γ to $\textsf{marker}[\gamma]$ without changing the control-state. (As discussed in Chapter 2, rewriting is a conservative extension when considering the ϵ-closure).
- Add edges labelled **removeMarker** that rewrite a $\textsf{marker}[\gamma]$ on top of the stack to γ without altering the control-state.
- Add edges $\boldsymbol{\epsilon_{<n}}$ for each ϵ-transition in $\mathcal{A}$ *not* performing a *collapse* on an n-link; a pop_n nor a $push_n$
- Add edges $\boldsymbol{\epsilon_{push_n}}$ for each ϵ-transition in $\mathcal{A}$ that performs a $push_n$ operation.

Now let ϕ_q be the μ-calculus assertion: 'We can perform **deployMarker** and then perform arbitrary ϵ transitions, beginning with a $push_n$ and immediately removing the marker from the copy, ending up back with the stack at which we started, and indeed stopping precisely when we end up back where we started, with the marker on top and in control-state q.'

This can be expressed in the μ-calculus by the following:

$$\phi_q := < \textsf{deployMarker} >< \boldsymbol{\epsilon_{push_n}} >< \textsf{removeMarker} > \mu X.((\boldsymbol{q} \wedge \textsf{marker}) \vee (< \boldsymbol{\epsilon} > X \wedge \neg \textsf{marker}))$$

We define an r_ϵ-edge to occur whenever we have an $\boldsymbol{\epsilon_{<n}}$-edge or an $\boldsymbol{\epsilon_{push_n}}$-edge. We additionally add an r_ϵ-edge to control-state q' via *nop* whenever $\phi_{q'}$ holds in the current configuration (possible since it is a μCPDA).

Observe that reachability via r_ϵ-edges preserves the original stack alphabet—markers are only implicitly deployed in the definition of each ϕ_q, they are never introduced by an actual transition of $\mathcal{A}^{\uparrow^\mu}$.

Now we argue for correctness. We disregard the single $a \in \Sigma$-transition at the end of the path since by our w.l.o.g. assumption this is an order $(n-1)$-operation and so not pertinent to the definition of a climb.

First suppose that $(q,s)\boldsymbol{r}^{\uparrow}_{\epsilon^*}(q',s')$ (derived from $\mathcal{A}$). All operations featuring in this path other than a pop_n or a *collapse* on an n-link can be replaced directly by an r_ϵ-edge. So we just need to show that pop_n and *collapse* on n-links can also be replaced. The fact that we are considering a climb rather

than an arbitrary run tells us that for every stack t occurring in the run witnessing $(q,s)\boldsymbol{r}^{\uparrow}_{\epsilon^*}(q',s')$ we must have $pop_n(s) \sqsubset_n t$. It must thus be that for every instance of *collapse* on an n-link or pop_n resulting in a configuration (p',t) there must be an earlier configuration of the form (p,t) in the run that is followed by a $push_n$ operation. But then $\phi_{p'}$ holds at this configuration and so there is an r_ϵ-transition from (p,t) to (p',t), as required.

Conversely suppose that there is an r^*_ϵ path from (q,s) to (q',s'). Argue by induction on the length of the path. If $pop_n(s) \sqsubset_n t$, then $t' = push_n(t)$ and $t' = \theta(t)$ for any $\theta \in \Theta_{n-1}$ must satisfy $pop_n(s) \sqsubset_n t'$. Moreover these operations for r_ϵ-edges are directly inherited from the original ϵ-edges and so the path is as required for these operations. It just remains to consider r_ϵ resulting from a $\phi_{p'}$-test at a configuration (p,t) with $s \sqsubset_n t$, resulting in (p',t). But $\phi_{p'}$ asserts the existence of precisely such an ϵ-path. □

Remark 5.3. Since the initial configuration has the empty stack $\bot_n$ and we can w.l.o.g. view '$pop_n(\bot_n)' \sqsubset_n t$ as holding for any stack t (for example by treating the initial stack as $push_n(\bot_n)$ and ensuring the automaton never pops down below this) we get that all reachable configurations are *monotonically reachable* from the initial configuration via $\{r_\epsilon \cup \Sigma\}$-labelled paths.

The dual of an ϵ-climb is *an ϵ-fall*; it captures the idea of a configuration with a higher stack reaching a configuration with a lower stack such that no configuration in the run descends below the lower stack.

Definition 5.4. Let $\mathcal{A}$ be an n-CPDA. An *ϵ-fall* of $\mathcal{A}$ from a configuration (q,s) to a configuration (q',s') is an ϵ^*-labelled run from (q,s) to (q',s') such that for every stack t occurring in the run (including s) we have $pop_n(s') \sqsubset_n t$.

The quasi-analogue of $\mathcal{A}^{\uparrow}$ for falls is $\mathcal{A}^{\downarrow}$. However, we avoid needing to perform any destructive operations by instead making $\mathcal{A}^{\downarrow}$ aware of the predicates that $\mathcal{A}$ could satisfy after performing an ϵ-fall. Whilst something similar could have be done for ϵ-climbs, we need access to the *actual result* of an ϵ-climbs from the base of a fall. This will become clearer when we introduce meta-annotations later in this chapter. Writing $\mathcal{G}\restriction_\Pi$ to mean the graph resulting for deleting all unary predicates not in Π from the graph $\mathcal{G}$:

Lemma 5.5. *Let $\mathcal{A}$ be an n-CPDA with unary predicates Π. Then there exists an n-CPDA $\mathcal{A}^{\downarrow}$ with stack-alphabet $\Gamma^{\downarrow}$ and control-state space $Q^{\downarrow}$ such that $\mathcal{G}(\mathcal{A}) \mathrel{\widehat{\cong}} \mathcal{G}(\mathcal{A}\restriction_{\Pi^{\downarrow}})$ and that also has a predicate $P^{\downarrow}$ for each $P \in \Pi$ such that P holds of precisely those configurations c from which $\mathcal{A}$ has an ϵ-fall to a configuration c' satisfying P.*

Proof. As with the proof of Lemma 5.2 we work with n-μCPDA instead of n-CPDA, as permitted by Theorem 2.24. In fact we begin the construction of $\mathcal{A}^{\downarrow\mu}$ (the μCPDA meeting the requirements that can then be translated to the

CPDA $\mathcal{A}^{\downarrow}$) in exactly the same way as $\mathcal{A}^{\uparrow^{\mu}}$ from Lemma 5.2. We further add an edge $\boldsymbol{q}$ for each $q \in Q$ that transitions to control-state q without altering the stack; a $\boldsymbol{pop_n}$ edge performing a pop_n operation without altering the control-state; and a $\mathbf{destroy_n}$ edge for every ϵ-edge performing a *collapse* on an n-link or a pop_n operation, making the same transition as the ϵ-edge.

We can define the property $\mathbf{marker}^{\downarrow}$ asserting that a marker has already been deployed to the top of some $(n-1)$-stack below in L_{μ}:

$$\mathbf{marker}^{\downarrow} \;:=\; \mu X.(\mathbf{marker} \vee \; < \boldsymbol{pop_n} > X)$$

For each predicate $P \in \Pi$ the following μ-calculus $\psi_{P^{\downarrow}}$ sentence defines the required predicate $P^{\downarrow}$. It asserts reachability whilst making sure the final result is an ϵ-fall by deploying the marker every time we make a $push_n$ operation, thereby enforcing that we should eventually descend below the marker:

$$\begin{aligned}\psi_{P^{\downarrow}} \;:=\; & \mu X.(((P \wedge \neg \mathbf{marker}^{\downarrow}) \\ & \vee \; < \mathbf{destroy_n} \cup \; < \boldsymbol{\epsilon_{<n}} > X \\ & \vee (\neg \mathbf{marker}^{\downarrow} \wedge \; < \mathbf{deployMarker} >< \boldsymbol{\epsilon_{push_n}} > X) \\ & \vee (\mathbf{marker}^{\downarrow} \wedge \; < \boldsymbol{\epsilon_{push_n}} > X)))\end{aligned}$$

Note that some stacks during the run asserted to exist by $\psi_{P^{\downarrow}}$ may contain multiple markers at one time (unlike with $\mathcal{A}^{\uparrow}$). With $\mathcal{A}^{\uparrow}$ we were concerned about knowing when we return to exactly the same stack, whereas here we just want to make sure that we do not stop until we have returned to the last stack at which we performed a $push_n$ (in order to get an ϵ-fall). □

We now formally introduce the idea of a bounce.

Definition 5.6. Let $\mathcal{A}$ be an n-CPDA. An *a-bounce* in $\mathcal{A}^{\uparrow}$ from (q,s) to (q',s') in $\mathcal{A}^{\uparrow}$ is a run consisting of an ϵ-fall from (q,s) to some configuration (q'',s'') followed by an $r_{\epsilon}^{*}a$-climb from (q'',s'') to (q',s'). Let us write $(q,s)\boldsymbol{b}_a(q',s')$ to indicate the existence of such a bounce.

The significance of bounces is summed up in the following lemma. Whilst we strictly only need to consider $\mathcal{A}^{\uparrow}$ we state the Lemma in terms of $\mathcal{A}^{\uparrow^{\downarrow}}$ as when we come to make use of it (with meta-annotations) we will need to reference predicates holding at the bottom of the bounce:

Lemma 5.7. *Let (q,s) and (q',s') be two configurations of a CPDA $\mathcal{A}$ and let $(q,L(s))$ and $(q',L(s'))$ be the corresponding configurations in $\mathcal{A}^{\uparrow^{\downarrow}}$ via the strong isomorphism. Then $(q,s)\boldsymbol{r}_{\epsilon^* a}(q',s')$ just in case $(q,L(s))\boldsymbol{b}_a L(q',L(s'))$.*

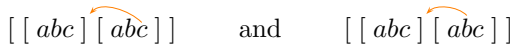
Proof. Suppose first that $(q,L(s))\boldsymbol{b}_a(q',L(s'))$. Then by definition there must be an ϵ-fall from $(q,L(s))$ to some configuration $(q'',L(s''))$ in $\mathcal{A}^{\uparrow^{\downarrow}}$ such that $(q'',L(s''))\boldsymbol{r}^{\uparrow}_{r_{\epsilon}^{*}\ a}(q',L(s'))$. But the latter implies $(q'',L(s''))\boldsymbol{r}^{\uparrow}_{\epsilon^{*}\ a}(q',L(s'))$ and

so there is an $\epsilon^*\ a$ path in $\mathcal{A}^{\uparrow\downarrow}$ from $(q, L(s))$ to $(q', L(s')$. But this uses edges inherited from the original $\mathcal{A}$ and so $(q,s)\boldsymbol{r}_{\epsilon^* a}(q',s')$ in $\mathcal{A}$.

Conversely suppose that $(q,s)\boldsymbol{r}_{\epsilon^* a}(q',s')$ in $\mathcal{A}$. This must be witnessed by a run and we may take the right-most element (q'',s'') in the run such that $pop_n(s'') \sqsubseteq_n pop_n(t)$ for stacks t to the left of s'' in the run. This is the 'final lowest point' the n-stack reaches in the run. By definition of ϵ-fall we have an ϵ-fall from (q,s) to (q'',s''). By Lemma 5.2 we must also have $(q'', L(s''))\boldsymbol{r}^{\uparrow}_{r^*_\epsilon\ a}(q', L(s'))$ in $\mathcal{A}^{\uparrow\downarrow}$since $pop_n(s'') \sqsubset_n pop_n(s')$ (due to it being the final lowest point in the run). We thus have the required bounce. □

5.2 Link Trails: Towards Link Elimination for Graphs

Part of our technique addressing the Σ_1 model-checking problem for CPDA graphs involves the elimination of outer-most links. The operational part of the simulation will make use of bouncing (Lemma 5.7). But it is not enough just to simulate collapse; even if links are never used operationally, they still provide a feature by which stacks may be distinguished. For example the stacks:



are different although removing the link from either gives us the same: $[\,[\,abc\,]\,[\,abc\,]\,]$. We introduce the idea of *link-trails* in order to capture the differences created by links after they have been removed. Stacks and atomic elements are 'coloured' in a manner that is unique to a particular arrangement of n-links.

Definition 5.8. We overload the $\boldsymbol{l_a(s)}$ operator to apply to stacks s as well as individual elements. Let $s = [\ s_1\ s_2\ \cdots\ s_m\ ]$ be a k-stack with $2 \leq k$ (located within an n-stack for $k \leq n$). Define $\boldsymbol{l_a(s)}$ to be the position of the highest $(n-1)$-stack pointed to by an n-link:

$$\boldsymbol{l_a(s)} = \max(\{\ \boldsymbol{l_a(s_i)}\ :\ 1 \leq i \leq m\ \} \cup \{\ 0\ \})$$

and when $s = [\ a_1\ a_2\ \cdots a_m\ ]$ is an order-1 stack:

$$\boldsymbol{l_a(s)} = \max(\{\ \boldsymbol{l_a(a_i)}\ :\ \boldsymbol{l_o(a_i)} = n \text{ and } 1 \leq i \leq m\ \} \cup \{\ 0\ \})$$

So in particular $\boldsymbol{l_a(s)} = 0$ when s contains no element with an n-link.

We now describe how stacks and atomic elements can be ascribed one of four colours in $\{\ c_<, c_=, c_>, \bot\ \}$.

Definition 5.9. We ascribe colours to stacks and atomic elements via a function $\mathbf{col(\_)}$. Let $s = [\ a_1\ a_2\ \cdots\ a_m\ ]$ be a 1-stack located in an n-stack. For

$1 \leq i \leq m$:

$$\mathsf{col}(a_i) := \begin{cases} \bot & \text{if } \boldsymbol{l_o}(a_i) \neq n \\ c_> & \text{if for every } j < i \text{ such that } \boldsymbol{l_o}(a_j) = n \text{ we have} \\ & \boldsymbol{l_a}(a_i) > \boldsymbol{l_a}(a_j) \\ c_= & \text{if } \boldsymbol{l_o}(a_i) = n \text{ and the greatest } j < i \text{ such that} \\ & \boldsymbol{l_o}(a_j) = n \text{ satisfies } \boldsymbol{l_a}(a_j) = \boldsymbol{l_a}(a_i) \end{cases}$$

Note that for constructible stacks the above is exhaustive (it is impossible to construct a stack containing a link with target lower than the target of a link below it in the same 1-stack).

Now let $s := [\ s_1\ s_2\ \cdots\ s_m\ ]$ be an order-k stack in an order-n stack for $n \geq k \geq 2$ (in particular we allow $k = n$ in which case s is the whole n-stack). We then set $\mathsf{col}(s_i)$ for $1 \leq i \leq m$ as follows:

$$\mathsf{col}(s_i) := \begin{cases} c_> & \text{if } \boldsymbol{l_a}(s_i) > \max(\{\ \boldsymbol{l_a}(s_j)\ :\ 1 \leq j < i\ \}) \\ c_= & \text{if } \boldsymbol{l_a}(s_i) = \max(\{\ \boldsymbol{l_a}(s_j)\ :\ 1 \leq j < i\ \}) \\ c_< & \text{if } \boldsymbol{l_a}(s_i) < \max(\{\ \boldsymbol{l_a}(s_j)\ :\ 1 \leq j < i\ \}) \end{cases}$$

The next step is to show how a CPDA can dynamically assign colours to its stacks correctly. We restrict ourselves to automata that *only* have n-links so that the only way to destroy internal stacks is using a higher-order *pop* operation. Let s be an n_n-CPD stack over the alphabet Γ. We define the *colour tracking* stack $\mathsf{colTr}(s)$ to be the stack over the alphabet:

$$\Gamma \quad \cup \quad \Gamma \times \{\ \bot, c_=, c_>\ \} \times \prod_{i=0}^{n-2} \{\ c_<, c_=, c_>\ \}^{n-1-i} \quad \cup \quad [\ 1..(n{-}1)\ ] \times \{\ c_<, c_=, c_>\ \}$$

We construct $\mathsf{colTr}(s)$ from s by first replacing each atomic element a *that emits a link* in s with an element

$$(a, c_0, (b_1^0, \ldots, b_{n-1}^0), (b_2^1, \ldots, b_{n-1}^1), \ldots, (b_{n-1}^{n-2}))$$

where:

- $c_0 := \mathsf{col}(a)$
- For each i, j such that $0 \leq i < j \leq n-1$ let $s_i := top_{i+1}(s_{\leq a})$. Then:
  - Abuse notation by taking $top_{n+1}(s) := s$. If there exists another element a' below s_i in $top_{j+2}(s_{<s_i})$ such that $\boldsymbol{l_a}(a) = \boldsymbol{l_a}(a')$ that has been assigned $b_j'^i$, then $b_j^i := b_j'^i$.
  - Otherwise $b_j^i := \mathsf{col}(s_i') := \mathsf{col}(top_{j+1}(s_{<s_i}))$.

We finish the construction of $\mathbf{colTr}(s)$ by adding a decoration $(i, \mathbf{col}(s_i))$ on top of each i-stack s_i in s for $1 \leq i \leq n-1$.

The idea is that each component stack of s is annotated with its colour and the additional decorations provide the necessary information to determine how a stack operation affects the colours.

It will be useful to have a procedure executable by an n-PDA that locates the lowest element a in $top_{i+1}(\mathbf{colTr}(s))$ such that $\boldsymbol{l_a}(a) \geq \boldsymbol{l_a}(a')$ for all other elements a' in $top_{i+1}(\mathbf{colTr}(s))$. We can do this by first finding the $(i-1)$-stack containing the requisite a, then the $(i-2)$-stack within that stack containing a and so on. More precisely we recursively apply the following to $top_{i+1}(\mathbf{colTr}(s))$:

- If $i = 1$ and the colour of the top_1 element is $c_>$, then the top_1 element is the required a.
- If the colour of the top_i stack for $i > 1$ is $c_>$, then recursively apply the procedure to the top_i stack.
- If the colour of the top_i stack for $i \geq 1$ is $c_=$ or $c_<$, then the requisite element must lie below (by definition of colours), so perform pop_i and recursively apply the procedure to the new top_{i+1} stack. (Note that pop_i will not alter any colours since the discarded stack cannot contribute anything beyond that which is below it since it does not have a $c_>$ colour.)

Let us call this procedure $\mathbf{FLHPE}_{i+1}$ *'find the lowest highest pointing element in the top_{i+1}-stack'*.

The utility of finding such an element a becomes clear when we want to perform a pop_{i+1} operation on $\mathbf{colTr}(s)$ (for $1 \leq i \leq n-2$) and learn the new colour of the top_{j+1}-stack (for $i < j \leq n-1$) after performing this operation. When the top_{i+1} stack being discarded has colour $c_>$ together with all of the top_{k+1} stacks for $i \leq k \leq j$, the required information will appear on the b^i_j decoration of a located by $\mathbf{FLHPE}_{i+1}$:

Lemma 5.10. *Let s be an n_n-stack. Let $a := (a, c_0, (b^0_1, \ldots, b^0_{n-1}), (b^1_2, \ldots, b^1_{n-1}), \ldots, (b^{n-2}_{n-1}))$ be the lowest occurrence of an atomic element in $top_{i+1}(\mathbf{colTr}(s))$ such that for all other occurrences of an element a' in $top_{i+1}(\mathbf{colTr}(s))$ we have $\boldsymbol{l_a}(a) \geq \boldsymbol{l_a}(a')$. Then $\mathbf{col}(top_{k+1}(\mathbf{colTr}(s))) = c_>$ for every $i \leq k \leq j$ implies $\mathbf{col}(top_{j+1}(pop_{i+1}(\mathbf{colTr}(s)))) = b^i_j$ for $0 \leq i < j < n$.*

Proof. By the definition of a we must have $\mathbf{col}(top_{k+1}(\mathbf{colTr}(s)_{\leq a})) = c_>$ for all $0 \leq k < i$. If additionally $\mathbf{col}(top_{k+1}(\mathbf{colTr}(s))) = c_>$ for every $i \leq k \leq j$, it must be the case that $\boldsymbol{l_a}(a) > \boldsymbol{l_a}(a')$ for all occurrences of an element a' in $top_{j+2}(\mathbf{colTr}(s))$ below a and so in particular below top_{i+1}. The result then follows by the definition of b^i_j. □

It is also helpful to be able to use colour annotations to recover which stacks contain fresh links.

Lemma 5.11. *Let s be an n_n-stack. Then for each $2 \leq i \leq n$, the stack $top_i(s)$ contains an n-link from an atom a with $\boldsymbol{l_r}(a) = 1$ iff $\mathbf{col}(top_n(s)) = c_>$ and additionally for each j such that $i \leq j < n$ we have $\mathbf{col}(top_j(s)) \in \{ c_=, c_> \}$.*

Proof. First suppose that $top_i(s)$ contains an n-link from an atom a with $\boldsymbol{l_r}(a) = 1$. Since no link in the $(n-1)$-stack below can source a link with the same target, we must have $\mathbf{col}(top_n(s)) = c_>$. Moreover, no link in the $top_n(s)$ stack can have a target above that of a. Since $top_i(s)$ contains a, $top_j(s)$ must contain a for $i \leq j < n$ and so $\mathbf{col}(top_j(s)) \in \{ c_=, c_> \}$, as required.

Now suppose that the right-hand-side of the 'iff' holds. Since $\mathbf{col}(top_n(s)) = c_>$ the top_n stack must contain a fresh n-link as all other n-links in the top_n stack would have been created and so exist in an n-stack below it. Suppose for contradition that $top_i(s)$ does not contain a fresh n-link. Then there must be a maximum j with $i < j \leq n-1$ such that $top_j(s)$ does not contain a fresh n-link. But since top_n *does* contain a fresh n-link there must exist an n-link below $top_j(s)$ in $top_{j+1}(s)$ whence we would have $\mathbf{col}(top_j(s)) = c_<$, a contradiction. □

The following Lemma tells us that we can preserve the correct annotations whilst manipulating an n_n-stack. Unfortunately we do not have a version of this Lemma for n-stacks containing links of other orders.

Lemma 5.12. *Let s be an n_n-stack over an alphabet Γ and let θ be an n-stack operation. There then exists a 'compound stack operation' θ' (that may include operations conditional on observations made using a finite number of control-states) such that $\mathbf{colTr}(\theta(s)) = \theta'(\mathbf{colTr}(s))$*—i.e. *$\theta'$ could be implemented by an n_n-CPDA. Moreover the number of operations in θ' is bounded.*

Proof. Consider each possible θ in turn.

If it is a $push_1$ operation, then if no link is attached, no colour is affected and we can simply take $\theta' = \theta$.

If it is a $push_1^{a,n}$ operation, then we first pop off the colour annotations on top of the stack. In the light of Lemma 5.11 these colour annotations allow us to deduce the set $F \subseteq [2..n]$ of elements i such that $top_i(s)$ contains a fresh n-link. The colour of any top_i-stack with $i \in F$ is unchanged since they already contain a link with the highest possible target. The colour of top_i-stacks with $i \notin F$ (with $i \in [1..n]$) are set as follows (which do not necessarily but may result in a change of colour):

- if $i+1 \in F$ and $\mathbf{col}(top_i(s)) \in \{c_<, c_=\}$, then a stack below $top_i(s)$ in $top_{i+1}(s)$ already contains a link and so the new colour of the new top_i stack is $c_=$.

- otherwise the new top_i stack has colour $c_>$ (it either already has this colour or else $i+1 \notin F$ and so this is the first fresh link in $top_{i+1}(s)$).

Note that this ensures the colour c_0 of the new atomic element to be created is either $c_=$ or $c_>$ depending on whether there already exists a fresh link below it in the top_2 stack. The actual element being pushed onto the stack has the form:

$$(a, c_0, (b_1^0, \ldots, b_{n-1}^0), (b_2^1, \ldots, b_{n-1}^1), \ldots, (b_{n-1}^{n-2}))$$

where we take b_j^i as follows:

- Note that it is impossible for there to be an element in a lower $(n-1)$-stack containing the same pointer as the freshly created a. If $j < n-1$ and $j+2 \in F$ and $\mathbf{col}(top_{k+1}(s)) \in \{c_<, c_=\}$ for some $i \leq k \leq j$ then there must exist another fresh element a' below the top_{i+1} stack in top_{j+2}—*i.e.* an element with a link of the same target as the element a being created. In order to comply with the definition of $\mathbf{colTr}(push_1^{a,n}(s))$ we can apply the procedure $\mathbf{FLHPE}_{j+2}$ to a *temporary copy* of $top_{j+2}(s)$ (such a temporary copy can be created using $push_{j+2}$ since $j < n-1$) in order to find the lowest such element a'. We then set $b_j^i := b_j'^i$ where $b_j'^i$ is the corresponding parameter for a'.

- Otherwise no such element a' exists so we must set b_j^i to be the resulting colour of the top_{j+1} stack if we were to perform a pop_{i+1} operation. If $i = 0$ then $b_j^i := \mathbf{col}(top_{j+1}(\mathbf{colTr}(s)))$ (*i.e.* the colour of the top_{j+1} stack prior to adding the new atom). If $i > 0$ then observe that $pop_{i+1}(\mathbf{colTr}(s)) = pop_{i+1}(\mathbf{colTr}(push_1^{a,n}(s)))$ and so in particular $\mathbf{col}(top_{j+1}(pop_{i+1}(\mathbf{colTr}(s)))) = \mathbf{col}(top_{j+1}(pop_{i+1}(\mathbf{colTr}(push_1^{a,n}(s)))))$. We thus set b_j^i to be the colour that we would have set the top_{j+1} to be had we performed pop_{i+1} on s. This procedure is detailed below (if $\mathbf{FLHPE}_{i+1}$ has to be carried out on top_{i+1} it should be done on a temporary copy of top_{i+1}).

If $\theta = pop_i$ for $1 \leq i < n$, then it is only necessary to update the colour annotations for the new top_{j+1} stack with $i \leq j < n$ (by popping them off and pushing on new appropriate annotations).

- If there exists a k with $i \leq k < j$ such that $\mathbf{col}(top_k(\mathbf{colTr}(s))) \in \{c_<, c_=\}$, then we must have $\mathbf{col}(top_j(pop_i(\mathbf{colTr}(s)))) = \mathbf{col}(top_j(\mathbf{colTr}(s)))$ (*i.e.* the colour of the top_j stack remains unchanged). This is because there is a link in the top_j stack below the top_k stack (in which the top_i

stack being discarded resides or else which is the same as the stack being discarded) that points at least as high as any link in the top_i stack.

- If no such k exists (so that $\mathsf{col}(top_k(\mathsf{colTr}(s))) = c_>$ for every $i \leq k < j$) but $\mathsf{col}(top_j(\mathsf{colTr}(s))) \in \{c_<, c_=\}$, then $\mathsf{col}(top_j(pop_i(\mathsf{colTr}(s)))) = c_<$ since we are discarding the highest pointing link from the top_j stack that only points at most as high as any $(j-1)$-stack below the top_j stack.

- Otherwise $\mathsf{col}(top_k(\mathsf{colTr}(s))) = c_>$ for every $i \leq k \leq j$. We can then apply FLHPE_i to $top_i(\mathsf{colTr}(s))$ to discover the new colour of the new top_j stack using Lemma 5.10.

If $\theta = pop_n$ then we can just have $\theta' = pop_n$ as no colour annotations will need changing. Likewise since we only have n-links we may take $\theta' = collapse$ when $\theta = collapse$.

If $\theta = push_k$ for $2 \leq k \leq n$, then we discard all colour annotations (i, c_i) on top of the top_{i+1} stack for $i \geq k$, perform a $push_k$ operation changing the colour annotation on top of the resulting top-most $k-1$ stack to $(k-1, c_=)$ and replacing all of the previously discarded decorations unchanged. The colour annotations on i-stacks for $i > k-1$ will remain correct as no new links are created and those on i-stacks for $i < k-1$ will remain correct as they depend only on what was copied in its entirety by the $push_k$ operation.

Observe how none of the b^i_j values of atomic elements in the copied stack need changing. Overloading the notation a consider an atom

$$a := (a, c_0, (b^0_1, \ldots, b^0_{n-1}), (b^1_2, \ldots, b^1_{n-1}), \ldots, (b^{n-2}_{n-1}))$$

occurring in the newly created $(k-1)$-stack.

- For $0 \leq i < j < k-1$ note that $top_{j+2}(push_k(\mathsf{colTr}(s)))$ is a copy of a $(j+1)$-stack either in or equal to the $(k-1)$-stack below. Since the meaning of b^i_j is completely determined by the $(j+1)$-stack in which it resides, this remains correct.

- For $0 \leq i < j = k-1$ note that a is a copy of an element (in particular with a link to the same target) in the top_{j+2} $(j+1)$-stack below the new top_{j+1} stack and so in particular below the new top_{i+1} stack.

- For $i \leq k-1 < j \leq n-1$ we must also have that a is a copy of an element with the same link in the top_{j+2} $(j+1)$-stack below the new top_{i+1} stack.

- For $i > k-1$ we have $pop_i(push_k(\mathsf{colTr}(s))_{\leq a}) = pop_i(\mathsf{colTr}(s)_{\leq a'})$, where a' is the original from which a is copied. Hence correctness is inherited.

□

We informally use the phrase *link trail* to refer to the extra information bourne by a stack $\mathsf{colTr}(s)$ in contrast to s.

Link trails allow us to uniquely reconstruct links in a stack. Write $\mathsf{stripln}(s)$ to denote the result of deleting links from a stack s.

Lemma 5.13. *Given two constructible n_n-stacks s and s' we have $s = s'$ iff $\mathsf{stripln}(\mathsf{colTr}(s)) = \mathsf{stripln}(\mathsf{colTr}(s'))$.*

Proof. The functionality of $\mathsf{colTr}(s)$ implies the left to right direction.

For the other direction suppose that $\mathsf{stripln}(\mathsf{colTr}(s)) = \mathsf{stripln}(\mathsf{colTr}(s'))$. We give a method that allows one to recover the target of a link in s from the corresponding position in $\mathsf{stripln}(\mathsf{colTr}(s))$, which means that this equality implies $s = s'$.

It suffices to give a procedure that given $\mathsf{colTr}(s)_{\leq a}$ for some atom a in $\mathsf{colTr}(s)$ will locate a lower element in the stack that has the same link. When colouring tells us that there is no lower element (using Lemma 5.11) we know that the link must point to the $(n-1)$-stack immediately below and thus have recovered the target of the original link. We may use Lemma 5.12 to ensure that the annotations in the stack remain correct during the procedure.

Consider an atom a in $\mathsf{colTr}(s)$ and let $s_0 := \mathsf{colTr}(s_{\leq a})$—this can be computed using Lemma 5.12 with a suitable sequence of *pop* operations to locate a. If there exists an a' below a in s with a pointer to the same target, then there must exist a least i with $0 \leq i < n$ such that $\mathsf{col}(top_{i+1}(s_0)) \neq c_>$. The leastness of i means that a is the highest pointing element in the $top_{i+1}(s_0)$ stack (when $i > 0$).

If $\mathsf{col}(top_{i+1}(s_0)) = c_=$, then we may apply FLHPE_{i+2} on $top_{i+2}(pop_{i+1}(s_0))$ to find a lower element with a pointer to the same target. This is correct since by assumption a is the highest pointing element in $top_{i+1}(s_0)$ and so by definition of $c_=$, a must share a target with the highest pointing elements in $top_{i+2}(s_0)$.

If $\mathsf{col}(top_{i+1}(s_0)) = c_<$ then there must be an element b below it in $top_{i+2}(pop_{i+1}(s_0))$ that points to a higher target. Since we are considering only stacks that are constructible (from the empty stack) we may consider the sequence of order-$(i{+}1)$ stack operations that were used to construct $top_{i+2}(s_0)$. After a $push_1^{b',n}$ operation creating an element with the same target as b there could not be any further $push_1^{a',n}$ operations creating a link with the same target as a (since a points lower than b and new links can only point to the stack immediately below). This means that the occurrence in a on top of $top_{i+1}(s_0)$ must have been created by a higher-order push operation. Indeed it must have been created by a $push_{i+2}$ operation since it is the lowest element with its target in $top_{i+1}(s_0)$. This means that $top_{i+1}(s_0)$ is a prefix of $top_{i+1}(pop_{i+2}(s_0))$ and we can use this fact to find a lower element with the same target. □

5.3 Link Elimination for Σ_1 properties on n_n-CPDA

In this subsection we construct an n-PDA over which we may express properties equivalent to a Σ_1 property holding of the original n_n-CPDA graph. Given an n_n-CPDA $\mathcal{A}$ we define the *illumination* $\mathsf{lum}(\mathcal{A})$ of $\mathcal{A}$ to be the result of adding link trails to its stack. The automaton $\mathsf{lum}(\mathcal{A})$ generates the same underlying graph (up to ϵ-closure) but makes use of Lemma 5.12 to ensure equality between two different $\mathsf{lum}(\mathcal{A})$ stacks does not depend on n-links.

Lemma 5.14. *Let $\mathcal{A}$ be an n_n-CPDA. Then there exists an n_n-CPDA $\mathsf{lum}(\mathcal{A})$ such that $\mathcal{G}^\epsilon(\mathsf{lum}(\mathcal{A})) \cong \mathcal{G}^\epsilon(\mathcal{A})$ and further such that for any reachable configurations $(q,s),(q,s')$ of $\mathsf{lum}(\mathcal{A})$ we have $s = s'$ iff $\mathsf{stripln}(s) = \mathsf{stripln}(s')$.*

Proof. We define $\mathsf{lum}(\mathcal{A})$ to be the n_n-CPDA with the stack alphabet induced by $\mathsf{colTr}(\_)$ that replaces all operations of $\mathcal{A}$ generating an a-edge with a compound operation from Lemma 5.12 where the compound generates a path of the form $\epsilon^* a$. Lemma 5.13 ensures that $s = s'$ iff $\mathsf{stripln}(s) = \mathsf{stripln}(s')$ for any stacks s and s' reachable by $\mathsf{lum}(\mathcal{A})$. The predicates for $\mathsf{lum}(\mathcal{A})$ are induced by those for $\mathcal{A}$ by projecting the stack alphabet and control-states of $\mathsf{lum}(\mathcal{A})$ onto those of $\mathcal{A}$. □

We now add a mechanism by which we can simulate edges in the graph without needing to actually perform the *collapse* operations. Indeed it will remove the need to perform any stack operations at all. This is achieved by decorating $(n-1)$-stacks within an n-stack with information about the control-states from which a complete n-stack could be reached via an $\epsilon^* a$-climb. By asserting the existence of a suitable ϵ-fall to such a decoration, we can assert the existence of a bounce witnessing reachability, as illustrated in Figure 5.1.

In order to assert the *non-existence* of an edge between (q,s) and (q',s') we assert that there is *no* ϵ-fall to a suitable point. As suggested by Figure 5.2, this requires the $(n-1)$-stacks to be decorated with information about *all* possible stacks that can be monotonically generated from it. Because this requires quantification over multiple runs and a CPDA can only perform one run at a time, it is necessary for the CPDA to guess the appropriate annotations and to externally verify them. We will thus add a predicate to our logic asserting that all of the decorations are 'correct' and then subsequently show that this logic is decidable for n-PDA (*i.e.* n_n-CPDA with their outer-most links eliminated).

Definition 5.15. Fix $k \in \mathbb{N}$ and let $\mathcal{A}$ be an n_n-CPDA with control-states Q and edge-labels in Σ. A *k-meta-annotation* for $\mathcal{A}$ is a $|\Sigma|.k$-tuple $((Q_1^a)_{a\in\Sigma},\ldots,(Q_k^a)_{a\in\Sigma})$ where each component is a subset of Q.

Given an n_n-CPDA $\mathcal{A}$ and $k \in \mathbb{N}$ the n-PDA $\mathsf{GrStripln}_k(\mathcal{A})$ is formed using the following recipe:

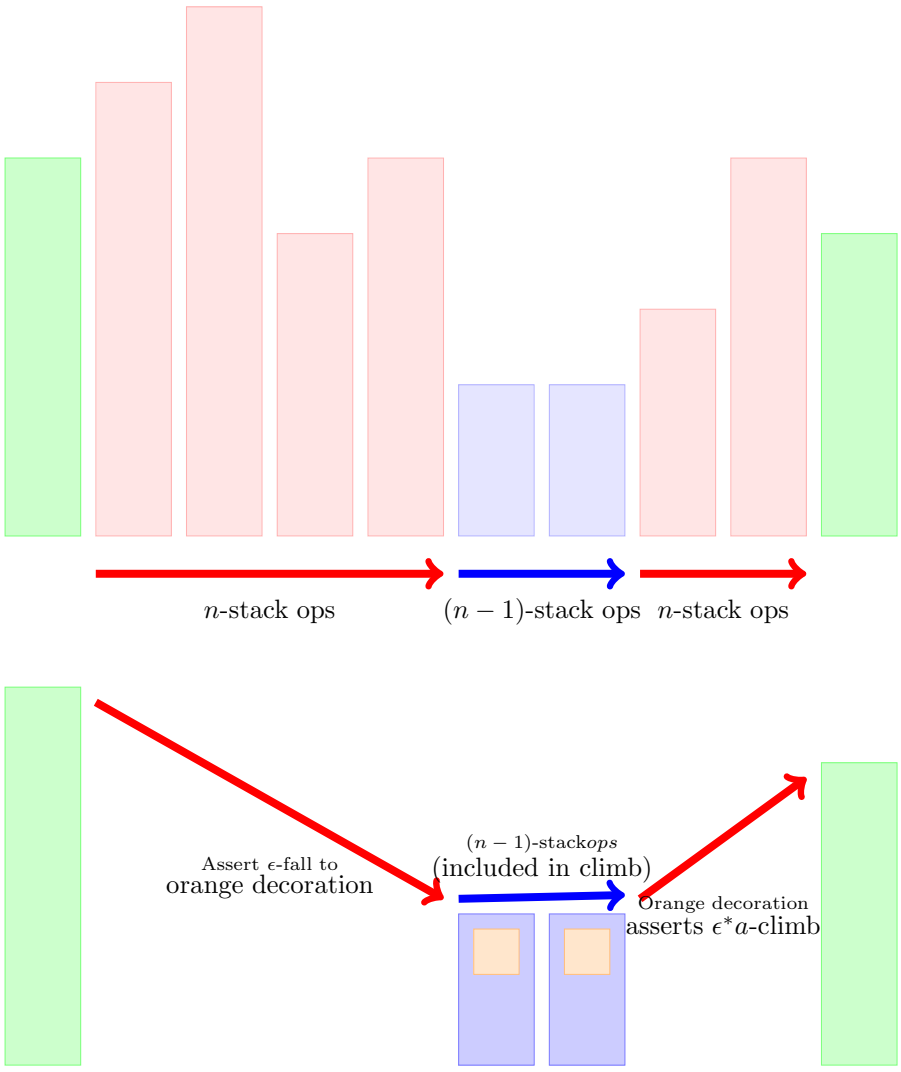


Figure 5.1: Asserting the existence of a bounce without performing any stack operations.

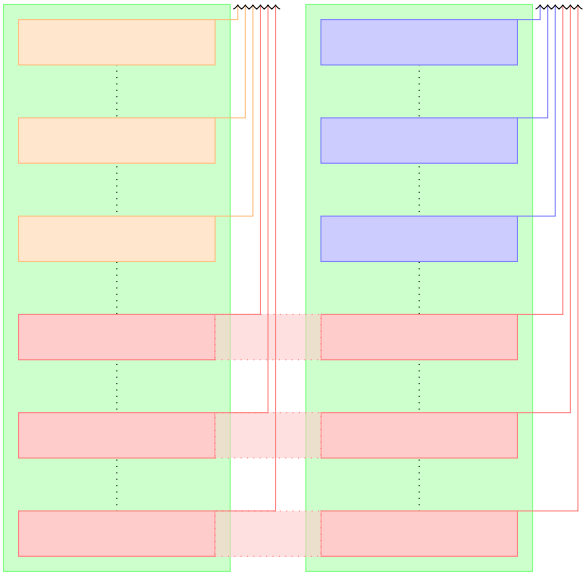


Figure 5.2: Guessing the manner in which a stack may be produced from bottom up. This involves decorating the top of each stack with a guess as to which control-states at that stack could yield which control-states upon generating the whole stack. When two stacks being compared share a common initial segment, the guesses for each stack should decorate both sets of stacks in the shared initial segment.

- Take the n_n-CPDA $\mathsf{lum}(\mathcal{A})$ and modify it so that a single k-meta-annotation $((Q_1^a)_{a\in\Sigma},\ldots,(Q_k^a)_{a\in\Sigma})$ is kept at the top of every $(n-1)$-stack. No restriction is placed on what this may be (it is non-deterministically chosen from amongst all k-meta-annotations). We add a predicate $\mathsf{Met}(((Q_1^a)_{a\in\Sigma},\ldots,(Q_k^a)_{a\in\Sigma}))$ holding at all configurations with the corresponding meta-annotation on top together with a predicate $\mathsf{Met}(q,((Q_1^a)_{a\in\Sigma},\ldots,(Q_k^a)_{a\in\Sigma}))$ additionally asserting that the automaton is in control-state q. Unary predicates are inherited directly from $\mathcal{A}$ on the basis of control-state and stack symbol immediately below the meta-annotation. Call this n_n-CPDA $\mathsf{lum}(\mathcal{A})_k^+$.

- Let $\mathsf{GrStripln}_{\boldsymbol{k}}(\mathcal{A})^-$ be the automaton $\mathsf{lum}(\mathcal{A})_k^{+\uparrow\downarrow}$.

- Finally $\mathsf{GrStripln}_{\boldsymbol{k}}(\mathcal{A})$ is $\mathsf{GrStripln}_{\boldsymbol{k}}(\mathcal{A})^-$ restricted to edges that do not perform a *collapse* or a pop_n operation, and we remove all links. Thus $\mathsf{GrStripln}_{\boldsymbol{k}}(\mathcal{A})$ is an n-PDA. We further add an edge $\mathsf{stackComp}$ from each configuration (q,s) to a configuration $(\boldsymbol{s?},s)$ for a distinguished control-state $\boldsymbol{s?}$. This allows stacks from different configurations to be manipulated and compared without prejudice to their control-

states. Also add an edge labelled $\boldsymbol{p}$ for every $p \in Q$ such that for any control-state $\hat{q}$ of $\mathbf{GrStripln_k}(\mathcal{A})$ and control-state $\hat{p}$ corresponding to p we have $(\hat{q}, s)\boldsymbol{p}(\hat{p}, s)$. Add an edge $\boldsymbol{pop_n}$ from each (q, s) to $(q, pop_n(s))$.

Let us break down each stage of this construction. We need to classify the stacks for which the k-meta-annotations are considered 'correct'—something that $\mathbf{lum}(\mathcal{A})^+_k$ has no control over itself—it is a constraint imposed from the outside. Indeed correctness is only defined for k-tuples of configurations since every meta-annotation references reachability to each of k different configurations. This correctness property is known as *consistency*.

Definition 5.16. Let $(q_1, s_1), \ldots (q_k, s_k)$ be reachable configurations of $\mathbf{lum}(\mathcal{A})^+_k$. Then we say that this k-tuple of configurations is *consistent* just in case the following conditions are met:

- For each i with $1 \leq i \leq k$ it is the case that each $(n-1)$-stack in s_i contains precisely one meta-annotation, which must occur on top of it.
- Suppose that $t \sqsubseteq_n s_i$ for some $1 \leq i \leq k$. Then the meta-annotation on top of $top_n(t)$ must be $((Q^a_1)_{a\in\Sigma}, \ldots, (Q^a_k)_{a\in\Sigma})$ where for each $1 \leq j \leq k$:

$$Q^a_j := \{\ q \in Q\ :\ (q, t)\boldsymbol{r}^{\uparrow}_{\epsilon^* a}(q_j, s_j)\ \}$$

  Note that $Q^a_j = \emptyset$ if there is no $\epsilon^* a$-climb from any configuration (q, t) to (q_j, s_j), which in particular is the case if $pop_n(t) \not\sqsubseteq_n s_j$.

Now let L be the map from $\mathbf{lum}(\mathcal{A})^+$-stacks to $\mathbf{GrStripln_k}(\mathcal{A})^-$-stacks witnessing the strong isomorphism $\mathcal{G}^\epsilon(\mathbf{lum}(\mathcal{A})^+) \mathrel{\widehat{\cong}} \mathcal{G}^\epsilon(\mathbf{GrStripln_k}(\mathcal{A})^-) \restriction_{\Pi,\Sigma}$, where Π and Σ are the unary predicates and edge labels from $\mathbf{lum}(\mathcal{A})^+$. So if (q, s) is a node of $\mathcal{G}^\epsilon(\mathbf{lum}(\mathcal{A})^+)$, the corresponding node in $\mathcal{G}^\epsilon(\mathbf{GrStripln_k}(\mathcal{A})^-)$ is $(q, L(s))$. Note further that since $\mathbf{GrStripln_k}(\mathcal{A})^-$ is monotonic, deleting destructive transitions will not change the set of reachable configurations. Thus $(q, \mathbf{stripln}(L(s)))$ is also a reachable configuration of $\mathcal{G}^\epsilon(\mathbf{GrStripln_k}(\mathcal{A}))$. By Lemma 5.14 $L(s)$ is completely determined by $\mathbf{stripln}(L(s))$, and so for notational convenience we drop the $\mathbf{stripln}()$ and view $(q, L(s))$ as the 'configuration of $\mathcal{G}^\epsilon(\mathbf{GrStripln_k}(\mathcal{A}))$ corresponding to (q, s)'.

Lemma 5.17. *For each $k \in \mathbb{N}$ and n_n-CPDA $\mathcal{A}$, there exists an MSO formula $\mathbf{con}(x_1, x_2, \ldots, x_k)$ such that reachable configurations $(q_1, s_1), \ldots, (q_k, s_k)$ of $\mathbf{lum}(\mathcal{A})^+$ are consistent just in case:*

$$\mathbf{GrStripln_k}(\mathcal{A}) \models \mathbf{con}((q_1, L(s_1)), \ldots, (q_k, L(s_k)))$$

and such that for every i with $1 \leq i \leq k$, $\mathbf{GrStripln_k}(\mathcal{A}) \models \mathbf{con}(c_1, \ldots, c_k)$ implies that $c_i = (q, L(s))$ for some reachable configuration (q, s) of $\mathbf{lum}(\mathcal{A})^+$.

Proof. First observe that for $\mathbf{lum}(\mathcal{A})^+_k$ configurations (q,s) and (q',s') with corresponding $\mathbf{GrStripln_k}(\mathcal{A})$ configurations $x=(q,L(s))$ and $y=(q',L(s'))$ we can MSO-define $s \sqsubseteq_n s'$ in $\mathcal{G}(\mathbf{GrStripln_k}(\mathcal{A}))$ using a standard least fixed-point construction:

$$x \sqsubseteq_n y := \exists X.(\exists x'.x\mathbf{stackComp}x')(\exists y'.y\mathbf{stackComp}y'). \\ (x' \in X \wedge \phi_{\sqsubseteq_n}(X,y') \wedge \forall Y.(\phi_{\sqsubseteq_n}(Y,y') \rightarrow X \subseteq Y))$$

where

$$\phi_{\sqsubseteq_n}(X,x',y') := \forall z.(z \in X \leftrightarrow (z = y' \vee (\exists z' \in X).z'\boldsymbol{pop_n}z))$$

We do indeed have $s \sqsubset_n s'$ iff $L(s) \sqsubseteq_n L(s')$ since strong isomorphisms preserve stack structure.

We additionally need a way of capturing the configurations of $\mathbf{GrStripln_k}(\mathcal{A})$ that correspond to the reachable configurations of $\mathbf{lum}(\mathcal{A})^+$, namely those monotonically generated by $\mathbf{GrStripln_k}(\mathcal{A})^-$:

$$\boldsymbol{R}(x) := \exists X.\exists x'.(x' \in X \wedge \phi_{\boldsymbol{R}}(X) \wedge \forall Y.(\phi_{\boldsymbol{R}}(Y) \rightarrow X \subseteq Y) \wedge \bigvee_{a\in\Sigma} x'\boldsymbol{a}x)$$

where

$$\phi_{\boldsymbol{R}}(X) := \forall z.(z \in X \leftrightarrow (z = c_0 \vee (\exists z' \in X).(z'\boldsymbol{r_\epsilon}z \vee \bigvee_{a\in\Sigma\cup\{\epsilon\}} z'\boldsymbol{a}z))$$

where c_0 is the initial configuration. Using a standard least-fixed-point construction, $\boldsymbol{R}(x)$ defines those configurations of $\mathbf{GrStripln_k}(\mathcal{A})$ reachable via an $(r_\epsilon + \Sigma + \epsilon)^*\Sigma$-labelled path. By Remark 5.3 these are precisely the configurations of $\mathbf{GrStripln_k}(\mathcal{A})$ corresponding to those in $\mathbf{lum}(\mathcal{A})^+$, noting that no r_ϵ-edge is deleted in forming $\mathbf{GrStripln_k}(\mathcal{A})$ from $\mathbf{GrStripln_k}(\mathcal{A})^-$ since the latter is monotonic via r_ϵ. Also note that our w.l.o.g. assumption that all Σ-labelled edges perform an order-$(n-1)$ operation prevents any Σ-labelled edge from being deleted.

For $\mathbf{lum}(\mathcal{A})^+_k$ configurations (q,s) and (q',s') with corresponding $\mathbf{GrStripln_k}(\mathcal{A})$ configurations $x=(q,L(s))$ and $y=(q',L(s'))$ we can MSO-define $(q,s)\boldsymbol{r}^{\uparrow}_{\epsilon^*a}(q',s')$ for $a \in \Sigma$ in $\mathcal{G}(\mathbf{GrStripln_k}(\mathcal{A}))$ with the ϵ^*a-climb interpreted over $\mathbf{lum}(\mathcal{A})^+$ by defining $x\boldsymbol{r}_{r_\epsilon^*a}y$ in $\mathcal{G}(\mathbf{GrStripln_k}(\mathcal{A}))$. These two are equivalent due to Lemma 5.2 together with the w.l.o.g assumption that all Σ-labelled operations are order-$(n-1)$.

$$x\boldsymbol{r}^{\uparrow}_{\epsilon^*a}y := x\boldsymbol{r}_{r_\epsilon^*a}y := \exists X.(\exists y'.y'\boldsymbol{a}y).(y' \in X \wedge \phi_{\boldsymbol{r}_{r_\epsilon^*}}(X,x) \wedge \\ \forall Y.(\phi_{\boldsymbol{r}_{r_\epsilon^*}}(Y,x) \rightarrow X \subseteq Y))$$

where

$$\phi_{\boldsymbol{r}_{r_\epsilon^*}}(X,x) := \forall z.(z \in X \leftrightarrow (z = x \vee (\exists z' \in X).z'\boldsymbol{r_\epsilon}z))$$

We define a predicate **meta** asserting that a configuration has a meta-annotation on top:

$$\mathbf{meta}(x) \;:=\; \bigvee_{m\in M} \mathbf{Met}(m)(x)$$

where M is the set of meta-annotations. We also define the following predicates that can be used to express some basic properties about the meta-annotation on top of a stack:

$$[\boldsymbol{q \in Q_i^a}](x) \quad := \quad \bigvee_{\substack{m=((Q_1^a)_{a\in\Sigma},\ldots,(Q_k^a)_{a\in\Sigma})\in M \\ q\in Q_i^a}} \mathbf{Met}(m)(x)$$

for each $1 \leq i \leq k$, $a \in \Sigma$ and $q \in Q$. We can now read off the definition of consistency to define:

$$\mathbf{con}(x_1, x_2, \ldots, x_k) := \forall x. \left(\bigvee_{i=1}^{k} x \;\sqsubseteq_n\; x_i\right) \;\rightarrow\; \mathbf{meta}(x) \;\wedge \bigwedge_{\substack{1\leq i\leq k \\ a\in\Sigma, q\in Q}} [\boldsymbol{q \in Q_i^a}](x) \;\leftrightarrow\; \exists y.(\boldsymbol{R}(y) \;\wedge\; x\boldsymbol{q}y \;\wedge\; y\boldsymbol{r}^{\uparrow}_{\epsilon^* a} x_i)$$

□

Lemma 5.18. *Let $\mathcal{A}$ be an n-CPDA with stack-alphabet Γ. For every quanfitifer free formula $\phi(x_1, \ldots, x_k)$ in **FO** and configurations $(q_1, s_1), \ldots, (q_k, s_k)$ in $\mathcal{G}^\epsilon(\mathcal{A})$:*

$$\mathcal{G}^\epsilon(\mathcal{A}) \;\vDash\; \phi((q_1, s_1), \ldots, (q_k, s_k)) \quad \Leftrightarrow \quad \mathcal{G}^\epsilon(\mathbf{lum}(\mathcal{A})_k^+) \;\vDash\; \phi((q_1, t_1), \ldots, (q_k, t_k))$$

whenever $(q_1, t_1), \ldots, (q_k, t_k)$ are consistent reachable configurations of $\mathbf{lum}(\mathcal{A})_k^+$ and $\pi_\Gamma(t_i) = s_i$ for each $1 \leq i \leq k$.

Moreover for every set of $\mathcal{A}$ configurations $(q_1, s_1), \ldots, (q_k, s_k)$ there exists a consistent set of reachable configurations $(q_1, t_1), \ldots, (q_k, t_k)$ of $\mathbf{lum}(\mathcal{A})_k^+$ such that $\pi_\Gamma(t_i) = s_i$ for each $1 \leq i \leq k$.

Proof. For the first part argue by induction on the structure of ϕ. For the base case note that unary predicates are inherited directly from $\mathcal{A}$ and the fact we are considering ϵ-closure ensures that binary relations are also directly inherited, despite the additional steps of maintaining meta-annotations. For equality we must appeal to consistency. The Q_i^a component of any meta-annotation m containined in one of the t_j is uniquely determined by (q_i, s_i) (or indeed by (q_i, t_i)) and $t_{j_{\leq m}}$ by definition of consistency. Thus for any $1 \leq i, j \leq k$ we will have $(q_i, t_i) = (q_j, t_j)$ just in case $s_i = s_j$.

Conjunction and negation are straightforward applications of the induction hypothesis.

The second part is immediate from the fact that we just need to choose Q_i^a for each meta-annotation to be the unique set specified by (q_i, s_i). □

Lemma 5.19. *Let $\mathcal{A}$ be an n-CPDA. For every Σ_1 sentence ϕ in **FO** we can construct an MSO sentence $\hat{\phi}$ such that:*

$$\mathcal{G}^\epsilon(\mathcal{A}) \models \phi \quad \Leftrightarrow \quad \mathcal{G}(\mathsf{GrStripln}_{\mathcal{A}}(k)) \models \hat{\phi}$$

Proof. Without loss of generality (due to prenex normal form) let us assume that $\phi = \exists x_1.\exists x_2.\cdots\exists x_k.\phi'(x_1,\ldots,x_k)$ where ϕ' is quantifier free. For each reachable configuration (q,s) of $\mathsf{lum}(\mathcal{A})_k^+$ let $(q, L(s))$ be the corresponding reachable configuration of $\mathsf{GrStripln}_{\mathcal{A}}(k)$. By Lemma 5.18 it suffices to construct a quantifier free MSO formula $\hat{\phi}'(x_1,\ldots,x_k)$ such that for *consistent* $(q_1,s_1),\ldots,(q_k,s_k)$:

$$\mathcal{G}^\epsilon(\mathsf{lum}(\mathcal{A})_k^+) \models \phi'((q_1,s_1),\ldots,(q_k,s_k)) \quad \Leftrightarrow \\ \mathcal{G}(\mathsf{GrStripln}_k(\mathcal{A})) \models \hat{\phi}'((q_1,L(s_1)),\ldots,(q_k,L(s_k)))$$

We can then take

$$\hat{\phi} := \exists x_1\cdots x_k \left(\bigwedge_{i=1}^{k} \boldsymbol{R}(x_i) \wedge \mathsf{con}(x_1,\ldots,x_k) \wedge \hat{\phi}'(x_1,\ldots,x_k) \right)$$

where con is the MSO formula taken from Lemma 5.17 and $\boldsymbol{R}$ is the reachability predicate taken from the proof of Lemma 5.17.

We define $\hat{\phi}'$ by induction on the structure of ϕ'. The atomic cases of equality and unary predicates can have $\hat{\phi}' = \phi'$ due to the strong isomorphism between $\mathsf{lum}(\mathcal{A})_k^+$ and $\mathsf{GrStripln}_k(\mathcal{A})^-$ together with the fact that removing links does not affect equality for $\mathsf{lum}(\mathcal{A})_k^+$. Binary relations must be given a more sophisticated translation since some edges are removed in forming $\mathsf{GrStripln}_k(\mathcal{A})$. We therefore appeal to Lemma 5.7 telling us that $(q,s)\boldsymbol{a}(q',s')$ in $\mathcal{G}^\epsilon(\mathsf{lum}(\mathcal{A})_k^+)$ just in case $(q,L(s))\boldsymbol{b}_a(q',L(s'))$ in $\mathsf{GrStripln}_k(\mathcal{A})^-$. When $\phi'(x,y) = x\boldsymbol{a}x_i$ (so $y = x_i$—every variable must be of the form x_j and the i is significant here for meta-annotations) we thus express the existence of such a bounce by taking:

$$\hat{\phi}'(x,x_i) := \bigvee_{q\in Q} \bigwedge_{m\in M_{q,a}^i} \mathsf{Met}(q,m)^{\downarrow}(x)$$

where Q is the set of control-states of $\mathcal{A}$ and $M_{q,a}^i$ is the set of meta-annotations $((Q_1^a)_{a\in\Sigma},\ldots,(Q_k^a)_{a\in\Sigma})$ such that $q \in Q_i^a$.

Negation and conjunction is a trivial application of the induction hypothesis. □

Remark 5.20. The method behind the proof of Lemma 5.19 does not generalise to sentences with quantifier alternation since consistency requires a k-tuple of

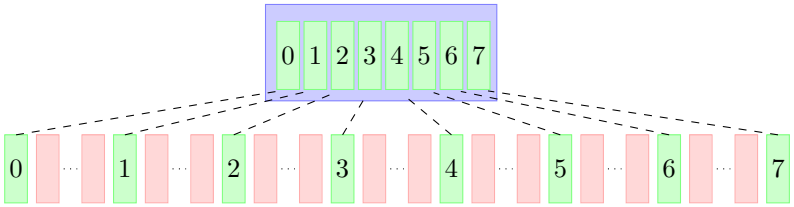


Figure 5.3: Simulating stacks with runs

stacks to be fixed. If one fixes a stack with an existential quantification and then endeavours to add a universal quantification, there may be some stack over which the universal quantifier ranges that does not honour the information in the meta-annotations embedded in the fixed (existentially quantified) stack.

Since $\mathsf{GrStripln}_k(\mathcal{A})$ is an n-PDA it must be the case that $\mathcal{G}^\epsilon(\mathsf{GrStripln}_{\mathcal{A}}(k))$ has decidable MSO theory [27]. Lemma 5.19 therefore implies:

Theorem 5.21. *Let $\mathcal{A}$ be an n_n-CPDA. Then the Σ_1 theory of $\mathcal{G}^\epsilon(\mathcal{A})$ is decidable.*

We now look at extending this result to cover $n_{(n-1)}$-CPDA, although we only have a complete proof for 3_2-CPDA.

5.4 The Derivative of a CPDA

It is possible to simulate an n-stack by a subsequence of a *run* of an $(n-1)$-CPDA, which we call the *the derivative* of the original automaton. Consider the illustration in Figure 5.3. A monotonic n-CPDA can build up its stack one $(n-1)$-stack at a time, never destroying any of its constituent $(n-1)$-stacks. Such an n-stack is illustrated at the top of the figure. After the automaton has constructed one $(n-1)$-stack it will perform a $push_n$ and move on to construct the next. A series of $(n-1)$-stack operations follow in order to construct this next stack prior to performing another $push_n$. In this respect, the construction of the n-stack can be viewed as a run, illustrated at the bottom of the figure. The red stacks represent the $(n-1)$-stacks produced whilst generating the constituent green coloured stacks. Of course, an associated run may not be unique as there may be multiple ways of generating each green stack from its predecessor—*i.e.* the intervening red stacks are not uniquely determined.

Definition 5.22. Let $\mathcal{A}$ be an n-CPDA. The *derivative* $\partial(\mathcal{A})$ of $\mathcal{A}$ is the $(n-1)$-CPDA that shares the same stack alphabet and set of control-states as $\mathcal{A}^\uparrow$ but removes all transitions performing an order-n operation (including *collapse* on n-links). Only those labelled r_ϵ performing a $push_n$ operation are replaced with a transition bearing a fresh label r^p and performing *nop*.

Example 5.23. Consider a 2-CPDA transition:

$$(q, [\,[\,abaabc\,]\,[\,abaac\,]\,]) \xrightarrow{r_\epsilon . r_\epsilon^m} (q', [\,[\,abaabc\,]\,[\,abaac\,]\,[\,abacca\,]\,])$$

where the first r_ϵ performs a $push_n$ and the subsequent ones order-$(n-1)$ operations. In the derivative this transition corresponds to:

$$(q, [\,abaac\,]) \xrightarrow{r^p . r_\epsilon^m} (q', [\,abacca\,])$$

Recall that $\mathbf{stripln}(s)$ for an n-stack s is just s with its n-links removed. The significance of the derivative is summarised by the following lemma:

Lemma 5.24. *Let $\mathcal{A}$ be an n-CPDA with initial control-state q_0. Let L be the strong isomorphism relating $\mathcal{A}$ to $\mathcal{A}^\uparrow$. Then $\mathcal{A}$ can reach a configuration (q, s) with $\mathbf{stripln}(s) = [s_1, \ldots, s_m]$ iff there exist control-states $q_1, \ldots, q_{m-1}$ such that*

$$(q_0, L(\bot_n))\boldsymbol{r}_{(r_\epsilon+\Sigma)^*}(q_1, L(s_1))\boldsymbol{r}_{r^p(r_\epsilon+\Sigma)^*} \cdots \boldsymbol{r}_{r^p(r_\epsilon+\Sigma)^*}(q_{m-1}, L(s_{m-1}))$$
$$\boldsymbol{r}_{r^p(r_\epsilon+\Sigma)^*\Sigma}(q_m, L(s_m))$$

Proof. This is just a definition chase (recalling that Σ-edges are assumed to be order-$(n-1)$ operations). In particular the monotonicity of $\mathcal{A}^\uparrow$ demands that all stacks can be constructed without performing pop_n or *collapse* on n-links. □

The Annotated Derivative

We wish to add meta-annotations to arbitrary n-stacks and express consistency in terms of derivatives. Since consistency is defined in terms of k-tuples of configurations, this needs to be paramaterised by k. This is the derivative equivalent of $\mathbf{lum}(\mathcal{A})_k^+$—it can non-deterministically select any meta-annotations that it likes; correctness will be imposed externally.

In addition to our assumption that Σ-transitions only perform order-$(n-1)$ operations, it is convenient to assume w.l.o.g. that the only stack operation that $\mathcal{A}$ can perform in its starting configuration is $push_n$.

Definition 5.25. Let $\mathcal{A}$ be an n-CPDA with edges Σ and control-states Q. Let $\mathcal{A}_k^+$ be $\mathcal{A}$ adapted to keep a non-deterministically chosen meta-annotation $((Q_1^a)_{a\in\Sigma}, \ldots, (Q_k^a)_{a\in\Sigma})$ on top of each stack (using ϵ-transitions). Predicates $\mathbf{metaConfig}(m)$ and $\mathbf{metaConfig}(q, m)$ are added for each $q \in Q$ and meta-configuration m that hold in a configuration with control-state q and meta-configuration m on top of the stack. The *k-annotated derivative* $\partial_k(\mathcal{A})$ is $\partial(\mathcal{A}_k^{+\uparrow\downarrow})$ where we add reachable configurations of the form:

$$(P, I, b, ((Q_1^a)_{a\in\Sigma}, \ldots, (Q_k^a)_{a\in\Sigma}), s)$$

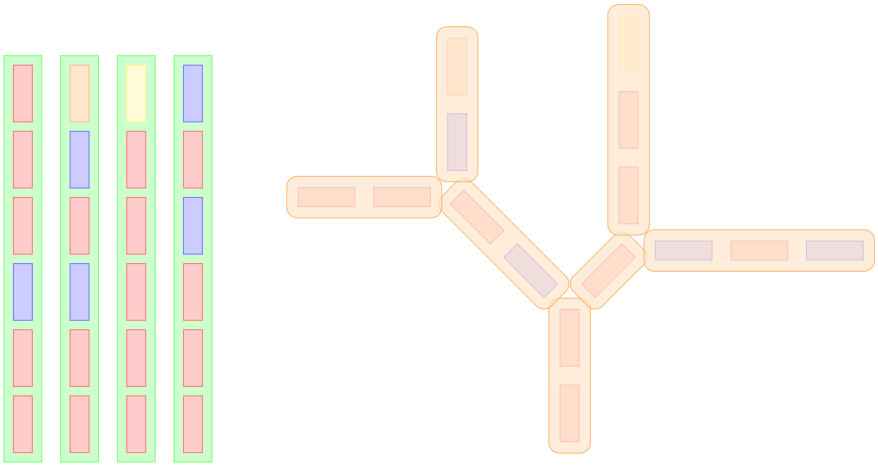


Figure 5.4: The derivative tree of four n-stacks.

where s is a stack reachable by $\partial_k(\mathcal{A})$ and $(P, I, b, ((Q_1^a)_{a\in\Sigma}, \ldots, (Q_k^a)_{a\in\Sigma}))$ is an additional 'control-state' where $P \subseteq Q$, $I \subseteq [1..k]$, $b \in \mathbb{B}$ and each $Q_i^a \subseteq Q$ so that $((Q_1^a)_{a\in\Sigma}, \ldots, (Q_k^a)_{a\in\Sigma})$ is a meta-annotation for $\mathcal{A}$. These additional configurations are referred to as *meta-configurations*. An edge **stackEq** is attached between every pair of configurations sharing the same stack. This facilitates stack comparison. We also add additional control-states (absorbed into ϵ-closure) and edges $\textbf{reachTest}_{(q,q',a)}$ and $\textbf{reachTest}_{(q,q',a^P)}$ for each $q, q' \in Q$ and $a \in \Sigma \cup \{\epsilon\}$ so that we have an $\epsilon^*\textbf{reachTest}_{(q,q',a)}$-path or an $\epsilon^*\textbf{reachTest}_{(q,q',a^P)}$-path from a configuration (p, s) to (p', s) just in case there is respectively an $r_\epsilon^* a$-path or $r^P r_\epsilon^* a$-path from (q, s) to (q', s) in $\partial(\mathcal{A}_k^{+\uparrow\downarrow})$.

The intuition is that $(P, I, b, ((Q_1^a)_{a\in\Sigma}, \ldots, (Q_k^a)_{a\in\Sigma}), s)$ represents an $(n-1)$-stack within an n-stack t of $\mathcal{A}$ where P contains precisely those control-states in which we may be after constructing $t_{\leq s}$, and $((Q_1^a)_{a\in\Sigma}, \ldots, (Q_k^a)_{a\in\Sigma})$ is the meta-annotation to be assigned to the top of s. The set I tracks which of the k stacks we are currently representing and the flag b is set to $\mathbf{t}$ just in case we are representing the *top* $(n-1)$-stack of one of the k stacks.

Our aim is to arrange some meta-configurations of $\partial_k(\mathcal{A})$ into a tree that represents a k-tuple of $\mathcal{A}$-configurations whose stacks are annotated with meta-configurations in a consistent manner. This tree representation will branch at a point where the $\mathcal{A}$-stacks contain $(n-1)$ stacks that differ. We illustrate this in Figure 5.4. The orange regions in the figure show regions where the stacks have $(n-1)$-stacks in common. Note that for a bounded number of $\mathcal{A}$-stacks the number of these orange regions will be bounded.

The following defines the child relations of the tree:

Definition 5.26. Let $\mathcal{A}$ be an n-CPDA with control-states Q. For each $1 \leq l \leq k$ we define the $(l+1)$-ary relation $\boldsymbol{\partial_l}$ between meta-configurations of $\partial_k(\mathcal{A})$ where

$$(P, I, \mathbf{f}, ((Q^a_1)_{a\in\Sigma}, \ldots, (Q^a_k)_{a\in\Sigma}), s) \; \boldsymbol{\partial_l} \; (P_j, I_j, b_j, ((Q^a_{1\,j})_{a\in\Sigma}, \ldots, (Q^a_{k\,j})_{a\in\Sigma}), s_j)_{1\leq j\leq l}$$

just in case:

1. $P_j = \{\ p' \ :\ (p,s)\boldsymbol{r}_{r^p r^*_\epsilon}(p', s_j) \text{ for } p \in P\ \}$, where reachability is interpreted in $\partial_k(\mathcal{A})$.
2. $I_j \cap I_{j'} = \emptyset$ for every $1 \leq j < j' \leq l$.
3. $\bigcup_{j=1}^{l} I_j = I$.
4. If $b_j = \mathbf{t}$, then $|I_j| = 1$ with say $I_j = \{\ \check{j}\ \}$ and $Q^a_{\check{j}\,j} = \{\ q_{\check{j}}\ \}$ for some $q_{\check{j}} \in P_j$ for every $a \in \Sigma$. The jth child would then represent the $\check{j}$th configuration in the k-tuple and the represented configuration would have control-state $q_{\check{j}}$.
5. For every $1 \leq j < j' \leq l$ such that $b_j = b_{j'} = \mathbf{t}$, either $s_j \neq s_{j'}$ or $Q^a_{\check{j}\,j} \neq Q^a_{\check{j}'\,j'}$ for every $a \in \Sigma$. (Ensures that every path in the stack represents a distinct configuration—*i.e.* differs in at least one of stack or control-state.)
6. $s_j \neq s_{j'}$ for $1 \leq j < j' \leq l$ where $b_j = b_{j'} = \mathbf{f}$.
7. For each $i \in I$, $a \in \Sigma$ and for every $1 \leq j' \leq l$ such that either $b_{j'} \neq \mathbf{t}$ or $b_{j'} = \mathbf{t}$ but $i \neq \check{j}'$ (*i.e.* excluding that covered by item 4):

$$Q^a_{i\,j'} = \begin{cases} \{\ q \in Q \ :\ (q, s_{j'})\boldsymbol{r}_{r^*_\epsilon a}(q_{\check{j}}, s_j)\ \} & \text{where there exists } j \in [1..l] \\ & \text{s.t. } b_j = \mathbf{t} \text{ and } i = \check{j} \\ \bigcup_{\substack{j\in[1..l] \\ q'\in Q^a_{i\,j}}} \{\ q \in Q \ :\ (q, s_{j'})\boldsymbol{r}_{r^*_\epsilon}(q', s_j)\ \} & \text{otherwise} \end{cases}$$

$$Q^a_i = \begin{cases} \{\ q \in Q \ :\ \exists p \in Q.(q,s)\boldsymbol{r}_{r^*_\epsilon}(p,s) \text{ and } (p,s)\boldsymbol{r}_{r_p.r^*_\epsilon a}(q_{\check{j}}, s_j)\ \} & \text{where there exists } j \in [1..l] \\ & \text{s.t. } b_j = \mathbf{t} \text{ and } i = \check{j} \\ \bigcup_{\substack{j\in[1..l] \\ q'\in Q^a_{i\,j}}} \{\ q \in Q \ :\ \exists p \in Q.(p,s)\boldsymbol{r}_{r^*_\epsilon}(p,s) \text{ and } (q,s)\boldsymbol{r}_{r_p.r^*_\epsilon}(q', s_j)\ \} & \text{otherwise} \end{cases}$$

8. If $i \notin I$ then $Q^a_{i\,j} = \emptyset$ for all $1 \leq j \leq l$ and $a \in \Sigma$.
9. For every $1 \leq j \leq l$ the jth configuration must satisfy $\textsf{metaConfig}((Q^a_{1\,j})_{a\in\Sigma}, \ldots, (Q^a_{k\,j})_{a\in\Sigma})$. This ensures the meta-configuration is reflected in the stack structure so that it can be refered to by the $^\downarrow$ predicates.

Definition 5.27. A k-derivative tree *root* for an n-CPDA $\mathcal{A}$ and $k \in \mathbb{N}$ is the meta-configuration $(\{q_0\}, [1..k], \mathbf{f}, M, [M])$ for any meta-annotation M, where q_0 is the initial control-state of $\partial_k(\mathcal{A})$. We say that a meta-configuration M is a *derivative leaf* just in case its Boolean flag is $\mathbf{t}$.

We are now in a position to formally define the k-derivative-tree of an $n_{n-1..2}$-CPDA $\mathcal{A}$, which is the tree illustrated in Figure 5.4.

Definition 5.28. Let $\mathcal{A}$ be an $n_{(n-1)..2}$-CPDA with stack-alphabet Γ and control-states Q and let $k \in \mathbb{N}$. A *k-derivative tree* of $\mathcal{A}$ is a tree of meta-configurations with the k-derivative tree root at the root of the tree; a node m has children $m_1, m_2, \ldots, m_l$ just in case $m\boldsymbol{\partial_l} m_1 m_2 \cdots m_l$; the leaves of the tree are derivative leaves.

Let $\mathcal{A}_k^+$ be as in Definition 5.25. Let Γ^+ be Γ extended with meta-annotations for $\mathcal{A}$ (*i.e.* the stack-alphabet of $\mathcal{A}_k^+$). We say that the $\mathcal{A}_k^+$-configuration *derived from a branch* $c_0, c_1, \ldots, c_r$ of a k-derivative tree of $\mathcal{A}$ (where c_0 is the root and c_r is a leaf) is the configuration of $\mathcal{A}^+$ of the form (q, s) where $s = \pi_{\Gamma^+}([s_0 s_1 \cdots s_r])$ such that:

- The $(n-1)$-stack s_j is the stack associated with the meta-configuration c_j for each $1 \leq j \leq r$.
- q is the unique control-state of $\mathcal{A}$ such that $Q_i^a = \{q\}$ in c_r where $I = \{i\}$ in c_r.

Remark 5.29. The restriction on leaves of k-derivative trees ensures that a k-derivative tree has precisely k branches. This is because the root of the tree has $I = [1..k]$ and the leaves of the tree have $|I| = 1$ with disjoint union equalling the I at the root.

We adapt our previous definition of consistency (Definition 5.16) to our current setting.

Definition 5.30. Let $(q_1, s_1), \ldots, (q_k, s_k)$ be reachable configurations of $\mathcal{A}_k^+$. Then we say that this k-tuple of configurations is *consistent* just in case the following conditions are met:

- For each i with $1 \leq i \leq k$ it is the case that each $(n-1)$-stack in s_i contains precisely one meta-annotation which occurs on top of it.
- Suppose that $t \sqsubseteq_n s_i$ for some $1 \leq i \leq k$. Then the meta-annotation on top of $top_{n-1}(t)$ must be equal to $((Q_1^a)_{a\in\Sigma}, \ldots, (Q_k^a)_{a\in\Sigma})$ where:

$$Q_i^a := \{\ q \in Q\ :\ (q,t)\boldsymbol{r}^{\uparrow}_{\epsilon^* a}(q_i, s_i)\ \}$$

The following Lemma characterises consistent configurations of $\mathcal{A}^+$ in terms of derivative trees of $\partial_k(\mathcal{A})$. The reader concerned about getting lost amongst

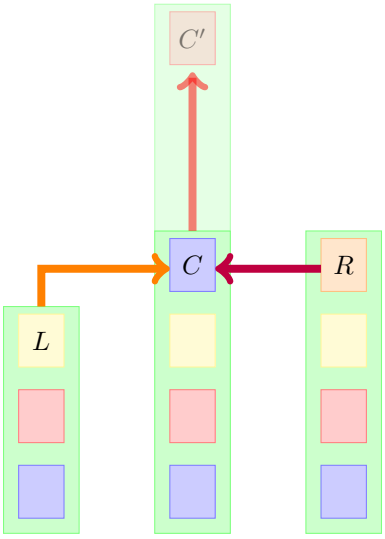


Figure 5.5: Why a derivative tree correctly represents consistent configurations: In order to go from L to C' monotonically, it is necessary to pass through C, corresponding in the derivative to $r^p r_\epsilon^*$ (preceded by r_ϵ^* from L back to itself). Likewise for R to get to C' it is necessary for it to pass through C, corresponding to r_ϵ^* in the derivative. Noting that top_n stacks of R and C are siblings in derivative tree and top_n of L the parent, we can see how rule 7 of the tree relation locally propagates global reachability.

subscripts might actually find Figure 5.5 more readable than the proof! This illustrates how the tree relation propagates global reachability in a local manner, which is the reason why the Lemma holds.

Lemma 5.31. *Let $\mathcal{A}$ be an $n_{(n-1)..2}$-CPDA. Then there exist* distinct *(pairwise unequal) reachable consistent configurations $(q_1, s_1), \ldots, (q_k, s_k)$ of $\mathcal{A}^+$ if and only if there exists a k-derivative tree of $\mathcal{A}$ with the configurations derived from its branches being precisely $(q_1, s_1), \ldots, (q_k, s_k)$.*

Proof. Let L be the map from $\mathcal{A}_k^+$-stacks to $\mathcal{A}_k^{+\uparrow\downarrow}$-stacks.

First suppose that there exist *distinct* reachable consistent configurations $(q_1, s_1), \ldots, (q_k, s_k)$ of $\mathcal{A}^+$. We now define the required k-derivative tree and simultaneousy establish by induction on the structure of the tree that it has the required property.

Take the root with stack:

$$[\ ((Q_1^a)_{a\in\Sigma}, \ldots, (Q_k^a)_{a\in\Sigma})\ ]$$

where $Q_i^a := \{\ q \in Q_{\mathcal{A}_k^+} : (q, L([\,]))\boldsymbol{r}^{\uparrow}_{\epsilon^*\,a}(q_i, L(s_i))\ \}$. Since we are making the w.l.o.g. assumption that all automata begin with a $push_n$ operation and

since $(q_1, s_1), \ldots, (q_k, s_k)$ are consistent it must be the case that this $(n-1)$-stack is also the bottom stack of each of the $s_1, s_2, \ldots, s_k$. Now suppose that we have already constructed a node

$$(P, I, b, ((Q_1^a)_{a\in\Sigma}, \ldots, (Q_k^a)_{a\in\Sigma}), s)$$

of the tree. As an induction hypothesis we assume the following:

1. Let h be the length of the path in the tree starting at the root and ending at this node. Let t_{hi} be the hth $(n-1)$-stack in s_i for each $i \in [1..k]$. Then there exists $\hat{i} \in [1..k]$ such that $s = top_n(L(s_{\hat{i}})_{\leq t_{h\hat{i}}})$.

2. Based on this $\hat{i}$:
$$b = \begin{cases} \mathbf{f} & \text{if } |s_{\hat{i}}|_n > h \\ \mathbf{t} & \text{if } |s_{\hat{i}}|_n = h \end{cases}$$

3. Then we have:
$$I = \begin{cases} \{\ i \in [1..k]\ :\ s_{i\leq t_{hi}} = s_{\hat{i}\leq t_{h\hat{i}}} \text{ and } |s_i|_n > h\ \} & \text{if } b = \mathbf{f} \\ s = s_{\hat{i}} & \text{if } b = \mathbf{t} \end{cases}$$

4. $P = \{\ p \in Q\ :\ (p, L(s_{\hat{i}\leq t_{h\hat{i}}})) \in \boldsymbol{R}(\mathcal{A}_k^{+\uparrow\downarrow})\ \}$.

5. If $b = \mathbf{f}$, then for every i with $1 \leq i \leq k$ we have:
$$Q_i^a = \{\ q \in Q\ :\ (q, L(s_{\leq t_h}))\boldsymbol{r}_{r_\epsilon^* a}(q_i, L(s_i))\ \}$$
with reachability interpreted in $\mathcal{A}^{+\uparrow\downarrow}$.

6. If $b = \mathbf{t}$ with $I = \{\ \hat{i}\ \}$, then for every $a \in \Sigma$ we have:
$$Q_{\hat{i}}^a = \{\ q_{\hat{i}}\ \}$$
and for $i \neq \hat{i}$ where $1 \leq i \leq k$:
$$Q_i^a = \{\ q \in Q\ :\ (q, L(s_{\leq t_h}))\boldsymbol{r}_{r_\epsilon^* a}(q_i, L(s_i))\ \}$$

By construction the root satisfies all of these (and in particular $b = \mathbf{f}$ at the root by definition). If $b = \mathbf{t}$ then we need not do anything as such a node has no children. So just consider the case when $b = \mathbf{f}$. Inducting on h let

$$\{\ s'_1, s'_2, \ldots, s'_l\ \} := \{\ top_n(L(s_{i\leq t_{h+1}i}))\ :\ i \in I \text{ and } |s_i|_n > (h+1)\ \}$$

where the s'_j are pairwise distinct for $j \in [1..l]$. Further define:

$$\{\ s'_{l+1}, \ldots, s'_{l+l'}\ \} := \{\ top_n(L(s_i))\ :\ i \in I \text{ and } |s_i|_n = (h+1)\ \}$$

where we view the above as an equality between *multi-sets* and so the s'_j are *not* necessarily pairwise distinct for $j \in [l+1..l+l']$. We then specify $l+l'$ children:

$$(P_j, I_j, b_j, ((Q^a_{1\,j})_{a\in\Sigma}, \ldots, (Q^a_{k\,j})_{a\in\Sigma}), s'_j)$$

where $j \in [1..(l+l')]$ with:

$$b_j := \begin{cases} \mathbf{f} & \text{if } j \in [1..l] \\ \mathbf{t} & \text{if } j \in [(l+1)..(l+l')] \end{cases}$$

and

$$I_j := \begin{cases} \{\ i \in I\ :\ s_j = top_n(s_{i \le t_{h+1} i})\ \} & \text{if } j \in [1..l] \\ \{\ \min(\{i \in I\ :\ s_j = top_n(s_{i \le t_{h+1} i}), & \\ \qquad |s_j|_n = (h+1), i \notin s_{j'} \text{ for } j' < j\})\ \} & \text{if } j \in [(l+1)..(l+l')] \end{cases}$$

Based on the definition we must have satisfaction of the first three elements of the induction hypothesis. We can then ensure satisfaction of the remaining three elements by reading them as definitions for the P_j, and $Q^a_{i\,j}$.

Let $\hat{i_j}$ be a representative of I_j. It just remains to check that:

$$(P, I, b, ((Q^a_1)_{a\in\Sigma}, \ldots, (Q^a_k)_{a\in\Sigma}), s)\ \boldsymbol{\partial_l}\ (P_j, I_j, b_j, ((Q^a_{1\,j})_{a\in\Sigma}, \ldots, (Q^a_{k\,j})_{a\in\Sigma}), s'_j)_{j\in[1..l]}$$

We work through the criteria for $\boldsymbol{\partial_l}$ in the same order as presented in the definition:

1. Recall that $\boldsymbol{R}(\mathcal{A}_k^{+\uparrow\downarrow})$ is witnessed monotonically and so for $(q', u) \in \boldsymbol{R}(\mathcal{A}_k^{+\uparrow\downarrow})$ there must be some $(q, pop_n(u)) \in \boldsymbol{R}(\mathcal{A}_k^{+\uparrow\downarrow})$ such that $(q, top_n(pop_n(u)))\boldsymbol{r}_{r^p\ r^*_\epsilon}(q', top_n(u))$ in the derivative and conversely if for $(q, pop_n(u)) \in \boldsymbol{R}(\mathcal{A}_k^{+\uparrow\downarrow})$ we have $(q, top_n(pop_n(u)))\boldsymbol{r}_{r^p\ r^*_\epsilon}(q', top_n(u))$ in the derivative it must be the case that $(q', u) \in \boldsymbol{R}(\mathcal{A}_k^{+\uparrow\downarrow})$.

2. This is ensured by the pairwise distinction between the s'_j for $j \in [1..l]$ (for which $b_j = \mathbf{f}$) and the pairwise distinction for $j \in [(l+1)..l']$ (for which $b_j = \mathbf{t}$) combined with the fact that the $I_j \neq I_{j'}$ whenever $b_j \neq b_{j'}$.

3. We are assured that all $i \in I$ such that $i \in [l+1..l+l']$ (so that $b_i = \mathbf{t}$) are covered due to the fact that we are considering multi-sets. We also cover $i \in [1..l]$ due to the definition of I_j when $b_j = \mathbf{f}$ subsuming equal stacks.

4. The facts that $|I_j| = 1$ with $I_j = \{\ \check{j}\ \}$ and $|Q^a_{\hat{j}\,j}| = 1$ are immediate from the definition. The requirement that $q \in P$ where $Q^a_{j\,j} = \{\ q\ \}$ must be satisfied by the assumption that $(q_{\hat{i_j}}, s_{\hat{i_j}})$ is reachable by $\mathcal{A}_k^+$ where $I_j = \{\ \hat{i_j}\ \}$

5. This is ensured by the assumption that the configurations (q_i, s_i) are pairwise distinct.

6. This is ensured by the imposition of pairwise distinction on the s'_j for $j \in [1..l]$.

7. By the induction hypothesis we have it that for every i, $(q, L(s_{\hat{i} \leq t_{h\hat{i}}}))\boldsymbol{r}_{r_\epsilon^* a}(q_{i'}, L(s_{i'}))$ iff $q \in Q_i^a$. Suppose that there is a $j \in [(l+1)..l']$ such that $b_j = \mathbf{t}$ with $\check{j} = i$. It must then be that $s'_j = top_n(L(s_i))$ and that $s = top_n(pop_n(L(s_i)))$. Thus monotonicity means that the derivative must be able to go from a configuration (q, s) to some configuration (p, s) via an r_ϵ^*-path leading onto configuration $(q_{\check{j}}, s'_j)$ via an $r^p.r_\epsilon^* a$ labelled path, just in case $(q, L(s_{\hat{i} \leq t_{h\hat{i}}}))\boldsymbol{r}_{r_\epsilon^* a}(q_{i'}, L(s_{i'}))$, as required.

   Suppose now that there is no such j. This means that there must be $j \in [1..l]$ such that $b_j = \mathbf{f}$ and so $|s_{\hat{i_j}}|_n > (h+1)$. Thus for any $i \in I_j$ in order to construct $L(s_i)$ it must be necessary in the monotonic automaton to pass through $L(s_{\hat{i_j}})$ from s, beginning with a $push_n$ and performing a subsequent $push_n$. Thus the initial $push_n$ and order-$(n-1)$ operations must correspond to ϵ-transitions. This corresponds in the derivative to a path of the form $r_p.r_\epsilon^*$. Moreover, this path must end up in a control-state q such that $(q, L(s_{\hat{i_j}}))\boldsymbol{r}_{r_\epsilon^* a}(q_i, L(s_i))$. Thus Q_i^a satisfies the second equality on part 7 of the $\boldsymbol{\partial_l}$ definition.

   Similiar considerations ensure that the $Q^a_{i\,j}$ satisfy the first equality of part 7; since we are considering siblings here (corresponding to $(n-1)$-stacks at the same height as an n-stack) no $push_n$ need be considered due to monotonicity.

8. If $i \neq I$, then by the induction hypothesis $pop_n(s_{j' t_{h+1\,j'}}) \not\sqsubseteq_n s_i$ for any $j' \in I_j$ with $j \in [1..l..l+l']$. Thus monotonic reachability from a configuration with stack $L(s_{j' t_{h+1\,j'}})$ to one with stack $L(s_i)$ is impossible and thus $Q^a_{i\,j'}$ will indeed be empty, as required.

9. This follows from what it means for the (q_i, s_i) to be consistent and the value of the $Q^a_{i\,j}$ that we have set in the induction hypothesis in the light of Lemma 5.2.

Note that the only leaves to which we do not add children have $b_i = \mathbf{t}$ and so the leaves of the tree we have defined are as required by a derivative tree. The tree we have defined is thus indeed a k-derivative tree for $\mathcal{A}$. Note further that the construction of the tree ensures that the configurations derived from the branches are precisely those required.

For the converse direction suppose that we have we have a k-derivative tree D of $\mathcal{A}$, which must have precisely k branches (Remark 5.29). Let (q_1, s_1),

$(q_2, s_2), \ldots, (q_k, s_k)$ be the configurations derived from these branches. First we argue that they are distinct. Suppose for contradiction that they are not—let us say that $(q, s) = (q', s')$ is an equal pair. Then there must be some node b in D that have two children with the same stack. The $\boldsymbol{\partial_{\hat{l}}}$ relation (for any $\hat{l}$) allows this only when one of the following holds:

- One of the children has the b flag set to $\mathbf{t}$ and the other set to $\mathbf{f}$. But then one is a leaf and the other is not, which means that $|s| \neq |s'|$ and so $s \neq s'$, a contradiction.
- Both of the children have the b flag set to $\mathbf{t}$ (so are both leaves) but then $q \neq q'$, which again is a contradiction.

So the configurations derived form branches of D must be distinct from each other. It just remains to show that they are both reachable and consistent.

For consistency it is sufficient to check that the following two properties hold of each node

$$(P, I, b, ((Q_1^a)_{a\in\Sigma}, \ldots, (Q_k^a)_{a\in\Sigma}), s)$$

in the k-derivative tree D:

1. Let $\hat{i} \in I$ be any representative of I. Let $I' = \{\ i' \in [1..k]\ :\ L(s_{i'})_{<s} = L(s_{\hat{i}})_{<s}\ \}$. If $b = \mathbf{f}$, then for every $i' \in I'$ we have:

   $$Q_{i'}^a = \{\ q \in Q\ :\ (q, L(s_{\hat{i}\leq s}))\boldsymbol{r}_{r_\epsilon^* a}(q_{i'}, L(s_{i'}))\ \} \text{ for a representative } \hat{i} \in I$$

   with reachability interpreted in $\mathcal{A}^{+\uparrow\downarrow}$.

2. If $b = \mathbf{t}$, then $I = \{\ i\ \}$ for some $1 \leq i \leq k$ and for every $a \in \Sigma$ we have:

   $$Q_i^a = \{\ q_i\ \}$$

   and for $i \neq i'$ where $i' \in I'$:

   $$Q_{i'}^a = \{\ q \in Q\ :\ (q, L(s_{\hat{i}\leq s}))\boldsymbol{r}_{r_\epsilon^* a}(q_{i'}, L(s_{i'}))\ \} \text{ for a representative } \hat{i} \in I$$

Combined with the fact that the $Q_{i'}^a$ from the meta-configuration must also be those on top of the meta-configuration's $(n-1)$-stack is precisely the condition required for consistency.

We argue by reverse induction on the distance of nodes in D from the root, starting with the leaves and working towards the root. Note that $(q, L(s_{\hat{i}})_{\leq s})\boldsymbol{r}_{r_\epsilon^* a}(q_{i'}, L(s_{i'}))$ implies that $pop_n(L(s_{\hat{i}})_{\leq s})) = pop_n(L(s_{i'})) \sqsubset_n pop_n(u)$ for all stacks u in the run witnessing reachability, due to the monotonicity requirement.

The 7th rule for the $\boldsymbol{\partial_l}$ relation ensures that any node defining a branch from the root deriving an n-stack t is a sibling of any node deriving an n-stack t' such that $pop_n(t) = pop_n(t')$.

It follows that the leaves (which must have $b = \mathbf{t}$) which are the furthest from the root must satisfy the properties due to the first equality of the 7th rule for $\boldsymbol{\partial_l}$. For other nodes that are not leaves, their Q_i^a for $i \in I$ will satisfy the properties by appealing to the 2nd equality of the 7th rule rules together with the induction hypothesis applied to the node's children.

For other nodes that are leaves we note that for $i \in I$ the properties hold by definition and that for $i \in I'$ we may appeal to the 1st equality of the 7th rule together with the paragraph above.

For reachability, it is sufficient (due to the 4th rule of $\boldsymbol{\partial_l}$) to check that for every node:

$$(P, I, b, ((Q_1^a)_{a\in\Sigma}, \ldots, (Q_k^a)_{a\in\Sigma}), s)$$

we have $P = \{\ q \in Q\ :\ (q, L(s_{\hat{i}\leq s})) \in \boldsymbol{R}(\mathcal{A}^{+\uparrow\downarrow})$. But this is easily seen by straightforward *forwards* induction on the structure of D, *starting at the root* and moving towards the leaves by appeal to the 1st rule. □

The Diagram of a Derivative Tree

Let us call a maximal chain of nodes each of which is the unique child of its parents *a log*[1].

Definition 5.32. Let S be a set of nodes in a k-derivative tree D. We say that S is a *partial log* just in case S forms a $\boldsymbol{\partial_1}$-chain and there is precisely one x in S such that x is either the root of D or x has a sibling in D. We say that S is a *log* just in case it is a maximal partial log. The $\boldsymbol{\partial_1}$-minimal element of a log (which must either be the root of D or a node with a sibling in D) is called *the base* of the log. The $\boldsymbol{\partial_1}$-maximal element of a log (which must either be a leaf of D or else the parent of more than one child) is called *the tip* of the log. The log containing the root of D is *the trunk* of the tree. A log containing a leaf (which must be the tip of that log) is called *a twig*.

As observed in Remark 5.29, a k-derivative tree has precisely k-branches. In particular this means that there is an upper bound on the number of logs such trees have and hence the number of ways in which logs may be arranged. We assign indices to the logs in a way that reflects their arrangement. The set of these indices, which we call the *diagram* of the tree thus specifies the overall arrangement of the logs.

Definition 5.33. The *log-index* $\mathsf{LInd}(S)$ of a log S of D is recursively defined by:

$$\mathsf{LInd}(S) := \begin{cases} \epsilon & \text{if } S \text{ is the trunk} \\ \mathsf{LInd}(S')i & \text{if the base of } S \text{ is the } i\text{th child of the tip of } S' \end{cases}$$

[1]This is intended in the sense of a component of a branch that one burns in the fire rather than the inverse of exponentiation!

The *diagram* of D is the set of indices of logs:

$$\mathbf{Diag}(D) \;:=\; \{\; \mathbf{LInd}(S) \;:\; S \text{ is a log of } D \;\}$$

Example 5.34. The logs in the illustration in Figure 5.4 are highlighted in orange. The diagram of this tree is: $\{\, \epsilon, 1, 11, 12, 21, 22 \,\}$.

We write $\mathbf{Diag}_k$ to denote the set of diagrams of k-derivative trees.

Lemma 5.35. $|\mathbf{Diag}_k| \leq 3.k$ *and so in particular is finite for each* $k \in \mathbb{N}$.

Proof. As observed in Remark 5.29 any k-derivative tree has at most k-branches. A new log is only created at a branching point in the tree. A single branching point results in the creation of at most three logs (the new branch together with the two new logs resulting from splitting a log in two at the point of branching—if a log splits into more than two parts just consider this to consist of multiple branching points each resulting in a split into two parts). Since there are at most k branching points, there can be no more than $3.k$ logs. □

As before, expressing consistency in a suitable logic will be key to the decidability result. In this case, consistency will be expressed via derivative trees, so we need to be able to define a derivative tree in a suitable logic.

Lemma 5.36. *Let* $\mathcal{A}$ *be an* $n_{(n-1)..2}$*-CPDA. Then for each* l *and* k *there exists a* Δ_1*-first-order formula* $\boldsymbol{\partial_l}(x, y_1, y_2, \ldots, y_l)$ *that defines* $\boldsymbol{\partial_l}$ *in* $\mathcal{G}^\epsilon(\partial_k(\mathcal{A}))$*. In the case when* $l = 1$ *this is a* Δ_0 *(quantifier free) first-order formula.*

Proof. Since there are only finitely many control-states associated with meta-configurations, we can simply construct a disjunction of all possible combinations of control-states in each of the $(l+1)$-meta-configurations and then express the conditions required by each combination according to the definition of $\boldsymbol{\partial_l}$. The relation **stackEq** allows for comparing stacks for equality and the relations $\mathbf{reachTest}_{(q,q',a)}$ and $\mathbf{reachTest}_{(q,q',a^P)}$ allow to express item 7. The following fragment from the formula for part of rule 7 illustrates how the whole thing is possible:

$$\bigwedge_{q \in Q^a_j} \left(\bigvee_{\substack{p \in Q, q \in (Q^a_j)_i \\ i \in [1..l]}} x\mathbf{reachTest}_{(q,p,r_\epsilon)} x \;\wedge\; x\mathbf{reachTest}_{(p,q',r^P)} y_i \right)$$

$$\bigwedge_{\substack{q \notin Q^a_j \text{ s.t. } j \in I_i \\ q' \in (Q^a_j)_i}} \neg \left(\bigvee_{\substack{p \in Q, q \in (Q^a_j)_i \\ i \in [1..l]}} x\mathbf{reachTest}_{(q,p,r_\epsilon)} x \;\wedge\; x\mathbf{reachTest}_{(p,q',r^P)} y_i \right)$$

where Q^a_i is part of the meta-annotation on the parent x and $(Q^a_j)_i$ part of the meta-annotation on the child y_i.

Note that the only use of quantifiers is when comparing stacks: both $\exists z.(x\mathbf{stackEq}z \wedge y\mathbf{stackEq}z)$ and $\forall z.(x\mathbf{stackEq}z \wedge y\mathbf{stackEq}z)$ capture 'x and y have the same stack'—this is thus Δ_1. Note that when $l = 1$ there is no need to compare any stacks (since stacks are only compared for siblings) and so in this case we have a Δ_0 formula. □

This enables us to Δ_1-define a k-derivative tree with any given diagram using $\boldsymbol{FO(TC[\Delta_0])}$ over $\partial_k(\mathcal{A})$.

Lemma 5.37. *Consider a k-derivative tree diagram Dia. Then there exists a Δ_1-$\boldsymbol{FO(TC[\Delta_0])}$-formula $\phi_{Dia}(x^t_{i_1}, x^b_{i_1}, \ldots, x^t_{i_m}, x^b_{i_m})$ such that $Dia = \{ i_1, \ldots, i_m \}$ and such that*

$$\mathcal{G}^\epsilon(\partial_k(\mathcal{A})) \models \phi(t_{i_1}, b_{i_1}, \ldots, t_{i_m}, b_{i_m})$$

just in case there exists a k-derivative tree D such that $\boldsymbol{Diag(D)} = Dia$ and every log S of D has tip $t_{\mathbf{Lind}(S)}$ and base $b_{\mathbf{Lind}(S)}$.

Proof. Let $x\mathbf{log}y$ be the relation asserting that there is a log with base x and tip y. From Lemma 5.36 in the case when $l = 1$ we know that $\mathbf{log}$ must be definable in $\mathcal{G}^\epsilon(\partial_k(\mathcal{A}))$ by a $\boldsymbol{FO(TC[\Delta_0])}$ formula with no quantifiers. In order to get this we take $x\overline{\boldsymbol{\partial_1}}y \vee x = y$. The $x = y$ covers the case when the log has just one element (and so the base and tip coincide).

We can then just take the conjunction of all formulae of the form:

$$x^b_{i_j}\mathbf{log}x^t_{i_j} \wedge x^t_{i_j}\boldsymbol{\partial_l}x^b_{i_j 1} \cdots x^b_{i_j l}$$

where the log with index i_j has l children. □

For any $k \in \mathbb{N}$ it is thus possible to assert using a Δ_1-$\boldsymbol{FO(TC[\Delta_0])}$ formula that $x_1, \ldots, x_k$ are the leaves of a k-derivative tree of $\mathcal{A}$.

Lemma 5.38. *Let $\mathcal{A}$ be an $n_{(n-1)..2}$-CPDA, and let $k \in \mathbb{N}$. Then there exists a Σ_1-$\boldsymbol{FO(TC[\Delta_0])}$-formula $\mathbf{con}(x_1, \ldots, x_k)$ such that*

$$\mathcal{G}^\epsilon(\partial_k(\mathcal{A})) \models \phi(u_1, \ldots, u_k)$$

just in case $u_1, \ldots, u_k$ are the leaves of some k-derivative tree of $\mathcal{A}$.

Proof. When considering a tree with a particular diagram Dia we can employ the formula from Lemma 5.37:

$$\exists \vec{y}.\phi_{Dia}(x_1, \ldots, x_k, \vec{y})$$

where *w.l.o.g.* we assume that the first k-parameters of ϕ_{Dia} correspond to the k leaves of the diagram. Lemma 5.35 tells us that there are only a finite number of k-diagrams and so we can take $\mathbf{con}(x_1, \ldots, x_k)$ to be the disjunction of all such formulae (for each diagram Dia). □

We have given the name **con** to the formula in Lemma 5.38 in order to be suggestive of its use in the light of Lemma 5.31.

5.5 Some Σ_1 decidability results

The work in the previous section can be used to reduce decidability of Σ_1 sentences on the ϵ-closure of an $n_{(n-1)..2}$-CPDA graph to the decidability of a Σ_1-$\boldsymbol{FO(TC[\Delta_0])}$ on its derivative, which is an $(n-1)$-CPDA graph. Since we have a way of asserting the existence of consistent and pairwise distinct k-tuples of configurations the idea is essentially the same as with Lemma 5.19.

Theorem 5.39. *Let $\mathcal{A}$ be an n_S-CPDA with $S \subseteq [2..(n-1)]$ and let ϕ be a Σ_1-sentence of $\boldsymbol{FO}$. Then we can compute an $(n-1)_S$-CPDA and a Σ_1-sentence ϕ' of $\boldsymbol{FO(TC[\Delta_0])}$ such that:*

$$\mathcal{G}^\epsilon(\mathcal{A}) \models \phi \text{ iff } \mathcal{G}^\epsilon(\partial_k(\mathcal{A})) \models \phi'$$

Proof. Without loss of generality we may assume that ϕ is in prenex normal form: $\exists x_1 x_2 \cdots x_k.\psi(x_1, x_2, \ldots, x_k)$ where ψ is quantifier free and is a *conjunction*. (This does not lose generality since existential quantification distributes over disjunctions). We further assume without loss of generality that ψ is of the form: $\bigwedge_{i \neq j} \neg x_i = x_j \wedge \chi(x_1, x_2, \ldots, x_k)$ where χ does not contain any occurrence of the equality relation. This does not lose generality because we may first change ψ to a disjunction (which can later be eliminated as above) over all possible exhaustive assertions of pairwise equality/inequality between the x_i. Since ψ is assumed to be a conjunction (after eliminating any introduced disjunction as previously) if $x_i = x_j$ occurs as a subformula of ψ we may remove this occurrence in exchange for replacing all instances of x_j with x_i provided that $\neg x_i = x_j$ does not also occur (in which case the sentence must be false as it would be inconsistent). All assertions of inequality may also be removed as these are already asserted.

By Lemma 5.31 and Lemma 5.38 we have $\mathcal{G}^\epsilon(\partial_k(\mathcal{A})) \models \mathsf{con}(u_1, \ldots, u_k)$ just in case there exist *pairwise distinct* and consistent $\mathcal{A}^+$-configurations $v_1, \ldots, v_k$ such that $u_i = \pi_{\Gamma^+}(top_n(v_i))$ for each $1 \leq i \leq k$ where Γ^+ is the stack-alphabet of $\mathcal{A}^+$. Crucially all unary predicates are fully determined by the top_n stack and control-state which are still present in the u_i. In particular the unary predicates of the form $(\mathsf{Met}(q,m)(x))^\downarrow$ inherited from $\mathcal{A}_k^{+\uparrow\downarrow}$ are determined completely by the top_n-stack. Due to consistency we may thus follow Lemma 5.19 in translating an atomic sub-formula of ψ of the form $x_i \; \boldsymbol{a} \; x$ to:

$$\hat{\boldsymbol{a}}(x, x_i) := \bigvee_{q \in Q} \bigwedge_{m \in M^i_{q,a}} (\mathsf{Met}(q,m)(x))^\downarrow$$

where Q is the set of control-states of $\mathcal{A}$ and $M^i_{q,a}$ is the set of meta-annotations $((Q^a_1)_{a\in\Sigma}, \ldots, (Q^a_k)_{a\in\Sigma})$ such that $q \in Q^a_i$. This translation is sound for the

same reasons as in Lemma 5.19. The translation of a unary predicate can be just the unary predicate itself due to fact that these are completely determined by the top stack symbol.

We can then take $\hat{\psi}(x_1, \ldots, x_k)$ to be the conjunction of all such translations corresponding to the conjunction that is $\psi(x_1, \ldots, x_k)$. Finally $\phi'(x_1, \ldots, x_k) := \exists x_1 \cdots x_k.(\mathsf{con}(x_1, \ldots, x_k) \ \wedge \ \psi(x_1, \ldots, x_k))$. □

One consequence of the next chapter is that the $\boldsymbol{FO(TC[\Delta_0])}$ theories of 2-CPDA are decidable and so a Corollary of Theorem 5.39 is:

Corollary 5.40. *The Σ_1 first-order theory of the ϵ-closure of any 3_2-CPDA graph is decidable.*

This result is of interest since the Π_2 theory of 3_2-graphs is only decidable when one avoids full ϵ-closure. It would be nice to extend this to all 3-CPDA but unfortuantely we are unable to see how to do so. It is interesting to note that the stumbling block is not to do with the *operational* treatment of 3-links as in fact these would never need to be 'simulated' in the derivative as they could be treated in the same way as ϵ-closure—namely via the annotations derived from $\mathcal{A}_k^{+\uparrow\downarrow}$ and appeal to consistency. Rather the problem is distinguishing between configurations whose only difference lies in their 3-links. It is unclear how to extend Lemma 5.14 to handle internal links, as an internal collapse would skip over information needed to keep track of the colour in the stack. We do, however, conjecture that at least in the case of order-3 this might be a surmountable problem.

Of course this raises the spectre that if no decidability result exists for general 3-CPDA, decidability might fail even if one includes 3-links but never performs a *collapse* operation.

It is also worth remarking that avoiding ϵ-closure does not appear to help this technique bear any further fruit. It can be shown that if the automaton has no ϵ transitions, then at most three elements in a log will have non-empty Q_i^a components in their meta-annotations. However, we also require multiple reachability tests in parallel to maintain the P components of meta-configurations, which *prime facie* is sufficient to require some kind of transitive closure logic.

It is also worth pointing out that we could use Theorem 5.39 to provide an alternative proof of decidability for n_n-CPDA since the annotated derivative would then be an $(n-1)$-PDA yielding graphs with decidable MSO and hence decidable $\boldsymbol{FO(TC[\Delta_0])}$ theories.

The author is aware that the reader may have felt afflicted by a firey assault of technicalities in this chapter, although we hope that (s)he may still have been able to gleen the intuitions and ideas that lie beneath them! We hope that the slight change of scenery in the next part of the dissertation will provide some

compensation as we move on to consider a new notion of automaticity that will eventually be used to characterise 2-CPDA graphs and provide an ***FO*** decidability result for 3_2-CPDA without ϵ-closure.

Chapter 6

Isophilic Structures and Rewrite Systems

Automatic structures are represented by finite automata recognising its domain and relations. Good closure properties then lead to the decidability of the first-order theory. Whilst traditionally automaticity concerns words and trees, we introduce a new notion of automaticity based on Alur *et al.*'s nested-words and nested-trees and the related visibly pushdown languages [6, 7, 4]. Originally introduced to represent the structure induced by runs of programs with calls and returns, nested-words can be viewed as conventional words together with 'back edges' or 'pointers' that respect a well-nested structure. Note that whilst also based on nested-words, our notion of automaticity is different from that of Arenas *et al.* [8] as it is intentionally more restrictive in order to precisely capture 2-CPDA graphs, which is one of our goals.

6.1 Nested-Words, Nested-Trees and Automata

Nested-Words

Recall that a Σ-labelled *word* is a map $w : \mathbf{dom}(w) \longrightarrow \Sigma$ where $\mathbf{dom}(w)$ is a downward closed subset of $\mathbb{N}$. A *(semi-)nested-word* is a word endowed with 'back pointers' that are arranged in a well-nested manner.

Definition 6.1. A *semi-nested-word* over Σ is a pair $w^{\curvearrowleft E}$ where w is a Σ-labelled word and $E : \mathbf{dom}(w) \rightharpoonup \mathbf{dom}(w)$ is a partial map such that for all $x \in \mathbf{dom}(E)$, $E(x) < x$ and $E(x) \leq E(y)$ for every $y \in \mathbf{dom}(E)$ such that $E(x) < y < x$. We deem it to be a *nested-word* if the last requirement is strengthened to $E(x) < y < x$ implying $E(x) \lneq E(y)$.

Graphically we represent E using pointers as in Figure 6.1. This set of nested-words is denoted $\mathbf{NWord}(\Sigma)$.

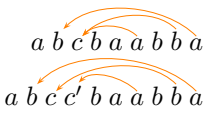


Figure 6.1: Semi-nested and nested-word

This definition is very similar to that given by Alur *et al.* [7] with the main difference that we disallow 'unmatched calls'—we cannot have a position in the word being the target of a pointer without the pointer having a corresponding source. This is important when considering a *prefix* of a nested-word $w^{\curvearrowright E}$ which we define to be a nested-word $w\restriction_S^{\curvearrowright E\restriction_S}$ for some downward closed subset $S \subseteq \mathbf{dom}(w)$. (We often abbreviate this to $u^{\curvearrowright E\restriction_u}$ where u is a prefix of w.) The fact that a prefix of a nested-word is itself a nested-word destroys information about pointers sourced in the suffix and targeted at the prefix. It is useful to have a notion of prefix for which such information is retained. For this reason we make a distinction with *visibly pushdown words* [6] which coincide with Alur *et al.*'s notion of nested-words but differ from our own.

Definition 6.2. Given an alphabet Σ the associated *visibly pushdown alphabet* $\mathfrak{V}(\Sigma)$ is given by $\mathfrak{V}(\Sigma) := \Sigma \cup \mathring{\Sigma} \cup \dot{\Sigma}$ where for every $a \in \Sigma$ the set $\mathring{\Sigma}$ consists precisely of elements of the form $\mathring{a}$ and $\dot{\Sigma}$ of elements of the form $\dot{a}$. Given $w^{\curvearrowright E} \in \mathbf{NWord}(\Sigma)$ we define the *visibly pushdown word* $\mathfrak{V}(w^{\curvearrowright E})$ to be the $\mathfrak{V}(\Sigma)$-word w' such that $\mathbf{dom}(w') = \mathbf{dom}(w)$ and for each $x \in \mathbf{dom}(w')$:

$$w'(x) := \begin{cases} \dot{a} & \text{if } x \in \mathbf{img}(E) \\ a & \text{if } x \notin \mathbf{img}(E) \cup \mathbf{dom}(E) \\ \mathring{a} & \text{if } x \in \mathbf{dom}(E) \end{cases}$$

Note that $\mathfrak{V}$ does not make sense for semi-nested words as the pointer structure could not be uniquely recovered from the image of the map. Figure 6.2 illustrates a nested-word and its associated visibly pushdown word together with the difference it makes to prefixes.

Another concept that we borrow from Alur *et al.* is the idea of the *summary* $\ulcorner w^{\curvearrowright E} \urcorner$ of a nested-word $w^{\curvearrowright E}$. This is the string that is obtained by reading the nested-word from left to right 'skipping over' the back edges as they are found. This concept applies equally well to semi-nested words although in practise we will not use it in this context.

Definition 6.3. The *summary* $\ulcorner w \urcorner$ of a (semi-)nested-word w is defined recursively as follows:

$$\ulcorner w\ a \urcorner := \ulcorner w \urcorner a \text{ if } a \text{ sources no pointer}$$

$$\ulcorner w_0\ a_0\ w_1\ a_1 \urcorner := \ulcorner w_0 \urcorner\ a_0\ a_1$$

This is illustrated in Figure 6.3. It is also possible to represent a nested-word as a tree whose paths are the summaries of its initial segments [7]. This

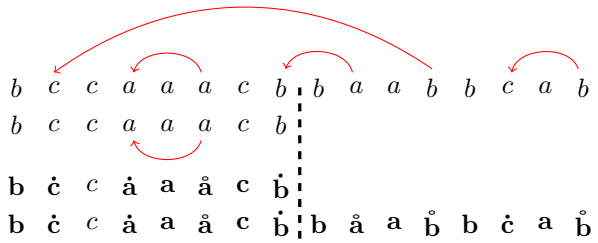


Figure 6.2: Prefix of a nested-word *vs.* visibly pushdown word.

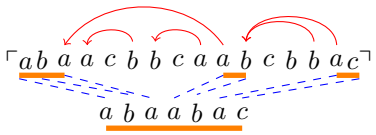


Figure 6.3: The summary of a semi-nested-word

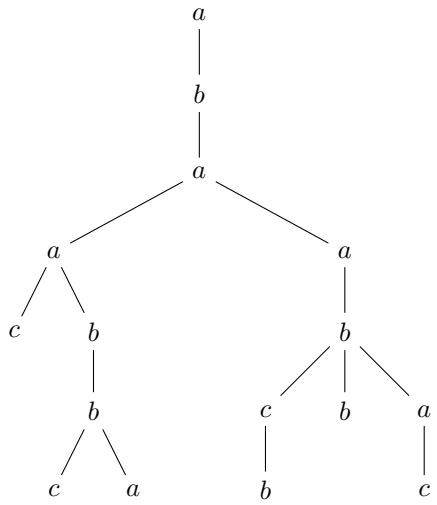


Figure 6.4: The tree presentation of the semi-nested-word in Figure 6.3

is illustrated in Figure 6.4. We will not make formal use of this notion, but it would be worth the reader keeping it at the back of his mind to understand the intuitions behind the progression from words, to nested-words, to nested-trees discussed in this chapter. Roughly speaking the targets of pointers in the nested-words correspond to branching points in the tree, with one branch ignoring the pointer and the other following it. Positions in the word immediately before the source of a pointer correspond to the tips of branches.

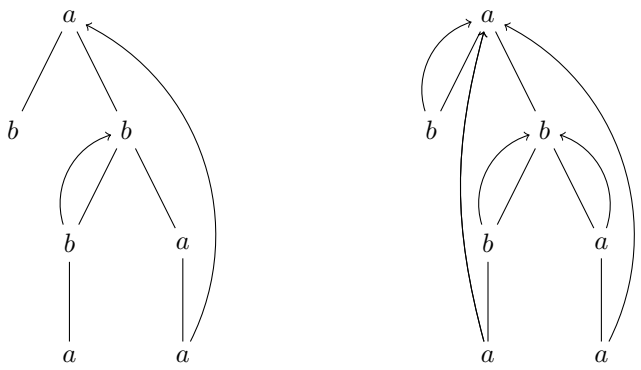


Figure 6.5: Path-wise nested-tree and nested-tree

Finally we will use two notions of projection on nested-words. Firstly for $w^{\frown_E} \in \mathbf{NWord}(\Sigma)$ the operation $\pi_{\Sigma'}(w^{\frown_E})$ removes positions that are neither $\mathbf{dom}(E)$ nor $\mathbf{img}(E)$ and are *not* labelled in $\Sigma' \subseteq \Sigma$.

Secondly if $f : \Sigma \longrightarrow \Sigma'$, then $f(w^{\frown_E})$ is the Σ'-labelled nested-word formed from $w^{\frown_E}$ by replacing every label $a \in \Sigma$ with $f(a) \in \Sigma'$. In particular we will sometimes take f to be π_i which is the ith projection of a Cartesian product.

Nested-Trees

Recall that a Σ-*labelled tree* with *degree bounded by* d (for $d \in \mathbb{N}$) is a map $T : \mathbf{dom}(T) \longrightarrow \Sigma$ where $\mathbf{dom}(T) \subseteq [1..d]^*$ is prefix-closed. A maximal element of $\mathbf{dom}(T)$ is known as a *leaf*, ϵ as *the root* and a *branch* is a path from the root to a leaf.

Definition 6.4. A *path-wise nested-tree* $T^{\frown_E}$ consists of a tree T together with a partial map $E : \mathbf{dom}(T) \rightharpoonup \mathbf{dom}(T)$ such that every branch is a nested-word. A *nested-tree* is a path-wise nested-tree $T^{\frown_E}$ such that for each leaf v and each node $v' \in \mathbf{img}(E)$ such that $v' \sqsubset v$ there exists a $u \in \mathbf{dom}(E)$ with $v' \sqsubset u \sqsubseteq v$ such that $E(u) = v'$.

The additional constraint for nested-trees ensures that if a node of the tree is the target of a pointer along some path, then it is the target of a pointer along every path passing through it. This definition is very similar to that of Alur *et al.* [4] with the difference that our trees are finite and as with nested-words, there is no notion of a node being the target of a pointer with no source. We illustrate the idea in Figure 6.5.

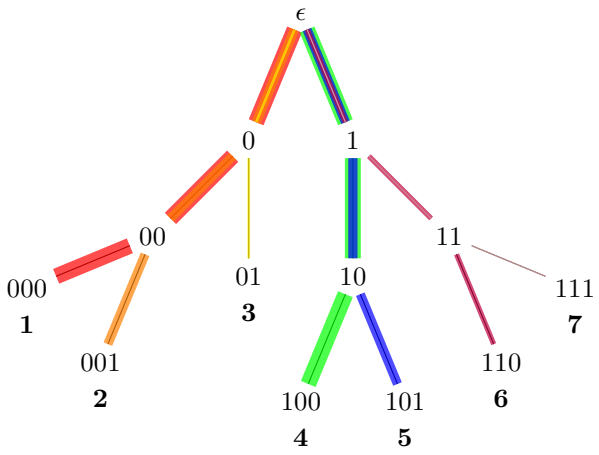


Figure 6.6: Branch ordering

Definition 6.5. Let **NTree**(Σ) be the set of Σ-labelled nested-trees (with any degree-bound). We write **NTree**$_k(\Sigma)$ to denote the set of nested-trees that have at most k leaves (branches).

Branch and Summary Ordering

We place a linear ordering on the branches of a tree following the manner in which a left-to-right depth first search would occur.

Definition 6.6. Let $l := a_1 \cdots a_r$ and $l' := a'_1 \cdots a'_{r'}$ be *leaves* of a (nested-)tree $T^{\frown E}$ and let b be the branch tipped by l and b' by l'. We say $b \prec b'$ iff $a_i < a'_i$ where i is the least j such that $a_j \neq a'_j$.

Figure 6.6 exhibits this ordering, where the red, orange, yellow, green, blue, indigo and violet branches are in $\prec$-ascending order.

We treat branches of a nested-tree as nested-words. Note though that distinct branches might be associated with the 'same' nested-word (same pointer structure and node labels). For a tree $T^{\frown E}$ we write $T^{\frown E}_{\prec_i}$ to denote the ith nested-word in its branch ordering.

Similarly we place a linear ordering on the summaries of certain prefixes of *semi*-nested-words. This amounts to the branch ordering on the standard tree representation thereof.

Definition 6.7. Let $w^{\frown E}$ be a semi-nested-word and let $succ(u_1) < succ(u_2) < \cdots < succ(u_r)$ be an exhaustive ordered list of the nodes in **dom**(E) (since the first node cannot be in **dom**(E) all must be of the form $succ(u)$). Writing ${v_i}^{\frown E_i}$ for the prefix of $w^{\frown E}$ ending with node u_i we define the summary ordering ◀

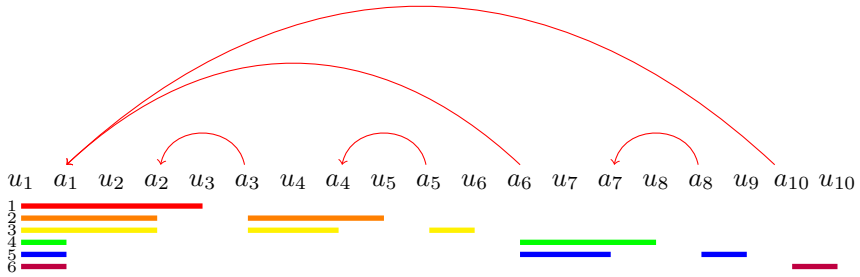


Figure 6.7: Summary ordering of a semi-nested word

of $w^{\curvearrowleft E}$:

$$\ulcorner v_1{}^{\curvearrowleft E_1}\urcorner \blacktriangleleft \ulcorner v_2{}^{\curvearrowleft E_2}\urcorner \blacktriangleleft \cdots \ulcorner v_r{}^{\curvearrowleft E_r}\urcorner \blacktriangleleft \ulcorner w^{\curvearrowleft E}\urcorner$$

.

We treat summaries as *non-nested* words and write $w^{\curvearrowleft E}_{\blacktriangleleft i}$ to denote the ith word in the summary ordering of $w^{\curvearrowleft E}$.

Traditional Nested-Tree Automata

We begin with Alur *et al.*'s nested-tree automata [4]. These read a nested-tree from top to bottom and have access to the state at the target of a pointer when reading its source. Formally a nested tree automaton (NTA) is a tuple:

$$\mathcal{A} \;=\; \langle \Sigma, Q, I, \delta_{\oplus}, \delta, \delta_{\ominus}, F \rangle$$

such that Σ is a finite alphabet, Q is a finite set of control-states, $I \subseteq Q$ is a set of initial states, $F \subseteq Q$ is a set of accepting states and $\delta_{\oplus}, \delta, \delta_{\ominus}$ are transition functions with the following types for some $k \in \mathbb{N}$:

$$\begin{aligned} \delta_{\ominus} \;:\; \Sigma \times Q &\longrightarrow 2^{\bigcup_{i=1}^{k} Q^i} \\ \delta \;:\; \Sigma \times Q &\longrightarrow 2^{\bigcup_{i=1}^{k} Q^i} \\ \delta_{\oplus} \;:\; \Sigma \times Q \times Q &\longrightarrow 2^{\bigcup_{i=1}^{k} Q^i} \end{aligned}$$

A *run-tree* of $\mathcal{A}$ on $T^{\curvearrowleft E} \in \mathbf{NTree}(\Sigma)$ is a Q-labelled tree $\mathcal{R}$ such that $\mathbf{dom}(\mathcal{R}) = \mathbf{dom}(T) \cup \{\, u0 \;:\; u \in \mathbf{dom}(T) \text{ is maximal} \,\}$ and:

- $\mathcal{R}(\epsilon) \in I$
- If $u \in \mathbf{img}(E)$ and u has i children in $\mathcal{R}$ (for some $1 \leq i \leq k$), then $(\mathcal{R}(u_1), \ldots, \mathcal{R}(u_i)) \in \delta_{\ominus}(T(u), \mathcal{R}(u))$.
- If $u \in \mathbf{dom}(T) - (\mathbf{dom}(E) \cup \mathbf{img}(E))$ and u has i children ($1 \leq i \leq k$), $(\mathcal{R}(u_1), \ldots, \mathcal{R}(u_i)) \in \delta(T(u), \mathcal{R}(u))$.

- If $u \in \mathbf{dom}(T)$ and u has i children (for some $1 \leq i \leq k$), then $(\mathcal{R}(u_1), \ldots, \mathcal{R}(u_i)) \in \delta_{\oplus}(T(u), \mathcal{R}(E(u)), \mathcal{R}(u))$.

Note that this is well-defined as by construction the nodes in $\mathcal{R}$ with children are precisely the nodes in T. A run-tree is deemed *accepting* just in case all of its leaves are labelled with a state in F. The language defined by $\mathcal{A}$ is:

$$\mathcal{L}(\mathcal{A}) \ := \ \{T^{\frown E} \in \mathbf{NTree}(\Sigma) \ : \ \mathcal{A} \ \textit{accepts} \ T^{\frown E}\}$$

A *nested-word automaton* is a nested-tree automaton that acts on nested-words. We call such a device *deterministic* if I is a singleton set and the image of each of the transition functions consists of singleton sets (so can in fact be viewed as being a subset of Q). It is known that all nested-word automata can be determinised (and hence complemented) [6, 7].

A *semi-nested-word automaton* is defined in exactly the same way as a nested-word automaton, except we allow semi-nested words into its language.

Some Automaton Variants

The next two kinds of device may be viewed as special-cases of nested-tree automata, and could be considered the *cascade product* of two automata. They take the form $\mathcal{B}^{\mathcal{C}}$ where $\mathcal{B}$ is a nested-tree automaton and $\mathcal{C}$ is a conventional finite tree automaton that has access to the state of $\mathcal{B}$. Formally, if $\mathcal{B} = \langle \Sigma, Q_{\mathcal{B}}, I_{\mathcal{B}}, \delta_{\oplus\mathcal{B}}, \delta_{\mathcal{B}}, \delta_{\ominus\mathcal{B}} \rangle$ we require $\mathcal{C}$ to have the form $\mathcal{C} = \langle \Sigma, Q_{\mathcal{B}}, Q_{\mathcal{C}}, I_{\mathcal{C}}, \delta_{\mathcal{C}}, F_{\mathcal{C}} \rangle$ where $I \subseteq Q_{\mathcal{C}}$ is a set of initial states, $F \subseteq Q$ is a set of final states and:

$$\delta_{\mathcal{C}} \ : \ \Sigma \times Q_{\mathcal{B}} \times Q_{\mathcal{C}} \longrightarrow 2^{\bigcup_{i=1}^{k} Q_{\mathcal{C}}^{i}}$$

. We do not require $\mathcal{B}$ to have any final states as they are irrelevant to the definition of accepting run.

A *run-tree* $\mathcal{R}$ for $\mathcal{B}^{\mathcal{C}}$ on $T^{\frown E} \in \mathbf{NTree}(\Sigma)$ is a $(Q_{\mathcal{B}} \times Q_{\mathcal{C}})$-labelled tree such that:

- $\pi_1(\mathcal{R})$ is a run-tree for $\mathcal{B}$.
- $\pi_2(\mathcal{R}(\epsilon)) = q$ for some $q \in I_{\mathcal{C}}$.
- If u has i children and $\mathcal{R}(u) = (p, q)$, then for each $1 \leq j \leq i$ we must have $\pi_2(\mathcal{R}(uj)) = q_j$ where $(q_1, \ldots, q_i) \in \delta_{\mathcal{C}}(T(u), p, q)$.

A run-tree is deemed *accepting* just in case all of its leaves have a label of the form $(\ \_\ , q)$ for some $q \in F_{\mathcal{C}}$. The language recognised by $\mathcal{B}^{\mathcal{C}}$ consists of precisely the elements of $\mathbf{NTree}(\Sigma)$ for which $\mathcal{B}^{\mathcal{C}}$ has an accepting run-tree.

In the special case when $\mathcal{B}$ is a *deterministic* nested-*word* automaton acting on each branch of the tree individually we call $\mathcal{B}^{\mathcal{C}}$ a *path-nested automaton.*

The determinism and branching-agnosticism of $\mathcal{B}$ ensures that the flow of information from $\mathcal{B}$ to $\mathcal{C}$ can only be one way—we can describe a path-nested automaton as having access to both the nesting structure and branching structure of the tree but 'not in combination'.

Definition 6.8. A path-nested automaton is an automaton $\mathcal{B}^{\mathcal{C}}$ such that $\mathcal{B}$ is deterministic and for any $(q_1, q_2, \ldots, q_k) \in \mathbf{img}(\delta_{\oplus\mathcal{B}}) \cup \mathbf{img}(\delta_{\mathcal{B}}) \cup \mathbf{img}(\delta_{\ominus\mathcal{B}})$ we have $q_1 = q_2 = \cdots = q_k$, where $\delta_{\oplus\mathcal{B}}, \delta_{\mathcal{B}}, \delta_{\ominus\mathcal{B}}$ are the transition functions of $\mathcal{B}$.

Despite the fact that $\mathcal{B}$ is technically a nested-tree automaton we may naturally view it as a nested-word automaton as it acts along every branch independently in a deterministic manner.

A different restriction of $\mathcal{B}$ gives a *spine nested automaton*, which behaves like a path-nested automaton along *'the spine'* of an input tree—the $\prec$-greatest path, which consists of right-most children. Elsewhere it can behave as an unrestricted nested-tree automaton.

Definition 6.9. We call $\mathcal{B}^{\mathcal{C}}$ a *spine nested automaton* if it has a subset of control-states $P \subseteq Q_{\mathcal{B}}$ such that:

- $I_{\mathcal{B}}$ is a singleton set (just one initial state) with element belonging to P.
- For every $p \in P$ there exists a unique $p' \in P$ such that $\delta(\_, p)$, $\delta_{\ominus}(\_, p)$ and $\delta_{\oplus}(\_, \_, p)$ map to sets S such that for every $\vec{v} \in S$ $\pi_{|\vec{v}|}(\vec{v}) = p'$.

So a spine nested automaton $\mathcal{B}^{\mathcal{C}}$ requires $\mathcal{B}$ to be 'deterministic along the spine of an input tree'.

One final variant we call a *trunk nested automaton* and this requires $\mathcal{B}$ to be deterministic along the *trunk* of an input tree—that is the path from the root prior to any branching.

Definition 6.10. We call $\mathcal{B}^{\mathcal{C}}$ a *trunk nested automaton* if it has a subset of control-states $P \subseteq Q_{\mathcal{B}}$ such that:

- $I_{\mathcal{B}}$ is a singleton set (just one initial state) with element belonging to P.
- For every $p \in P$ there exists a $p' \in P$ such that $\delta(\_, p)$, $\delta_{\ominus}(\_, p)$ and $\delta_{\oplus}(\_, \_, p)$ map to sets S such that for every $q \in S$ (where q is a one-element vector) $q = p'$.

These three variants, which all arise by restricting $\mathcal{B}$ in different ways, are illustrated in Figure 6.8.

Recognising Visibly Pushdown Trees

We have only explicitly defined $\mathfrak{V}$ to act on words as this will be an essential part of defining prefix rewriting. However, the definition can also be applied to

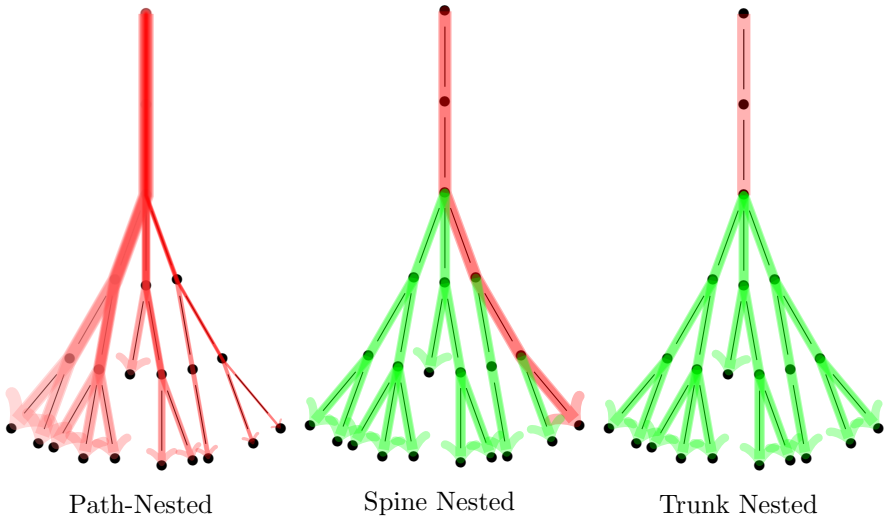


Figure 6.8: The manner in which path-nested, spine nested and trunk nested automata differ—the red lines indicate the branches along which $\mathcal{B}$ must be deterministic and ignore branching whilst the green lines indicate the branches along which it is allowed to behave as an unrestricted tree automaton.

trees and this is useful for some of the internals of proofs. A *frontier* delimits the analogue of a prefix for trees:

Definition 6.11. Let $T^{\curvearrowleft E} \in \mathbf{NTree}(\Sigma)$. A *frontier* $\mathcal{F}$ of $T^{\curvearrowleft E}$ is a set of nodes $\mathcal{F} \subseteq \mathbf{dom}(T)$ such that for all $x, y \in \mathcal{F}$ we have $x \not\sqsubseteq y$ and $y \not\sqsubseteq x$ but for every *leaf* $z \in \mathbf{dom}(T)$ there is a $w \in \mathcal{F}$ such that $w \sqsubseteq z$. We write $T_{\mathcal{F}}^{\curvearrowleft E}$ to denote the subtree of $T^{\curvearrowleft E}$ with domain restricted to the set $\{\, u \in \mathbf{dom}(T) \;:\; u \sqsubseteq z \text{ for some } z \in \mathcal{F} \,\}$. We say a frontier is *uniform* if $T_{\mathcal{F}}^{\curvearrowleft E}$ is a nested-tree (in particular has a source on *every* branch corresponding to any target).

The following automaton is designed to behave on a prefix of a word or tree restricted by a frontier in the same way regardless of whether that prefix or subtree occurs in its original context complete with its original pointers. The Boolean flag is used to keep track of whether a $\dot{a}$-labelled node could plausibly be the target of a pointer in some well-nested extension of the word or tree.

Definition 6.12. Let $\mathcal{A} = \langle \Sigma, Q, I, \delta_{\oplus}, \delta, \delta_{\ominus}, F \rangle$ be a top-down nested-tree automaton working over the alphabet Σ. The automaton $\mathfrak{V}(T^{\curvearrowleft E}) = \langle \mathfrak{V}(\Sigma), Q \times \mathbb{B}, I \times \{\mathbf{t}\}, \delta'_{\oplus}, \delta', \delta'_{\ominus}, F \times \mathbb{B} \rangle$ behaves in exactly the same way as $\mathcal{A}$ except that it operates over the alphabet $\mathfrak{V}(\Sigma)$ with the following adjustments:

- The automaton enforces that every node u labelled with $a \in \Sigma$ should satisfy $u \notin \mathbf{dom}(E) \cup \mathbf{img}(E)$. That is, $\delta'(a,(q,b)) := \delta(a,q) \times \{b\}$ and $\delta'_{\oplus}(a,(p,\_),(q,\_)) := \delta'_{\ominus}(a,(q,\_)) := \emptyset$ for all $p,q \in Q$ and all $a \in \Sigma$.

- The automaton enforces that every node u labelled with $\dot{a}$ but with $u \notin \mathbf{img}(E)$ should nevertheless be treated under the δ' function of $\mathfrak{V}(\mathcal{A})$ in the manner in which an a label would be treated under $\delta_{\ominus}$ of $\mathcal{A}$, *provided that there is no* $v \in \mathbf{img}(E)$ *such that* $u \sqsubseteq v$ *but* $E(v) \sqsubseteq u$. This is enforced by setting the flag to $\mathbf{f}$ which will prevent any such pointers from being introduced. So $\delta'(\dot{a},(q,\_)) := \delta_{\ominus}(a,q) \times \{\mathbf{f}\}$ for every $q \in Q$.

- The automaton enforces that every node $u \in \mathbf{img}(E)$ should be labelled with a label of the form $\dot{a}$ and should be treated in the same way as by $\mathcal{A}$. It sets the flag to $\mathbf{t}$ since any subsequent well-nested pointers will not target a node preceding the current node. That is $\delta'_{\ominus}(a,(q,\_)) := \delta'_{\ominus}(\mathring{a},(q,\_)) := \emptyset$ and $\delta'_{\ominus}(\dot{a},(q,\_)) := \delta_{\ominus}(a,q) \times \{\mathbf{t}\}$ for all $a \in \Sigma$ and all $q \in Q$.

- The automaton enforces that every node $u \in \mathbf{dom}(E)$ should have label of the form $\mathring{a}$ and should be treated in the same way as by $\mathcal{A}$. It may only proceed, however, if the flag is $\mathbf{t}$ so as to avoid a pointer passing over a $\dot{a}$-labelled pointerless element. The previous flag value can be recalled. That is, $\delta'_{\oplus}(\dot{a},(p,\_),(q,\_)) := \delta'_{\oplus}(a,(p,\_),(q,\_)) := \delta'_{\oplus}(\mathring{a},(p,\_),(q,\mathbf{f})) := \emptyset$ and $\delta'_{\oplus}(\mathring{a},(p,b),(q,\mathbf{t})) := \delta_{\oplus}(a,p,q) \times \{b\}$ for every $a \in \Sigma$ and $p,q \in Q$.

Let us say that a *pseudo run-tree* of an NTA on $T^{\frown E}$ is a tree $\mathcal{R}$ sharing exactly the same definition as a run-tree except that no extra leaves are added so that $\mathbf{dom}(R) = \mathbf{dom}(T)$.

Lemma 6.13. *A tree $\mathcal{R}$ is a run-tree of $\mathcal{A}$ on $T^{\frown E}$ just in case $\mathcal{R}$ is a run-tree of $\mathfrak{V}(\mathcal{A})$ on $\mathfrak{V}(T^{\frown E})$. Moreover if $\mathcal{F}$ is a frontier of $T^{\frown E}$ (and hence also of $\mathcal{R}$ and $\mathfrak{V}(T^{\frown E})$), then $\mathcal{R}_{\mathcal{F}}$ is a pseudo run-tree of $\mathfrak{V}(\mathcal{A})$ on $\mathfrak{V}(T^{\frown E})_{\mathcal{F}}$.*

Bottom-Up Automata

It will be useful to have a notion of bottom-up nested-tree automaton (BUNTA). Whilst the NTA introduced above begin at the root of a tree and move down towards its leaves, a bottom-up automaton begins at the leaves and moves towards the root. We are only interested in these devices when it is possible to determinise them. We thus restrict our attention to the case when the trees fed to the automaton have a bounded number of branches, or equivalently a bounded number of leaves. Formally a k-BUNTA $\mathcal{A}$ is an automaton acting on Σ-labelled nested-trees with at most k branches ($k \in \mathbb{N}$) where:

$$\mathcal{A} = \langle \Sigma, Q, I_1, I_2, \ldots, I_k, \delta_{\oplus}, \delta, \delta_{\ominus}, F \rangle$$

where Q is a finite set of control-states; $I_j \subseteq Q$ is a set of initial states for the jth leaf for each $1 \leq j \leq k$; $F \subseteq Q$ is a set of final states and there are transition functions:

$$\begin{aligned} \delta_{\oplus} &: \Sigma \times \bigcup_{1 \leq j \leq k} Q^j \longrightarrow 2^Q \\ \delta &: \Sigma \times \bigcup_{1 \leq j \leq k} Q^j \longrightarrow 2^Q \\ \delta_{\ominus} &: \Sigma \times \bigcup_{1 \leq j \leq k} Q^j \times \bigcup_{1 \leq j \leq k} Q^j \longrightarrow 2^Q \end{aligned}$$

We still intend $\delta_{\oplus}$ to act on sources of pointers and $\delta_{\ominus}$ on targets, but because we are reading the tree in a bottom-up manner, sources will be read prior to targets and so the types of $\delta_{\oplus}$ and $\delta_{\ominus}$ are 'the other way around' with respect to the original top-down version.

A *run-tree* of $\mathcal{A}$ on $T^{\frown E} \in \mathbf{NTree}_k(\Sigma)$ is a Q-labelled tree $\mathcal{R}$ such that $\mathbf{dom}(\mathcal{R}) = \mathbf{dom}(T) \cup \{ u0 : u \in \mathbf{dom}(T) \text{ is maximal } \}$ and where $b_1 \prec b_2 \prec \cdots \prec b_m$ for $m \leq k$ is the complete branch ordering of $\mathcal{R}$ and l_i is the leaf tipping b_i for each $1 \leq i \leq m$ we have:

- $\mathcal{R}(l_j) \in I_j$ for each $1 \leq j \leq m$
- If $u \in \mathbf{dom}(E)$ and u has i children $u_1, \ldots, u_i$, then $\mathcal{R}(u) \in \delta_{\oplus}(T(u), (\mathcal{R}(u_1), \ldots, \mathcal{R}(u_i)))$.
- If $u \notin \mathbf{dom}(E) \cup \mathbf{img}(E)$ and u has i children $u_1, \ldots, u_i$, then $\mathcal{R}(u) \in \delta(T(u), (\mathcal{R}(u_1), \ldots, \mathcal{R}(u_i)))$.
- If $u \in \mathbf{img}(E)$ and u has i children $u_1, \ldots, u_i$, then

  $$\mathcal{R}(u) \in \delta_{\ominus}(T(u), (\mathcal{R}(v_1), \ldots, \mathcal{R}(v_j)), (\mathcal{R}(u_1), \ldots, \mathcal{R}(u_i)))$$

  where $v_1 \prec v_2 \prec \cdots \prec v_j$ and $E^{-1}(u) = \{ v_1, \ldots, v_j \}$.

As before, note that the nodes with children in $\mathcal{R}$ are precisely the nodes in T and the leaves of $\mathcal{R}$ are precisely the nodes that $\mathcal{R}$ posesses but which T lacks.

We say that $\mathcal{R}$ is *accepting* just in case $\mathcal{R}(\epsilon) \in F$. The language defined by $\mathcal{A}$ is given by:

$$\mathcal{L}(\mathcal{A}) := \{T^{\frown E} \in \mathbf{NTree}_k(\Sigma) : \mathcal{A} \textit{ accepts } T^{\frown E}\}$$

The following is analogous to the top-down case, except that two Boolean flags are needed due to the fact that the $\delta_{\oplus}$ function deploys a control-state on the source of a pointer rather than reading the control-state on the source of a pointer.

Definition 6.14. Let $\mathcal{A} = \langle \Sigma, Q, I_1, I_2, \ldots, I_k, \delta_\oplus, \delta, \delta_\ominus, F\rangle$ be a k-BUNTA. Then the k-BUNTA $\mathfrak{V}(\mathcal{A}) := \big\langle \mathfrak{V}(\Sigma), Q \times \mathbb{B} \times \mathbb{B}^\perp, Q \times \{\mathbf{t}\}, Q \times \{\mathbf{t}\}, \ldots, Q \times \{\mathbf{t}\}, \delta'_\oplus, \delta', \delta'_\ominus, F\big\rangle$ behaves in a similar way to $\mathcal{A}$ except that it operates over the alphabet $\mathfrak{V}(\Sigma)$ with the following adjustments:

- The automaton enforces that every node u labelled with $a \in \Sigma$ should satisfy $u \notin \mathbf{dom}(E) \cup \mathbf{img}(E)$. That is, $\delta'(a, (q_1, b_1, \_), \ldots, (q_j, b_j, \_)) := \delta(a, q_1, \ldots, q_j) \times \{\bigwedge_{i=1}^{j} b_i\} \times \{\perp\}$ and $\delta'_\oplus(a, (p_1, \_, \_), \ldots, (p_i, \_, \_), (q_1, \_, \_), \ldots, (q_j, \_, \_)) := \delta'_\ominus(a, (q_1, \_, \_), \ldots, (q_j, \_, \_)) := \emptyset$ for all $1 \leq i, j \leq k$ and $p_1, \ldots p_i, q_1, \ldots, q_j \in Q$ and all $a \in \Sigma$. A conjunction of the Boolean values is appropriate as this will then be set to $\mathbf{f}$ if there is any $\mathring{a}$ (on any branch) requiring a subsequent $\dot{a}$ to be the target of a pointer.
- The automaton enforces that every node u labelled with $\dot{a}$ but with $u \notin \mathbf{img}(E)$ should nevertheless be treated under the δ' function of $\mathfrak{V}(\mathcal{A})$ in the manner in which an a label would be treated under $\delta_\ominus$ of $\mathcal{A}$, again ensuring an extended tree could render it the target of a pointer (by means of the Boolean flag). The lack of a corresponding source and associated state is dealt with via over-approximation. That is,

  $$\delta'(\dot{a}, (q_1, \mathbf{t}, \_), \ldots, (q_j, \mathbf{t}, \_)) := \bigcup_{\substack{p_1, \ldots, p_i \\ 1 \leq i \leq k}} \delta_\ominus(a, p_1, \ldots, p_i, q_1, \ldots, q_j) \times \{\mathbf{t}\} \times \{\perp\}$$

  for every $q_1, \ldots, q_j \in Q$. But $\delta'(\dot{a}, (q_1, \mathbf{f}, \_), \ldots, (q_j, \mathbf{f}, \_)) := \delta'(\mathring{a}, (q_1, \_), \ldots, (q_j, \_)) := \emptyset$.
- The automaton enforces that every node $u \in \mathbf{img}(E)$ should be labelled with a label of the form $\dot{a}$ and should be treated in the same way as by $\mathcal{A}$. We recall the Boolean values from the sources of the pointers. That is $\delta'_\ominus(a, (p_1, \_, \_), \ldots, (p_i, \_, \_), (q_1, \_, \_), \ldots (q_j, \_, \_)) := \delta'_\ominus(\mathring{a}, (p_1, \_, \_), \ldots, (p_i, \_, \_), (q_1, \_, \_), \ldots, (q_j, \_, \_)) := \emptyset$ and $\delta'_\ominus(\dot{a}, (p_1, \_, b_1), \ldots, (p_i, \_, b_i), (q_1, \_, \_), \ldots, (q_j, \_, \_)) := \delta_\ominus(a, p_1, \ldots, p_i, q_1, \ldots, q_j) \times \{\bigwedge_{r=1}^{i} b_r\} \times \{\perp\}$ for all $a \in \Sigma$ and all $p_1, \ldots, p_i, q_1, \ldots, q_j \in Q$.
- The automaton enforces that every node $u \in \mathbf{dom}(E)$ should have label of the form $\mathring{a}$ and should be treated in the same way as by $\mathcal{A}$. This means that the next $\dot{a}$ must be the target of a pointer, and so we set the Boolean flag to $\mathbf{f}$, whilst recording the current Boolean value (conjunction) in the third component to be recalled at the target of the pointer. That is, $\delta'_\oplus(\dot{a}, (q_1, \_, \_), \ldots, (q_j, \_, \_)) := \delta'_\oplus(a, (q_1, \_, \_), \ldots, (q_j, \_, \_)) := \emptyset$ and $\delta'_\oplus(\mathring{a}, (q_1, b_1, \_), \ldots, (q_j, b_i, \_)) := \delta_\oplus(a, q_1, \ldots, q_j) \times \{\mathbf{f}\} \times \{\bigwedge_{r=1}^{i} b_r\}$ for every $a \in \Sigma$, $1 \leq j \leq k$ and $q_1, \ldots, q_j \in Q$.

Again the following Lemma is a direct result from the construction above, except we need uniform frontiers for a bottom-up run-tree to make sense (so that behaviour is well-defined at the target of a pointer).

Lemma 6.15. *A tree $\mathcal{R}$ is a run-tree of a k-BUNTA $\mathcal{A}$ with leaf-states $I_1, \ldots I_k$ on a nested-tree $T^{\frown_E} \in \mathbf{NTree}_k(\Sigma)$ iff it is a run-tree of $\mathfrak{V}(\mathcal{A})$ on $\mathfrak{V}(T^{\frown_E})$ such that the jth leaf of $\mathcal{R}$ has label in I_j. Now let $\mathcal{F}$ be a* uniform *frontier of $T^{\frown_E}$. Then $\mathcal{R}_{\mathcal{F}}$ is a pseudo run-tree of $\mathfrak{V}(\mathcal{A})$ on $\mathfrak{V}(T^{\frown_E})_{\mathcal{F}}$.*

We now have the terminology needed to prove the following Lemma showing the top-down and bottom-up automata to be equi-expressive.

Lemma 6.16. *Let $\mathcal{A}$ be an NTA such that $\mathcal{L}(\mathcal{A}) \subseteq \mathbf{NTree}_k(\Sigma)$. It is then the case that there exists a k-BUNTA $\mathcal{A}'$ such that $\mathcal{L}(\mathcal{A}') = \mathcal{L}(\mathcal{A})$. Conversely let $\mathcal{A}'$ be a k-BUNTA. Then there exists an NTA $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$.*

Proof. Let $\mathcal{A} = \langle \Sigma, Q, I, \delta_\oplus, \delta, \delta_\ominus, F \rangle$ be an NTA such that $\mathcal{L} \subseteq \mathbf{NTree}_k(\Sigma)$. We define an equivalent k-BUNTA $\mathcal{A}' = \langle \Sigma, Q', I'_1, I'_2, \ldots, I'_k, \delta'_\oplus, \delta', \delta'_\ominus, F' \rangle$ as follows:

- Take $Q' := Q \cup Q \times Q$. For the purposes of this proof, let us us write $(\_, q)$ to mean 'either q or (p, q) for any p'.
- Take $F' := I$ and $I'_1 := I'_2 := \cdots := I'_k := F$
- Take $\delta'_\oplus$ to be the function such that $(p, q) \in \delta'_\oplus(a, (\_, q_1), (\_, q_2), \ldots, (\_, q_j))$ iff $(q_1, q_2, \ldots, q_j) \in \delta_\oplus(a, p, q)$ for every $q_1, q_2, \ldots, q_j, p, q \in Q$.
- Take δ' to be the function such that $q \in \delta'(a, (\_, q_1), (\_, q_2), \ldots, (\_, q_j))$ iff $(q_1, q_2, \ldots, q_j) \in \delta_\oplus(a, q)$ for every $q_1, q_2, \ldots, q_j, q \in Q$.
- Take $\delta'_\ominus$ to be the function such that $p \in \delta'_\ominus(a, (p, r_1), (p, r_2), \ldots, (p, r_l), (\_, q_1), (\_, q_2), \ldots, (\_, q_j))$ iff $(q_1, q_2, \ldots, q_j) \in \delta_\ominus(a, p)$. Any input to $\delta'_\ominus$ not matching this pattern is mapped to the empty set $\emptyset$.

Intuitively a run-tree of $\mathcal{A}'$ looks more or less the same as a run-tree of $\mathcal{A}$. The BUNTA $\mathcal{A}'$ is constrained when reading the source of pointers in the sense that it does not yet know what the control-state at the target of the pointer will be. Permissible run-trees of $\mathcal{A}$ depend on this information, which of course would be available to an automaton reading the tree from top to bottom. We circumvent this problem by allowing $\mathcal{A}'$ to *guess* the control-state at the target of a pointer when reading its source. This is indicated by $p \in Q$ in the definition above. When the target is eventually reached, $\mathcal{A}'$ constrains itself by refusing to perform any transition that is not consistent with this guess.

We want to show that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$. By Lemmas 6.13 and 6.15 it suffices to show that $\mathcal{L}(\mathfrak{V}'(\mathcal{A})) = \mathcal{L}(\mathfrak{V}(\mathcal{A}'))$, where $\mathfrak{V}'(\mathcal{A}')$ is the same as $\mathfrak{V}(\mathcal{A}')$ except that it shares the same leaf-state sets $I'_1, \ldots, I'_k$ as $\mathcal{A}'$.

Due to the choice of F' and $I'_1, \ldots, I'_k$ it is, in turn, sufficient to show that for every $T^{\frown_E} \in \mathbf{NTree}(\Sigma)$ there is a (not necessarily accepting) run-tree $\mathcal{R}$ of $\mathfrak{V}(\mathcal{A})$ on $T^{\frown_E}$ iff there is an *accepting* run-tree $\mathcal{R}'$ of $\mathfrak{V}(\mathcal{A}')$ on $T^{\frown_E}$

such that for each $u \in \mathbf{dom}(\mathcal{R}) = \mathbf{dom}(\mathcal{R}')$ we have either $\mathcal{R}(u) = \mathcal{R}'(u)$ or $\mathcal{R}(u) = \pi_2(\mathcal{R}'(u))$.

In turn this is implied by the claim that for for every $T^{\frown E} \in \mathbf{NTree}(\Sigma)$ and frontier $\mathcal{F}$ of $T^{\frown E}$, there is a pseudo run-tree $\mathcal{R}$ of $\mathfrak{V}(\mathcal{A})$ on $T^{\frown E}_{\mathcal{F}}$ iff there is an *accepting* pseudo run-tree $\mathcal{R}'$ of $\mathfrak{V}(\mathcal{A}')$ on $T^{\frown E}_{\mathcal{F}}$ such that for each $u \in \mathbf{dom}(\mathcal{R}) = \mathbf{dom}(\mathcal{R}')$ we have either $\mathcal{R}(u) = \mathcal{R}'(u)$ or $\mathcal{R}(u) = \pi_2(\mathcal{R}'(u))$.

We argue by induction on the maximum length of an element of $\mathcal{F}$. The base case when $\mathcal{F} = \{\epsilon\}$ is straightforward since the accepting states of $\mathfrak{V}(\mathcal{A}')$ are precisely the initial states of $\mathfrak{V}(\mathcal{A})$ (since this is the case for $\mathcal{A}'$ and $\mathcal{A}$). For the induction step consider extending the tree by adding children $u_1, \ldots, u_k$ to a particular leaf, which may each be given a label of one of the forms $a, \mathring{a}, \dot{a}$. For the cases when u_i is decorated by a or $\dot{a}$, the result follows immediately from the definition of the transition relations, recalling that for $\dot{a}$ in the absence of a corresponding $\mathring{a}$ the automaton $\mathfrak{V}(\mathcal{A}')$ will over approximate. A control-state of the form $(q, \_)$ from $\mathfrak{V}(\mathcal{A})$ is assigned to node u_i of its extended run-tree iff a control-state of the form $(q, \_, \_)$ is assigned from $\mathfrak{V}(\mathcal{A}')$, noting that the Boolean flags of each control-state are completely determined by the structure of the tree in each case.

Now suppose that one of the u_i is given a $\mathring{a}$ annotation and so must be the source of a pointer. The fact that the frontier is not uniform does not matter since in this particular case the validity of the run-tree can be preserved by choosing a state of each $\mathring{a}$ independently. We assign it state $(q, \_)$ in the run-tree of $\mathfrak{V}(\mathcal{A})$ iff we assign it a state $((p, q), \_, \_)$ in the run-tree of $\mathfrak{V}(\mathcal{A}')$ (again noting that the Boolean flags are uniquely determined by the pointer structure), where p is the $\mathcal{A}$ control-state associated with the corresponding $\dot{a}$ node v. This choice of p ensures that the run-tree of $\mathfrak{V}(\mathcal{A}')$ remains valid at v iff the extended run-tree of $\mathfrak{V}(\mathcal{A})$ is valid at u_i. An essentially similar argument will cover the second half of the claim.

Now let us consider the converse. Let $\mathcal{A}' = \langle \Sigma, Q', I'_1, I'_2, \ldots, I'_k, \delta'_\oplus, \delta', \delta'_\ominus, F' \rangle$ be a k-BUNTA. We construct an equivalent NTA $\mathcal{A} = \langle \Sigma, Q, I, \delta_\oplus, \delta, \delta_\ominus, F \rangle$ using a similar idea to the other direction of the lemma. This time it is the top-down automaton doing the guessing at the target of a pointer using its $\delta_\oplus$ function. The guess specifies several control-states that it postulates will occur at the corresponding sources of the pointers by means of a partial function from sets of branch indices. So if the guess suggests that there will be a corresponding pointer source with state q lying on a node belonging to branches $2, 3$ and 4, the partial function will map $\{2, 3, 4\}$ to q. The partial function is undefined on a set S when it is guessed that S will not reflect the branches on which any corresponding pointer source will lie—in this case the function would be undefined at $\{2, 3\}$ since a node lying only on branches 2 and 3 cannot source an appropriate pointer if a node lying on branches $\{2, 3, 4\}$ does. In order to

verify these guesses (using $\delta_\ominus$) it is necessary to keep track of which branch is which, which is done using a subset of $[1..k]$ contained in the state.

- Take $Q := \mathcal{P} \times Q' \times 2^{[1..k]}$ where $\mathcal{P}$ is the set of partial functions $f : 2^{[1..k]} \rightharpoonup Q$ where all $S \in \mathbf{dom}(f)$ are disjoint and such that if $i, j \in S$ and $i < r < j$ then $r \in S$. Note that these constraints induce a natural ordering $L < L'$ on the elements of $\mathbf{dom}(f)$ where $L < L'$ iff $l < l'$ for any representatives $l \in L$ and $l' \in L'$.

- Take $F := \{\ (\bot, q, \{j\})\ :\ q \in I'_j\ \}$ (where q is a single element vector and $\bot$ is the everywhere undefined partial function) and $I := \mathcal{P} \times F' \times \{S \subseteq [1..k] : S \text{ downward closed}\}$.

- Take $\delta_\ominus$ to be the function such that

$$((f_1, q_1, L_1), (f_2, q_2, L_2), \ldots, (f_j, q_j, L_j)) \in \delta_\ominus(a, (f, q, L))$$

  where $f_1, \ldots, f_j \in \mathcal{P}$, $q \in Q$, $L_1 \uplus \cdots \uplus L_j = L$ with $L_1 < L_2 < \cdots < L_j$, iff $q \in \delta'_\ominus(f(S_1), \ldots, f(S_i), q_1, \ldots, q_j)$ where $\mathbf{dom}(f) = \{S_1 < S_2 < \cdots < S_i\}$. The idea is that f records a guess as to which branches will bear the corresponding pointer sources.

- Take δ to be the function such that

$$((f_1, q_1, L_1), (f_2, q_2, L_2), \ldots, (f_j, q_j, L_j)) \in \delta(a, (q, L))$$

  where $q \in Q$, $L_1 \uplus \cdots \uplus L_j = L$ with $L_1 < L_2 < \cdots < L_j$ and $f_1, \ldots, f_j$ are any partial functions in $\mathcal{P}$, iff $q \in \delta'(a, q_1, \ldots, q_j)$.

- Take $\delta_\oplus$ to be the function such that

$$((f_1, q_1, L_1), (f_2, q_2, L_2), \ldots, (f_j, q_j, L_j)) \in \delta_\oplus(a, (f, \_, \_), (\bot, q, L))$$

  where $f_1, \ldots, f_j \in \mathcal{P}$, $q \in Q$, and $L_1 \uplus \cdots \uplus L_j = L$ with $L_1 < L_2 < \cdots < L_j$, iff $q \in \delta'_\oplus(q_1, \ldots, q_j)$ and $q = f(L)$.

First observe that for any accepting run-tree of $\mathcal{A}$ the L component of the state contains precisely the indices of the branches to which that node belongs (where branches are indexed in a left-right manner). This can be seen by induction on the tree being read using the hypothesis: 'If the initial state at the root of the tree $T^{\frown E}$ has L component $[1..i]$, then the L component of all nodes of the run tree correctly indicate branch indices for some other tree of which $T^{\frown E}$ is an extension'. This is verified by checking each rule in turn. Conversely the claim that for any input tree there is a run-tree of an automaton only taking account of the L components correctly labelling each node with the branches to which it belongs—again verifiable by induction, checking each rule on turn, taking

the initial state to use L component $[1..i]$ where i is the number of branches in the tree.

So let us assume that every run-tree of $\mathcal{A}$ has L component correctly labelling each node of the tree with the branches to which it belongs. In a similar manner to before, it is sufficient to see that $\mathfrak{V}(\mathcal{A})$ has a pseudo run-tree on $T^{\frown E}_{\mathcal{F}}$ just in case $\mathfrak{V}(\mathcal{A}')$ has a pseudo run-tree on $T^{\frown E}_{\mathcal{F}}$ where at each corresponding pair of nodes in the run-trees, the Q' component is the same in each, for every $T^{\frown E} \in \mathbf{NTree}_k(\Sigma)$ and *uniform* frontier $\mathcal{F}$. Uniformity is required since an appropriate choice of partial function at the target of a pointer depends on the location and state annotation of *all* of its sources. The induction can be performed in a similar manner to before but nodes added in the induction step must be done in a fair manner so that all source nodes sharing a single target are added simultaneously. When adding sources $\mathring{a}_1, \mathring{a}_2, \ldots, \mathring{a}_i$ to form a uniform frontier, the induction step is possible by replacing the partial function in the run tree of $\mathfrak{V}(\mathcal{A})$ at the corresponding target $\dot{a}$ with $f(L_j) := q_j$ where L_j is the L annotation of the node at $\mathring{a}_j$ and q_j is its control-state. This must preserve correctness of the run-tree iff correctness is preserved in the extension of $\mathfrak{V}(\mathcal{A}')$ by the definition of behaviour if $\delta_{\ominus}$.

□

Boolean Closure

In order to develop a notion of automaticity, we need to have some kind of Boolean closure for the languages recognised by these automata. We are only interested in top-down automata for this result, although bottom-up automata will come into the proof. It is easy to get closure under intersection since all of our automata can be run in parallel.

Lemma 6.17. *Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be languages of nested-trees recognised by nested-tree automata. Then we can construct a nested-tree automaton recognising $\mathcal{L} := \mathcal{L}_1 \cap \mathcal{L}_2$. Moreover if $\mathcal{L}_1$ and $\mathcal{L}_2$ are both recognised by respectively path-nested, spine nested or trunk nested automata, then $\mathcal{L}$ is also recognised by respectively a path-nested, spine nested or trunk nested automaton.*

Proof. A standard product construction suffices. □

It is known that nested-word automata can be determinised and hence complemented [7][6]. Unfortunately nested-tree automata can be neither determinised nor complemented in general [4]. We can, however, complement a path-nested automaton $\mathcal{B}^{\mathcal{C}}$ since $\mathcal{B}$ is, in particular, deterministic. It is thus just a matter of complementing $\mathcal{C}$ in the same manner as a conventional tree automaton.

Lemma 6.18. *Let $\mathcal{A}$ be a path-nested automaton operating over the alphabet Σ. There exists a path-nested automaton $\overline{\mathcal{A}}$ recognising $\overline{\mathcal{L}(\mathcal{A})} := \mathbf{NTree}(\Sigma) - \mathcal{L}(\mathcal{A})$.*

Proof. Let $\mathcal{B}^{\mathcal{C}}$ be a path-nested automaton. The automaton $\mathcal{B}$ is deterministic and so we only need to complement the finite tree automaton $\mathcal{C}$, which can be done using standard techniques. □

By contrast, complementing spine nested and trunk nested automata is impossible in general since being able to do so would imply that general nested-tree automata could be complemented. After all, both of these variants act as general nested-tree automata on certain components of the tree. Adapting Alur *et al.*'s proof for the determinisation of nested-words [7][6] we can, however, complement a set of nested-trees with a bounded number of leaves. We obtain this result by first showing that bottom-up automata can be determinised. This allows them to be complemented and so we can then just appeal to Lemma 6.16.

Lemma 6.19. *Let $\mathcal{A}$ be a k-BUNTA. Then there exists a* deterministic *k-BUNTA $\mathcal{A}'$—that is one such that the image of the transition functions consists of singleton sets and the leaf-state sets are also singleton—such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$.*

Proof. We follow the same proof idea as used by Alur *et al.* for nested-words [7][6]. Recall that bottom-up automata on standard trees can be determinised using a power set contruction. If the bottom up automaton simply skips over pointers, then such a power set construction would still work. This is in effect what happens, except that before skipping over a pointer, a new automaton is spawned that reads the sub-tree between the sources and targets. This spawned automaton is responsible for considering all possible combinations of control-states at the sources; in order to keep track of the required information we rely on the fact that the tree has at most k branches. The result of this automaton is additionally made available at the target of the pointer.

With this intuition in mind we define $\mathcal{A}' = \langle \Sigma, Q', I'_1, I'_2, \ldots, I'_k, \delta'_{\oplus}, \delta', \delta'_{\ominus}, F' \rangle$ from $\mathcal{A} = \langle \Sigma, Q, I_1, I_2, \ldots, I_k, \delta_{\oplus}, \delta, \delta_{\ominus}, F \rangle$ to be as follows:

- $Q' := 2^{Q \times \bigcup_{0 \leq i \leq k} Q^i} \cup 2^{Q \times \bigcup_{0 \leq i \leq k} Q^i} \times 2^{Q \times \bigcup_{0 \leq i \leq k} Q^i}$, writing $(\_, T)$ to denote either T or (S, T) for some S.
- $I'_j := \{I_j \times \{\epsilon\}\}$ for each $1 \leq i \leq k$.
- $\delta'_{\oplus}(a, (\_, T_1), \ldots, (\_, T_j)) := \{(\{\ (q', w_1 w_2 \cdots w_j)\ :\ q' \in \delta_{\oplus}(a, q_1, \ldots, q_j) \text{ and } (q_i, w_i) \in T_i \text{ for each } 1 \leq i \leq j\ \}, \{\ (q, q)\ :\ q \in Q\ \})\}$
- $\delta'(a, (\_, T_1), \ldots, (\_, T_j)) := \{\{\ (q', w_1 w_2 \cdots w_j)\ :\ q' \in \delta(a, q_1, \ldots, q_j) \text{ and } (q_i, w_i) \in T_i \text{ for each } 1 \leq i \leq j\ \}\}$

- $\delta'_{\ominus}(a, ((S'_1, T'_1), \ldots, (S'_l, T'_l)), ((\_, T_1), \ldots, (\_, T_j))) :=$
  $\{\{\ (q', w'_1 w'_2 \cdots w'_l)\ :\ q' \in \delta_{\ominus}(a, p_1, \ldots, p_l, q_1, \ldots, q_j)$ for some $(q_i, w_i) \in T_i$ for each $1 \leq i \leq j$ such that
  $w_1 w_2 \cdots w_j = p_1 p_2 \cdots p_l$, and $(p_i, w'_i) \in S'_i$ for each $1 \leq i \leq l\ \}\}$

- $F' := \{F \times \{\epsilon\}\}$

We claim that the unique run tree $\mathcal{R}$ of $\mathcal{A}'$ on a nested-tree $T^{\curvearrowleft E} \in \mathbf{NTree}_k(\Sigma)$ has the following properties:

- At a node $u \notin \mathbf{dom}(E)$ we have $\mathcal{R}(u) = T \in 2^{Q \times \bigcup_{0 \leq i \leq k} Q^i}$. At a node $u \in \mathbf{dom}(E)$ we have $\mathcal{R}(u) = (S, T) \in 2^{Q \times \bigcup_{0 \leq i \leq k} Q^i} \times 2^{Q \times \bigcup_{0 \leq i \leq k} Q^i}$. We keep this notation u, S and T in the items below (which hold irrespective of whether $u \in \mathbf{dom}(E)$).

- Let $N_{\leq u} := \{\ v \in \mathbf{dom}(E)\ :\ v \sqsubseteq u$ but $E(v) \sqsupset u$ and for all v' s.t. $v \sqsubset v' \sqsubset u, v' \in \mathbf{dom}(E)$ implies $E(v') \sqsubseteq u\ \}$. That is $N_{\leq}$ is the set of sources of pointers closest to u in the subtree rooted at u whose pointers have not yet been discharged by reaching their targets. Note that if $u \in \mathbf{dom}(E)$ then $N_{\leq u} = \{u\}$. We can order $N_{\leq u}$ (if it is non-empty, which is not necessarily the case) using the branch ordering: $v_1 \prec v_2 \prec \cdots \prec v_m$. Then the set T consists of precisely those elements $(q, p_1 p_2 \cdots p_m)$ such that $\mathcal{A}$ could have a run-tree starting at the subtree with leaves $v_1, v_2, \ldots, v_m$ in respective states $p_1, p_2, \ldots, p_m$ ending at the node u in control-state q. If $N_{\leq u}$ is empty, then T consists of precisely those elements (q, ϵ) such that $\mathcal{A}$ would have a run-tree starting at the leaves of $T^{\curvearrowleft E}$ in initial states and ending at u in state q.

- Now consider $u \in \mathbf{dom}(E)$ so that (S, T) is the associated state. Let $N_{<u} := \{\ v \in \mathbf{dom}(E)\ :\ v \sqsubset u$ but $E(v) \sqsupset u$ and for all v' s.t. $v \sqsubset v' \sqsubset u, v' \in \mathbf{dom}(E)$ implies $E(v') \sqsubseteq u\ \}$—that is the same definition as $N_{\leq u}$ except that we consider the nodes $v \in \mathbf{dom}(E)$ that are *strictly below* u. Again order the members of $N_{<u}$ with the branch ordering: $v_1 \prec v_2 \prec \cdots \prec v_m$. Then the set S consists of precisely those elements $(q, p_1 p_2 \cdots p_m)$ such that $\mathcal{A}$ could have a run-tree starting at the subtree with leaves $v_1, v_2, \ldots, v_m$ in respective states $p_1, p_2, \ldots, p_m$ ending at the node u in control-state q. If $N_{<u}$ is empty, then T consists of precisely those elements (q, ϵ) such that $\mathcal{A}$ would have a run-tree starting at the leaves of $T^{\curvearrowleft E}$ in initial states and ending at u in state q.

It is mechanical to verify that the transition rules maintain these invariants on the assumption that they have been maintained whilst reading the nodes strictly below u in the subtree rooted at u. The third item of the invariant is necessary to verify that $\delta_{\ominus}$ preserves the second item of the invariant.

The second item of the invariant implies (due to the choice of F') that the unique run-tree of $\mathcal{A}'$ is accepting iff $\mathcal{A}$ has a run-tree. □

We can now get closure under complementation for nested-tree automata acting on trees with a bounded number of branches.

Lemma 6.20. *Fix $k \in \mathbb{N}$. Let $\mathcal{A}$ be a nested-tree automaton operating over the alphabet Σ. There exists a nested-tree automaton $\overline{\mathcal{A}}$ recognising $\mathbf{NTree}_k(\Sigma) - \mathcal{L}(\mathcal{A})$.*

Proof. Apply Lemma 6.16 to get a k-BUNTA $\mathcal{A}'$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$. We may then determinise $\mathcal{A}'$ using Lemma 6.19. Since the automaton has a unique run-tree on any given $T^{\curvearrowleft E} \in \mathbf{NTree}_k(\Sigma)$ we may complement it by complementing its set of final states. The automaton $\overline{\mathcal{A}}$ may be obtained by applying Lemma 6.16 a final time to go from the complemented k-BUNTA to the top-down NTA. □

Remark 6.21. The reader may wonder why we do not encode elements of $\mathbf{NTree}_k(\Sigma)$ as nested-words and then appeal to the boolean closure of nested-word automata. Whilst such an encoding is possible it is perhaps not as neat as it would be for non-nested trees—encodings of branches would have to be embedded within one another rather than just be concatenated. This is due to the need to preserve well-nesting of pointers. It is felt that it is as easy to reformulate the proof in terms of nested-trees as it is to deal with encodings and decodings. The reader might also ask why we do not just forget about $\mathbf{NTree}_k(\Sigma)$ altogether and deal exclusively with nested-words. The reason for this lies in our application of them, which relies on having both the pointer *and genuine* branching structure.

And as a corollary to Lemma 6.17, Lemma 6.18 and Lemma 6.20 we have:

Lemma 6.22. *Over the universe $\mathbf{NTree}(\Sigma)$ path-nested automata are closed under Boolean operations. Over the universe $\mathbf{NTree}_k(\Sigma)$ for fixed $k \in \mathbb{N}$ nested-tree automata (which include spine nested automata) are closed under Boolean operations.*

Note that the lemma above guarantees only that the complement of a spine nested automaton is a nested-tree automaton; it does not say that its complement can be represented by another spine nested automaton. It will be important to proving the $\boldsymbol{FO(TC[\Delta_0])}$ component of Theorem 7.20 that a spine nested automaton can be complemented to another spine nested automaton when operating over trees with at most two branches.

Lemma 6.23. *Over the universe $\mathbf{NTree}_2(\Sigma)$ spine nested automata are closed under complement and since they are closed under intersection must thus be closed under Boolean operations.*

Proof. Let us call the branch that is not part of the spine of a tree in $\mathbf{NTree}_2(\Sigma)$ the *rib*. When acting over elements of $\mathbf{NTree}_2(\Sigma)$ the $\mathcal{B}$ component of a spine nested automaton $\mathcal{B}^{\mathcal{C}}$ can be decomposed into a deterministic nested-word automaton $\mathcal{B}_S$ reading the spine and then for every pair of control-states (q, p) of $Q_{\mathcal{B}} \times Q_{\mathcal{C}}$ a nested-word automaton $\mathcal{B}_{q,p}$ behaving as $\mathcal{B}^{\mathcal{C}}$ would along the *rib* starting in state (q, p) at the first node of the rib after branching from the spine. Noting that the target-states of pointers targeting the spine but sourced along the rib can be treated as part of the input alphabet for the purposes of determinisation, we can determinise each nested-word automaton $\mathcal{B}_{q,p}$. A nested-tree automaton $\mathcal{B}'$ can then be constructed that behaves as $\mathcal{B}_S$ along the spine and then at the branching point runs all of the deterministic $\mathcal{B}_{q,p}$ in parallel along the rib. The finite tree automaton $\mathcal{C}'$ behaves as $\mathcal{C}$ does along the spine and at the branching point picks an $\mathcal{B}_{q,p}$ such that a branch to state (q, p) would have been possible and accepts the rib just in case $\mathcal{B}_{p,q}$ is accepting.

The spine nested automaton ${\mathcal{B}'}^{\mathcal{C}'}$ thus simulates $\mathcal{B}^{\mathcal{C}}$ and recognises the same language. Moreover, the run-tree of $\mathcal{B}'$ is unique on every input tree and so it can be complemented by complementing the finite tree automaton $\mathcal{C}'$. □

Note that this cannot be generalised beyond $\mathbf{NTree}_2(\Sigma)$ since spine nested automata can act completely non-deterministically on the trunks of subtrees not part of the spine. The construction relies on there being just one rib.

Other Properties of Automata

We turn our attention to closure under a notion of 'projection of branches'. For this we use a *skeleton selector* which is a *deterministic* tree automaton $\mathcal{S}$ whose state space can be partitioned into two sets P and S with transition function having the property that every state in S yields at least one child with a state in S and no state in P can yield a child with a state in S.

Definition 6.24. A *skeleton selector* is a finite tree automaton $\mathcal{S} = \langle \Sigma, Q, q_0, \delta, F, P, S \rangle$ where Q is a finite set of control-states; $q_0 \in Q$ is the initial state; $\delta : \Sigma \times Q \longrightarrow 2^{\bigcup_{i=1}^{k} Q^i}$ is a transition function specifying allowable behaviours of the automaton at a node with i children for $1 \leq i \leq k$, F is a set of accepting states and $P, S \subseteq Q$ are such that $Q = P \cup S$ and $P \cap S = \emptyset$. We additionally require that if $q \in S$ and $\vec{r} \in \delta(a, q)$, then *at least one* element in $\vec{r}$ must be in S. Moreover if $q \in P$ and $\vec{r} \in \delta(a, q)$, then *every* element in $\vec{r}$ must be in P.

A *run-tree* of a skeleton selector $\mathcal{S} = \langle \Sigma, Q, q_0, \delta, F, P, S \rangle$ on a Σ-labelled tree T (which may or may not be nested) is a tree $\mathcal{R}$ such that $\mathbf{dom}(\mathcal{R}) = \mathbf{dom}(T)$ and such that:

- $\mathcal{R}(\epsilon) = q_0$

- Suppose that $u \in \mathbf{dom}(T)$ has i children $u_1, \ldots, u_i$. Then for $1 \leq j \leq i$ we have $(\mathcal{R}(u_1), \mathcal{R}(u_w), \ldots, \mathcal{R}(u_i)) \in \delta(a, \mathcal{R}(u))$.

We say that a run-tree is *accepting* if all of its leaves are decorated by elements of F. A $\mathcal{S}$-skeleton of $T^{\frown E} \in \mathbf{NTree}(\Sigma)$ is the unique subtree consisting of precisely the nodes assigned a state in S by an accepting run of $\mathcal{S}$. We denote the set of $\mathcal{S}$-skeletons by $\mathcal{S}(T^{\frown E})$. The *$\mathcal{S}$-skeleton language* and *$\mathcal{S}^\infty$-skeleton language* recognised by a nested-tree automaton $\mathcal{A}$ are respectively denoted and defined by:

$$\mathcal{L}^\pi_{\mathcal{S}}(\mathcal{A}) := \{\ S^{\frown D}\ :\ S^{\frown D} \in \mathcal{S}(T^{\frown E}) \text{ for some } T^{\frown E} \in \mathcal{L}(\mathcal{A})\ \}$$
$$\mathcal{L}^\infty_{\mathcal{S}}(\mathcal{A}) := \{\ S^{\frown D}\ :\ S^{\frown D} \in \mathcal{S}(T^{\frown E}) \text{ for infinitely many } T^{\frown E} \in \mathcal{L}(\mathcal{A})\ \}$$

So a skeleton belongs to $\mathcal{L}^\pi_{\mathcal{S}}(\mathcal{A})$ if there is some tree recognised by $\mathcal{A}$ with that skeleton. A skeleton belongs to $\mathcal{L}^\infty_{\mathcal{S}}(\mathcal{A})$ if there are *infinitely many* trees recognised by $\mathcal{A}$ that all have that same skeleton.

Very often we will be interested in skeleton selectors $\mathcal{S}$ that choose precisely one subtree—that is such that $\mathcal{S}(T^{\frown E})$ is a singleton for every $T^{\frown E}$. An important example is the *exoskeleton* selector that chooses precisely the $\prec$-greatest and $\prec$-least branches of a tree. For trees with branching degree bounded by d this can be defined by a deterministic tree automaton with $S = \{q_0, l, r\}$, $P = \{p\}$ where q_0 is the initial state and the transition function is given by:

$$\begin{aligned}
\delta(_, q_0) &:= \{q_0, (l,r), (l,p,r), (l,p,p,r), \ldots, (l, \underbrace{p,p,\ldots,p}_{d-2 \text{ times}}, r)\} \\
\delta(_, l) &:= \{l, (l,p), (l,p,p), \ldots, (l, \underbrace{p,p,\ldots,p}_{d-1 \text{ times}})\} \\
\delta(_, r) &:= \{r, (p,r), (p,p,r), \ldots, (\underbrace{p,p,\ldots,p}_{d-1 \text{ times}}, r)\} \\
\delta(_, p) &:= \bigcup_{i=1}^{d} \{p^i\}
\end{aligned}$$

where all states are accepting states.

These skeleton languages provide a form of projection under which (almost all) our automata are closed.

Lemma 6.25. *Let $\mathcal{S}$ be a skeleton-selector and let $\mathcal{A}$ be a nested-tree (*resp. *path-nested) automaton. There exists nested-tree (*resp. *path-nested) automata $\mathcal{S}(\mathcal{A})$ and $\mathcal{L}^\infty_{\mathcal{S}}(\mathcal{A})$ such that:*

$$\mathcal{L}(\mathcal{S}(\mathcal{A})) = \mathcal{L}^\pi_{\mathcal{S}}(\mathcal{A}) \quad \textit{and} \quad \mathcal{L}(\mathcal{S}^\infty(\mathcal{A})) = \mathcal{L}^\infty_{\mathcal{S}}(\mathcal{A})$$

If $\mathcal{S}$ selects the spine of every tree (amongst possibly other branches) then the above also holds for spine nested automata. If $\mathcal{S}$ selects both the $\prec$-greatest and $\prec$-least branches (amongst possibly other branches), then it also holds for trunk nested automata.

Proof. We first exhibit $\mathcal{S}(\mathcal{A})$ recognising the required language. Let us assume that $\mathcal{A}$ is of the form $\mathcal{B}^{\mathcal{C}^-}$ (if $\mathcal{A}$ is a generic nested-tree automaton then it can be represented in this form by ignoring the $\mathcal{C}$ component). We begin by replacing $\mathcal{C}^-$ with the product of $\mathcal{S}$ and $\mathcal{C}^-$, which we call $\mathcal{C}$.

For $(p, q) \in Q_{\mathcal{B}} \times Q_{\mathcal{C}}$ we define the set:

$$\begin{aligned} \mathit{Reach}(p,q) \;:=\; \{\, S \subseteq Q_{\mathcal{B}} \times Q_{\mathcal{C}} \;:\; \exists T^{\curvearrowleft E}.\ &\text{there is a run-tree} \\ &\text{of } \mathcal{B}^{\mathcal{C}} \text{ on } T^{\curvearrowleft E} \text{ with root labelled } (p,q) \text{ and leaves} \\ &\text{labelled by pairs of states in } S\} \end{aligned}$$

This set is computable. Consider $(p, q) \in Q_{\mathcal{B}} \times Q_{\mathcal{C}}$ and $S \subseteq Q_{\mathcal{B}} \times Q_{\mathcal{C}}$. We can decide whether $S \in \mathit{Reach}(p, q)$ by simply checking for emptiness the nested-tree automaton formed from $\mathcal{B}^{\mathcal{C}}$ by changing the initial state to (p, q) and setting the final states to S.

We also define a set of control-states $Q^+_{\mathcal{B}} := Q_{\mathcal{B}} \times 2^{Q_{\mathcal{B}} \times Q_{\mathcal{C}}}$. The automaton $\mathcal{B}^+$ endowed with this state set behaves in the same manner as $\mathcal{B}$ keeping track of additional information in the second component of its state. If it is in state (q, S) at a node u of the input tree $T^{\curvearrowleft E}$ this means that it is possible to replace the subtree of $T^{\curvearrowleft E}$ rooted at u with another subtree (whose root is glued to u) such that $\mathcal{B}^{\mathcal{C}}$ would be able to complete the run-tree from u to become an accepting run-tree starting in a state $(q', p) \in S$.

$\mathcal{B}^+$ is able to compute this set S on the fly in a manner that also respects any deterministic behaviour of the original $\mathcal{B}$. In its initial state $(q_{0\mathcal{B}}, S_0)$ we can just take $S_0 := \{(p, q) : \exists F \in \mathit{Reach}(p, q)$ containing only accepting states$\}$. So long as neither the target nor source of a link is traversed the set S remains the same and $\mathcal{B}^+$ need not update it. This is because the set of ancestors of the current node that are targets with unmatched sources would not have changed. When transitioning from a node that is a source of a link u to its children, the set of ancestor nodes that are targets with unmatched sources returns to exactly what it was at the target $E(u)$ of u. Thus the transition from the source of a link to its children can simply restore S to what it was at $E(u)$.

So we only need to consider the case when u is the target of a pointer—*i.e.* $u \in \mathbf{img}(E)$. In this case the children of u will have an additional unmatched target amongst its ancestors in comparison to u. We define the following operator on S assuming the simulated $\mathcal{B}$ is in control-state p at u:

$$\begin{aligned} \mathit{AddTarget}(p,S) :=\{\, (p',q) \in Q_{\mathcal{B}} \times Q_{\mathcal{C}} \;:\; &\exists F \in \mathit{Reach}(p',q) \text{ s.t. } \forall (r,s) \in F. \\ &\exists a \in \Sigma.\ \exists k \in \mathbb{N}.\ \exists((r_1,s_1),\ldots,(r_k,s_k)) \in \delta_{\oplus \mathcal{B}^{\mathcal{C}}}(a,p,(r,s)) \\ &\text{s.t. } (r_i,s_i) \in S \text{ for each } 1 \le i \le k \,\} \end{aligned}$$

So if the simulated control-state of $\mathcal{B}$ is p, the S sent to the children of u is $\mathit{AddTarget}(p, S)$. In summary the transition functions of $\mathcal{B}^+$ can be described

thus:

$$\begin{aligned}
\delta_{\ominus\mathcal{B}^+}(a,(q,S)) &:= \{\ ((q_1,\mathit{AddTarget}(q,S)),(q_2,\mathit{AddTarget}(q,S)),\\
&\quad \ldots,(q_k,\mathit{AddTarget}(q,S)))\ :\ (q_1,q_2,\ldots,q_k)\in\delta_{\ominus\mathcal{B}}(a,q)\ \}\\
\delta_{\mathcal{B}^+}(a,(q,S)) &:= \{\ ((q_1,S),(q_2,S),\ldots,(q_k,S))\ :\ (q_1,q_2,\ldots,q_k)\in\delta_{\mathcal{B}}(a,q)\ \}\\
\delta_{\oplus\mathcal{B}^+}(a,(p,R),(q,S)) &:= \{\ ((q_1,R),(q_2,R),\ldots,(q_k,R))\ :\ (q_1,q_2,\ldots,q_k)\in\delta_{\oplus\mathcal{B}}(a,p,q)\ \}
\end{aligned}$$

and $\mathcal{B}^+$ will maintain the invariant that when in control-state (q,S) at node u there exists a subtree that could replace the subtree rooted at u such that $\mathcal{B}^{\mathcal{C}}$ could continue with a run from u in this replacement tree from a control-state in S and finish at every leaf of the replacement subtree in accepting states.

We can now modify the automaton $\mathcal{B}^+$ to form $\mathcal{B}^{++}$ that actually implements deletion. In order to preserve path nestedness/spine nestedness/trunk nestedness we do not give $\mathcal{B}^+$ the power to directly consider the skeleton selector as this may introduce undesirable non-determinism (despite the skeleton selector being non-deterministic, missing out branches of the tree might introduce non-determinsim).

The automaton $\mathcal{B}^{++}$ makes a transition to k children $u_1,u_2,\ldots,u_k$ in control states $(q_1,q_2,\ldots,q_k)$ where $\mathcal{B}^+$ could make a transition to $k'\geq k$ children arranged as $\vec{v_1}u_1\vec{v_2}u_2\ldots\vec{v_k}u_k\vec{u_k}$ in control states $\vec{p_1},q_1,\vec{p_2},q_2,\ldots,\vec{p_k}q_k\vec{p_{k+1}}$. No constraints are placed on the choice of $u_1,\ldots,u_k$ for generic nested-tree automata or for path-nested automata. Observe that if $\mathcal{B}$ is path-nested, $\mathcal{B}^+$ will also be path-nested and because path-nestedness is immune to influence from behaviour at siblings. If $\mathcal{B}$ is spine-nested or trunk-nested, however, then we impose some additional constraints:

- If $\mathcal{B}$ is spine nested and $\mathcal{S}$ selects the spine of *every* tree, then we insist that $|\vec{p_{k+1}}|=0$—*i.e.* $\mathcal{B}^{++}$ always assumes that the spine of the tree being read is also the spine of the tree on which a run of $\mathcal{B}^+$ is being simulated. Note that this means that $\mathcal{B}^{++}$ will also be spine nested. It also will never violate the constraints imposed by the skeleton selector.

- If $\mathcal{B}$ is trunk nested and $\mathcal{S}$ selects both the leftmost and rightmost branches of *every* tree, then we insist that $|\vec{p_1}|=|\vec{p_{k+1}}|=0$. The assumption about $\mathcal{S}$ means that this will never be too strong a constraint to prohibit selecting the branches that $\mathcal{S}$ would select. Moreover, this constraint also means that the trunk of both the original tree and the skeleton will be shared and so $\mathcal{B}^{++}$ will also be a trunk nested automaton.

For technical reasons we additionally extend the state space of $\mathcal{B}^{++}$ to include a record of the $\mathcal{B}$ control-state at $E(u)$ on the children of a node $u\in\mathbf{dom}(E)$. (We make the record on the children of u rather than u itself so as to avoid introducing any non-determinism as a result of this record).

We now define a finite tree automaton $\mathcal{C}^+$ that has access to the control-states of $\mathcal{B}^{++}$. $\mathcal{C}^+$ bases its behaviour on a simulation of $\mathcal{C}$ (recall that $\mathcal{C}$ in turn includes a simulation of $\mathcal{S}$). When it makes a transition to k children $u_1, u_2, \ldots, u_k$, it guesses pairs of control states in $Q_\mathcal{B} \times Q_\mathcal{C}$ as a vector $\vec{r_1}(p_1, q_1)\vec{r_2}(p_2, q_2)\vec{r_3} \cdots \vec{r_k}(p_k, q_k)\vec{r_{k+1}}$ such that $\mathcal{B}^\mathcal{C}$ could make a transition to children $\vec{v_1}u_1\vec{v_2}u_2 \ldots \vec{v_k}u_k\vec{u_k}$ in these control-states. It verifies the legitimacy of this guess when reading the state of $\mathcal{B}^{++}$ at the children, which contains a record of the control-state at the target of a pointer should the ability of $\mathcal{B}^\mathcal{C}$ to perform the guessed transition depend on this. It also checks (via its simulation of $\mathcal{S}$) that precisely the nodes $u_1, u_2, \ldots, u_k$ would be selected out of the postulated vector $\vec{v_1}u_1\vec{v_2}u_2 \ldots \vec{v_k}u_k\vec{u_k}$. A final consistency check ensures that the guess has the property that the control-state of $\mathcal{B}$ simulated by $\mathcal{B}^{++}$ at the child u_i is p_i.

$\mathcal{C}^+$ should also ensure that the pairs in the $\vec{r_i}$ can all yield accepting run trees. This occurs just in case any one of the children u_j could be replaced with a subtree from which $\mathcal{B}^\mathcal{C}$ would have an accepting run tree were it to start at the root of the new subtree in a state specified by such a pair. This is precisely the information contained in the set S from $\mathcal{B}^+$ in the control state of $\mathcal{B}^{++}$ at children—$\mathcal{C}^+$ just needs to check that the pairs in $\vec{r_i}$ all belong to the set S associated with the children (which will be the same for each child).

Note that we do not need to adjust the accepting states of $\mathcal{C}^+$—we can take them to be those that simulate accepting states of $\mathcal{C}$. The transition function of $\mathcal{C}^+$, as described above, will ensure that only $\mathcal{S}$ states that select a branch are propagated down the actual tree asserted to be a skeleton.

We can thus take $\mathcal{S}(\mathcal{A}) := \mathcal{B}^{++^{\mathcal{C}^+}}$.

Now let us turn our attention to $\mathcal{S}^\infty(\mathcal{A})$. In addition to the set $\mathit{Reach}(p, q)$ we need a set representing the fact that an infinite number of trees can be found giving a run with leaves ending in the given set of states:

$$\begin{aligned}\mathit{Inf}(p, q) := \{\ S \subseteq Q_\mathcal{B} \times Q_\mathcal{C}\ :\ &\exists^\infty T^{\frown E}.\ \text{there is a run-tree}\\ &\text{of } \mathcal{B}^\mathcal{C} \text{ on } T^{\frown E} \text{ with root labelled } (p, q) \text{ and leaves}\\ &\text{labelled by pairs of states in } S\}\end{aligned}$$

Again this set is computable. Consider $(p, q) \in Q_\mathcal{B} \times Q_\mathcal{C}$ and $S \subseteq Q_\mathcal{B} \times Q_\mathcal{C}$. Again we form an automaton from $\mathcal{B}^\mathcal{C}$ by changing the initial state to (p, q) and the final states to S. We have $S \in \mathit{Inf}(p, q)$ just in case the language accepted by this modified automaton is infinite. If it is infinite it must contain a large member on which the Pumping Lemma may be applied. Conversely if it contains a large member we may apply the Pumping Lemma to produce an infinite number of members. Thus we just need to test whether the language contains a large member—we can intersect the automaton with an automaton recognising large trees and test this for emptiness.

To form $\mathcal{S}^{\infty}(\mathcal{A})$ we adapt $\mathcal{B}^{+}$ to form $\mathcal{B}^{\infty}$. This automaton has state space $Q_{\mathcal{B}^{\infty}} := Q_{\mathcal{B}} \times 2^{Q_{\mathcal{B}} \times Q_{\mathcal{C}}} \times 2^{Q_{\mathcal{B}} \times Q_{\mathcal{C}}}$. A state (q, S, T) consists of q and S, which have exactly the same meaning as with $\mathcal{B}^{+}$ and are updated in exactly the same manner, together with T which is a set of $\mathcal{B}^{\mathcal{C}}$ from which the current pointer context would allow an accepting run on infinitely many extensions of the current tree.

The initial T_0 is given by $T_0 := \{(p, q) : \exists F \in \mathit{Inf}(p, q)$ containing only accepting states$\}$. At the target of a pointer T is updated to T' in a manner similar to the move from S to S' except that in this case we are also interested in the possibility of there being infinitely many possible extensions prior to the pop occurring. The analogue of the operator $\mathit{AddTarget}(p, S)$ for T is $\mathit{AddTarget}^{\infty}(p, S, T)$ defined as follows:

$$
\begin{aligned}
\mathit{AddTarget}^{\infty}(p, S, T) := \{\ & (p', q) \in Q_{\mathcal{B}} \times Q_{\mathcal{C}}\ :\ \exists F \in \mathit{Inf}(p', q) \text{ s.t. } \forall (r, s) \in F. \\
& \exists a \in \Sigma.\ \exists k \in \mathbb{N}.\ \exists ((r_1, s_1), \ldots, (r_k, s_k)) \in \delta_{\oplus \mathcal{B}^{\mathcal{C}}}(a, p, (r, s)) \\
& \text{s.t. } (r_i, s_i) \in S \text{ for each } 1 \leq i \leq k\ \} \\
& \cup \\
& \{\ (p', q) \in Q_{\mathcal{B}} \times Q_{\mathcal{C}}\ :\ \exists F \in \mathit{Reach}(p', q) \text{ s.t. } \forall (r, s) \in F. \\
& \exists a \in \Sigma.\ \exists k \in \mathbb{N}.\ \exists ((r_1, s_1), \ldots, (r_k, s_k)) \in \delta_{\oplus \mathcal{B}^{\mathcal{C}}}(a, p, (r, s)) \\
& \text{s.t. } (r_i, s_i) \in S \text{ for each } 1 \leq i \leq k \\
& \text{and } (r_i, s_i) \in T \text{ for some } 1 \leq i \leq k\ \}
\end{aligned}
$$

The first component of the union captures the cases when there are an infinite number of subtrees preceding the corresponding source of the target being added and the second component of the union captures the cases when there are an infinite number of extensions of some subtree beyond the corresponding source. If there are an infinite number of extensions, then the extension must fall into one of these two categories. Note that we depend on S as well as T since not all branches need to admit an infinite number of possibilities for there to be an infinite number of possible trees—only one such branch need admit this.

Thus we implement $\mathcal{B}^{\infty}$ to update T in the same manner as S at source nodes and nodes that are neither a source nor a target and at a target node u we update T in a state (p, S, T) to $\mathit{AddTarget}^{\infty}(p, S, T)$ at the children of u. The automaton $\mathcal{B}^{\infty +}$ is then created from $\mathcal{B}^{\infty}$ in exactly the same way as $\mathcal{B}^{++}$ is created from $\mathcal{B}^{+}$. The finite tree automaton $\mathcal{C}^{\infty}$ is then defined in exactly the same manner as $\mathcal{C}^{+}$ (but with respect to $\mathcal{B}^{\infty +}$ instead of $\mathcal{B}^{++}$) except that $\mathcal{C}^{\infty}$ *additionally* ensures that T can be used in place of S in *at least one* instance of deletion.

We then take $\mathcal{S}^{\infty}(\mathcal{B}^{\mathcal{C}}) := \mathcal{B}^{\infty +^{\mathcal{C}^{\infty}}}$. This recognises precisely the trees that

belong to the set of skeletons for an infinite number of trees. First suppose that $T^{\frown E} \in \mathcal{L}(\mathcal{S}^{\infty}(\mathcal{B}^{\mathcal{C}}))$. Then it must be possible to decorate $T^{\frown E}$ with states of $\mathcal{B}^{\mathcal{C}}$ such that this constitutes a subtree of some accepting run tree $\mathcal{R}$ of $\mathcal{B}^{\mathcal{C}}$ such $T^{\frown E}$ is precisely a subtree selected by $\mathcal{S}$. Additionally there must exist at least one node of $\mathcal{R}$ shared with $T^{\frown E}$ with a child in $\mathcal{R}$ not shared by $T^{\frown E}$ that can be replaced by infinitely many possible alternative subtrees ending in accepting states. This ensures that $T^{\frown E}$ does indeed belong to the set of skeletons of an infinite number of trees.

Conversely assume that $T^{\frown E}$ belongs to the set of skeletons of an infinite number of trees. Since $T^{\frown E}$ has finite size, this must mean that there exists at least one assignment σ of states of $\mathcal{B}^{\mathcal{C}}$ to the nodes of $T^{\frown E}$ that forms a skeleton-selected subtree of infinitely many run-trees of $\mathcal{B}^{\mathcal{C}}$. Moreover there must exist at least one node u in $T^{\frown E}$ from which there are infinitely many extensions to accepting run-trees consisting of nodes not belonging to $T^{\frown E}$. Observe that the existence of an extension of a node u in $T^{\frown E}$ with a run-tree depends exclusively on the assignment σ (to ancestors of u) and the states spawned down to the children of u—both the children in $T^{\frown E}$ and those in the extended run tree. However, since all trees accepted by $\mathcal{B}^{\mathcal{C}}$ have nodes with bounded degree and since there are only finitely many control-states of $\mathcal{B}^{\mathcal{C}}$ it follows that there are only finitely many combinations of possible states on the children of u. Thus there must be at least one child of u that can be added as part of an extension of $T^{\frown E}$ decorated with σ to a run tree of $\mathcal{B}^{\mathcal{C}}$ that can be assigned a control-state from which there are infinitely many extensions to accepting run trees. It follows that $\mathcal{S}^{\infty}(\mathcal{B}^{\mathcal{C}})$ will accept $T^{\frown E}$, as required.

□

6.2 Introducing Automaticity for Nested Trees

Traditional Tree Automaticity

The idea of a 'tree automaticity' is to present a relational structure as having a set of finite trees as a domain and recognising n-ary relations using finite tree automata that act on n trees in a synchronous manner. The latter is best viewed in terms of *convolutions* of n trees that can be defined as follows. Given Σ-labelled trees $T_1, T_2, \ldots, T_n$ the *standard convolution* $\bigotimes_! \langle\, T_1, T_2, \ldots, T_n \,\rangle$ of $T_1, T_2, \ldots, T_n$ is defined to be the Σ^n-labelled tree T such that:

- $\mathbf{dom}(T) := \bigcup_{i=1}^{n} \mathbf{dom}(T_i)$
- For each $u \in \mathbf{dom}(T)$ and each $1 \leq i \leq n$ we have:

$$\pi_i(T(u)) = \begin{cases} T_i(u) & \text{if } u \in \mathbf{dom}(T_i) \\ \boxtimes & \text{if } u \notin \mathbf{dom}(T_i) \end{cases}$$

We use the term *standard* and write the ! annotation in $\otimes_!$ in order to distinguish it from the revised notion of convolution to be introduced a bit later, which is what will be used for the bulk of this chapter. A standard tree automatic structure can then be defined to be a relational structure whose domain is a regular tree language and whose n-ary relations are regular languages of n-ary convolutions of the n-tuples belonging to the relation. We write ***TAut*** to denote the class of standard tree automatic structures and we write ***WAut*** to denote the class of standard word automatic structures (defined in the same way but with domains restricted to words—*i.e.* unary trees).

First-order logic is decidable on tree automatic structures as the good closure properties of tree automata allow one to recognise any set of convolutions that is first-order definable. However we cannot directly extend it to nested trees since standard convolutions would present a problem when the nested-trees being conjoined have differing structure with respect to pointers. If corresponding nodes in two trees are respectively the source and target of the pointer, it is not possible to naïvely add a pointer to the convolution that covers both cases. One should also note that tree automatic (and indeed just word automatic) structures can capture the configuration graphs of Turing machines. This precludes any nice treatment of non-locality. For example, it would be good to be able to define a 'reachability' predicate in *all* automatic structures; but so long as we wish to return first order definability this could not be done by extending tree automaticity directly.

We will thus use an alternative definition of convolution for the purposes of this chapter, which we now introduce.

Automaticity for Nested-Trees and Words

The *pointer pattern* $pat_{T^{\frown E}}(u)$ of a node $u \in \mathbf{dom}(T)$ in a nested-tree $T^{\frown E}$ is a string over the alphabet $\{t, n, s\}$ (target, neutral, source) such that if $u_1, \ldots, u_m$ is the path from the root to u, then $pat_{T^{\frown E}}(u) = p_1 \cdots p_m$ where:

$$p_i = \begin{cases} t & \text{if } u_i \in \mathbf{img}(E) \\ n & \text{if } u_i \notin \mathbf{img}(E) \cup \mathbf{dom}(E) \\ s & \text{if } u_i \in \mathbf{dom}(E) \end{cases}$$

Also consider the *trace* $trace_{T^{\frown E}}(u)$ of a node $u \in \mathbf{dom}(T)$ where T is Σ-labelled, which is an element of Σ^* indicating the sequence of Σ-labels of the nodes in the path from the root to u inclusively. Let us also abuse notation so that the domain of a tree may be a subset of $\{ (u, p, v) \in S_1^* \times S_2^* \times S_3^* : |u| = |p| = |v| \}$ for finite sets S_1, S_2 and S_3. Due to the length constraint we may also view an element (u, p, v) as being in $(S_1 \times S_2 \times S_3)^*$ and any 'prefix closed' set of such elements can naturally be viewed as the domain of a tree.

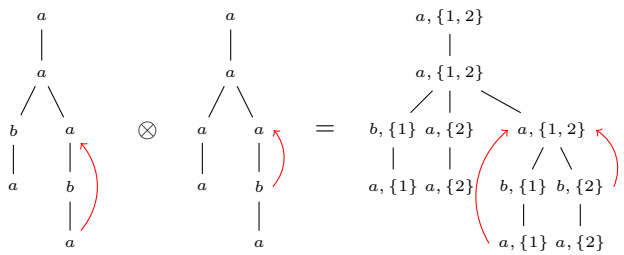


Figure 6.9: Convolution of Two Trees

Assuming an ordering on S_1, S_2 and S_3 we may also view such a tree as being ordered by taking the lexicographic ordering on $S_1 \times S_2 \times S_3$.

Let $T_1{}^{\curvearrowleft E_1}, T_2{}^{\curvearrowleft E_2}, \ldots, T_n{}^{\curvearrowleft E_n} \in \mathbf{NTree}(\Sigma)$. We define their *convolution* $\bigotimes \langle T_1{}^{\curvearrowleft E_1}, T_2{}^{\curvearrowleft E_2}, \ldots, T_n{}^{\curvearrowleft E_n} \rangle$ to be the $\Sigma \times 2^{[1..n]}$-labelled nested-tree $T^{\curvearrowleft E}$ where:

- $\mathbf{dom}(T) = \{(u, p, v) \in ([1..k]^* \times \{t, n, s\}^* \times \Sigma^*) : \exists i.1 \leq i \leq n.u \in \mathbf{dom}(T_i)$ and $pat_{T^{\curvearrowleft E}}(u) = p$ and $trace_{T^{\curvearrowleft E}}(u) = v\}$
- $\mathbf{dom}(E)$ consists of nodes in $\mathbf{dom}(T)$ of the form $(u, p.s, v)$ and we set $E(u, p.s, v)$ to be the node $(u', p'.t, v')$ bearing the matching t to the s at u.
- We set $T(u, p, v'.a) = (a, S)$ where:

$$i \in S \text{ iff } u \in \mathbf{dom}(T_i), pat_{T_i{}^{\curvearrowleft E_i}}(u) = p \text{ and } trace_{T_i{}^{\curvearrowleft E_i}}(u) = v'.a$$

The ordering on $\{t, n, s\}$ and Σ is immaterial at the level of abstraction at which we will work, although the reader should bear in mind the standard ordering on $[1..k]$ which is the dominant component of the lexicographic ordering.

We define the set $Convo^n_T(\Sigma)$ to be the set of n-ary convolutions over Σ (for trees) and $Convo^n_W(\Sigma)$ to be the set of n-ary convolutions over *nested-words* (rather than trees). $Convo^n_W(\Sigma)$ thus contains nested-trees with at most n leaves.

Let us fix a signature $\sigma = (R_1^{m_1}, \ldots, R_k^{m_k})$ (each $m_i \in \mathbb{N}$) for relational structures. We describe six different ways of presenting σ-structures $\mathfrak{A} = \langle \boldsymbol{A}, \boldsymbol{R_1}, \ldots, \boldsymbol{R_k} \rangle$ using nested-tree automata. In each case $\boldsymbol{A} \subseteq \mathbf{NTree}(\Sigma)$ (sometimes $\mathbf{NWord}(\Sigma)$) for some finite alphabet Σ and $\boldsymbol{R_i}$ is encoded as a subset of $Convo^n_T(\Sigma)$ (sometimes $Convo^n_W(\Sigma)$) such that $\bigotimes \langle\, T_1{}^{\curvearrowleft E_1}, \ldots, T_k{}^{\curvearrowleft E_k} \,\rangle$ belongs to the encoding iff $(\, T_1{}^{\curvearrowleft E_1}, \ldots, T_k{}^{\curvearrowleft E_k} \,) \in \boldsymbol{R_i}$. If there is an automaton $\mathcal{A}$ recognising $\boldsymbol{A}$ and automata $\mathcal{R}_i = \mathcal{B}_i{}^{\mathcal{C}_i}$ recognising each of the encodings of $\boldsymbol{R_i}$ such that $\mathcal{B}_i$ only depends on the Σ-label of nodes in the convolution (*i.e.*

the transition function of $\mathcal{B}_i$ only considers the first projection of the labels in $\Sigma \times 2^{[1..n]}$), then we deem the structure to be:

- *isophilic* if $\mathcal{A}$ is a nested-*word* automaton (so the domain is a set of nested-*words*) and each of the $\mathcal{R}_i$ is a pathwise-nested automaton. This class is denoted by $\boldsymbol{Iso_2}$.
- *dendrisophilic* if the structure is a *graph*, $\mathcal{A}$ is a nested-word automaton and each of the $\mathcal{R}_i$ is a spine nested automaton. This class is denoted by $\boldsymbol{dIso_2}$.
- *symmetric-dendrisophilic* if the structure is a *graph*, $\mathcal{A}$ is a nested-word automaton and each of the $\mathcal{R}_i$ is a trunk nested automaton. This class is denoted by $\boldsymbol{sIso_2}$.
- *nondisophilic* if $\mathcal{A}$ is a nested-word automaton and each $\mathcal{R}_i$ are nested-tree automata. This class is called $\boldsymbol{nIso_2}$.
- *tree-isophilic* if all of the automata are pathwise-nested automata (the domain may consist of nested-trees, not just words). This class is denoted by $\boldsymbol{Iso_3}$.
- *tree-nondisophilic* if all of the automata are nested-tree automata. This class is denoted by $\boldsymbol{nIso_3}$.

(Whilst we will establish first-order decidability results for some of these classes when viewed as including relations of arbitrary arity, it should be emphasised that when characterising these classes using systems of (coloured) *graph* generators we obviously must assume that we are only dealing with those structures employing relations of arity at most two.)

The first three enjoy elegant characterisations in the form of prefix rewrite systems (for graphs) and the first two correspond precisely to the transition graphs of second-order (collapsible) automata. We suggest that tree-isophilic structures are a natural third member of the progression from word to tree automatic structures. They also subsume the third level of the Caucal hierarchy (although we lack a precise characterisation as with the second level). Tree-nondisophilic structures seem to be less interesting, but we introduce them as they provide the tool by which we get our decidability result for $\boldsymbol{FO}$ on a subclass of order-3 collapsible pushdown graphs.

It is also worth having the name *flat-isophilic* to refer to the special case of isophilic structures where the words are non-nested—thus the automata may all be considered finite-word/tree automata (with 'flat-nondisophilic' and 'flat-nondisophilic' amounting to the same thing). This class of structures is denoted by $\boldsymbol{Iso_1}$.

Let us say that a class C of structures is *closed under* $\boldsymbol{FO^\infty}$*-definability* if for every $\mathfrak{A} \in C$ and $\boldsymbol{FO^\infty}$-formula $\phi(x_1, \ldots, x_m)$, adding the relation defined by ϕ to $\mathfrak{A}$ also results in a structure belonging to C.

Theorem 6.26. $\boldsymbol{Iso_2}$, $\boldsymbol{nIso_2}$ *and* $\boldsymbol{Iso_3}$ *are closed under* $\boldsymbol{FO^\infty}$ *definability. When we restrict consideration to structures with domains contained in* $\mathbf{NTree}_k(\Sigma)$ *for some fixed* $k \in \mathbb{N}$*, tree-nondisophilic structures are also closed under* $\boldsymbol{FO^\infty}$ *definability.*

Proof. Consider any such structure $\mathfrak{A}$ and an $\boldsymbol{FO^\infty}$ formula $\phi(x_1, \ldots, x_m)$. Argue by induction on the structure of ϕ, constructing an appropriate automaton recognising the encoding of a relation defined by each subformula. Boolean operations are covered by Lemma 6.22.

The induction cases of $\exists$ and $\exists^\infty$ are covered by Lemma 6.25. We use a skeleton selector selecting every node in the convolution except for those associated exclusively with component of the relation being quantified. We then project the labels of the convolution tree—in the isophilic case this will retain the determinism of the nested-word automata since originally their transitions depend only on the Σ-component of labels in the convolution and the construction in the proof of Lemma 6.25 does not change this. □

Since dendrisophilic structures are just a special case of nondisophilic structures, emptiness testing together with Theorem 6.26 gives us:

Corollary 6.27. $\boldsymbol{FO}$ *and* $\boldsymbol{FO^\infty}$ *on isophilic, dendrisophilic, nondisophilic and tree-isophilic structures are decidable.*

Handling Tree-nondisophilicity and First-Order Logic

We run into difficulty when it comes to tree-nondisophilic structures as we have no Boolean closure in general. Instead we need to restrict ourselves to a subclass of these structures where relations can be determined by examining only the right-most k branches of a convolution for some fixed k. This enables us to appeal to the closure properties of nested-tree automata that act only on trees with a bounded number of branches. The inherent locality of first-order logic as encompassed by Gaifman's Locality Theorem [38] then allows us to obtain first-order decidability.

Gaifman's Locality Theorem

Let us fix some standard terminology relating to Gaifman Locality from [38].

Definition 6.28. Let $\mathfrak{A} = \langle \boldsymbol{A}, \boldsymbol{R_1}, \ldots, \boldsymbol{R_m} \rangle$ be a relational structure. The *Gaifman graph* $\mathsf{Gaif}(\mathfrak{A})$ of $\mathfrak{A}$ is the graph with domain $\boldsymbol{A}$ and an (undirected)

edge between $a, b \in \boldsymbol{A}$ whenever there exists $(c_1, c_2, \ldots, c_l) \in R_i$ for some $1 \leq i \leq m$ such that $a, b \in \{c_1, c_2, \ldots, c_l\}$.

Given $a \in \boldsymbol{A}$ the *local ball of radius* $r \in \mathbb{N}$ *about* a is the subset of $\boldsymbol{A}$:

$$B_r(a) := \{\ b \in \boldsymbol{A}\ :\ \text{there exists a path of at most length } r \text{ in } \mathbf{Gaif}(\mathfrak{A}) \text{ from } a \text{ to } b\ \}$$

Noting that edges in the Gaifman graph are definable by the same formula in all relational structures over a given signature and further noting that we can easily construct a formula $d_r(x, y)$ for each $r \in \mathbb{N}$ asserting that the shortest path between x and y in the Gaifman graph of the structure is of length at least r, the following definition makes sense:

Definition 6.29. An *r-local formula* is a first-order formula $\phi(x, \vec{y})$ with at least one free variable x whose quantifiers are restricted to $B_r(x)$. A *basic local sentence* is a sentence of the form:

$$\exists x_1 . \exists x_2 . \cdots \exists x_k . \left(\bigwedge_{i=1}^{k} \phi_i(x_i) \ \wedge \bigwedge_{1 \leq i < j \leq k} d_r(x_i, x_j) \right)$$

for some $r \in \mathbb{N}$ where the $\phi_i(x_i)$ are all r-local sentences.

Gaifman's Locality Theorem states the following:

Theorem 6.30 (Gaifman's Locality Theorem [38])**.** *Every first-order sentence is equivalent to a Boolean combination of basic local sentences.*

This allows us to partially reduce the decision problem for first-order logic to deciding first-order logic on local balls.

Locality In Tree-Nondisophilic Structures

Definition 6.31. Let us say that a set of n-ary convolutions R is *k-compact* for $k \in \mathbb{N}$ just in case there exists a nested-tree automaton $\mathcal{A}$ (acting on trees with at most k branches) such that:

$$C^{\frown E} \in R \quad \text{iff} \quad \mathcal{S}_k(C^{\frown E}) \in \mathcal{L}(\mathcal{A}) \text{ and each node of } C^{\frown E} \text{ not selected by } \mathcal{S}_k \text{ is associated with all of the } n \text{ trees yielding } C^{\frown E}$$

where $\mathcal{S}_k$ is the skeleton selector selecting the k right-most branches of a tree (thereby selecting a unique tree so that $\mathcal{S}_k(C^{\frown E})$ is a singleton set and so we may treat it as its element). We just say that a relation is *compact* if it is *k-compact* for some k.

In order to use this definition in the context of relational structures, it will be helpful to separate the two concerns of ensuring membership of the domain and determining whether elements are related. Thus for the purposes of this

section we will view relations on a domain as being n-tuples of arbitrary nested-trees, not necessarily restricted to those nested-trees belonging to the domain. The following definition captures the place where we *do* want to be able to consider membership of the domain as well (the second condition):

Definition 6.32. We say that a tree nondisophilic structure $\mathfrak{A} = \langle \boldsymbol{A}, \boldsymbol{R_1}, \ldots, \boldsymbol{R_m} \rangle$ is *compact* if both of the following conditions are met:

- Every relation $\boldsymbol{R_i}$ is compact for $1 \leq i \leq m$.
- The relation $\{ \ (a, b_1, b_2, \ldots, b_k) \ : \ b_1, b_2, \ldots, b_k \in B_r(a) \ \} \ \cap \ \boldsymbol{A}^{k+1}$ is compact for every $r \in \mathbb{N}$ on the assumption that $a \in \boldsymbol{A}$.

We can use Gaifman's Locality Theorem together with the closure properties of nested-tree automata on trees with a bounded number of branches (Lemma 6.22) to get:

Theorem 6.33. ***FO*** *is decidable on compact tree-nondisophilic structures.*

Proof. Let $\mathfrak{A} = \langle \boldsymbol{A}, \boldsymbol{R_1}, \ldots, \boldsymbol{R_l} \rangle$ be a compact tree-nodisophilic structure with domain recognised by $\mathcal{A}$. We begin by showing that for every r-local formula $\phi(x, y_1, \ldots, y_k)$ there exists a nested-tree automaton $\mathcal{A}^B_\phi$ acting on trees of bounded width m such that:

$$\begin{aligned}
&\{ \ C^{\frown E} \in \mathcal{L}(\mathcal{A}_\phi) \ : \ \mathcal{S}_m(C^{\frown E}) \in \mathcal{L}(\mathcal{A}^B_\phi) \text{ and } \pi_1(C^{\frown E}) \in \mathcal{L}(\mathcal{A}) \ \} \\
&= \Big\{ \bigotimes \langle S^{\frown D}, {T_1}^{\frown E_1}, \ldots, {T_k}^{\frown E_k} \rangle \ : \ \mathfrak{A} \vDash \phi({T_1}^{\frown E_1}, \ldots, {T_k}^{\frown E_k}, S^{\frown D}) \\
&\qquad\qquad \text{and } {T_i}^{\frown E_i} \in B_r(S^{\frown D}) \text{ for each } 1 \leq i \leq k \Big\}
\end{aligned}$$

where $\mathcal{A}_\phi$ is an automaton that ensures that all nodes not belonging to the right-most m branches of the convolution (*i.e.* not selected by $\mathcal{S}_m$) are associated with every tree making up the convolution (*i.e.* that the convolution does not split on these branches). It also ensures that the labels on these nodes of the tree are consistent with it being an n-ary convolution. It ignores the right-most m branches. By $\pi_1(C^{\frown E}) \in \mathcal{L}(\mathcal{A})$ we mean that the first component of the convolution is in $\mathcal{L}(\mathcal{A}) = \boldsymbol{A}$. We argue by induction on ϕ.

In the case when ϕ is atomic, it will be of the form $Ry_1 \cdots y_k$. Noting that equality $=$ is 0-compact (nothing needs to be checked to determine equality on the assumption that corresponding branches are the same in each tree being compared!) and from the assumption that $\mathfrak{A}$ is compact we get that the relation R is m-compact for some m. We also have it that the relation $y_1, \ldots, y_k \in B_r(x) \ \cap \ \boldsymbol{A}$ is m'-compact for some m'. Note that we may view both of these relations as being $\max(m, m')$-compact. If w.l.o.g. $m < m'$ then we simply extend the automaton acting on the right-most m branches of the convolution (recognising R) to act on the right-most m' branches, asserting

equality between the nodes not lying in the right-most m branches. We can thus take $\mathcal{A}^B_\phi$ to be the intersection of these two nested-tree automata acting on the right-most $\max(m, m')$ branches of the convolution.

For a conjunction $(\phi \wedge \psi)$ we can take $\mathcal{A}^B_{(\phi\wedge\psi)}$ the intersection of $\mathcal{A}^B_\phi$ and $\mathcal{A}^B_\psi$. Note that ϕ and ψ might contain variables not contained in the other. However the convolutions being recognised by $\mathcal{A}^B_\phi$ and $\mathcal{A}^B_\psi$ will both contain one variable in common, namely x, the centre of the local ball. So if $\mathcal{A}^B_\phi$ acts on the right-most m branches of a convolution $C^{\frown E}$ and $\mathcal{A}^B_\psi$ on the right-most m' branches of a convolution $C'^{\frown E'}$, then everything outside the right-most m branches of $C^{\frown E}$ must be shared by all elements, in particular x, and everything outside the right-most m' branches of $C'^{\frown E'}$ must be shared by all elements, in particular x. Since x is shared in both convolutions, merging them must result in a convolution for which everything outside the right-most $m + m'$ branches is shared by all elements in the convolution (including x). Thus the intersections and unions above can be recognised by automata acting on at most the right-most $m + m'$ branches of a convolution.

For negation we must obtain an automaton $\mathcal{A}_{\neg\phi}$ from $\mathcal{A}_\phi$. We do this by complementing $\mathcal{A}^B_\phi$ (possible due to Lemma 6.20) and intersecting with the automaton for $B_r(x) \cap \boldsymbol{A}$ in the same manner as for the atomic case. Because we are restricting the elements of the convolution to those belonging to $B_r(x)$ this is sound as membership of $B_r(x)$ requires being identical to x at nodes not read by $\mathcal{A}_\phi$.

Now we come to existential quantification—representing a formula of the form $\exists y.\phi(x, y, y_1, \ldots, y_k)$. Since this is assumed to be r-local about x the existential quantifier should only range over $B_r(x)$ $(\cap \boldsymbol{A})$. But this is exactly the constraint that $\mathcal{A}^B_\phi$ places on y. Since $y \not\equiv x$ (because the variable defining the centre of the ball is always kept free) we can be sure that the nodes of the convolution associated only with y would only have be read by $\mathcal{A}^B_\phi$ (and not by $\mathcal{A}$, which only reads x and not by $\mathcal{A}_\phi$ which must by assumption only accept when it has read nodes belonging to all elements of the convolution). We can thus obtain the requisite automata by projecting away nodes belonging only to y from $\mathcal{A}_\phi$ using Lemma 6.25 and for every automaton projecting node labels to eliminate references to y.

It follows that we can constructed a nested-tree automaton recognising the set of nodes defined by an r-local formula $\phi(x)$ with a single free variable representing the centre of its ball. Additionally note that the relation $d_r(x, y)$ holds iff $y \notin B_{r-1}(x)$. But by assumption membership of local balls is compact as witnessed by an automaton $\mathcal{A}_{B_{r-1}}$ reading the right-most m branches of a convolution. We thus have $y \notin B_{r-1}(x)$ if either the right-most m branches are rejected by $\mathcal{A}_{B_{r-1}}$ or there is a node differing between y and x outside the right-most m branches. The latter can easily be detected by a nested-tree

automaton and the former can be detected since $\mathcal{A}_{B_{r-1}}$ can be complemented due to the fact it acts on a bounded number of branches. We can thus construct a nested-tree automaton recognising the relation $d_r(x, y)$.

Since basic local sentences can be formed from such formulae without the need for negation (*i.e.* without the need of complementing automata) and since intersection and projection can be performed on all nested-tree automata, it then follows that for any given basic local sentence ϕ we can construct a nested-tree automaton that recognises a non-empty language iff $\mathfrak{A} \models \phi$. Thus by Gaifman's Locality Theorem (re-stated here as Theorem 6.30) we can decide an arbitrary first-order sentence in $\mathfrak{A}$ by deciding a finite number of basic local sentences and applying appropriate Boolean operations on the Boolean results for each (which do not need to be represented directly with automata). □

A crucial assumption for the proof above to work (subsumed by the definition of compactness) is that membership of $B_r(x) \cap \boldsymbol{A}$ is compact rather than just membership of $B_r(x)$ alone. This is because projection needs to be performed in a manner that does not interfere with the ability to detect membership of a defined relation on the basis of examining a bounded number of branches. In particular this means that membership of the domain needs to be compact with reference to the centre of our ball x, which is the only variable that will never be 'projected away' whilst forming the automata for our r-local formulae. The following Lemma will be useful in setting out a sufficient criterion for this to be possible in compact structures. It basically says of compact structures that if given an element $x \in \boldsymbol{A}$ one can determine whether $y \in \boldsymbol{A}$ by comparing y to x using a compact relation, then $B_r(x) \cap \boldsymbol{A}$ is also compact.

Lemma 6.34. *Let $\mathfrak{A}$ be a tree-nondisophilic structure with domain $\boldsymbol{A}$ consisting of nested-trees over an alphabet Σ. Suppose further that for each $m \in \mathbb{N}$ there is a compact relation $A_m \subseteq \mathbf{NTree}(\Sigma) \times \mathbf{NTree}(\Sigma)$ such that*

$$A_m \cap \boldsymbol{A} \times \mathbf{NTree}(\Sigma) = \{ (a, b) \in \boldsymbol{A} \times \boldsymbol{A} : \text{all nodes of } \bigotimes \langle a, b \rangle \text{ not in the right-most } m \text{ branches belong to both } a \text{ and } b \}$$

Then if all of the relations of $\mathfrak{A}$ are compact, $\mathfrak{A}$ is a compact structrure.

Proof. We just need to show that the $(k+1)$-ary relations $(y_1, y_2, \ldots, y_k) \in B_r(x) \cap \boldsymbol{A}^k$ where $x \in \boldsymbol{A}$ are compact.

First observe that the edge relation in $\mathsf{Gaif}(\mathfrak{A})$ must be recognised by nested-tree automata. This can be achieved by taking the nested-tree automaton recognising an n-ary relation R and applying the projection from Lemma 6.25 to it with a skeleton selector that chooses an arbitrary two members of the convolution. This is the only time when we will use a skeleton selector that

can non-deterministically choose multiple different skeletons for a given input tree. We can then take the union of all such automata derived from all the relations of $\mathfrak{A}$. In fact we can go further and say that the edges of $\mathsf{Gaif}(\mathfrak{A})$ are compact. This is due to the fact that the original relations are themselves compact and so we can recognise Gaifman edges by performing this projection on the automaton acting on the bounded number of right-most branches of a convolution. We then take the union of these automata acting on a bounded number of right-most branches. Note that we must also take the symmetric closure of this automaton (which is another union) as the edges of the Gaifman graph are not directed.

For any given r this allows us to construct an automaton recognising paths $x, z_1, z_2, \ldots, z_{r-2}, y$ of length r in the Gaifman graph indeed we can see that this r-ary relation is compact assuming that $x \in \boldsymbol{A}$. We argue by induction on r. When $r = 2$ the path is just x, y, so x and y must be related by a single Gaifman edge. Since Gaifman edges are compact, there must exist some $m \in \mathbb{N}$ such that all nodes of $\bigotimes \langle x, y \rangle$ not lying on the right-most m branches of the convolution must belong to both x and y. So we intersect the compact relation A_m that guarantees $y \in \boldsymbol{A}$ (assuming that $x \in \boldsymbol{A}$) with the compact edge relation of $\mathsf{Gaif}(\mathfrak{A})$. This intersection yields a compact relation for the same reasons as in the intersection case for local formulae in the proof of Theorem 6.33. The induction step reasons in exactly the same way, noting as part of the induction hypothesis that all elements in the path being extended must belong to $\boldsymbol{A}$ on the assumption that $x \in \boldsymbol{A}$.[1] We can then just apply projection from Lemma 6.25 on the intermediate nodes in the path in order to get the relation $y \in B_r(x) \cap \boldsymbol{A}$ on the assumption that $x \in \boldsymbol{A}$.

In order to extend to $y_1, \ldots, y_k \in B_r(x) \cap \boldsymbol{A}$ we simply observe that this is a finite intersection of k compact relations having a variable x in common, again in a similar manner to the intersection case in the proof of Theorem 6.33. □

As a corollary of Lemma 6.34 and Theorem 6.33 we get:

Corollary 6.35. *Let $\mathfrak{A}$ be a tree-nondisophilic structure with domain $\boldsymbol{A}$ consisting of nested-trees over an alphabet Σ whose relations are all compact. Suppose further that for each $m \in \mathbb{N}$ there is a compact relation $A_m \subseteq \mathbf{NTree}(\Sigma) \times$*

[1] Even if our structure is a graph consisting of reachable nodes, reachability in the Gaifman graph where we allow all possible nested-trees as nodes is still not necessarily the same as reachability in the structure. Thus additional assumptions have to be made in terms of assuming compactness of relative membership of $\boldsymbol{A}$ as going along a Gaifman edge that corresponds to a *backwards*-edge in the original graph requires some additional information to ensure the node could be obtained via *forwards* reachability from the origin.

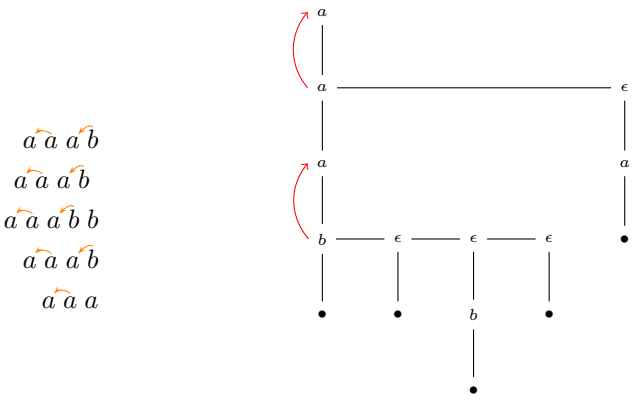


Figure 6.10: Isophilic Chain

NTree(Σ) *such that*

$$A_m \cap \boldsymbol{A} \times \mathbf{NTree}(\Sigma) = \{\ (a,b) \in \boldsymbol{A} \times \boldsymbol{A}\ :$$
$$\textit{all nodes of } \bigotimes \langle a,b\rangle \textit{ not in the right-most } m$$
$$\textit{branches belong to both } a \textit{ and } b\}$$

Then ***FO*** *is decidable on* $\mathfrak{A}$.

6.3 Isophilic and (Symmetric-)Dendrisophilic Chains

For this section we restrict our attention to isophilic, dendrisophilic and symmetric dendrisophilic *graphs*; so only (nested-)words make up the domains and all relations are either binary or unary. The fact that convolutions must branch as soon as components begin to differ means that an arbitrary number of convolutions can be 'glued together' whilst retaining a finite alphabet. This allows us to represent *paths* in isophilic and dendrisophilic graphs as trees denoting *chains* of words, which in turn can be recognised by automata.

A *path* in a graph $\mathfrak{A}$ is a sequence of nodes $u_1, \ldots, u_m$ such that for $1 \leq i < m$ there is an edge from u_i to u_{i+1}. We say that $u\boldsymbol{r}u'$ if u' is *reachable* from u—that is there is a path from u to u'. We say that $u\boldsymbol{r}^{\infty}u'$ if there exist *infinitely many* distinct paths from u to u'—note that if there is a cycle in the graph on some path between u and u' this automatically implies $u\boldsymbol{r}^{\infty}u'$ (although not conversely in an infinite graph).

Definition 6.36. For each graph $\mathfrak{A}$ let $\mathcal{R}_{\mathfrak{A}}$ be the set of its binary relations. A $\mathcal{Q}$-*decorated structure* $\mathfrak{A}_{\mathcal{Q}}$ adds a map $\mathcal{Q} : \mathcal{R}_{\mathfrak{A}} \longrightarrow Q \times Q$ for some finite set Q. This allows for a finely controlled notion of path. A path $u_1, \ldots, u_m$ in $\mathfrak{A}$ is deemed $\mathcal{Q}$-*compatible* if there is a function $f : [1..m] \longrightarrow Q$ and relations

$R_1, \ldots, R_{m-1}$ such that for every $1 \leq i < m$ we have $u_i R_i u_{i+1}$ and $\mathcal{Q}(R_i) = (f(i), f(i+1))$.

We say that $u\boldsymbol{r}_{\boldsymbol{q},\boldsymbol{q'}}u'$ if there is a $\mathcal{Q}$-compatible path p beginning with u and ending in u' for which $f(1) = q$ and $f(|p|) = q'$ and $u\boldsymbol{r}^{\boldsymbol{\infty}}_{\boldsymbol{q},\boldsymbol{q'}}u'$ if there exist infinitely many such paths. Note $\boldsymbol{r}$ and $\boldsymbol{r}^{\boldsymbol{\infty}}$ arise when Q is a singleton.

Example 6.37. The motivating example is when the nested(-words) $s_1, s_2, \ldots, s_k$ represent the contents of a stack of a pushdown automaton and we take Q to be its set of control states. The relations would then represent stack operations and a suitable $\mathcal{Q}$ would assign an origin and target control-state that accompany that each transition. If $s_1, s_2, \ldots, s_k$ forms a $\mathcal{Q}$-compatible path, then this means that there exists control-states $q_1, q_2, \ldots, q_k \in Q$ such that $(q_1, s_1), (q_2, s_2), \ldots, (q_k, s_k)$ is a run of the automaton. Likewise $s\boldsymbol{r}_{\boldsymbol{q},\boldsymbol{q'}}s'$ would mean that there is a run from (q, s) to (q', s') and $s\boldsymbol{r}^{\boldsymbol{\infty}}_{\boldsymbol{q},\boldsymbol{q'}}s'$ would mean that there are infinitely many runs from (q, s) to (q', s').

Branch Chains

Assume the nested-words in $\mathfrak{A}$'s domain use the alphabet Σ. A *(symmetric)-dendrisophilic (*resp. *isophilic) chain* over $\mathfrak{A}_{\mathcal{Q}}$ is a $\Sigma \cup \{\epsilon, \bullet\}$-labelled nested-tree $C^{\curvearrowleft E}$ such that:

- It is a binary tree.
- Every node $u \in \mathbf{dom}(C)$ with $C(u) = \bullet$ satisfies $u \notin \mathbf{dom}(E)$ and $u \notin \mathbf{img}(E)$ and may have no children.
- If a node is not labelled $\bullet$, then it must have a child.
- Every node $u \in \mathbf{dom}(C)$ with $C(u) = \epsilon$ satisfies $u \notin \mathbf{dom}(E)$ and $u \notin \mathbf{img}(E)$ and cannot be the root.
- The *left* child *or only* child of a node may *not* be ϵ-labelled. If a node has two children, then the *right* child *must* be ϵ-labelled.
- For $1 \leq i < wth(C^{\curvearrowleft E})$ let $T_i{}^{\curvearrowleft E_i}$ be the subtree consisting of just the i and $(i+1)$th branches of $C^{\curvearrowleft E}$. It should be the case that $\pi_\Sigma(T_i{}^{\curvearrowleft E_i}) = \pi_1\left(\bigotimes \left\langle\, C^{\curvearrowleft E}_{\prec_i}, C^{\curvearrowleft E}_{\prec_{i+1}} \,\right\rangle\right)$. The value $wth(C^{\curvearrowleft E})$ is the number of leaves of $C^{\curvearrowleft E}$.
- $\pi_\Sigma(C^{\curvearrowleft E}_{\prec_1}), \ldots, \pi_\Sigma(C^{\curvearrowleft E}_{\prec_{wth(C^{\curvearrowleft E})}})$ forms a $\mathcal{Q}$-compatible path in $\mathfrak{A}$.

where we write $\pi_\Sigma(T^{\curvearrowleft E})$ to denote the tree formed by deleting all nodes not labelled in Σ under the assumption that all nodes in $T^{\curvearrowleft E}$ not labelled in Σ have at most one child.

Let us denote the space of (symmetric-)dendrisophilic (*resp.* isophilic) chains over $\mathfrak{A}_{\mathcal{Q}}$ by $\boldsymbol{Ch}(\mathfrak{A}_{\mathcal{Q}})$.

Lemma 6.38. *For any $\mathcal{Q}$-decorated symmetric-dendrisophilic, dendrisophilic or isophilic graph $\mathfrak{A}_{\mathcal{Q}}$ it is the case that:*

1. *There exists respectively a trunk nested, spine nested or path-nested automaton recognising $\boldsymbol{Ch}(\mathfrak{A})$.*
2. *Let $u_1, \ldots, u_m$ in $\mathfrak{A}$ be a $\mathcal{Q}$-compatible path in $\mathfrak{A}$. Then there exists a* unique *$C^{\curvearrowleft E} \in \boldsymbol{Ch}(\mathfrak{A})$ such that $wth(C^{\curvearrowleft E}) = m$ and $u_i = \pi_\Sigma(C_i{}^{\curvearrowleft E_i})$ for each $1 \leq i \leq m$. The converse also holds.*

Proof. We begin with the most general case—when the graph is symmetric-dendrisophilic. Let $\mathcal{A}_{p,q} = \mathcal{B}_{p,q}{}^{\mathcal{C}_{p,q}}$ be a trunk nested automaton recognising the union of all relations decorated by $p, q \in Q$ (where $Q \times Q$ is the image of $\mathcal{Q}$). Recall that this means that the $\mathcal{B}_{p,q}$ are deterministic along the trunks of trees (the initial segment of the tree that occurs prior to any branching). It is also worth reminding the reader that the convolutions read by $\mathcal{A}_{p,q}$ are binary relations between nested-words, so the trees upon which they act have at most two branches. We refer to the component leaving the trunk to the left as the *left-branch* and the component leaving the trunk to the right as the *right-branch*.

We define an automaton $\mathcal{A} = \mathcal{B}^{\mathcal{C}}$ recognising $\boldsymbol{Ch}(\mathfrak{A})$. This will act on a trees with an arbitrary number of branches where adjacent branches should be related in a manner that would be accepted by one of the $\mathcal{A}_{p,q}$. It has control-states $Q_{\mathcal{B}}$ where:

$$Q_{\mathcal{B}} = \underbrace{\prod_{\in Q\times Q}^{(p,q)} Q_{\mathcal{B}_{p,q}}}_{\text{Trunk-Sim}} \times \underbrace{\prod_{\in Q\times Q}^{(p,q)} Q^{\perp}_{\mathcal{B}_{p,q}} \times \mathbb{B}}_{\text{Left-Branch Sim}} \times \underbrace{\prod_{\in Q\times Q}^{(p,q)} Q^{\perp}_{\mathcal{B}_{p,q}} \times \mathbb{B}}_{\text{Right-Branch Sim}} \times \Sigma^{\perp}$$

The set $Q_{\mathcal{B}_{p,q}}$ is the set of control-states of the automaton $\mathcal{B}_{p,q}$. For a set S we write $S^{\perp}$ to denote $S \cup \{\perp\}$. We use the value $\perp$ when the value is 'irrelevant' or undefined.

The idea is that the *trunk sim* component will always simulate the behaviour that each $\mathcal{B}_{p,q}$ would exhibit when reading the trunk of a binary convolution. Since $\mathcal{A}_{p,q}$ is a trunk nested automaton, by definition this behaviour must be deterministic. This means that $\mathcal{B}_{p,q}$ is able to carry out this simulation despite a node in a chain potentially belonging to the trunk of an unbounded number of binary convolutions. The *left-branch sim* and the *right-branch sim* components will respectively simulate the behaviour of each $\mathcal{B}_{p,q}$ at the left or right branch of a binary convolution. Behaviour here may be non-deterministic, but this is not a problem as any given node in a chain will belong to the left-branch of at most one convolution and the right-branch of at most one convolution. This fact will be exhibited when we describe the transition functions of $\mathcal{B}$ which is able to soundly carry out this simulation without any conflict.

The Boolean flag $\mathbb{B}$ associated with the left and right branch simulations is used to keep track of whether any pointer encountered during the left/right simulation should be treated as a pointer to the trunk of the convolution or a pointer to the left/right branch of the convolution. A value $\mathbf{f}$ means that subsequent pointer sources should be simulated as pointing to a target in the trunk and a value $\mathbf{t}$ means that subsequent pointer sources should be treated as pointing to a target in the left/right branch.

The Σ component is just used to remember the most recent non-ϵ symbol visited. This is useful when skipping over ϵ nodes in a chain.

The automaton $\mathcal{B}$ begins in state $(q_{0\vec{p,q}}, \vec{\perp}, \mathbf{f}, \vec{\perp}, \mathbf{f}, \perp)$ where $q_{0\vec{p,q}}$ is the vector of initial states for each $\mathcal{B}_{p,q}$. It then transitions as follows:

- The *trunk sim* component is updated according to the transition functions of the $\mathcal{B}_{p,q}$ as they would act when a node has at most one child. If transitioning from a node that is the source of the pointer, the trunk component of the target is used as the target state. This results in it simulating the behaviour of the automata under the assumption that they never come accross any branching. This update must be deterministic. Any nodes labelled ϵ in the chain are simply ignored by this component. The Σ component is set to the node label just read whenever this node-label belongs to Σ (*i.e.* is not ϵ).

- The automaton only allows nodes of the input tree to have at most two children. When such a node is reached it simulates reaching the single point of branching in a binary convolution. Let $a \in \Sigma$ be the current node symbol if this belongs to Σ or the value in the Σ component of the control state if the current node symbol is ϵ. Let $s_{p,q}$ be the state simulated in the *trunk* component for $\mathcal{B}_{p,q}$. For each pair $(p,q) \in Q \times Q$ the automaton should non-deterministically pick a pair of states $(s^l_{p,q}, s^r_{p,q}) \in Q_{\mathcal{B}_{p,q}} \times Q_{\mathcal{B}_{p,q}}$ belonging to the set given by the transition function of $\mathcal{B}_{p,q}$ acting on an a-labelled node in state $s_{p,q}$ and appropriate for the current node's pointer status. If the current node is the source of a pointer, the state in the trunk component of the target is used as the target state. The *left sim* component of the *left child* should be set to $(\vec{s^l_{p,q}}, \mathbf{f})$ and the *right sim* component of the *right child* should be set to $(\vec{s^r_{p,q}}, \mathbf{f})$. The value $\mathbf{f}$ is correct for the Boolean flags since we have not yet passed any node that is the target of a pointer since beginning this fresh left/right branch simulation.

- The *left sim* and *right sim* components are updated non-deterministically according to the transition functions of the $\mathcal{B}_{p,q}$ when transitioning from a node with at most one child. Nodes labelled ϵ are ignored. The Boolean flag in each component is set to $\mathbf{t}$ when transitioning from a node that

is the target of a pointer. This is correct behaviour as a source node occurring subsequently must point to a node visited after the start of this left/right branch simulation. When transitioning from the source of a node, the Boolean flag is set to the value of the Boolean flag in the corresponding component at the target of the pointer.

- When the current node has two children, the *left sim* simulation is propagated down the *right* child and the *right sim* simulation is propagated down the *left* child. This is correct behaviour since the left component of one of the binary convolutions in the chain will be the right-most branch beginning at the left child of the branching point of the convolution and the right component will similarly be the left-most branch beginning at the right child of the branching point of the convolution. Note further that this does not conflict with the spwaning of fresh left and right-sims in the second item, which are spawned in the opposite directions.

Note that $\mathcal{C}_{p,q}$ has no obligation to be deterministic anywhere on its input tree. However, in contrast to $\mathcal{B}$ whose transitions can depend on states at non-local nodes in the tree (which is why this proof does not work for general non-deterministic automata), the locality of the dependencies for transitions of $\mathcal{C}_{p,q}$ allows us to make it deterministic along the trunk of a convolution (before it splits) using the power-set construction. This enables us to simulate each $\mathcal{C}_{p,q}$ in the same manner as the simulation for each $\mathcal{B}_{p,q}$. In fact the simulation is a little simpler as it does not need to lookup the state at the targets of pointers and so the Boolean flags are not necessary. The only other difference is that we simulate only one of the $\mathcal{C}_{p,q}$ in the left and right branch simulations rather than all of them simultaneously as we did with $\mathcal{B}$. This is reflected in the fact that we have taken unions rather than products. This is important since it forces $\mathcal{C}$ to decide which $\mathcal{C}_{p,q}$ it is going to use at each convolution in the chain, which amounts to guessing the $\mathcal{Q}$-decoration that will be placed on the elements of the chain.

Define $\mathcal{C}$ to have state space:

$$Q_{\mathcal{C}} := \underbrace{\prod_{\substack{(p,q)\\ \in Q\times Q}} 2^{Q_{\mathcal{C}_{p,q}}}}_{\text{trunk-sim}} \times \underbrace{\left(\bigcup_{\substack{(p,q)\\ \in Q\times Q}} Q_{\mathcal{C}_{(p,q)}}\right)^{\perp}}_{\text{left-branch-sim}} \times \underbrace{\left(\bigcup_{\substack{(p,q)\\ \in Q\times Q}} Q_{\mathcal{C}_{(p,q)}}\right)^{\perp}}_{\text{right-branch-sim}} \times \Sigma$$

The initial state of $\mathcal{Q}_{\mathcal{C}}$ is $(\vec{I_{p,q}}, \perp, \perp)$ (where $\vec{I_{p,q}}$ are the sets of initial states of each $Q_{\mathcal{C}_{p,q}}$). The trunk-sim component behaves the same on all branches of the chain, keeping track of all possible states that each $\mathcal{Q}_{p,q}$ could be in from the root of a convolution and prior to it splitting.

The *left*-branch-sim always propagates along the *right* child or *only* child and the *right*-branch-sim always propagates along the *left* child or *only* child, simulating behaviour *after* the convolution has split. When the chain branches, some $(p, q) \in Q \times Q$ is chosen and a fresh left-branch-sim is propagated down the *left* child and a fresh right-branch-sim down the *right* child based on a permissible transition of $\mathcal{C}_{p,q}$ at the splitting point of a convolution from one of the states in the trunk-sim component. Note that this commits the simulation to a *particular* $\mathcal{C}_{(p,q)}$ connecting these components of the chain.

The automaton $\mathcal{C}$ has access to the states of $\mathcal{B}$ to enable its simulation to depend on the simulated $\mathcal{B}_{p,q}$. It accepts if all of its left-branch-sim and right-branch-sim components simulate accepting states with the additional condition that if the left-branch-sim at a leaf is simulating $\mathcal{C}_{p,q}$, then the right-branch-sim of that same leaf must be simulating $\mathcal{C}_{q,r}$ for some $r \in Q$.

So this gives the requisite trunk automaton $\mathcal{B}^{\mathcal{C}}$. If we are considering dendrisophilic graphs so that the $\mathcal{B}_{p,q}^{\mathcal{C}_{p,q}}$ are spine-nested automata, then we can make $\mathcal{B}^{\mathcal{C}}$ a spine-nested automaton as well by simply removing the right-sim component from the states of $\mathcal{B}$. We could rename the 'trunk sim' component the 'spine sim' component. The simulation in the right-sim component of $\mathcal{C}$ will look at the state in the trunk-sim of $\mathcal{B}$ all of the time, which is correct by the definition of spine nested automata.

For isophilic graphs we can make $\mathcal{B}^{\mathcal{C}}$ path-nested by removing both the right-sim *and* left-sim components of $\mathcal{B}$. We could rename the 'trunk sim' component the 'path sim' component. Both the left and right-sim components of $\mathcal{C}$ refer to the path sim simulation of $\mathcal{B}$ all of the time, which is adequate for path-nested automata. □

We are now in a position to show that reachability is definable in symetric-dendrisophilic, dendrisophilic and isophilic graphs.

Theorem 6.39. *Let $\mathfrak{A}$ be a symetric-dendrisophilic, dendrisophilic or isophilic graph. The graph obtained by adding the relations $\boldsymbol{r}$ and $\boldsymbol{r}^{\boldsymbol{\infty}}$ is also respectively symmetric-dendrisophilic, dendrisophilic or isophilic. Where $\mathfrak{A}_{\mathcal{Q}}$ is $\mathcal{Q}$-decorated the same applies to $\boldsymbol{r}_{\boldsymbol{q},\boldsymbol{q'}}$ and $\boldsymbol{r}^{\boldsymbol{\infty}}_{\boldsymbol{q},\boldsymbol{q'}}$ for each $q, q' \in Q$.*

Proof. First consider the $\mathcal{Q}$-decorated case. In order to get an automaton recognising $\boldsymbol{r}_{\boldsymbol{q},\boldsymbol{q'}}$ we first obtain an automaton recognising $\boldsymbol{Ch}(\mathfrak{A})$ using Lemma 6.38. We modify this to ensure that the simulation on the left-most binary convolution is of $\mathcal{C}_{q,r}$ for some $r \in Q$ and that the simulation on the right-most binary convolution is of $\mathcal{C}_{r',q'}$ for some r'. We then project onto the exoskeleton of the chain using Lemma 6.25. The only remaining issues are to add indices $\{1\}, \{2\}, \{1, 2\}$ to the nodes of the tree recognised in order to give it the form of a binary convolution; to ignore the $\bullet$s at the end of the branches, both of which are trivial, and also remove the ϵ symbols that may appear in the right-branch.

The last item can be achieved by first noting any node labelled ϵ can be neither the source nor the target of the pointer. The automaton can thus easily be modified to ignore ϵ-transitions.

In order to get an automaton for $\boldsymbol{r}^{\infty}_{q,q'}$ we do exactly the same thing except that when applying Lemma 6.25 using the exoskeleton we opt for the version of projection asserting the existence of an infinite number of trees with the given skeleton.

The $\boldsymbol{r}$ and $\boldsymbol{r}^{\infty}$ relations are just special cases of the above when Q is a singleton set. □

Flat Isophilic Chains based on Summaries

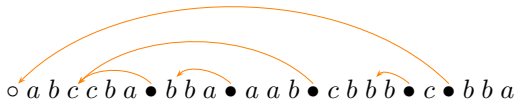
We also define a notion of *flat isophilic chain*, which takes the form of a *semi*-nested-word representing a path in a *flat isophilic* structure. There is no dendrisophilic counterpart since flat isophilic structures are by definition formed from ordinary non-nested words and so the distinction collapses. This is essentially the nested-word representation of the isophilic chain defined above (noting that in this special case it would be a non-nested tree). Formally, given a $\mathcal{Q}$-indexed flat isophilic structure $\mathfrak{A}_{\mathcal{Q}}$ (defined in the same way as before) a *flat isophilic chain* is a *semi*-nested-word $c^{\curvearrowleft E}$ over the alphabet $\Sigma \cup \{\circ, \bullet\}$ such that:

- $c(0) = \circ$ and $i \in \mathbf{dom}(E)$ iff $c(i) = \bullet$.
- For each $i \in \mathbf{dom}(E)$ it must be the case that $c(succ(i))$ is distinct from the label of the position coming after $E(i)$ (if it exists) in $\ulcorner c^{\curvearrowleft E}{}_{<i} \urcorner$.
- Suppose that $c_1 \blacktriangleleft c_2 \blacktriangleleft \cdots \blacktriangleleft c_m$ is the summary ordering of $c^{\curvearrowleft E}$. We require $\pi_{\Sigma}(c_1), \ldots, \pi_{\Sigma}(c_m)$ to be a path in $\mathfrak{A}$ and moreover there to exist a function $f : [1..m] \longrightarrow Q$ such that $\pi_{\Sigma}(c_i) R_i \pi_{\Sigma}(c_{i+1})$ for some relations of the form $R_i \in \mathcal{Q}(f(i), f(i+1))$.

Example 6.40. Suppose we have the following sequence of non-nested words:

$$abccba, abcbba, abcbaab, abccbbb, abccbbc, bba$$

This flat-isophilic chain is encoded as the following semi-nested word:



The set of flat-isophilic chains of $\mathfrak{A}$ is denoted by $\boldsymbol{Ch}_{\blacktriangleleft}(\mathfrak{A})$. We have an analogue of Lemma 6.38:

Lemma 6.41. *Let $\mathfrak{A}$ be a $\mathcal{Q}$-decorated flat-isophilic structure.*

1. *There exists a semi-nested-word automaton recognising* $\boldsymbol{Ch}_{\blacktriangleleft}(\mathfrak{A})$.

2. *For every* $\mathcal{Q}$*-compatible path in* $\mathfrak{A}$ *there exists a unique corresponding element in* $\boldsymbol{Ch}_{\blacktriangleleft}(\mathfrak{A})$ *(and conversely).*

Proof. This is a very similar idea to proof of Lemma 6.38. Indeed the construction is essentially the same as the tree automaton given in the proof of Lemma 6.38 mapped to a semi-nested-word automaton recognising precisely the semi-nested-words encoding the trees recognised by the tree automaton.

Take $\mathcal{C}_{p,q}$ to be the union of the finite tree automata recognising the convolutions of relations decorated by (p,q) for each $(p,q) \in Q \times Q$, the co-domain of $\mathcal{Q}$.

The semi-nested-word automaton $\mathcal{C}$ recognising $\boldsymbol{Ch}_{\blacktriangleleft}(\mathfrak{A})$ is given state-space:

$$Q_{\mathcal{C}} := \underbrace{\prod_{\in Q\times Q}^{(p,q)} 2^{Q_{\mathcal{C}_{p,q}}}}_{\text{trunk-sim}} \times \underbrace{\left(\prod_{\in Q\times Q}^{(p,q)} 2^{Q_{\mathcal{C}_{p,q}}\times Q_{\mathcal{C}_{p,q}}}\right)^{\perp}}_{\text{left-branch-sim}} \times \underbrace{\left(\bigcup_{\in Q\times Q}^{(p,q)} Q_{\mathcal{C}_{(p,q)}}\right)^{\perp}}_{\text{right-branch-sim}}$$

The initial state is $(\vec{I_{(p,q)}}, \perp, \perp)$ where $I_{p,q}$ is the set of initial states of $\mathcal{C}_{p,q}$.

Suppose that $\mathcal{Q}_{\mathcal{C}}$ has read the first segment u of a semi-nested-word. The trunk-sim deterministically keeps track of the state that $\mathcal{C}_{p,q}$ would be in if it read $\ulcorner u \urcorner$ from its initial state. In order to do this, it ignores the targets of pointers but at the source of a pointer v it will pick up from where it left off at $E(v)$ rather than from the predecessor of v.

Whenever transitioning to the *successor of a target* a fresh *left*-branch-sim is started. If u' is the segment of the word that has been read since and including the immediate successor of the target, then the left-branch-sim component keeps track of the pairs (r, r') for each $\mathcal{C}_{p,q}$ that means the automaton could have started in state r and read $\ulcorner u' \urcorner$ ending in state r'.

Whenever transition is made from the *successor of a source* v a fresh *right*-branch-sim is started. This simulates a specific $\mathcal{C}_{p,q}$ which may be chosen arbitrarily. The state the simulation begins with must meet the following constraints: (p,q) may be arbitrary if the previous right-branch-sim was $\perp$, otherwise only q may be arbitrary and p must be such that the previous right-branch-sim was simulating (r,p) for some $r \in Q$. The actual simulated state c is chosen so that there exists some pair (a,b) in the current left-branch-sim of $Q_{\mathcal{C}_{p,q}}$ such that b is accepting for $Q_{\mathcal{C}_{p,q}}$ and there exists some state d in the trunk-sim component of the state at $E(v)$ such that $\mathcal{C}_{p,q}$ could branch from d to (a,c).

Whenever a fresh right-branch-sim is begun $Q_{\mathcal{C}}$ must also ensure that the previous right-branch-sim ended terminated with an accepting state and reject otherwise.

(Note the asymmetry between the left-branch-sim and right-branch-sim is necessary since in a semi-nested-word multiple pointers may share the same target and so we need to 'determinise' the left-branch-sim in order to cover the multiple corresponding right-branch-sims whose associated left-branch sims all start from the same point.) □

Whilst this could be used to show that $\boldsymbol{r}$ and $\boldsymbol{r}^{\boldsymbol{\infty}}$ can be added to flat-isophilic structures, this can already be seen by applying Theorem 6.39 to this special case. The utility of flat-isophilic chains lies in the fact that they themselves naturally form isophilic and dendrisophilic structures.

6.4 Graphs Constructed From Chains

Graphs from Flat Chains

Consider a $\mathcal{Q}$-decorated flat-isophilic graph $\mathfrak{A}_{\mathcal{Q}}$. Let $\mathcal{R}_{\mathfrak{A}}$ be the set of its *binary* relations and let $\mathcal{P}_{\mathfrak{A}}$ be the set of its *unary* predicates. Let us also consider two maps $\mathcal{Q}_{pop}, \mathcal{Q}_{push} : \mathcal{R}_{\mathfrak{A}} \longrightarrow Q \times Q$ and two further maps $\mathcal{Q}_{panic}, \mathcal{Q}_{calm} : \mathcal{P}_{\mathfrak{A}} \times \mathcal{P}_{\mathfrak{A}} \longrightarrow Q \times Q$. The structure $\boldsymbol{Ch}^{\boldsymbol{I}}_{\blacktriangleleft}(\mathfrak{A}_{\mathcal{Q}}, \mathcal{Q}_{pop}, \mathcal{Q}_{push})$ has domain $\boldsymbol{Ch}_{\blacktriangleleft}(\mathfrak{A})$, and decoration $\mathcal{Q}_{\blacktriangleleft}$ where it has relations:

- For each binary relation R of $\mathfrak{A}$ a relation $R_{\blacktriangleleft}$ relating each encoding of the path $w_1, \ldots, w_{m-1}, w_m$ to the encoding of the path $w_1, \ldots, w_{m-1}, w'_m$ where $w_m R w'_m$. We set $\mathcal{Q}_{\blacktriangleleft}(R_{\blacktriangleleft}) := \mathcal{Q}(R)$.
- For each binary relation R of $\mathfrak{A}$ a relation R_{pop} relating the encoding of a path $w_1, \ldots, w_{m-1}, w_m$ to $w_1, \ldots, w_{m-1}$ whenever $w_{m-1} R w_m$. We set $\mathcal{Q}_{\blacktriangleleft}(R_{pop}) := \mathcal{Q}_{pop}(R)$.
- For each $(p, q) \in Q_{push}$ a relation R_{push} relating the encoding of a path $w_1, \ldots, w_{m-1}, w_m$ to $w_1, \ldots, w_{m-1}, w_m, w'_m$ whenever $w_m R w'_m$. We define $\mathcal{Q}_{\blacktriangleleft}(R_{push}) := \mathcal{Q}_{push}(R)$.

We will now consider some extensions of this structure with additional relations:

- For each pair of *unary* predicates P_1 and P_2 we have a relation $(P_1, P_2)_{panic}$ that relates every encoding of a path $w_1, \ldots, w_{m-1}, w_m$ where $w_m \in P_1$ to the encoding of the shortest sub-path $w_1, \ldots, w_r$ such that $r < m$ and w_m is a prefix of w_i for every $r < i \leq m$ with w_m *not* a prefix of w_r, and $w_r \in P_2$. We set $\mathcal{Q}_{\blacktriangleleft}((P_1, P_2)_{panic}) := \mathcal{Q}_{panic}(P_1, P_2)$. The 'panic' terminology is inspired by [53].

- For each pair of *unary* predicates P_1 and P_2 we have a relation $(P_1, P_2)_{calm}$ that is the inverse of $(P_2, P_1)_{panic}$—*i.e.* $c(P_1, P_2)_{calm}c'$ iff $c'(P_2, P_1)_{panic}c$. We set $\mathcal{Q}_{\blacktriangleleft}((P_1, P_2)_{calm}) := \mathcal{Q}_{calm}(P_1, P_2)$.

The structure $\boldsymbol{Ch}^{\boldsymbol{D}}_{\blacktriangleleft}(\mathfrak{A}, \mathcal{Q}_{pop}, \mathcal{Q}_{push}, \mathcal{Q}_{panic})$ extends $\boldsymbol{Ch}^{\boldsymbol{I}}_{\blacktriangleleft}(\mathfrak{A}, \mathcal{Q}_{pop}, \mathcal{Q}_{push})$ with the relation $(P_1, P_2)_{panic}$ for every $P_1, P_2 \in \mathcal{P}_{\mathfrak{A}}$.

The structure $\boldsymbol{Ch}^{\boldsymbol{S}}_{\blacktriangleleft}(\mathfrak{A}, \mathcal{Q}_{pop}, \mathcal{Q}_{push}, \mathcal{Q}_{panic}, \mathcal{Q}_{calm})$ extends $\boldsymbol{Ch}^{\boldsymbol{D}}_{\blacktriangleleft}(\mathfrak{A}, \mathcal{Q}_{pop}, \mathcal{Q}_{push}, \mathcal{Q}_{panic})$ with the relation $(P_1, P_2)_{calm}$ for every $P_1, P_2 \in \mathcal{P}$.

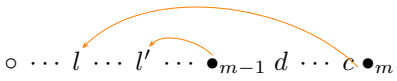
First we give a lemma explaining how to 'compute' the sub-chain resulting from the *panic* (and hence also *calm*) relation given a starting chain.

Lemma 6.42. *Let c be a chain encoding $w_1, \ldots, w_{m-1}, w_m$ where $m \geq 2$. Then there exists a unique chain c' encoding $w_1, \ldots, w_r$ such that $r < m$ and w_m is a prefix of w_i for every $r < i \leq m$ and such that r is minimal. If the final position of c is not $\bullet$ then $r = m - 1$. Otherwise it is $\bullet$ and hence the source of a pointer with target l. Then the unique c' relating to c in this way is the subword of c finishing with the right-most $\bullet$ or $\circ$ symbol to the left of l.*

Proof. We argue by induction on m. The case when $m = 2$ is immediate from the fact that $r = 1$ is the only shorter chain and trivially satisfies the property since $i = 2$ is the only possible value for i. For the induction step suppose first that c does not end in $\bullet$. This means that c has the form:

where no $\bullet$ occurs amongst the final $\cdots$. The definition of chain ensures that $\pi_\Sigma(\ulcorner\circ \cdots (l)l \cdots (b-l)\bullet\urcorner)$ is the largest common prefix of w_{m-1} and w_m as a must be different to whatever follows l in $\pi_\Sigma(\ulcorner\circ \cdots (l)l \cdots c\urcorner)$. Thus w_m cannot be a prefix of w_{m-1} and so $r = m - 1$ is the necessary value of r (so that i only takes the value m).

Suppose now that c has the form:



where the right-most two $\cdots$ do not contain any $\bullet$ symbols. Thus we can see that w_m is a prefix of w_{m-1}. Now let us adjust the chain by replacing w_{m-1} with w'_{m-1} formed by moving l' to the left to the same position as l and removing the right-most $d \cdots c$ segment. Indeed $w'_{m-1} = w_m$. The r given by the induction hypothesis to this modified chain is thus correct for $w'_{m-1} = w_m$. But since w_m is a prefix of the original w_{m-1} it must be correct for the original chain too.

The only other form that c could take is:

$$\circ \cdots \bullet_{m-1} \; a \cdots l \cdots \bullet_m$$

where the right-most $\cdots$ does not contain any $\bullet$ symbol. By the induction hypothesis $\pi_\Sigma(\ulcorner \circ \cdots \bullet_{m-1} \; a \cdots l \urcorner)$ is not a prefix of w_{m-2} and so neither can be w_m. Thus again $r = m-1$ is the appropriate value for r, as required. □

Now we can place the chain derived structures in our hierarchy.

Theorem 6.43. *Let $\mathfrak{A}_\mathcal{Q}$ be a Q-decorated flat-isophilic graph. Then for every $\mathcal{Q}_{pop}, \mathcal{Q}_{push} : \mathcal{R}_\mathfrak{A} \longrightarrow Q \times Q$ and $\mathcal{Q}_{panic}, \mathcal{Q}_{calm} \subseteq Q \times Q$ the structure $\boldsymbol{Ch}^{\boldsymbol{S}}_{\blacktriangleleft}(\mathfrak{A}, \mathcal{Q}_{pop}, \mathcal{Q}_{push}, \mathcal{Q}_{panic}, \mathcal{Q}_{calm})$ is symmetric-dendrisophilic, $\boldsymbol{Ch}^{\boldsymbol{D}}_{\blacktriangleleft}(\mathfrak{A}, \mathcal{Q}_{pop}, \mathcal{Q}_{push}, \mathcal{Q}_{panic})$ is dendrisophilic and the structure $\boldsymbol{Ch}^{\boldsymbol{I}}_{\blacktriangleleft}(\mathfrak{A}, \mathcal{Q}_{pop}, \mathcal{Q}_{push})$ is isophilic.*

Proof. It is first necessary to encode the semi-nested words making up the domain of these structures as nested-words, since isophilic and dendrisophilic structures require a domain of nested-words. A naïve encoding is suggested by Figure 6.1, but this results in convolutions splitting too early to decide the $R_{\blacktriangleleft_{p,q}}$ relations. Instead we use an encoding Υ illustrated in Figure 6.11. Υ ensures the convolution does not split until the *sources* of the differing pointers. This is achieved by treating all positions not sourcing a pointer as the target of a pointer. This results in 'spurious pointers' that are discharged using additional ▲-labelled poistions. Positions labelled ▼ are targets used to simulate pointers sharing the same target (which are disallowed in nested-words).

We define a map Υ from semi-nested-words over an alphabet Σ to $\mathbf{NWord}(\Sigma)$. Suppose that $w^{\curvearrowleft E}$ is such a semi-nested-word. We can view $\mathbf{dom}(w) \subseteq \star^*$ and prefix closed. However it will be helpful to view $\mathbf{dom}(\Upsilon(w))$ as being a prefix closed subset of $(\star + ▲ + ▼)^*$. Thus it still defines a word (rather than a tree) but gives us a bit more flexibility with notation. For such a string u in the domain we write $\pi_\star(u)$ to denote the string that results from deleting the ▲ and ▼ elements from u. Note also that well-nesting ensures that we can uniquely define the pointers in a nested-word purely by specifying which positions belong to $\mathbf{img}(E)$ and which to $\mathbf{dom}(E)$.

So $\Upsilon(w^{\curvearrowleft E}) =: w'^{\curvearrowleft E'}$ is defined as follows. The definition is technical and the reader may prefer just to look at Figure 6.11 to see what the map really does. Take $\mathbf{dom}(w')$ to be the prefix closure of the set with longest element u' that satisfies the following where u is the longest element of $\mathbf{dom}(w)$:

- We have $\pi_\star(u') = u$.
- If the ith position of u is in $\mathbf{dom}(E)$ then the ith $\star$ in u' is in $\mathbf{dom}(E')$ and is followed immediately in u' by a single ▼ which is in $\mathbf{img}(E')$

- If the ith position of u is *not* in $\mathbf{dom}(E)$ (and may or may not be in $\mathbf{img}(E)$) then the ith $\star$ in u' is in $\mathbf{img}(E')$.
- If the ith position of u is in $\mathbf{dom}(E)$ and maps under E to the jth position, then the ith $\star$ in u', which we denote l, is immediately preceded by a sequence of $\blacktriangle$s in $\mathbf{dom}(E')$ such that $E'(l)$ is either the jth $\star$ in u' or else is a $\blacktriangledown$ following a kth element of u where that kth element maps to the jth element of u under E. Well-nesting of E' ensures that only one of these options is possible and that when the $\blacktriangledown$ option is taken there is precisely one suitable $\blacktriangledown$.
- We add $\blacktriangle$'s in $\mathbf{dom}(E)$ to the end to discharge any remaining elements in $\mathbf{img}(E)$.

Finally we label each $\star$ position in u' with the corresponding label from u and label each $\blacktriangledown$ position with $\blacktriangledown$ and each $\blacktriangle$ position with $\blacktriangle$. The unicity of the fourth item and the intrinsic unicity of the second and third conditions tell us that Υ must be injective. It thus respects equality.

Given a semi-nested word automaton $\mathcal{A}$ we can construct a nested-word automaton $\mathcal{A}'$ recognising:

$$\mathcal{L}(\mathcal{A}') = \{\ \Upsilon(w^{\frown E}) \in \mathbf{NWord}(\Sigma \cup \{\ \blacktriangle, \blacktriangledown\ \})\ :\ w^{\frown E} \in \mathcal{L}(\mathcal{A})\ \}$$

$\mathcal{A}'$ simulates $\mathcal{A}$ when reading $\star$ positions. It must non-deterministically guess whether a $\star$ position in ${w'}^{\frown E'}$ belonging to $\mathbf{img}(E')$ corresponds to a position of $w^{\frown E}$ in $\mathbf{img}(E)$ or in neither $\mathbf{img}(E)$ nor $\mathbf{dom}(E)$. It flags its guess in the control-state at this position l. It may then proceed to verify its guess at the position in ${w'}^{\frown E'}$ in $\mathbf{dom}(E')$ mapping to l under E'. If this position is labelled with $\blacktriangle$ then l is not in $\mathbf{dom}(E)$, otherwise it is. The only other feature of $\mathcal{A}'$ needed to facilitate the simulation is to repeat the state at $E(l)$ where l is a $\star$ node in $\mathbf{dom}(E')$ at the following $\blacktriangledown$ node. This enables the correct simulation of multiple pointers with the same target.

Since this is a nested-word automaton, it may be determinised, as would be required for acting on elements of convolutions to be read by restricted forms of nested-tree automata. Indeed we now turn our attention to recognising the various relations in the structures considered by the theorem. Let R be a relation of $\mathfrak{A}$. Since $\mathfrak{A}$ is *flat*-isophilic it must be that R is recognised by a standard (non-nested) tree automaton.

Recall that the relation $R_{\blacktriangleleft}$ relates flat-isophilic chains c and c' (semi-nested-words) made up respectively of elements $w_1, w_2, \ldots, w_{m-1}, w_m$ and $w_1, w_2, \ldots, w_{m-1}, w'_m$ where $w_m R w'_m$. Recall that $w_m = \ulcorner c \urcorner$ and $w'_m = \ulcorner c' \urcorner$. Let i_c be the last position in c to be a source of a pointer. Let $i_{c'}$ be the analogue for c'. Recall further that we can obtain the semi-nested-word representation of the chain consisting of the first $m-1$ elements in c by taking the prefix consisting of

everything strictly before i_c (so the target of i_c may no longer be the target of a pointer). Similarly we can obtain a representation of the first $m-1$ elements of c'. We thus have $cR_{\blacktriangleleft}c'$ iff the following two conditions are met:

- $\ulcorner c \urcorner R \ulcorner c' \urcorner$
- The prefix of c consisting of everything strictly before i_c is identical to the prefix of c' consisting of everything strictly before $i_{c'}$.

In order to understand the purpose of Υ, observe that i_c and $i_{c'}$ might source pointers with different targets. This would result in a 'semi-nested word convolution' splitting at the target of i_c or $i_{c'}$, preventing the necessary comparisons between the target of i_c and i_c itself from being carried out. This problem is eliminated under the Υ mapping as a pointer-caused difference can only occur at a source—non-sources are all treated as targets. We can see that the two conditions above are met under precisely the following conditions:

- $\ulcorner c \urcorner R \ulcorner c' \urcorner$
- The prefix of $\Upsilon(c)$ consisting of everything strictly before the position corresponding to i_c is identical to the prefix of $\Upsilon(c')$ consisting of everything strictly before the position corresponding to $i_{c'}$.

This time the second condition holds just in case everything strictly prior to either i_c in c or $i_{c'}$ in c' belongs to the trunk of the convolution $\bigotimes \langle \Upsilon(c), \Upsilon(c') \rangle$. Note also that under this assumption i_c and $i_{c'}$ would be the first positions in $\ulcorner c \urcorner$ and $\ulcorner c' \urcorner$ respectively that might differ from each other as these constitute the right-most sources of pointers. This means that everything prior to i_c or $i_{c'}$ in $\bigotimes \langle \ulcorner c \urcorner, \ulcorner c' \urcorner \rangle$ must occur in the trunk of the convolution $\bigotimes \langle \Upsilon(c), \Upsilon(c') \rangle$. Since R can be recognised by a non-nested tree automaton, it must be recognised by a non-nested automaton $\mathcal{A}_R$ that acts *deterministically* on the trunk of the tree. We can use the powerset construction to determinise the trunk. We must then have a nested-word automaton $\mathcal{A}_R^N$ whose state after reading a nested-word $\Upsilon(w^{\frown E})$ for semi-nested-word $w^{\frown E}$ is the same as that of $\mathcal{A}_R$ after reading $\ulcorner w^{\frown E} \urcorner$. This can be achieved by programing $\mathcal{A}_R^N$ to recall its state at the target of a pointer when reading its source (at Σ-labelled positions). We also encode in each $\mathcal{A}_R^N$ state following a source the state and label that was at that source's target.

We can then construct a non-nested tree automaton $\mathcal{A}_R^S$ that ignores the trunk of the convolution $\bigotimes \langle c, c' \rangle$. At the point of branching it does one of two things:

- Guesses that i_c and $i_{c'}$ both occur on the trunk of the convolution (in which case they would occur in the same position). It then behaves in the same manner as $\mathcal{A}_R$ after branching, beginning at the state of $\mathcal{A}_R^N$

at the end of the trunk. This will be sound assuming the correctness of the guess as the branching of the convolution $\bigotimes \langle c, c' \rangle$ will occur at the point corresponding to the branching of $\bigotimes \langle \ulcorner c \urcorner, \ulcorner c' \urcorner \rangle$. In order to verify the guess, it suffices just to check that the only ▲ symbols occur after branching are at the end of the branch, as this means there indeed are no further pointer source in the original words after the alleged i_c and $i_{c'}$. This indicates that the branch does not contain any position corresponding to a $\mathbf{dom}(E)$ position in the semi-nested word.

- Guess that i_c and $i_{c'}$ occur immediately after branching. In this case $\mathcal{A}_R^S$ will guess a pair (q_1, q_2) of $\mathcal{A}_R$ states and simulate a $\mathcal{A}_R$ branching with q_1 down the left branch and q_2 down the right. It can verify that i_c and $i_{c'}$ do indeed occur immediately after branching by ensuring that there are Σ symbols sourcing pointers immediately after the tip of the trunk. (Any preceding ▲ symbols should occur prior to branching as i_c and $i_{c'}$ should have the same target in the semi-nested words). It proceeds to check that (q_1, q_2) is a valid branching transition for $\mathcal{A}_R$ from the state and symbol that was at the target of i_c and $i_{c'}$ (which would be the state and symbol at the preceding position in the summary). The only remaining thing to verify is that there are no more positions corresponding to the source of a pointer in the semi-nested words. In order to do this we again just check that the only ▲ symbols occurring after branching are at the end of the branch.

- If both of the above guesses would be incorrect, then as previously explained the relation does not hold.

The path-nested automaton $\mathcal{A}_R^{N\mathcal{A}_R^S}$ thus recognises $R_{\blacktriangleleft}$.

Now we wish to show that the R_{push} and R_{pop} relations are recognisable by path-nested automata. Note that these relations are just the inverses of each other and so it is sufficient to show that just R_{pop} is recognisable by a path-nested automaton. Recall that $cR_{pop}c'$ just in case c is a flat isophilic chain of $w_1, w_2, \ldots, w_{m-1}, w_m$ and c' of $w_1, w_2, \ldots, w_{m-1}$ where $w_m R w_{m-1}$. So we can say that $cR_{pop}c'$ just in case:

- The convolution $\bigotimes \langle c, c' \rangle$ does not branch at all.

- If the convolution does not branch at all let i_c be the position in the convolution that corresponds to the final Σ-labelled element that sources a pointer. Then all Σ-labelled nodes strictly prior to i_c should be associated with both c and c' and everything after and including i_c should be associated exclusively with c.

- $\ulcorner c \urcorner R \ulcorner c' \urcorner$.

The first two conditions can easily be confirmed using a non-nested tree automaton. In order to test the final condition, we may assume that the first two conditions are also met (as these are tested independently). This means that $c' = c_{<i_c}$. Let l be the target of the pointer sourced at i_c. By the definition of flat-chains, this ensures that the label to the right of the node corresponding to l in $\ulcorner c' \urcorner$ will either not exist or will be different from the label of i_c. Thus l corresponds to the end of the trunk of the convolution $\bigotimes \langle \ulcorner c \urcorner, \ulcorner c' \urcorner \rangle$.

We can construct a non-deterministic nested-word automaton that recognises $\Upsilon(c)$ such that $\bigotimes \langle \ulcorner c \urcorner, \ulcorner c_{<i_c} \urcorner \rangle \in R$. Due to the above paragraph this is what is required to verify the third condition on the assumption that the first two (independently verifiable) conditions hold. Since nested-word automata can be determinised we can just augment the non-nested tree automaton verifying the first two conditions with a check that the tip of the c component of the convolution yields an accepting state on this deterministic automaton. This then gives us the required path-nested automaton recognising R_{pop}.

So we do indeed just need to construct a non-deterministic nested-word automaton that recognises $\Upsilon(c)$ such that $\bigotimes \langle \ulcorner c \urcorner, \ulcorner c_{<i_c} \urcorner \rangle \in R$. But this is straightforward. We just simulate $\mathcal{A}_R$ on the view of the semi-nested word on the summary up to a point that we non-deterministically guess to be l the target of i_c. At l we leave a flag in our state to mark the guess and proceed simulating a right-branching of $\mathcal{A}_R$ on the remaining view. As soon as we reach reach the position sourcing the pointer targetting our flagged position, we treat it as i_c and start to simulate the corresponding left-branching. We verify the guess was correct by making sure that after the associated guessed i_c the only ▲ nodes are at the very end of the word.

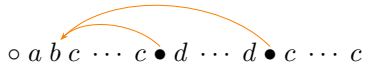
Now let us turn our attention to the claims that $\boldsymbol{Ch}^{\boldsymbol{S}}_{\blacktriangleleft}(\mathfrak{A}, \mathcal{Q}_{pop}, \mathcal{Q}_{push}, \mathcal{Q}_{panic}, \mathcal{Q}_{calm})$ is symmetric-dendrisophilic and that $\boldsymbol{Ch}^{\boldsymbol{D}}_{\blacktriangleleft}(\mathfrak{A}, \mathcal{Q}_{pop}, \mathcal{Q}_{push}, \mathcal{Q}_{panic})$ is dendrisophilic. Again it suffices to establish the dendrisophilic claim only since the *calm* relation is the inverse of *panic* and so the addition of *calm* must fall under symmetric-dendrisophlicity. So we need to show that there exists a spine nested automaton $\mathcal{B}^{\mathcal{C}}$ recognising the relation $(P_1, P_2)_{panic}$ for every pair of unary predicates P_1 and P_2.

Given flat-isophilic chain c it is easy to construct a deterministic nested-word automaton acting on $\Upsilon(c)$ to check whether the final element in the chain c satisfies some unary predicate P. After all, we just need to check whether $\ulcorner c \urcorner \in P$ and P itself will be recognised by some deterministic non-nested word automaton $\mathcal{A}_P$. So we just need to have a nested-word automaton that reads $\Upsilon(c)$ simulating $\mathcal{A}_P$, picking up from the state at the target of a pointer when reading the source of a pointer at a Σ-labelled position. In particular, given two flat-isophilic chains c and c' and *two* predicates P_1 and P_2 we can have a single nested-word automaton simulating both $\mathcal{A}_{P_1}$ and $\mathcal{A}_{P_2}$ simultaneously. This

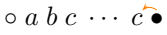
single automaton can act on each branch of $\bigotimes \langle \Upsilon(c), \Upsilon(c') \rangle$ independently.

Thus we just need to check that we can construct a spine nested automaton that recognises convolutions $\bigotimes \langle \Upsilon(c), \Upsilon(c') \rangle$ where c encodes $w_1, \ldots, w_{m-1}, w_m$ and c' encodes the longest $w_1, \ldots, w_r$ for some $r < m$ such that w_m is a prefix of w_i for every $r < i \leq m$. By Lemma 6.42 we just need to ensure that $\Upsilon(c')$ is the same as an initial prefix of $\Upsilon(c)$ and that the the first position in $\Upsilon(c)$ corresponding to a target in c after the end of $\Upsilon(c')$ is as dictated by Lemma 6.42. This can be checked by a nested-tree automaton that marks this target. This will require a non-deterministic guess in the case of $\Upsilon(c)$—we have seen how the fact a Σ symbol is a target in the underlying semi-nested words can only be verified at the source. However, since this occurs in the left-branch from the trunk of the convolution, it can be carried out by a spine nested automaton. (Note that spine nested would be require irrespective of the Υ issue since flagging the first target after branching would still require a post-branching behaviour change). □

In order to understand why the preconditions for *panic* and *calm* relations have to be based on unary predicates rather than binary relations, as with *push* and *pop* relations, consider the following chain:



There is no way for an automaton reading left-to-right to be able to determine whether the two instance of $c \ \ldots \ c$ have the same contents. If they were next to each other without the intervening $d \ \cdots \ d$ such a comparison would be possible. If they were equal we would have:



This is not possible to encode in the presence of the intervening $d \ \cdots \ d$ as it would violate well-nesting:

So binary comparisons can only be done between adjacent elements in the chain.

Graphs from Chains of Nested-Words

We build the 'next level' from $\mathcal{Q}$-decorated graphs $\mathfrak{A}_{\mathcal{Q}}$ that are either isophilic or dendrisophilic. The structure $\boldsymbol{Ch}^{\boldsymbol{T}}_{\prec}(\mathfrak{A}, \mathcal{Q}_{pop}, \mathcal{Q}_{push})$ is a structure whose

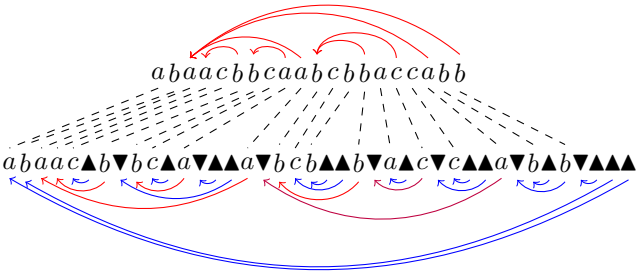


Figure 6.11: The Υ-mapping of a semi-nested word.

domain consists of either isophilic or dendrisophilic chains (depending on $\mathfrak{A}$), which are nested-*trees* and is defined in a way completely analogous to $\boldsymbol{Ch}^{\boldsymbol{I}}_{\blacktriangleleft}$—the definition can almost be read verbatim except that we name the decorating function $\mathcal{Q}_{\prec}$ instead of $\mathcal{Q}_{\blacktriangleleft}$, write $R_{\prec}$ instead of $R_{\blacktriangleleft}$ and 'representation of a chain' refers to an isophilic chain (nested-tree) rather than a flat-isophilic chain (nested-word). We sometimes write $\boldsymbol{Ch}^{\boldsymbol{N}}_{\prec}$ instead of $\boldsymbol{Ch}^{\boldsymbol{T}}_{\prec}$ when $\mathfrak{A}$ is dendrisophilic (instead of isophilic).

Theorem 6.44. *Let $\mathfrak{A}_Q$ be a Q-decorated isophilic (*resp. *dendrisophilic) structure. For every $\mathcal{Q}_{pop}, \mathcal{Q}_{push} : \mathcal{R}_{\mathfrak{A}} \longrightarrow Q \times Q$ $\boldsymbol{Ch}^{\boldsymbol{T}}_{\prec}(\mathfrak{A}, \mathcal{Q}_{pop}, \mathcal{Q}_{push})$ is tree-isophilic (*resp. *tree-nondisophilic).*

Proof. To recognise a relation $R_{\prec}$ we just need to check that the two trees in the binary convolution differ only in their right-most branches and then run the automaton recognising R on this pair of right-most branches. For R_{push} and R_{pop} we just need to check that one tree is precisely the subtree consisting of all nodes not belonging exclusively to the right-most branch of the other. We then just need to check that the right-most branch and branch immediately to the left of that branch are related by R, which can be done by treating this pair of branches like a convolution and applying the automaton for R.

Lack of interesting consequence for our purposes means that we do not consider symmetric-dendrisophilic chains in the above, although this would technically be possible. □

Remark 6.45. We could have added *panic* and *calm* relations defined in an analogous manner. However, we choose not to as these will not be as useful for our purposes in assisting with precise characterisations of higher-order automata. For one thing we would need some artificial restrictions forcing the differing branches in a convolution to be right-most and contiguous in the branch ordering. We also lose a nice notion of dendrisophilicity with nested-trees that we have in the nested-word case—we would thus be forced to talk

about the more general nondisophilic structures, which do not admit such a nice characterisation.

Bounces

We introduce a more abstract notion of bouncing (compared to that introduced in the previous chapter) that applies to (dendr)isophilic structures in general. It is useful to 'globalise' the pushes and pops of these chain structures in the form of a *bounce*. In a structure $\boldsymbol{Ch}^{\boldsymbol{T}}_{\prec}(\mathfrak{A}, \mathcal{Q}_{pop}, \mathcal{Q}_{push})$ we say that a node u *bounces* to a node u' if there exists a $\mathcal{Q}_{\prec}$-compatible path of the form

$$uR_{1\,pop}u_1R_{2\,pop}\cdots u_{i-1}R_{i\,pop}u_iR_{i+1\,\prec}u_{i+1} \\ R_{i+2\,push}u_{i+2}R_{i+3\,push}\cdots R_{m\,push}u'$$

i.e. pops followed by an $R_{\prec}$ followed by pushes. Let us write $u\boldsymbol{b}u'$ to denote this relation; note that this can be defined in the same way for $\boldsymbol{Ch}^{\boldsymbol{I}}_{\blacktriangleleft}$ and $\boldsymbol{Ch}^{\boldsymbol{D}}_{\blacktriangleleft}$. For the latter the following theorem is a special case of reachability, but for tree-isophilic structures and tree-nondisophilic structures we have no general definability-of-reachability theorem and so a few extra comments are needed to establish it for $\boldsymbol{Ch}^{\boldsymbol{T}}_{\prec}$:

Theorem 6.46. *1. Every structure of the form $\boldsymbol{Ch}^{\boldsymbol{T}}_{\prec}(\mathfrak{A}, \mathcal{Q}_{pop}, \mathcal{Q}_{push})$ remains tree-isophilic (*resp. *tree-nondisophilic) where $\mathfrak{A}$ is isophilic (*resp. *dendrisophilic) when the bounce relation $\boldsymbol{b}$ is added.*

2. Every structure of the form $\boldsymbol{Ch}^{\boldsymbol{I}}_{\blacktriangleleft}(\mathfrak{A}, \mathcal{Q}_{pop}, \mathcal{Q}_{push})$ remains isophilic when $\boldsymbol{b}$ is added.

3. Every structure of the form $\boldsymbol{Ch}^{\boldsymbol{D}}_{\blacktriangleleft}(\mathfrak{A}, \mathcal{Q}_{pop}, \mathcal{Q}_{push}, \mathcal{Q}_{panic})$ remains dendrisophilic when $\boldsymbol{b}$ is added.

4. Every structure of the form $\boldsymbol{Ch}^{\boldsymbol{S}}_{\blacktriangleleft}(\mathfrak{A}, \mathcal{Q}_{pop}, \mathcal{Q}_{push}, \mathcal{Q}_{panic})$ remains symmetric-dendrisophilic when $\boldsymbol{b}$ is added.

Proof. Firstly we should mention that the individual R_{pop} and R_{push} are definable as they just require the comparison of neighbouring branches which may be treated as convolutions. Iterated popping and pushing may also be defined using the same construction as in the proof of Lemma 6.38 as these amount to recognising that a suffix of the $\prec$-ordering forms a chain.

We can thus recognise a bounce from $T^{\frown E}$ to $T'^{\frown E'}$ by detecting iterated pop from $T^{\frown E}$ to some tree $U^{\frown F}$ and iterated push from some $U'^{\frown F'}$ to $T'^{\frown E'}$ such that all but the $\prec$-last branch of each of $U^{\frown F}$ and $U'^{\frown F'}$ are shared. Since all of the branches before them in the $\prec$-ordering are shared, these last two branches can then be treated as a binary convolution of nested-words to test an intermediate relation for the bounce. □

6.5 Relationship with Tree Automatic Structures

We wish to demonstrate the way in which the hierarchy of chains can be exploited. We will do this by constructing ordinal numbers, which also enables us to separate the various levels of the hierarchy and indeed see that the tree-isophilic structures is a strictly bigger class than the tree automatic structures.

It is known that the word and tree-automatic ordinals are precisely those below ω^ω and ω^{ω^ω} respectively [36]. We illustrate the utility of the flat, isophilic, tree-isophilic hierarchy by re-exhibiting the definability of ω^ω, ω^{ω^ω} and additionally $\omega^{\omega^{\omega^\omega}}$ as a natural progression exploiting isophilic chains.

All ordinals α have a unique *Cantor Normal Form* $\alpha = \sum_{i=1}^{k} \omega^{\beta_i}$ where $\beta_1 \geq \beta_2 \geq \cdots \geq \beta_k \geq 0$ are ordinals. When $\alpha < \epsilon_0$ we also have $\beta_1 < \alpha$. If $\alpha' = \sum_{i=1}^{k'} \omega^{\beta'_i}$, then $\alpha \leq \alpha'$ iff $k \leq k'$ and for every $1 \leq i \leq k$ we have $\beta_i \leq \beta'_i$.

We can thus encode every ordinal below ω^ω (for which the β_i will be natural numbers) as a finite-word over the alphabet $\{a, b\}$. This will take the form $b^{\beta_1}\ a\ b^{\beta_2}\ a\ \cdots\ a\ b^{\beta_k}$. This is how it is shown that the set of ordinals below ω^ω is word-automatic; in fact they are flat-isophilic. As soon as a difference in two encodings is spotted (which is when a convolution would split) the ordering can be inferred. So let $\boldsymbol{On(\omega^\omega)}$ be this flat-isophilic structure representing the ordinals below ω^ω under the *strict* ordering $<$.

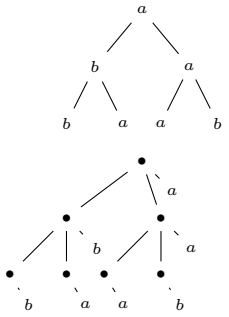


Figure 6.12: 'Leafication' of a tree

Progressing up the hierarchy, let $\boldsymbol{On(\omega^\omega)_U}$ be the structure $\boldsymbol{On(\omega^\omega)}$ augmented with two instances of the universal relation (that relates everything) U_{pop} and U_{push}. Assume that it is decorated over a set $Q = \{q_{pop}, q_{push}\}$ where $\mathcal{Q}(U_{pop}) := (q_{pop}, q_{pop})$, $\mathcal{Q}(<) := (q_{pop}, q_{push})$ and $\mathcal{Q}(U_{push}) := (q_{push}, q_{push})$. An ordinal below ω^{ω^ω} has a Cantor Normal Form for which the $\beta_i < \omega^\omega$—that is it can be represented as a chain of ordinals less than ω^ω and $\alpha < \alpha'$ if there is a common prefix of the chains for α and α' that is followed by β in α and β' in α' such that $\beta < \beta'$. This is exactly the *bounce* relation for $\boldsymbol{Ch^I_\blacktriangleleft}(\boldsymbol{On(\omega^\omega)}_{U\,\mathcal{Q}}, \mathcal{Q}, \mathcal{Q})$ and so taking $<$ to be $\boldsymbol{b}$ we obtain $\boldsymbol{On(\omega^{\omega^\omega})}$. In a similar vein we can finish the hierarchy with $\boldsymbol{On(\omega^{\omega^{\omega^\omega}})} := \boldsymbol{Ch^T_\prec}(\boldsymbol{On(\omega^{\omega^\omega})}_{U\,\mathcal{Q}}, \mathcal{Q}, \mathcal{Q})$. Thus all ordinals below $\omega^{\omega^{\omega^\omega}}$ are tree-isophilic.

Delhommé's result [36] implies that there exist tree-isophilic structures that are not tree-automatic.

It is also straightforward to see that every tree-automatic structure is tree-isophilic. To prevent convolutions from splitting due to a difference in labels we shift the labels to an additional child of each node. An automaton reading the

convolution guesses the label at each node and verifies the guess by checking the child containing the label. This transformation is illustrated in Figure 6.12.

Word (***WAut***) and tree (***TAut***) automatic structures are *not* closed under reachability and considering the relationship between nested-words and trees:

Theorem 6.47. $\boldsymbol{Iso_1} \subsetneq \boldsymbol{WAut} \subsetneq \boldsymbol{TAut} \subsetneq \boldsymbol{Iso_3}$ *and* $\boldsymbol{Iso_1} \subsetneq \boldsymbol{Iso_2} \subsetneq \boldsymbol{TAut}$.

It is also worth noting that every word-automatic structure is nondisophilic:

Theorem 6.48. $\boldsymbol{WAut} \subsetneq \boldsymbol{nIso_2}$

Proof. Let $\mathfrak{A}$ be a word-automatic structure. We encode each element w of the domain A of $\mathfrak{A}$ as a nested-word of the form $\bullet^{|w|}w^{\leftarrow}$, where $w^{\leftarrow}$ is the word w written backwards. Each element of $w^{\leftarrow}$ should source a pointer, pointing to a canonical target in $\bullet^{|w|}$ determined by well-nesting. Suppose that A is over the alphabet Σ. Word-automatic n-ary convolutions can then be simulated along the trunk of a nested-word convolution since the nested-tree automaton can non-deterministically guess an n-tuple in Σ^n representing the Σ-label of each of the components in the nested-word convolution. These guesses can then be verified when reading the $w^{\leftarrow}$ component of the encoding of each word. This may occur after the nondisophilic convolution has split, but this does not matter since the simulation of the synchronisation of the word-automatic convolution would have been based on the guess whilst reading the trunk.

Strictness of the inclusion comes from the fact that $\boldsymbol{nIso_2}$ contains $\boldsymbol{Iso_2}$, which contains the ordinal $\omega^{\omega^{\omega}}$, which is not word-automatic [36]. □

6.6 Closure Under Graph Transformations

We have already seen that the classes $\boldsymbol{sIso_2}$, $\boldsymbol{dIso_2}$, $\boldsymbol{Iso_2}$ of symmetric-dendrisophilic, dendrisophilic and isophlic structures are closed under the addition of reachability predicates. Observe that Theorem 6.39 can be generalised to reachability witnessed by strings of relations restricted by some regular expression. Kartzow considered this in his work on 2-CPDS graphs [48] and indeed this forms the basis of rational mappings, devices that can be used as part of the construction of the Caucal Hierarchy [29]. New graphs can be defined via relations specified by regular sets of paths in the original.

Definition 6.49. Let $\mathfrak{A} = \langle \boldsymbol{A}, \boldsymbol{R_1}, \ldots, \boldsymbol{R_m} \rangle$ be a directed graph. If E is a finite set of symbols, then a *rational map* on $\mathfrak{A}$ is a function:

$$\rho \ : \ E \ \longrightarrow \ 2^{[1..m]^*}$$

where the image of ρ consists of regular languages. An *inverse rational map* on $\mathfrak{A}$ is a function:

$$\rho \ : \ E \ \longrightarrow \ 2^{([1..m]+\overline{1}..[\overline{m}])^*}$$

where the image of ρ consists of regular languages.

So a rational map associates a fresh edge symbol in E with a rational set of sequences of relations in the original graph. An *inverse* rational map will allow for edges (relations) in the original graph to be traversed *backwards*. A directed edge $\boldsymbol{R_{\overline{i}}}$ is the directed edge $\boldsymbol{R_i}$ backwards.

Definition 6.50. Let $\mathfrak{A} = \langle \boldsymbol{A}, \boldsymbol{R_1}, \ldots, \boldsymbol{R_m} \rangle$ be a graph. Let ρ be a(n inverse) rational map on $\mathfrak{A}$ with domain E. There exists a set of relations between the nodes of the graph $\boldsymbol{r}_e$ for each $e \in E$ such that $u \rho_e u'$ iff there is a path $u\boldsymbol{R_{i_1}}u_1\boldsymbol{R_{i_2}}u_2\boldsymbol{R_{i_3}}\cdots\boldsymbol{R_{i_{k-1}}}u_{i_{k-1}}\boldsymbol{R_{i_k}}u'$ where $i_1 i_2 i_3 \cdots i_{k-1} i_k \in \rho(e)$ and we define $R_{\overline{i}} := \{\ (y,x)\ : xR_i y\ \}$—*i.e.* $R_{\overline{i}}$ is the inverse of R_i.

The graph $\rho(\mathfrak{A})$ defined in $\mathfrak{A}$ by ρ is set to be the graph with domain $\boldsymbol{A}$ and directed relations $\boldsymbol{R_e} := \rho_e$ for each $e \in E$.

We obtain the following closures under rational and inverse rational maps:

Theorem 6.51. *The isophilic and symmetric-dendrisophlic graphs are closed under inverse rational maps. The dendrisophilic graphs are closed under just rational maps.*

Proof. We need to show that given a graph $\mathfrak{A}$ that is isophilic, dendrisophilic or symmetric-dendrisophilic then for any rational map $\rho : E \longrightarrow 2^{[1..m]}$ the relation ρ_e is respectively isophilic/dendrisophilic/symmetric-dendrisophilic for each $e \in E$. We can obtain the result for inverse rational maps for isophilic and symmetric-dendrisophilic structures by first adding the inverse of each relation in $\mathfrak{A}$. Since path-nested and trunk nested automata recognising binary relations can always swap the way they act on each branch, this is always possible. Note that this cannot be done for dendrisophilic structures as spine nested automata are restricted differently on each branch (they must behave differently on the spine to the non-spine) and so it is not possible, in general, to have them recognise branches in the reversed order.

So suppose $e \in E$ maps under ρ to a regular word-language $\mathcal{L}$ recognised by a finite automaton $\mathcal{A}_\mathcal{L}$ with transition function $\delta : [1..m] \times Q \longrightarrow Q$ where Q is its state space. Let us begin by taking an automaton $\mathcal{A}$ recognising $\boldsymbol{Ch}(\mathfrak{A})$. Recall that an accepting run of $\mathcal{A}$ will have a state at the leaf of the $(i+1)$th branch of the chain recording the index (in $[1..m]$) of the relation relating the ith branch to the $(i+1)$th branch. If the chain has k elements we need to check that the string of relation indices of the $(i+1)$th branch for $i \in [1..(k-1)]$ belongs to $\mathcal{L}$.

We can simulate a run of $\mathcal{A}_\mathcal{L}$ along the leaves of a tree using a finite tree automaton $\mathcal{A}^T_\mathcal{L}$. We take the state space of $\mathcal{A}^T_\mathcal{L}$ to be $((Q \times [1..m]^\perp) \times Q) \cup [1..m]^\perp$ with initial states:

$\{\ ((q_0, \perp), q)\ :\ q_0$ is initial state of $\mathcal{A}_\mathcal{L}$ and q is final state of $\mathcal{A}_\mathcal{L}\ \} \cup \{\ \perp\ :\ $ if q_0 is accepting$\}$

A state of $\mathcal{A}_{\mathcal{L}}^{T}$ indicates a guess about the run of $\mathcal{A}_{\mathcal{L}}$ along the leaves of the tree. A state $((q,i),q') \in (Q \times [1..m]^{\perp}) \times Q)$ at the root of a subtree T in the chain asserts that the subtree has at least two leaves and that the leaf of the left-most branch of the sub-tree will have state q in a run of $\mathcal{A}_{\mathcal{L}}$ along the leaves and that the left-most branch is related to the branch to the left of it in the chain by the ith relation of $\mathfrak{A}$. If the left-most branch of T is the left-most branch of the whole chain, there is no branch to the left of it in the chain and so i is set to $\perp$. It further asserts that the right-most branch of the sub-tree T will be tipped with state q in the run.

A state $i \in [1..m]^{\perp}$ simply asserts that the sub-tree T does not branch and that its leaf is related to the previous element of the chain by the ith relation of $\mathfrak{A}$. If T lies entirely within the left-most branch of the chain, then i is set to $\perp$.

Note that the initial state of the automaton will be placed at the root of the chain and so is at the case when T is the whole chain. Note how given the meaning of the states of $\mathcal{A}_{\mathcal{L}}^{T}$ defined above, the initial states all assert that the left-most branch of the chain is associated with an initial state of $\mathcal{A}_L$ and the right-most with an accepting state. This thus asserts the existence of an accepting run of $\mathcal{A}_L$ on the leaves. The automaton $\mathcal{A}_{\mathcal{L}}^{T}$ proceeds to spawn up to two children such that the correctness of the assertions made at the two children implies the correctness of the assertion at the parent. It will fail (abort the run) at a node only if the parent assertion is false.

- If the state is $i \in [1..m]^{\perp}$, then we fail if there are two children and if there is one child set the child state to be i again. This is correct because if the child is rooted at a subtree that does not branch, then under these circumstances so must the parent.

- If the state is $((q,i),q') \in (Q \times [1..m]^{\perp}) \times Q)$, then if there is only one child we simply propagate the same state to it. If there are two children, then we non-deterministically choose one of the following (all of which apply the truth of the assertion made by the state at the parent node):

  - Down the left child we send i and down the right child we send $j \in [1..m]$ such that $q' \in \delta(j,q)$
  - Down the left child we send i and down the right child we send $((p,j),q')$ such that $p \in \delta(j,q)$
  - Down the left child we send $((q,i),p)$ and down the right child we send $j \in [1..m]$ such that $q' \in \delta(j,p)$
  - Down the left child we send $((q,i),p)$ and down the right child we send $((r,j),q')$ such that $r \in \delta(j,p)$

We now run $\mathcal{A}$ in tandem with $\mathcal{A}_{\mathcal{L}}$ and deem a run to be accepting just in case $\mathcal{A}$ accepts and additionally $\mathcal{A}_{\mathcal{L}}$ ends in states of the form $i \in [1..m]^{\perp}$ such that i is either $\perp$ (which by design can only ever occur on the left-most leaf) or matches the index provided by $\mathcal{A}$ of the relation relating the branch to its predecessor in th chain. Clearly this condition would imply the truth of the assertion at the leaf made by the state of $\mathcal{A}$ and fail only if the assertion made by $\mathcal{A}$ is false. It must thus be that the assertion made at the root of the tree is true iff there is an accepting run-tree and given the assertion made at the root we thus have an automaton recognising all chains denoting paths that are specified by $\mathcal{L}$.

We can then apply projection on the exoskeleton (Lemma 6.25) and eliminate the ϵ nodes in the same manner as in the proof of Theorem 6.39. □

We will later see that the dendrisophilic graphs are 'inherently asymmetric' in the sense that they are definitely not closed under inverse rational maps.

A related closure property is the following:

Corollary 6.52. *The ϵ-closure of a symmetric-dendrisophilic, dendrisophilic or isophilic graphs is respectively symmetric-dendrisophilic, dendrisophilic or isophilic.*

Proof. The edges of the ϵ-closure are given by the rational map $\epsilon^* a \mapsto a$ for every edge-label a. If the graph is rooted and we additional want to restrict the reachable configurations under ϵ-closure (as with CPDA graphs) we can recognise the domain as those elements connected to the root by an edge induced by the regular language $(\epsilon + \Sigma)^*.\Sigma$ where Σ is the set of edge labels. □

The next form of closure is the first use we make of nondisophilic structures (not necessarily graphs). We show that they are closed under 'projection' on the labels of the nodes belonging to the nested-words making up their domains. This is fairly predictable and obvious. What is more significant (albeit not difficult to prove) is that the nondisophilic structures are *precisely* those that arise when one projects on the alphabets of the isophilic structures, the smallest of our hierarchy of classes. So projection allows us to go from the smallest of our classes to the largest. When we come to alternative characterisations of these structures, we will be able to say a little more about how to interpret this result—in particular it will become our only characterisation of nondisophilic structures which do not appear to admit characterisations enjoying the same kind of naturality as the other classes.

Definition 6.53. A *projection* on an alphabet Σ is nothing more than a map $\pi : \Sigma \longrightarrow \Sigma'$. (We use the word 'projection' as often such maps are used to eliminate certain pieces of information from node labels). Given a tree $T^{\curvearrowleft E} \in \mathbf{NTree}(\Sigma)$ we define $\pi(T^{\curvearrowleft E})$ to be the tree in $\mathbf{NTree}(\Sigma')$ formed

by replacing each label l with $\pi(l)$ at each node. Given a relational structure $\mathfrak{A} = \langle \boldsymbol{A}, \boldsymbol{R_1}, \ldots, \boldsymbol{R_m} \rangle$ whose domain consists of Σ-labelled nested-trees we define $\pi(\mathfrak{A})$ to be the structure:

$$\pi(\mathfrak{A}) := \langle \pi(\boldsymbol{A}), \pi(\boldsymbol{R_1}), \ldots, \pi(\boldsymbol{R_m}) \rangle$$

where $\pi(\boldsymbol{A}) := \{\ \pi(T^{\frown E})\ :\ T^{\frown E} \in \boldsymbol{A}\ \}$ and

$$\pi(\boldsymbol{R_i}) := \{\ (\pi(T_1{}^{\frown E_1}), \ldots, \pi(T_n{}^{\frown E_n}))\ :\ (T_1{}^{\frown E_1}, \ldots, T_n{}^{\frown E_n}) \in \boldsymbol{R_i}\ \}$$

for each $1 \leq i \leq m$.

Theorem 6.54. *The nondisophilic structures are closed under projection*—i.e. *if* $\mathfrak{A} \in \boldsymbol{nIso_2}$*, then* $\pi(\mathfrak{A}) \in \boldsymbol{nIso_2}$ *for any projection* π*.*

Proof. We first show that nondisophilic structures are closed under projection. Let $\mathfrak{A}$ be a nondisophilic structure and let $\pi : \Sigma \longrightarrow \Sigma'$ be a projection. First observe that it is straightforward to adapt a nested-tree automaton recognising a tree $T^{\frown E}$ to one recognising $\pi(T^{\frown E})$—just replace each occurrence of $a \in \Sigma$ in the transition function with $\pi(a)$, taking the union of elements of the co-domain of the transition functions that end up being the image of the same element of the domain. This may introduce further non-determinism but since nondisophilic structures place no restrictions on where their automata may be non-deterministic this is not a problem. This takes care of recognising the domain of $\pi(\mathfrak{A})$. A small modification is needed for the relations, however, as the convolutions may change their structure—the point at which elements in the relation begin to differ may be closer to the root after projection.

Suppose that $\mathcal{A}$ is an automaton recognising an n-ary relation $\boldsymbol{R}$ in $\mathfrak{A}$. Suppose further that $\mathcal{A}$ has state space Q. We can recognise the convolutions corresponding to $\pi(\boldsymbol{R})$ with an automaton having control-states $(Q^{\perp})^{2^{[1..n]}}$. A state $f \in (Q^{\perp})^{2^{[1..n]}}$ satisfies $f(S) \neq \perp$ iff we are simulating a branch of a convolution in R for which S specifies a maximal group of components that have not yet separated. The elements S such that $f(S) \neq \perp$ will thus be disjoint. We begin reading a proposed element of $\pi(\boldsymbol{R})$ in state $[1..n] \mapsto q_0$ and $[1..n] \neq S \mapsto \perp$, where q_0 is the initial state of $\mathcal{A}$.

We then simulate a transition of $\mathcal{A}$ acting on a convolution over the alphabet Σ. Let f be the current state and let a be the current symbol. Suppose that $S_1, \ldots, S_k$ are all of the disjoint sets such that $f(S_i) \neq \perp$ for $i \in [1..n]$. If the current node label is (S, a) and there exists T with $f(T) \neq \perp$ and $T \not\subseteq S$ then we fail, because we are attempting to simulate branches that could not have been assimilated into the actual branch of the convolution that we are reading. Otherwise suppose that the current node has m children. We pick $b_1, b_2, \ldots, b_k$ (a list which may contain repetitions) such that $\pi(b_i) = a$ for every $1 \leq i \leq k$. Suppose that $\mathcal{A}$ would allow (S_i, b_i) to spawn l_i children with states $q_1, \ldots, q_{l_i}$

from state $f(S_i)$ (looking at the $\mathcal{A}$ state $g(V)$ where V is the set with $S_i \subseteq V$ such that $g(V) \neq \bot$ where g is the state at the target when reading the source of a pointer). We then partition S_i into sets $U^i_1, U^i_2, \ldots, U^i_{l_i}$. We then partition the set:

$$\{\ (U^i_{j_i}, q_{j_i})\ :\ 1 \leq i \leq k \text{ and } 1 \leq j_i \leq l_i\ \}$$

into m sets $W_1, \ldots, W_m$. We put a state f_j at the jth child ($1 \leq j \leq m$) where $f_j(U) := q$ iff $(U, q) \in W_j$, otherwise $f_j(U) := \bot$.

In this manner we can simulate a run on the original convolution. The reader might find this somewhat technical but what is going on is very simple. We are just compensating for the fact that we need to be able to simulate a split between elements of the *projected* convolution when they might have split in the original convolution of $\mathfrak{A}$ but before the split in $\pi(\mathfrak{A})$'s convolution. The split in the latter might be delayed because projection could cause more corresponding positions in the elements to 'become equal'. □

Theorem 6.55. *The nondisophilic structures are precisely the isophilic structures under projection. In the light of Theorem 6.54 this amounts to: for every* $\mathfrak{A} \in \boldsymbol{nIso_2}$ *there exists* $\mathfrak{B} \in \boldsymbol{Iso_2}$ *and projection* π *such that* $\mathfrak{A} \cong \pi(\mathfrak{B})$.

Proof. Let $\mathfrak{A} = \langle \boldsymbol{A}, \boldsymbol{R_1}, \boldsymbol{R_2}, \ldots, \boldsymbol{R_m} \rangle \in \boldsymbol{nIso_2}$ be a nondisophilic structure. The idea is that we will create an isophilic structure $\mathfrak{B}$ that consists of 'run-trees' of the automata used to recognise the domain and relations in $\mathfrak{A}$. Since run-trees are decorated with states, this has the effect of resolving non-determinism as any non-deterministic choice would be specified by the decorating states. This in turn allows for recognition by the constrained automata of an isophilic structure. Projection can then be used to eliminate the states from the run trees just leaving the original labels—this yields $\mathfrak{A}$.

In actual fact since nested-word automata can be determinised, we only need concern ourselves with the automata for the relations. Suppose that Σ' is the alphabet for the words in $\boldsymbol{A}$. Let $\mathcal{A}$ be a *deterministic* nested-word automaton recognising $\mathcal{A}$. Let $\mathcal{R}_i$ be a (possibly non-deterministic) nested-tree automaton with control-states Q_i recognising the convolutions belonging to $\boldsymbol{R_i}$ for each $1 \leq i \leq m$.

We define the alphabet $\Sigma := \bigcup_{i=1}^{m} (\Sigma' \times Q_i \times (Q_i \cup \{\bullet, \boxtimes\})$. The projection $\pi : \Sigma \longrightarrow \Sigma'$ is defined by $\pi(a, q, q') := a$. The structure $\mathfrak{B} = \langle \boldsymbol{B}, \boldsymbol{S_1}, \boldsymbol{S_2}, \ldots, \boldsymbol{S_m} \rangle$ is defined as follows:

$$\boldsymbol{B} := \{\ w^{\curvearrowleft E} \in \mathbf{NWord}(\Sigma)\ :\ \pi_1(w^{\curvearrowleft E}) \in \boldsymbol{A} \text{ and for } u \in \mathbf{dom}(w)$$

$$\pi_3(w(u)) = \begin{cases} \bullet & \text{if } u \in \mathbf{img}(E) \\ \boxtimes & \text{if } u \notin \mathbf{img}(E) \cup \mathbf{dom}(E)\ \} \\ \pi_2(E(u)) & \text{if } u \in \mathbf{dom}(E) \end{cases}$$

$$\begin{aligned}\boldsymbol{S_i} := \{\ \bigotimes \langle {w_1}^{\frown E_1}, {w_2}^{\frown E_2}, \ldots, {w_n}^{\frown E_n} \rangle \in \mathit{Convo}^n(\Sigma)\ :\\ (\pi_1({w_1}^{\frown E_1}), \pi_1({w_2}^{\frown E_2}), \ldots, \pi_1({w_n}^{\frown E_n})) \in \boldsymbol{R_i}\\ \text{and } \pi_2({w_j}^{\frown E_j}) \in Q_i \text{ for each } 1 \leq j \leq n\\ \text{and } \mathbf{dom}(\bigotimes \langle {w_1}^{\frown E_1}, {w_2}^{\frown E_2}, \ldots, {w_n}^{\frown E_n} \rangle)\\ = \mathbf{dom}(\bigotimes \langle \pi_1({w_1}^{\frown E_1}), \pi_1({w_2}^{\frown E_2}), \ldots, \pi_1({w_n}^{\frown E_n}) \rangle)\\ \pi_2(\bigotimes \langle {w_1}^{\frown E_1}, {w_2}^{\frown E_2}, \ldots, {w_n}^{\frown E_n} \rangle) \text{ is an accepting run-tree of } \mathcal{R}_i \text{ on } \pi_1\ \}\end{aligned}$$

for each $1 \leq i \leq m$. The assertion about the domains of convolutions can be viewed as saying that the elements in the convolution of the ${w_i}^{\frown E_i}$ split in the same places as they do after being projected. In other words, a split must occur when the Σ' component of a label in Σ differs rather than as a result of different states in Q being assigned to separate instances of the same element of Σ.

We immediately get that $\pi(\boldsymbol{B}) = \boldsymbol{A}$ and $\pi(\boldsymbol{S_i}) = \boldsymbol{R_i}$ for each $1 \leq i \leq m$. That is to say that $\pi(\mathfrak{B}) = \mathfrak{A}$. It just remains to show that $\mathfrak{B}$ is isophilic; to do that we need to check that there exist path-nested automata recognising $\boldsymbol{B}$ and $\boldsymbol{S_i}$ for each $1 \leq i \leq m$.

Observe that $\boldsymbol{B}$ can easily be recognised by a (deterministic) nested-word automaton. We just take the deterministic nested-word automaton for $\boldsymbol{A}$ and add the trivial checks to enforce the conditions for the π_3 elements of the node-labels. We now claim that the relations can be recognised by standard (non-nested) finite-tree automata, a special case of path-nested automata. Such an automaton $\mathcal{S}_i$ recognising $\boldsymbol{S_i}$ is given control-states Q_i (the same as the automaton $\mathcal{R}_i$ recognising $\boldsymbol{R_i}$) and behaves in the same manner as $\mathcal{R}_i$ with the following adaptations compensating for the fact it is not a nested-tree automaton:

- It uses the π_3 component of a label to distinguish between nodes of $\mathbf{img}(E)$, $\mathbf{dom}(E)$ and those in neither.
- It ensures that the π_2 component of any node is equal to the state that it is in at that node.
- It uses the π_3 component of a label to get the state at the target of a pointer when at its source.
- It ensures that whenever the convolution branches there is either a discrepancy in pointer-status or π_1 of the labels.

□

Note that the arguments in both the theorems above could also be adapted to tree-isophilic and tree-nondisophilic structures, although they might become

even more fiddly. We do not do this as we do not have any nice characterisations of the tree-isophilic and tree-nondisophilic structures and so this result would be less meaningful than in the word case where nice characterisations do exist.

Chapter 7

Prefix Rewriting and Higher Order Pushdown Graphs

In this chapter we introduce three natural systems of prefix rewriting based on nested-words that generate precisely the respective classes of isophilic, dendrisophilic and symmetric-dendrisophilic structures. Arguably prefix-rewriting is a more pleasant presentation of these classes and can be viewed as a natural generalisation of rational prefix rewriting on standard words [28, 68]. Whilst this previously studied class captures precisely the ϵ-closures of (order-1) pushdown graphs, our rewrite systems respectively capture precisely the ϵ-closures of 2-PDA graphs (the second level of the Caucal hierarchy); 2-CPDA graphs and 2-CPDA graphs under symmetric closure. This reinforces the robustness of the first class and provides evidence for the robustness and naturality of the second two classes. We are able to separate the symmetric and standard 2-CPDA graphs showing that the *collapse* operation is indeed inherently asymmetric. This contrasts with 2-PDA graphs which are inherently symmetric—indeed Carayol *et al.* have made use of a graph-equivalent symmetric variant in studying their configuration graphs [23, 25, 27].

It should be noted that Carayol has his own form of generalised prefix rewriting [23] that generalises to all orders rather than just order-2 as in our case. This takes the form of a canonical presentation of a sequence of operations to generate a given stack, which in the special case of order-1 takes the form $pop_1^* push_1^*$, essentially a prefix rewrite rule. In some respects our nested-word presentation could be viewed as a recipe for generating a stack in terms of operations, a little like Carayol's canonical sequences of operations. In other respects it is quite different; isophilic structures arise naturally independent of any intuition concerning higher-order stack operations and the prefix rewriting more closely resembles the operation on words in the Caucal sense. Moreover, we are able to account for the addition of a *collapse* operation in a manner that fits neatly in our framework. The fact that we are able to re-obtain Kartow's

decidability result also suggests a substantive difference in our approach.

Finally we are able to use techniques regarding compact dendrisophilic structures from the previous chapter to show for the first time that 3_2-CPDA graphs (without ϵ-closure) have decidable ***FO*** theories.

7.1 Prefix Rewrite Systems

A *prefix-rewrite system* [28] traditionally consists of a regular string-language $\mathcal{L}$ together with pairs of regular languages $(\mathcal{L}_i, \mathcal{L}'_i)$ each assigned a label e_i. The *prefix recognisable graph* generated by such a system has $\mathcal{L}$ as its node-set and an e_i labelled edge from a word of the form uv to a word of the form $u'v$ whenever $u \in \mathcal{L}_i$ and $u' \in \mathcal{L}'_i$. Call the resulting class of structures $\boldsymbol{\mathcal{RW}}$. Such graphs coincide with the ϵ-closures of order-1 pushdown automata [68] and indeed from the resemblance of flat-isophilic structures to Blumensath's 'generalised prefix rewrite systems' [12] the reader may not be surprised to learn that they coincide precisely with flat-isophilic structures as well. The intuition behind the relationship is given in Figure 7.1. This suggests a generalisation of the idea to capture isophilic, dendrisophilic and symmetric-dendrisophilic structures.

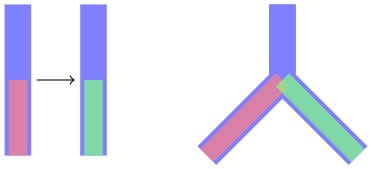


Figure 7.1: The relationship between isophilic structures and rewriting.

All generalisations have the form:

$$\mathcal{RW} = \left\langle \mathcal{L}, \mathcal{L}_1 \xrightarrow{e_1} \mathcal{L}'_1, \ldots, \mathcal{L}_k \xrightarrow{e_k} \mathcal{L}'_k \right\rangle$$

where $\mathcal{L}$ is a nested-word language over some alphabet Σ and $\mathcal{L}_i$, $\mathcal{L}'_i$ are regular languages over that same alphabet. The nodes of the graph generated are always the elements of $\mathcal{L}$. Given a nested-word of the form $(uv)^{\frown E}$ let us write $\mathfrak{V}(v^{\frown E})$ to mean the suffix of $\mathfrak{V}((uv)^{\frown E})$ corresponding to v. A

- *rat-rat* system has an e_i-labelled edge from $(uv)^{\frown E}$ to $(u'v)^{\frown E'}$ if $u \in \mathcal{L}_i$, $u' \in \mathcal{L}'_i$ and $\mathfrak{V}(v^{\frown E}) = \mathfrak{V}(v^{\frown E'})$. Denote this class by $\boldsymbol{\mathcal{RW}_{rr}}$.
- *sum-rat* system has an e_i-labelled edge from $(uv)^{\frown E}$ to $(u'v)^{\frown E'}$ if $\ulcorner u^{\frown E \restriction u} \urcorner \in \mathcal{L}_i$, $u' \in \mathcal{L}'_i$ and $\mathfrak{V}(v^{\frown E}) = \mathfrak{V}(v^{\frown E'})$. Denote this class by $\boldsymbol{\mathcal{RW}_{sr}}$.
- *sum-sum* system has an e_i-labelled edge from $(uv)^{\frown E}$ to $(u'v)^{\frown E'}$ if $\ulcorner u^{\frown E \restriction u} \urcorner \in \mathcal{L}_i$, $\ulcorner u'^{\frown E' \restriction u'} \urcorner \in \mathcal{L}'_i$ and $\mathfrak{V}(v^{\frown E}) = \mathfrak{V}(v^{\frown E'})$. Denote this class by $\boldsymbol{\mathcal{RW}_{ss}}$.

Given a re-write system $\mathcal{RW}$ we write $\mathcal{G}(\mathcal{RW})$ to denote the graph that it generates.

Example 7.1. The following is an example of a traditional rational prefix rewrite system [28]. Consider $\mathcal{L} = (a + b)^*$ and a rule:

$$a \ : \ a^*b \longrightarrow b^*a$$

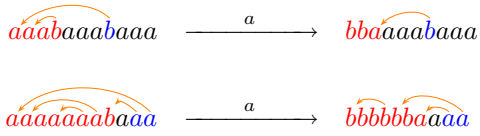
Then we have the following edges in the graph (the colour is only for illustrative purposes and does not differentiate symbols):

$$aaabaaabaaa \xrightarrow{a} bbaaaabaaa$$
$$aaaaaaabaaa \xrightarrow{a} bbbbbbaaaa$$

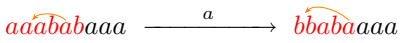
Example 7.2. We now give an example of a rat-rat system. Again consider an $\mathcal{L}$ based on $(a + b)^*$ that allows arbitrary well-nested pointers. The rule:

$$a \ : \ a^*b \longrightarrow b^*a$$

interpreted as a rat-rat rule this would give the following edges in the graph (again colours are purely illustrative and there is no formal distinction between letters of different colours):



To give an example of a sum-sum system, let us interpret the rule as a sum-sum rule. This now gives the following edge (which would belong to neither the rat-rat interpretation nor indeed the sum-rat interpretation for that matter).



Under a sum-sum rule we consider only the *summaries* of the left and right hand sides. (For a sum-rat rule we could only use summaries for the right-hand side). If more control over permissible pointer structure is needed this has to be enforced by restricting the domain of the graph $\mathcal{L}$.

7.2 Equivalence with Nested-Word Automataticity

We now explain how to precisely capture our notions of automaticity over nested-words using prefix rewriting. This requires decorating nested-words with states from automata that act upon them. This enables rewrite systems to observe the behaviour of automata used to define automatic structures.

State Decorations

Let $\mathcal{A}$ be a deterministic nested-word automaton. Define a map

$$\mathfrak{D}_{\mathcal{A}} \ : \ \mathbf{NWord}(\Sigma) \longrightarrow \mathbf{NWord}(\Sigma \times Q_{\mathcal{A}} \times Q_{\mathcal{A}}^{\perp} \times \mathbb{B})$$

such that $\mathfrak{D}_{\mathcal{A}}(w^{\frown E}) = w'^{\frown E}$ where $\pi_1(w') = w$; $\pi_2(w')$ gives the unique run of $\mathcal{A}$ on w; $\pi_3(w')$ is $\perp$ in each position u unless u lies in $\mathbf{dom}(E)$ in which case it gives the state of the run at $E(u)$; $\pi_4(w')$ is $\mathbf{t}$ only at those points in $w'^{\frown E}$ at which there is no unmatched target strictly preceding it. We write $\boldsymbol{symb}$, $\boldsymbol{st}_{\mathcal{A}}$, $\boldsymbol{st^{pr}}_{\mathcal{A}}$ and $\boldsymbol{cld}$ for π_1, π_2, π_3 and π_4 respectively with the subscripts used to disambiguate when we decorate with multiple deterministic nested-word automata at once.

It is easy to see that if $\mathcal{L}$ is recognised by a nested-word automaton, then its image $\mathfrak{D}_{\mathcal{A}}(\mathcal{L})$ must also be nested-word automaton recognisable. Observe that $\mathfrak{D}_{\mathcal{A}}(\mathcal{L})$ and $\mathcal{L}$ are in a natural bijective correspondence with each other. This will ensure that the prefix-rewrite system we define do indeed yield graphs isomorphic to the isophilic/dendrisophilic structures we consider.

We also define $\mathfrak{D}^{\star}_{\mathcal{A}}$ to be a map on *languages* that maps a Σ-labelled nested-word language $\mathcal{L}$ to a nested-word language $\mathcal{L}'$ over the alphabet $\Sigma \times Q_{\mathcal{A}} \times Q_{\mathcal{A}}^{\perp} \times \mathbb{B}$ such that $\pi_1(\mathcal{L}') = \mathcal{L}$ and for every $w^{\frown E} \in \mathcal{L}$ of length k and sequence $r_1, \ldots, r_k$ in $(Q_{\mathcal{A}} \times Q_{\mathcal{Q}}^{\perp} \times \mathbb{B})^*$, there exists $w'^{\frown E} \in \mathcal{L}'$ with $\pi_1(w'^{\frown E}) = w^{\frown E}$ and $\pi_{2,3,4}(w'^{\frown E}) = r_1, \ldots, r_k$. In particular $\mathfrak{D}^{\star}_{\mathcal{A}}(\Sigma)$ produces the alphabet for the image of Σ-labelled languages under $\mathfrak{D}_{\mathcal{A}}$. We can think of $\mathfrak{D}^{\star}_{\mathcal{A}}$ as ignoring the actual transition function of $\mathcal{A}$ but just providing a way to decorate words with all possible arbitrary combinations of state information.

The Equivalence Theorem

In the proof of the equivalence theorem it is easier to deal with 'suffix rewrite systems' rather than prefix rewriting. This is because the branching of convolutions are read by the automata defining an isophilic structure *after* the trunk of the convolution is read. However, suffix and prefix rewriting can be viewed as the same thing due to the following Lemma:

Lemma 7.3. *Let $\mathcal{L}$ be a language recognised by a nested-word automaton. Then the language*

$$\mathcal{L}^{\leftarrow} := \{ \ w^{\frown E^{\leftarrow}} \ : \ w^{\frown E} \in \mathcal{L} \ \}$$

where $w^{\frown E^{\leftarrow}}$ is the result of writing w front to back and E is the result of reversing the directions of pointers (so in particular interchanging $\mathbf{dom}(E)$ and $\mathbf{img}(E)$) is also recognised by a nested-word automaton.

Proof. Let $\mathcal{A}$ be a nested-word automaton recognising $\mathcal{L}$ with state space Q. We can construct the required automaton $\mathcal{A}^{\leftarrow}$ as having state space $Q \times Q \cup$

Q. When reading a word $w^{\curvearrowleft E}$ the automaton $\mathcal{A}^{\leftarrow}$ will begin in a final state of $\mathcal{A}$ and when transitioning non-deterministically choose a new state that could have yielded the current state if reading in a right-to-left direction. When transitioning to a node in $\mathbf{img}(E)$ it must simulate a right-to-left $\mathcal{A}$ transition from a node sourcing a pointer. In order to do that it needs to guess the state that will be at the corresponding $\mathbf{dom}(E)$ node and indicates this additional information using a pair in $Q \times Q$. This guess can then be verified when the corresponding $\mathbf{dom}(E)$ is reached (by checking back along the pointer for the guess that was made). □

Theorem 7.4. $\boldsymbol{\mathcal{RW}_{rr} = Iso_2}$ *and* $\boldsymbol{\mathcal{RW}_{sr} = dIso_2}$ *and* $\boldsymbol{\mathcal{RW}_{ss} = sIso_2}$.

Proof. First let us say that $\boldsymbol{\mathcal{RW}_{rr} \subseteq Iso_2}$ and $\boldsymbol{\mathcal{RW}_{sr} \subseteq dIso_2}$ and $\boldsymbol{\mathcal{RW}_{ss} \subseteq sIso_2}$ will be exhibited as a consequence of theorems later in this chapter. We will show that rewrite systems are subsumed by variants on order-2 pushdown graphs which are in turn subsumed by $\boldsymbol{Iso_2}$, $\boldsymbol{dIso_2}$ and $\boldsymbol{sIso_2}$. The difficulty of doing a 'direct' proof stems from the fact that *a priori* a rewrite rule may map a portion of the original word back to itself. This cannot be naïvely reconciled with the determinism constraints on the trunk of a convolution. In fact as a consequence of the equivalence (which includes the proof via higher-order automata) we see that it is never necessary to map a portion of the original word back to itself. This is highlighted by the proof below which produces rewrite systems with this property.

Here we will therefore just show that $\boldsymbol{\mathcal{RW}_{rr} \supseteq Iso_2}$ and $\boldsymbol{\mathcal{RW}_{sr} \supseteq dIso_2}$ and $\boldsymbol{\mathcal{RW}_{ss} \supseteq sIso_2}$.

Consider an isophilic structure $\mathfrak{A}$ with *deterministic* nested-word automaton $\mathcal{A}$ recognising its domain (we can determinise if necessary) and path-nested automata ${\mathcal{B}_i}^{\mathcal{C}_i}$ recognising convolutions for its e_i-labelled edges for $1 \leq i \leq k$. We define the deterministic word automaton $\wp(\mathcal{C}_i)$ to have state space $2^{Q_{\mathcal{C}_i}}$ and keep track of the possible states that $\mathcal{C}_i$ could be in if it had not yet branched (we implicitly assume that it has access to the current state of $\mathcal{B}_i$).

We define an equivalent rat-rat *suffix*-rewrite system, which by Lemma 7.3 is sufficient to deduce an equivalent *prefix*-rewrite system. We take as the domain of the rat-rat suffix-rewrite system the nested-word language:

$$\mathfrak{D}_{\mathcal{A},\mathcal{B}_1,\ldots,\mathcal{B}_k,\wp(\mathcal{C}_1),\ldots,\wp(\mathcal{C}_k)}(\mathfrak{V}(\mathcal{L}(\mathcal{A})))$$

Recall that this set of nodes is in bijective correspondence with $\mathcal{L}(\mathcal{A})$ and that $\mathfrak{V}$ just adds decorations indicating whether a position is the source of a pointer, target or neither.

We then add multiple suffix rewrite rules labelled e_i of the form:

$$\mathcal{L}_{i\vec{a},b,(p,q)} \longrightarrow \mathcal{L}'_{i\vec{a},b',(p,q)}$$

for each $\vec{a}$ in the alphabet $\Sigma \times Q_{\mathcal{A}} \times Q_{\mathcal{A}}^{\perp} \times \mathbb{B}$ of the domain, $b \neq b' \in \mathfrak{V}(\Sigma)$ and each pair $p, q \in Q_{\mathcal{C}_i}$ such that there exists an $r \in \boldsymbol{st}_{\mathcal{C}_i}(\vec{a})$ such that $\mathcal{C}_i$ could branch to (p, q) from r taking into account $\boldsymbol{st}_{\mathcal{B}_i}(a)$.

The elements of $\mathcal{L}_{i\vec{a},b,(p,q)}$ are defined to be those of the form $\vec{a}\vec{b}w$ where $\boldsymbol{symb}(\vec{b}) = b$ and $\boldsymbol{symb}(\vec{b})\boldsymbol{symb}(w)$ could be recognised by $\mathcal{C}_i$ starting in state p and taking the $\boldsymbol{st}_{\mathcal{B}_i}$ decorations to be the state of $\mathcal{B}_i$. There is no other restriction placed on w. $\mathcal{L}'_{i\vec{a},b'(p,q)}$ is defined to be similar except that it considers $\mathcal{C}_i$ beginning in state q and we consider b' in place of b. We thereby simulate the corresponding forking of the convolution both in terms of the behaviour of automata at the fork and the fact that a fork actually does occur in this position (as enforced by $b \neq b'$, which due to $\mathfrak{V}$ takes into account pointers). Note also that since $\mathcal{C}_i$ is a finite non-nested automaton it must be the case that both of these languages are regular. Hence this is indeed a rat-rat suffix rewrite system, as required.

Now consider a *dendrisophilic* structure $\mathfrak{A}$, again with deterministic nested-word automaton $\mathcal{A}$ recognising its domain, and spine nested automata $\mathcal{B}_i^{\mathcal{C}_i}$ recognising its e_i-edge convolutions. Define ${\mathcal{B}_i}^{\mathcal{C}_i{}^{\pitchfork}}$ to be the deterministic nested-word automaton with state space $2^{Q_{\mathcal{B}_i} \times Q_{\mathcal{C}_i} \times Q_{\mathcal{B}_i} \times Q_{\mathcal{C}_i}}$ where an element $((q, p), (q', p'))$ belonging to the state means that if ${\mathcal{B}_i}^{\mathcal{C}_i}$ had started in state (q, p) at the most recent target not discharged by a source, then ${\mathcal{B}_i}^{\mathcal{C}_i}$ could now be in state (q', p') assuming no branching. This is the same construction as used in the determinisation proof of nested-word automata [7] and, as seen there, a ${\mathcal{B}_i}^{\pitchfork}$ can be constructed to deterministically maintain this state.

We form the equivalent sum-rat suffix rewrite system (which as before may be converted to a prefix rewrite system) taking the domain to be:

$$\mathfrak{D}_{\mathcal{B}_1^s,\ldots,\mathcal{B}_k^s,\wp(\mathcal{C}_1),\ldots,\wp(\mathcal{C}_k),{\mathcal{B}_1}^{\mathcal{C}_1{}^{\pitchfork}},\ldots,{\mathcal{B}_k}^{\mathcal{C}_1{}^{\pitchfork}}}(\mathfrak{V}(\mathcal{L}(\mathcal{A})))$$

where $\mathcal{B}_i^s$ is $\mathcal{B}_i$ restricted to acting (deterministically) on a spine (*i.e.* pretending no branching ever takes place). In a manner similar to before we take a family of rewrite rules for each e_i of the form: $\mathcal{L}_{i\vec{a},b,(p,q)} \longrightarrow \mathcal{L}'_{i\vec{a},b',(p,q)}$ whenever $b \neq b' \in \mathfrak{V}(\Sigma)$ and $\mathcal{C}_i$ could branch from a state in $\boldsymbol{st}_{\wp(\mathcal{C}_i)}(\vec{a})$ to p down the left-hand branch and q down the right-hand branch.

The language $\mathcal{L}'_{i\vec{a},b',(p,q)}$ may be defined exactly as before because the right-hand branch of the convolution is essentially the same as in the isophilic case, treating $\mathcal{B}_i^s$ like the path-nested automaton. This language must thus also be regular.

The language $\mathcal{L}_{i\vec{a},b,(p,q)}$ is different to before (and not necessarily regular) as the left-hand branch of the convolution may be read non-deterministically. It is defined to consist of all words of the form $\vec{a}\vec{b}w$ where $\boldsymbol{symb}(\vec{b}) = b$ and ${\mathcal{B}_i}^{\mathcal{C}_i}$ could recognise $\vec{b}w$:

- starting in some state (r, p) where $\mathcal{B}_i$ would be able to branch-left off the

spine into state r from $\boldsymbol{st}_{\mathcal{B}_i^s}(\vec{a})$.

- treating all positions that are not pointer sources in $\vec{a}w$ but for which $\boldsymbol{st}^{\boldsymbol{pr}}{}_{\mathcal{B}_i^s} \neq \bot$ as being the source of a pointer with state $\boldsymbol{st}^{\boldsymbol{pr}}{}_{\mathcal{B}_i^s}$ at its target.

Given that $\vec{a}w$ is the suffix of a nested-word in the domain, the last item above ensures that we give $\mathcal{B}_i$ the correct state at the target of pointers where the target lies outside of the suffix.

It is easy to derive a nested-word automaton from ${\mathcal{B}_i}^{\mathcal{C}_i}$ that would recognise $\mathcal{L}_{i\vec{a},(p,q)}$. However, we wish to present this in terms of a language of summaries—*i.e.* we need to be able to recognise $\ulcorner \vec{a}\vec{b}w \urcorner$. We can do this with a finite-word automaton that reads $\ulcorner \vec{a}w \urcorner$ simulating ${\mathcal{B}_i}^{\mathcal{C}_i}$ starting in some state (r,p) as specified by the first point above. The second point above can also be achieved using a non-nested automaton since there is no need to actually look up a state at the target of a pointer in order to achieve this. So the only problem is simulating ${\mathcal{B}_i}^{\mathcal{C}_i}$ on pointers whose source and target both lie within the suffix $\vec{a}\vec{b}w$. This is enabled by looking at $\boldsymbol{st}_{{\mathcal{B}_i}^{\mathcal{C}_i}\Uparrow}$ in $\ulcorner \vec{a}\vec{b}w \urcorner$. Suppose there

is a segment $\cdots ab \cdots$ in the summary that originates from $\cdots \overset{\frown}{a \cdots b} \cdots$. The state $\boldsymbol{st}_{{\mathcal{B}_i}^{\mathcal{C}_i}\Uparrow}(b)$ will specify pairs $((q_1,q_2),(q_1',q_2'))$ such that starting at a in state (q_1,q_2), ${\mathcal{B}_i}^{\mathcal{C}_i}$ could begin in state (q_1,q_2) and end in state (q_1',q_2') at b. Our finite automaton in reading the summary is thus able to recall its state at a when reading b and compute a possible state of $\mathcal{B}_i^{\mathcal{C}_i}$ to simulate at b. Thus $\mathcal{L}_{i\vec{a},(p,q)}$ is summary-regular, as required.

Showing that every symmetric-dendrisophilic graph is also generated by a sum-sum rewrite system is very similar to the above. We just need to construct a summary language to handle the right-hand branch of a convolution in the same way that we handled the left-hand branch in the dendrisophilic case above. The only other difference is that we need to synchronise the $\mathcal{B}_i$ on the left and right branches. This was unnecessary in the dendrisophilic case as the right-hand branch was deterministic. In order to do this we just have rewrite rules of the form:

$$\mathcal{L}_{i\vec{a},b,((r_1,p),(r_2,q))} \longrightarrow \mathcal{L}'_{i\vec{a},b',((r_1,p),(r_2,q))}$$

where ${\mathcal{B}_i}^{\mathcal{C}_i}$ can branch from $(\boldsymbol{st}_{\mathcal{B}_i}, \boldsymbol{st}_{\mathcal{C}_i})$ to (r_1,p) down the left-hand branch and (r_2,q) down the right-hand branch. We then make (r_1,p) the starting state for the simulation of ${\mathcal{B}_i}^{\mathcal{C}_i}$ in $\mathcal{L}_{i\vec{a},((r_1,p),(r_2,q))}$ and (r_2,q) for $\mathcal{L}'_{i\vec{a},((r_1,p),(r_2,q))}$. □

The power of summaries over the mere underlying word arises from the fact that a summary implicitly reflects information about pointer structure. The proof of the theorem uses ideas from the determinisation proof for nested-word automata [7].

7.3 Collapsible Pushdown Automata

This section establishes the equivalence between our prefix rewrite systems or automatic structures and higher-order pushdown automata. The rat-rat rewrite systems/isophilic structures are precisely the ϵ-closures of order-2 pushdown graphs; the sum-rat rewrite systems/dendrisophilic structures are precisely the ϵ-closures of order-2 collapsible pushdown graphs and the sum-sum rewrite systems/symmetric-dendrisophilic structures correspond precisely to the ϵ-closures of 2-CPDA graphs transformed by some inverse rational map. We will see how the move from isophilic to dendrisophilic structures and the added power of sum-rat rewrite rules compared to rat-rat rewriting is precisely the expressivity needed to capture the power of the collapse operation at order-2. The equivalence of sum-rat rewriting with 2-CPDA graphs also confirms the intuition that *collapse* is an inherently asymmetric operation. Indeed we can use the results in this chapter to formally show that sum-sum rewriting is strictly more powerful than sym-rat so indeed the rational symmetric closure of 2-CPDA graphs generates more structures than the 2-CPDA graphs alone; this is another formal statement of the fact that *collapse* is inherently asymmetric. By contrast the ϵ-closures of 2-PDA graphs closed under rational symmetry. This is known from consideration of the Caucal hierarchy and indeed equivalence with Carayol's symmetric PDA [23, 27]. In our setting the rational symmetric closure of isophilic structures turns out to be another way of demonstrating this.

We will also see how tree-isophilic and sparse tree-nondisophilic structures subsume 3-PDA and 3-CPDA graphs, which gives us decidability of first-order logic. However, we have no corresponding tight characterisation as we do in the order-2 case, so other than it acting as a device for the decidability proof there is no further interest here.

Classes of (C)PDA Graphs

We write $\mathbf{Pd}_{\boldsymbol{\epsilon} n}$ to denote the nth level of the Caucal hierarchy—equivalently the ϵ-closures of n-PDA graphs. We write $\mathbf{Pd}_{\boldsymbol{\epsilon} n}^{\mathcal{C}}$ to denote the class of ϵ-closures of n-CPDA and $\mathbf{Pd}_{\boldsymbol{\epsilon} n}^{\mathcal{S}}$ to denote the symmetric closures thereof—*i.e.* where we may for a particular graph reverse the direction of some transitions (including ϵ-transitions). The class $\boldsymbol{\pi}\mathbf{Pd}_{\boldsymbol{\epsilon} n}$ is the class of n-PDA ϵ-closures *quotiented by a stack projection*. If π is a projection on atomic stack elements, then we form a member of $\boldsymbol{\pi}\mathbf{Pd}_{\boldsymbol{\epsilon} n}$ from a graph in $\mathbf{Pd}_{\boldsymbol{\epsilon} n}$ where we deem configurations (q, s) and (q', s') to be the same node if $\pi(s) = \pi(s')$. Likewise the class $\boldsymbol{\pi}\mathbf{Pd}_{\boldsymbol{\epsilon} n}^{\mathcal{C}}$ is the class of n-CPDA ϵ-closures quotiented by stack projection.

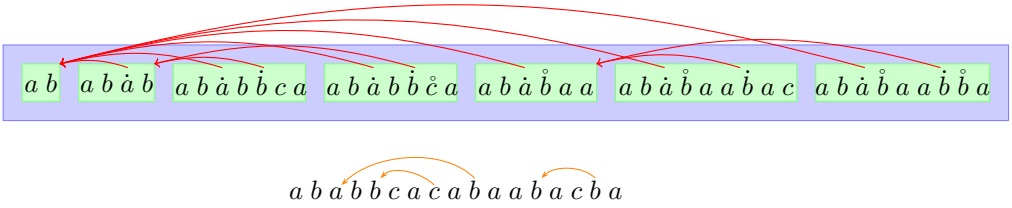


Figure 7.2: The 2-CPDA stack corresponding to a nested-word

From Rewriting to Higher-Order Pushdown Graphs

We define maps encoding nested-words as order-2 stacks (where $stack_2(\Gamma)$ and $stack_2^{\mathcal{C}}(\Gamma)$ respectively denote the 2-PD and 2-CPD stacks over the stack-alphabet Γ):

$$\mathfrak{S} : \mathbf{NWord}(\Sigma) \longrightarrow stack_2(\mathfrak{V}(\Sigma))$$
$$\mathfrak{S}^{\mathcal{C}} : \mathbf{NWord}(\Sigma) \longrightarrow stack_2^{\mathcal{C}}(\mathfrak{V}(\Sigma))$$

recalling that $\mathfrak{V}(\Sigma) := \mathring{\Sigma} \uplus \Sigma \uplus \dot{\Sigma}$ where $\mathring{\Sigma}$ and $\dot{\Sigma}$ are disjoint copies of Σ whose elements are respectively indicated as $\mathring{\sigma}$ and $\dot{\sigma}$ for each $\sigma \in \Sigma$. Consider $w^{\curvearrowleft E} \in \mathbf{NWord}(\Sigma)$. We define the map $\mathfrak{S}^{\mathcal{C}}$ as follows:

$$\begin{aligned}
\mathfrak{S}^{\mathcal{C}}(\epsilon) &:= \bot_2 \\
\mathfrak{S}^{\mathcal{C}}(v\ a) &:= \begin{cases} (push_2; push_1^{\dot{a},2})(\mathfrak{S}(v)) \text{ if } a \in \mathbf{img}(E) \\ push_1^{a}(\mathfrak{S}(v)) \text{ if } a \notin \mathbf{img}(E) \cup \mathbf{dom}(E) \end{cases} \\
\mathfrak{S}^{\mathcal{C}}(v\ a \cdots c\ b) &:= push_1^{\mathring{b}}(\mathfrak{S}^{\mathcal{C}}(v\ a\ \cdots\ c)\ ::\ top_2(\mathfrak{S}^{\mathcal{C}}(v\ a)))
\end{aligned}$$

We define $\mathfrak{S}(w^{\curvearrowleft E})$ in the same way as $\mathfrak{S}^{\mathcal{C}}(w^{\curvearrowleft E})$, ignoring stack pointers.

Example 7.5. Consider the nested-word:

$$w^{\curvearrowleft E} := a\ b\ a\ b\ b\ c\ a\ c\ a\ b\ a\ a\ b\ a\ c\ b\ a$$

We illustrate $\mathfrak{S}^{\mathcal{C}}(w^{\curvearrowleft E})$ in Figure 7.2.

The next lemma gives us some information on how a CPDA might 'compute' a representation of a nested-word on the fly.

Lemma 7.6. *For any $w^{\curvearrowleft E} \in \mathbf{NWord}(\Sigma)$ the following equalities hold:*

$$\ulcorner w^{\curvearrowleft E} \urcorner = top_2(\mathfrak{S}^{\mathcal{C}}(w^{\curvearrowleft E}))$$
$$\mathfrak{S}(w^{\curvearrowleft E\restriction_{\mathbf{dom}(w)}}) = \begin{cases} pop_2(\mathfrak{S}^{\mathcal{C}}((w\ a)^{\curvearrowleft E})) & \text{if } a \in \mathbf{dom}(E) \cup \mathbf{img}(E) \\ pop_1(\mathfrak{S}^{\mathcal{C}}((w\ a)^{\curvearrowleft E})) & \text{otherwise} \end{cases}$$
$$\mathfrak{S}^{\mathcal{C}}(w^{\curvearrowleft E}) = (pop_1; collapse)(\mathfrak{S}^{\mathcal{C}}(w^{\curvearrowleft E}\ a \cdots b))$$

Proof. Let $\mathfrak{S}^{\mathfrak{V}}$ be a map defined on nested-words over $\mathfrak{V}(\Sigma)$ that is defined exactly as $\mathfrak{S}$ except that references to $a \in \mathbf{img}(E)$ are replaced with 'a is of the form $\dot{b}$'. It is sufficient to show that the statement of the lemma holds for every prefix $u^{\frown F}$ of $\mathfrak{V}(w^{\frown E})$ replacing $\mathfrak{S}$ with $\mathfrak{S}^{\mathfrak{V}}$ and $a \in \mathbf{img}(F)$ with 'a is of the form $\dot{b}$'.

We establish this by induction on the length of the prefix u matching against the left-hand-side of the first item and the right-hand-side of the second, third and fourth items. We need a slightly strengthened induction hypothesis with respect to the third item. We use:

$$\ulcorner u^{\frown F} \urcorner = top_2(\mathfrak{S}^{\mathfrak{V}}(u^{\frown F}))$$

$$\mathfrak{S}^{\mathfrak{V}}(u^{\frown F \restriction_{\mathbf{dom}(u)}}) = \begin{cases} pop_2(\mathfrak{S}^{\mathfrak{V}}((u\ a)^{\frown F})) & \text{if } a \in \mathbf{dom}(F) \text{ or of the form } \dot{b} \\ pop_1(\mathfrak{S}^{\mathfrak{V}}((u\ a)^{\frown F})) & \text{otherwise} \end{cases}$$

$$top_2(\mathfrak{S}^{\mathfrak{V}}(u^{\frown F \restriction_{\mathbf{dom}(u)}}\ \dot{a})) = top_2(pop_1(\mathfrak{S}^{\mathfrak{V}}(u\ a \overset{\frown}{\cdots} b)))$$

$$\mathfrak{S}^{\mathfrak{V}}(u^{\frown F}) = collapse(\mathfrak{S}^{\mathfrak{V}}(u^{\frown F}\ \dot{a}))$$

The third and fourth items above together imply the third item in the lemma since the target of a link in a CPDA remains fixed and so *collapse* on a copy of a particular stack element will always yield the same result regardless of the copy on which it is applied. The empty word trivially satisfies the conditions (in particular it matches neither pattern in the second item nor the patterns in the third or fourth items, so these hold vacuously). For the induction step consider the prefix to be of the form $u\ a^{\frown F}$.

If a is of the form $\dot{b}$ (so $a \in \mathbf{img}(E)$) we have:

$$\mathfrak{S}(u\ a^{\frown F}) = push_1^{\dot{b},2}(push_2(\mathfrak{S}^{\mathfrak{V}}(u^{\frown F \restriction_{\mathbf{dom}(u)}})))$$

This immediately gives us $pop_2(\mathfrak{S}(u\ a^{\frown F})) = \mathfrak{S}(u^{\frown F \restriction_{\mathbf{dom}(u)}})$, as required for the second item. Since the top_2 stack is the same other than the addition of $\dot{a}$ on top the induction hypothesis also gives us the first item. The fourth item holds by the induction hypothesis because *collapse* will have the effect of pop_2, which returns us to the original stack. The third item holds vacuously because we do not match the pattern on the right-hand-side.

Now consider the case when a is not of the form $\dot{b}$ and is also not the source of a pointer. Then a is just pushed onto the stack and so the second item holds and the first does by the induction hypothesis. The third and fourth hold trivially as we do not match the pattern on the right-hand side.

Now consider the case when the prefix is of the form: $u^{\frown F} a \overset{\frown}{v^{\frown F'}} b$. Then we have:

$$\mathfrak{S}^{\mathfrak{V}}(u^{\frown F}\ a \overset{\frown}{v^{\frown F'}}\ b) \ := \ [\ \mathfrak{S}^{\mathfrak{V}}(u^{\frown F}\ a\ v^{\frown F'})\ [\ top_2(\mathfrak{S}^{\mathfrak{V}}(u^{\frown F}\ \dot{a}))\ \mathring{b}\ ]\ ]$$

Thus $top_2(pop_1(\mathfrak{S}^{\mathfrak{V}}(u \frown^F a \overset{\frown}{v \frown^{F'} b})))$ $= top_2(\mathfrak{S}^{\mathfrak{V}}(u \frown^F \dot{a}))$ as required by the third item. Also note that by the induction hypothesis the top_2 of the stack will be the summary (which deletes v), thereby satisfying item one. Since pop_2 gives us the stack corresponding to the prefix before adding $\mathring{b}$ the second item must be satisfied and the fourth item is trivially satisfied as we do not match the pattern.

□

Lemma 7.7. *Every rat-rat prefix-rewrite graph is the ϵ-closure of a 2-PDA graph.*

Proof. By Lemma 7.3 we can consider rat-rat *suffix* rewrite systems. Consider a rat-rat suffix-rewrite system:

$$\mathcal{RW} := \left\langle \mathcal{L}, \mathcal{L}_1 \xrightarrow{e_1} \mathcal{L}'_1, \ \ldots, \ \mathcal{L}_k \xrightarrow{e_k} \mathcal{L}'_k \right\rangle$$

Let $\mathcal{A}$ be a *deterministic* nested-word automaton recognising $\mathcal{L}$. Consider the following suffix rewrite system:

$$\mathcal{RW}' := \left\langle \mathfrak{D}_{\mathcal{A}}(\mathcal{L}), \mathfrak{D}^{\star}_{\mathcal{A}}(\mathcal{L}_1) \xrightarrow{e_1} \mathfrak{D}^{\star}_{\mathcal{A}}(\mathcal{L}'_1), \ \ldots, \ \mathfrak{D}^{\star}_{\mathcal{A}}(\mathcal{L}_k) \xrightarrow{e_k} \mathfrak{D}^{\star}_{\mathcal{A}}(\mathcal{L}'_k) \right\rangle$$

Since $\mathcal{A}$ is deterministic, $\mathcal{L}$ and $\mathfrak{D}_{\mathcal{A}}(\mathcal{L})$ are in bijective correspondence and so $\mathcal{G}(\mathcal{RW}) = \mathcal{G}(\mathcal{RW}')$. It thus suffices to construct a 2-PDA $\mathcal{B}$ such that $\mathcal{G}(\mathcal{RW})'$ is the ϵ-closure of $\mathcal{B}$. We describe the permissible behaviours of the automaton when generating an e_i edge (with the ϵ-closure implicit in the unbounded number of steps that this might take—we do not mention control-states explicitly as we may assume that there is a fixed (initial) control-state which begins and ends each edge). Each such run terminates with an e_i-labelled edge (and all other transitions ϵ).

Let $\mathcal{A}_i$ and $\mathcal{A}'_i$ be *deterministic* finite-state (non-nested) automata recognising the regular languages $\mathcal{L}_i^{\leftarrow}$ (the result of reversing all elements of $\mathcal{L}_i$ as in Lemma 7.3) and $\mathcal{L}'_i$ respectively with respective state spaces $Q_{\mathcal{A}_i}$ and $Q_{\mathcal{A}'_i}$ and transition functions $\delta_{\mathcal{A}_i}$ and $\delta_{\mathcal{A}'_i}$. An e_i-edge is generated by $\mathcal{B}$ as follows, noting that it keeps a register $\mathbf{q}$ with value in $Q_{\mathcal{A}_i}$ and a register $\mathbf{q}'$ with value in $Q_{\mathcal{A}'_i}$:

- Set $\mathbf{q} := q_0$ where q_0 is the initial state of $\mathcal{A}_i$
- *Phase 1:* If the value of $\mathbf{q}$ is an accepting state of $\mathcal{A}_i$, then a non-deterministic choice is made whether to jump to *phase 2a* or to continue with *phase 1*. If it is not accepting, then *phase 1* must continue.
- – If $\boldsymbol{symb}(top_1) = a \in \Sigma$, then a pop_1 is performed and we set $\mathbf{q} := \delta_{\mathcal{A}_i}(a, \mathbf{q})$.

  - If $\boldsymbol{symb}(top_1) = \mathring{a} \in \mathring{\Sigma}$, then we perform pop_2 and set $\mathbf{q} := \delta_{\mathcal{A}_i}(a, \mathbf{q})$
  - If $\boldsymbol{symb}(top_1) = \dot{a} \in \dot{\Sigma}$, then we perform pop_2 and set $\mathbf{q} := \delta_{\mathcal{A}_i}(a, \mathbf{q})$

- Return to the start of *phase 1.*
- *Phase 2a:* Set $\mathbf{q}' := q'_0$ where q'_0 is the initial state of $\mathcal{A}'_i$.
- *Phase 2b:* If the value of $\mathbf{q}'$ is an accepting state of $\mathcal{A}'_i$ *and* $\boldsymbol{st}_{\mathcal{A}}(top_1)$ is an accepting state of $\mathcal{A}$ *and* $\boldsymbol{cld}(top_1) = \mathbf{t}$, then we non-deterministically decide whether to perform an e_i transition and terminate or continue.
- Non-deterministically pick $a \in \Sigma$. Do one of the following:
  - Perform $push_1^{(a,q',c)}$ where $q' := \delta_{\mathcal{A}}(a, \boldsymbol{st}_{\mathcal{A}}(top_1))$ and
    $$c = \begin{cases} \boldsymbol{cld}(top_1) & \text{if } \boldsymbol{symb}(top_1) \notin \dot{\Sigma} \\ \mathbf{f} & \text{if } \boldsymbol{symb}(top_1) \in \dot{\Sigma} \end{cases}$$
    Set $\mathbf{q}' := \delta_{\mathcal{A}'_i}(a, \mathbf{q}')$.
  - Perform $(push_2; push_1^{(\dot{a},q,c)})$ where $q := \delta_{\mathcal{A}}(a, \boldsymbol{st}_{\mathcal{A}}(top_1))$ and $c = \boldsymbol{cld}(top_1)$.
    Set $\mathbf{q}' := \delta_{\mathcal{A}'_i}(a, \mathbf{q}')$.
  - If $\boldsymbol{cld}(top_1) = \mathbf{t}$ then we may do the following. Perform $push_2$ followed by iterated pop_1 precisely until one has as the top element u such that $\boldsymbol{symb}(u) \in \dot{\Sigma}$ that is *not* followed by an element with $\boldsymbol{symb}$ in $\mathring{\Sigma}$. We should then perform $push_1^{(\mathring{a},q',q,c)}$ where $c = \boldsymbol{cld}(top_1)$, $q' = \boldsymbol{st}_{\mathcal{A}}(u)$ and $q = \delta_{\mathcal{A}}(a, q', q)$.
    Set $\mathbf{q}' := \delta_{\mathcal{A}'_i}(a, \mathbf{q}')$.

For the following induction argument we need to consider a nested-word $w^{\curvearrowleft E}$ to really be talking about $\mathfrak{V}(w^{\curvearrowleft E})$ (so that we can have 'dangling pointers').

Suppose that the initial stack configuration represents $\mathfrak{S}(w^{\curvearrowleft E})$ for some $w^{\curvearrowleft E} \in \mathfrak{D}_{\mathcal{A}}(\mathbf{NWord}(\Sigma))$. For phase one we can check by induction on k that after k iterations (of phase one) we have a stack equal to $\mathfrak{S}(w'^{\curvearrowleft E\restriction_{w'}})$ where $w = w'\ a_1 \cdots a_k$ and $\mathbf{q}$ is set to the state in which $\mathcal{A}_i$ would be after reading $\boldsymbol{symb}(a_k) \cdots \boldsymbol{symb}(a_1)$. The induction step for the former comes via the second equality of Lemma 7.6 and the induction step for the latter is immediate from the way in which $\mathbf{q}$ is updated.

Given the condition for terminating phase one (and moving to phase two) this tells us that if phase one begins with stack configuration $\mathfrak{S}(w^{\curvearrowleft E})$, then if it terminates it must terminate with stack configuration of the form $\mathfrak{S}(w'^{\curvearrowleft E\restriction_{w'}})$ where $w = w'\ a_1 \cdots a_k$ and $a_k \cdots a_1 \in \mathfrak{D}^{\star}_{\mathcal{A}}(\mathcal{L}_i^{\leftarrow})$ and so $a_1 \cdots a_k \in \mathfrak{D}^{\star}_{\mathcal{A}}(\mathcal{L}_i)$.

Conversely since $\mathcal{A}_i$ is deterministic and we can easily check by induction that $\mathcal{B}$ can proceed with phase one indefinitely (at least until the stack becomes empty) we can see that if $a_1 \cdots a_k \in \mathfrak{D}^\star_{\mathcal{A}}(\mathcal{L}_i)$ then $\mathcal{B}$ can begin phase one with $\mathfrak{S}(w^{\frown E})$ and end with $\mathfrak{S}(w'^{\frown E 1_{w'}})$ with w and w' as above.

Now suppose that at the end of phase one the stack configuration is $\mathfrak{S}(w'^{\frown E'})$. We may again argue by induction on k that after k steps of phase two we have a stack $\mathfrak{S}(v^{\frown F})$ such that $v^{\frown F} \in \mathfrak{D}_{\mathcal{A}}(\mathbf{NWord}(\Sigma))$ and $v^{\frown F} = w'^{\frown E'}\ a_1 \cdots a_k$ for some $a_1, \ldots, a_k$ and moreover $\mathbf{q}'$ is the state in which $\mathcal{A}'_i$ would be after reading $\boldsymbol{symb}(a_1) \cdots \boldsymbol{symb}(a_k)$. The only substantive part of the induction step is when pushing an element with $\boldsymbol{symb}$ of the form $\mathring{a}$ onto the stack. For this consider the first part of Lemma 7.6. Since the top_2 stack prior to this operation must be the summary of the word hitherto created, the highest $\dot{b}$ that is not followed immediately by a $\mathring{b'}$ must indeed be the most recent unmatched target of a pointer.

Given the condition for terminating phase two this tells us that if phase two begins with the stack $\mathfrak{S}(w'^{\frown E'})$ and terminates with the stack $\mathfrak{S}(v^{\frown F})$ where $v = w'^{\frown E'}\ a_1 \cdots a_k$ then $a_1 \cdots a_k \in \mathcal{A}'_i$, and so $a_1 \cdots a_k \in \mathcal{L}'_i$. Moreover we must have $v^{\frown F} \in \mathcal{L}$ since it is ensured that the $\boldsymbol{st}_{\mathcal{A}}$ of the top element is an accepting state of $\mathcal{A}$. Since $\mathcal{B}$ is allowed to simulate the addition of an arbitrary symbol with arbitrary link (except for an element in $\dot{\Sigma}$ when there is no corresponding $\mathring{\Sigma}$ to link to) the converse must also hold.

By combining these properties of phases one and two we may thus conclude that $\mathcal{B}$ emits an e_i edge precisely when required and that starting with a stack corresponding to an element of $\mathcal{L}$ ensures that the stack at the end of phase 2 also corresponds to an element of $\mathcal{L}$. Therefore all configurations of $\mathcal{B}$ reachable from a configuration encoding an element of the domain of the rewrite system do themselves encode elements in the domain of the rewrite system. In order to ensure that the domain of the ϵ-closure contains the correct elements we can pick an arbitrary (either fresh or non-fresh) automaton edge label ρ and map it to the rewrite rule $\{\epsilon\} \longrightarrow \mathcal{L}$. This will then generate $\mathcal{L}$ from the initial configuration, and by the definition of ϵ-closure for CPDA, the ρ in these edges will be discounted and it will be used purely to generate the domain, provided that we ensure the initial configuration is never subsequently reached. This requirement can be fulfilled by choosing a fresh initial control-state that is never reused.

□

Lemma 7.8. *Every sum-rat prefix-rewrite graph is the ϵ-closure of some 2-CPDA graph.*

Proof. Again Lemma 7.3 allows us to consider instead suffix rewrite systems.

Consider a sum-rat suffix-rewrite system:

$$\mathcal{RW} := \left\langle \mathcal{L}, \mathcal{L}_1 \xrightarrow{e_1} \mathcal{L}'_1, \ldots, \mathcal{L}_k \xrightarrow{e_k} \mathcal{L}'_k \right\rangle$$

Let $\mathcal{A}$ be a *deterministic* nested-word automaton recognising $\mathcal{L}$. Consider the following suffix rewrite system:

$$\mathcal{RW}' := \left\langle \mathfrak{D}_{\mathcal{A}}(\mathcal{L}), \mathfrak{D}^{\star}_{\mathcal{A}}(\mathcal{L}_1) \xrightarrow{e_1} \mathfrak{D}^{\star}_{\mathcal{A}}(\mathcal{L}'_1), \ldots, \mathfrak{D}^{\star}_{\mathcal{A}}(\mathcal{L}_k) \xrightarrow{e_k} \mathfrak{D}^{\star}_{\mathcal{A}}(\mathcal{L}'_k) \right\rangle$$

Let $\mathcal{A}_i$ be a deterministic finite automaton recognising the regular language $\ulcorner \mathcal{L}_i{}^{\leftarrow} \urcorner$, which must exist and have the property '$w^{\curvearrowleft E} \in \mathcal{L}_i{}^{\leftarrow}$ iff $\ulcorner w^{\curvearrowleft E} \urcorner \in \ulcorner \mathcal{L}_i{}^{\leftarrow} \urcorner$' since $\mathcal{L}_i$ (and hence $\mathcal{L}_i{}^{\leftarrow}$) is summary recognisable. Let $\mathcal{A}'_i$ be a finite automaton recognising the regular language $\mathcal{L}'_i$. We can construct a 2-CPDA $\mathcal{B}$ in a manner very similar to the 2-PDA in the proof of Lemma 7.7. Indeed phase 2 is identical with the exception that all $push_1$ operations should now attach 2-links. This works since $\mathcal{L}'_i$ is a regular language as with the case for rat-rat rewrite systems. We thus give only the analogue for phase one here (in preparation to emit an e_i labelled edge):

- Set $\mathbf{q} := q_0$ where q_0 is the initial state of $\mathcal{A}_i$:
- *Phase 1:* If the value of $\mathbf{q}$ is an accepting state of $\mathcal{A}_i$, then a non-deterministic choice is made whether to jump to the second phase (as in proof of Lemma 7.7) or to continue with *phase 1.* If it is not accepting, then *phase 1* must continue.
- – If $\boldsymbol{symb}(top_1) = a \in \Sigma$, then a pop_1 is performed and we set $\mathbf{q} := \delta_{\mathcal{A}_i}(a, \mathbf{q})$.
  – If $\boldsymbol{symb}(top_1) = a \in \mathring{\Sigma}$, then a non-deterministic choice is made to either:
    * Perform a pop_2 and set $\mathbf{q} := \delta_{\mathcal{A}_i}(a, \mathbf{q})$
    * Perform a pop_1 to result in an element $b \in \dot{\Sigma}$ (by point 1 of Lemma 7.6) and set $\mathbf{q} := \delta_{\mathcal{A}_i}(a, \delta_{\mathcal{A}_i}(b, \mathbf{q}))$. Then perform *collapse*.
  – If $\boldsymbol{symb}(top_1) \in \dot{\Sigma}$, then we must terminate with failure (i.e. no edge is emitted).

We argue by induction on k that after k iterations of phase one starting with a stack $\mathfrak{S}(w^{\curvearrowleft E})$ the stack will be of the form $\mathfrak{S}(w'^{\curvearrowleft E \restriction w'})$ such that $w' = tv$ with $v = a_1u_1b_1a_2u_2b_2 \cdots a_ku_kb_k$ where $u_i = b_i = \epsilon$ iff $a_i \notin \mathbf{img}(E \restriction_v)$. Otherwise $E(b_i) = a_i$. Moreover $\mathbf{q}$ is the state in which $\mathcal{A}_i$ would be after reading $b_ka_kb_{k-1}a_{k-1} \cdots b_1a_1$. The induction step is provided by Lemma 7.6 points two and three.

Conversely it must be possible for the automaton to arrive after k steps to any such $\mathfrak{S}(w'^{\curvearrowleft E\restriction w'})$. This is because the only reason the automaton will stall is if it begins an iteration with a symbol of the form $\dot{a}$ on top. But this will only happen if the corresponding $\mathring{a}'$ was discarded previously without discarding $\dot{a}$ (by means of a pop_2). Thus one could reach the representation of the word resulting from discarding $\dot{a}$ by instead going with the $pop_1; collapse$ operation at this previous point when pop_2 was instead used. Opting not do discard $\dot{a}$ corresponds to the case when one does not want to include $\dot{a}$ in the suffix but one does want to include the corresponding $\mathring{a}$—*i.e.* the case when the pointer from a' to a is not included in the suffix despite its source being included.

This ensures that a phase one may start with $\mathfrak{S}(w^{\curvearrowleft E})$ and end with $\mathfrak{S}(w'^{\curvearrowleft E\restriction w'})$ just in case $w = tv$ and $v^{\curvearrowleft E\restriction v} \in \mathcal{L}_i$.

As with Lemma 7.7 we can set up 'dummy ρ transitions' for the rewrite rule $\{\epsilon\} \longrightarrow \mathcal{L}$ to correctly set the domain. □

Lemma 7.9. *Every sum-sum prefix-rewrite graph is the symmetric ϵ-closure of some 2-CPDA graph.*

Proof. Let $\mathcal{L}$ over the alphabet Σ be the domain of the sum-sum prefix rewrite system and suppose the system has rules $\mathcal{L}_i \xrightarrow{e_i} \mathcal{L}'_i$ for $i \in I$ where I is an index set. Consider the language:

$$\mathcal{L}' := \{\, iw' \,:\, ww' \in \mathcal{L} \text{ for some } w \text{ and } i \in I\}$$

over the alphabet $\Sigma \uplus I$. We can then construct a sum-rat rewrite system with domain $\mathcal{L}'$ and rules of the form:

$$\mathcal{L}_i \xrightarrow{\epsilon} \{\, i \,\}$$
$$\mathcal{L}'_i \xrightarrow{\overline{e_i}} \{\, i \,\}$$

for each $i \in I$. We can also add a sum-rat rewrite rule:

$$\{\, \epsilon \,\} \xrightarrow{\rho'} \mathcal{L}$$

By Lemma 7.8 there exists a 2-CPDA $\mathcal{A}$ whose ϵ closure is the graph generated by the sum-rat rewrite system. We can thus take the symmetric ϵ-closure of $\mathcal{A}$. □

From Automata to Isophilic Structures

This is inspired by Kartzow's work on 2-CPDA and tree-automaticity [48]. We reformulate it to show how the progression from words to nested-words to nested-trees corresponds well to the journey from order-1 to order-3 automata.

Given an automaton $\mathcal{A}$ we consider its *stack graph* which represents its behaviour in terms of edges annotated with pairs of control-states and whose configurations consist only of a stack.

Definition 7.10. Let $\mathcal{A}$ be an n-(C)PDA for some $n \in \mathbb{N}$. Its *stack graph* $\mathcal{G}_s(\mathcal{A})$ has *stacks* belonging to reachable configurations of $\mathcal{A}$ as nodes and a directed relation $\boldsymbol{R_{p,a,q}}$ between stacks s and s' whenever there is an a-edge from (p, s) to (q, s') in $\mathcal{G}(\mathcal{A})$. The *ϵ-closed stack-graph* $\mathcal{G}^{\epsilon}_s(\mathcal{A})$ is defined in the same way except there is a directed relation $\boldsymbol{R_{p,a,q}}$ between stacks s and s' whenever there is an a-edge from (p, s) to (q, s') in $\mathcal{G}^{\epsilon}(\mathcal{A})$ and the nodes of the graph are the stacks belonging to configurations in the ϵ-closure.

We are naturally able to make both $\mathcal{G}_s(\mathcal{A})$ and $\mathcal{G}^{\epsilon}_s(\mathcal{A})$ a $\mathcal{Q}$-decorated structure, assigning pairs of control-states of $\mathcal{A}$ to each relation. In both cases we set $\mathcal{Q}(\boldsymbol{R_{p,a,q}}) := (p, q)$.

It is easy to see that for any order-1 automaton $\mathcal{A}_1$ the graph $\mathcal{G}_s(\mathcal{A})$ is flat-isophilic. If Γ is the stack alphabet, then stacks can be represented by finite-words in Γ^*. Since an individual stack operation changes the height of the stack by at most one, convolutions representing single transitions have branches of length at most one eminating from the tip of the trunk and so can easily be recognised by a finite tree automaton. Given the judicious choice of $\mathcal{Q}$-decoration, Theorem 6.39 then gives us a flat-isophilic representation of $\mathcal{G}^{\epsilon}_s(\mathcal{A})$—we first get a representation of ϵ^* labelled paths using the theorem, and can then concatenate a single non-ϵ edge. This is the same as Stirling's theorem that the ϵ-closures of 1-PDA graphs coincide with rational prefix rewrite systems [68]. We express this result as:

Lemma 7.11. *Let $\mathcal{A}$ be a 1-PDA. Then $\mathcal{G}^{\epsilon}_s(\mathcal{A})$ is flat-isophilic.*

With the use of derivatives and flat-isophilic chains we can continue the hierarchy with the following:

Lemma 7.12. *Let $\mathcal{A}$ be a 2-PDA, then $\mathcal{G}^{\epsilon}_s(\mathcal{A})$ is isophilic. Let $\mathcal{A}'$ be a 3-PDA. Then $\mathcal{G}^{\epsilon}_s(\mathcal{A}')$ is tree-isophilic.*

Proof. Let $\mathcal{A}$ be a 2-PDA. We first show that just the slow 2-PDA graph is isophilic, which is the restricted ϵ-closure for which no ϵ-transition uses a $push_2$ or pop_2 operation.[1]

Consider its derivative $\partial(\mathcal{A})$ (introduced in Chapter 5), which is a 1-PDA. From the remarks above we know that $\mathcal{G}^{\epsilon}_s(\partial(\mathcal{A}))$ is flat isophilic. We add the relation: $\boldsymbol{R_{q,r^p(r_\epsilon+\Sigma)^*,q'}}$ relating (q, s) and (q', s') such that $(q, s)\boldsymbol{r}_{r^p(r_\epsilon+\Sigma)^*}(q', s')$. We give this relation the $\mathcal{Q}$-decoration (q, q').

Let us further add a unary predicate $\boldsymbol{c_0 R_{r^*_\epsilon,q}}$ that is satisfied by precisely those nodes s such that (q, s) is reachable from the initial configuration of

[1] The reader may notice that this contradicts our generality preserving assumption used in Chapter 5 that says the opposite: $push_2$ and pop_2 should always be ϵ-transitions. However, this assumption was mainly helpful for meta-annotations, which are not used here. For derivatives we may think of r^p as a $push_2$ operation for *either* an ϵ-edge or a Σ-labelled edge.

$\partial(\mathcal{A})$ via an r_ϵ^*-labelled path. All of these additions preserve flat-isophilicity by Theorem 6.39. Let us give the graph modifed to include *only* $\boldsymbol{R_{q,r^{\mathcal{P}}(r_\epsilon+\Sigma)^*,q'}}$ and $\boldsymbol{c_0 R_{r_\epsilon^*,q}}$ the name $\mathcal{G}_s^\epsilon(\partial(\mathcal{A}))^+$.

By Lemma 5.24 we can represent precisely the nodes of $\mathcal{G}_s^\epsilon(\mathcal{A})$ by the elements of the set:

$$S := \{\, C \in \boldsymbol{Ch_\blacktriangleleft}(\mathcal{G}_s^\epsilon(\partial(\mathcal{A}))^+) \,:\, \text{the initial item in } C \text{ satisfies } \boldsymbol{c_0 R_{r_\epsilon^*,q}} \text{ and has decoration } q \text{ for some } q \,\}$$

This set is over approximated by $\boldsymbol{Ch_\blacktriangleleft}(\mathcal{G}_s^\epsilon(\partial(\mathcal{A}))^+)$. Note how the automaton recognising a chain in $\boldsymbol{Ch_\blacktriangleleft}(\mathcal{G}_s^\epsilon(\partial(\mathcal{A}))^+)$ can be adapted to recognise S; we simply send down the left-most branch of the chain a copy of the automaton for the unary predicate $\boldsymbol{c_0 R_{r_\epsilon^*,q}}$ for each state q and require that the instance for the state p accepts where p is used as the decoration for the first element of the chain. In order to show that $\mathcal{G}_s^\epsilon(\mathcal{A})$ is isophilic, it thus suffices to show that the graph $\boldsymbol{Ch_\blacktriangleleft}(\mathcal{G}_s^\epsilon(\partial(\mathcal{A}))^+)$, together with the relations for stack operations on the 2-PDA stacks encoded as chains, is isophilic.

We appeal to Lemma 6.41. Recall that we are considering the slow version of a 2-PDA and so we are not treating any pop_2 or $push_2$ operations as ϵ-transitions. First observe that order-1 operations only affect the top stack—*i.e.* the final element of the stack in the chain representation thereof. We can therefore represent these by flat-isophilic relations of the form $R_{q,\epsilon^*a,q'}$ for each $a \in \Sigma$ relating stacks s and s' when $(q,s)\boldsymbol{r}_{\epsilon^*a}(q',s')$ and assigning these the decoration (q,q'). The relation $R_{q,\epsilon^*a,q'}{}_\blacktriangleleft$ on chains then does the job.

For a pop_2 we just need to discard the final element of the chain. We can have a relation $R_{q,a\downarrow,q'}$ that relates a stack s to s' just in case a pop_2 could be performed by $\mathcal{A}$ via an edge a when in control-state q with top stack symbol $top_1(s)$ and transitioning into control-state q' (stack s' is ignored). We can then represent pop_2 operations by relations of the form $R_{q,a\downarrow,q'\,pop}$, assigning $\mathcal{Q}_{pop}(R_{q,a\downarrow,q'}) := (q,q')$.

For $push_2$ we can have a relation $R_{q,a\uparrow,q'}$ that relates a stack s to s' just in case a $push_2$ could be performed by $\mathcal{A}$ via an edge a when in control-state q with top stack symbol $top_1(s)$ and transitioning into control-state q' *and additionally* $s = s'$. (Equality is flat-isophilic so this is possible). We can then represent pop_2 operations by relations of the form $R_{q,a\uparrow,q'\,push}$, assigning $\mathcal{Q}_{push}(R_{q,a\uparrow,q'}) := (q,q')$.

We thus see that a slow $\mathcal{G}_s^\epsilon(\mathcal{A})$ is isophilic. We could obtain the result for full ϵ-closure by appealing to Corollary 6.52, but instead we do so using bounces. This enables us to recreate the proof for 3-CPDA in a *completely* analogous manner (except we deal with isophilic chains instead of flat-isophilic chains).

We make use of the fact that ϵ^*a-reachability in a 2-PDA $\mathcal{A}$ can be rep-

resented by a bounce of $\mathcal{A}^{\uparrow}$—this fact was stated as Lemma 5.7. Moreover, Theorem 6.46 tells us that adding a bounce in the more abstract isophilic sense of the word preserves isophilicity. We can represent a single $push_2$ as part of an ϵ-climb by $\boldsymbol{R_{q,r_p r_\epsilon^*,q'}}_{push}$. For a single pop_2 as part of the ϵ-fall we can just take the relation ${R_{q,r_\epsilon^*,q'}}_{pop}$ that relates s to s' just in case there is an r_ϵ^*-path in $\partial(\mathcal{A})$ from (q,s) to (p,t) where there is also a single ϵ-transition in $\mathcal{A}$ performing a pop_2 from a configuration with control-state p and top_1 element $top_1(t)$ (with s' being ignored).

This witnesses the fact that $\mathcal{G}_s^{\epsilon}(\mathcal{A})$ is isophilic.

Armed with this fact we can see that $\mathcal{G}_s^{\epsilon}(\mathcal{A}')$ is tree-isophilic for a 3-PDA $\mathcal{A}'$, since its derivative is a 2-PDA and hence has an isophilic stack-graph. The argument proceeds in *exactly* the same way as going from 1-PDA to 2-PDA except that we use the analogous results for isophilic chains (yielding tree-isophilic structures) instead of flat-isophilic chains (which yield isophilic structures). □

In order to handle 2-links we need to record them without explicit representation. We could almost use the $\mathsf{lum}(\mathcal{A})$ from Chapter 5, but this is a little top heavy and complicates matters; at order-2 we can be much simpler. It needs to be the case that *collapse* on an element in a stack will discard all of the top-most stacks of which the top_2 stack is a prefix. This can be achieved by having two colours **red** and **blue** annotating atomic elements that alternate every time we have a link to a different stack. We also have an annotation 1 that indicates a 1-link (or equivalently no link) is attached. At the top of each 1-stack we indicate the colour that any new links should use. We write $\mathsf{Trail}(\mathcal{A})$ to denote this modified automaton.

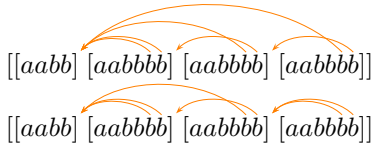
Definition 7.13. Let $\mathcal{A}$ be a 2-CPDA with stack-alphabet Γ. The 2-CPDA $\mathsf{Trail}(\mathcal{A})$ has stack-alphabet $\Gamma \times \{1, \mathsf{red}, \mathsf{blue}\} \times \mathbb{B} \cup \{ \mathsf{red}, \mathsf{blue} \} \times \mathbb{B}^{\perp}$. The automaton $\mathsf{Trail}(\mathcal{A})$ begins in '**red**-mode'.

The following modifications are made. Where $\mathcal{A}$ would have performed $push_1^{a,1}$, $\mathsf{Trail}(\mathcal{A})$ performs $push_1^{(a,1,\perp),1}$. Where the original would have performed a $push_1^{a,2}$, it performs $push_1^{(a,X,b),2}$ when in X-mode, where b is $\mathbf{t}$ if this is the first fresh link in the stack and $\mathbf{f}$ otherwise. Prior to performing $push_2$ it first pushes (X,b) onto the stack (which is immediately popped off the copy) where X is its mode and $b = \mathbf{t}$ iff the 1-stack being copied contains a fresh link. (Note that with these annotations together with the marking of the 'lowest' fresh link in a 1-stack the presence of fresh links is easy for the automaton to track.).

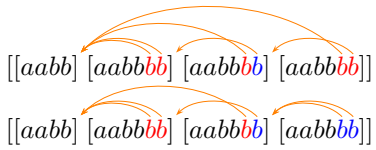
The mode is changed from X-mode (either **red**-mode or **blue**-mode) to the other colour Y just in case one of the following occurs:

- The automaton performs a pop_1 operation discarding a 2-link that is *not* fresh in the current 1-stack and that is the same colour as X.
- The automaton performs a pop_2 or a *collapse* operation in which case it adopts the mode specified by the element on top of the stack.

Example 7.14. Suppose that:



are stacks of $\mathcal{A}$. Then ignoring the additional Boolean values on the stack required for 'tracking the existence of fresh links', the corresponding stacks of $\mathsf{Trail}(\mathcal{A})$ are:



Note how colouring indicates that the top b's in each of the top three stacks source pointers with different targets.

The significance of colouring is given by the following Lemma:

Lemma 7.15. *1.* $\mathcal{G}^{\epsilon}(\mathcal{A}) \cong \mathcal{G}^{\epsilon}(\mathsf{Trail}(\mathcal{A}))$

2. *If s and s' are two stacks belonging to reachable configurations of* $\mathsf{Trail}(\mathcal{A})$ *such that* $\mathsf{stripln}(s) = \mathsf{stripln}(s')$*, then* $s = s'$.

3. *If* $\mathsf{stripln}(s) = [s_1\ s_2\ \cdots\ s_r\ s_{r+1}\ \cdots\ s_m]$ *is a stack in a reachable configuration of* $\mathsf{Trail}(\mathcal{A})$*, then if* $top_1(s)$ *has a 2-link,* $\mathsf{stripln}(collapse(s)) = [s_1\ s_2\ \cdots\ s_r]$ *where* $s_m \sqsubseteq_1 s_i$ *for every* $r+1 \leq i \leq m$ *but* $s_m \not\sqsubseteq_1 s_r$.

Proof. 1. The isomorphism comes from the fact that $\mathsf{Trail}(\mathcal{A})$ mimics the same operations as $\mathcal{A}$ and colours for new elements are deterministically determined from the stack content in any given configuration. The 'mode' and 'freshness tracking' components of the control-state (and Boolean annotations on the stack for tracking freshness) are also deterministically determined from the stack content in any given configuration.

2. Suppose for contradiction that the statement is false. We can then pick the smallest h such that there exist s and s' with $\mathsf{stripln}(s) = \mathsf{stripln}(s')$ and $|s| = |s'| = h$ but $s \neq s'$. It must thus be the case that there are instances of an atomic element γ in the same position in s and s' such that $\boldsymbol{l_o}(a) = \boldsymbol{l_o}(a') = 2$ but $\boldsymbol{l_a}(a) \neq \boldsymbol{l_a}(a')$. Indeed these must be the respective top elements of s and s' or else we would contradict minimality

of h. But this in turn means that one of them must have been created afresh in the top 1-stack and the other must have been copied from the 1-stack below. W.l.o.g. suppose that a was copied from the 1-stack below and that a' was created afresh. In order to create a in s it would have been necessary to pop_1 the copy of a' resulting from $push_2$ on the stack below; no other links would be pop_1'ed or else we would violate the minimality of h. But by assumption a' would not have been fresh in the top_2-stack and so the mode would have changed to be the opposite colour of a' at the point when a is created.

3. Ignoring the $\mathsf{stripln}(\_)$ the statement holds since the target of a 2-link in a 2-CPDA is completely determined by the 1-stack in which it was first created and moreover the atomic elements below a link cannot have been changed (since then the link would have been discarded). The statement must thus hold by item 2.

□

Now we can extend Lemma 7.12 to handle an order-2 collapse to give an analogous result for both 2-CPDA and 3_2-CPDA. When we stick with slow 3_2-CPDA stack graphs, we are able to see that the structures are *compact* nondisophilic.

Lemma 7.16. *Let $\mathcal{A}$ be a 2-CPDA. Then $\mathcal{G}_s^\epsilon(\mathcal{A})$ is dendrisophilic. Let $\mathcal{A}'$ be a 3_2-CPDA, then $\mathcal{G}_s^\epsilon(\mathcal{A})$ is tree-nondisophilic. If for the latter we consider the slow graph only, then it is* compact *nondisophilic.*

Proof. The proof proceeds in exactly the same way as for Lemma 7.12. For 2-CPDA we just need to handle collapse. We begin by working with $\mathsf{Trail}(\mathcal{A})$ (whose ϵ-closure is isomorphic to that of $\mathcal{A}$) instead of $\mathcal{A}$. Item two of Lemma 7.15 ensures that equality of chains from the derivative implies equality of the stacks being represented (in the absence of links). Due to the third item of Lemma 7.15 this means that we can extend the proof to work with *collapse* by recognising such transitions by predicates of the form $(P_1, P_2)_{panic}$ where P_1 checks that the top element of the starting stack is such that the transition function permits *collapse* and P_2 is the universal predicate, which should be satisfied by all stacks. The use of *panic* means that we obtain a *dendrisophilic* structure.

Whilst we could extend the abstract notion of bounce to include *panic* to go from slow graphs to full ϵ-closure, we instead find it easier to appeal to Theorem 6.52. We feel that we have already made the point about the similarity in construction with each level in Lemma 7.12.

Since we do not make use of 3-links in a 3_2-CPDA, however, we can obtain the nondisophilicity of $\mathcal{G}_s^\epsilon(\mathcal{A}')$ in exactly the same way as with Lemma 7.12,

this time using dendrisophilic chains in the derivative. Bouncing works for ϵ-closure since no analogue of *panic* is required (nor indeed has been defined for nondisophilic structures) at the third level.

Considering a slow 3_2-CPDA graph gives us a 2-compact nondisophilic structure: Since only the two right-most elements of the dendrisophilic chain need be compared in order to establish whether a pop_3 or $push_3$ relation holds (under the assumption that all other branches of the two chains are the same), all transition relations must be compact. We want to use Lemma 6.34 and so must supply a suitable A_m for each $m \in \mathbb{N}$. This requires a small modification with the encoding of stacks. At the tip of ith element s_i of the chain we must record a set S_i for each $1 \leq i \leq k$ where the chain has length k. The set S_i contains precisely those control-states belonging to reachable configurations associated with the stack encoded by the first i elements of the chain. Note that $S_{i+1} = \{\ q' \in Q\ :\ (q, s_i)\boldsymbol{r}r_p(r_\epsilon + \Sigma)^*\ \}$ and so each step in the chain remains dendrisophilic and so the chain as a whole is still tree-nondisophilic. Moreover, since each S_i decoration is uniquely determined by the chain, the replacement chains are in bijective correspondence to the originals.

So given a modified chain encoding of a reachable stack s and a chain encoding of an arbitrary stack s' such that all but the right-most m branches are shared by s and s', we can now define an A_m, as needed by Lemma 6.34, acting on the right-most $(m+1)$ branches of $\bigotimes \langle s, s' \rangle$ that accepts just in case s' is also a reachable stack. This automaton has a look at the S decoration on the last nested-word that the chains s and s' have in common. Since this S also belongs to a substack of s, which is assumed to be reachable, making the S correct, it provides the necessary data to continue the chain for the remaining m branches of s', which can be done by an $(m+1)$-compact automaton A_m. □

We can go from a representation of $\mathcal{G}^\epsilon_s(\mathcal{A})$ to a representation of $\mathcal{G}^\epsilon(\mathcal{A})$ by simply adding to the right-most leaf of a representation of the stack a control-state of $\mathcal{A}$. Each element of the domain then represents a *configuration* rather than just a stack. We can then construct a representation of an a-labelled edge R_a by constructing an automaton recognising the same binary convolutions as $R_{q,a,q'}$ in the stack-graph whenever the first configuration has right-most-leaf labelled q and the second has right-most-leaf labelled q'. Thus the lemmas above give us:

Lemma 7.17. *Let $\mathcal{A}$ be a 2-PDA. Then $\mathcal{G}^\epsilon(\mathcal{A})$ is isophilic. Let $\mathcal{A}$ be a 2-CPDA. Then $\mathcal{G}^\epsilon(\mathcal{A})$ is dendrisophilic. Let $\mathcal{A}$ be a 3-PDA. Then $\mathcal{G}^\epsilon(\mathcal{A})$ is tree-isophilic. Let $\mathcal{A}$ be a 3_2-CPDA. Then $\mathcal{G}^\epsilon(\mathcal{A})$ is nondisophilic. If we consider a slow 3_2-CPDA, then $\mathcal{G}^\epsilon(\mathcal{A})$ is* compact *nondisophilic.*

We also need to say something about symmetric 2-CPDA graphs:

Lemma 7.18. *Let $\mathcal{A}$ be a 2-CPDA. Then any symmetric closure based on $\mathcal{G}^{\epsilon}(\mathcal{A})$ is symmetric dendrisophilic.*

Proof. Symmetric closure is just a special case of closure under inverse rational maps and so this is a consequence of: Theorem 6.51 and the fact that $\mathcal{G}^{\epsilon}(\mathcal{A})$ must be dendrisophilic, in particular symmetric-dendriosphilic. □

7.4 Concluding Remarks

We can summarise the consequences of the principal results of this chapter with following Theorem.

Theorem 7.19. $\boldsymbol{Iso_2} = \boldsymbol{\mathcal{RW}_{rr}} = \mathbf{Pd}_{\epsilon 2} \subsetneq \mathbf{Pd}_{\epsilon 2}^{\mathcal{C}} = \boldsymbol{\mathcal{RW}_{sr}} = \boldsymbol{dIso_2} \subsetneq \boldsymbol{\mathcal{RW}_{ss}} = \mathbf{Pd}_{\epsilon 2}^{\mathcal{S}} \subsetneq \boldsymbol{nIso_2} = \boldsymbol{\pi}\mathbf{Pd}_{\epsilon 2} = \boldsymbol{\pi}\mathbf{Pd}_{\epsilon 2}^{\mathcal{C}}$ *and* $\mathbf{Pd}_{\epsilon 3} \subseteq \boldsymbol{Iso_3}$.

Proof. $\boldsymbol{Iso_2} = \boldsymbol{\mathcal{RW}_{rr}} = \mathbf{Pd}_{\epsilon 2}$ and $\mathbf{Pd}_{\epsilon 2}^{\mathcal{C}} = \boldsymbol{\mathcal{RW}_{sr}} = \boldsymbol{dIso_2}$ are obtained by combining the inclusion lemmas for each direction from this chapter, and $\mathbf{Pd}_{\epsilon 3} \subseteq \boldsymbol{Iso_3}$ is also stated as an inclusion result in a lemma above. The equality $\boldsymbol{nIso_2} = \boldsymbol{\pi}\mathbf{Pd}_{\epsilon 2} = \boldsymbol{\pi}\mathbf{Pd}_{\epsilon 2}^{\mathcal{C}}$ is a result of Theorems 6.54 and 6.55.

The strictness of the inclusion of $\mathbf{Pd}_{\epsilon 2}$ in $\mathbf{Pd}_{\epsilon 2}^{\mathcal{C}}$ was estabilished by Hague *et al.* [40] when they observed that there exists a 2-CPDA whose configuration graph has undecidable MSO theory whilst all members of the Caucal hierarchy (including $\mathbf{Pd}_{\epsilon 2}$) have decidable MSO theories.

The strictness of the inclusion of $\mathbf{Pd}_{\epsilon 2}^{\mathcal{C}}$ in $\mathbf{Pd}_{\epsilon 2}^{\mathcal{S}}$ is obtained by the following observations. First note that the translation from a (symmetric) (dendr)isophilic structure to the corresponding prefix rewrite system creates rules of the form $\mathcal{L} \longrightarrow \mathcal{L}'$ where the common prefix of elements of $\mathcal{L} \cup \mathcal{L}'$ has length precisely one. The circle of equivalences thus tells us that every prefix rewrite system can be written in a form $\mathcal{L} \longrightarrow \mathcal{L}'$ where the common prefix of the elements of $\mathcal{L} \cup \mathcal{L}'$ has precisely length one. Thus for each edge label a, we can have just a single rewrite rule for each of these common prefixes of length one.

Suppose for contradiction that $\boldsymbol{\mathcal{RW}_{ss}} = \boldsymbol{\mathcal{RW}_{sr}}$. Then given a sum-sum rewrite system with rules of the form $a : \mathcal{L} \longrightarrow \mathcal{L}'$ we can add reverse rules of the form $\overline{a} : \mathcal{L}' \longrightarrow \mathcal{L}$ and still retain a sum-sum system. By our assumption there must be an equivalent sum-rat rewrite system. Moreover, the rules of this equivalent sum-rat rewrite system can be expressed in the special form in the paragraph above. Due to this form, it must be that for each rule of the form $a : \mathcal{M} \longrightarrow \mathcal{M}'$ in the new system, the sum-rat rule in the reverse direction of the special form is $\overline{a} : \mathcal{M}' \longrightarrow \mathcal{M}$. Since both of these are sum-rat rules it must be that both are actually rat-rat rules. But then $\boldsymbol{\mathcal{RW}_{ss}} = \boldsymbol{\mathcal{RW}_{rr}}$ and so in particular $\mathbf{Pd}_{\epsilon 2}^{\mathcal{C}} = \mathbf{Pd}_{\epsilon 2}$, which contradicts the separation of 2-PDA and 2-CPDA graphs.

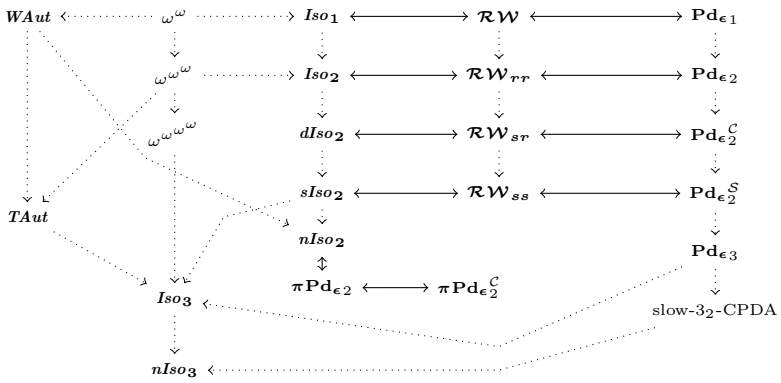


Figure 7.3: Classes of graphs. Double headed arrow indicates equality, dots indicate strict inclusions.

The strictness of the inclusion of $\mathbf{Pd}_{\epsilon 2}^{\mathcal{S}}$ in $\boldsymbol{nIso_2} = \boldsymbol{\pi}\mathbf{Pd}_{\epsilon 2}$ comes from the fact that $\boldsymbol{nIso_2}$ is not closed under reachability (whilst $\mathbf{Pd}_{\epsilon 2}^{\mathcal{S}}$ is, as we have seen, closed under reachability). This is because the nondisophilic structures all have decidable first-order theories but we can represent the transition graph (without reachability) of a Turing Machine as such a graph due to the fact that such graphs are word-automatic and by Theorem 6.48 all word-automatic graphs are nondisophilic. □

The known strict inclusions and equivalences between the various classes we have considered are illustrated in Figure 7.3. In particular note that the strictness of the inclusion of $\mathbf{Pd}_{\epsilon 2}^{\mathcal{C}}$ in $\mathbf{Pd}_{\epsilon 2}^{\mathcal{S}}$ can be viewed as a formal statement of the fact that *collapse* is inherently asymmetric. This confirms what has been intuitively clear—in particular the impossibility of symmetric CPDA generating the same graphs as ordinary CPDA in the way that Carayol *et al.* were able able to design symmetric PDA [25, 27].

The coincidence of $\boldsymbol{nIso_2}$, $\boldsymbol{\pi}\mathbf{Pd}_{\epsilon 2}$ and $\boldsymbol{\pi}\mathbf{Pd}_{\epsilon 2}^{\mathcal{C}}$ indicates that a certain form of non-determinism, namely that introduced by projection, removes the distinction between 2-PDA and 2-CPDA. This could therefore be viewed, in some respects, as the graph analogue of Aehlig *et al*'s result that introducing nondeterminism makes 2-CPDA and 2-PDA equivalent for word-languages [47].

Finally we can state our new decidability result. The fact that slow 3_2-CPDA graphs are compact tree-nondisophilic means that they have decidable first-order theory by Theorem 6.33. As required by Chapter 5 we can also see that the ϵ-closure of 2-CPDA graphs have decidable $\boldsymbol{FO(TC[\Delta_0])}$ theories. This follows from Lemma 6.23. Δ_0 formulae with only two variables must be formed from Boolean combinations of binary relation symbols and unary predicates, defining a set of convolutions consisting of trees with at most two

branches. Since spine nested automata are closed under Boolean operations when we restrict the universe to trees with at most two branches (Lemma 6.23), this means that the binary relation defined by a Δ_0 formula with only two variables over dendrisophilic predicates must itself be dendrisophilic. The transitive closure must also be dendrisophilic since dendrisophilic structures are closed under reachability (Theorem 6.39).

Theorem 7.20. ***FO*** *is decidable on $\mathcal{G}(\mathcal{A})$ for every 3_2-CPDA $\mathcal{A}$ (and indeed is decidable on $\mathcal{G}^\epsilon(\mathcal{A})$ when $\mathcal{A}$ is slow).*

FO**(*****TC*****[*****Δ*****$_0$])** *is decidable on all dendrisophilic structures and in particular is decidable on $\mathcal{G}^\epsilon(\mathcal{A}')$ where $\mathcal{A}'$ is a 2-CPDA.*

Chapter 8

Epilogue and Further Directions

Having begun with an analysis of the semantic significance of links in CPDA we proceeded to investigate the decidability of first-order logic on CPDA graphs, which turns out to be highly sensitive to which links are allowed. Some very strong undecidability results were established which show that even Σ_1 sentences are undecidable for most parts of the hierarchy. We introduced a new technique for reasoning about stacks that employed monotonic automata and the notion of derivative. We obtained some limited undecidability results for Σ_1 sentences, which when combined with Kartzow's work and our own in the final chapter, leaves open only the decidability of Σ_1-sentences on $n_{n,(n-1)}$ graphs (both with and without ϵ-closure) when $n > 3$. This would of course be a nice gap to fill in, and we hope that the machinery we have developed to tackle the other Σ_1 decidability problems may yet be adaptable to this task.

We have also introduced several new classes of structures based on notions of automaticity arising from nested-words and nested-trees. Some of these classes have very nice closure properties and indeed we have shown the isophilic structures to coincide with the second level of the Caucal Hierarchy and the dendrisophilic graphs to coincide with the ϵ-closures of 2-CPDA graphs. Whilst CPDA were not designed to be interesting graph generators, this shows that the 2-CPDA graphs admit a non-trivial alternative representation and hence may well be worth further study.

Indeed further work on the various types of isophilic structures would be in order, including the quest for some other examples of interesting structures residing within these classes. There is plenty of scope for improving our understanding of them in a more general setting as well—for example it would be nice to see whether $\omega^{\omega^{\omega^{\omega}}}$ is an upper-bound on the well-orderings that are tree-nondisophilic as well as a lower-bound. An approach to this question that might yield further results concerning linear orderings would relate to Braud and Carayol's analysis of linear orderings in the Caucal hierarchy [17]. They provide a precise characterisation of linear orders occurring at each level

and in particular an upper bound of $\omega^{\omega^{\omega^{\omega}}}$ at level 3. Since tree-nondisophilic structures include graphs outside of the third level of the Caucal hierarchy, we cannot directly infer any bound from this result. However, another part of *op cit.* demonstrates that the ordinals in the $(n+1)$th level of the hierarchy are precisely those induced by the lexicographic ordering on leaves of trees in the nth level. The precise coincidence of isophilic structures and 2-PDA graphs allows us to conclude that it is precisely the ordinals less than $\omega^{\omega^{\omega^{\omega}}}$ which occur at the frontiers of isophilic trees. Perhaps a correspondence between the frontier ordinals of isophilic trees and the tree-nondisophilic (or more weakly tree-isophilic) structures could be established.

Blumensath established that the first level of the Caucal hierarchy consists of all graphs MSO definable in the infinite binary tree Δ_2 [12]. Carayol *et al.* extended this result [27, 23] to show that the nth level consists of those graphs MSO definable in Δ_2^n, which can be viewed as the canonical structure resulting from Δ_2 via an n-fold iteration of the 'treegraph' operation. This is related to Caucal's generalisation of prefix rewriting in terms of finding canonical sequences of stack operations witnessing the nodes and transitions in a pushdown graph. Our own notion of rat-rat prefix rewriting could yield a similar MSO definability result at order-2 in a canonical structure $\Delta_4^{\curvearrowleft}$, the infinite 4-ary tree with well-nested back-edges. The idea is that the 4-ary tree would allow for 0 and 1 branching as in Δ_2 but with extra annotations specifying pointers whenever they are permitted by well-nesting—*i.e.* $\mathring{0}$,$\dot{0}$, $\mathring{1}$ and $\dot{1}$. An MSO with a matching predicate to handle pointers [6] might then be able to capture our rewrite rules. Indeed if a μ-calculus based logic with matching is employed, such as a branching version of CARET [5], it might be possible to derive the preservation of μ-calculus decidability, which would be of interest in the collapsible pushdown hierarchy could be captured in this way.

In this setting, order-2 (C)PDA graphs would be obtained in a manner analogous to 1-PDA graphs in the non-nested setting. An adaption of the treegraph operation to introduce nested word structure in the second dimension created by treegraphing might be a way of generalising Carayol's work on the Caucal hierarchy to the hierarchy of CPDA graphs. Indeed this would be a helpful exercise in understanding the connections with his work; in particular whilst Carayol begins with prefix rewriting capturing order-1 and then generalises using 'canonical operation sequences', we would begin with prefix rewriting capturing order-2 and then generalise after that using something similar to canonical operation sequences.

As an alternative to the treegraph transformation to generate levels 3 and above, it would be worth investigating exstensions of $\Delta_4^{\curvearrowleft}$ with pointers allowing different amounts of 'bad-nesting', with the extent of bad-nesting corresponding to the level in the hierarchy.

Further understanding of the structures could arise from investigating pumping on paths in the graphs. Kartzow's use of a tree automatic presentation of 2-CPDA graphs to obtain a 'pumping lemma' for them [49] is intriguing and it would be interesting to see whether such results could be extended to 3_2-CPDA (and indeed 3-PDA) using a pumping technique on nested-trees.

The nested-word representation of 2-CPDA stacks also suggests an alternative model of automaton where stacks are nested-words and higher-order stacks are nested-words of stacks, rather than just words of stacks with unstructured pointers. In many respects CPDA are ugly; the links lack structure, there exist stacks that cannot be constructed from the empty stack and unlike higher-order PDA they lack a true inductive structure as shown by the possibility for 'dangling links'. Stacks based on nested-words might offer a more elegant alternative—we already know from this dissertation that such an automaton could be defined at order-1 that is equi-expressive with 2-CPDA. Unfortunately it seems that an order-2 nested-word stack of nested-words would already come very close to being able to simulate a two counter machine. Nevertheless, the hierarchy of graphs so generated may admit other favourable properties; after all the transition graph of a two counter machine is word-automatic. For example, one might look at developing a notion of automaticity based on higher-order stacks of this nature that synchronises at the end of each component stack.

Another possibility for further work would be to exploit monotonicity and the derivative construction. For example, it would probably be possible to obtain a proof of Parikh's Theorem using a monotonic 1-PDA and it would be interesting to see whether any analogues could be extracted at higher-orders. In choosing the terminlogy for 'derivative' we partly had in mind the idea of 'going down an order'. Hopkins and Kozen have a proof of Parikh's theorem by 'going down an order' from polynomials over a commutative Kleene algebra using formal differentiation [43].

Of course we must not forget that CPDA originally arose from the study of higher-order recursion schemes, which in turn have applications in modelling higher-order functional programs. To the author's knowledge, there is currently no study in the literature comparing CPDA graphs to the *graphs* (rather than trees) generated by recursion schemes, with nodes comprising of applicative terms and edges generated via (non-deterministic) head reduction rules. It seems likely that the standard compilation of an n-(C)PDA to a (safe) recursion scheme would respect graphs as well as trees, but the converse is less apparent. Reasoning about graphs of recursion schemes using first-order logic might have some practical applications, for example when considering the structure of memory during the evaluation of a term. Non-deterministic edges could be introduced to extract particular subterms by means of a reduction along an edge that spits out a particular argument. So it might be possible to make a useful

first-order assertion (at least when ϵ-closure a.k.a. reachability is available) stating, for example, that some particular subterm keeps reappearing during evaluation.

However, the author in all honesty is sceptical about the utility of such a venture; in any case we have not come anywhere near offering practical algorithms, even for the decidable fragments of the decision problem. We feel it is best to view the work as furthering our general mathematical understanding of structures that are associated with devices with practical application, rather than offering practical utility in themselves! We hope that not only our results but some of the machinery that we have developed to obtain them will prove interesting to the reader and invite further lines of enquiry.

Bibliography

[1] K Aehlig, J G de Miranda, and C.-H. L Ong. Safety is not a restriction at level 2 for string languages. Technical Report RR-04-23, Oxford University Computing Laboratory, October 2004.

[2] K Aehlig, J G de Miranda, and C.-H L Ong. The Monadic Second Order Theory of Trees Given by Arbitrary Level-Two Recursion Schemes is Decidable. In *Proc. TLCA*, volume 3461, pages 39–54, Berlin/Heidelberg, 2005. Springer-Verlag.

[3] Alfred V Aho. Indexed Grammars, An Extension of Context-Free Grammars. *J. ACM*, 15(4):647–671, October 1968.

[4] Rajeev Alur, Swarat Chaudhuri, and P Madhusudan. Languages of nested trees. In *CAV*, pages 329–342. Springer, 2006.

[5] Rajeev Alur and K Etessami. A temporal logic of nested calls and returns. In *TACAS*, volume 2988, pages 467–481, 2004.

[6] Rajeev Alur and P Madhusudan. Visibly Pushdown Languages. In *STOC*, pages 202–211. ACM, 2004.

[7] Rajeev Alur and P. Madhusudan. Adding nesting structure to words. *Journal of the ACM (JACM)*, 56(3):1–43, 2009.

[8] Marcelo Arenas, P. Barceló, and Leonid Libkin. Regular languages of nested words: Fixed points, automata, and synchronization. In *ICALP*, pages 888–900. Springer, 2007.

[9] William Blum. Encoding of a safe order-n recursion scheme into a n-PDA. 2009.

[10] William Blum. *The Safe Lambda Calculus*. PhD thesis, Oxford University, February 2009.

[11] William Blum and C.-H L Ong. The Safe Lambda Calculus. In *Proceedings of Typed Lambda Calculus and its Applications*, Lecture Notes in Computer Science. Springer Verlag, 2007.

[12] A. Blumensath. Prefix-recognisable graphs and monadic second-order logic, 2001.

[13] A. Blumensath and E. Gradel. Automatic structures. In *LICS*, pages 51–62. IEEE Comput. Soc, 2000.

[14] Achim Blumensath and Erich Grädel. Finite Presentations of Infinite Structures: Automata and Interpretations. *Theory of Computing Systems*, 37(6):641–674, September 2004.

[15] Ahmed Bouajjani and Meyer Antoine. Symbolic reachability analysis of higher-order context-free processes. In *FSTTCS*, number 1, 2004.

[16] J Bradfield and C P Stirling. *Modal logics and mu-calculi: an introduction*, pages 293–332. Elsevier, North-Holland, 2001.

[17] Laurent Braud and Arnaud Carayol. Linear Orders in the Pushdown Hierarchy. In *ICALP*, pages 88–99, 2010.

[18] C H Broadbent. A Proof of Blum's Conjecture. 2009.

[19] C.H. Broadbent, Arnaud Carayol, C.-H.L. Ong, and Olivier Serre. Recursion Schemes and Logical Reflection. In *LICS*, pages 120—-129. IEEE Computer Society, 2010.

[20] C.H. Broadbent and C.-H.L. Ong. On Global Model-Checking Trees Generated by Higher-Order Recursion Schemes. In *FoSSaCS*, pages 107—-121. Springer, 2009.

[21] T Cachat. Uniform solution of parity games on prefix-recognizable graphs. In *Proc. VISS*, volume 68 of *ENTCS*. Elsevier, 2002.

[22] Thierry Cachat. Higher order pushdown automata, the Caucal hierarchy of graphs and parity games. In *Proceedings of ICALP*, volume 2719 of *Lecture Notes in Computer Science*, pages 556–569. Springer, 2003.

[23] A. Carayol. Regular Sets of Higher-Order Pushdown Stacks. In *Proc. MFCS*, volume 3618 of *LNCS*, pages 168–179. Springer, 2005.

[24] A Carayol, M Hague, A Meyer, C.-H. L Ong, and O Serre. Winning Regions of Higher-Order Pushdown Games. In *Proc. LICS*, pages 193–204. IEEE Computer Society, 2008.

[25] A. Carayol and M. Slaats. Positional Strategies for Higher-Order Pushdown Parity Games. In *Proc. MFCS*, volume 5162 of *LNCS*, pages 217–228. Springer, 2008.

[26] Arnaud Carayol and Olivier Serre. Higher-order recursion schemes and their automata models 1 Introduction 2 Preliminaries. 2010.

[27] Arnaud Carayol and S. Wöhrle. The Caucal hierarchy of infinite graphs in terms of logic and higher-order pushdown automata. In *FSTTCS*, pages 112–123. Springer, 2003.

[28] Didier Caucal. On infinite transition graphs having a decidable monadic theory. In *ICALP*, volume 6317, pages 194–205. Springer, 1996.

[29] Didier Caucal. On Infinite Terms having a Decidable Monadic Theory. In *Mathematical Foundations of Computer Science, 2002 : 27th Internation Symposium*, volume 24, pages 165–176. Springer, 2002.

[30] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement. In E Emerson and Aravinda Sistla, editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer Berlin / Heidelberg, 2000.

[31] P Crégut. An abstract machine for Lambda-terms normalization. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, LFP '90, pages 333–340, New York, NY, USA, 1990. ACM.

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

[33] W Damm and a Goerdt. An automata-theoretical characterization of the OI-hierarchy. *Information and Control*, 71(1-2):1–32, October 1986.

[34] V Danos and L Regnier. Head Linear Reduction, 2004.

[35] J G de Miranda. *Structures generated by higher-order grammars and the safety constraint*. PhD thesis, 2006.

[36] Christian Delhommé. Automaticité des ordinaux et des graphes homogčnes. *Comptes Rendus Mathematique*, 339(1):5–10, 2004.

[37] E A Emerson and C S Jutla. Tree automata, mu-calculus and determinacy. In *Proc. FOCS*, pages 368–377. IEEE computer society, 1991.

[38] Haim Gaifman. On Local and Non-Local Properties. In J Stern, editor, *Proceedings of the Herbrand Symposium*, volume 107 of *Studies in Logic and the Foundations of Mathematics*, pages 105–135. Elsevier, 1982.

[39] Robert H Gilman. A shrinking lemma for indexed languages. *Theoretical Computer Science*, 1–2(163):277—-281, 1996.

[40] M Hague, A S Murawski, C.-H L Ong, and O Serre. Collapsible Pushdown Automata and Recursion Schemes. In *LICS*. IEEE Computer Society, 2008.

[41] M Hague and C H L Ong. Symbolic Backwards-Reachability Analysis for Higher Order Pushdown Systems. In *FoSSaCS*. Springer, 2007.

[42] Takeshi Hayashi. On derivation trees of indexed grammars: an extension of the uvwxy-theorem. *Publications of the Research Institute for Mathematical Sciences*, 9(1):61–92, 1973.

[43] M W Hopkins and D C Kozen. Parikh's theorem in commutative Kleene algebra, 1999.

[44] M Hyland and C.-H L Ong. On full abstraction for {PCF}: {I, II and III}. *Information and computation*, 163(2):285–408, 2000.

[45] Neil Immerman. Languages which Capture Complexity Classes. In *STOC*, 1983.

[46] David Janin and Igor Walukiewicz. On the Expressive Completeness of the Propositional mu-Calculus with Respect to Monadic Second Order Logic. In *CONCUR '96: Proceedings of the 7th International Conference on Concurrency Theory*, pages 263–277, London, UK, 1996. Springer-Verlag.

[47] K. Aehlig, J. G. de Miranda, and C.-H. L. Ong. Safety is not a restriction at level 2 for string languages. In *Proc. FoSSaCS*, pages 490–501, 2005.

[48] Alexander Kartzow. Collapsible Pushdown Graphs of Level 2 are Tree-Automatic. In *STACS*, pages 501–512, 2010.

[49] Alexander Kartzow. A Pumping Lemma for Collapsible Pushdown Graphs of Level 2. In *CSL*, 2011.

[50] Bakhadyr Khoussainov and Anil Nerode. Automatic presentations of structures. In Daniel Leivant, editor, *Logic and Computational Complexity*, volume 960 of *Lecture Notes in Computer Science*, pages 367–392. Springer Berlin / Heidelberg, 1995.

[51] T Knapik, D Niwinski, and P Urzyczyn. Deciding Monadic Theories of Hyperalgebraic Trees. In *Proc. TLCA*, volume 2044 of *LNCS*, pages 253–267. Springer, 2001.

[52] T Knapik, D Niwinski, and P Urzyczyn. Higher-Order Pushdown Trees are Easy. In *Proc. FoSSaCS*, volume 2303 of *LNCS*, pages 205–222. Springer, 2002.

[53] T Knapik, D Niwinski, P Urzyczyn, and I Walukiewicz. Unsafe Grammars and Panic Automata. In *Proc. ALP*, volume 3580, pages 1450–1461, Berlin/Heidelberg, 2005. Springer-Verlag.

[54] Naoki Kobayashi. TRecS (Types for RECursion Schemes).

[55] Naoki Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *POPL*, volume 44, pages 416–428. ACM, 2009.

[56] Dexter Kozen. Results on the Propositional mu-Calculus. *Theoretical computer science*, 27(3):333——354, 1983.

[57] Martin Lester, Robin Neatherway, C.-H.L. Ong, and Steven Ramsay. THORS (Types for Higher-Order Recursion Schemes).

[58] A N Maslov. The Hierarchy of Indexed Languages of an Arbitrary Level. *Soviet Math. Dokl.*, 15:1170–1174, 1974.

[59] David E Muller and Paul E Schupp. The theory of ends, pushdown automata, and second-order logic. *Theoretical Computer Science*, 37:51–75, 1985.

[60] C.-H L Ong. On model-checking trees generated by higher-order recursion schemes. In *Proc. LICS*. IEEE Computer Society, 2006.

[61] C.-H.L. Ong and Steven J Ramsay. Verifying Higher-Order Functional Programs with Pattern-Matching Algebraic Data Types. In *POPL*, 2011.

[62] Pawel Parys. Collapse Operation Increases Expressive Power of Deterministic Higher Order Pushdown Automata. In *STACS*, 2011.

[63] E. Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52(4):264–268, April 1946.

[64] S. Rubin. *Automatic structures: richness and limitations*. PhD thesis, The University of Auckland, 2004.

[65] S Salvati and I Walukiewicz. Krivine machines and higher-order schemes. In *ICALP*, pages 162–173. Springer, 2011.

[66] O Serre. Note on winning positions on pushdown games with ω-regular conditions. *Information Processing Letters*, 85(6):285–291, 2003.

[67] C P Stirling. A game-theoretic approach to deciding higher-order matching. volume 4052 of *Lecture Notes in Computer Science*, pages 348–359. Springer, 2006.

[68] Colin Stirling. Decidability of Bisimulation Equivalence for Pushdown Processes Pushdown processes. (Unpublished). 2000.

[69] Igor Walukiewicz. Pushdown processes: Games and model checking. In *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 62–74, Berlin/Heidelberg, 1996. Springer.

Index