Structures Generated by Higher-Order Grammars and the Safety Constraint

Jolie G. de Miranda
Merton College

Trinity Term 2006

Submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy

Oxford University Computing Laboratory
Programming Research Group

To my mother and father.

Abstract

We study higher-order grammars as generators of potentially infinite structures as first introduced by Damm (1982). One specialisation of a higher-order grammar is the higher-order recursion scheme: a higher-order recursion scheme produces a single potentially infinite *term tree*. Recently, it was shown by Knapik, Niwiński and Urzyczyn (2001, 2002) that the term tree defined by a higher-order recursion scheme possesses a decidable monadic second-order (MSO) theory provided the recursion scheme satisfies the syntactic constraint of *safety*. This was a seminal result in the field of infinite-state verification.

In this thesis, we consider higher-order grammars in their full generality and ask (1) whether the safety constraint restricts the expressive power of higher-order grammars and (2) whether it is necessary for the MSO decidability of term trees defined by higher-order recursion schemes. Thus, we investigate "unsafety." Prior to the results in this thesis, there were no known results (to the author's knowledge) on unsafe grammars nor unsafe recursion schemes.

We present a unified framework in which to study higher-order grammars and recursion schemes, much in the style of Damm (1982). We consider what the safety restriction means and its equivalence with the restriction of derived types. We present a catalogue of the known results for safe grammars, and contrast this with the absence of results on unsafe grammars. Our key contributions concern level 2 of the hierarchy of OI higher-order grammars[1]; the first level where safety manifests itself. Our first result is an extension of Damm and Goerdt's result (1986) who showed that every level-n safe OI word grammar can be converted into a level-n pushdown automaton and vice versa. We show that every level-2 unsafe OI word grammar can be converted into a level-2 pushdown automaton. A corollary of this result is that every level-2 unsafe OI word grammar is equivalent to a level-2 safe OI word grammar. This result was presented at FoSSaCS 2005. We are unable to say whether this result extends to level-2 grammars where the alphabet is not monadic – but we conjecture that the answer is negative. Our second key result shows that at level 2, safety is not a requirement to guarantee MSO decidability: we show that every term tree defined by a level-2 (potentially unsafe) recursion scheme possesses a decidable MSO theory. This result was presented at TLCA 2005.

[1]Higher-order recursion schemes are implicitly OI.

Acknowledgements

First and foremost I would like to thank my supervisor, Luke Ong, without whom none of this would have been possible. Not only am I thankful to him for his academic insights and guidance that have helped make my time as a research student extremely enjoyable and productive, but I am also indebted to him for his kindness and belief in my abilities. It has been a privilege to work with him.

I thank my wonderful mother and father, to whom this thesis is dedicated, for their love, encouragement and tremendous support (both emotional and financial).

Thank you to Colin Stirling and Joel Ouaknine for being my viva examiners and making the experience an enjoyable one; their comments and corrections are very much appreciated. In the same vein, I would also like to thank Alexandru Baltag and Tom Melham who have acted as my examiners for departmental vivas.

I am also indebted to Klaus Aehlig for fruitful collaborations. Thank you also to Andrzej Murawski and Dan Ghica for numerous bouts of proof reading and interesting discussions over the years.

I am extremely grateful to my colleague and friend Damien Sereni not only for taking an interest in my work, but also for assuming the role of my right arm in Oxford and helping me to submit my thesis remotely! I have lost count of the number of cocktails I owe him.

I thank the attic dwellers of the Oxford University Computing Laboratory for providing a stimulating work environment yet at the same time ensuring I never worked too hard.

Last, but not least, I would like to thank the members of the FX Quant Team at the Royal Bank of Scotland for their support in helping me to finish writing up this thesis.

Contents

1 Introduction **1**
- 1.1 Infinite State Verification 1
- 1.2 Recent Developments 4
  - 1.2.1 Finitely-Representable Infinite Structures 4
  - 1.2.2 Finitely-Representable Infinite Structures with a Decidable Model-Checking Problem 5
- 1.3 Higher-order Grammars and the Safety Restriction 8
- 1.4 Contributions and Organisation of this Thesis 9

2 Preliminaries **11**
- 2.1 Types, Terms and Term Trees 12
  - 2.1.1 Simple Types and Applicative Terms 12
  - 2.1.2 Term Trees 13
  - 2.1.3 Σ-algebras 14
- 2.2 Higher-order Grammars and their Term Tree Languages 15
  - 2.2.1 Modes of Derivation 17
  - 2.2.2 A Special Case: Monadic Terminal Alphabet 19
- 2.3 Higher-order Recursion Schemes and their Infinite Trees 19
  - 2.3.1 Higher-order Recursion Schemes 19
  - 2.3.2 The Monadic Theory of the Infinite Tree Generated by a Higher-Order Recursion Scheme 21
- 2.4 The Safety Restriction for Higher-order Grammars 24
  - 2.4.1 The Knapik et al. Formulation 25
  - 2.4.2 The Safe λ-Calculus 25
- 2.5 Machine Characterisations of Higher-Order Grammars 31
  - 2.5.1 Higher-order Pushdown Automata 32
- 2.6 Higher-order Grammars Through the Ages 36
  - 2.6.1 Semantics of Programming Languages 37
  - 2.6.2 The Safety Restriction and the Restriction of “Derived Types” 41
  - 2.6.3 Higher-Order Pushdown Automata 46
  - 2.6.4 Verification 48

3 Expressibility **51**
- 3.1 Word-generating Grammars 53
  - 3.1.1 The Language of a Word Grammar 53
  - 3.1.2 Safe and Unsafe Word Grammars 54

3.2 Urzyczyn's Language . . . 54
3.2.1 Definition of U . . . 55
3.2.2 Unique Decomposition of U-words . . . 56
3.2.3 Constructing a 2PDA that Accepts U . . . 58
3.3 Machine Characterisations of Word Grammars . . . 58
3.3.1 Definition of Pointer Machine . . . 58
3.3.2 Useful Properties . . . 61
3.3.3 Proof that Pointer Machines Implement Word Grammars . . . 63
3.4 Equivalence at Level 2 . . . 66
3.4.1 Simulating 2PMs by 2PDALs . . . 66
3.4.2 Simulating 2PDALs by Non-deterministic 2PDAs . . . 77
3.5 Investigating Non-Determinism . . . 86
3.5.1 Deterministic Higher-order Grammars and Pushdown Automata . . . 87
3.5.2 Equivalence between Deterministic nPDAs and Safe Deterministic Level-n Word Grammars . . . 89
3.5.3 Unsafe Deterministic Grammars . . . 90
3.6 Related Work and Concluding Remarks . . . 94
3.6.1 Related Work . . . 94
3.6.2 Concluding Remarks and Future Work . . . 96

4 Decidability **99**
4.1 MSO Decidability of Level-2 Term Trees . . . 100
4.1.1 Preliminaries . . . 100
4.1.2 Botanical Gardens . . . 104
4.1.3 Definition of the Simulating Automaton . . . 119
4.1.4 Direction 1: From Annotated Canonical Traversal Trees to Run Trees 125
4.1.5 Direction 2: From Run Trees to Annotated Canonical Traversal Trees 130
4.1.6 Putting it all together . . . 132
4.2 Related Work and Concluding Remarks . . . 132
4.2.1 An Independent Proof of the Level-2 Result . . . 132
4.2.2 Extensions to Higher Levels . . . 134
4.2.3 Concluding Remarks . . . 136

5 Conclusion **137**
5.1 Summary of Contributions . . . 137
5.2 Future Work . . . 138

Chapter 1

Introduction

1.1 Infinite State Verification

Higher-order recursion schemes are a well-known and well-established model of computer programs. They are a concept that can be said to date back to as early as the 1980s where they were introduced as a means of providing a semantics to programming languages. However, they have recently come to the forefront of research again, but with another purpose. They have been reintroduced as part of an ongoing effort to investigate and classify various means of generating infinite structures, with a particular view to understanding which of these have decidable model-checking properties.

At its simplest, a higher-order recursion scheme is a system of typed equations, such as the one presented in Fig. 1.1.

$$\begin{aligned}\underline{S} &= Fgab\\ F\varphi xy &= f(F(F\varphi x)yx)(G\varphi x)\\ G\varphi x &= \varphi(G\varphi x)\end{aligned}$$

Figure 1.1: A recursion scheme.

There are several ways of giving such a system of equations a semantics, and for our current purposes it will be beneficial to think of the equations as a system of rewrite rules, the semantics of which is given by the infinite term that results through the repeated unfolding from a designated start symbol, in this case the underlined one, S. In Fig. 1.2 we show the infinite term tree (truncated to the first 4 levels) that results from the recursion scheme in Fig. 1.1. It should be obvious from this example, even without formal definitions[1], that a higher-order recursion scheme is a finite description of a potentially infinite structure.

One of the classical problems in computer science is that of verification. Given a system S (where S denotes a program, a protocol, or similar) and a property p, is it true that the behaviour of S satisfies the property p? Model-checking [CGP99] is a method that aims to solve this problem *automatically*, i.e. without human intervention. It does so by constructing

[1]These will be introduced in due course.

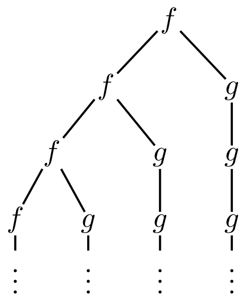


Figure 1.2: The infinite term tree defined by the recursion scheme in Fig 1.1

a model M_S that captures the behaviour of S; and p will most commonly be described by a formula φ of a chosen logic. The missing ingredient is then, of course, an algorithm to verify whether M_S satisfies φ. The model-checking problem is thus parameterised by two variables: (1) the set of the models we wish to consider and (2) the logic we will use to express the properties of interest. It should be clear that the choice of these two parameters will influence the success of our model-checking approach. Let (1) be the set of Turing machines (or some logical representation thereof) and (2) any logic capable of expressing the property that a Turing machine halts. Clearly, we have no hope of ever getting off the ground here as we have reduced the model-checking problem to the halting problem [Tur36].

Let us sharpen the definition of a *model*. For our purposes, the most general models we will consider will be simple directed graphs $\langle V, (E_a)_{a\in A}, (P_l)_{l\in L} \rangle$, where V is a set of vertices and $E_a \subseteq V \times V$ denotes the a-edge relation so that if $(u, v) \in E_a$ there exists an edge from u to v labeled by $a \in A$. Finally, $P_l \subseteq V$ denotes a unary relation, and if $v \in P_l$ this indicates that the vertex v has label l. Note that we have the additional property that the $P_l \cap P_k = \emptyset$ for $l \neq k$ and also $\bigcup P_l = V$. In other words, each vertex has precisely one label. We assume the alphabets A and L to be finite. It is the cardinality of V that determines whether a model is finite or not.

Even with its limitations, *finite-state* model-checking[2] has proved to be one of the success stories of computer science as many systems can be adequately described by finite systems, circuits being the obvious and most widely quoted example. Commonly used logics will include LTL, CTL*, and μ-calculus in increasing order of expressibility. See [Sti01b] for an overview of each of these along with weaker logics. Selected examples of the results and applications of the model-checking problem of these logics with respect to finite structures can be found in [EMC86, QS81, CS92, EJS01]. See also [BS01] for a survey of the model-checking problem with particular emphasis on the modal μ-calculus.

Indeed, even industrial scale model-checking takes place [IDM94, BBDEL96]. And while this remains a very active area of research[3], one cannot ignore the fact that limiting ourselves to finite models *is* a restriction.

Thus, considerable effort in the past few years has been dedicated to transferring the model-checking paradigm from the domain of finite-state systems to that of potentially infinite-state systems, such as the tree shown in Fig 1.2. However, this transition is far from straightforward. From the outset we are faced with two concerns that are a non-issue

[2]In other words, of models with a finite vertex set.

[3]One particularly sought-after result is a proof or disproof of whether μ-calculus model-checking is in P.

with finite-state systems. As hinted at previously, first we must limit ourselves to only finitely-representable infinite structures (for input into the model-checking algorithm), secondly, we must ensure that the model-checking problem for this finitely-representable infinite structure is indeed decidable (the existence of an algorithm) with respect to the properties in which we are interested.

This has been an incredibly fertile area of computer science resulting in many publications and many theses [Mey05, Cac03, Wöh05], including this one. The subject matter of this thesis is one particular mechanism for generating infinite structures: higher-order recursion schemes.

Higher-order recursion schemes have featured prominently in several areas of computer science; particularly programming language semantics and language theory (as well as connections between the two). They are also a very natural way of generating structures recursively, as demonstrated informally at the beginning of this thesis; not to mention they are not unlike a system of equations defining a LISP or Haskell program. From a computer science point of view, it is therefore natural to want to explore the exact nature of the structures they produce and consider the model-checking problem for these. Not only can this help to better understand the landscape of infinite structures that possess a decidable model-checking problem, but it may also offer new results for other areas of computer science, or indeed recast old results in a model-checking framework[4].

Recently, Knapik et al. have considered the problem of whether or not the term trees that result from the infinite unfolding of a higher-order recursion scheme possess a decidable model-checking problem with respect to monadic second-order logic. In other words: investigating whether the term trees defined by higher-order recursion schemes possess a decidable monadic second-order theory. They have succeeded in proving the remarkable result [KNU01, KNU02] that higher-order recursion schemes possess a decidable monadic second-order theory *provided the recursion scheme is safe*. Here, safety is a syntactic restriction on the right hand sides of the equations of a recursion scheme. Knapik et al. went on to prove the equivalence between the hierarchy[5] of *safe* higher-order recursion schemes and higher-order pushdown automata [KNU02]. The significance and robustness of safe higher-order recursion schemes was then asserted again by its correspondence with the Caucal hierarchy – we return to this later.

Thus, although the above would seem to suggest that the safety constraint is a natural phenomenon, very little is understood about it. At the time of starting this thesis, no results concerning unsafe recursion schemes (those that do not satisfy the safety constraint) existed. This brings us to the subject matter of this thesis.

Our thesis is an investigation of higher-order grammars (a generalisation of higher-order recursion schemes). We seek to understand the classes of structures they can generate and, in particular, what impact safety has on these structures. Precisely, we seek an answer to the following two questions:

1. (Expressibility) Let G be a level-n unsafe grammar defining a structure $[\![G]\!]$. Does there exist a level-n safe grammar G' such that $[\![G]\!] = [\![G']\!]$?

[4]One may even be interested in applying this to verifying properties of pure functional programs, given that such programs lend themselves easily to representation by higher-order recursion schemes. However, this is more subtle than it first appears as ultimately the term tree produced by a higher-order recursion scheme corresponds to an *expression* tree or, equivalently, an uninterpreted syntax tree in this case. So any *direct* model-checking on this tree is likely to say more about the syntax and control structures of the program rather than the actual behaviour of this program.

[5]Higher-order recursion schemes can be classified by the level of the types occurring in the equations and one can thus form an infinite hierarchy of recursion schemes.

2. (Decidability) In the case where G is a level-n recursion scheme, does the term tree defined by G, $[\![G]\!]$, possess a decidable monadic second-order theory?

Before we provide an overview of the results contained herein and their ramifications we start with a brief tour of other recent developments in the wider field of infinite-state verification. This should help place into context our own results.

1.2 Recent Developments

Recall that, unlike the finite case, we must ensure that an infinite model has a finite representation and furthermore has a decidable theory with respect to our chosen logic. Let us briefly survey some of the mechanisms for generating finitely-representable infinite structures.

As mentioned before, the most general structure we wish to consider will be simple, directed infinite graphs with edge-labels and possibly vertex labels as well. We will later focus our attention to term trees (these are, effectively, deterministic graphs). For a systematic study of the classification of infinite graphs and their properties a good place to start is [Mey05].

1.2.1 Finitely-Representable Infinite Structures

There are various mechanisms for generating finitely-representable infinite structures. A useful classification has been devised by Carayol and Colcombet [CC03] and is widely used throughout the literature:

Equational: One considers a finite system of equations whose solution (i.e. semantics) is given by an infinite structure. The family of structures obtained in this way depends on the choice of operators. For example, we can consider the equational graphs of Courcelle (hyperedge replacement [Cou89], vertex replacement [Cou90]) where atomic operations define operations that capture the concept of gluing graphs together, renaming nodes and so forth. Higher-order recursion schemes can also be considered equational.

Internal: This gives an exact description of both the universe (the set of vertices) and the relations of the structure. An example of this is the configuration graph of a pushdown automaton. The vertices of the graph represent the total states of the pushdown automaton, for example, (q, s), where q is the current state and s is the current stack. Then, an edge from (q, s) to (p, t) exists if the pushdown automaton can change from the global state of (q, s) to (p, t) in one transition. In the same vein, the configuration graph of a Turing machine, is also an internal description.

Transformational: These are described by a finite sequence of applications operations to a finite initial system. Such operations include tree iterations [Sem84, Wal96], unfoldings [Cou95, CW98], and MSO-definable transductions [Cou94]. In this case, the operations we have mentioned are all MSO-compatible in the sense that if a structure has a decidable MSO theory, applying such an operation will result in a new structure in such a way that the decidability of MSO is preserved.

These techniques are not independent of one another, and often a class of graphs defined via one technique has later been found to be equivalent to a class defined by other techniques. For example, the prefix-recognisable graphs [Cau92] have at least four well-known characterisations in the literature:

1. equational: VR equational systems [Cou90, Bar97].
2. transformational: via an MSO-definable transduction in the complete binary tree, when the latter is represented as the following logical structure $\{\{1,2\}^*, S_1, S_2\}$ where $S_i = \{(w, wi) : w \in \{1,2\}^*\}$. See [Blu01] for details.
3. transformational: resulting from the application of two transformations applied in alternation. First an inverse rational mapping, followed by an unfolding, applied to the complete binary tree [Cau96].
4. internal: given as a finite set of rewrite rules over an alphabet Γ; each rule is of the following form $U_1 \xrightarrow{a} U_2$, where U_1 and U_2 are both regular sets of words over Γ. The resulting graph then consists of the vertices $V \subseteq \Gamma^*$ that form a regular set, and we have that there exists an a-edge from the vertex u_1w to u_2w where $u_1 \in U_1$ and $u_2 \in U_2$ if and only if $U_1 \xrightarrow{a} U_2$.

Two short but excellent surveys introducing the construction of infinite graphs (but with a bias towards those with a decidable MSO theory) may be found in [Tho01, Tho03]. For yet another survey on infinite but finitely representable structures of interest and relevance to computer science see [BG04].

1.2.2 Finitely-Representable Infinite Structures with a Decidable Model-Checking Problem

We consider the model-checking problem with respect to monadic second-order (MSO) logic. This means that the properties of interest to us are those expressible as an MSO formula. Formal details of this logic will be provided later, but it can be defined succinctly as first-order logic augmented with *set* variables and quantification over these set variables. MSO logic is a very expressive logic expressing not only local modalities such as X happens next but also temporal properties such as liveness (something eventually happens), safety (something never happens), and fairness (if something happens infinitely often, then something else happens infinitely often as well).

Our interest in MSO logic is due to the fact that it *is* the benchmark logic for verification in terms of expressibility. It subsumes all commonly used logics such as LTL, CTL*, and the μ-calculus. See [JW96] for a perspective on what constitutes a good verification logic. The high price to pay for this expressive power is, as might be expected, in the complexity of model-checking (see [Tho97] for an overview of some of the complexity results in the finite case).

Thus, in this thesis we are particularly concerned with finitely-representable infinite structures for which we can construct an algorithm, so that given any sentence φ in MSO we can verify automatically whether φ is true for this infinite structure. We can also rephrase this in slightly more mathematical terminology by saying that we are interested in classes of finitely-representable infinite structures possessing a *decidable monadic second-order theory* – or simply monadic theory for short.

In the following we consider the range of structures that are known to possess a decidable monadic second-order theory. We progress from individual infinite structures, to classes of infinite structures and finally hierarchies of classes in roughly chronological order.

Individual Infinite Structures

Our starting point is Rabin's result on the decidability of the monadic second-order theory of the complete and infinite binary tree [Rab69]. This result is one of the most widely applied in theoretical computer science[6]. In his paper [Rab69], Rabin already infers many other decidability results from the complete and infinite binary tree by the technique of interpretation. Here, a structure B is defined "inside" a structure A by defining the domain and relations of B via MSO formulas over the structure A. One may then infer the decidability of the monadic theory of B provided A has a decidable monadic theory. Using this technique Rabin showed the decidability of the complete k-ary tree (for any fixed k) and also the decidability of several structures of interest to mathematics, such as that of $(Q, <)$.

Classes of Infinite Structures

Rabin's results above are about a *specific* graph or a *specific* structure. We now move to *classes* of structures, beginning with Muller and Schupp.

In 1985, Muller and Schupp [MS85] introduced pushdown graphs and showed that they possessed a decidable monadic theory. This was done by way of introducing an alternative characterisation of such graphs, known as the context-free. Muller and Schupp then go on to show that every context-free graph can be MSO interpreted in the infinite binary tree and thus the result is a direct application of Rabin's result.

Later, Courcelle [Cou95] showed the monadic theory remains decidable for the equational graphs (with hyperedge replacement). This decidability result, again, is proved via an interpretation in the infinite and complete binary tree. For rooted graphs of finite degree, these are the context free graphs, hence Courcelle's result is a generalisation of Muller and Schupp's.

However, both results were subsumed by Caucal's result [Cau96] showing the decidability of the prefix-recognisable graphs. It is important to note that all of the above results are extensions of Rabin's Theorem via interpretation.

Instead of merely considering MSO interpretations, a more sophisticated use of Rabin's theorem leads to the decidability of structures beyond those that are directly interpretable in the complete and infinite binary tree. For example, already in [Cou95], Courcelle showed that algebraic trees possess a decidable MSO theory. Algebraic term trees correspond to the term trees produced by level-1 recursion schemes and are trees that result from the unfolding[7] (from a given vertex) of deterministic prefix-recognisable graphs.

Hierarchies of Classes of Infinite Structures

The progression above, from considering the monadic theory of a *single* structure (such as the complete and infinite binary tree) to *classes* of structures (such as pushdown graphs or prefix-recognisable graphs) is now extended to *hierarchies* of classes of structures.

[6]Often quoted as the "mother of all decidability results."

[7]Given a graph, the unfolding from a vertex v gives rise to a tree where each vertex corresponds to a path in the graph (from v) and node π has an edge to π' if and only if π' is an extension of the path in π by exactly one node. It was later shown [CW98] that the unfolding is an MSO-compatible operation.

As indicated earlier, the present thesis will concern itself with one particular hierarchy of infinite objects: the set of term trees produced by higher-order recursion schemes[8]. Let us recall the foundations of this hierarchy and its relationship with other hierarchies.

In [KNU01, KNU02], Knapik et al. revisited the notion of a higher-order recursion scheme (originally introduced by Damm). As mentioned earlier, a level-n recursion scheme defines a single, potentially infinite term tree. At level-0 we have the regular terms, at level-1 the algebraic terms, and levels 2 and beyond have not received much study, although it is known that the hierarchy is strict. Knapik et al. sought to extend Courcelle's result for level 1 to recursion schemes of all levels. In [KNU01], Knapik et al. succeeded in showing that for level-2 recursion schemes subject to the restriction of safety the term trees produced possess a decidable MSO theory. In [KNU02], they extended this result to recursion schemes of all levels – again conditional on the safety constraint. Furthermore, Knapik et al. proved the correspondence between level-n safe recursion schemes and level-n pushdown automaton over term trees. Here, a level-0 pushdown automaton corresponds to a finite automaton, a level-1 pushdown automaton consists of a stack, and a level-n pushdown automaton is the generalisation of this, possessing a stack of level-$(n-1)$ stacks.

This latter correspondence (between safe recursion schemes and higher-order pushdown automata) highlighted the robustness of the class of term trees generated by safe recursion schemes; which we shall, from now on, denote by safe term trees. In fact, the hierarchy of safe term trees turned out to be equivalent to another important hierarchy: the Caucal hierarchy [Cau02a, Cau02b, CW03, Cac03].

Both the transformations of unfoldings and inverse rational mappings have been shown to be MSO-compatible [CW98] and [Cau96] respectively. Caucal takes this one step further by iterating them, in alternation, to obtain a hierarchy of graphs and trees often referred to as the Caucal hierarchy. We define:

$$\begin{aligned} Tree_0 &= \text{Finite trees} \\ Graph_n &= Rat^{-1}(Tree_n) \\ Tree_{n+1} &= Unf(Graph_n) \end{aligned}$$

where $Tree_n$ is the set of trees of the nth level of the Caucal Tree hierarchy, and $Graph_n$ is the set of graphs of the nth level of the Caucal Graph hierarchy. Note that we have two hierarchies: one for trees and one for graphs. Here, given a set S, $Rat^{-1}(S)$, is the set of all graphs that can be generated by applying an inverse rational mapping to a structure in S. Similarly, given a set S, $Unf(S)$ is the set of all trees that can be generated by applying the unfolding operation from a given vertex in a structure in S. Note that $Tree_0$ is the set of finite trees, $Graph_0$ the set of finite graphs, $Tree_1$ the set of regular trees, and $Graph_1$ the prefix-recognisable graphs. Little is know or understood about the trees and graphs occurring at higher levels.

It was shown, by Caucal, that under suitable "determinacy" restrictions, we can produce, instead of $Tree_n$, the hierarchy $Term_n$, consisting of only term trees over a finite signature, and that this corresponds exactly to the level-n safe term trees. For this to work, however, we must instead begin with $Term_0$ equal to the set of regular terms.

One final result we mention is one due to Courcelle and Knapik [CK02], who consider the

[8]Although we will, in fact, be considering the more general concept of higher-order grammars.

operation of evaluating a first-order substitution in a tree. Courcelle and Knapik prove that when applied to a tree generated by a level-n safe recursion scheme, this evaluation yields a tree that is generated by a level-$n+1$ safe recursion scheme. They show that this operation of evaluation is MSO-compatible, and hence, repeated application can generate a term tree of arbitrary level with a decidable monadic theory. The question of whether every safe term tree could be generated in this fashion was answered by Caucal [Cau02b]. In fact:

$$(Scheme_n) = (Term_n) = (Sub_n) \quad \text{for } n \geq 0$$

where $(Scheme_n)$ is the set of term trees produced by safe higher-order recursion schemes of level-n, $Term_n$ is the nth level of the Caucal hierarchy restricted to terms, and Sub_n is the set of term trees produced by evaluating a first-order substitution in a tree from Sub_{n-1}. Note that $Scheme_0 = Term_0 = Sub_0$ are defined to be the regular terms.

1.3 Higher-order Grammars and the Safety Restriction

The above should hopefully illustrate the recent attention demanded by *safe* recursion schemes. It also prompts a couple of obvious questions. For example, what is the relationship between safe and unsafe recursion schemes? Is safety necessary for a recursion scheme to guarantee MSO decidability of the resulting term tree? It is these questions from infinite-state verification that have largely motivated the topic of this thesis.

However, we would like to be able to answer more than this. Higher-order recursion schemes can be considered a special case of higher-order grammars. Building on earlier work [Ind76, ES77, ES78], Damm [Dam82] defined the OI and IO hierarchy of safe[9] higher-order grammars. Here, a higher-order grammar is a potentially non-deterministic system of equations. As we will see, a higher-order grammar can be configured to produce any number of different structures from a language of words (in the classical sense), to a language of finite term trees, to a forest of infinite term trees and so forth. Associated with a higher-order grammar is a mode of derivation, which dictates how one can define a structure from the system of equations. The three modes considered by Damm are the OI, IO and unrestricted. The OI and IO correspond, roughly, to call-by-name and call-by-value evaluations, and the unrestricted mode should be self-explanatory. Damm also showed that the OI mode of derivation is sufficient, in the sense that if there exists a derivation of a particular structure, then there exists an OI derivation that generates it as well.

As we will see, a higher-order recursion scheme as recently considered by Knapik et al. is really a special case of an OI higher-order grammar. Therefore, instead of focusing exclusively on recursion schemes, we shift focus to *grammars*.

The preceding results from the realm of infinite-state verification should hopefully illustrate safe recursion schemes have very good properties. Not only do the term trees they produce enjoy a decidable monadic theory but they also have several other characterisations. However, these "good" properties were hinted at even in the earlier investigations of Damm and Goerdt [Dam82, DG86]. Although Damm considered higher-order grammars in their full generality, some of the most notorious results pertain to the word language case. In [DG86], Damm and Goerdt gave the first correspondence showing that a language L is produced by

[9]Damm's terminology and notation is distinct from our own and that of Knapik et al.'s but we will make amends for this in Chapter 2.

a level-n safe OI *word* grammar if and only if it is the language of a level-n pushdown automaton over *words*. Here a language is taken to be a set of words. They also showed that the class of level-n safe OI languages forms a full AFL [Dam82]. Whereas Damm and Goerdt did not use the terminology "safe" they did use the restriction of "derived types" and we will see that the two are synonymous (equivalent) as far as grammars are concerned.

It is interesting to point out that, at the time this thesis began, results concerning potentially unsafe grammars (or schemes) – those that do not satisfy the safety restriction – were nonexistent. The goal of this thesis is to rectify this. The aim is to understand whether safety is a genuine or spurious constraint in terms of expressibility, and, in the case of recursion schemes, decidability. Whilst this has always been a legitimate question to ask, the need for an answer now seems to have become more urgent. We are now in a position where, depending on the outcome of our results, we may be able to define a new set of infinite terms trees with a decidable MSO theory. Or, in the other direction, we may have found a suitable normal form for (until recently) unwieldy unsafe grammars.

1.4 Contributions and Organisation of this Thesis

In Chapter 2 we lay down the framework that will support the rest of this thesis. This chapter introduces the concept of higher-order grammars, recursion schemes and the safety constraint. In addition we also introduce higher-order pushdown automaton as automata-theoretic characterisations of safe grammars. We conclude with some of the key results in this field as well as chronicle of how higher-order grammars and related concepts have evolved with the times, both in terms of purpose and appearance. Although the bulk of this chapter consists of mainly preliminaries and the statement of some standard results, one particular highlight is the safe λ-calculus; which serves as an alternative characterisation of safety.

Chapter 3 focusses on the issue of expressibility. Ultimately, we would like to address the following question. Let G be a level-n unsafe grammar with mode of derivation m producing a structure $[\![G]\!]$; does there exist a level-n safe grammar G' with mode of derivation m producing a structure $[\![G']\!]$ such that $[\![G]\!] = [\![G']\!]$? To answer this in full generality (for any G and for any m) is a very formidable question. As far as infinite-state verification is concerned we would be content with the restriction to G being a recursion scheme. Unfortunately, we are unable to answer this, but provide some first steps in this direction. We consider the restricted setting where G defines an OI word language (in the classical sense), and for succinctness we refer to such a grammar as a *word grammar*. Our main contribution of this chapter is an extension of Damm and Goerdt's result [DG86] who show that every level-n safe OI word grammar can be converted into a level-n pushdown automaton and vice versa. We show that every level-2 *unsafe* OI word grammar can be converted into a level-2 pushdown automaton. A corollary of this result is that safety is a spurious constraint at level 2 for OI word languages. This result was published in FoSSaCS '05 [AdMOb]. The proof proceeds by the introduction of a new automata-characterisation of OI word grammars called the pushdown automaton with links and we show that at level-2 we can convert this into a level-2 non-deterministic pushdown automaton. Our proof makes essential use of non-determinism and we explore this further by studying languages of deterministic higher-order pushdown automaton and showing their equivalence with deterministic *safe* OI higher-order grammars. This investigation eventually leads us to our conjecture that there exists a level-2 unsafe deterministic OI word grammar that is not equivalent to any level-2 safe deterministic OI word grammar. Lastly, as an interesting aside we provide an alternative decomposition of Urzyczyn's language [Urz03] – this is a language that is straightforward to define with an unsafe level-2 OI word grammar but was conjectured to be inherently unsafe – i.e. it could not be captured by a safe one. The main result of this chapter clearly shows this is not the case, however, due to the interest garnered by this language we also give a unique decomposition of the language where it is very easy to see that this language can be accepted by a level-2 pushdown automaton.

Chapter 4 concerns the other half of the problem: that of decidability. Here we *do* focus our attention to recursion schemes exclusively. We will show that for every level-2 recursion scheme (whether safe or not) the resulting infinite term tree produced possesses a decidable MSO theory. This result was first presented at TLCA '05 [AdMOa]. This chapter presents all the technical details behind the sketch proof outlined in this article.

Finally, we conclude with Chapter 5, where we present a summary of our results and potential research directions.

To summarise: our two main contributions are in the form of two theorems. Both concern level-2 OI grammars. The first is a theorem on expressibility where we show that level-2 unsafe OI word grammars can be converted into level-2 safe OI word grammars [AdMOb]. Our second is a theorem on decidability where we show that all level-2 recursion schemes possess a decidable MSO theory [AdMOa]. Two smaller highlights of this thesis are the alternative decomposition of Urzyczyn's language and, a simple but desirable proof of the equivalence between level-n safe deterministic OI word grammars and deterministic pushdown automata.

The reader should note that related work will be presented concurrently with the chapter to which it pertains. However, one result that deserves special mention is that of Luke Ong's [Ong06c, Ong06b]: extending the results of Chapter 4 to the whole hierarchy. This was published shortly before the completion of this thesis and it generalises the concepts and framework set down in Chapter 4 to recursion schemes of all levels. In addition he proves that the modal μ-calculus model-checking problem for level-n is n-EXPTIME complete.

Chapter 2

Preliminaries

In this chapter we introduce the concept of higher-order grammars, higher-order recursion schemes and the safety restriction. For our purposes, we are purely interested in higher-order grammars and higher-order recursion schemes as generators of *structures*. These structures will be, respectively: sets of finite term trees and single infinite term trees. We now give some introductory remarks to explain the format this chapter has assumed.

The reader familiar with the key concepts in this thesis can probably sympathise that the terms "higher-order grammars" and "higher-order recursion schemes" appear in the literature under many definitions, furthermore numerous other terms communicating similar concepts exist. Aside from fixing the definitions that *we* will use throughout the rest of this thesis, one of the principal aims of this chapter is to relate our definitions to those occurring in other literature.

We adopt the following approach. We define higher-order grammars as a system of non-deterministic equations, and with each such system we define its semantics to be the set of finite term trees that can be derived from one distinguished equation. Thus, a higher-order grammar as we define it here produces a term tree language (where each term tree in the language is finite).

A higher-order recursion scheme, on the other hand, defines a single (potentially) infinite term tree. Appearance-wise, a higher-order recursion scheme is indistinguishable from a *deterministic* higher-order grammar. However, to ascribe its semantics we perform two steps. First, with every higher-order recursion scheme we associate a higher-order grammar called the *schematological tree grammar*. The semantics of this higher-order grammar is then, by the definition of higher-order grammars, a set of finite term trees. However this set of finite term trees possesses the property that, under an appropriate partial ordering, it is a directed set. Then, taking the join (least upper bound) of this set renders the (potentially) infinite term tree defined by the higher-order recursion scheme. These definitions and the distinction between higher-order grammars and higher-order recursion schemes is consistent with the definition scheme laid out by Damm [Dam82].

As mentioned in the introduction, this thesis is largely motivated by recent results on higher-order recursion schemes [KNU01, KNU02][1]. However, almost all results in this chapter can and will be formulated in terms of higher-order grammars. The reason we have opted for this is that we believe it to be the most general presentation of many of the results we wish

[1]Knapik, Niwiński and Urzyczyn use the terminology "higher-order grammar" to be equivalent to *our* definition of a "higher-order recursion scheme."

communicate and furthermore, the corresponding result for higher-order recursion schemes will often "fall out" as a simple corollary.

This chapter is organised as follows. In the first part, we will introduce higher-order grammars and higher-order recursion schemes along with the structures that they generate.

We then introduce the concept of a *safe* higher-order grammar. The definition of safety that we use is borrowed from Knapik et al. [KNU01, KNU02]. This, however, is merely a characterisation and we present an alternative formulation of the safety restriction: via the safe λ-calculus, developed alongside my colleagues Luke Ong and Klaus Aehlig. We present this because it is interesting in its own right, and also because the author believes it offers an interesting perspective on why safety has proven to be so useful in the past – something that is not so obvious from the more compact definition due to Knapik et al. Our definition of the safe λ-calculus will not be widely exploited in this thesis, but it is hoped that it may be the starting point of further research. We return to this in the conclusion of this thesis.

Next we introduce higher-order pushdown automata as a machine characterisation of *safe* higher-order grammars. The first incarnation of this result is due to Damm and Goerdt, who, in their paper [DG86], who consider higher-order grammars and higher-order pushdown automata as definitions of word languages (as opposed to tree languages). It was later proved by Knapik et al. [KNU02] in the context of higher-order recursion schemes (although their terminology differs from our own).

We dedicate the final section to outlining how higher-order grammars, higher-order recursion schemes and related concepts have evolved over the past few decades and also state some key results. The reader will note in reading this chapter that the early sections contain little in the way of historical remarks. This is intentional, as all such remarks are relegated to this section. It is felt that these may be better appreciated in hindsight once we have fixed our own definitions as a basis. We culminate in a restatement of the problem our thesis aims to investigate; that of the safety restriction and whether or not it is a genuine constraint on the structures generated by higher-order grammars.

2.1 Types, Terms and Term Trees

2.1.1 Simple Types and Applicative Terms

We shall consider ***simple types*** (or just ***types*** for short) as defined by the grammar $A ::= o \mid A \rightarrow A$. Each type A, other than the ground type o, can be uniquely written as $(A_1, \cdots, A_n, o)$ for some $n \geq 1$, which is a shorthand for $A_1 \rightarrow \cdots \rightarrow A_n \rightarrow o$ (by convention, $\rightarrow$ associates to the right). We define the ***level***[2] of a type by $\mathsf{level}(o) = 0$ and $\mathsf{level}(A \rightarrow B) = \max(\mathsf{level}(A) + 1, \mathsf{level}(B))$.

If the simple type A can be uniquely written as $(A_1, \cdots, A_n, o)$, then we say that A has ***arity*** n. We write $\mathsf{ar}(A) = n$.

Following [KNU01, KNU02], we say that o is ***homogeneous***, and for $n \geq 1$, $A = (A_1, \cdots, A_n, o)$ is ***homogeneous*** just if $\mathsf{level}(A_1) \geq \mathsf{level}(A_2) \geq \cdots \geq \mathsf{level}(A_n)$, and each A_i is homogeneous. For example, the type $(o \rightarrow o) \rightarrow (o \rightarrow o \rightarrow o) \rightarrow o \rightarrow o$ is homogeneous, but $o \rightarrow (o \rightarrow o) \rightarrow o$ is not.

A ***typed alphabet*** is a set Γ of symbols such that each symbol in Γ has a simple type. Γ can also be presented as a family indexed by the simple types, $\{\Gamma^A\}_A$, where Γ^A is the set of

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

all symbols of Γ of type A.

Given a typed alphabet Γ, the set $\mathcal{T}(\Gamma) = \{\mathcal{T}^A(\Gamma)\}_A$ of ***applicative terms*** is defined inductively by

1. $\Gamma^A \subseteq \mathcal{T}^A(\Gamma)$
2. if $t \in \mathcal{T}^{A\to B}(\Gamma)$ and $s \in \mathcal{T}^A(\Gamma)$, then $(ts) \in \mathcal{T}^B(\Gamma)$

As usual, application associates to the left, and we will often write $t_1t_2t_3\cdots t_n$ for the expression $(((t_1t_2)t_3)\cdots t_n)$. Note that if t is an applicative term in $\mathcal{T}^A(\Gamma)$ then we sometimes write $t : A$ to indicate its type. If t is a term of type A, then we define the arity of t, denoted $\mathsf{ar}(t)$, to be $\mathsf{ar}(A)$.

2.1.2 Term Trees

We define a ***ranked alphabet*** as a non-empty set Σ such that there exists an associated ***rank*** function $\varrho : \Sigma \longrightarrow \omega$. In particular, if $f \in \Sigma$ and $\varrho(f) = 3$, this means that f has rank 3. In addition to a rank function ϱ, we supply a direction function, Dir, where $Dir(f)$ is a totally ordered set of size $\varrho(f)$. In particular, we will always assume that $Dir(f)$ can be written as the ordered sequence $x_1, x_2, \cdots, x_{\varrho(f)}$.

The free monoid generated by a set X is written X^* and the empty word is written ϵ. A ***tree*** is any non-empty prefix-closed subset of X^* (with ϵ as the root). The length of a word $w \in X^*$ is denoted by $|w|$.

A ***term tree*** t ***over a ranked alphabet*** Σ is a function $t : T \longrightarrow \Sigma$ where T is tree over the free monoid generated by $\bigcup_{f\in\Sigma} Dir(f)$. Furthermore, if $t(w) = f$ for some $w \in T$ and $\varrho(f) = n$, then w has exactly n children. In particular, as we assume that $Dir(f)$ is totally ordered and therefore can be written as the ordered sequence $x_1, x_2, \cdots x_n$, these children will be $wx_1, wx_2, \cdots, wx_n$. Here, wx_i is referred to as the ith child. Note that if $Dir(f) = \emptyset$, then f is a terminal node and therefore it has no children. More conveniently, we call t a Σ-tree. The set of all Σ-trees is denoted by $T^\infty(\Sigma)$.

Remark 2.1.1. Unless otherwise stated, we take $Dir(f)$ for $f \in \Sigma$ to be $\{1, 2, \cdots, \varrho(f)\}$. And therefore if $t : T \longrightarrow \Sigma$ is a term tree, and $t(w) = f$, f will have precisely the following children: $w1, w2, \cdots, w\varrho(f)$. Indeed, for all but Chapter 4, we will adopt the convention that $Dir(f) = \{1, 2, \cdots, \varrho(f)\}$. We will often write $[k]$ as shorthand for $\{1, 2, \cdots, k\}$ where k is a natural number.

For $v \in Dom(t)$, the subtree of t induced by v is a Σ-tree, $t.v$, such that (1) $Dom(t.v) = \{w \in \omega^* : vw \in T\}$ and (2) $t.v(w) = t(vw)$.

Note that it will be convenient for us to understand Σ, the ranked alphabet, to be a typed alphabet of level at most 1. Hence a symbol f with rank n can be understood to have type $\underbrace{o \to o \to \cdots \to o}_{\text{n times}} \to o$. Note that if f has rank 0, then f has type o.

With this alternative interpretation, we can identify finite trees in $T^\infty(\Sigma)$ with applicative terms of type o over the alphabet Σ. Here, as might be expected, a tree $t : T \to \Sigma$ is said to be finite if and only if T is finite.

For example, the tree below is identified with the following term in the obvious way $fa(fa(ga))$ where $\varrho(f) = 2, \varrho(g) = 1, \varrho(a) = 0$. Equivalently, $f : o \to o \to o, g : o \to o, a : o$.

We can therefore think of the infinite trees in $T^\infty(\Sigma)$ as being identified with infinite terms.

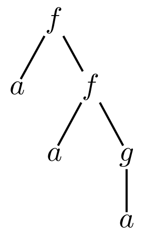


Figure 2.1: The term tree $fa(fa(ga))$.

2.1.3 Σ-algebras

Our definitions of the structures defined by higher-order grammars and higher-order schemes are algebraic in nature. We will need only a few standard definitions, which we present here.

First, let us recall that a ***partial order*** $(A, \leq)$ is a set A equipped with a relation $\leq \subseteq A \times A$, obeying the following:

1. $\forall x \in A. x \leq x$
2. $\forall x, y \in A. x \leq y \wedge y \leq x \Rightarrow x = y$
3. $\forall x, y, z \in A. x \leq y \wedge y \leq z \Rightarrow x \leq z$

Let $D \subseteq A$, D is said to be ***directed*** if:

$$\forall x, y \in D. \exists z \in D. x \leq z \wedge y \leq z$$

A ***complete partial order*** $(A, \leq, \bot)$ is a partial order $(A, \leq)$ such that A contains a minimal element $\bot$ (so that $\forall x. \bot \leq x$) and every directed set D in A has a least upper bound in A, and is denoted by $\bigsqcup D$.

Let Σ be a ranked alphabet, as above.

Definition 2.1.2. A Σ-algebra is a pair $\underline{A} = (A, \varphi_A)$, where A is a set and φ_A assigns each $f \in \Sigma$ a function $A^{\varrho(f)} \longrightarrow A$ for $\varrho(f) \geq 1$ and for each $a \in \Sigma$ with $\varrho(a) = 0$ an element of A.

Example 2.1.3. Consider the following set of base operations: $\Sigma = \{\texttt{binop}^2, \texttt{id}^0\}$ with associated ranks written as superscripts. Next, consider the following Σ-algebra, $\underline{A} = (\omega, \varphi_A)$ where

$$\begin{array}{rcl} \varphi_A(\texttt{binop}) & : & \omega \times \omega \longrightarrow \omega \text{ where} \\ \varphi_A(\texttt{binop})(n, m) & \mapsto & n + m \\ \varphi_A(\texttt{id}) & : & \omega \text{ where} \\ \varphi_A(\texttt{id}) & \mapsto & 0 \end{array}$$

Definition 2.1.4. Let $\underline{A} = (A, \varphi_A)$ and $\underline{B} = (B, \varphi_B)$ be Σ-algebras, we say that $h : A \longrightarrow B$ is a Σ-homomorphism if and only if, for every $f \in \Sigma$:

$$h(\varphi_A(f)(a_1, \cdots, a_{\varrho(f)})) = \varphi_B(f)(h(a_1), \cdots, h(a_{\varrho(f)}))$$

Intuitively, h preserves the meaning of all functions $f \in \Sigma$, and we can reason about B by lifting results to A. If $h : A \longrightarrow B$ is a Σ-homomorphism then we write $h : \underline{A} \longrightarrow \underline{B}$.

Definition 2.1.5. A Σ-algebra $\underline{A} = (A, \varphi_A)$ is ***ordered*** if and only if A is a partially ordered set $(A, \leq)$ with a minimal element $\perp$ and all operations $\varphi_A(f)$ are monotone.

Furthermore, we say that a Σ-algebra $\underline{A} = (A, \varphi_A)$ is ***continuous*** if and only if $\underline{A}$ is a complete partial order and all operations are continuous: in other words, if $f \in \Sigma$ and $D_j \subseteq A$ is directed for $1 \leq j \leq \varrho(f)$ then

$$\varphi_A(f)(\sqcup D_1, \cdots, \sqcup D_{\varrho(f)}) = \sqcup \varphi_A(f)(D_1, \cdots, D_{\varrho(f)})$$

Note that we can organise $T^\infty(\Sigma)$ into a Σ-algebra, which we denote as follows $\underline{T^\infty(\Sigma)} = (T^\infty(\Sigma), \varphi_{T^\infty(\Sigma)})$ where:

$$\varphi_{T^\infty(\Sigma)}(f)(t_1, \cdots, t_{\varrho(f)}) \quad \mapsto \quad \{(\epsilon, f)\} \cup \bigcup_{1 \leq i \leq \varrho(f)} \{(iw, g) | (w, g) \in t_i\}$$

In other words, the function f takes $\varrho(f)$ term trees from $T^\infty(\Sigma)$ and maps them onto the unique tree with root f where the subtree rooted at child i is t_i.

Furthermore, we can organise $\underline{T^\infty(\Sigma)}$ into an *ordered* Σ-algebra, which we denote by $\underline{CT^\infty(\Sigma)}$ by:

1. Introducing $\perp$ as a symbol disjoint from Σ with arity 0 and intended as the minimal element.

2. On the set of term trees over $\Sigma \cup \{\perp\}$, $T^\infty(\Sigma \cup \{\perp\})$, we introduce the following order: $t \leq t'$ if and only if either $t = \perp$ or $t = f t_1 \cdots t_{\varrho(f)}$ and $t' = f t'_1 \cdots t'_{\varrho(f)}$ for some t_i and t'_i, and for each $1 \leq i \leq \varrho(f)$ we have $t_i \leq t'_i$.

3. Thus, we define $\underline{CT^\infty(\Sigma)} = (T^\infty(\Sigma \cup \{\perp\}), \varphi)$, where φ is defined as $\varphi_{T^\infty(\Sigma)}$.

Proposition 2.1.6. *With this ordering, $CT^\infty(\Sigma)$ is a continuous Σ-algebra.*

Proof. See [JAGW77]. □

2.2 Higher-order Grammars and their Term Tree Languages

As mentioned in the introduction, we are interested in finite means of defining infinite structures, and in particular, infinite term trees. For the moment, however, we will introduce higher-order grammars, which define a term tree language, in other words, a set of trees. Furthermore, each tree in this set will be finite. Infinite trees will be discussed in the following section.

Definition 2.2.1. A ***higher-order grammar*** is a five-tuple $G = \langle N, \Sigma, V, S, \mathcal{R} \rangle$ such that

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

(ii) V is a finite set of typed variables.

(iii) Σ is a finite simply-typed alphabet of level at most 1. Thus, each term in Σ will be of type $o^n \to o$, for some $n \geq 0$. (As we know, one may also consider this as a ranked alphabet.)

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

$$Fx_1 \cdots x_n \; = \; r$$

where $F : (A_1, \cdots, A_n, o) \in N$, each $x_i : A_i \in V$, and r is a term in $\mathcal{T}^o(N \cup \Sigma \cup \{x_1, \cdots, x_n\})$. We say that F has ***formal parameters*** $x_1, \cdots, x_n$. Following [KNU01] we make the assumptions (w.l.o.g):

- If $F \in N$ has type $(A_1, \cdots, A_n, o)$ and $n \geq 1$, then $A_n = o$. Thus, each non-terminal has at least one level-0 variable. Note that this is not really a restriction – as this variable need not occur on the righthand side.
- S never occurs on the righthand side of a rewrite rule.
- In the case where the grammar has two or more rules with the non-terminal F on the lefthand side we assume (w.l.o.g.) that both rules have the same formal parameters in the same order.

As will become clearer later, none of these are really restrictions. However, we assume them as they greatly simplify the number of cases required in later proofs.

We say that the rewrite rule has *name* F and has level n just in case the type of its name has level n.

We say that the grammar G is of level n just in case n is the level of the rewrite rule that has the highest level.

We now define the single-step rewrite relation associated with the grammar G. We extend $\mathcal{R}$ to a family of binary relations $\rightarrow_G$ over $\mathcal{T}^o(N \cup \Sigma)$ by the rules:

- if $Fx_1 \cdots x_n = r$ is a rule in $\mathcal{R}$ where $x_i : A_i$ then for each $M_i \in \mathcal{T}^{A_i}(N)$ we have

$$FM_1 \cdots M_n \rightarrow_G r[\overline{M_i/x_i}].$$

Note that these substitutions are *simultaneous.*

- If $M \rightarrow_G M'$ then $(MN) \rightarrow_G (M'N)$ and $(PM) \rightarrow_G (PM')$ whenever the expressions in question are applicative terms.

Remark 2.2.2. Note that if $M \in \mathcal{T}^o(N \cup \Sigma)$ contains an occurrence of a non-terminal F, then, we can rewrite that occurrence if and only if F occurs in the context $\cdots (FM_1 \cdots M_n) \cdots$ for some $M_i : A_i$ and $Fx_1 \cdots x_n = r$ is a G-rule with each $x_i : A_i$. In other words, F must be applied to all of its arguments.

A ***reduction sequence*** is a (possibly infinite) sequence $S = T_0 \rightarrow_G T_1 \rightarrow_G T_2 \rightarrow_G \cdots$. We denote by $\twoheadrightarrow_G$ the reflexive and transitive closure of $\rightarrow_G$. In future we will drop the subscript G when it is understood from the context.

We are now in a position to define the term tree language of a grammar G.

Definition 2.2.3. Given a grammar, $G = \langle N, \Sigma, V, S, \mathcal{R} \rangle$, we define the tree language of G, $L(G)$ as follows:

$$L(G) = \{t \in T(\Sigma) \mid S \twoheadrightarrow t\}$$

In other words, $L(G)$ consists of all those finite terms consisting only of terminal symbols, and thus can be identified, as explained earlier, with finite trees over Σ.

Example 2.2.4. Consider the following grammar G where $N = \{S : o, F : (o, o, o)$, $V = \{x : o, y : o\}$, $\Sigma = \{f : (o, o, o), a : o, b : o\}$ and has the following production rules:

$$\begin{array}{rcl} S & = & Fab \\ Fxy & = & f(Fxy)(Fxy) \\ Fxy & = & x \\ Fxy & = & y \end{array}$$

It should not be difficult to see that the set of trees in the tree language, $L(G)$, of this grammar consists exactly of those trees such that all internal nodes are labelled by f and all leaf nodes are labelled by either a or b. Below we include an example of a derivation of $fa(fbb)$ (ignore the underlined expression for now):

$$S \rightarrow Fab \rightarrow \underline{f(Fab)(Fab)} \rightarrow fa(Fab) \rightarrow fa(f(Fab)(Fab)) \rightarrow fa(f(Fab)b) \rightarrow fa(fbb)$$

2.2.1 Modes of Derivation

Note that at each step we allow ourselves to substitute the righthand side of any nonterminal occurring in a sentential form provided it is applied to all of its arguments. For example, in the preceding derivation, in the underlined expression we have a choice of either rewriting the lefthand Fab or the righthand Fab.

A derivation is referred to as ***unrestricted*** if we do not in any way impose on the choices that can be made when we are about to perform a reduction.

In this section we define two further modes of derivation that will be of interest to us: innermost-outermost and outermost-innermost.

Definition 2.2.5. We say that $S = t_0 \rightarrow t_1 \rightarrow \cdots \rightarrow t_N \in T(\Sigma)$ is an ***innermost-outermost***[3] derivation if and only if for each $0 \leq i < N$ there exists a unary context $C_i[X]$ satisfying the following:

- There exists a production rule $Fx_1 \cdots x_n = r$ such that $C_i[Fs_1 \cdots s_n] = t_i$ for appropriate s_j.
- No s_i for $1 \leq i \leq n$ in the above contains a subexpression of the form $Hp_1 \cdots p_m$ where $Hy_1 \cdots y_m = r'$ is a production rule of the grammar.
- Lastly, $C_i[r[s_1/x_1, \cdots, s_n/x_n]] = t_{i+1}$.

Definition 2.2.6. We say that $S = t_0 \rightarrow t_1 \cdots \rightarrow t_N \in T(\Sigma)$ is an ***outermost-innermost***[4] derivation if and only if, for each $0 \leq i < N$ there exists a unary context $C_i[X]$ satisfying the following:

- There exists a production rule $Fx_1 \cdots x_n = r$ such that $C_i[Fs_1 \cdots s_n] = t_i$ for appropriate s_j.
- The "hole" X in $C_i[X]$ does not occur as a subexpression of a term $Hp_1 \cdots p_m$ where $Hy_1 \cdots y_m = r'$ is a production rule of the grammar.

[3]Also referred to as "inside-out" in the literature.
[4]Also referred to as "outside-in" in the literature.

- Lastly, $C_i[r[s_1/x_1, \cdots, s_n/x_n]] = t_{i+1}$.

We will abbreviate outermost-innermost to OI and similarly, innermost-outermost to IO.

Definition 2.2.7. Given a grammar G we denote the following languages:

$$\begin{aligned}
L_{unr}(G) &= L(G) = \{t \in T(\Sigma) \mid S \twoheadrightarrow t\} \\
L_{IO}(G) &= \{t \in T(\Sigma) \mid S = t_0 \to t_1 \to \cdots \to t_n = t \wedge \text{ derivation is IO}\} \\
L_{OI}(G) &= \{t \in T(\Sigma) \mid S = t_0 \to t_1 \to \cdots \to t_n = t \wedge \text{ derivation is OI}\}
\end{aligned}$$

Example 2.2.8. It should be quite clear that $L_{IO}(G) \subseteq L_{unr}(G)$ and similarly $L_{OI}(G) \subseteq L_{unr}(G)$, however, we now give an example to compare OI and IO languages for the following grammar. Consider the following grammar G with non-terminals $N = \{S : o, F : (o,o,o), G : o, H : (o,o,o)\}$, $V = \{x : o, y : o\}$, and terminals $\Sigma = \{f : (o,o,o), a : o, b : o\}$ and the following production rules:

$$\begin{aligned}
S &= Fab \\
Fxy &= G(Hxy) \\
Gx &= fxx \\
Hxy &= x \\
Hxy &= y
\end{aligned}$$

It should not be difficult to see that there are only finitely many derivations; which we list below:

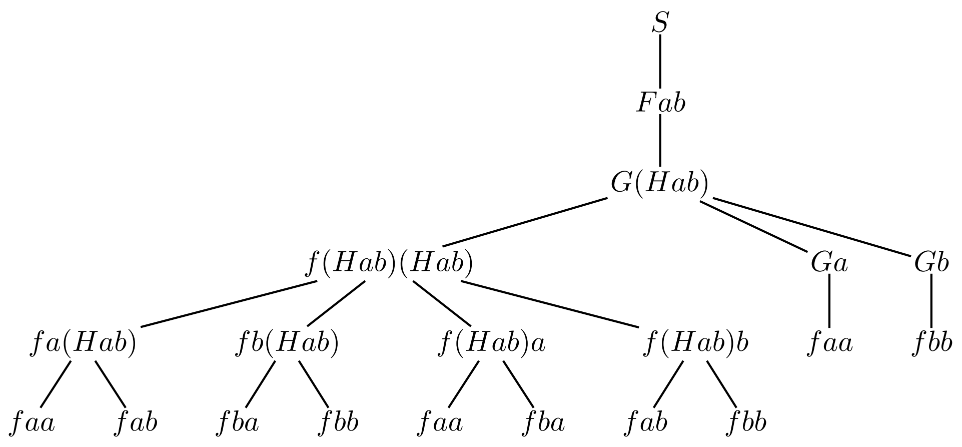


Figure 2.2: Derivations for G.

From this tree of all derivations it should be easy to see that:

$$\begin{aligned}
L_{unr}(G) &= \{faa, fab, fba, fbb\} \\
L_{OI}(G) &= \{faa, fab, fba, fbb\} \\
L_{IO}(G) &= \{faa, fbb\}
\end{aligned}$$

The Sufficiency of Outermost-Innermost Derivations

Note that in the preceding example we calculated three types of languages for a grammar: L_{unr}, L_{OI} and L_{IO}. This example, although very simple, illustrates two important properties of regarding the differences between OI and IO derivations.

- $L_{OI}(G) = L_{unr}(G)$. This is not a coincidence. For any grammar G, if $t \in T(\Sigma)$ is in $L_{unr}(G)$, then there exists an OI derivation of t.
- The above example also captures the "essence" of the difference between OI and IO derivations: namely that with IO derivations we must always choose and then copy, whereas with OI derivations we copy and then choose. This is aptly illustrated by the fact that the trees in L_{IO} (for this example) are such that both children of f are always the same. This is because we had to reduce the subterm Hab in $G(Hab)$ first, as opposed to reducing G first.

Theorem 2.2.9. *OI derivations are sufficient: for every $t \in L_{unr}(G)$, there exists an OI derivation $S = t_1 \to t_2 \to \cdots \to t_n = t$.*

Proof. Similar to standardization theorem for the λ-calculus. See [Dam82]. □

2.2.2 A Special Case: Monadic Terminal Alphabet

We now present a special case of higher-order grammars. Here, we impose some constraints on the terminal alphabet Σ:

- Σ contains a distinguished symbol $e : o$, to be understood as marking the end of a word.
- For all other $f \in \Sigma$, $f : (o, o)$, i.e. every symbol other than e is monadic in that it expects only one argument.

Note that any higher-order grammar G satisfying the above constraints will only produce monadic – or linear – trees, each of the from we where $w \in (\Sigma - \{e\})^*$.

In particular, $L_m(G)$ is isomorphic to the string language:

$$\{w \in (\Sigma - \{e\})^* | we \in L_m(G)\}$$

where m is the mode of derivation (unr, OI, or IO).

2.3 Higher-order Recursion Schemes and their Infinite Trees

2.3.1 Higher-order Recursion Schemes

We now introduce higher-order recursion schemes as a definitional device for (potentially) infinite term trees.

Definition 2.3.1. A ***higher-order recursion scheme*** is a higher-order grammar $G = \langle N, \Sigma, V, S, \mathcal{R} \rangle$ where $\mathcal{R}$ is *deterministic*. In particular, this means that for each $F \in N$ there exists precisely one production rule with F on the left hand side. We will often denote a recursion scheme by R instead of G.

Example 2.3.2. Let us consider the following recursion scheme R with non-terminals $N = \{S : o, F : ((o,o), o, o)\}$, variables $V = \{\varphi : (o,o), x : o\}$, terminals $\Sigma = \{f : (o,o,o), g : (o,o), a : o\}$ and the following production rules:

$$\begin{array}{rcl} S & = & Fga \\ F\varphi x & = & f(F\varphi(\varphi x))(\varphi x). \end{array}$$

Thus, appearance-wise, a higher-order recursion scheme is very much like a higher-order grammar. However, we will not treat it in the same way. Instead, our intent is that every higher-order recursion scheme defines exactly one (potentially infinite) term tree.

Definition 2.3.3. Given a higher-order recursion scheme $R = \langle N, \Sigma, V, S, \mathcal{R} \rangle$ we define an auxiliary higher-order grammar, G_R, referred to as the ***schematological tree grammar of*** R, and it is defined as follows, $G_R = \langle N, \Sigma \cup \{\perp\}, V, S, \mathcal{R}' \rangle$ where $\mathcal{R} \subseteq \mathcal{R}'$ and for each non-terminal $F \in N$ we introduce a production $Fx_1 \cdots x_m = \perp$ in $\mathcal{R}'$.

Example 2.3.4. Using the higher-order recursion above, its associated schematological tree grammar is given by G_R with non-terminals $\{S : o, F : ((o,o), o, o)\}$, variables $\{\varphi : (o,o), x : o\}$, terminals $\{f : (o,o,o), g : (o,o), a : o, \perp : o\}$ and the following production rules:

$$\begin{array}{rcl} S & = & Fga \\ S & = & \perp \\ F\varphi x & = & f(F\varphi(\varphi x))(\varphi x) \\ F\varphi x & = & \perp \end{array}$$

Given the associated schematological grammar of a recursion scheme we can now consider its tree language.

Example 2.3.5. With the above running example, we note that:

$$L_{unr} = \{\perp, f\perp(ga), f(f\perp(gga)))(ga), \cdots\} \tag{2.1}$$

Proposition 2.3.6. *Let R be a recursion scheme and G_R the associated schematological tree grammar, then $L(G_R)$ is a directed subset of $T^\infty(\Sigma \cup \{\perp\})$ in the continuous algebra $\underline{CT^\infty(\Sigma)}$.*

Proof. See [Dam82], for example. □

We are now in a position to define the semantics of a higher-order recursion scheme.

Definition 2.3.7. Given a higher-order recursion scheme R, we define the semantics of R to be $[\![R]\!]$, where $[\![R]\!]$ is a tree in $T^\infty(\Sigma \cup \{\perp\})$ and:

$$[\![R]\!] = \sqcup L_{unr}(G_R)$$

Example 2.3.8. Below is the infinite tree generated by the recursion scheme in the running example of this section.

Our definition of the unique term tree produced by a higher-order recursion scheme relies on the notion of the schematological tree grammar associated with the recursion scheme. This pattern of definition is borrowed from Damm [Dam82].

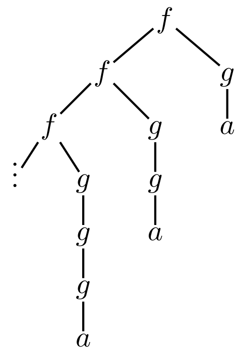


Figure 2.3: $[\![R]\!]$, the infinite term tree produced by the recursion scheme R.

It is possible to define the unique term tree generated by a higher-order recursion scheme without recourse to an intermediary higher-order grammar, and such a definition is presented by Knapik, Niwiński, and Urzyczyn [KNU01, KNU02]; and this is done by considering the limit of an *infinite and fair* reduction sequence. However, the two definitions amount to the same.

We find the separation of higher-order grammars and higher-order recursion schemes as presented in this chapter to be very useful for our purposes. Firstly, all key results presented here will be stated in terms of higher-order grammars. Then, the corresponding result for higher-order recursion schemes will often fall out as a corollary of the result for the associated schematological tree grammar. This is indeed preferable for the purposes of this thesis.

Remark 2.3.9. Note that it is conceivable that we could lift the restriction of determinism for our higher-order recursion schemes to allow a *class* of (potentially infinite) term trees. However, this will not be pursued here.

2.3.2 The Monadic Theory of the Infinite Tree Generated by a Higher-Order Recursion Scheme

We now consider how the infinite term tree $[\![R]\!]$ generated by a higher-order recursion scheme R is represented as a logical structure. Furthermore, we consider its monadic-second order theory. This section assumes a basic understanding of first-order logic such as presented in [vD83].

Monadic Second-order Logic

Definition 2.3.10. A ***relational vocabulary*** $\sigma = \langle R_1, \cdots, R_n \rangle$ consists of a finite ranked alphabet $\{R_1, \cdots, R_n\}$ where R_i has rank r_i. A ***relational structure*** over relational vocabulary σ is denoted by $\mathcal{U} = \langle U, R_1^{\mathcal{U}}, \cdots, R_n^{\mathcal{U}} \rangle$, where U – the universe – is a non-empty set, and for each $R_i \in \sigma$, $\mathcal{U}$ contains a relation $R_i^{\mathcal{U}} \subseteq \underbrace{U \times U \times \cdots \times U}_{r_i}$.

Remark 2.3.11. A relational structure over σ can be thought of as a σ-algebra.

Monadic second-order logic is an extension of first-order logic with quantification over set variables. Therefore, instead of merely having variables denoting elements, we also have variables over *sets* of elements, and of course, quantification over both types of variables. As sets are second-order relations, and furthermore monadic, the expression "monadic second-order logic" is justified (as opposed to polyadic relations, i.e. those of arity greater than 1).

This results in a very expressive logic, as we will see in a moment. If $\sigma = \langle R_1, \cdots, R_n \rangle$ is a relational vocabulary, the set of ***well-formed*** monadic second-order formulae over σ, denoted $MSOL[\sigma]$, is given by:

$$\varphi \ := \ R_i(x_1, \cdots, x_{r_i}) \mid x \in X \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi \mid \forall X.\varphi \mid \exists X.\varphi \mid \forall x.\varphi \mid \exists x.\varphi$$

where x (lowercase) denote variables over elements, and X (uppercase) denote variables over sets of elements.

Given an MSO formula φ, we denote the set of ***free variables***, $FV(\varphi)$, as follows:

$$\begin{array}{rcl} FV(R_i(x_1, \cdots, x_{r_i})) & = & \{x_1, \cdots, x_{r_i}\} \\ FV(x \in X) & = & \{x, X\} \\ FV(\varphi_1 \Box \varphi_2) & = & FV(\varphi_1) \cup FV(\varphi_2) \text{ where } \Box = \vee, \wedge \\ FV(\neg\varphi) & = & FV(\varphi) \\ FV(\forall v.\varphi) & = & FV(\varphi) - \{v\} \text{ where } v = x, X \\ FV(\exists v.\varphi) & = & FV(\varphi) - \{v\} \text{ where } v = x, X \end{array}$$

We say that $\varphi \in MSOL[\sigma]$ is a ***sentence*** if and only if it contains no free variables, in other words $FV(\varphi) = \emptyset$.

Let V be a countable set of both first-order $x, y, z, \cdots$ and second-order $X, Y, Z, \cdots$ variables. Given a universe U, a ***valuation function*** for V is a map $v : V \to U \cup 2^U$, where $v(x) \in U$ and $v(X) \subseteq U$.

Let α be a variable in V, and if α is first-order, then let $u \in U$, otherwise let $u \subseteq U$. Given a valuation v, we denote $v[u/\alpha]$ as the valuation that behaves exactly the same as v except with respect to the variable α. In particular, if β is a variable (either first- or second-order) then:

1. If $\beta \not\equiv \alpha$, then $v[u/\alpha](\beta) = v(\beta)$
2. If $\beta \equiv \alpha$, then $v[u/\alpha](\beta) = u$

The semantics of an MSO formula $\varphi \in MSOL[\sigma]$ where $\sigma = \langle R_1, \cdots R_n \rangle$ is given with respect to a relational structure over σ, $\mathcal{U} = (U, R_1^{\mathcal{U}}, \cdots R_n^{\mathcal{U}})$ and a valuation of the free variables in φ, $v : FV(\varphi) \longrightarrow U \cup 2^U$.

$$\begin{array}{rcl}
\mathcal{U}, v \models x \in X & \text{iff} & v(x) \in v(X) \\
\mathcal{U}, v \models R_i(x_1, \cdots, x_{r_i}) & \text{iff} & (v(x_1), \cdots, v(x_{r_i})) \in R_i^{\mathcal{U}} \\
\mathcal{U}, v \models \varphi_1 \wedge \varphi_2 & \text{iff} & \mathcal{U}, v \models \varphi_1 \text{ and } \mathcal{U}, v \models \varphi_2 \\
\mathcal{U}, v \models \varphi_1 \vee \varphi_2 & \text{iff} & \mathcal{U}, v \models \varphi_1 \text{ or } \mathcal{U}, v \models \varphi_2 \\
\mathcal{U}, v \models \neg\varphi & \text{iff} & \mathcal{U}, v \not\models \varphi \\
\mathcal{U}, v \models \forall X.\varphi & \text{iff} & \forall B \subseteq U.\, \mathcal{U}, v[B/X] \models \varphi \\
\mathcal{U}, v \models \exists X.\varphi & \text{iff} & \exists B \subseteq U.\, \mathcal{U}, v[B/X] \models \varphi \\
\mathcal{U}, v \models \forall x.\varphi & \text{iff} & \forall b \in U.\, \mathcal{U}, v[b/x] \models \varphi \\
\mathcal{U}, v \models \exists x.\varphi & \text{iff} & \exists b \in U.\, \mathcal{U}, v[b/x] \models \varphi
\end{array}$$

Example 2.3.12. A common representation for directed graphs consisting of vertices V and edges $E \subseteq V \times V$ is the relational structure (V, E), note that the relational vocabulary is given by $\langle\, E \,\rangle$ only. Let us consider the following closed MSO formula:

$$\begin{array}{l}
\exists X.\exists Y.\forall z.((z \in X \wedge \neg z \in Y) \vee (\neg z \in X \wedge z \in Y)) \\
\qquad \wedge(z \in X \rightarrow (\forall z'.E((z, z')) \rightarrow z' \in Y)) \\
\qquad \wedge(z \in Y \rightarrow (\forall z'.E((z, z')) \rightarrow z' \in X))
\end{array}$$

The sentence can be translated as saying that there are two sets X and Y of elements such that they form a partition, and if an element z is contained in one of them, then it has no edges to elements in the same set. The reader should be able to deduce that this sentence is satisfied only by a bipartite graph, in other words, 2-colourable. This is just one example of the expressive power of monadic second-order logic over such graphs.

Example 2.3.13. Let us now consider edge-labelled (with labels in finite alphabet A) and vertex-labelled (with labels in finite alphabet L) directed graphs. One common representation for such graphs is over the relational vocabulary $\sigma = \langle\, (E_a)_{a \in A}, (P_l)_{l \in L} \,\rangle$, where E_a has rank 2 for each $a \in A$ and P_l has rank 1 for each $l \in L$. Such a graph would then be represented by the structure $G = \{V, (E_a^G)_{a \in A}, (P_l^G)_{l \in L}\}$ where V is the set of vertices, and if $(u, v) \in E_a^G$ this denotes that there exists an edge from the vertex u to the vertex v labelled by a. Similarly if $u \in P_l^G$, this denotes that the vertex u has label l (it may have more than one label, as we have not insisted that $(P_l^G)_{l \in L}$ form a partition of V).

Using this logical representation of graphs, where the universe consists of the set of vertices, let us discuss some of the properties that we can and cannot express via MSO formulae:

1. *Reachability*, we can express the property that there exists a path from vertices u to v,
2. *Fairness along infinite paths*, we can describe the existence of an infinite path in the graph, such that if an a-edge is traversed infinitely often then so is a b-edge.
3. We *cannot count*, however. For example, it is impossible to express a property such as there exist twice as many nodes labelled by l as there are labelled with k.
4. We cannot express the property that a graph contains a *Hamiltonian circuit*. However, by a change in representation (see [Cou90]), where the universe now consists of both vertices *and* edges, this is indeed possible.

Monadic second-order logic is a remarkably expressive logic, and can define all the standard safety, liveness and fairness properties for graphs using the above representation. It is often considered as the "ultimate" verification logic in terms of expressibility for at least two reasons. Firstly, it subsumes all commonly used verification logics in use today, such as LTL, CTL, CTL* and even the modal μ-calculus. Secondly, it is often considered to be on the brink of decidability. Any non-trivial extensions to monadic second-order logic will often remove any chance of decidability. For an excellent discussion on monadic second-order logic and its applicability to verification see [Tho97].

Viewing Term Trees as a Logical Structure

A term tree $t : T \longrightarrow \Delta$ can be represented by the following logical structure:

$$\mathbf{t} = \langle\, T, (d_i^{\mathbf{t}})_{1\le i\le N}, (p_f^{\mathbf{t}})_{f\in\Delta}\,\rangle$$

where $N = \max\{\varrho(f) : f \in \Delta\}$, and $d_i^{\mathbf{t}}$ is binary for each $1 \le i \le N$, whereas each $p_f^{\mathbf{t}}$ is unary for each $f \in \Delta$. Hence, the relational vocabulary is $\langle\, (d_i^{\mathbf{t}})_{1\le i\le N}, (p_f^{\mathbf{t}})_{f\in\Delta}\,\rangle$.

For a given tree $t : T \longrightarrow \Delta$, the following must hold:

1. $d_i^{\mathbf{t}} = \{(u, ui) : ui \in T\}$
2. $p_f^{\mathbf{t}} = \{u : u \in T \ \wedge\ t(u) = f\}$

We can now reason about a Δ-tree in monadic second-order logic by considering monadic second-order logic formulae over $\langle\, (d_i^{\mathbf{t}})_{1\le i\le N}, (p_f^{\mathbf{t}})_{f\in\Delta}\,\rangle$.

Given a term tree $t : T \longrightarrow \Delta$ its ***monadic second-order theory*** is the set of *closed* formulae φ such that $\mathbf{t} \models \varphi$. Mathematically:

$$MTh(t) = \{\varphi : FV(\varphi) = \emptyset \ \wedge\ \mathbf{t} \models \varphi\}$$

Intuitively, $MTh(t)$ is the set of properties expressed as MSO sentences that hold for t. We are interested in the decidability of this set.

2.4 The Safety Restriction for Higher-order Grammars

The concept of *safety* will now be introduced. It is the central theme of this thesis. Safety is a syntactic restriction on higher-order grammars, and, as mentioned in the introduction, safety has, historically, been key to ensuring good algorithmic and behavioural properties of the resulting structure defined by the grammar. What this thesis aims to investigate is whether the restriction is actually *necessary* for these desirable properties.

Safety has appeared in more than one guise throughout the course of computer science. Indeed, as Knapik, Niwiński and Urzyczyn pointed out [KNU01, KNU02], it is very similar to the restriction of *derived types* introduced by Damm [Dam82]. A later section of this chapter will be devoted to outlining some of Damm's contributions as well as an in-depth comparison of safety and the restriction of derived types.

In this section, we will introduce safety using the definition provided in [KNU01, KNU02]. This is a technical definition, which, at a first glance may not give much insight into how it arose or how this impacts on the structures produced.

However, to remedy this, we present a second definition, based on the *safe* λ-calculus. This definition was developed jointly with Luke Ong and Klaus Aehlig. This definition introduces a fragment of the λ-calculus, whereby one of the key properties of all safe λ-terms is that no variable capture can occur when performing substitutions and therefore one need not perform α-conversions when performing β-reductions.

2.4.1 The Knapik et al. Formulation

Definition 2.4.1. Given a higher-order grammar G of type level n, we say that it is ***unsafe*** if and only if there exists a rewrite rule $Fx_1 \cdots x_m = r$, where each of the following three conditions hold:

1. r contains a subterm t of level k where $k > 0$,

2. t contains a parameter (one of the x_i's) of level strictly less than k.

3. t occurs unapplied, in other words, t occurs in the operand position.

Definition 2.4.2. A grammar is ***safe*** if it is not unsafe.

Example 2.4.3. Below is an example of an unsafe grammar, where $N = \{S : o, F : ((o,o),o,o,o)\}$, $\Sigma = \{f : (o,o,o), g : (o,o), a : o, b : o\}$, $V = \{\varphi : (o,o), x : o, y : o\}$ and the set of production rules is given below:

$$\begin{array}{rcl} S & = & Fgab \\ F\varphi xy & = & f(F(\underline{fx})(\varphi x)y)(\varphi y) \end{array}$$

It is unsafe precisely because of the underlined subexpression fx which is of type level 1 and is unapplied, yet it contains an occurrence of the formal parameter x which is of type level 0.

2.4.2 The Safe λ-Calculus

In this section we take a brief moment to consider an alternative formulation of the safety restriction. It is based on a subset of the simply-typed λ-calculus that we refer to as the ***safe λ-calculus***. A notion of the safe λ-calculus was formulated jointly with Luke Ong and Klaus Aehlig and an earlier version was first cited in a technical report [AdMO04]. As indicated in the introduction, it is not a formulation we will exploit particularly in this thesis, but we do feel that it offers some additional insight into why this restriction has lent itself so well to proving many good properties of safe higher-order grammars.

Organising Types

We recall the notion of simple types and homogeneity. Assuming

$$A = (\underbrace{A_{11}, \cdots, A_{1l_1}}_{\overline{A_1}}, \cdots, \underbrace{A_{r1}, \cdots, A_{rl_r}}_{\overline{A_r}}, o)$$

is homogeneous, we write

$$A = (\overline{A_1} \,|\, \cdots \,|\, \overline{A_r} \,|\, o)$$

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

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

whenever this makes sense for the types involved.

Safe λ-Calculus

The ***Safe λ-Calculus*** is a subsystem of the simply-typed λ-calculus. Typing judgments (or terms-in-context) are of the form:

$$\Gamma : k \vdash s : A$$

where, Γ is a *set* of typed *variables*, all of which are assumed to be homogeneously typed, k is a positive integer, s is a safe λ-term of type A. Also, it is to be understood that there exists a set of constants Δ containing homogeneously-typed symbols disjoint from Γ.

Intuitively, it can be read as follows, "We can construct the term s of type A using only constants from Δ, and free variables from Γ, where the free variables occurring in s must be of level k or greater, where $k = \mathsf{level}(A)$." In this sense, the k is redundant, as it will *always* be equal to $\mathsf{level}(A)$, however, we retain it for emphasis. From the point of view of characterising safe higher-order grammars, this should match our intuition as when we construct a level-n unapplied term, it should only contain variables of levels n and above.

Given a set of typed variables Γ, we write $\Gamma_{\geq k}$, for the restriction of Γ to variables of level k and above. Similarly, $\Gamma_{>k}$ is the restriction of Γ to the variable with levels strictly greater than k. Finally, $\Gamma_{=k}$ is the restriction of Γ to only those variables with level k. So, for example if $\Gamma = \{\Psi : ((o,o),o,o), \varphi : (o,o,o), x : o, y : o\}$, then $\Gamma_{\geq 1} = \{\Psi, \varphi\}, \Gamma_{>1} = \{\Psi\}$ and $\Gamma_{=1} = \{\varphi\}$.

Valid typing judgments of the system are defined by induction over the following rules, where Γ is a set of homogeneously typed variables, and Δ is a disjoint homogeneously typed alphabet of constants.

$$\frac{\Gamma : k \vdash s : A}{\Gamma \cup \Gamma' : k \vdash s : A}\ (wk)$$

where Γ' is a set of variables disjoint from those in Γ, and where $k = \mathsf{level}(A)$.

$$\frac{}{\Gamma \cup \{x : A\} : k \vdash x : A}\ (var)$$

where $k = \mathsf{level}(A)$.

$$\frac{}{\Gamma : k \vdash F : A}\ (const)$$

where $k = \mathsf{level}(A)$ and $F : A \in \Delta$.

$$\frac{\Gamma : l \vdash s : (\overline{B_1}|\overline{B_2}|\cdots|\overline{B_n}|o) \qquad \Gamma : l-1 \vdash t_1 : B_{11} \quad \cdots \quad \Gamma : l-1 \vdash t_N : B_{1N}}{\Gamma : m \vdash st_1 \cdots t_N : (\overline{B_2}|\cdots|\overline{B_n}|o)}\ (app)$$

where $\overline{B_1} = B_{11}, \cdots, B_{1N}$, and $l = \mathsf{level}((\overline{B_1}|\overline{B_2}|\cdots|\overline{B_n}|o))$, and $m = \mathsf{level}((\overline{B_2}|\cdots|\overline{B_n}|o))$.

$$\frac{\Gamma : m \vdash s : (\overline{B_1}|\overline{B_2}|\cdots|\overline{B_n}|o) \qquad \Gamma_{\geq m} : m-1 \vdash t_1 : B_{11} \quad \cdots \quad \Gamma_{\geq m} : m-1 \vdash t_j : B_{1j}}{\Gamma : m \vdash st_1 \cdots t_j : (\overline{B}|\overline{B_2}|\cdots|\overline{B_n}|o)}\ (app+)$$

where $\overline{B_1} = B_{11}, \cdots, B_{1j}, \overline{B}$ and $m = \mathsf{level}((\overline{B}|\overline{B_2}|\cdots|\overline{B_n}|o)) = \mathsf{level}((\overline{B_1}|\overline{B_2}|\cdots|\overline{B_n}|o))$.

Let us consider the above equation. It must be employed should we have a level-m term s and wish to create an applicative term $st_1 \cdots t_j$ also of level m (meaning that we have not applied s to *all* of its level-$(m-1)$ arguments) that may potentially occur unapplied. Based on the Knapik et al. characterisation we know we cannot allow free variables to occur in $st_1 \cdots t_j$ unless they are of level m or greater, hence the restriction $\Gamma_{\geq m} : m-1 \vdash t_i : B_{1i}$ for $i = 1, \cdots, j$

Finally, for abstractions we have:

$$\frac{\Gamma : k \vdash s : A \qquad \Gamma_{\geq k} = \Gamma_{>l} \cup \Gamma_{=l} \qquad \Gamma_{=l} = \{x_1 : B_{11}, \cdots, x_n : B_{1n}\}}{\Gamma - \Gamma_{=l} : l+1 \vdash \lambda x_1 \cdots x_n.s : (B_{11} \cdots B_{1n}|A)}\ (abs)$$

$k = \mathsf{level}(A)$. Again, this merits some explanation. Suppose we have a term $s : A$ of type level k, defined with respect to the set of variables in Γ. When forming an abstraction with body s, we are forced to abstract all those variables in Γ of level l, where l is the lowest level of a variable occurring in Γ of level greater than (or equal to) k.

As a summary note, in the safe λ-calculus, when constructing λ-abstractions, all variables of the relevant lowest level must be abstracted; when constructing applications, the operator-term must be applied to all operand terms, or, in the case that it is not, no variable of level less than the level of the resulting term must be included.

Examples

Example 2.4.4. Let $\Gamma = \{\varphi : (o,o,o), x : o, y : o\}$ and the set of constants Δ is given by $\{F : ((o,o),o,o,o)\}$.

$$\frac{\dfrac{\dfrac{}{\Gamma : 2 \vdash F : ((o,o),o,o,o)}\ (const) \qquad \dfrac{}{\Gamma : 1 \vdash \varphi : (o,o)}\ (var)}{\Gamma : 1 \vdash F\varphi : (o,o,o)}\ (app) \qquad \dfrac{}{\Gamma : 0 \vdash x : o}\ (var) \qquad \dfrac{}{\Gamma : 0 \vdash y : o}\ (var)}{\Gamma : 0 \vdash F\varphi xy : o}\ (app)$$

Example 2.4.5. Let $\Gamma = \{\psi : (o,o,o), x : o, y : o\}$ and the set of constants Δ is given by $\{G : ((o,o),o,o), a : o\}$. As shorthand, we will write A as shorthand for $((o,o),o,o)$ and C as shorthand for (o,o).

$$\frac{\dfrac{\dfrac{}{\Gamma : 2 \vdash G : A}\ (const) \qquad \dfrac{\dfrac{}{\Gamma : 1 \vdash \psi : C}\ (var) \qquad \dfrac{}{\{\psi\} : 0 \vdash a : o}\ (const)}{\Gamma : 1 \vdash (\psi a) : (o,o)}\ (app+)}{\Gamma : 1 \vdash G(\psi a) : (o,o)}\ (app) \qquad \dfrac{}{\Gamma : 0 \vdash x : o}\ (var)}{\Gamma : 0 \vdash G(\psi a)x : o}\ (app)$$

Properties

Lemma 2.4.6. *Let $\Gamma : k \vdash s : A$, where $\mathsf{level}(A) = k$, be a valid typing judgment of the safe λ-calculus. The following two properties hold:*

1. *A is a homogeneous simple type.*

2. *If $x : B$ is a free variable occurring in s, then $\mathsf{level}(B) \geq A$.*

Proof. Straightforward induction on the last typing rule used. □

Lemma 2.4.7. *Let $\Gamma : k \vdash s : A$, where $\mathsf{level}(A) = k$, be a valid typing judgment of the safe λ-calculus. Suppose that the lowest level of a free variable occurring in s is M, where $M > k$. Then, for all m where $M \geq m \geq k$:*

$$\Gamma_{\geq m} : k \vdash s : A$$

is also a valid typing judgment of the safe λ-calculus.

Proof. Straightforward induction on the last typing rule used. □

Lemma 2.4.8. *Let $\Gamma : k \vdash \lambda \vec{x}.s : A$, where $\mathsf{level}(A) = k$, be a valid typing judgment of the safe λ-calculus. If φ is a variable occurring in s where $\mathsf{level}(\varphi) < \mathsf{level}(x)$, then φ occurs bound in s.*

Proof. Recall the rule for creating an abstraction:

$$\frac{\Gamma : k \vdash s : A \qquad \Gamma_{\geq k} = \Gamma_{>l} \cup \Gamma_{=l} \qquad \Gamma_{=l} = \{x_1 : B_{11}, \cdots, x_n : B_{1n}\}}{\Gamma - \Gamma_{=l} : l+1 \vdash \lambda x_1 \cdots x_n.s : (B_{11} \cdots B_{1n} | A)} \; (abs)$$

By Lemma 2.4.6(ii), we know that s in $\Gamma : k \vdash s : A$ can contain variables of level greater than or equal to k. However, because $\Gamma_{\geq k} = \Gamma_{>l} \cup \Gamma_{=l}$, the lowest level of a variable that might actually free occur in s is of type level l, where $l \geq k$. Hence if φ occurs in s and $\mathsf{level}(\varphi) < \mathsf{level}(x) = l$ for some $x \in \vec{x}$, it must be bound. □

What does "safe" mean?

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

$$(\boldsymbol{\lambda} x.s)[t/y] \overset{\text{def}}{=} \boldsymbol{\lambda} z.((s[z/x])[t/y]) \quad \text{where "}z\text{ is fresh"}$$

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

Lemma 2.4.9. *In the safe λ-calculus, there is no need to rename bound variables afresh when performing substitution*

$$s[t_1/x_1, \cdots, t_n/x_n]$$

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

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

- Case 1 : $\mathsf{level}(x) < \mathsf{level}(\varphi)$. This is impossible; note that it must be the case that $\mathsf{level}(\varphi) = \mathsf{level}(t)$, and as t is a safe λ-term it cannot contain any free variables of level less than $\mathsf{level}(\varphi)$ (Lemma 2.4.6).
- Case 2 : $\mathsf{level}(x) > \mathsf{level}(\varphi)$. Again, this is impossible; by Lemma 2.4.8, as φ has level strictly less than that of x, φ must occur bound in s'.
- Case 3 : $\mathsf{level}(x) = \mathsf{level}(\varphi)$. Here, we need to consider the formation rule for λ-abstractions. Note that when creating $\lambda\vec{x}.s''$, such that $\lambda x_1.\lambda x_2.\cdots.\lambda x_j.\lambda x.s' = \lambda\vec{x}.s''$, then φ, being of the same type level as x appears free in the expression s'' but $\varphi \equiv x_k$ for some k and therefore is bound, so substitution does not occur.

$\square$

Safe Higher-order Grammars

Consider the following definition, let G be a level-n grammar:

Definition 2.4.10. A rewrite rule $Fx_1 \cdots x_n = r$ is ***safe'*** if we have that

$$\{x_1 : A_1, \cdots, x_n : A_n\} : 0 \vdash r : o$$

is a valid typing judgement of the safe λ-calculus (without abstraction), where r is constructed using symbols from $N \cup \Sigma$ as constants. Otherwise, the rule is ***unsafe'***.

Definition 2.4.11. A grammar is safe' if and only if all of its rewrite rules are safe'.

λ-abstractions

In the context of higher-order grammars, there is nothing to be gained by allowing λ-abstractions. Consider, for example, a rewrite rule of the following form:

$$Fx_1 \cdots x_n = \cdots (\lambda\vec{y}.s) \cdots$$

where $\lambda\vec{y}.s$ is a "largest" abstraction, in the sense that, for example, it does not occur in a unary context $C[X] = \cdots (\lambda y'.X) \cdots$, and s is not an abstraction. Suppose that $\lambda\vec{y}.s : (B_1|B_2|\cdots|B_n|A)$, where $s : A$. Then this can be replaced by adding the following rewrite rule:

$$G\vec{y}\vec{z} = s\vec{z}$$

where $\vec{z} = z_1 : A_1, z_2 : A_2, \cdots, z_m : A_m$ and $A = (A_1, A_2, \cdots, A_m, o)$, and rewrite the above production rule as:

$$Fx_1 \cdots x_n = \cdots G \cdots$$

Equivalence with the Knapik et al. Formulation

Finally, we show the equivalence of this formulation via the safe λ-calculus and the one that is presented in Knapik, Niwińiski and Urzycyzn in [KNU01, KNU02].

In order to prove this, we will first need the following technical definition.

Definition 2.4.12. Let $s \in \mathcal{T}^A(\Sigma \cup N \cup V)$, where Σ and N are treated as homogeneously typed constants, and V is a set of homogeneously typed variables. The set of ***maximal subterms*** of s, $M(s)$, the smallest set S such that:

1. $s \in S$
2. If $s = \$t_1 \cdots t_n$, where $\$ \in \Sigma \cup N \cup V$ then let j be the smallest integer such that $t_j, t_{j+1}, \cdots, t_n$ all have the same level. Then $M(\$t_1 \cdots t_{j-1}) \subseteq S$ and also $M(t_i) \subseteq S$ for $i = j, \cdots, n$.

Example 2.4.13. Consider $G\psi^1(\psi^2 a)xy$. Note that ψ^1 and ψ^2 are the same variable, but we have superscripted them to differentiate between occurrences.

level-0 maximal subterms	$\{G\psi^1(\psi^2 a)xy, a, x, y\}$
level-1 maximal subterms	$\{\psi^1, (\psi^2 a), \psi^2\}$
level-2 maximal subterms	$\{G\}$

Note that the set of maximal subterms is, in general, a subset of the set of subterms of a term, for example, in the above neither $G\psi^1$ nor $G\psi^1(\psi^2 a)x$ are maximal subterms.

Remark 2.4.14. Although the definition of maximal subterms may seem rather arbitrary, on closer inspection the reader will note that it reflects the terms that will appear at each node of a proof tree in the (attempted) construction of a safe λ-term.

Lemma 2.4.15. *Let $\Gamma : k \vdash s : B$ be a valid typing judgment of the safe λ-calculus where s is an applicative term. Let τ be any proof tree for this typing judgement. $\Gamma' : l \vdash t : A$ is the conclusion of a node in this tree if and only if $t \in M(s)$.*

Proof. A trivial induction on the structure of s. □

Lemma 2.4.16. *Let Δ be a set of homogeneously typed symbols and let s be an applicative term over Δ. If t is a subterm of s and t occurs unapplied, then $t \in M(s)$.*

Proof. Again, a straightforward induction on the structure of s. □

Proposition 2.4.17. *Let $G = \langle N, \Sigma, V, S, \mathcal{R} \rangle$ be grammar and suppose $Fv_1 \cdots v_n = r$ is in $\mathcal{R}$:*

$$\{v_1, \cdots, v_n\} : 0 \vdash r : o$$

is a valid typing judgement (without abstractions) if and only if r (the rule) is safe in the sense of Knapik et al.

Proof. Suppose that $\{v_1, \cdots, v_m\} : 0 \vdash r : o$ is a valid typing judgement, we need to show that there is no subterm t of level k such that t contains a parameter v_i where $\mathsf{level}(v_i) < k$, and t occurs unapplied. Lemma 2.4.6 (ii) shows that for each node of the proof tree: if $\Gamma : k \vdash s : A$, then s contains only free variables of level greater that on equal to k. In particular, this is true of those that correspond to unapplied terms in s and the result now follows from Lemmas 2.4.15 and 2.4.16.

For the converse, we need to show that we can construct a proof tree so that

$$\{v_1, \cdots, v_m\} : 0 \vdash r : o$$

is a valid typing judgement. We thus show that for any *maximal* subterm s of type B (and therefore this holds for r itself) that :

$$\{v_1, \cdots, v_m\} : k \vdash s : B$$

where $\mathsf{level}(B) = k$.

Let s be a maximal subterm of r; the proof follows via an induction over the structure of s. We prove only the case where s is an applicative term; the others are straightforward. Then s may be written as $\$t_1 \cdots t_n$, where $\$ \in \Sigma \cup N \cup V$. Let j be the smallest integer such that $t_j, t_{j+1}, \cdots, t_n$ all have the same level. Suppose that $\$t_1 \cdots t_{j-1} : (\overline{B}|A)$. Then, either $\$t_1 \cdots t_n : A$ or $\$t_1 \cdots t_n : \overline{B'}|A$, where $\overline{B} = B_1, B_2, \cdots, B_{j-1}, \overline{B'}$. We prove the hypothesis for the second case (as it is straightforward to prove it for the first).

By the induction hypothesis, we know that:

$$\{v_1, \cdots, v_m\} : k \vdash \$t_1 \cdots t_{j-1} : (\overline{B}|A)$$

where $k = \mathsf{level}((\overline{B}|A))$. Again, by the induction hypothesis we also know that for each $i = j, \cdots, n$ the following holds:

$$\{v_1, \cdots, v_m\} : l \vdash t_i : B_i$$

where $l = \mathsf{level}(B_i)$ (note that of course l is the same for each i), and that it must be the case that $k = l + 1$. However, the following is also true for each $i = j, \cdots, n$:

$$\{v_1, \cdots, v_m\}_{\geq k} : l \vdash t_i : B_i$$

because as s occurs unapplied and is of level k, then no variable occurring in s can be of type level k or less – this is guaranteed by the safety assumption. So, by Lemma 2.4.7, we get the above. And the result holds.

□

2.5 Machine Characterisations of Higher-Order Grammars

In this section we introduce higher-order pushdown automata as machine characterisations of *safe* higher-order grammars. In Chapter 3 we will introduce pointer machines as machine characterisations of unrestricted higher-order grammars; unrestricted in the sense that we do not assume them to be safe. Pointer machines are due to Colin Stirling.

2.5.1 Higher-order Pushdown Automata

Higher-order pushdown automata are a well-known machine model first defined in 1976 by Maslov [Mas76] as a generalisation of the pushdown automaton. In his setting, a higher-order pushdown automaton (or iterated stack automaton as it was then known) defined a word language.

Here, we adjust the definition (in a similar vein to that of Knapik, Niwiński and Urzyczyn) so that instead of accepting merely a word language, it accepts a set of finite term trees, in other words: a term tree language.

Definition

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

$$\begin{array}{rcl} \mathsf{push}_1(a)\,[a_m, \cdots, a_1] & = & [a, a_m, \cdots, a_1] \\ \mathsf{pop}_1\,[a_m, a_{m-1}, \cdots, a_1] & = & [a_{m-1}, \cdots, a_1] \end{array}$$

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

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

In addition we define

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

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

Definition 2.5.1. A ***level-n pushdown automaton*** (or nPDA for short) is a 6-tuple $\langle Q, \Sigma, \Gamma, \delta, q_0, \bot \rangle$ where:

(i) Q is a finite set of states, $q_0 \in Q$ is the start state.

(ii) Σ is the finite input alphabet containing symbols of level at most 1.

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

(iv) $\delta \subseteq Q \times \Gamma \to ((Q \times \mathsf{Op}_n) \cup \mathsf{TreeOp}_n)$ is the transition relation. Here, $\mathsf{TreeOp}_n = \{f(p_1, \cdots, p_{\varrho(f)}) : f \in \Sigma \wedge p_1, \cdots, p_{\varrho f} \in Q\}$.

A ***configuration*** is denoted by a pair (q, s) where $q \in Q$ and s is an n-store, with initial configuration given by $(q_0, \bot_n)$. We denote by C the set of all configurations. Given a configuration (q, s) we say that $(q, s) \to (p, s')$ if $(q, \mathsf{top}_1(s), (p, \theta)) \in \delta$ where $(p, \theta) \in Q \times \mathsf{Op}_n$ and $\theta(s) = s'$. We denote by $\to^*$ the reflexive and transitive closure of $\to$.

Like a higher-order grammar of terminal alphabet Σ, a higher-order pushdown automaton defines a set of terms (identified with finite term trees) over Σ. Before we can define the term-tree language of a higher-order pushdown automaton we require the following auxiliary definitions. The first definition is applicable to trees in general and will be used later in this thesis; the latter is specific to higher-order pushdown automata.

Definition 2.5.2. Let $t : Dom(t) \to \Sigma \cup C$ be a (potentially infinite) term tree over the ranked alphabets Σ and C. Furthermore, let C be such that all symbols in C have rank 1. We define the ***collapse of*** t ***onto*** Σ as the tree $t^\Sigma : T \to \Sigma \cup \{\bot\}$ defined as follows[5]. In fact, t^Σ will be defined along with a partial mapping $I : T \to Dom(t)$ in such a way that $t^\Sigma(w) = t(I(w))$ when $I(w)$ is defined. In other words, $I(w)$ defines a node in the domain $Dom(t)$ and t^Σ gives the corresponding label of this node.

1. First, consider the *shortest* path $\pi_1 \cdots \pi_n$ in t from the root such that $t(\pi_n) \in \Sigma$. Clearly there is at most one such shortest path as nodes labeled by symbols in C have precisely one child. If this path exists, we set $I(\epsilon) = \pi_n$ and $t^\Sigma(\epsilon) = t(\pi_n)$. If no such path exists then we leave $I(\epsilon)$ as undefined and set $t^\Sigma = \bot$.

2. Now suppose that $I(w)$ and $t^\Sigma(w)$ are *both* defined and that $I(w) = v$ and $t(v) = f$ (and so $t^\Sigma(w) = f$). Let $Dir(f) = \{x_1, \cdots, x_n\}$. For each $x_i \in Dir(f)$ we consider the shortest path from vx_i in t ending in a node labeled by a symbol in Σ. For the same reason as above, there exists at most one shortest path and suppose it terminates in the node u_i, then we set $I(wx_i) = u_i$ and $t^\Sigma(wx_i) = t(u_i)$. If no such path exists then we leave $I(wx_i)$ undefined and set $t^\Sigma(wx_i) = \bot$.

In the following, let C be the set of configurations for a level-n pushdown automaton. We will interpret C as ranked alphabet, each symbol of which has rank 1 with direction [1].

Definition 2.5.3. A ***total run tree*** of a level-n pushdown automaton $\langle Q, \Sigma, \Gamma, \delta, q_0, \bot \rangle$ is a finite tree $r : T_r \to C \cup \Sigma$ that satisfies the following.

- $r(\epsilon) = (q_0, \bot_n)$
- If v in T_r and $r(v) = (q, s)$ then $\delta(q, \mathsf{top}_1(s)) \neq \emptyset$. Furthermore, one of the following must hold:
  1. $(p, \theta) \in \delta(q, \mathsf{top}_1(s))$ where $(p, \theta) \in Q \times \mathsf{Op}_n$, and we have that $r(v1) = (p, \theta(s))$;
  2. Or $f(p_1, \cdots, p_{\varrho f}) \in \delta(q, \mathsf{top}_1(s))$ and we have that $r(v1) = f$ and $r(v1j) = (p_j, s)$ for $1 \leq j \leq \varrho f$. In particular, if f has arity 0, then $v1$ has label f and is a terminal node.
- Finally, if v is a terminal node in r then $r(v) \in \Sigma$.

Definition 2.5.4. Let t be a finite Σ-tree, we say that it is accepted by the level-n pushdown automaton A if and only if there exists a total run tree r of A with collapsed tree (onto Σ) equal to t.

[5]Here, $\bot$ is a symbol of rank 0. Not to be confused with the use of the bottom-of-store symbol $\bot$.

Example 2.5.5. Let $A = \langle Q, \Sigma, \Gamma, \delta, q_0, \perp \rangle$ be a level-1 pushdown automaton where δ is given below (and therefore, suitable Q, Σ, Γ may be inferred).

$$\begin{array}{rcl}
\delta(q_0, \perp) & = & \{(q_0, \mathsf{push}_1(X))\} \\
\delta(q_0, X) & = & \{(q_0, \mathsf{push}_1(X)), f(q_1, q_2), f(q_2, q_2)\} \\
\delta(q_1, X) & = & \{g(q_3)\} \\
\delta(q_1, \perp) & = & \{a\} \\
\delta(q_3, X) & = & \{(q_1, \mathsf{pop}_1)\} \\
\delta(q_2, X) & = & \{(q_2, \mathsf{push}_1 X), a\}
\end{array}$$

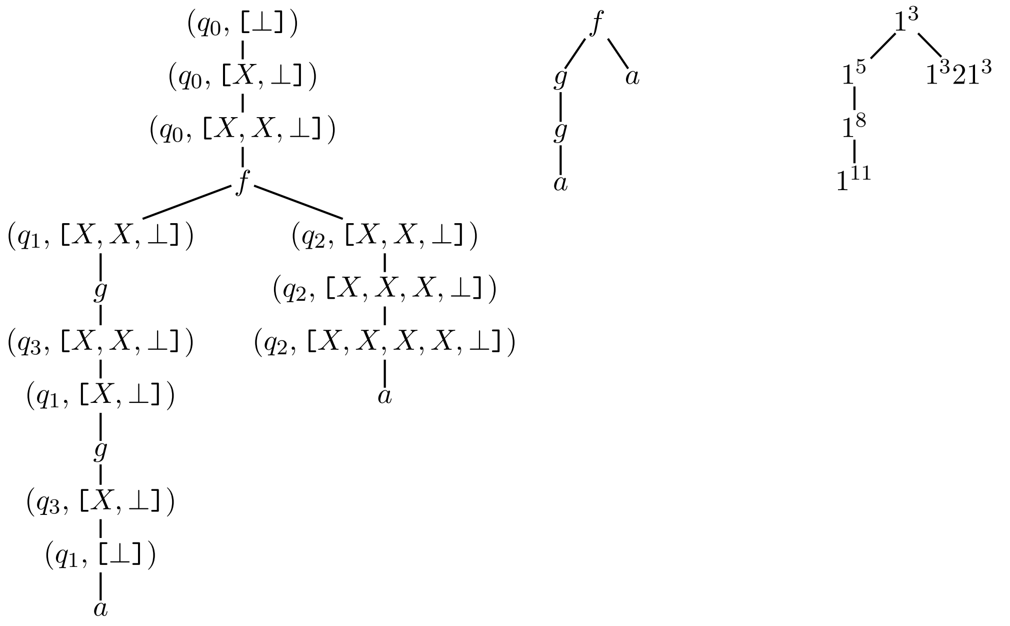


Figure 2.4: From left to right: A total run tree, r, the corresponding trees r^{Σ} and I.

Definition 2.5.6. Let $A = \langle Q, \Sigma, \Gamma, \delta, q_0, \perp \rangle$ be a level-n pushdown automaton. The language of A, denoted by $L(A)$, is the set of Σ-trees accepted by A.

Definition 2.5.7. A higher-order pushdown automaton is said to be ***deterministic*** if and only if $|\delta(q, Z)| \leq 1$ for all $q \in Q$ and $Z \in \Gamma$. Otherwise it is said to be ***non-deterministic***.

Remark 2.5.8. Unless otherwise specified, we assume that a higher-order pushdown automaton is non-deterministic.

Equivalence with Safe Higher-order Grammars

The following seminal result will be exploited many times throughout this thesis:

Theorem 2.5.9. *Let $L \subseteq T(\Sigma)$ be a term tree language. L is the language of a level-n safe OI-grammar if and only if L is the language of a level-n pushdown automaton.*

Proof. See Knapik et al. [KNU02]. □

Remark 2.5.10. Strictly, Knapik et al.'s proof [KNU02] shows that T is the (infinite) term tree produced by a level-n safe *recursion scheme* if and only if T is the term tree accepted by a level-n deterministic pushdown automaton (over term trees). However, the adaptation of their proof from recursion schemes to grammars requires no modification whatsoever. Whilst we do not replicate the proof details here, both directions of the proof will be considered in the following chapter (in separate contexts). Briefly, Knapik et al. show how to construct a level-n PDA given a safe level-n recursion schemes as well as the converse.

Remark 2.5.11.

1. Note that Knapik et al.'s is not the sole incarnation of this result. The first appearance of such a result can be attributed to Damm and Goerdt [DG86] who showed this was indeed true for the OI string language case (in other words, imposing the restriction that Σ contains a special end of word character $e : o$ and all other symbols in Σ have type (o, o)). Only later was this result proved for higher-order recursion schemes by Knapik, Niwiński and Urzyczyn. The statement of the above theorem therefore offers nothing new except for the fact that we have recast the result in terms of grammars as opposed to recursion schemes.

2. Knapik et al.'s proof can be considered more general than Damm and Goerdt's methodology in at least two ways. First, the former gives us the machinery to handle the more general structures of trees (of which strings are a special case). Secondly, Damm and Goerdt's result requires the conversion of safe grammars into a particular normal form for one direction of the proof.

Higher-order Pushdown Automata for String Languages

We take a moment to reconcile the definition of our higher-order pushdown automaton (Definition 2.5.1) with the more conventional definition found in literature.

In [Mas74, Mas76, DG86, Eng91] higher-order pushdown automata are viewed as acceptors of *word languages* and there is often a notion of an alphabet.

Definition 2.5.12. A word-language accepting level-n pushdown automaton is given by a tuple $\langle Q, \Sigma, \Gamma, \delta, q_0, \bot, F \rangle$ where

1. Q is a finite set of states where q_0 is the initial state and $F \subseteq Q$ is a set of final states.

2. Σ is a finite word alphabet in the conventional sense.

3. Γ is a finite stack alphabet with $\bot \in \Gamma$ as the bottom-of-store symbol.

4. Finally, $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \times Q \times \mathsf{Op}_n$ is the transition function. Note that there are no tree operations $f(p_1, \cdots p_r)$.

In this setting, the higher-order pushdown automaton operates on an input word in Σ^*. Thus, a total configuration is given by a triple (q, w, s) where q denotes the current state, $w \in \Sigma^*$ denotes the portion of the input word that remains to be read, and s is the current n-store. We say that $(q, aw, s) \to (p, w, s')$ if and only if $(p, \theta) \in \delta(q, a, \mathsf{top}_1(s))$ where $\theta(s) = s'$. Note that we may have either $a \in \Sigma$ (so one letter of the input word is consumed) or $a = \epsilon$ denoting a silent transition where the input is left unchanged but changes to the stack and state may occur. We denote by $\twoheadrightarrow$ the reflexive and transitive closure of $\rightarrow$. The word language ***accepted*** by the pushdown automaton is given by the set of words

$$\{w \in \Sigma^* \mid \exists s.\exists q \in F.(q_0, w, \bot_n) \twoheadrightarrow (q, \epsilon, s)\}.$$

In other words, it consists of all those words over Σ such that after consuming all letters of input we can reach a final state.

Remark 2.5.13. It is also possible to define acceptance by empty store, in which a word $w \in \Sigma$ is accepted if and only if after reading all input letters of w we reach a configuration $(q, \epsilon, \bot_n)$, in other words, an empty stack is reached. The equivalence between final state acceptance (as above) and that of empty store is a well known and simple result; we omit it here. One such proof may be constructed as a direct generalisation of the proof for the level-1 case, as given in [HU79].

As was indicated earlier, our definition of a higher-order grammar (as a definition of a tree language) defines a word language when $\Sigma = \Sigma' \cup \{e\}$ where all symbols in Σ' are of type (o, o) and $e : o$. This defines a term tree language $\{we : we \in L(A)\}$ that we can identify with a word language $\{w : we \in L(A)\}$. We thus take a moment to reconcile the following:

Proposition 2.5.14. *Let $A = \langle Q, \Sigma, \Gamma, \delta, q_0, \bot \rangle$ be a tree language accepting level-n pushdown automaton with $\Sigma = \Sigma' \cup \{e\}$ where all symbols in Σ' are of type (o, o) and $e : o$. Then L is the language of A if and only if $L' = \{w : we \in L\}$ is the language of A' where A' a level-n word-accepting pushdown automaton $A' = \langle Q', \Sigma', \Gamma, \delta', q'_0, \bot', F' \rangle$ for suitable Q', F' and δ'.*

Proof. It should be clear to see that we identify final states and the outputting of the end-of-word character e. Such a transformation is outlined below.

In one direction, we create a single final accepting state, q_f which has no outgoing transitions. First, every transition in δ of A of the form $(q, a, g(p))$ is transformed into a transition $(q, g, a, p, \mathsf{id})$, and transitions of the form $(q, a, (p, \sigma))$ are transformed into transitions $(q, \epsilon, a, p, \sigma)$. Finally, every transition (q, a, e) is transformed into $(q, \epsilon, a, q_f, \mathsf{id})$. It is routine to check that this indeed suffices.

In the other direction, each transition $\delta(q, g, a)$ must be converted into two steps: first consumption of the letter g translates into outputting the letter g, then the non-deterministic choice of which transition in $\delta(q, g, a)$ to take. Finally, we ensure that $e \in \delta(q_f, a)$ for any accepting state q_f. ☐

2.6 Higher-order Grammars Through the Ages

To conclude this chapter we present an outline of the evolution of higher-order grammars, schemes, and related concepts. This is not, however, merely a historical interlude before the next chapter; it can be thought of as a section summing up key results in the field in approximately chronological order; and therefore allowing us to put into context our own results.

This section is divided into four subsections. First we will consider the original context in which higher-order recursion schemes were introduced: the study of programming language semantics and how this led to the study of higher-order grammars as generators of term trees. Most of the results listed here are due to Damm [Dam82] and concern his level-n tree grammars.

The next subsection is dedicated to understanding the relationship between Damm's definition of a level-n tree grammar and our own definition of a level-n grammar. We will see that a level-n tree grammar as defined by Damm corresponds to a level-n *safe* grammar as defined here. Thus, all his results pertaining to level-n tree grammars are true for our level-n safe grammars.

In the third subsection we consider developments in level-n pushdown automata; and these can be thought of as having been developed in "parallel" to those on higher-order grammars. We culminate with the statement of the result showing the equivalence between Damm's level-n grammars (or safe level-n grammars using our definition) with level-n pushdown automata, therefore uniting these three subsections together.

Finally, with the fourth subsection we focus on recent verification-oriented results.

2.6.1 Semantics of Programming Languages

Higher-order recursion schemes as introduced in Definition 2.3.1, are a natural representation of higher-order recursive programs. As an informal example, let us consider the following functional program written in Haskell:

```
apply :: (int -> int) -> int -> int
apply f x = fx

factorial :: int -> int
factorial x = if (x == 0) then 1 else x*(factorial (x-1))

axiom :: int
axiom = apply factorial 5
```

This can be translated into the following higher-order recursion scheme S with $\Sigma = \{c : (o,o,o,o), m : (o,o,o), p : (o,o), \underline{1} : o, a : o\}$.

$$\begin{array}{rcl} S & = & AFa \\ Fx & = & cx\underline{1}(mx(F(px))) \\ A\varphi x & = & \varphi x \end{array}$$

The semantics of this higher-order recursion scheme, $[\![S]\!]$, is the infinite Σ-tree depicted in Figure 2.5. Note that it corresponds to the uninterpreted syntax tree of the functional program if we interpret the functions over the domain of the positive integers with the following semantics:

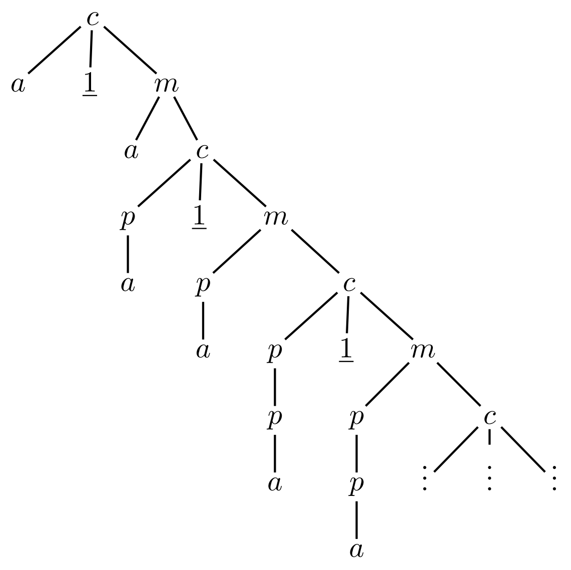


Figure 2.5: The abstract syntax tree $[\![S]\!]$.

$$\begin{aligned} c(a,t,f) &= \text{ if } a = 0 \text{ then } t \text{ else } f \\ m(a,b) &= a \times b \\ p(a) &= a - 1 \\ \underline{1} &= 1 \end{aligned}$$

However, let us consider the higher-order recursion scheme if the function symbols were instead interpreted over the domain of positive integers but with the following function interpretations:

$$\begin{aligned} c(a,t,f) &= \text{ if } a = 0 \text{ then } t \text{ else } f \\ m(a,b) &= a + b \\ p(a) &= a - 1 \\ \underline{1} &= 0 \end{aligned}$$

Our higher-order recursion scheme under this interpretation, from the axiom $S = FAa$ would compute $\sum_{i=0}^{a} i$, rather than $a!$.

This informal example highlights the power and the purpose of higher-order recursion schemes.

Recursive Applicative Program Schemes

Recursion schemes were originally introduced as recursive programs in which the basic functions are unspecified and appear as mere symbols. Then, depending on an interpretation (as indicated above) of the basic function symbols over a specified domain one can retrieve the intended meaning of the program.

Nivat introduced the idea of *recursive applicative program schemes* [Niv72] that correspond to level-1 recursion schemes in our definition. For Nivat, a recursive applicative program scheme is a finite system of equations, each of the form:

$$F_i(x_1, \cdots, x_{n_i}) = p_i$$

where the x_j are ground-level parameters and p_i is some ground-level applicative term over the nonterminals (F_i's), terminals Σ and formal parameters $x_1, \cdots, x_{n_i}$.

However, along with a program scheme (a system of equations), Nivat considered the *pair* of a program scheme *and* an interpretation. Given a program scheme, an interpretation consists of a nonempty set dom_I called the domain of interpretation I and assigns to each $f \in \Sigma$ a function f_I over dom_I of the correct rank. For example, for our example higher-order recursion scheme we gave two such interpretations. Both had as domain of interpretation the set of positive integers, and each specified their own set of functions over the positive integers for the symbols $m, p, c, \underline{1}$.

The pair of a program scheme and an interpretation are then referred to as a "program," and its *value* or semantics can be calculated in a variety of ways. Intuitively, the value corresponds to the function computed by the program. In our example, the first program calculated the factorial of a, and the second calculated the sum of integers from 0 to a.

Thus, schematology, or the study of program schemes, involves the splitting of a program into two parts: it makes the distinction between the *control structure* of the program – embodied by the scheme itself – from other aspects such as the semantic domains involved in the computation. Schematology has several goals in mind. Perhaps the first and foremost is the presentation of a precise model of the notion of a computer program. Based on this one can consider the notions of equivalences between schemes, optimisations [Gue79] of control structures, and of course, abstractions of programs that may serve to help verify correctness properties of the classes of programs represented by the same scheme.

For example, in Nivat's paper on recursive applicative program schemes, Nivat gives us both an operational and fixed-point approach to calculate the function computed by a program (i.e. the pair of a scheme and the interpretation). A notion of equivalence was also defined, saying that two schemes are equivalent if and only if, under all interpretations, they compute the same function.

Subsequently, Courcelle and Nivat [CN78] showed the seminal result that equivalence of schemes can be embodied by the infinite term tree generated in the free interpretation (also known as the Herbrand interpretation), which we have denoted as $\overline{CT^\infty(\Sigma)}$ earlier on this chapter. In particular, if two schemes produce the same infinite term tree in $\overline{CT^\infty(\Sigma)}$ then given any interpretation I, both schemes will compute exactly the same function. This result indicates that we can compute symbolically on trees first and then interpret, instead of having to directly compute in the domain of interpretation. This further highlighted the importance of term tree language theory in the study of programming language semantics.

Remark 2.6.1. Although Nivat [Niv72, CN78] considered only recursive applicative program schemes many other paradigms in "schematology" exist. One useful way to classify them is in terms of schemes that result from (1) imperative-style programs or flowcharts [AN98, Man73] and (2) those that arise from applicative or functional languages [dBS69, Eng74]. Clearly recursive applicative program schemes as defined by Nivat fall into the second category and their derivation trees give rise to a structure representing an "uninterpreted" syntax tree, or expression tree. The other variety tends to produce an execution path. See [GL73] for an interesting discussion on the translatability between various (first-order) schemes.

Remark 2.6.2. An alternative semantics for recursive program schemes was pursued by Courcelle [Cou78a, Cou78b] where Courcelle sought to characterise such schemes by word languages showing their intimate connection with context-free grammars.

Extension to Higher Levels and the Importance of Tree Language Theory

Following results on first-order recursive program schemes, many authors considered the idea of extending the result to higher-order recursive programs.

Some early papers on the topic include [Ind76, Dam77a, Dam77b] and contain various formulations of how one can obtain a semantics for higher-order recursive programs by considering program schemes. One of the main pioneers is Damm, who collects and extends the results of earlier papers and presents a systematic study to the material [Dam82].

Damm introduced level-n λ-schemes, where a level-n λ-scheme is a generalisation of Nivat's recursive program schemes to higher-levels. A level-n λ-scheme as defined by Damm is, intuitively, equivalent to a safe level-n recursion scheme in our definition. The main difference lies in a slightly different notion of types and also allowing for the possibility of λ-abstractions on the right hand side of an equation.

Damm generalised Nivat's and Courcelle's semantic results to higher-levels: showing that if two schemes produce the same infinite term tree then given any interpretation I both schemes compute the same function. In addition to this, Damm's paper offers a number of other contributions that we will explore shortly.

Of particular interest is the study of the term trees and term tree languages associated with schemes, and showing how, by considering these structures associated with a scheme one can compare the "expressive" power of schemes based on their level. Damm claims that the original aim of his investigation was to understand the additional expressive power conferred by higher-order recursion via the lifting of properties of programs to the level of term tree languages – and this was indeed achieved.

We recount some of his main contributions to the field of language theory – note that by language theory we speak of both tree language theory and string language theory. While these results pertain to level-n grammars as defined by Damm, we will shortly see that these results do indeed hold for level-n *safe* grammars as defined here.

Language-Theoretic Results

Damm gives two ways of defining the infinite term tree associated with a higher-order recursion scheme. The first is akin to the one presented here: one associates a schematological higher-order tree-grammar that generates a set consisting of term trees that are finite approximations to the infinite term tree of the recursion scheme; then by taking their join we have the desired term tree. This is the "operational" approach. However, Damm also gives a straightforward "fixed-point" semantics; by simply solving the set of equations defining the higher-order recursion scheme in the continuous algebra $\underline{CT^{\infty}(\Sigma)}$. Both approaches, of course, result in the same term tree.

In the second part of his paper [Dam82], Damm considers higher-order grammars as objects on their own and investigates the term tree languages produced. In particular, he studies the families $n - \mathcal{L}_{OI}$ of term tree languages generated by level-n grammars using the

OI (or, equivalently, unrestricted) mode of derivation as well as the families $n - \mathcal{L}_{IO}$ of term tree languages generated by level-n grammars using the IO mode of derivation. His resulting hierarchies coincide with those introduced earlier by Engelfriet and Schmidt [ES77, ES78] (as generalisations of context-free tree languages), and indeed even earlier proposals by Wand [Wan74] (defining the OI hierarchy in a category-theoretical framework) and by Maibaum (defining the IO hierarchy[6]) [Mai74].

We now present a summary of the key results in [Dam82] concerning the OI- and IO-hierarchies. Please bear in mind that although Damm's definition of a level-n grammar is similar to our own, it is *not* the same. However, we will see that level-n grammars as defined by Damm correspond to level-n *safe* grammars in our definition; therefore the following theorems apply to *safe* level-n grammars as defined earlier. The results presented below concern closure properties, decidability results, and characterisations of the two families.

Theorem 2.6.3. *(Damm 82) $n - \mathcal{L}_{OI}$ and $n - \mathcal{L}_{IO}$ both form infinite hierarchies of term tree languages. Furthermore, for $n - \mathcal{L}_{IO}$, the hierarchy is strict at every level.*

Note that at this point, only strictness of the IO hierarchy of term tree languages could be established. Damm did, however, show that the OI hierarchy was strict in at least the first three levels.

Theorem 2.6.4. *(Damm 82) The emptiness problem for level-n OI tree languages is decidable. The emptiness problem for level-n IO tree languages is decidable.*

Damm also identified some key relationships between term tree language theory and string language theory. For example (we give his contributions to the OI setting):

Theorem 2.6.5. *(Damm 82) The class of level-n OI string languages (where the terminal alphabet consists of only monadic symbols; hence giving rise to only linear term trees or strings) correspond to the path[7] languages of level-n OI tree languages and also the frontier[8] languages of level-$n - 1$ OI tree languages.*

Theorem 2.6.6. *(Damm 82) The class of level-n OI string languages form a substitution closed AFL.*

2.6.2 The Safety Restriction and the Restriction of "Derived Types"

The safety restriction as we have introduced it in Definition 2.4.2 is due to Knapik, Niwiński and Urzyczyn [KNU01, KNU02]. These authors have mentioned that this syntactic constraint on grammars is similar to the restriction of derived types. This is investigated here.

[6]Although it is reported that Maibaum has the terminology of OI- and IO- confused, thus it is believed that the OI hierarchy is the hierarchy *intended*.

[7]Given a set of finite trees S, the path language is defined as

$$path(S) := \{w : w \text{ is branch for some tree } s \in S\}.$$

[8]Given a set of finite trees S, the frontier language is defined as

$$frontier(S) := \{yield(s) : s \in S\}$$

where, for a given finite tree s, $yield(s)$ denotes the concatenation of the symbols on the leaves read from left to right.

In [Dam82], Damm introduces the concept of a *level-n tree grammar G* with terminals Ω, nonterminals X and parameters (variables) Y. Damm denotes the class of level-n tree grammars over Ω by $n - N\lambda(\Omega)$. His definition of a level-n tree grammar G is analogous to our own definition of a level-n grammar and, like our own definition, a term tree language can be associated with it for each mode of derivation. A level-n tree grammar as defined by Damm *by construction* satisfies the so-called restriction of "derived types". For example, with the definitions *we* have introduced we are able to consider both safe and unsafe grammars. However, with Damm's definition, one cannot create a grammar that does not satisfy the "derived types" property – the restriction is built in to the definition.

It is certainly true that a level-n tree grammar as defined by Damm looks and feels very much like a safe level-n grammar defined here. We dedicate this section to proving this formally. We will introduce a new restriction referred to as "safe complete." A grammar that is safe complete is not only safe but there is an additional restriction on the construction of the right hand sides of the production rules. It is straightforward to show that level-n safe complete grammars are equivalent to level-n tree grammars as defined by Damm. Finally, we point out that every safe grammar can be made safe complete.

We concede that devising this new restriction of "safe complete" may appear excessive. However, we hope that upon reading this section the reader will agree that it is a restriction that arises naturally for our purposes and facilitates the proof of the correspondence we wish to show.

Definition of Damm's Level-n Tree Grammars

Definition 2.6.7. Let I be a set of base types. The set $D^*(I)$ of ***derived types over*** I is defined by:

$$\begin{aligned} D^0(I) &:= I \\ D^{n+1}(I) &:= D^n(I)^* \times D^n(I) \\ D^*(I) &:= \bigcup_n D^n(I) \end{aligned}$$

Remark 2.6.8. Note that Damm allows for the possibility of a *set* of ground types, for example such a set might be $\{b, i\}$ where b is the base type representative of Boolean values and i is the base type representative of Integer values.

We, on the other hand, following Knapik, Niwiński and Urzyczyn have limited ourselves to a single base (or ground) type o, noting, as they do, that our results can be generalised to a set of base types.

Example 2.6.9. Some examples of derived types over $I = \{o\}$:

$$o,\ (o, o),\ (\epsilon, (o, o)),\ ((o, o)(ooo, o)(\epsilon, o)(oo, o), (\epsilon, o))$$

where ϵ denotes the empty string.

Definition 2.6.10. The ***level*** of a type $\tau \in D^n(I)$ is its functional depth n.

Remark 2.6.11. Derived types are almost like homogeneous simple types (as we have defined them), but not quite. For example, there is the peculiarity of having a type such as $(\epsilon, (o, o))$. Intuitively, this is a level-2 term that takes *no* level 1 arguments and exactly one level-0 argument. Similarly, $(\epsilon,(\epsilon,(o,o)))$ is a level-3 term that takes *no* level-2 arguments, *no* level-1 arguments and exactly one level-0 argument. The reader may be tempted to think that both are isomorphic to (o,o), but this is not the case. For example, let x be a variable of type $(\epsilon,(o,o))$. Then the following applicative term where x is applied to the empty tuple $(\ )$ $(x(\ ))$ is of type (o,o), where here, the empty tuple is interpreted as an empty list of level-1 arguments.

Definition 2.6.12. Let Δ be a typed alphabet; typed in the sense that each symbol in Δ has a derived type. Similarly, let V be a set of variables; again each variable is of derived type. The set $T_{\Delta,V}$ of ***typed λ-terms over Δ and V*** is the smallest $D^*(I)$-set T with:

1. $\Delta \subseteq T$ and $V \subseteq T$.
2. If $t \in T^{(\alpha,\nu)}$ and $s \in T^{\alpha}$, then $t(s) \in T^{\nu}$.
3. Lastly, if $t \in T^{\nu}$ and $(\alpha,\nu) \in D^*(I)$ then $\lambda y_\alpha.t \in T^{\alpha,\nu}$.

Remark 2.6.13. The above notation deserves some explanation:

1. In the above, if T is a set of symbols, each with derived type, then $T^\tau \subseteq T$ is the set consisting of all symbols in T with type τ.
2. Given a set I, we say that T is an I-set if and only if T is a family of sets indexed by I. In other words, $T = (T_i)_{i\in I}$. Thus, if we say that T is a $D^*(I)$-set, this means that $T = (T^\tau)_{\tau\in D^*(I)}$.
3. Also, given a non-empty word $\alpha = \alpha_1 \cdots \alpha_k \in D^n(I)^*$, so that each $\alpha_i \in D^n(I)$, we identify T^α with the cartesian product $T^{\alpha_1} \times T^{\alpha_2} \times \cdots \times T^{\alpha_k}$. Therefore, the s in (2) of the above definition is, in fact, a vector $(s_1,\cdots,s_k) \in T^\alpha$. By the same reasoning, y_α in (3) is a vector of variables: $(y_1,\cdots,y_k) \in T^\alpha$
4. Note that in the case we have an empty word ϵ, then T^ϵ is identified with the empty set.

Bearing the above remarks in mind, the reader should see that in “essence” the restrictions and safe λ-terms are quite similar. For example, if we restrict ourselves to applicative terms only; then the above rules correspond to applicative terms in the safe λ-calculus without the (*app*+) rule. Regarding abstractions, Damm’s definition does not require that lower level variables occurring in the body of the of the abstraction be bound. However, these discrepancies need not worry us as we need only to consider them in the context of tree grammars and not as standalone terms.

Definition 2.6.14. (Damm) A level-n tree grammar G consists of a $D^*(I)$-set of nonterminals X, a $D^*(I)$-set of parameters Y and finally a set of terminals Ω of type $D^1(I)$, and a $D^*(I)$ map $S : X \longrightarrow \mathcal{P}(T_{\Omega,X,Y})$ such that:

1. $X = \{x_0,\cdots,x_N\}$ is a finite $D^*(I)$ set.
2. The type of x_0 is $(\epsilon,\cdots,(\epsilon,o)\cdots)$.
3. For all x, $S(x)$ is finite and for each element in $S(x)$ is a closed λ-term[9].

[9]Closed in the usual sense of the word, where $X \cup \Omega$ are treated as constants, and only members of Y are understood as variables.

Remark 2.6.15. Again, we require some clarifying remarks.

1. The fact that $S : X \longrightarrow \mathcal{P}(T_{\Omega,X,Y})$ is a $D^*(I)$ map means that if x_i is of type τ then so is its image.

2. x_0 can be thought of as the start non-terminal.

3. To give the reader an intuitive picture we include an example of a level-n tree grammar as defined by Damm:

$$\begin{aligned} x_0 &= \{\lambda.\lambda.x_2(a)(e(\,))\} \\ x_1 &= \{\lambda y_1.\lambda y_0.y_1(y_1(y_0)), \lambda y_1.\lambda y_0.y_0\} \\ x_2 &= \{\lambda y_1.\lambda y_0.f(x_2(x_1(y_1))y_0)(y_0), \lambda y_1.\lambda y_0.y_1(y_0))\} \end{aligned}$$

   where $X = \{x_0 : (\epsilon, (\epsilon, o)), x_1 : ((o, o), (o, o)), x_2 : ((o, o), (o, o))\}$ are the non-terminals, $Y = \{y_0 : o, y_1 : (o, o)\}$ are the parameters or variables, and the terminals, Ω are given by $\{f : (oo, o), a : (o, o), e : (\epsilon, o)\}$.

   Note that this may be rewritten as the following system of equations (that look somewhat more familiar) by omitting all the outermost abstractions:

$$\begin{aligned} x_0 &= \{x_2(a)(e(\,))\} \\ x_1 p_1 p_0 &= \{p_1(p_1(p_0)), p_0\} \\ x_2 p_1 p_0 &= \{f(x_2(x_1(p_1))p_0)(p_0), p_1(p_0))\} \end{aligned}$$

   Damm also notes ([Dam82], Corollary 4.12) that it is sufficient to consider only applicative terms as right hand sides (therefore, not only omitting the outermost abstractions as we have done, but also any that may still be present after doing so).

4. The notions of derivations and associated term tree languages are analogous to our own, and we will use them without formal introduction.

Comparing Safety and the Restriction of "Derived Types"

Definition 2.6.16. Let A be a simple homogeneous type. A is said to be ***complete*** if and only if A is of ground type or; $A = A_1 \to A_2 \to \cdots \to A_n \to o$ (where $n \geq 1$) is such that:

1. For each $i = 1, \cdots, n$ we have that either $\mathsf{level}(A_i) = \mathsf{level}(A_{i+1}) + 1$ or $\mathsf{level}(A_i) = \mathsf{level}(A_{i+1})$

2. $A_n = o$

3. Each A_i is homogeneous and complete.

Example 2.6.17. Intuitively, a homogeneous type is complete if and only if it is of ground type, or in the case it is of level n where $n > 0$, then it expects at least one argument of level k for each $k \in \{0, 1, \cdots, n-1\}$. Consider the following type, $A = ((o, o), o)$. This is homogeneous, however it is not complete. On the other hand $((o, o), (o, o), o, o)$ is homogenous and complete.

Definition 2.6.18. Let $G = \langle N, \Sigma, V, S, \mathcal{R} \rangle$ be a level-n grammar as given in Definition 2.2.1. G is said to be ***safe complete*** if and only if all symbols in N and V have homogeneous and complete types. Furthermore, the righthand sides of the rules in $\mathcal{R}$ are safe λ-terms constructed only using rules (wk), (var), $(const)$, and (app).

Claim 2.6.19. *Every safe complete level-n grammar can be converted into a level-n tree grammar (as defined by Damm) such that it produces the same term tree language.*

Proof. We define a map from homogeneous and complete types to derived types as follows.

$$\begin{array}{rcl} M(o) & = & o \\ M(A_1 \to A_2 \to \cdots \to A_n \to o) & = & (M(A_1) \cdots M(A_k), M(A_{k+1} \to \cdots \to A_n \to o)) \end{array}$$

In the 2nd equation, k is the greatest integer (from 1 to n) such that $\mathsf{level}(A_1) = \mathsf{level}(A_2) = \cdots = \mathsf{level}(A_k)$ and juxtaposition expresses concatenation. Then for any symbol of type A we simply assign it a new type $M(A)$. It is routine to check that a well-formed term of type A using only the rules (wk), (var), $(const)$, and (app) of the safe λ-calculus is also a well-formed term λ-term of type $M(A)$ according to Damm's definition using the new types. □

Claim 2.6.20. *Every level-n tree grammar (as defined by Damm) can be converted into a safe complete level-n grammar that produces the same term tree language.*

Remark 2.6.21. Before giving the proof let us comment that we will use a similar mapping on types to the one above (from derived types to homogeneous and complete types) and then extend it to terms and finally grammars.

This construction requires a bit more care. For example, consider the following type $(\epsilon, (\epsilon, o))$. As intimated earlier, although it is tempting to merely delete empty types we cannot get away with this. Consider the less trivial derived type $((e, o)(o, o), (oo, o))$ – deleting empty tuples would result in the type $o \to (o, o) \to o \to o \to o$, which is no longer homogeneous. Thus the empty types occurring at functional level n will be converted into a suitable type of level n.

Definition 2.6.22. We define the ***standard simple types*** to be the following:

$$\begin{array}{rcl} \underline{0} & := & o \\ \underline{n+1} & := & \underline{n} \to \underline{n} \end{array}$$

Proof. Let M' be a map from the derived types $D^*(I)$ to the simple homogeneous and complete types where:

$$\begin{array}{rcl} M'(o) & := & o \\ M'(\alpha_1 \cdots \alpha_n, \nu) & := & M'(\alpha_1) \to \cdots \to M'(\alpha_n) \to M'(\nu) \text{ where } n > 0 \\ M'(\epsilon, \nu) & := & \underline{n} \to M'(\nu) \text{ where } \mathsf{level}(\nu) = n \end{array}$$

We extend the map M' to terms, where $M' : T_{\Omega,X,Y} \longrightarrow S$. Here, S consists of those terms built from the constants

$$\{N : M'(\tau) \mid N \in X^\tau\} \cup \{f : M'(\tau) \mid f \in \Omega^\tau\} \cup \{\bot_n : \underline{n} \mid n \geq 0\}$$

and variables

$$\{y : M'(\tau) \mid y \in Y^\tau\}$$

using only rules (*wk*), (*var*), (*const*), and (*app*) of the safe λ-calculus.

The map $M' : T_{\Omega,X,Y} \longrightarrow S$ is defined inductively below:

1. If $x \in X \cup Y \cup V$ and has type τ then $M'(x)$ is the symbol x with type $M'(\tau)$.
2. If $s(\ )$ is an applicative term where $s : (\epsilon, \nu)$, then $M'(s(\ )) = M'(s)\bot_n$ where $\mathsf{level}(\nu) = n$.
3. If $s(t_1 \cdots t_n)$ is an applicative term where $s : (\alpha_1 \cdots \alpha_n, \nu)$ where $n \geq 1$ and each $t_i \in T^{\alpha_i}$ then $M'(s(t_1 \cdots t_n)) = M'(s)M'(t_1) \cdots M'(t_n)$.

It is, again, routine to check that a well-formed term $t \in T^\tau_{\Omega,X,Y}$ implies that $M'(t)$ a well-formed term of S with type $M'(\tau)$. (In effect, we have merely replaced empty tuples with a single argument whose type reflects the functional level of the empty tuple.)

□

Remark 2.6.23. From the above two claims, we have shown that level-n tree grammars as defined by Damm correspond exactly to a subset of safe level-n grammars; namely the safe complete ones. However, we have not yet explained the relationship between safe and safe complete grammars. Let us illustrate the difficulty in establishing this relationship with the following safe (but not complete) grammar where $N = \{S : o, F : ((o,o), o, o, o)\}$, $\Sigma = \{f : (o,o,o), g : (o,o), a : o, b : o\}$ and $V = \{\varphi : (o,o), x : o, y : o\}$:

$$\begin{aligned} S &= Fgab \\ F\varphi xy &= F(fa)(\varphi x)y \end{aligned}$$

Note that $f : (o,o,o)$ and $a : o$. Therefore the term $(fa) : (o,o)$ cannot be constructed without the use of the (*app*+) rule of the safe λ-calculus. There does not seem to be an obvious way of converting such a grammar into a level-n tree grammar as defined by Damm.

However, fortunately, a proof that every safe grammar corresponds to a safe complete one has already been shown (due, inadvertently, to Knapik, Niwiński and Urzyczyn [KNU02]). As stated earlier (Section 2.5.1), Knapik et al. have shown the correspondence between level-n safe OI grammars and level-n pushdown automata. Thus, given a safe level-n grammar we apply their conversion from a level-n safe grammar to a level-n pushdown automaton. We then apply their conversion from the resulting level-n pushdown automaton back to a level-n grammar. However, closer inspection of the last application reveals that the resulting grammar is not only safe, but it is also safe complete. See Section 5.1 of [KNU02] for complete details. This completes the proof.

Corollary 2.6.24. *All the properties pertaining to level-n tree grammars hold for level-n safe grammars.*

2.6.3 Higher-Order Pushdown Automata

In comparison with the preceding subsection on higher-order grammars and higher-order recursion schemes, this section on higher-order pushdown automata is surprisingly brief. This is not to suggest that contributions in this area are sparse, but merely that it is less of a focus for this thesis, and the main result we need has already been stated, namely the equivalence between level-n pushdown automata and level-n safe grammars.

Nevertheless, we take a moment to consider some the advances in this area. The results that following pertain almost exclusively to formal language theory in the conventional sense: string languages.

To our knowledge, the concept of the iterated or higher-level pushdown automaton was first mentioned in Greibach [Gre69] and first defined by Maslov [Mas74]. Maslov introduced "iterated stack automata" as an automata-theoretical characterisation of generalised indexed languages [Mas74, Mas75].

Indexed Languages were first introduced by Aho [Aho68] as an example of a class of languages strictly between the context-free and context-sensitive languages. The canonical example of a non-context-free language is $\{a^n b^n c^n : n \geq 0\}$ and this was shown to be an indexed language. Maslov generalised this idea to indexed languages of level-n for all n, generating an infinite hierarchy of languages, the nth level of which is the class of languages defined by level-n indexed grammars. Note that Aho's "indexed languages" correspond to level-2 indexed languages in Maslov's generalised setting.

In 1982, Damm and Goerdt [DG86] showed that for level-n grammars over a monadic alphabet, the OI language corresponds to the language of a level-n pushdown automata and vice versa. Recall that a level-n grammar in Damm's definition corresponds to a safe level-n grammar in our definition. Note that we now have a three-way correspondence: level-n safe grammars are equivalent to level-n pushdown automata and these are in turn equivalent to level-n indexed grammars.

As we may recall from the previous subsection, Damm showed the strictness of the IO hierarchy but not that of the OI hierarchy. However, in 1983, Engelfriet [Eng83] (a full version of this paper may be found in [Eng91]) considered the characterisation of complexity classes by higher-order pushdown automata. In particular, he showed that non-deterministic 2-way and multi-head pushdown automata characterise deterministic iterated exponential time complexity classes. Furthermore, an application of this result showed that 1-way iterated pushdown automata (or simply higher-order pushdown automata as we have defined them) form a proper hierarchy with respect to the number of iterations (level). This, combined with the earlier result of Damm and Goerdt shows that the OI string language hierarchy is indeed strict.

Furthermore, recall that level-n OI tree languages have the property that their path languages correspond to level-n OI string languages. It therefore follows that:

Theorem 2.6.25. *The OI hierarchy of tree languages is strict.*

Another result shown by Engelfriet in the same paper is that the emptiness problem is complete in deterministic iterated exponential time.

A final result we shall mention is due to Engelfriet and Vogler [EV87]: using the concept of lookahead on pushdown automata, in 1987, they proved that:

Theorem 2.6.26. *(Engelfriet and Vogler, 1987) The class of languages generated by level-n* deterministic *pushdown automata is strictly contained in the class of level-n non-deterministic pushdown automata, for all levels $n \geq 1$.*

The generalisation of higher-order pushdown automata to acceptors of term trees was first given by Knapik et al. [KNU02] and it is what our formulation is based on (Definition

2.5.1). Strictly speaking, Knapik et al. defined a higher-order pushdown automaton as an acceptor of a *single* (potentially infinite) term tree. We have modified this definition to accept a set of finite term trees. However, under suitable conditions (such as determinacy and an interpretation in the continuous algebra $\underline{CT^{\infty}(\Sigma)}$) we can indeed define a single (potentially infinite) term tree, much in the same way the term tree language of the schematological higher-order grammar of a higher-order recursion schemes forms a directed set whose join is the desired infinite term tree.

2.6.4 Verification

We now consider the issue of which logics are decidable over structures generated by higher-order recursion schemes. To the author's knowledge the very first such higher-order result is due to Hungar [Hun99] who shows that the monadic second-order theory restricted to *paths* of the term trees generated by level-2 safe recursion schemes is decidable. However, the key result from which this thesis can be said to take its cue is the following.

In [KNU01], Knapik et al. defined a higher-order grammar as a definitional device for a single (potentially infinite) term tree. Their definition of a higher-order grammar corresponds exactly to our definition of a higher-order recursion scheme[10]. They then showed the remarkable result that for grammars of level-2, provided the grammar was safe, the resulting infinite term tree possessed a decidable MSO theory.

We summarise the intuition behind their proof in three steps. We focus on the intuition only for two reasons: (1) the paper [KNU01] is very readable and thus we do not attempt to reproduce details here; (2) neither the details of the proof nor the ideas from it (apart from one) will be reused in this thesis.

In the first stage, given a level-2 grammar, we associate with this grammar an infinite λ-term (or rather, a λ-tree), that effectively β-reduces to the original term.

The second phase involves showing that the the MSO theory of the term tree defined by the level-2 grammar can be expressed in the MSO theory of the associated λ-tree. The proof requires a detailed investigation of how paths in the tree are affected by β-reductions. It is then shown that these "deformations" that result in the path after a β-reduction can be captured by finite automaton (and thus can be described by monadic second-order logic). Knapik et al. base this construction on techniques used by Caucal [Cau96].

Finally, in the third phase, it is shown that these associated λ-trees can be generated by level-1 grammars *provided* the original level-2 grammar is safe. We give an example below.

Example 2.6.27. Consider the following level-2 unsafe grammar.

$$\begin{array}{rcl} S & = & Fgab \\ F\varphi xy & = & F(fx)y(\varphi x) \end{array}$$

where non-terminals $S : o, F : ((o,o),o,o,o)$, terminals $f : (o,o,o), g : (o,o), a : o, b : o$ and variables $\varphi : (o,o), x : o, y : o$.

Knapik et al. propose a reduction of this level-2 grammar into a level-1 grammar. This is done in such a way that the level-1 grammar produces a *representation* of an (infinite) λ-term that ultimately reduces to the tree of the original level-2 grammar. The corresponding level-1 grammar using Knapik et al.'s construction is the following:

[10]This interchange of terminology is slightly confusing and we hope the reader can accept higher-order grammar to mean higher-order recursion scheme for the next few paragraphs.

$$\begin{array}{rcl} S & = & @(@(Fg)a)b \\ F\varphi & = & \boldsymbol{\lambda}\mathbf{x}(\boldsymbol{\lambda}\mathbf{y}(@(@(F(@f\mathbf{x}))\mathbf{y})(@\varphi\mathbf{x}))) \end{array} \tag{2.2}$$

where

$$\{\boldsymbol{\lambda}\mathbf{x} : (o,o), \boldsymbol{\lambda}\mathbf{y} : (o,o), @ : (o,o,o), f : o, g : o, a : o, b : o, \mathbf{x} : o, \mathbf{y} : o\}$$

are the terminals and the non-terminals are now $S : o, F : (o,o)$. Note that $\boldsymbol{\lambda}\mathbf{x}$ is a *single* symbol and not two symbols. Finally, note that there is only one variable, $\varphi : o$.

Under the understanding that $((@a)b)$ means "a is applied to b" and that $(\boldsymbol{\lambda}\mathbf{z}t)$ is to be interpreted as $\lambda z.t$, then the above is a *representation* of the following recursive equations involving λ-terms:

$$\begin{array}{rcl} S & = & Fgab \\ F\varphi & = & \lambda x.\lambda y.F(fx)y(\varphi x) \end{array} \tag{2.3}$$

This example should hopefully illustrate why we might expect the term-tree of the level-1 grammar in Equation 2.2 to be a λ-tree whose β-normal form is the term tree of the original grammar. However, it cannot be emphasised enough that this is a *representation* of a λ-tree. Hence, when rewrite rules are applied they are performed in a context-free manner[11]. For example, let us consider an expression:

$$\cdots @(F(@f\underline{\mathbf{x}}))\cdots$$

applying the 2nd rewrite rule would result in the following:

$$\cdots @(\boldsymbol{\lambda}\mathbf{x}(\boldsymbol{\lambda}\mathbf{y}(@(@(F(@f\mathbf{x}))\mathbf{y})(@(f\underline{\mathbf{x}})\mathbf{x}))))\cdots$$

Note, in particular, the underlined x has been captured! Clearly this changes the semantics of the λ-tree we are trying to represent.

Thus, the term tree that results is only semantically correct provided that one can ensure that one is always able to perform substitutions *without* α-renaming. And safety was the condition that Knapik et al. developed and showed to be sufficient. In light of our remarks on the safe λ-calculus, the reader should be able to readily agree with this.

Thus, given a safe level-2 grammar, Knapik et al. can construct a level-1 grammar that generates an infinite λ-tree that β-reduces to the infinite term tree of the original level-2 grammar. They then rely on the proof by Courcelle to show that level-1 grammars always produce term trees with decidable MSO theories [Cou95]. This renders the result that the λ-tree has a decidable MSO theory and by the earlier result of step (2), so does the term tree of the original higher-order grammar. This completes the proof.

In [KNU02] Knapik et al. extended their proof to all safe level-n grammars, by a simple reduction of a level-$n+1$ grammar to a level-n grammar that produced the associated λ-tree in the above sense. The proof then followed by induction.

[11]In the sense that no α-conversion is performed and the substitution is blind.

Note that the concept of safety as presented by Knapik et al., although similar to that of derived types, introduced the possibility of “unsafe grammars” – something that was never considered before. All results pertaining to Damm’s OI and IO grammar concern safe grammars, and indeed with Damm’s presentation there was no scope to consider unsafe grammars (or rather, grammars not of derived type).

Although the concept of safety can thus be said to date back to the the 1980s the concept of “unsafety” can be entirely attributed to Knapik et al.

Having now read the results and characterisations of safe higher-order grammars, we hope that the reader will also feel a need to understand the relationship between safe grammars and unsafe grammars and investigate whether the good properties of safe grammar hold for unsafe grammars.

Chapter 3

Expressibility

In this chapter we aim to address the question of whether or not safety restricts the classes of structures that can be generated by higher-order grammars. More specifically, we fix a derivation mode m, where m is either OI or IO. Let L be the term-tree language generated by a level-n unsafe grammar using derivation mode m; does there exist a level-n *safe* grammar capable of generating the same term tree language using derivation mode m? If not, does there exist a *safe* grammar of a higher level that generates the same term tree language?

Thus, this chapter aims to provide a first step towards understanding whether safety is a restriction in terms of *expressibility.* As mentioned in the introduction, in the next chapter we will examine the issue of *decidability* with respect to monadic second-order logic. Whilst we present a collection of results here, our key results in both this chapter and the next focus on level-2 grammars (and recursion schemes). By definition, all level-0 and level-1 grammars are safe, and thus level 2 is the first level where unsafe grammars can occur. Our focus on level-2 is, of course, a limitation and means we are unable to speak of the hierarchy as a whole; however, we believe that our level-2 results are an important first step and should pave the way to generalising the results for the whole hierarchy.

Another warning should be issued to the reader before progressing any further. The central result of this chapter as well as that of the next are not only both about level-2, but both concern grammars (respectively recursion schemes) with *unrestricted* derivations, or, equivalently OI derivations. Our focus on the OI hierarchy is a natural progression from the current state of affairs. For this chapter, our main result depends very much on the "OI-ness" of OI grammars by way of their implementation through pushdown automata. Similarly, in the next chapter we sought to build on top of the framework suggested by Knapik et al. which is implicitly OI. Thus, although it would be desirable to include results on the IO hierarchy this is left as further work.

The key result of this chapter is the following. We will show that for OI grammars restricted to a *monadic alphabet*, every level-2 unsafe grammar can be converted into a level-2 safe one that generates the same language. The restriction to monadic alphabets implies that these are, of course, word languages. So this can be thought of as a result for formal languages in the classical sense. This result was first published in FoSSaCS 2005 [AdMOb]; published jointly with Luke Ong and Klaus Aehlig.

This chapter is divided into six sections. As the key result of this chapter focuses on OI word languages, we dedicate the first section of this chapter to studying OI higher-order grammars in a purely word-language setting. We first change the notation slightly, so that rather than a production rule of the form $Fx_1 \cdots x_n = r$, we change this to $Fx_1 \cdots x_n \xrightarrow{a} r$, where a is a symbol in $\Sigma \cup \{\epsilon\}$. Although this is a superficial adjustment it will prove convenient for our purposes. We will refer to such a grammar as a *word-generating* grammar (or simply word grammar) – and all word grammars will be, by construction, OI.

In the second section, to encourage a better understanding for the power of OI word-grammars we consider a case study: Urzyczyn's [Urz03] language. This is a language that is straightforward to define via an unsafe level-2 word grammar; but which has attracted some attention as a language that was conjectured not to be accepted by any level-n pushdown automaton. Our level-2 language result refutes this, but we also give a bespoke proof of this, by giving an alternative characterisation of the language which can very easily be seen to be accepted by a level-2 non-deterministic pushdown automaton. We feel this example provides a good insight into how both grammars and pushdown automaton can be used to define languages. Furthermore we consider our alternative characterisation of Urzyczyn's language to be of independent interest given the status of the language.

In the third section, we introduce pointer machines as a machine characterisation of word-generating grammars (whether safe or unsafe). Pointer machines are due to Colin Stirling [Sti02] and little literature exists at the time of writing. We hope to rectify this in this thesis. It should be noted that pointer machines can be defined more generally as machine characterisations of OI higher-order grammars (without the restriction to monadic alphabets), but we will not need this generalisation here and merely point out that it is a straightforward extension.

Finally, we are in a position to give our main result showing that level-2 unsafe word grammars are no more expressive than level-2 safe word grammars; and this is done in the fourth section. This is achieved by a 2-step transformation. We show that the pointer machine associated with any level-2 word grammar can be simulated by a level-2 pushdown automaton with links; a machine yet to be defined. We complete the proof by showing that the derived level-2 pushdown automaton with links can then be simulated by a non-deterministic pushdown automaton. This completes the proof.

In the fifth section we consider some of the ramifications of the above result. We will see that our construction of the 2PDA introduces a "layer" of non-determinism in an essential way. In fact, this "layer" can only be safely removed in the case where the word grammar is safe. To formalise this and other notions, we will introduce the concept of deterministic word grammars (versus non-deterministic) and prove the simple, but desirable result that safe deterministic word grammars are equivalent to deterministic pushdown automata over words. However, this leads to some interesting questions; what is the relationship between languages generated by safe deterministic, safe non-deterministic, unsafe deterministic and unsafe non-deterministic grammars? There is one key relationship that we wish to understand: the relationship between safe deterministic and unsafe deterministic grammars. Whilst we have been unable to establish what this relationship is, we do prove the following. We show that if we can prove the existence of a language L that is generated by a level-2 unsafe deterministic grammar and not by any level-2 safe deterministic grammar then this is equivalent to a proof of the existence of a term tree that can be accepted by an unsafe level-2 recursion scheme but not by any safe one. We believe that Urzyczyn's language is such a language; but this remains a conjecture.

We conclude with a summary of related work and future directions.

3.1 Word-generating Grammars

As detailed in the introduction, this chapter is devoted almost exclusively to higher-order grammars with the restriction that the terminal alphabet, Σ, is monadic. This was introduced in Section 2.2.2 as a special case of higher-order grammars $\langle N, \Sigma, V, S, \mathcal{R} \rangle$ satisfying the following:

- Σ contains a distinguished symbol $e : o$, to be understood as marking the end of a word.
- For all other $f \in \Sigma$, $f : (o, o)$, i.e. every symbol other than e is monadic in that it expects only one argument.

Any higher-order grammar G satisfying the above constraints will only produce monadic – or linear – trees, each of the from we where $w \in (\Sigma - \{e\})^*$.

In particular, $L_m(G)$ is isomorphic to the string language:

$$\{w \in (\Sigma - \{e\})^* \mid we \in L_m(G)\}$$

where m is the mode of derivation (unr, OI, or IO).

In this chapter we will restrict our attention to the unrestricted (or equivalently, OI) mode of derivation. To aid and emphasise the focus on languages, we introduce the following:

Definition 3.1.1. A ***word-generating grammar*** (or simply word grammar) of level-n is a six-tuple $W = \langle N, \Sigma, V, S, \mathcal{R}, e \rangle$, such that the following hold:

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

(ii) V is a finite set of typed variables.

(iii) Σ is a finite alphabet, containing the unique end-of-word character e of type o and all other symbols in Σ are of type (o, o).

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

$$Fx_1 \cdots x_n \xrightarrow{a} E \tag{3.1}$$

where $F : (A_1, \cdots, A_n, o) \in N$, each $x_i : A_i \in V$, and E is either a term in $\mathcal{T}^o(N \cup \{x_1, \cdots, x_n\})$ or is equal to e. We say that F has ***formal parameters*** $x_1, \cdots, x_n$. Furthermore, $a \in \Sigma$ or $a = \epsilon$, where ϵ is the empty string.

We make the same assumptions as in Definition 2.2.1. We say that W is of level n if and only if n is the level of the rewrite rule that has the highest-level.

3.1.1 The Language of a Word Grammar

We extend $\mathcal{R}$ to a family of binary relations $\xrightarrow{\alpha}$ over $\mathcal{T}^o(N) \cup \{e\}$, where α ranges over $\Sigma \cup \{\epsilon\}$, by the rule: if $Fx_1 \cdots x_n \xrightarrow{\alpha} E$ is a rule in $\mathcal{R}$ where $x_i : A_i$ then for each $M_i \in \mathcal{T}^{A_i}(N)$ we have

$$FM_1 \cdots M_n \xrightarrow{\alpha} E\overline{[M_i/x_i]}.$$

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

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

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

Remark 3.1.2. Note that encapsulated in the definition of the rewrite relation above is an OI derivation. This is enforced by insisting that we always reduce the expression headed by the outermost non-terminal.

3.1.2 Safe and Unsafe Word Grammars

The definition of a safe word grammar and an unsafe grammar can be given as follows.

A grammar is ***safe*** if and only if for each rewrite rule $Fx_1 \cdots x_n \xrightarrow{\alpha} E$ we have that

$$\{x_1 : A_1, \cdots, x_n : A_n\} : 0 \vdash E : o$$

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

Equivalently, using Knapik et al.'s formulation: a grammar is ***safe*** if for each rewrite rule $Fx_1 \cdots x_n \xrightarrow{\alpha} E$ we have that E is safe as defined in Section 2.4.2.

Remark 3.1.3. It should be clear that the definition of a word-grammar of level-n results from merely a cosmetic change to the definition of a higher-order grammar when restricted to a monadic alphabet and an OI mode of derivation. The proof is trivial.

3.2 Urzyczyn's Language

We consider an example of a word grammar that sets the pace for the remainder of this chapter. We introduce an unsafe level-2 word grammar that generates a language that we call ***Urzyczyn's language***, or simply U. Not only is this a non-trivial example of a level-2 unsafe word grammar but it was also originally conjectured to be an example of a language that is inherently unsafe, i.e. that it cannot be accepted by level-n pushdown automaton [Urz03] for any n. Should this conjecture – Urzyczyn's conjecture – have been proven correct this would indeed show that safety is a restriction in terms of expressive power for grammars (and not just word grammars).

This section refutes this. After introducing U, we show that it *can* be accepted by a level-2 *non-deterministic* pushdown automaton. The emphasis on non-determinism is important; indeed we believe that Urzyczyn's conjecture is correct if one restricts attention to *deterministic* pushdown automata. We return to this later.

The proof was first published in the technical report [AdMO04] and it proceeds by giving an alternative characterisation of the language U from which it is straightforward to show that this language can be accepted by a level-2 non-deterministic pushdown automaton. Of course, this "bespoke" proof showing that U can be accepted by a 2PDA may seem redundant given that U is exactly the type of language amenable to our key result of this chapter. However we believe there is much to be gained from our case study:

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

3.2.1 Definition of U

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

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

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

Example 3.2.1. For example, the following are words in U (together with their respective sequences of labels):

(i)

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

(ii)

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

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

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

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

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

By mimicking these transitions, we can define an unsafe level-2 word grammar that generates U. We set

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

with production rules as follows:

$$\left\{ \begin{array}{rcl} S & \xrightarrow{(} & D\,G\,E\,E\,E \\ D\,\phi\,x\,y\,z & \xrightarrow{(} & D\,(D\,\phi\,x)\,z\,(F\,y)\,(F\,y) \\ D\,\phi\,x\,y\,z & \xrightarrow{)} & \phi\,y\,x \\ D\,\phi\,x\,y\,z & \xrightarrow{*} & z \\ F\,x & \xrightarrow{*} & x \\ E & \xrightarrow{\epsilon} & e \end{array} \right.$$

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

3.2.2 Unique Decomposition of U-words

We now turn to our proof that U can be accepted by a level-2 non-deterministic pushdown automaton (and hence can be generated by a safe level-2 OI grammar by Damm and Goerdt [DG86]). The crux of the proof lies in the following decomposition.

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

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

(1) is a prefix of a well-bracketed word ending with final symbol (, such that no prefix of it (including itself) is a well-bracketed word,

(2) is a well-bracketed word,

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

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

$((((\,))\,($ and $(\,)\,((\,))$.

Remark 3.2.2. One simple way to decompose a U-word into its V components is turn the expression upside down. Using Example 3.2.1.(ii) as a concrete example, this renders an expression of the following form $*****(\,(\,)\,)\,(\,)\,)\,(\,(\,)\,)\,)\,)$. With this new expression one finds the longest well-bracketed expression from left to right; this constitutes part (2) and what remains to the right of this is part (1). Of course, one needs to turn them back the right way up!

Proposition 3.2.3. *Let $y \in \{(,),*\}^*$. Then $y \in U$ if an only if it has a unique decomposition into $wx*^n$ where w, x, n satisfy conditions (1),(2),(3) above respectively.*

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

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

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

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

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

To show *uniqueness*, suppose that $y \in U$ has two decompositions: $wx*^k$ and $w'x'*^k$. It suffices to observe that these decompositions are distinct if and only if, w.l.o.g., w is a strict prefix of w'. However, by assumption both w and w' end in (and it must follow that $|w|_(< |w|_($, this is untenable as both must be equal to k.

$\Leftarrow$: Again, we perform a case analysis. Let $y = wx*^n$ be a word in V. By definition x is a well-bracketed expression and therefore we must have that $x = x_1 \cdots x_m$ for some $m \geq 0$ where each x_i is a well-bracketed expression. We perform a case analysis on m:

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

(ii). Suppose that $m > 0$. Clearly wx is the proper prefix of a well-bracketed expression. Furthermore each x_i in the expression $x = x_1 \cdots x_m$ (where $m > 0$ in this instance) must be of the form $x_i = (z_i)$ and each z_i is well-bracketed. Thus we have $y = w(_1 z_1)_1 \cdots (_m z_m)_m *^n$. Parentheses have been labelled for demonstrative purposes. For y to be in U the number of stars, n, should be the label of the parenthesis $)_m$. We show this is indeed the case. The label of $)_m$ should be equal to the label of the parenthesis preceding $(_m$, which in this case would be $)_{m-1}$. Continuing in this way, we see that the label of $)_m$ is the label of the last parenthesis of w, which is, by definition a (and this has label $|w|_($.

□

3.2.3 Constructing a 2PDA that Accepts U

How do we construct a 2PDA that accepts U? Let y be the input. Let us reuse the notation from Proposition 3.2.3, so each word in U has a unique decomposition into the form $wx*^n$. The 2PDA guesses what prefix of y constitutes w – say it is the first k characters. As it reads the first k symbols it acts like a 1PDA checking that w is indeed a prefix of a well-bracketed word such that no prefix of it is well-bracketed. However, at the same time we need it to keep track of the number of (’s read. In order to perform this dual job we need the power of the 2PDA. Essentially, the topmost 1-store behaves like the stack of a normal PDA which checks for a (proper prefix of a) well-bracketed word. However, each time we find a (we execute both a push_1 and a push_2. Each time we find a), we execute only a pop_1. Thus, the number of 1-stores is the number of (’s read in w plus one[1]. If it turns out that the first k characters of the input y do not satisfy the criteria for what w should be, we immediately reject. Otherwise, we enter phase 2 of the operation. In phase 2 we check that the next portion of the string, x, is well-bracketed (for this we only need the power of a 1PDA), until we come across the first $*$. If x was indeed well-bracketed we proceed to phase 3, otherwise we abort. In phase 3, we perform a pop_2 for each $*$ we meet. If we end up with an empty 2-store after reading the all the $*$’s, we accept. Otherwise, reject. It should be straightforward to see that due to the fact each U word has exactly 1 V-decomposition, if we guess w incorrectly, this will always lead to an abortive computation.

We now move on to the subject of generalising the above statement (namely that U can be accepted by a level-2 non-deterministic PDA) to all level-2 unsafe grammars.

3.3 Machine Characterisations of Word Grammars

In this section we introduce the pointer machine as a machine characterisation of word grammars. They are a concept due to Colin Stirling [Sti02] and we include them here for two reasons; firstly we use them as part of our proof for the key result of this chapter; second, very little literature exists on them at the time of writing and it is felt that they are an important contribution to this field.

Pointer machines as introduced here can be better thought of as an *implementation* of a word grammar; their definition is tied to the definition of the word grammars they implement.

We introduce them in this language-theoretic setting, i.e. for our word grammars. However, the definition can easily be adapted to OI higher-order grammars in general. For our needs the word-grammar setting is sufficient and illustrates all the key concepts.

3.3.1 Definition of Pointer Machine

Pointed Alphabet

Let B be a set. Given an alphabet Γ, we define the ***pointed alphabet*** Γ^B as the alphabet : $\Gamma \times B$. To simplify notation, we will write (a, β) as a^β. We refer to the second component (or equivalently, the superscript) as the ***pointer*** of a.

In the case where Γ is typed, then Γ^B is also typed. In particular, a^β has type A if and only if $a : A$. Note that the superscript (pointer), β, makes no contribution to the type. As

[1] Alternatively, we can adjust the transition function slightly to make it the number of (’s in w exactly

Γ^β is a typed alphabet, we can define the set of applicative terms over this alphabet in the usual way. Namely, if $u^\beta : A_1 \to A_2$ and $v^\gamma : A_1$, then $(u^\beta v^\gamma) : A_2$.

Unless otherwise specified, we will assume that Γ is typed. It will be convenient for us to use the following notational convention: instead of associating an element of β with *individual* elements in Γ, we allow ourselves the ability to associate it with *applicative terms.* This is entirely to avoid the proliferation of superscripts. We define:

$$((st), \beta) = ((s, \beta)(t, \beta))$$

where $st \in \mathcal{T}(\Gamma)$. For example, instead of writing:

$$F^\beta \varphi^\beta x^\gamma (\varphi^\gamma (\varphi^\gamma x^\gamma))$$

we can write this as:

$$(F\varphi)^\beta x^\gamma (\varphi(\varphi x))^\gamma.$$

Therefore for any applicative term $s \in \mathcal{T}(\Gamma)$, the term $s^\beta \in \mathcal{T}(\Gamma^B)$ denotes that each symbol occurring in the term has pointer β. We will make use of this notation extensively in this chapter – we hope the reader will find that it is actually very natural for our applications.

For the remainder of this section, we will use the following conventions, given a typed alphabet Γ and a set B:

1. $\sigma, \tau, \cdots$ are elements of $\mathcal{T}(\Gamma^B)$;
2. $s, t, \cdots$ are elements of $\mathcal{T}(\Gamma)$
3. $\alpha, \beta, \gamma, \cdots$ are elements of B.
4. Finally, every applicative σ term in $\mathcal{T}(\Gamma^B)$ can be written uniquely as $\sigma_0\sigma_1 \cdots \sigma_n$ for some n, where $\sigma_i \in \mathcal{T}(\Gamma^B)$ for $1 \le i \le n$ and $\sigma_0 \in \Gamma^B$. Thus, we will often write an applicative term in $\mathcal{T}(\Gamma^B)$ as $s^\beta\sigma_1 \cdots \sigma_n$ for some $s \in \Gamma$, $\beta \in B$ and $\sigma_i \in \mathcal{T}(\Gamma^B)$ for $1 \le i \le n$.

Pointer Machine Stacks

Definition 3.3.1. Given a word grammar $W = \langle N, \Sigma, V, \mathcal{R}, S, e \rangle$ a ***pointer machine stack*** is a non-empty list:

$$[a_n, a_{n-1}, \cdots, a_1]$$

where $n \ge 1$ such that each element a_i for $i = 1, \cdots, n$ is an applicative term in $\mathcal{T}^o(N^{B_i} \cup V^{B_i})$ or is an element of e^{B_i}. Here, B_i is defined as the following set:

$$\{\beta : \beta \sqsupseteq [a_{i-1}, \cdots, a_1]\}$$

In other words, B_i is the set of all suffixes[2] of $[a_{i-1}, \cdots, a_1]$. Note that this is a mutually recursive definition of a stack item and the stack.

[2]For example, given $[3, 2, 1]$, the set of suffixes are:

$$\{[3, 2, 1], [2, 1], [1], [\ ]\}.$$

The following operations are defined on a non-empty stack:

$$\begin{aligned}\mathsf{head}\,[a_n, a_{n-1}, \cdots, a_1] &= a_n\\ a : [a_n, a_{n-1}, \cdots, a_1] &= [a, a_n, a_{n-1}, \cdots, a_1] \text{ for } a \in \mathcal{T}^o(N^B \cup V^B) \cup e^B.\end{aligned}$$

where $B = \{\beta : \beta \sqsupseteq [a_n, a_{n-1}, \cdots, a_1]\}$.

Furthermore, for a pointer machine stack containing at least 1 element, the following operation is also defined:

$$\mathsf{tail}\,[a_n, a_{n-1}, \cdots, a_1] = [a_{n-1}, \cdots, a_1]$$

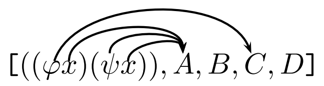
Remark 3.3.2. We will see that if we ever have a pointer machine stack of length 1, then it will always be equal to $[S^{[\,]}]$. Note that $[\,]$ is the *only* possibility for the superscript by the above definition.

Example 3.3.3. As an example:

$$[((\varphi^{p_C} x^{p_A})(\psi x)^{p_A}), A, B, C, D]$$

where $p_A = [A, B, C, D]$, $p_C = [C, D]$. Note that we have made use of the notational saver where ψ^{p_A} and x^{p_A} in the expression (ψx) both have the same pointer, and therefore write it as $(\psi x)^{p_A}$.

The graphical representation would be:



Remark 3.3.4. The above illustration should give the intuition behind the choice of the name "pointer."

Definition 3.3.5. ***Transitions of the pointer machine.*** In the following, let β denote the current pointer machine stack. The next transition depends on $\mathsf{head}\,\beta$. We assume that $s_0^{\beta_0}\sigma_1 \cdots \sigma_n = \mathsf{head}\,\beta$, where $s_0 \in N \cup V \cup \{e\}$ and β_0 is a proper suffix of β. We perform a case analysis on s_0:

1. *Expand.* Assume $s_0 = F$ for some non-terminal $F \in N$. If $F x_1 \cdots x_n \xrightarrow{\alpha} r$ is a production rule of W then
$$\beta \xrightarrow{\alpha} r^\beta : \beta.$$

2. *Search.* Assume s_0 is a variable x where $\mathsf{level}(s_0) > 0$ and that $\mathsf{head}\,\beta_0 = F^{\gamma_0}\tau_1 \cdots \tau_l$. Furthermore, suppose that x is the ith formal parameter of F (from the left). Then:
$$\beta \xrightarrow{\epsilon} \tau_i\sigma_1\sigma_2 \cdots \sigma_n : \mathsf{tail}\,\beta.$$

3. *Compact.* Assume s_0 is a variable x of level 0. Furthermore suppose that $\mathsf{head}\,\beta_0 = F^{\gamma_0}\tau_1 \cdots \tau_l$ and that x is the ith formal parameter of F (from the left) and $\tau_i = \tau_{i0}\tau_{i1} \cdots \tau_{ik}$ where $\tau_{i0} = t^{\gamma_0}$ for some $t \in N \cup V$. Then:
$$\beta \xrightarrow{\epsilon} \tau_i : \gamma_0.$$

4. *Halt.* Assume $s_0 = e$, then the pointer machine halts.

Definition 3.3.6. A pointer machine stack β is ***reachable*** if and only if

- $\beta = [S^{[]}]$; or
- β' is reachable and $\beta' \xrightarrow{\alpha} \beta$ for one of the above rules.

We write $\twoheadrightarrow$ for the reflexive and transitive closure of $\rightarrow$.

Definition 3.3.7. We define the language of pointer machine as follows:

$$\{w_0 w_1 \cdots w_n : [S^{[]}] \xrightarrow{w_0} \beta_1 \xrightarrow{w_1} \beta_2 \xrightarrow{w_2} \cdots \xrightarrow{w_n} \beta_{n+1}\}$$

where $\mathsf{head}\,\beta_{n+1} = e^{\beta}$ for some β. Note that in $w_0 w_1 \cdots w_n$ we express concatenation by juxtaposition.

3.3.2 Useful Properties

Before we prove that a pointer machine implements word grammars, we consider a few useful properties.

Pointers

The stack items of the pointer machine are of two types: *incomplete* and *complete*. A complete item $s_0^{\beta_0}\sigma_1 \cdots \sigma_n$ is one that is headed by a non-terminal, i.e. $s_0 = F$ for some $F \in N$. An incomplete item is headed by a variable. We call it incomplete, because it is, in a sense, work in progress: we will not be able to output any non-ϵ symbols until (after some number of *Search* or *Compact* operations) we reach a stack configuration where the head item is headed by a non-terminal.

Each of the lemmas proved here follows by a straightforward induction on the number of stack transitions and therefore their proofs are omitted.

Lemma 3.3.8. *In the following, let σ be an item in the reachable pointer machine stack β:*

(i) *σ has type level 0;*

(ii) $\mathsf{tail}\,\beta$ *consists of only complete items;*

(iii) *If x^{γ} for some $x \in V$ occurs in σ, then if* $\mathsf{head}\,\gamma = F^{\delta_0}\tau_1 \cdots \tau_k$, *$x$ is a formal parameter of the non-terminal F.*

Lemma 3.3.9. *Let $\sigma = \sigma_0\sigma_1 \cdots \sigma_m$ be an item in a reachable pointer machine stack β. Then $\sigma_0\sigma_1 \cdots \sigma_m$, can be uniquely written as either: (1) e^{β_0} where β_0 is a proper suffix of β or (2) $s_0^{\beta_0} s_1^{\beta_1} \cdots s_m^{\beta_m}$ where $s_i \in \mathcal{T}(N \cup V) \cup \{e\}$ and β_i is a proper suffix of β for each $1 \leq i \leq m$ and $s_0 \in N \cup V$.*

The above says that for a stack item $\sigma_0\sigma_1 \cdots \sigma_m$ in a reachable pointer machine stack β, then all the symbols that occur in σ_i (for a given $0 \leq i \leq m$) share the same pointer.

Remark 3.3.10. From this point onwards, we will generally write stack items as $t_0^{\beta_0} t_1^{\beta_1} \cdots t_m^{\beta_m}$ where each $t_i \in \mathcal{T}(N \cup V)$ and the β_i are pointers. For example, consider the example used earlier : $F^\beta \varphi^\beta x^\gamma(\varphi^\gamma(\varphi^\gamma x^\gamma))$, and let us suppose that $\gamma \neq \beta$, we can write this as: $F^\beta \varphi^\beta x^\gamma(\varphi(\varphi x))^\gamma$ or simply as $(F\varphi)^\beta x^\gamma(\varphi(\varphi x))^\gamma$. Note that in the latter we have made use of the fact that $F\varphi$ is a subterm of the expression such that both share the same pointer, β.

Lemma 3.3.11. *Let $s_0^{\beta_0} s_1^{\beta_1} \cdots s_m^{\beta_m}$ be an item in a reachable pointer machine stack β. If $i < j$, then $\beta_i \sqsupseteq \beta_j$. In other words, β_i points deeper back than does β_j.*

Lemma 3.3.12. *Let $\beta = [a_n, a_{n-1}, \cdots, a_1]$ be a reachable pointer machine stack. Then, for any $a_i = s_0^{\beta_0} s_1^{\beta_1} \cdots s_m^{\beta_m}$ it is always the case that $\beta_m = [a_{i-1}, \cdots, a_1]$.*

Stack Alphabet

Let us consider the stack alphabet *without* pointer information. In particular, if $s_0^{\beta_0} s_1^{\beta_1} \cdots s_m^{\beta_m}$ is an item in a reachable pointer machine stack, then $s_0 s_1 \cdots s_m$ is the result of removing pointer information. It is easy to show that the stack alphabet *without* pointer information is finite.

We write R for the set consisting of the rhs of each rule from $\mathcal{R}$. Let $\xi E_0 \cdots E_m$ be a ground-type term, where ξ is either a variable or a non-terminal. We define

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

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

$$\overline{R \subseteq \Gamma}$$

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

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

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

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

Lemma 3.3.13. Γ *is finite.*

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

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

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

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

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

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

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

3.3.3 Proof that Pointer Machines Implement Word Grammars

Pointer machines are a very natural implementation of word grammars – a moment of experimenting with them should convince the reader as such. However, for the sake of completeness we prove this formally.

Definition 3.3.14. Let β be a reachable configuration of the pointer machine induced by a word grammar W. We define the following operation on β:

$$[\![\beta]\!] = [\![\mathsf{head}\,\beta]\!]$$

where $[\![\sigma]\!]$ is defined by induction on a pointer machine stack *item* σ as follows. Recall that σ can be *uniquely* written as either (1) e^{β_0} or as (1) $s_0^{\beta_0} s_1^{\beta_1} \cdots s_m^{\beta_m}$ where $s_0 \in N \cup V$ and $s_i \in \mathcal{T}(N \cup V)$ for $1 \leq i \leq m$:

1. $[\![s_0^{\beta_0} s_1^{\beta_1} \cdots s_m^{\beta_m}]\!] = (([\![s_0^{\beta_0}]\!][\![s_1^{\beta_1}]\!]) \cdots [\![s_m^{\beta_m}]\!])$ for $m > 1$;
2. $[\![x^{\beta}]\!] = [\![t_i^{\beta_i}]\!]$ for x a variable and $\mathsf{head}\,\beta = F^{\beta_0} t_1^{\beta_1} \cdots t_n^{\beta_n}$ and x corresponds to the ith variable of F;
3. $[\![F^{\beta}]\!] = F$ for F a non-terminal;
4. $[\![e^{\beta}]\!] = e$, where e is the end-of-word character;

Lemma 3.3.15. *Let $W = \langle N, \Sigma, V, S, \mathcal{R}, e \rangle$ and let β be a reachable pointer machine stack, then:*

$$[\![r^{\beta}]\!] = r[[\![t_1^{\beta_1}]\!]/x_1 \cdots [\![t_n^{\beta_n}]\!]/x_n]$$

for any $r \in \mathcal{T}(N \cup V) \cup \{e\}$ such that $FV(r) \subseteq \{x_1, \cdots, x_n\}$, where $F x_1 \cdots x_n$ is the left hand side of a production rule in $\mathcal{R}$ and $\mathsf{head}\,\beta = F^{\beta_0} t_1^{\beta_1} \cdots t_n^{\beta_n}$.

Proof. Induction on the structure of r.

- $r = e$. Trivial.
- $r = F$. Trivial.
- $r = x_i$. We have $[\![x_i^{\beta}]\!] = [\![t_i^{\beta_i}]\!] = x_i[[\![t_1^{\beta_1}]\!]/x_1 \cdots [\![t_n^{\beta_n}]\!]/x_n]$
- $r = s_0 s_1 \cdots s_m$, where $m > 0$ and $s_0 \in N \cup V$. We have $[\![r^{\beta}]\!] = [\![(s_0 s_1 \cdots s_m)^{\beta}]\!] = [\![s_0^{\beta} s_1^{\beta} \cdots s_m^{\beta}]\!] = (([\![s_0^{\beta}]\!][\![s_1^{\beta}]\!]) \cdots [\![s_m^{\beta}]\!])$. The result now follows by the induction hypothesis and the definition of substitution.

$\square$

Proposition 3.3.16. *Let β be a reachable configuration of the pointer machine induced by the word grammar W. Then $S \twoheadrightarrow [\![\beta]\!]$.*

Proof. Induction on the number of transition steps required to reach β.

- (Base case) With zero steps $\beta = \texttt{[}S^{\texttt{[]}}\texttt{]}$ and $[\![\beta]\!] = S$ therefore trivially true.
- (Inductive step) Suppose true for n and suppose that $\beta \to \beta'$, we perform a case analysis on the last transition step (from β to β'):
  - $\mathsf{head}\,\beta = F^{\beta_0} t_1^{\beta_1} \cdots t_m^{\beta_m}$. If $F x_1 \cdots x_m \xrightarrow{\alpha} r$ is a production rule then we have that
  $$\beta \xrightarrow{\alpha} r^{\beta} : \beta.$$
  We now note that:
  $$\begin{array}{rcl} [\![\beta]\!] & = & [\![\mathsf{head}\,\beta]\!] \\ & = & [\![F^{\beta_0} t_1^{\beta_1} \cdots t_m^{\beta_m}]\!] \\ & = & [\![F^{\beta_0}]\!][\![t_1^{\beta_1}]\!] \cdots [\![t_m^{\beta_m}]\!] \\ & \xrightarrow{\alpha} & r\ [[\![t_1^{\beta_1}]\!]/x_1 \cdots [\![t_m^{\beta_m}]\!]/x_m] \\ & = & [\![r^{\beta}]\!] \text{ by Lemma 3.3.15.} \\ & = & [\![r^{\beta} : \beta]\!] \end{array}$$
  - $\mathsf{head}\,\beta = x^{\beta_0} s_1^{\beta_1} \cdots s_m^{\beta_m}$ where x is a variable of level > 0 and $\mathsf{head}\,\beta_0 = F^{\gamma_0} t_1^{\gamma_1} \cdots t_l^{\gamma_l}$. Furthermore, suppose that x is the ith formal parameter of F (from the left) and $t_i = u_0 u_1 \cdots u_k \in \mathcal{T}(N \cup V)$. Then:
  $$\beta \xrightarrow{\epsilon} u_0^{\gamma_i} u_1^{\gamma_i} \cdots u_k^{\gamma_i} s_1^{\beta_1} \cdots s_m^{\beta_m} : \mathsf{tail}\,\beta.$$
  We now have:
  $$\begin{array}{rcl} [\![\beta]\!] & = & [\![\mathsf{head}\,\beta]\!] \\ & = & [\![x^{\beta_0} s_1^{\beta_1} \cdots s_m^{\beta_m}]\!] \\ & = & [\![x^{\beta_0}]\!][\![s_1^{\beta_1}]\!] \cdots [\![s_m^{\beta_m}]\!] \\ & = & [\![u_0^{\gamma_i}]\!][\![u_1^{\gamma_i}]\!] \cdots [\![u_k^{\gamma_i}]\!][\![s_1^{\beta_1}]\!] \cdots [\![s_m^{\beta_m}]\!] \\ & = & [\![u_0^{\gamma_i} u_1^{\gamma_i} \cdots u_k^{\gamma_i} s_1^{\beta_1} \cdots s_m^{\beta_m}]\!] \\ & = & [\![u_0^{\gamma_i} u_1^{\gamma_i} \cdots u_k^{\gamma_i} s_1^{\beta_1} \cdots s_m^{\beta_m} : \mathsf{tail}\,\beta]\!] \end{array}$$
  - $\mathsf{head}\,\beta = x_i$ where $\mathsf{level}(x_i) = 0$. The proof for this case is very similar to the above and therefore omitted.
  - $\mathsf{head}\,\beta = e$. Untenable; there are no further transitions.

$\square$

Remark 3.3.17. The reader should be able to deduce that one transition of the pointer machine either corresponds to one application of a rule in $\mathcal{R}$, or none.

Claim 3.3.18. *Let β be a reachable configuration of the pointer machine induced by the word grammar W. If $\mathsf{head}\,\beta = \varphi^{\beta_0} t_1^{\beta_1} \cdots t_n^{\beta_n}$, where $n \geq 0$ and $\varphi \in V$ there exists a* finite *sequence or pointer machine stacks $\gamma_1, \gamma_2, \cdots, \gamma_k$ where $\gamma_1 = \beta$ and the following holds:*

$$[\![\gamma_1]\!] = [\![\gamma_2]\!] = \cdots = [\![\gamma_k]\!]$$

Furthermore, γ_k is headed by a non-terminal.

Proof. This follows from a closer inspection of 2 of the cases of the previous lemma. One merely needs to note that if β is a reachable pointer machine stack and $\mathsf{head}\,\beta = \varphi^{\beta_0} t_1^{\beta_1} \cdots t_n^{\beta_n}$, then the next reachable configuration has the same semantic meaning under $[\![\cdot]\!]$, i.e. $[\![\gamma_1]\!] = [\![\gamma_2]\!]$. If this next reachable configuration is again headed by a variable, then the same holds of the next reachable configuration and so forth $[\![\gamma_1]\!] = [\![\gamma_2]\!] = [\![\gamma_3]\!] = \cdots$. However, this process can only be iterated finitely many times. □

Proposition 3.3.19. *Let β be a reachable configuration of the pointer machine induced by the word grammar W. If $[\![\beta]\!] \xrightarrow{\alpha} r$, then there exists a sequence of configurations $\beta = \beta_0 \to \beta_1 \to \cdots \to \beta_n$ such that $[\![\beta_0]\!] = [\![\beta_1]\!] = \cdots = [\![\beta_{n-1}]\!]$ and $[\![\beta_n]\!] = r$.*

Proof. We consider $\mathsf{head}\,\beta$.

1. $\mathsf{head}\,\beta = F t_1^{\beta_1} \cdots t_m^{\beta_m}$. Clearly, $[\![\beta]\!] = F[\![t_1^{\beta_1}]\!][\![t_2^{\beta_2}]\!] \cdots [\![t_m^{\beta_m}]\!]$. Suppose $F x_1 x_2 \cdots x_m \xrightarrow{\alpha} r$. Therefore we can have that $F[\![t_1^{\beta_1}]\!][\![t_2^{\beta_2}]\!] \cdots [\![t_m^{\beta_m}]\!] = r[[\![t_1^{\beta_1}]\!]/x_1 [\![t_2^{\beta_2}]\!]/x_2 \cdots [\![t_m^{\beta_m}]\!]/x_m] = [\![r^\beta : \beta]\!]$, and of course $\beta \xrightarrow{\alpha} r^\beta : \beta$.

2. On the other hand, suppose that $\mathsf{head}\,\beta$ is headed by a variable. The result now holds with the help ofthe preceding lemma.

□

Corollary 3.3.20. *Let W be a word grammar, then w is in $L(W)$ if and only if w is generated by the pointer machine induced by W.*

Proof. This now follows from Proposition 3.3.16 and Proposition 3.3.19 and noting that $[S^{[]}]$ is a reachable pointer machine stack. □

Example 3.3.21. We conclude with an example. We consider the pointer machine induced by the unsafe grammar that defines U. We derive the word $(()*$:

$$
\begin{aligned}
& [S^{[]}] \\
\xrightarrow{(}\ & [(DGEEE)^{\beta_1}, S^{[]}] \\
\xrightarrow{(}\ & [(D(D\varphi x)z(Fy)(Fy))^{\beta_2}, (DGEEE)^{\beta_1}, S^{[]}] \\
\xrightarrow{)}\ & [(\varphi y x)^{\beta_3}, (D(D\varphi x)z(Fy)(Fy))^{\beta_2}, (DGEEE)^{\beta_1}, S^{[]}] \\
\xrightarrow{\epsilon}\ & [D^{\beta_2}\varphi^{\beta_2}x^{\beta_2}y^{\beta_3}x^{\beta_3}, (D(D\varphi x)z(Fy)(Fy))^{\beta_2}, (DGEEE)^{\beta_1}, S^{[]}] \\
\xrightarrow{*}\ & [z^{\beta_4}, D^{\beta_2}\varphi^{\beta_2}x^{\beta_2}y^{\beta_3}x^{\beta_3}, (D(D\varphi x)z(Fy)(Fy))^{\beta_2}, (DGEEE)^{\beta_1}, S^{[]}] \\
\xrightarrow{\epsilon}\ & [x^{\beta_3}, (D(D\varphi x)z(Fy)(Fy))^{\beta_2}, (DGEEE)^{\beta_1}, S^{[]}] \\
\xrightarrow{\epsilon}\ & [z^{\beta_2}, (DGEEE)^{\beta_1}, S^{[]}] \\
\xrightarrow{\epsilon}\ & [E^{\beta_1}, S^{[]}] \\
\xrightarrow{\epsilon}\ & [e^{\beta_5}, E^{\beta_1}, S^{[]}]
\end{aligned}
$$

where:

$$\begin{aligned}
\beta_1 &= [S^{[]}] \\
\beta_2 &= [(DGEEE)^{\beta_1}, S^{[]}] \\
\beta_3 &= [(D(D\varphi x)z(Fy)(Fy))^{\beta_2}, (DGEEE)^{\beta_1}, S^{[]}] \\
\beta_4 &= [D^{\beta_2}\varphi^{\beta_2}x^{\beta_2}y^{\beta_3}x^{\beta_3}, (D(D\varphi x)z(Fy)(Fy))^{\beta_2}, (DGEEE)^{\beta_1}, S^{[]}] \\
\beta_5 &= [E^{\beta_1}, S^{[]}]
\end{aligned}$$

3.4 Equivalence at Level 2

We are now in a position to present the main result of this chapter.

Theorem 3.4.1. *Let W be a level-2 word grammar (not assumed to be safe). There exists a level-2 non-deterministic pushdown automaton that accepts the same language. Moreover, this conversion is effective.*

The proof is split into two transformations. Given a level-2 word-grammar W we show:

1. The pointer machine for W is simulated by a *level-2 pushdown automaton with links* (2PDAL), where the 2PDAL is a machine that has yet to be introduced.

2. We then show that the 2PDAL for W constructed in the first part can be simulated by a non-deterministic 2PDA.

Combining our result with that of [DG86], we have that every unsafe level-2 word grammar can be rewritten as a safe level-2 word grammar.

3.4.1 Simulating 2PMs by 2PDALs

Understanding Knapik et al.'s Construction

Ultimately, our proof is an extension of the method employed by Knapik et al. [KNU02] where they show that safe level-n grammars can be simulated by level-n pushdown automata. Whilst we wish to avoid a replication of their proof in this thesis, we would like remind the reader of the intuition behind it. We do so by means of an example with detailed comments, however, we will be applying their result to an unsafe grammar. Not only will this illustrate why it fails in the unsafe case, but our analysis will highlight what can be done to rectify it. This is what also provides the inspiration for the level-2 pushdown automaton with links we are about to introduce.

Remark 3.4.2. Recall, however, that Knapik et al.'s [KNU02] result on the equivalence between safe higher-order recursion schemes and higher-order pushdown automaton has a predecessor; namely Damm and Goerdt's [DG86] corresponding result for the OI hierarchy of word languages.

Given a level-n *safe* recursion scheme, Knapik et al. construct a level-n pushdown automaton that accepts the unique term tree generated by the scheme. It is straightforward to reformulate this proof in our setting of word-grammars.

Theorem 3.4.3. *Let W be a safe level-n word-grammar with language $L(W)$. Then $L(W)$ is accepted by some nPDA.*

Proof. We use exactly the same setup as in Section 5.2 of [KNU02], but now we incorporate an input string over the alphabet Σ. Recall that there are 6 cases for the transition function; they are given in Fig. 3.1. $\square$

$$\begin{array}{rcll}
(q_0, a, Dt_1 \cdots t_r) & \to & (q_0, \mathsf{push}_1(E)) \text{ if } Dx_1 \cdots x_m \xrightarrow{a} E,\ r \leq m & (R1) \\
(q_0, \epsilon, e) & \to & \text{accept} & (R2) \\
(q_0, \epsilon, x_j) & \to & (q_j, \mathsf{pop}_1) \text{ if } x_j : o & (R3) \\
(q_0, \epsilon, x_j t_1 \cdots t_r) & \to & (q_j, \mathsf{push}_{n-k+1}\ ;\ \mathsf{pop}_1) \text{ if } x_j \text{ has level } k > 0 & (R4) \\
1 < j \leq r, (q_j, \epsilon, \$t_1 \cdots t_r) & \to & (q_0, \mathsf{pop}_1\ ;\ \mathsf{push}_1(t_j)) & (R5) \\
j > r, (q_j, \epsilon, \$t_1 \cdots t_r) & \to & (q_{j-r}, \mathsf{pop}_{n-k+1}) \text{ if } \$t_1 \cdots t_r \text{ has level } k > 0 & (R6)
\end{array}$$

Figure 3.1: Adapted transition rules from [KNU02].

Convention. In the Figure, \$ serves as a placeholder for either a non-terminal or a variable. Furthermore, x_j means the j-th formal parameter of the relevant non-terminal.

Before examining why their proof fails if we attempt to apply it (blindly) to a level-2 unsafe grammar, let us briefly remind the reader of the intuition behind Knapik et al.'s proof [KNU02]. Their proof idea is based on that of [KU88], where Kfoury and Urzyczyn show how to implement recursive programs of level 2 with a level-2 pushdown store.

In Knapik et al.'s formulation, the automaton possesses states $\{q_0, q_1, \cdots q_M\}$ along with some auxiliary states. The stack alphabet Γ consists of the subterms of the right hand sides of the production rules. Given an input word w, the pushdown automaton mimics an OI reduction sequence over w should one exist (and will result in an abortive computation in case there is no derivation for w). Generally speaking, if the automaton is in state q_0 (to be interpreted as the default state of operation) with topmost symbol $\$s_1 \cdots s_r$, then $\$s_1 \cdots s_r$ is an *approximation* of the current sentential form of the derivation sequence – note that this is necessarily an approximation for the alphabet must be finite. If the approximation is evaluated in the correct environment (namely the rest of the pushdown store) then this evaluation would result in the exact sentential form. On the other hand, if the automaton is in state q_j for some $j > 0$ reading $\$s_1 \cdots s_r$, then we are interested in evaluating the jth argument of \$ from the left.

We examine why their proof fails if we attempt to apply it blindly to a level-2 unsafe grammar. We use the grammar below:

Example 3.4.4. Consider the following level-2 word grammar, where

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

the typed non-terminals are

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

with rules:

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

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

We are getting ahead of ourselves slightly here, but we will see (later in the chapter) that this a deterministic word grammar and this means that each word in the language has a unique derivation. Hence, it should be easy to check by hand that the the words $h_1h_3h_2f_1b$ and $h_1h_3h_2f_2a$ are part of the language, whereas $h_1h_3h_2f_1a$ is not.

We take the induced level-2 pushdown automaton for the grammar above and attempt to accept the word $h_1h_3h_2f_1a$; as indicated above this should not be accepted.

The automaton starts in the configuration $(q_0, h_1h_3h_2f_1a, \texttt{[[}S\texttt{]]})$, after a few steps the following configuration is reached:

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

As the topmost item, φB, is headed by a level-1 variable, we need to find out what φ is in order to proceed. Note that φ is the 1st formal parameter of the preceding item: $D(D\varphi x)y(\varphi y)$, i.e., it refers to $D\varphi x$. To this end, we perform a push_2 and then perform a pop_1, and replace the topmost item with $D\varphi x$. In other words, we have applied rule $(R4)$ followed by rule $(R5)$ to arrive at:

$$\begin{array}{lll}(q_0, & h_2f_1a, & \texttt{[[}(D\varphi x)^{\langle 1-\rangle}, DGAB, S\texttt{]},\\ & & \texttt{[}\varphi B^{\langle 1+\rangle}, D(D\varphi x)y(\varphi y), DGAB, S\texttt{]]})\end{array}$$

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

The crux behind Knapik et al.'s construction is the following. Suppose we meet the item $D\varphi x^{\langle 1-\rangle}$ later on in the computation, and suppose that we would like to request its third argument, meaning we would be in state q_3. Note, however, that $D\varphi x^{\langle 1-\rangle}$ has only 2 arguments. The missing argument can be found by visiting the item $\varphi B^{\langle 1+\rangle}$. Hence the labelling. We need to ensure that there is a systematic way to get from $D\varphi x^{\langle 1-\rangle}$ to $\varphi B^{\langle 1+\rangle}$ whenever we are in a state q_n for $n > 2$ and we have $D\varphi x^{\langle 1-\rangle}$ as our topmost symbol. This systematic way suggested by [KNU02] is embodied by rule $(R6)$ of Fig. 3.1. In particular, it says that all we need to do is perform a pop_2, followed by a change in state to q_{n-2}, and to repeat if necessary. Note that if we applied rule $(R6)$ to the current configuration, we would indeed be brought to the right place, $\varphi B^{\langle 1+\rangle}$. We will see, however, that with an unsafe grammar, this invariant may be violated.

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

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

Therefore, we next get:

$$\begin{array}{lll}(q_0, & f_1 a, & \texttt{[[}Fy^{\langle 2-\rangle}, (D\varphi x)^{\langle 1-\rangle}, DGAB, S\texttt{]},\\ & & \texttt{[}\varphi x^{\langle 2+\rangle}, H(Fy)x, (D\varphi x)^{\langle 1-\rangle}, DGAB, S\texttt{]},\\ & & \texttt{[}\varphi B^{\langle 1+\rangle}, D(D\varphi x)y(\varphi y), DGAB, S\texttt{]]})\end{array}$$

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

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

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

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

By rule ($R6$) we arrive at: (in the following $\rightarrow_n$ means n steps of $\rightarrow$)

$$\begin{array}{lll}
(q_1, & a, & \texttt{[}\texttt{[}\varphi x^{\langle 2+\rangle}, H(Fy)x, (D\varphi x)^{\langle 1-\rangle}, DGAB, S\texttt{]}, \\
 & & \texttt{[}\varphi B^{\langle 1+\rangle}, D(D\varphi x)y(\varphi y), DGAB, S\texttt{]}\texttt{]}) \\
\rightarrow_1 (q_0, & a, & \texttt{[}\texttt{[}x, H(Fy)x, (D\varphi x)^{\langle 1-\rangle}, DGAB, S\texttt{]}, \\
 & & \texttt{[}\varphi B^{\langle 1+\rangle}, D(D\varphi x)y(\varphi y), DGAB, S\texttt{]}\texttt{]}) \\
\rightarrow_1 (q_2, & a, & \texttt{[}\texttt{[}H(Fy)x, (D\varphi x)^{\langle 1-\rangle}, DGAB, S\texttt{]}, \\
 & & \texttt{[}\varphi B^{\langle 1+\rangle}, D(D\varphi x)y(\varphi y), DGAB, S\texttt{]}\texttt{]}) \\
\rightarrow_2 (q_2, & a, & \texttt{[}\texttt{[}(D\varphi x)^{\langle 1-\rangle}, DGAB, S\texttt{]}, \\
 & & \texttt{[}\varphi B^{\langle 1+\rangle}, D(D\varphi x)y(\varphi y), DGAB, S\texttt{]}\texttt{]}) \\
\rightarrow_2 (q_2, & a, & \texttt{[}\texttt{[}DGAB, S\texttt{]}, \\
 & & \texttt{[}\varphi B^{\langle 1+\rangle}, D(D\varphi x)y(\varphi y), DGAB, S\texttt{]}\texttt{]}) \\
\rightarrow_2 (q_0, & \epsilon, & \texttt{[}\texttt{[}e, S\texttt{]}, \\
 & & \texttt{[}\varphi B^{\langle 1+\rangle}, D(D\varphi x)y(\varphi y), DGAB, S\texttt{]}\texttt{]})
\end{array}$$

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

- We define the level-2 pushdown automaton with links; this not only permits us the use of links but also has the atomic operation of following a link. We show that this machine can simulate level-2 word grammars.
- More interestingly, we show that the way a level-2 pushdown automaton with links simulates a level-2 word grammar can be captured by a level-2 non-deterministic pushdown automaton.

2PDAL: 2PDA with Links

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

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

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

$$\begin{array}{rlll} & (q_1, & a, & [\,[\varphi B^{\langle 1+\rangle}, D(D\varphi x)y(\varphi y), DGAB]\,]) \\ \to_1 & (q_0, & a, & [\,[B, D(D\varphi x)y(\varphi y), DGAB]\,]) \end{array}$$

which is indeed what we wanted, and the word is rejected, as the only rule for B is $B \xrightarrow{b} e$. In fact, using the same computation, but on the word $h_1h_3h_2f_1b$ we end up in an accepting state.

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

Level-2 Pushdown Store with Links

Like the level-2 pushdown automaton, the level-2 pushdown automaton with links will make use of a level-2 store. However, as might be deduced from the preceding examples, the store alphabet must now accommodate for links.

To represent links in the store, we develop the concept of labeling store items. Formally, the set of labels is given by the set:

$$\mathsf{LABELS} := \{n+ : n \in \omega\} \cup \{n- : n \in \omega\}$$

where ω is the set of natural numbers.

Given a finite *base* stack alphabet Γ, each item of the level-2 store will be an element of the pointed alphabet Γ^β, where $\beta = 2^{\mathsf{LABELS}}$. Note that an element of β is, in fact, a set, for example $\langle\, 3-, 4+, 7- \,\rangle \in \beta$. We can thus equate this representation with $\langle\, 4+, 3-, 7- \,\rangle$ or with $\langle\, 3- \,\rangle \cup \langle\, 4+, 7- \,\rangle$ and so forth.

Given an element $s^\lambda \in \Gamma^\beta$, we refer to λ as the *label set* of s. If $m-$ (or $m+$) is contained in λ, then we say that s *has* label $m-$ (respectively $m+$).

Intuitively, a link is a connection between two items in the store. When a link is first created it is given a unique name, and this takes on the form of a natural number, say m. One of the items that comprises this links is referred to as the *start* of the link m and receives the label $m-$. The other item is referred to as the *end* of the link m and receives the label $m+$. For the sake of simplicity, we represent $s^{\langle\,\rangle}$ (i.e. s has no labels) as simply s.

Store Operations

The set of store operations on a level-2 store Op_2 is defined below, where $s = [s_n, s_{n-1}, \cdots, s_1]$ and $s_n = [a_m, a_{m-1}, \cdots, a_1]$, where each $a_i \in \Gamma^\beta$. First, the following operations are held in common with the 2PDA:

$$\begin{aligned}
\mathsf{push}_1(a)s &= [[a, a_m, a_{m-1}, \cdots, a_1], s_{n-1}, \cdots, s_1] \text{ where } a \in \Gamma \\
\mathsf{push}_2 s &= [s_n, s_n, s_{n-1}, \cdots, s_1] \\
\mathsf{pop}_1 s &= [[a_{m-1}, \cdots, a_1], s_{n-1}, \cdots, s_1] \\
\mathsf{pop}_2 s &= [s_{n-1}, \cdots, s_1]
\end{aligned}$$

Note that in the above, the $\mathsf{push}_1(a)$ operation only pushes an element of Γ and *not* of Γ^β. The labeling of items to represent start and end points of links are handled exclusively by the two new operations we introduce: $\mathsf{pushlink}_2$ and follow_2 defined below.

Let $s = [s_n, s_{n-1}, s_{n-2}, \cdots, s_1]$:

$$\mathsf{pushlink}_2(a)\ s = [s'_n, s'_{n-1}, s_{n-2}, \cdots, s_1] \text{ where } a \in \Gamma.$$

If $s_n = [a_m, a_{m-1}, \cdots, a_1]$, then $s'_n = [a^{\langle p- \rangle}, a_m, a_{m-1}, \cdots, a_1]$. As for s'_{n-1}, suppose that $\mathsf{top}_1(s_{n-1}) = b^\gamma$, then s'_{n-1} is exactly the same as s_{n-1} except that $\mathsf{top}_1(s'_{n-1}) = b^{\gamma \cup \langle p+ \rangle}$. Here, $p \in \omega$ is a fresh name for a link. By fresh, we mean that it is not currently in use and *has not been used in the past.*

In other words, the operation $\mathsf{pushlink}_2(a)$ has the effect of performing a $\mathsf{push}_1(a)$, with the added effect of creating a new link (here, given the name p) between this newly pushed item and the 1-store directly beneath it.

Next we define the follow_2 operation, where s is a 2-store over Γ^β.

$$\mathsf{follow}_2\ (p)\ s = \begin{cases} s & \text{if } \mathsf{top}_1(s) = a^{\lambda \cup \langle p+ \rangle} \\ \mathsf{follow}_2\ (p)\ (\mathsf{pop}_2(s)) & \text{otherwise.} \end{cases}$$

Note that follow_2 is really just a parameterised form of pop_2. We perform as many pop_2's as necessary until the topmost symbol is the end point of the link p, i.e. it contains the label $p+$ in its label set.

Example 3.4.5. Consider the following 2-store:

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

At the moment, no items have any labels. We perform a $\mathsf{pushlink}_2(b)$:

$$\begin{bmatrix} [& b^{\langle 1- \rangle}, & a^{\langle\rangle}, & b^{\langle\rangle}, & c^{\langle\rangle} &], \\ [& & d^{\langle 1+ \rangle}, & e^{\langle\rangle}, & f^{\langle\rangle} &] \end{bmatrix}$$

If we now perform a push_2 (below), the labels are, of course, copied:

$$\begin{bmatrix} [& b^{\langle 1- \rangle}, & a^{\langle\rangle}, & b^{\langle\rangle}, & c^{\langle\rangle} &], \\ [& b^{\langle 1- \rangle}, & a^{\langle\rangle}, & b^{\langle\rangle}, & c^{\langle\rangle} &], \\ [& & d^{\langle 1+ \rangle}, & e^{\langle\rangle}, & f^{\langle\rangle} &] \end{bmatrix}$$

If we now perform a $\mathsf{follow}_2(1)$, we return to:

$$\Big[\ \texttt{[}\ \ d^{\langle 1+\rangle},\ \ e^{\langle\,\rangle},\ \ f^{\langle\,\rangle}\ \ \texttt{]}\ \Big]$$

i.e. the 1-store that was directly below the item $b^{\langle 1-\rangle}$ when it was first created.

Remark 3.4.6. The above example illustrates that when a link is first created, say with name p, there exists exactly one start point (labeled with $p-$) and exactly one end point (labeled with $p+$). However, subsequent stack operations may mean that we end up in a situation where (1) there is exactly one end point but several start points for a given p, or (2) there is one (or more) end points of the link p, but with no start point. Both of these situations were illustrated with the above example. Note that it is never possible to have one or more start points of the link p but no end points.

Definition 3.4.7. An ***instance of the link*** p in level-2 pushdown store (of a 2PDAL) is any *pair* of items such that one is labeled with $p+$ and the other with $p-$.

The Level-2 Pushdown Automaton with Links

A level-2 pushdown automaton with links is given by the following tuple $\langle\, Q, \Sigma, \Gamma, \delta, q_0, \perp, F\,\rangle$, where:

- Q is a finite set of states where q_0 is the initial state and $F \subseteq Q$ is the final set of states;
- Σ is the finite input alphabet;
- Γ is the *base* stack alphabet (also finite), where $\perp \in \Gamma$ is the designated "bottom-of-store" symbol;
- δ is the transition function where $\delta \subseteq Q \times \Sigma \cup \{\epsilon\} \times \Gamma^{\beta} \times Q \times \mathsf{Op}_2$, where $\beta = 2^{\mathsf{LABELS}}$ and Op_2 were defined previously.

The definitions of a total configuration (given by a tuple (q, w, s) where q is the current state, w the portion of the input that remains to be read, and s the current 2-store), acceptance and the language accepted by a 2PDAL are identical to those of the 2PDA for string languages. Given a 2PDAL A, we denote by $L(A)$ the language accepted by the automaton A.

Simulating a Word Grammar W with a 2PDAL

Given a level-2 word grammar $W = \langle\, N, \Sigma, V, S, \mathcal{R}, e\,\rangle$, which is not assumed to be safe, we construct a 2PDAL, $2PDAL_W$ such that $L(2PDAL_W) = L(W)$. We adopt (w.l.o.g.) the following assumptions and conventions:

1. Each production rule of the grammar assumes the following format:

$$F\varphi_1 \cdots \varphi_m x_{m+1} \cdots x_{m+n} \xrightarrow{a} E \tag{3.2}$$

   where the φ's are used for level-1 parameters, and the x's are used for level-0 parameters. Furthermore, as with Knapik et al.'s construction [KNU02], each non-terminal expects at least one level-0 parameter.

2. As with Knapik et al.'s construction [KNU02] the start nonterminal S does not occur on the right hand side of any production rule.

Definition 3.4.8. Given the word grammar W, the corresponding 2PDAL is given by $2PDAL_W = \langle Q, \Sigma, \Gamma, \delta, q_0, S, F \rangle$, where:

1. As in [KNU02], the automaton works in phases beginning and ending in distinguished states q_i, with some auxiliary states in between. We assume, for the sake of clarity, that these auxiliary states are disjoint from $\{q_i : 0 \le i \le M\}$, where M is the maximal arity of any symbol occurring in $N \cup V$.
2. Γ consists of a subset of the subterms of the right hand sides of $\mathcal{R}$ and also includes S.
3. The set of accepting states F consists of a single accepting state.
4. Finally, δ is given in Fig. 3.2. Note that in this figure \$ is a placeholder for either a non-terminal or a variable.

$$
\begin{array}{rcll}
(q_0, a, Dt_1 \cdots t_n^\lambda) & \rightarrow & (q_0, \mathsf{push}_1(E)) \text{ if } Dx_1 \cdots x_m \xrightarrow{a} E, \text{ where } n \le m & \text{(L1)} \\
(q_0, \epsilon, e) & \rightarrow & \text{accept} & \text{(L2)} \\
(q_0, \epsilon, x_j) & \rightarrow & (q_j, \mathsf{pop}_1) & \text{(L3)} \\
(q_0, \epsilon, \varphi_j t_1 \cdots t_n^\lambda) & \rightarrow & (q_0, \mathsf{push}_2; \mathsf{pop}_1; \mathsf{pop}_1; \mathsf{pushlink}_2(s_j)) & \\
 & & \quad \text{where } Ds_1 \cdots s_{n'}{}^{\lambda'} \text{ precedes } \varphi_j t_1 \cdots t_n^\lambda & \text{(L4)} \\
 & & & \\
1 \le j \le n, (q_j, \epsilon, \$t_1 \cdots t_n^\lambda) & \rightarrow & (q_0, \mathsf{pop}_1; \mathsf{push}(t_j)) & \text{(L5)} \\
j > n, (q_j, \epsilon, \$t_1 \cdots t_n^\lambda) & \rightarrow & (q_{j-n}, \mathsf{follow}_2(m)) \text{ if } m- \in \lambda & \text{(L6)}
\end{array}
$$

Figure 3.2: Transition rules of the 2PDAL, $2PDAL_W$.

Proof of Correctness

Theorem 3.4.9. *The language generated by a (possibly unsafe) level-2 word grammar W is accepted by $2PDAL_W$.*

We will actually show that the pointer machine associated with the word grammar W can be simulated by the $2PDAL_W$. First, some notation. Let s be a 2-store, we define:

$$
\begin{array}{rcl}
s' \sqsubset s & \Longleftrightarrow & s' \text{ is obtained from } s \text{ by performing one or more } \mathsf{pop}_1\text{'s and } \mathsf{pop}_2\text{'s} \\
s' \sqsubset_i s & \Longleftrightarrow & s' \sqsubset s \text{ and the final action is a } \mathsf{pop}_1 \text{ (hence the "}i\text{" for internal)} \\
s' \sqsubset_1 s & \Longleftrightarrow & s' = \mathsf{pop}_1^n(s) \text{ for some } n > 0 \\
s' \sqsubset_2 s & \Longleftrightarrow & s' = \mathsf{pop}_2^n(s) \text{ for some } n > 0.
\end{array}
$$

For $\sqsubset, \sqsubset_1$ and $\sqsubset_2$ we define $\sqsubseteq$, $\sqsubseteq_1$ and $\sqsubseteq_2$ as the respective reflexive closures of the above.

We define a function $[\![\cdot]\!]$ that transforms a 2-store of $2PDAL_W$ to an equivalent stack of a 2PM as follows:

$$[\![s]\!] \;=\; \begin{cases} [S^{[\,]}] & \text{if } \mathsf{top}_1 s = S \\ (\mathsf{top}_1 s)^{[\![\mathsf{pop}_1 s]\!]} \;:\; [\![\mathsf{pop}_1 s]\!] & \text{elseif } \mathsf{top}_1 s : o \\ \mathsf{join}\;((\mathsf{top}_1 s)^{[\![\mathsf{pop}_1 s]\!]}, [\![\mathsf{follow}(m)\; s]\!]) & \text{elseif } \mathsf{top}_1 s \text{ has label } m-. \end{cases}$$

Note that in the above, $(\mathsf{top}_1 s)^{[\![\mathsf{pop}_1 s]\!]}$ is a term in $\mathcal{T}(\Gamma^B)$ where B is the set of pointer machine stacks. Recall that this notation implies that every symbol shares the same pointer, $[\![\mathsf{pop}_1 s]\!]$.

The auxiliary function join is defined below, where β ranges over 2-pointer machine stacks:

$$\mathsf{join}\;(\sigma, \beta) = \sigma u_1^{\beta_1} \cdots u_r^{\beta_r} : \mathsf{tail}\,\beta$$

where $\sigma \in \mathcal{T}(\Gamma^B)$ and $\mathsf{head}\,\beta$ can be uniquely written as $u_0^{\beta_0} u_1^{\beta_1} \cdots u_r^{\beta_r}$ with $u_0 \in N \cup V$. Note that the case where $r < 1$ never arises.

Lemma 3.4.10. *If (q_i, w, s) is a reachable configuration of $2PDAL_W$, then the following hold, where $s = [s_n, s_{n-1}, \cdots, s_1]$:*

(i) If $s' \sqsubseteq s$ and $\mathsf{top}_1 s' = u^\lambda$ such that $m- \in \lambda$, and $u : \tau$, then there exists unique $s'' \sqsubset_2 s'$ such that $\mathsf{top}_1 s'' = \varphi \cdots^{\lambda'}$, $m+ \in \lambda'$ and $\varphi : \tau$.

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

(iii) If $s' \sqsubseteq s$ and $\mathsf{top}_1(s') = D s_1 \cdots s_k^\lambda$ where D is a non-terminal, then $\mathsf{level}(D s_1 \cdots s_k) \leq 1$. Furthermore, if $\lambda = \langle m- \rangle$ for some m then $\mathsf{level}(D s_1 \cdots s_k) = 1$. If $\lambda = \langle\ \rangle$ then $\mathsf{level}(D s_1 \cdots s_k) = 0$. No other possibilities exist for λ in this case.

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

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

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

Proof. By induction on the number of transitions, where a transition constitutes the application of one rule in Fig. 3.2. □

Lemma 3.4.11. *Let (q_i, w, s) be a reachable configuration of $2PDAL_W$, and let*

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

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

Proof. Routine induction on the number of transitions. □

Lemma 3.4.12. *Let (q_i, w, s) be a reachable configuration of $2PDAL_W$ and let $s' \sqsubseteq s$. If pop_1 is defined on s' then*

$$[\![s']\!] = ((\mathsf{top}_1 s')^{[\![\mathsf{pop}_1 s']\!]}.\cdots) : \cdots : [\![\mathsf{pop}_1 s']\!]$$

In the case where pop_1 *is not defined, then we have* $[\![s']\!] = \texttt{[}S^{\texttt{[]}}\texttt{]}$.

Proof. By induction on the number of transitions, where a transition constitutes the application of one rule. For the inductive step one performs a case analysis on $\mathsf{top}_1 s$. The only case that requires some care is when we are in state q_0 with $\mathsf{top}_1 s = \varphi_i u_1 \cdots u_n{}^{\lambda}$; we do this here. If $\mathsf{top}_1 s = \varphi_i u_1 \cdots u_n{}^{\lambda}$, then by Lemma 3.4.10(iv) we must have that $s = \texttt{[}s_k, s_{k-1}, \cdots, s_1\texttt{]}$ where $s_k = \texttt{[}\varphi_i u_1 \cdots u_n{}^{\lambda}, D v_1 \cdots v_{n'}{}^{\lambda'}, s_{k1}, \cdots, s_{kN}\texttt{]}$. Now, the next transition results in (q_0, w, t) where $t = \texttt{[}s_{k+1}, s'_k, s_{k-1}, \cdots s_1\texttt{]}$, where

$$\begin{array}{rcl} s_{k+1} & = & \texttt{[}v_i{}^{\langle m-\rangle}, s_{k1}, \cdots, s_{kN}\texttt{]} \\ s'_k & = & \texttt{[}\varphi_i u_1 \cdots u_n{}^{\lambda \cup \langle m+\rangle}, D v_1 \cdots v_{n'}{}^{\lambda'}, s_{k1}, \cdots, s_{kN}\texttt{]} \end{array}$$

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

1. $s' = t$ and the case where
2. $s' \sqsubset_1 t$

For (1) we note that:

$$[\![t]\!] = \mathsf{join}((v_i)^{[\![\mathsf{pop}_1 t]\!]}, [\![\mathsf{follow}(m)t]\!])$$

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

$$\begin{array}{rcl} [\![t]\!] & = & \mathsf{join}((v_i)^{[\![\mathsf{pop}_1 t]\!]}, [\![s]\!]) \\ & = & ((v_i)^{[\![\mathsf{pop}_1 t]\!]} \cdots) : (\mathsf{tail}\ [\![s]\!]) \end{array}$$

Now, by the induction hypothesis we have:

$$\begin{array}{rcl} [\![t]\!] & = & ((v_i)^{[\![\mathsf{pop}_1 t]\!]} \cdots) : (\mathsf{tail}\ (\varphi_i u_1 \cdots u_n{}^{[\![\mathsf{pop}_1 s]\!]} \cdots) : \cdots : [\![\mathsf{pop}_1 s]\!]) \\ & = & ((v_i)^{[\![\mathsf{pop}_1 t]\!]} \cdots) : \cdots : [\![\mathsf{pop}_1 s]\!] \end{array}$$

Now, by another application of the induction hypothesis, we have must have that $[\![\mathsf{pop}_1 s]\!] = \cdots : [\![\mathsf{pop}_1(\mathsf{pop}_1 s)]\!]$. Furthermore, it is easy to check with the aid of Lemma 3.4.11 that $[\![\mathsf{pop}_1(\mathsf{pop}_1 s)]\!] = [\![\mathsf{pop}_1 t]\!]$. Hence we have:

$$[\![t]\!] = ((v_i)^{[\![\mathsf{pop}_1 t]\!]} \cdots) : \cdots : [\![\mathsf{pop}_1 t]\!]$$

As for case (2), this follows easily with the aid of Lemma 3.4.11.

□

Lemma 3.4.13. *Let* (q_j, w, s) *be a reachable configuration of* $2PDAL_W$ *for* $j > 0$ *such that* $[\![s]\!] = s_0{}^{\beta_0} s_1{}^{\beta_1} \cdots s_j{}^{\beta_j} \cdots s_n{}^{\beta_n} : \cdots : \beta_j$ *where* $s_0 \in N \cup V$. *After a finite number of transitions we will reach a configuration* (q_0, w, s') *such that* $[\![s']\!] = (s_j)^{\beta_j} : \beta_j$.

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

With the aid of Lemmas 3.4.12 and 3.4.13, we can now prove the following propositions with ease. In the following, let $W = \langle N, \Sigma, V, S, \mathcal{R} \rangle$ be a level-2 (potentially unsafe) word grammar. Furthermore, let (q_0, aw, s) be a reachable configuration over some word uaw, where $a \in \Sigma \cup \{\epsilon\}$ and $u, w \in \Sigma^*$.

Proposition 3.4.14. *Let the following*

$$(q_0, aw, s) \to (q_{i_1}, w, s_{j_1}) \to \cdots \to (q_{i_m}, w, s_{j_m}) \to (q_0, w, s')$$

be a sequence of transitions where each $i_j > 0$ for $0 \le j \le m$. Then $[\![s]\!] \xrightarrow{a} [\![s']\!]$.

Proof. Follows by a case analysis of $\mathsf{top}_1(s)$. □

Proposition 3.4.15. *If $[\![s]\!] \xrightarrow{a} \beta'$ for some β', then there exists a sequence of transitions:*

$$(q_0, aw, s) \to (q_{i_1}, w, s_{j_1}) \to \cdots \to (q_{i_m}, w, s_{j_m}) \to (q_0, w, s')$$

where each $i_j > 0$ for $0 \le j \le m$ such that $[\![s']\!] = \beta'$.

Proof. Follows by a case analysis of $\mathsf{head}\,[\![s]\!]$. □

Corollary 3.4.16. *Let W be a level-2 word grammar, the language of $2PDAL_W = L(W)$.*

3.4.2 Simulating 2PDALs by Non-deterministic 2PDAs

The incorporation of labels (as names of links) into the stack alphabet would, in general, lead to an infinite alphabet, therefore a direct translation from the definition of the $2PDAL_W$ into a 2PDA that accepts the same language seems far from direct.

However, in this section we give a careful analysis of how links are used and manipulated by the $2PDAL_W$. We will see that this behaviour is sufficiently well-behaved that it will enable us to mimic the behaviour using a non-deterministic 2PDA.

The Use of Links in $2PDAL_W$

Let s be 2-store reachable from the initial configuration of the $2PDAL_W$ given in Figure 3.2 and let a^λ be a stack item in s. From the preceding section, we know that the following are the only possibilities for the label set λ:

1. $\lambda = \langle\ \rangle$ (i.e. no labels),
2. $\lambda = \langle m- \rangle$, i.e. a single label marking the start point of a link,
3. $\lambda = \langle m+ \rangle$, i.e. a single label marking the end point of a link,
4. $\langle m-, n+ \rangle$ where $m \ne n$, i.e. a pair of labels, one marks the start point and the other an end point of a link.

The following technical lemma is fundamental to this section:

Lemma 3.4.17. *In any run of $2PDAL_W$, if (q_j, w, s) is a reachable configuration where $top_1(s) = \$t_1 \cdots t_n{}^{\langle m-\rangle \cup \lambda}$ and $j > 0$, then, for all (q_k, y, s') such that $(q_j, w, s) \to_+ (q_k, y, s')$, $top_1(s')$ is not labelled by $m-$.*

Proof. The key to this proof is recognising that pop_2 is never used, only the parameterised form follow_2 is used. There are two cases. If $j \leq n$, then the result follows from Lemma 3.4.11. However, if $j > n$, note that the next configuration after (q_j, w, s) is (q_{j-n}, w, s'') where $\mathsf{top}_1(s'') = \varphi u_1 \cdots u_{n'}{}^{\langle m+\rangle \cup \lambda'}$. By Lemma 3.4.10(i), no item in s'' can be labelled by $m-$. Furthermore, any new labels introduced will be fresh. $\square$

Let $(q^1, w^1, s^1) \to (q^2, w^2, s^2) \to \cdots$ be a computation of the $2PDAL_W$, we say that a link m is ***followed*** if and only if there exists a transition in the computation $(q^i, w^i, s^i) \to (q^{i+1}, w^{i+1}, s^{i+1})$, where $\delta(q^i, \epsilon, \mathsf{top}_1(s^i)) = \mathsf{follow}_2(m)$. Two corollaries of the above lemma are the following:

Corollary 3.4.18. *A given link m is followed* at most once.

Corollary 3.4.19. *Let (q, w, s) be a configuration of the $2PDAL_W$. If there exists an item in s such that it has label $m-$, then the link m has not yet been followed.*

Intuition

We are now in a position to describe the intuition behind the non-deterministic 2PDA that will simulate the $2PDAL_W$. The important thing to note about $2PDAL_W$ is that not all links are followed. In the example illustrated in Section 3.4.1, no link apart from 1 was followed. In fact, we may as well not have bothered to label the remaining links.

Let Γ be the base stack alphabet described in the preceding section. Our non-deterministic 2PDA will have stack alphabet $\Gamma \cup \{a^+ : a \in \Gamma\} \cup \{a^- : a \in \Gamma\} \cup \{a^{+/-} : a \in \Gamma\}$. The $+$ and $-$ take on the same role as before, i.e. the $-$ marks the start point of an instance of a link, whereas the $+$ the end point. Note however that they are *anonymous* – they are no longer prefixed with natural numbers. We will explain how to use them shortly and why we can do so.

The simulating non-deterministic 2PDA will follow the rules in Fig. 3.2 almost exactly. The difference is that each time a link is about to created (i.e. Rule (L4) of Fig. 3.2), we guess whether the link will ever be followed in the future or not: this indeed makes sense as we know that *a link will either be followed or not, and if it is followed, this occurs at most once.* We label the start and end points of the link if and only if we guess it will be followed. Furthermore, instead of a fresh label m, we simply mark the start point with a $-$ and the end point with a $+$.

A Controlled Form of Guessing

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

> *Assume that the topmost* 1*-store has at least one item labelled by* $-$*. For the leftmost (closest to the top) of these, the corresponding end point can be found in the first* 1*-store beneath it whose topmost item is marked with a* $+$*.*[3]

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

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

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

(i) For $i = 1, \cdots, k-1$, the j_ith level-1 argument of A_i is a variable $\varphi_{j_{i+1}}$. We remind the reader of the notational convention for level-1 parameters set out in Equation 3.2.

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

Example 3.4.20. For example, using the grammar in Example 3.4.4, the following is a reachable 2-store for $2PDAL_W$ on input $h_1 h_3 h_3$:

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

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

Following the idea that the simulating 2PDA follows the transitions of the $2PDAL_W$ exactly but without the benefit of named links we illustrate the our idea of a controlled form of guessing below (and formalise it later). Suppose that we are in a configuration (q_0, w, s) of the non-deterministic 2PDA where

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

where ? may either denote a $-$ or no label at all. Furthermore, suppose that φ ultimately refers to A_j (in the $2PDAL_W$). There are two possibilities:

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

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

In the first case we leave it up to the 2PDA to non-deterministically label $\varphi t_1 \cdots t_n$ (with $+$) as well as its matching partner (with $-$). However, in the second case, we *insist* that the 2PDA label $\varphi t_1 \cdots t_n$ as well as its matching partner, as given in Case (L4) of Fig. 3.2.

[3]The invariant is actually more powerful than this, but this is sufficient to ensure that the simulation works correctly.

Let us illustrate why this is necessary[4] to maintain the above invariant with an example. Suppose we have the following configuration:

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

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

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

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

Penalty for Guessing Wrongly

The cost of using non-determinism (regardless of how controlled it may be) is that we will commit ourselves to following our guesses. When we find out that we have guessed wrongly, we shall have to abort the run. There are two cases. Suppose we find ourselves in a configuration (q_j, w, s) where $\mathsf{top}_1(s) = \$x_1 \cdots x_n{}^-$ and $j \leq n$. The fact that the topmost item is labelled by $-$ means that at some point in the past, we guessed that we would follow this link. But, by Lemma 3.4.17 we have not yet followed the link nor will we ever follow it in the future. The machine has guessed wrongly, and we abort immediately. Symmetrically if we find ourselves in a configuration (q_j, w, s) where $\mathsf{top}_1(s) = \$x_1 \cdots x_n$ and $j > n$, then we also abort. Why? The absence of a $-$ label means that at some point in the past we guessed that we would *not* follow this link, but we are now about to turn against our original guess.

Definition of the Non-deterministic 2PDA, $2PDA_W$

Let W be a (possibly unsafe) level-2 word grammar. The transition rules of the non-deterministic 2PDA, $2PDA_W$, are given in Fig. 3.3. We assume the same conventions and notation as those used in Fig. 3.1 and Fig. 3.2. Note that, again, we assume that production rules of the grammar assume the format given in Equation 3.2. The function pop_2^+ performs a pop_2 and then repeats until the topmost symbol it reads is marked with a $+$. Formally, let

[4]Sufficiency will be shown in the formal proof.

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

Figure 3.3: Transition rules of the non-deterministic 2PDA, $2PDA_W$.

s range over 2-stores, we define $\mathsf{pop}_2^{+}(s) = p(\mathsf{pop}_2(s))$ where

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

Note that although we have expressed pop_2^{+} as a separate operation it can be easily achieved using a pop_2 and an auxiliary state. Observe that the rules of the simulating non-deterministic 2PDA (Fig. 3.3) are almost identical to those of the 2PDAL (Fig. 3.2). There is only one case that might be of concern, and that is when the topmost symbol is $\varphi_j t_1 \cdots t_n^{?}$. The sequence of transitions for the $2PDAL_W$ and the $2PDA_W$ look slightly different. Closer inspection, however, will reveal that they amount to the same thing, and the only difference lies in the fact that with the latter we must label a link (if necessary) by labeling its end point first and then its start point. However, with the former labeling a link can done in one atomic step.

Remark 3.4.21. In the definition of the transition rules (Fig. 3.3), in case the top_1 item of the 2-store is headed by a level-1 variable, the 2PDA has to work out whether it is situation A or B. This can be achieved by a little scratch work on the side: do a push_2, inspect the topmost 1-store for as deep as necessary, followed by a pop_2. Alternatively we could ask the oracle to tell us whether it is A or B, taking care to ensure that a wrong pronouncement will lead to an abort.

Proof of Correctness

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

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

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

Finally, we fix a possibly unsafe level-2 word-grammar W. Let (q, s) and (q', s') be reachable configurations of $2PDA_W$ and $2PDAL_W$ respectively (we ignore the input for now). Intuitively we write $s \precsim s'$ to mean that s simulates s'. Precisely $s \precsim s'$ holds if either both s and s' are the empty 2-store, or the following hold

(i) If $\mathsf{top}_1(s) = \mathsf{top}_1(s') = S$, then $\mathsf{pop}_2(s) \precsim \mathsf{pop}_2(s')$, and

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

  a. $a = a'$ and

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

  c. $\mathsf{pop}_1(s) \precsim \mathsf{pop}_1(s')$

Lemma 3.4.22. *Let* $(q_0, w, [[S]]) = (p_0, w_0, s_0) \to \cdots \to (p_n, w_n, s_n)$ *be a transition sequence of* $2PDA_W$. *Then there exists a unique (modulo renaming of labels) transition sequence* $(p_0, w_0, s'_0) \to \cdots \to (p_n, w_n, s'_n)$ *of* $2PDAL_W$ *such that for all* $i = 0, \cdots, n$:

(i) $s_i \precsim s'_i$

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

Proof. Induction on the number of transitions. □

Proposition 3.4.23. *If a string is accepted by* $2PDA_W$, *it is also accepted by* $2PDAL_W$.

Proof. This follows from Lemma 3.4.22 and the correctness of $2PDAL_W$. □

Proposition 3.4.24. *If a string is accepted by* $2PDAL_W$*, it is also accepted by* $2PDA_W$*.*

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

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

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

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

$$\begin{array}{l}
[t_{A_n}^-, \cdots] \\
[t_{A_n-1}^{+/-}, A_n^{\lambda_n}, \cdots] \\
\vdots \\
[t_{A_k}^{+/-}, A_{k+1}^{\lambda_{k+1}}, \cdots, A_{n-1}^{\lambda_{n-1}}, A_n^{\lambda_n}, \cdots] \\
\vdots \\
[t_{A_2}^{+/-}, A_3^{\lambda_3}, \cdots, A_k^-, \cdots, A_{n-1}^{\lambda^{n-1}}, A_n^{\lambda_n}, \cdots] \\
[\varphi_j t_1 \cdots t_n^{+\cup\lambda_1}, A_2^{\lambda_2}, \cdots, A_k^-, \cdots, A_{n-1}^{\lambda^{n-1}}, A_n^{\lambda_n}, \cdots] \\
\vdots
\end{array}$$

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

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

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

For case (1), the proof of Lemma 3.4.17 says that the link represented by A_k^- has not yet been followed. However, it should not be difficult to see that by not marking t_{A_n} we have eliminated the possibility of ever returning to A_k^-. In particular, this contradicts that the oracle's previous information that we would perform a pop_2^+ at A_k^- (or a duplicate) in the future. For case (2), let $I = \max\{i : A_i \text{ is not } -\text{marked}\}$. Then, according to the oracle, we will perform a pop_2^+ at $t_{A_{I+1}}$, however, the next configuration will be (q_k, w', s') for $k > 0$ and $\mathsf{top}_1(s') = t_{A_I}$. This is untenable, as by assumption $t_{A_I} = \varphi$ for some level-1 variable φ, thus we must follow a link here unconditionally or we will be stuck. Again, this contradicts the oracle's information that we would not need to follow a link at t_{A_I}. □

Combining Propositions 3.4.23 and 3.4.24 we can conclude:

Theorem 3.4.25. *There is an effective transformation of any (possibly unsafe) level-2 word grammar to a non-deterministic* $2PDA$ *that accepts the same language.* □

Some Examples

Example 3.4.26. We demonstrate the non-deterministic 2PDA for the grammar in Example 3.4.4 on input $h_1h_3h_2f_1a$. Recall that this word does not belong in the language. As before, the automaton starts in the configuration $(q_0, h_1h_3h_2f_1a, \texttt{[[}S\texttt{]]})$. After a few steps the configuration is:

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

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

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

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

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

This time suppose it chooses not to label:

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

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

Example 3.4.27. As another example using the same grammar, let us attempt a computation on the same word $h_1 h_3 h_2 f_1 a$ but this time using a different set of guesses.

$$
\begin{array}{rcl}
(q_0, & h_1 h_3 h_2 f_1 a, & \texttt{[[}S\texttt{]]}) \\
\rightarrow_+ (q_0, & h_2 f_1 a, & \texttt{[[}(D\varphi x), DGAB, S\texttt{]}, \\
 & & \texttt{[}\varphi B, D(D\varphi x)y(\varphi y), DGAB, S\texttt{]]}) \\
\rightarrow_+ (q_0, & f_1 a, & \texttt{[[}\varphi x, H(Fy)x, (D\varphi x), DGAB, S\texttt{]}, \\
 & & \texttt{[}\varphi B, D(D\varphi x)y(\varphi y), DGAB, S\texttt{]]}) \\
\rightarrow_+ (q_0, & f_1 a, & \texttt{[[}(Fy)^-, (D\varphi x), DGAB, S\texttt{]}, \\
 & & \texttt{[}\varphi x^+, H(Fy)x, (D\varphi x), DGAB, S\texttt{]}, \\
 & & \texttt{[}\varphi B, D(D\varphi x)y(\varphi y), DGAB, S\texttt{]]}) \\
\rightarrow_+ (q_0, & a, & \texttt{[[}x, (Fy)^-, (D\varphi x), DGAB, S\texttt{]}, \\
 & & \texttt{[}\varphi x^+, H(Fy)x, (D\varphi x), DGAB, S\texttt{]}, \\
 & & \texttt{[}\varphi B, D(D\varphi x)y(\varphi y), DGAB, S\texttt{]]}) \\
\rightarrow_+ (q_1, & a, & \texttt{[[}(Fy)^-, (D\varphi x), DGAB, S\texttt{]}, \\
 & & \texttt{[}\varphi x^+, H(Fy)x, (D\varphi x), DGAB, S\texttt{]}, \\
 & & \texttt{[}\varphi B, D(D\varphi x)y(\varphi y), DGAB, S\texttt{]]})
\end{array}
$$

But now the automaton aborts prematurely. The fact the the topmost symbol is labelled with a $-$ indicates that earlier the automaton guessed it would follow the link here. However, it is now in state q_1 and the first argument *is* present, hence, the automaton guessed incorrectly. There are two other combinations of guesses that can consume the prefix $h_1 h_3 h_2 f_1$; we leave it to the reader to check that neither results in acceptance of $h_1 h_3 h_2 f_1 a$.

Example 3.4.28. Finally, as a last example, we use the same grammar, but this time our input word is $h_1 h_3 h_3 b$. It can easily be checked that this is indeed a word in the language.

$$
\begin{array}{rcl}
(q_0, & h_1 h_3 h_3 b, & \texttt{[[}S\texttt{]]}) \\
\rightarrow_+ (q_0, & h_3 b, & \texttt{[[}\varphi B, D(D\varphi x)y(\varphi y), DGAB, S\texttt{]]}) \\
\rightarrow_+ (q_0, & h_3 b, & \texttt{[[}(D\varphi x)^-, DGAB, S\texttt{]}, \\
 & & \texttt{[}\varphi B^+, D(D\varphi x)y(\varphi y), DGAB, S\texttt{]]}) \\
\rightarrow_+ (q_0, & b, & \texttt{[[}\varphi B, (D\varphi x)^-, DGAB, S\texttt{]}, \\
 & & \texttt{[}\varphi B^+, D(D\varphi x)y(\varphi y), DGAB, S\texttt{]]})
\end{array}
$$

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

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

And the word is accepted as required.

3.5 Investigating Non-Determinism

The use of non-determinism (albeit controlled) is essential to our construction of the 2PDA that accepts an arbitrary level-2 word grammar. However, it is important to make a distinction between:

- Non-determinism "inherent" in the underlying grammar; intuitively, this is brought about by having a non-terminal, say F, on the left hand side of more than production rule.
- Non-determinism introduced by our construction; namely the choice of whether or not we will label.

In this section, we make precise the notion of a deterministic word grammar. Surprisingly, this does not seem to have been formalised in any literature[5]. We will then show that safe deterministic level-n word grammars are equivalent to level-n deterministic pushdown automata. This result, although not difficult, is a desirable one and, again, to the author's knowledge has not been previously recorded. Indeed, as a consequence, we have a grammatical characterisation of *deterministic* pushdown languages.

After this, we will be in a position to discuss the ramifications of the non-determinism introduced by our construction.

3.5.1 Deterministic Higher-order Grammars and Pushdown Automata

Definition 3.5.1. A word grammar is ***deterministic*** if for every $F\overrightarrow{x} \xrightarrow{a} E$ and $F\overrightarrow{x} \xrightarrow{a'} E'$, (1) if $a = a'$ then $E = E'$ (2) if $a = \epsilon$ and $E \neq e$, then $a' = \epsilon$.

Example 3.5.2.

$$\begin{array}{lcl@{\qquad\qquad}lcl}
F\overrightarrow{x} & \xrightarrow{a} & E_1 & F\overrightarrow{x} & \xrightarrow{a} & E_1 \\
F\overrightarrow{x} & \xrightarrow{b} & E_2 & F\overrightarrow{x} & \xrightarrow{b} & E_2 \\
F\overrightarrow{x} & \xrightarrow{\epsilon} & e & F\overrightarrow{x} & \xrightarrow{\epsilon} & E_4 \text{ where } E_4 \neq e \\
G\overrightarrow{x} & \xrightarrow{\epsilon} & E_3 & G\overrightarrow{x} & \xrightarrow{\epsilon} & E_3
\end{array}$$

The set of rules on the left result in a deterministic word-grammar by our definition. However, the set on the right do not.

Lemma 3.5.3. *If a grammar W is deterministic and $w \in L(W)$, then there is a* unique *derivation sequence for w.*

The obvious proof is omitted.

Definition 3.5.4. An nPDA (specialised for word languages), $\langle Q, \Sigma, \Gamma, \delta, q_0, \bot, F \rangle$ as defined in Definition 2.5.12 with acceptance by final state, is said to be ***deterministic*** if and only if the following conditions hold:

1. $|\delta(q, a, Z)| \leq 1$ for all $a \in \Sigma \cup \{\epsilon\}$, $q \in Q$ and $Z \in \Gamma$.
2. If $\delta(q, a, Z) \neq \emptyset$ for $a \in \Sigma$ then $\delta(q, \epsilon, Z) = \emptyset$.

In order to show equivalence between deterministic nPDAs and safe deterministic level-n grammars, we will need to make use of the following lemma. It is a normal form lemma. It concerns all nPDAs who possess a transition of the form $(q_f, \epsilon, Z, p, op)$, where q_f is a final state.

[5]A definition of a deterministic indexed grammar (in the sense of Aho) exists due to Parchmann et al. [PDS80a, PDS80b] via the notion of an indexed pushdown automaton. Parchmann et al. went on to show numerous properties of this resulting class of deterministic indexed languages, including closure under complementation. At higher levels, Engelfriet and Vogler [EV87] showed that LL(k) higher-order indexed grammars can be parsed by higher-order deterministic pushdown automata, but not the converse. They also succeeded in an extension of Parchmann et al.'s result by showing that the class of languages generated by level-k deterministic pushdown automata is closed under complementation.

Lemma 3.5.5. *Given any deterministic nPDA* $\mathcal{A} = \langle\, Q, \Sigma, \Gamma, \delta, q_0, \perp, F \,\rangle$*, we can convert it into an equivalent deterministic nPDA* $\mathcal{A}' = \langle Q', \Sigma, \Gamma, \delta', q_0, \perp, F\rangle$ *such that for every* $q \in F$ *we have* $\sum_{Z\in\Gamma} |\delta'(q, \epsilon, Z)| = 0$.

Proof. If $\mathcal{A}$ does not possess any transition rule of the form $(q_f, \epsilon, Z, p, op)$ for some $q_f \in F$, some Z, p, op then we are done.

However, if such a transition rule does exist, then the new automaton will be: $\mathcal{A}' = \langle Q', \Sigma, \Gamma, \delta', q_0, \perp, F\rangle$, where $Q' = Q \cup \{q^a : q \in Q \wedge a \in \Sigma\}$. Furthermore, we define δ' as

$$\begin{aligned} \delta' \quad &= \quad (\delta - \{s : s \in \delta \wedge s = (q_f, \epsilon, Z, p, op) \text{ where } q_f \in F\}) \\ &\quad\quad \cup \bigcup_{a\in\Sigma} \delta_a \\ &\quad\quad \cup\, \delta_{init} \end{aligned}$$

where, for each $a \in \Sigma$, $Z \in \Gamma$, and each $q_f \in F$ we have:

$$(q_f, \epsilon, Z, p, op) \in \delta \Leftrightarrow (q_f, a, Z, p^a, op) \in \delta_{init}$$

and furthermore, for each $a \in \Sigma$, $Z \in \Gamma$, and $q \in Q$:

$$\begin{aligned} (q, a, Z, p, op) \in \delta \quad &\Leftrightarrow \quad (q^a, \epsilon, Z, p, op) \in \delta_a \\ (q, \epsilon, Z, p, op) \in \delta \quad &\Leftrightarrow \quad (q^a, \epsilon, Z, p^a, op) \in \delta_a \end{aligned}$$

It is easy to see that we still have a deterministic nPDA on our hands. It is now routine to show by induction on the length of a transition sequence that: w is accepted by $\mathcal{A}$ if and only if w is accepted by $\mathcal{A}'$. We leave this to reader but we explain the intuition.

Suppose $(q_f, \epsilon, Z, p, op) \in \delta$ and $q_f \in F$. Also, suppose that during the computation of the deterministic nPDA we enter a configuration where we are in state q_f with topmost store symbol Z. There are two possibilities:

1. We finished reading the input, and therefore, as we are in an accepting state, the word is accepted. In this case, following the ϵ-transition is pointless.

2. We have not finished reading the input, thus, we need to follow the ϵ-transition in order to (hopefully) consume more of the input.

This is why we have replaced each rule $(q_f, \epsilon, Z, p, op)$ by a family rules:

$$\begin{gathered} (q_f, a_1, Z, p^{a_1}, op) \\ \vdots \\ (q_f, a_m, Z, p^{a_m}, op) \end{gathered}$$

where $\Sigma = \{a_1, \cdots, a_m\}$. In particular, for case (1) we have finished reading the word on an accepting state, so provided q_f remains an accepting state, we still accept the input. However, if we are in case (2), then we are indeed expecting more input. Thus, say the next letter of the unconsumed input is a_i, then we follow the rule $(q_f, a_i, Z, p^{a_i}, op)$ and consume a_i. However, we consume in a_i in "good faith." In other words, we do this in the hope that had we followed the original δ, then, after some finite number of ϵ-transitions we would indeed have consumed a_i. To this end we create a new set of states $Q^{a_i} = \{q^{a_i} : q \in Q\}$ which may be thought of as a holding pen where we can move only via ϵ-transitions, and are only released from it once we find ourselves in a configuration (q^{a_i}, w, s) with $\mathsf{top}_1(s) = Y$ for some Y such that $\delta(q, a_i, Y) \neq \emptyset$. This fulfils the "good faith" agreement, and we leave Q^{a_i} via an ϵ-transition[6]. □

3.5.2 Equivalence between Deterministic nPDAs and Safe Deterministic Level-n Word Grammars

Proposition 3.5.6. *If L is generated by a safe deterministic level-n word grammar then it is accepted by a deterministic nPDA.*

Proof. We use the construction of Section 5.2 in [KNU02] that, given a safe higher-order grammar, transforms it into a pushdown automaton of the same level. The adapted rules (for word languages) were given in Fig. 3.1.

If W is a deterministic word-grammar then there is only one possible source of non-determinism in the transition function of the corresponding level-n pushdown automaton. This scenario arises as we allow both ϵ- and non-ϵ- transitions from a single non-terminal. In particular, we might have that:

- $F\overrightarrow{x} \xrightarrow{\epsilon} e$, and
- there exist $a_1, \cdots, a_n \in \Sigma$ for $n \geq 1$, such that $F\overrightarrow{x} \xrightarrow{a_i} E_i$.

This would mean the following transition rules are present in the corredponding nPDA (for any suitable $u_1 \cdots u_d$):

$$\begin{array}{rcl}
(q_0, \epsilon, Fu_1 \cdots u_d) & \rightarrow & (q_0, \mathsf{push}_1(e)) \\
(q_0, a_1, Fu_1 \cdots u_d) & \rightarrow & (q_0, \mathsf{push}_1(E_1)) \\
& \vdots & \\
(q_0, a_n, Fu_1 \cdots u_d) & \rightarrow & (q_0, \mathsf{push}_1(E_n))
\end{array}$$

However, it is straightforward to determinise this nPDA. We introduce a new state q_F, for each non-terminal F where the above phenomenon occurs (i.e. there are both ϵ-transitions and non-ϵ-transitions). We make each such q_F into an accepting state and replace the above rules by the following:

$$\begin{array}{rcl}
(q_0, \epsilon, Fu_1 \cdots u_d) & \rightarrow & (q_F, \mathsf{noop}) \\
(q_F, a_1, Fu_1 \cdots u_d) & \rightarrow & (q_0, \mathsf{push}_1(E_1)) \\
& \vdots & \\
(q_F, a_n, Fu_1 \cdots u_d) & \rightarrow & (q_0, \mathsf{push}_1(E_n))
\end{array}$$

This is clearly deterministic and accepts the same languages. □

[6]Note that obviously we must not consume a_i again (as we have done so in the past).

Proposition 3.5.7. *If L is the language of a deterministic nPDA, then it is generated by a safe deterministic level-n word grammar.*

Proof. This proof requires very little work on our behalf. Given an nPDA, $\mathcal{A}$, we apply Lemma 3.5.5 to the nPDA to generate a new *nPDA*, $\mathcal{A}'$. This ensures that if $(q, \epsilon, Z, p, op) \in \delta$ then q is not an accepting state.

We then invoke Knapik et al.'s Theorem 5.1 [KNU02], which, given an nPDA transforms it into an equivalent safe grammar. We replicate the details below (but not the actual proof, which may be found in [KNU02]). We assume that our automaton $\mathcal{A}' = \langle Q, \Sigma, \Gamma, \delta, q_0, \bot, F \rangle$ has m states: $q_0, \cdots, q_{m-1}$. We use the following abbreviations $\mathbf{1} = 0^m \to 0$, $\mathbf{2} = \mathbf{1}^m \to \mathbf{1}$ and so on up to $\mathbf{n} = (\mathbf{n-1})^m \to (\mathbf{n-1})$.

We introduce a non-terminal F_q^Z for each pair (q, Z) where q is a state of the nPDA and Z a store symbol. This non-terminal will be of type $\mathbf{n}$. For each $k = 1, \cdots, n$ there is a non-terminal $Void_k : \mathbf{n} - \mathbf{k}$. And finally, there is an initial nonterminal $S : 0$.

The initial production of the grammar is:

$$S \xrightarrow{\epsilon} F_{q_0}^{\bot} \overrightarrow{Void}_1 \cdots \overrightarrow{Void}_n$$

And, here are the remaining productions (where each vector $\overrightarrow{x}_k = x_k^0 \cdots x_k^{m-1}$):

1. $F_q^Z \overrightarrow{x}_1 \cdots \overrightarrow{x}_n \xrightarrow{a} F_p^Z \overrightarrow{x}_1 \cdots \overrightarrow{x}_{k-1} (F_{q_0}^Z \overrightarrow{x}_1 \cdots \overrightarrow{x}_k) \cdots (F_{q_{m-1}}^Z \overrightarrow{x}_1 \cdots \overrightarrow{x}_k) \overrightarrow{x}_{k+1} \cdots \overrightarrow{x}_n$, if $\delta(q, a, Z) = (p, \mathsf{push}_k)$ for $k > 1$.

2. $F_q^Z \overrightarrow{x}_1 \cdots \overrightarrow{x}_n \xrightarrow{a} F_p^Y (F_{q_0}^Z \overrightarrow{x}_1) \cdots (F_{q_{m-1}}^Z \overrightarrow{x}_1) \overrightarrow{x}_2 \cdots \overrightarrow{x}_n$, if $\delta(q, a, Z) = (p, \mathsf{push}_1(Y))$.

3. $F_q^Z \overrightarrow{x}_1 \cdots \overrightarrow{x}_n \xrightarrow{a} x_k^j \overrightarrow{x}_{k+1} \cdots \overrightarrow{x}_n$ if $\delta(q, a, Z) = (q_j, \mathsf{pop}_k)$.

4. Finally, for each F_q^Z such that q is an accepting state we add the rule $F_q^Z \overrightarrow{x}_1 \cdots \overrightarrow{x}_n \xrightarrow{\epsilon} e$.

Item (4) shows the significance of Lemma 3.5.5. If we had not applied Lemma 3.5.5, we would allow for an nPDA which may have a transition rule (q, ϵ, Z, p, op), where q is an accepting state and in this case F_q^Z would have two righthand sides labelled by an ϵ.

□

Combining Propositions 3.5.6 and 3.5.7, we have:

Corollary 3.5.8. *A language L is accepted by a deterministic nPDA if and only if L is the language of a deterministic safe level-n grammar.*

Furthermore, given the equivalence between our definition of a level-n grammar and that of Damm's, we have a grammatical characterisation of a deterministic OI-language.

3.5.3 Unsafe Deterministic Grammars

As mentioned at the beginning of this section, our construction introduces a layer of non-determinism by way of allowing the level-2 pushdown automaton to choose whether or not to label a link when we are about to expand a level-1 variable. This non-deterministic choice of whether to label a link can be safely removed in the event that the underlying grammar is safe.

Lemma 3.5.9. *Let W be a* safe *word grammar and let $2PDAL_W$ be the corresponding level-2 pushdown automaton with links. If (q_i, w, s) is a reachable configuration on a word $z = vw$, with $i > 0$ and $\mathsf{top}_1(s) = \$t_1 \cdots t_k^{\lambda}$, where $m- \in \lambda$ for some m, then it will always be the case that $i > k$. Furthermore, $\mathsf{top}_1(\mathsf{pop}_1 s)$ has label $m+$.*

Proof. Induction on the number of transitions of the $2PDAL_W$. Clearly this is vacuously true for the initial configuration $(q_i, z, \texttt{[[}S\texttt{]]})$. To show this for the inductive step, we perform a case analysis based on the next rule (from Figure 3.2); there are only two cases that demand some attention:

1. The first case is given by the following rule:

$$(q_0, \epsilon, x_j) \to (q_j, \mathsf{pop}_1)$$

The next configuration after application of this rule will result in state q_j reading topmost symbol $Dt_1 \cdots t_n^{\lambda}$. If $\lambda = \emptyset$, then there is nothing to verify. However, if $m- \in \lambda$, then we must verify that (1) $j > n$ and (2) that the end point of m can be found in the 1-store directly below. From item (iv) in Lemma 3.4.10, the presence of $m-$ implies that $Dt_1 \cdots t_n$ must have level > 0. However, $x_j : o$. It cannot be the case that $j \leq n$, as this would indicate that $t_j : o$ and would contradict the fact that D is homogeneously typed. Now we consider the end point of m: suppose the hypothesis is false; therefore we must have that $\mathsf{top}_1(\mathsf{pop}_1 s)$ has label $l+$ for some $l \neq m$. By Corollary 4.18, this indicates that the link l has not yet been followed. However, it cannot be the case that $l-$ exists in the same 1-store as $\mathsf{top}_2(s)$, due to Lemma 4.16. Therefore, it must be the case that at some point earlier in the computation we must have been in state (q_i, w, s') where $i > 0$ and such that $\mathsf{top}_1(s) = s_1 \cdots s_k$ and $i \leq k$. This, however is a contradiction of the induction hypothesis.

2. The other case is given by:

$$(q_i, \epsilon, \$t_1 \cdots t_r^{\lambda}) \to (q_{i-r}, \mathsf{follow}_2(m)) \text{ where } m- \in \lambda$$

Thus resulting in a configuration in state q_{i-r} with topmost symbol $\varphi s_1 \cdots s_q^{\lambda'}$; we must verify whether the hypothesis holds in the case that $k- \in \lambda'$ for some k. The argument proceeds exactly as the second half of the argument above.

□

Corollary 3.5.10. *Bearing the above in mind, we can adjust the definition of the simulating $2PDA$ so that we* always *label a link when the underlying word grammar is safe – there is no more choice in the matter.*

Remark 3.5.11.
1. When W is a safe grammar, the effect of always labeling a link means that the two abortive computations associated with being in state q_j for $j > 0$ never arise.

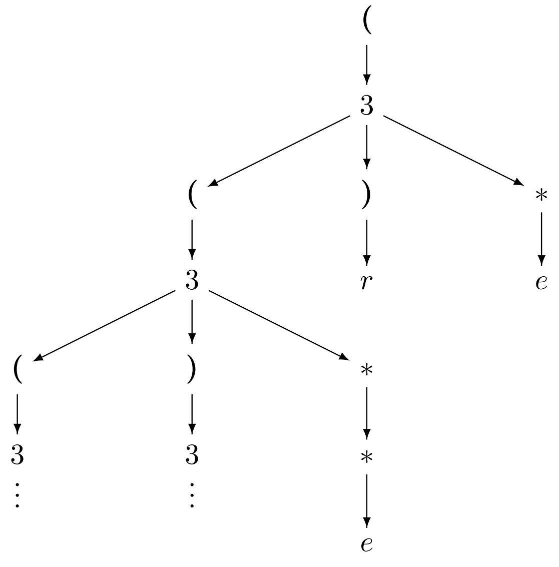
2. Note that the above proof showing that we can remove the non-deterministic choice of whether or not to label in the case of safe grammars is nothing new. Indeed, the same could have deduced by noticing that the definition of the 2PDA (with the clause to always label) and the original definition of the 2PDA constructed by Knapik et

al. coincide. However, we have included our exposition to illustrate how links aid in understanding why this is the case.

This section has shown that safe deterministic word grammars correspond to deterministic pushdown automata. However, this of course begs the question of what is the relationship between safe deterministic and unsafe deterministic word grammars at level 2 (and beyond). Unfortunately, this is not a question we are able to answer.

However, we do show that if there is a language L such that L can be accepted by an unsafe level-2 deterministic grammar but not by a level-2 safe deterministic one, then there exists a term tree T such that T can be generated by a level-2 unsafe recursion scheme, but not by any level-2 safe one[7].

In fact, we believe that U is such a language. Below we give a proof that if U cannot be accepted by any level-2 safe deterministic grammar then we will have an inherently unsafe level-2 term tree.



$$\begin{tikzcd}
 & & (\arrow[d] & \\
 & & 3 \arrow[dl] \arrow[d] \arrow[dr] & \\
 & (\arrow[d] &) \arrow[d] & * \arrow[d] \\
 & 3 \arrow[dl] \arrow[d] \arrow[dr] & r & e \\
(\arrow[d] &) \arrow[d] & * \arrow[d] & \\
3 & 3 & * \arrow[d] & \\
\vdots & \vdots & e &
\end{tikzcd}$$

Figure 3.4: The term tree $[\![G_U]\!]$.

Let us convert the unsafe word grammar for U defined previously into the following recursion scheme, G_U. Our signature is $\Sigma = \{ (: (o,o),\ ) : (o,o),\ * : (o,o),\ 3 : (o,o,o,o),\ e : o,\ r : o \}$ and we have the following production rules[8]:

[7]Intuitively, this should make sense as a recursion scheme is deterministic in the sense that each non-terminal occurs on the left hand side of precisely one rule.

[8]The terminals have been circled to avoid confusion with semantic brackets.

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

The term tree generated by G_U is shown in Figure 3.4. Note that all we have done is the following: for any non-terminal, such as D, that occurs on more than one left hand side of a production rule (with right hand sides $r_1, r_2 \cdots, r_n$), we simply amalgamate these into one of the form $n r_1 r_2 \cdots r_n$ where $n : o^n \to o$; and the n represents n-way non-deterministic choice.

Proposition 3.5.12. *Suppose that $[\![G_U]\!]$ can be generated by a safe level-2 recursion scheme. Then it must be the case that the language U can be accepted by a deterministic 2PDA.*

Proof. By [KNU02], if $[\![G_U]\!]$ is generated by a safe level-2 recursion scheme then there exists a level-2 pushdown tree automaton (as given in Definition 2.5.1) $A = \langle Q, \Sigma, \Gamma, \delta, q_0, \bot \rangle$ that accepts $[\![G_U]\!]$.

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

- For each $(q, Z, (p, op)) \in \delta$ we add the rule (q, ϵ, Z, p, op) to δ_s.
- If $f \in \Sigma^{(o,o)}$, then for each $(q, Z, f(p_1)) \in \delta$ we add the rule $(q, f, Z, p_1, \mathsf{id})$ to δ_s.
- For every $(q, Z, 3(p_1, p_2, p_3)) \in \delta$, we add the following the family of rules to δ_s:

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

  where, for each $a \in \{(,), *\}$, $(p^a, \epsilon, Z, q^a, op) \in \delta_s$ if and only if $(p, Z, (q, op)) \in \delta$ and $(p^a, \epsilon, Z, q, \mathsf{id}) \in \delta_s$ if and only if $(p, Z, a(q)) \in \delta$.

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

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

Conjecture 3.5.13. *U cannot be accepted by a deterministic 2PDA, and hence, there exists a term-tree generated by an unsafe level-2 grammar that cannot be generated by any safe level-2 grammar.*

3.6 Related Work and Concluding Remarks

3.6.1 Related Work

There are few results to date that can be said to be *directly* related to the safety "problem" as we have pursued it here: namely whether the safety constraint restricts the expressive power of higher-order grammars. This is not entirely surprising given that the constraint of safety was only recently introduced by Knapik et al. in 2001 [KNU01] and we believe the main result of this chapter is the first published result to investigate the relationship between safe and unsafe grammars.

Consequently, we only list one item here that can claim to be directly related to our investigation of "unsafety" in terms of expressibility. We will consider the panic automaton as introduced by Knapik et al. [KNUW05] and compare it with our own level-2 pushdown automaton with links.

In [KNUW05], Knapik, Niwiński, Urzyczyn and Walukiewicz introduce the *panic automaton* as a machine characterisation of level-2 recursion *schemes* (potentially unsafe). Intuitively, a panic automaton is a level-2 pushdown automaton (defined as an acceptor of a single term tree) with an additional destructive operation known as panic. In addition to this extra operation, all store items are given a time stamp, which serves as a record of when the item was first pushed onto to the store with a push_1. For example, suppose that the current global configuration consists of 5 level-1 stores : $[s_5, s_4, \cdots, s_1]$[9], where s_5 is the topmost 1-store. In a standard level-2 pushdown automaton the action $\mathsf{push}_1(a)$ would push a onto the top of s_5. However, with a panic automaton the *pair* $(a, 4)$ would be pushed onto the store. The second component of the pair records the fact that when this occurrence of a was first pushed onto the store via a push_1 there were precisely 4 level-1 stores below it. Subsequently the symbol can be duplicated many times by executing a push_2, however, the time stamp keeps a record of when it first appeared on the stack.

Now we turn to the panic operation. Suppose this is applied when the topmost item of the store is (a, m) where m is a natural number. Then the effect of applying a panic operation deletes all the existing level-1 stores until only the bottom m remain.

Given that Knapik et al.'s aim was to characterise potentially unsafe higher-order recursion schemes via a machine that was based on pushdown automaton, it is not too surprising that our 2PDAL is very similar to the panic automaton. In fact we show that they are equivalent.

First let us remark that there is a slight mismatch between what the 2PDAL and the panic automaton define. The 2PDAL, as defined earlier, is used to accept word languages, whereas the panic automaton accepts a single term tree. This is of little consequence, however, and

[9]The reader familiar with [KNUW05] will note that our notation differs from theirs in that our store grows from right to left, whereas in [KNUW05] it grows from left to right. We modify their definitions to convene with our right-to-left notation so that it is in keeping with the rest of this thesis.

we can compare them simply by considering the *set of operations* that each can perform on a 2-store.

Panic Automata

As indicated above, a panic automaton makes use of a 2-store $[s_l, s_{l-1}, \cdots, s_1]$, where $l \geq 1$, and each s_i is a 1-store. As before, assume that s_l is the topmost 1-store. Each s_i is a list of store items : $[b_k, b_{k-1}, \cdots, b_1]$ where $b_i \in \Gamma \times \omega$, where Γ is the store alphabet and ω is the set of natural numbers. Formally, the set of operations, over level-2 stores for panic automata is defined below. In the following we assume that $s_l = [(a_n, m_n), (a_{n-1}, m_{n-1}), \cdots, (a_1, m_1)]$

$$\begin{aligned}
\mathsf{push}_1 a\,[s_l, s_{l-1}, \cdots, s_1] &= [[(a, l-1), (a_n, m_n), (a_{n-1}, m_{n-1}), \cdots, (a_1, m_1)], s_{l-1}, \cdots, s_1] \\
\mathsf{pop}_1\,[s_l, s_{l-1}, \cdots, s_1] &= [[(a_{n-1}, m_{n-1}), \cdots, (a_1, m_1)], s_{l-1}, \cdots, s_1] \\
\mathsf{push}_2\,[s_l, s_{l-1}, \cdots, s_1] &= [s_l, s_l, s_{l-1}, \cdots, s_1] \\
\mathsf{pop}_2\,[s_l, s_{l-1}, \cdots, s_1] &= [s_{l-1}, \cdots, s_1] \\
\mathsf{panic}\,[s_l, s_{l-1}, \cdots, s_1] &= [s_k, s_{k-1}, \cdots, s_1] \text{ if } k = m_n
\end{aligned}$$

Definition 3.6.1. A ***panic automaton*** [KNUW05] is a tuple $\langle Q, \Sigma, \Gamma, \delta, q_0, \perp \rangle$ where Σ is a finite signature, Q a finite set of states with initial state q_0, Γ a store alphabet with $\perp \in \Gamma$ as the bottom-of-store symbol, and finally, δ is the transition *function.* δ is defined so that for every $q \in Q$ and $a \in \Gamma$ we have that either (i) $\delta(q, a) = (p, op)$ for some $p \in Q$ and op a stack operation defined above or (2) $\delta(q, a) = f(q_1, \cdots, q_n)$ for $f \in \Sigma$ and f has rank n. The definitions of acceptance, configurations and so forth are analogous to those of the level-2 pushdown tree automaton (see Chapter 2).

Theorem 3.6.2. *([KNUW05]) A term tree accepted by a panic automaton must be accepted by a level-2 recursion scheme.*

Theorem 3.6.3. *([KNUW05]) Every term tree generated by a level-2 recursion scheme is accepted by a panic automaton.*

Proofs of both of the above theorems are given in the citation.

Comparison with the 2PDAL

Intuitively, it should not be difficult to see why the panic automaton and the 2PDAL are equivalent, not only in the strict sense but also in "essence." In the former, links are "built-in" by way of the timestamps that are recorded for each push_1 action. In the latter, only when we choose to use the command $\mathsf{pushlink}_2(a)$ versus the $\mathsf{push}_1(a)$ action, will such a time stamp be recorded. In both cases, the time stamps capture the same information: they create a connection between the most recently created item and the 1-store beneath it. Thus, the 2PDAL can be thought of a "lazy" version of the panic automaton, given that it only records such information when told to do so. As for the operations $\mathsf{follow}_2(m)$ and panic, the correspondence is obvious. In both cases, if the topmost item is a, then performing the above operations returns us to the topmost item of the 1-store that was directly underneath the topmost 1-store at the time of a's creation[10]. A formal proof of the correspondence is straightforward but will not be pursued here.

[10]Of course, for the 2PDAL this requires that the link was labelled at the time of creation.

Remark 3.6.4. Whereas we have used the 2PDAL as an intermediary step to show that every level-2 word-grammar can be accepted by a non-deterministic 2PDA, Knapik et al. used their panic automaton for a different purpose: to prove a decidability result for term trees generated by unsafe level-2 recursion schemes. We return to this in the next chapter.

Remark 3.6.5. In the author's opinion, the panic automaton is a much better presented machine compared to the 2PDAL. However, for the purposes of our result, i.e. showing that a level-2 unsafe word grammar can be accepted by a level-2 pushdown automaton, the link concept was indeed a fruitful one. It made explicit the relationship between items *of significance to us* rather than every item pushed onto the stack.

3.6.2 Concluding Remarks and Future Work

We conclude this chapter by summarising our contributions and detailing possibilities for future work.

Ultimately, we sought to understand whether the class of structures generated by safe level-n grammars is smaller than the class of structures generated by unrestricted level-n grammars (in other words, those that are potentially unsafe). At their most general, higher-order grammars as we have introduced them in Chapter 2 generate a set of (finite) term trees over a signature Σ. In this chapter, we focused on the restricted setting where the underlying signature Σ is monadic and therefore the higher-order grammar generates a word language in the classical sense. All of the results of this chapter can thus be referred to as language-theoretic results. We believe this chapter has 4 original contributions to offer. The first and most important is the proof that at level-2 every unsafe word grammar is equivalent to a safe level-2 word grammar. The second contribution is the development of the level-2 pushdown automaton with links that was used as an intermediate step in the above proof. Although we believe the level-2 pushdown automaton served its purpose well, we do believe that the recently introduced panic automaton [KNUW05] (which is, in fact, equivalent to the 2PDAL) is a more elegant model for capturing level-2 grammars. Our third contribution is the definition of deterministic word grammars and the proof of the equivalence between deterministic safe level-n word grammars and deterministic level-n pushdown automata; thus rendering a grammatical characterisation of deterministic pushdown languages. This also enabled us to distinguish between the non-determinism integral to our construction and that of the underlying word grammar. This also led us to ask the question of whether we can establish the relationship between safe deterministic and unsafe deterministic word grammars. Finally, our fourth contribution is the alternative decomposition of Urzyczyn's language – whilst this result is superseded by the main result of this chapter, we believe that the status of Urzycyzn's language make this an interesting side note.

Obvious Extensions

It is evident that there is much future work to be done in analysing the expressive power of safe versus unsafe grammars. The most obvious direction is an extension of our main result to all levels of the hierarchy of word grammars. The generalisation of the PDAL (or equivalently, the panic automaton) to any level n is actually very straightforward[11]. Correspondingly, given a level-n word grammar W the generalisation of our construction to that of a level-n PDAL to accept the language of W is also a straightforward exercise. However, the second half of

[11]It simply has not been included in this thesis due to the fact that we are unable to use it.

the proof is more problematic. Although seemingly innocuous, the fact that in the definition of $2PDAL_W$, a link is followed at most once is crucial to the proof. This result is not true of level 3 word-grammar. In fact, it is easy to construct an example of a level-3 word-grammar where not only can a link be followed more than once, but it may be the case that sometimes it is followed and sometimes it is not. Thus transforming this into a equivalent 3PDA using ideas from our level-2 proof is not entirely straightforward – if at all possible.

The other obvious direction to consider is an extension of our result to higher-order grammars in general (i.e. removing our restriction of monadic signatures); perhaps starting with level-2 grammars. A direct application of our result to a level-2 (term tree) grammar failed. Once again, it was straightforward to extend the notion of the 2PDAL over words to a 2PDAL over term trees; in fact the panic automaton is evidence for this. The difficulty, however, again lies with the 2nd part of the proof: simulating links with a 2PDA (over term trees). Specifically, the problem stems from the following: given a signature symbol $f \in \Sigma$ with, say, n children, then, although each *path* from f obeys our analysis of how links behave; considering the paths *together*, it fails. This is because one set of oracle decisions for one path may not be suited to another, and there is no obvious way to reconcile this in the labeling of an item. Another possible line of attack, of course, is an attempt to prove our conjecture concerning Urzyczyn's language, which brings us to the next set of possible avenues for future work.

The OI (Word Language) Hierarchy

Very little is known about the OI (word language) hierarchy and its properties, and even less so regarding level-n unrestricted word grammars. For example, normal-form lemmas, pumping lemmas[12], or decision procedures for whether two grammars or automata are equivalent are all in short supply. Another important line of work is to establish the status of the OI (word language) hierarchy with respect to the Chomsky hierarchy. Whilst the first two levels of the OI and Chomsky hierarchy coincide, it is not known, for example, whether the OI hierarchy is contained within the context-sensitive languages. Though it is easy to see that they are not equal (for the simple reason that the safe level-n grammars are closed under ϵ-homomorphism – [HMU01] for an excellent introduction to formal language theory) little else is known, and certainly less so regarding unrestricted grammars.

Recently it was shown by Sénizergues [Ś97] that it is decidable whether two level-1 deterministic pushdown automaton produce the same languages. An alternative (and shorter) proof is given by Stirling [Sti01a]. Note that this, of course, translates into a decision procedure for the equivalence between two level-1 safe deterministic grammars. In fact, this translation is made explicit in the article [Sti00]. In fact, it is interesting to note that the problem of deciding equivalence between deterministic pushdown automata (of level-1) was shown to be inter-reducible to the problem of deciding equivalence of deterministic recursive

[12]For example, Hayashi [Hay83] gave an extension of the *uvwxy*-theorem for context-free languages to the indexed languages in 1973 (a slimmer and more employable version is available by Gilman [Gil96]) and, to the author's knowledge, little progress – if any – has been made since then. Even more interesting would be pumping lemmas or tools to show a language cannot be accepted by a *deterministic* n-grammar or nPDA, although success in this area has been limited even for the context-free languages. One result which certainly deserves mention is Blumensath's [Blu04] recent efforts to develop a pumping lemma for higher-order pushdown automaton. He attempts to derive bounds on the lengths of paths in the configuration graph of a pushdown automaton in a bid to aid the classification of the graphs of higher-order pushdown automata. Although his analysis does not yield such bounds it does provide methods for analysing runs of a pushdown automaton and suggests how these may be intercalated.

applicative programs (level-1 grammars) long ago by Courcelle [Cou83, Cou76]. In fact, a direct proof (showing how to construct a DPDA from a recursive applicative program scheme and vice versa) was first given by Gallier [Gal81]. Such transformations have, of course, been superseded by Damm and Goerdt and Knapik et al. Clearly, an extension of the equivalence result for DPDAs to all deterministic pushdown automata of level-n for arbitrary n would be very exciting.

The IO hierarchy

Last, but not least should be an investigation of the IO hierarchy. Surprisingly little is known about the IO hierarchy; it is not even known whether a level-n word grammar using an IO mode of derivation can be captured by a level-n pushdown automaton or not.

Chapter 4

Decidability

In the previous chapter we examined whether safety restricts the expressive power of higher-order grammars. In this chapter we examine the effects of the safety restriction on decidability. In [KNU02], Knapik et al. proved the seminal result that any level-n *safe* recursion scheme defined an infinite term tree with a decidable monadic second-order theory. However, this left open the question of whether or not the term trees defined by *unsafe* recursion schemes also enjoyed an MSO decidable theory. This is the starting point for this chapter.

We will show that every level-2 recursion scheme (whether safe or not) produces a term tree with a decidable MSO theory. This is the sole result of this chapter and it was first published alongside Klaus Aehlig and Luke Ong (TLCA 2005 [AdMOa]). Given a level-2 recursion scheme R and an MSO sentence φ, we will construct an alternating parity tree automaton such that the set of trees accepted by the automaton is the empty set if and only if R does not satisfy φ. The proof relies on defining the notion of a *computation tree* for a given recursion scheme R. Intuitively, the computation tree can be thought of as an infinite λ-term that ultimately β-reduces to the term tree defined by the grammar – this is not unlike the use of λ-trees in [KNU02]. Most importantly, however, the computation tree can be defined by a level-0 grammar and is therefore a regular tree. Next we construct an alternating parity tree automaton that traverses the computation tree in such a way that (1) it effectively extracts the underlying term tree and (2) whilst doing so checks against the MSO formula.

The result presented here is very long and is also technically very involved. Ultimately, this chapter fills in the technical details behind the sketch proof for the published version in [AdMOa][1]. Where possible we will try to include as many examples and intuitive sketches before giving a formal proof.

Although the result presented here is only valid for the first level of the hierarchy where safety manifests itself, it is an exciting step forward. And indeed, this can be evidenced by the simultaneous publication of an independent proof of this level-2 result by Knapik, Niwiński, Urzyczyn and Walukiewicz in ICALP 2005 [KNUW05].

Recent developments have come to light since the author started writing up this thesis. A few months prior to the completion of this thesis Luke Ong extended the result of this chapter to the entire hierarchy, thus showing that the term tree produced by any level-n recursion

[1]Strictly, the proof methodology presented here has evolved somewhat from the original one. The intuition and key ideas remain very much the same, but after numerous discussions with Luke Ong the proof has evolved to take the format presented here.

scheme (whether safe or not) possesses a decidable monadic second-order theory [Ong06c]. This and other recent developments will be discussed towards the end of this chapter.

This chapter has only two parts. First, we present our main result. The proof will require notions and results from game semantics, logic, and tree automata theory and relationships between them. To give a satisfactory introduction to each of these fields would almost certainly double the size of this thesis. Therefore, we will, unfortunately, very often have to state these theorems without proof and introduce them on an "as-needed" basis pointing the reader to the correct reference.

In the second part, we conclude with a discussion of related results such as the independent proof of our level-2 result by Knapik et al. and also Luke Ong's extension.

4.1 MSO Decidability of Level-2 Term Trees

4.1.1 Preliminaries

Notation and Terminology for Trees

Let $t : T \longrightarrow \Sigma$ be a labelled tee (not necessarily a term tree) as defined in Chapter 2, where T is a prefix-closed subset of the monoid generated by a set X. A ***path*** in t from a node $\alpha \in T$, is a sequence of nodes $\alpha_1, \alpha_2, \cdots, \alpha_k$ in T, such that $\alpha_1 = \alpha$ and $\alpha_{i+1} = \alpha_i p_i$ where $p_i \in X$. Unless we offer any qualification, a path π in t is assumed to mean a path from the root, ϵ. Let α, β be nodes in T, if $\beta = \alpha\gamma$ for some $\gamma \in X^*$, when convenient we will write $\alpha \longrightarrow \beta$ to denote the path from α to β (inclusive of both end points). A path is said to be ***maximal*** if and only if it ends in a leaf node or is infinite.

Term Tree Automata

In the following we assume that A is a ranked alphabet with rank function ϱ and direction function Dir. We denote by A^k the set of symbols in A that have rank k. We introduce two varieties of term-tree automata over the signature A, i.e. automata whose inputs are term trees in $T^\infty(A)$: these are the non-deterministic term tree automaton and the alternating term tree automaton. Both are well-known models of computation – see [Tho97, GTW02] and references therein. Both are introduced here with the parity acceptance condition.

A ***non-deterministic term tree automaton*** is given by a tuple $\langle Q, A, \delta, q_0, Acc \rangle$ where Q is the finite set of states, A is a ranked alphabet, $q_0 \in Q$ is the start state and $\delta \subseteq \bigcup_k Q \times A^k \times \underbrace{Q \times Q \times \cdots \times Q}_{k \text{ times}}$. The acceptance condition, Acc, will be explained shortly.

Suppose, for example, that $f \in A$ and f has rank 3. The transition function δ may contain an entry such as the one below:

$$\delta(q, f) = \{(q_0, q_2, q_2), (q_1, q_1, q_0)\}.$$

This means that if we are in state q reading a node labelled f, then we can either:

1. Visit the first child in state q_0, the second child in state q_2 and the third child in state q_2 also (this corresponds to the first tuple); or:

2. We can visit the first child in state q_1, the second child in state q_1, and the third child in state q_0 (this corresponds to the second tuple).

A non-deterministic term tree automaton always travels downwards, starting at the root node in state q_0 and at each node it can only send 1 automaton to each child, as illustrated above.

The acceptance condition, Acc is given by $Parity \cup Fin$, where $Parity$ is a parity condition [EJ91] given by the colouring function $\Omega : Q \to \omega$. On the other hand, $Fin \subseteq Q \times A^0$. Intuitively, the parity condition, $Parity$, applies to infinite paths in the tree, and Fin to finite paths.

Definition 4.1.1. Given an A-tree $t : T \longrightarrow A$, we define the ***run tree*** of the non-deterministic term tree automaton N (using the notation from above) over t as the partial function $r : T \longrightarrow Q$ defined inductively as follows:

1. $r(\epsilon) = q_0$.
2. Let $\alpha \in T$ where $r(\alpha) = q$ and $t(\alpha) = f$. Let $Dir(f) = x_1 < x_2 < \cdots < x_n$, if $r(\alpha x_i)$ is defined and equal to q_i for each $1 \leq i \leq n$ then $(q_1, q_2, \cdots, q_n) \in \delta(q, f)$.

Definition 4.1.2. Let r be a run tree over $t : T \longrightarrow A$ of the automaton N. We say that a run tree is ***accepting*** if and only if r is a *total* function and the following hold:

1. If $\pi = \pi_1\pi_2\cdots$ is an infinite path in r then $\max\{\Omega(q) : q \in \mathsf{Inf}(\pi)\}$ is even, where $\mathsf{Inf}(\pi)$ consists of the states that occur infinitely often in π.
2. If π is a finite path ending with terminal node α, then $(r(\alpha), t(\alpha)) \in Fin$.

We say that t is ***accepted*** by the non-deterministic term tree automaton A if and only if there is an accepting run tree over t.

With the non-deterministic variety of term tree automaton, a copy of the automaton sitting on a node $\alpha \in t$, where α has k children spawns *precisely* k children; one in each direction as directed by δ. We now introduce the alternating parity term-tree automaton as a generalisation of this (first introduced in [MS87]). We can send, possibly 10 copies of ourself to one child, 0 to another child, and 2 to another, and so forth. In fact, we introduce the 2-way variety [KV00, Var98], so the automaton may also travel upwards (instead of always travelling in a top-to-bottom fashion).

A (2-way) ***alternating term tree automaton*** is given by a tuple $\langle Q, A, \delta, q_0, Acc \rangle$ where Q, q_0, A and Acc are as before, however, $\delta \subseteq \bigcup_{f\in\Sigma} Q \times \{f\} \longrightarrow \mathcal{B}(Q \times (Dir(f) \cup \{0, -1\}))$, where $\mathcal{B}(P)$, is the set of positive formulae over a set P. In particular:

$$\mathcal{B}(P) = \mathsf{tt} \quad | \quad \mathsf{ff} \quad | \quad p \quad | \quad \mathcal{B}(P) \wedge \mathcal{B}(P) \quad | \quad \mathcal{B}(P) \vee \mathcal{B}(P)$$

where $p \in P$. Here $\{0, -1\}$ are directions *in addition* to and distinct from those in $Dir(f)$. In particular, direction 0 indicates to remain at the current node and direction -1 means to visit the parent node.

Let us assume that $f \in \Sigma$ has rank 3 and $Dir(f) = [3]$. Suppose

$$\delta(q, f) = (q_1, 1) \vee ((q_2, 1) \wedge (q_3, 3)).$$

We say that this transition is satisfiable if there exists a set S where $S \subseteq Q \times [3]$ such that by assigning tt to S and ff to $Q \times [3] - S$, the above formula holds true. Note that there may be many satisfying assignments for one such formula (or indeed, none). The intuition behind a transition of the automaton is that a satisfying assignment S is chosen, and for each $(q, k) \in S$ a copy of the automaton is spawned in direction k in state q. In this example $\{(q_1, 1)\}$ is a satisfying assignment and therefore we could simply send one automaton in direction 1.

As with the non-deterministic term tree automaton we must define the notion of a run tree and an accepting run tree for an alternating term tree automaton.

Definition 4.1.3. Given an alternating term tree automaton $M = \langle Q, A, \delta, q_0, Acc\rangle$ we define a ***run tree*** r over t as follows: $r : T \longrightarrow Q \times Dom(t)$, where $T \subseteq \{1, 2, \cdots, l\}^*$ for some fixed l that depends on M. A run-tree is defined inductively as follows:

- $r(\epsilon) = (q_0, \epsilon)$;
- if $r(\alpha)$ is defined and equal to (q, z), this means that the node α represents one copy of the automaton M sitting on node $z \in t$ in state q. Suppose that $r(\alpha)$ has precisely the children $r(\alpha i)$ for $i \in I$. Then if, $r(\alpha i) = (q_i, y_i)$ for $i \in I$ one of the following must hold for each $i \in I$:

  1. $y_i = zx_i$ for some $x_i \in X$ (where X is the monoid underlying $Dom(T)$), in which case we define $k_i = x_i$;
  2. $y_i = z$, in which case we define $k_i = 0$; or
  3. y_i is the parent of z, in which case we define $k_i = -1$.

  And finally, $\{(q_i, k_i) : i \in I\}$ is a satisfying assignment for $\delta(q, t(z))$.

Definition 4.1.4. Let r be a run tree over the A-tree $t : T \longrightarrow A$ of the alternating term tree automaton M. We say that a run tree is ***accepting*** if and only if the following hold:

1. If $\pi = \pi_1\pi_2 \cdots$ is an infinite path in r then
$$\max \{\Omega(q) : q \in \mathsf{Inf}(\pi)\}$$
is even, where $\mathsf{Inf}(\pi)$ consists of the states that occur infinitely often in π.
2. If π is a finite path ending with terminal node α, then $(\pi_1(r(\alpha)), t(\pi_2(r(\alpha)))) \in Fin$.

As before, we say that t is ***accepted*** by M if and only if there exists an accepting run tree over t.

Remark 4.1.5. We draw the attention of the reader to the difference between the run trees of non-deterministic automaton versus those of the alternating kind. Given a tree $t : T \longrightarrow A$, the former has the same domain T. Whereas with the latter the run tree can take on a completely different shape to that of T.

We will require the following well-known results from tree automata theory.

Theorem 4.1.6. *The emptiness problem for a non-deterministic tree automaton with parity acceptance condition is decidable.*

Proof. See [Rab72] for the proof of this result with respect to non-deterministic tree automata with Rabin chain acceptance condition. Conversions between parity acceptance and other kinds of acceptance (Muller, Streett, etc.) are summarised and explained in the survey [Tho97]. □

Theorem 4.1.7. *Let M be a 2-way alternating term tree automaton with parity acceptance condition over A-trees. There exists a non-deterministic term tree automaton N with parity acceptance condition such that M accepts the A-tree t if and only if N accepts t.*

Proof. This is another standard construction in automata theory [Var98]. A particularly readable account is given here [Cac01]. □

Theorem 4.1.8. *Let φ be an MSO sentence over the relational vocabulary induced by an A-tree as given in Chapter 2. There exists a non-deterministic term tree automaton M_φ with parity acceptance condition such that for any A-tree s, M_φ has an accepting run over s if and only if $\mathbf{s} \models \varphi$, where $\mathbf{s}$ is the logical representation of the term tree s as defined in Chapter 2.*

Proof. We refer the reader to [Tho97] for an exposition of this construction. □

In this section we will make use of both the non-deterministic and alternating varieties of term tree automata with parity acceptance condition. In future, we will just refer to them without explicit mention to the parity acceptance condition.

Furthermore, we modify the definition of the alternating term tree automaton as follows. Instead of a transition function where $\delta(q, f) = \varphi$ where φ is a positive boolean formula over

$$Q \times (Dir(f) \cup \{0, -1\})$$

we will allow $\delta(q, f)$ to be a positive boolean formula[2] over

$$Q \times 2^Q \times Dir(f)$$

For example, suppose that $(q_2, \{q_1, q_2\}, 3)$ and $(q_1, \{q_4\}, 2)$ are a satisfying assignment for δ. This means that we send one automaton in direction 3 in state q_2 but *echoing* the states q_1, q_2 on the way there. Similarly, we send one automaton in direction 2 in state q_1 but echoing the state q_4 on the way there. This really serves as a shortcut for the operation of remaining on the current node for a finite number of steps and at each step being in a different state, before progressing down to a particular child.

Remark 4.1.9. The resulting run tree for an alternating parity tree automaton with echoing will have *labeled* edges. In particular, the run tree r is defined as follows:

1. $r(\epsilon) = (q_0, \epsilon)$;

2. if $r(\alpha)$ is defined and equal to (q, z), this means that the node α represents one copy of the automaton M sitting on node $z \in t$ in state q. We consider $\delta(q, t(z))$ and suppose it is satisfiable. Let S be one such satisfying assignment. Suppose that $S = \{s_1, s_2, \cdots, s_p\}$ where $s_i = (q_i, P_i, x_i)$ and $x_i \in Dir(t(z))$. Then we set $r(\alpha i) \xrightarrow{P_i} (q_i, \alpha x_i)$ for $1 \leq i \leq p$.

A run tree is then said to be accepting if for every path π, the sequence of states occurring along π (now including those along the edges as well) satisfy the acceptance condition.

[2]Note that in doing so, we will never actually make use of direction -1, i.e. to visit the parent node.

4.1.2 Botanical Gardens

We fix a level-2 recursion scheme $R = \langle N, \Sigma, V, S, \mathcal{R} \rangle$ as defined in Chapter 2. In this section we define no less than three different types of trees that we associate with a recursion scheme. In particular, we develop the notion of the *value tree*, the *computation tree* and the *canonical traversal tree*. We will make extensive use of all three and therefore it is important to understand not only what each tree represents but also the relationships between them.

The value tree is, fortunately, nothing new and is defined to be the unique term tree defined by the recursion scheme, namely $[\![R]\!]$, using the notation from previous chapters. Recall that this is a term tree over the ranked alphabet $\Sigma^\perp$ (not Σ).

The computation tree is, as hinted at in the introduction, an infinite λ-term that ultimately β-reduces to $[\![R]\!]$. The computation tree will be defined by transforming the level-2 scheme R into a level-0 scheme R^0 and it is then set to $[\![R^0]\!]$. The inspiration for this comes directly from Knapik et al. who use a similar construction in [KNU01] although they generate the computation tree with a level-1 scheme. Intuitively, the computation tree, due to its regular nature, can be viewed as a more tractable representation of the value tree.

As described above, the relationship between the computation tree and the value tree is β-reduction. The canonical traversal tree captures exactly this: it gives an explicit *strategy* to traverse the computation tree in such a way that the normal form (namely $[\![R]\!]$) is extracted from the computation tree.

For the remainder of this section we will assume the following simplifying assumptions for the recursion scheme R (1) every non-terminal has type level 2, and (2) each non-terminal has at least one level-0 argument. Note that these assumptions can be made without loss of generality as any grammar that does not conform to them can be converted into one that does.

Value Trees

Definition 4.1.10. The ***value tree*** of the level-2 recursion scheme R is the $\Sigma^\perp$-tree given by $[\![R]\!]$, as defined in Chapter 2.

Computation Trees

In order to define the computation tree for a level-2 recursion scheme $R = \langle N, \Sigma, V, S, \mathcal{R} \rangle$, we require two transformations. The first transformation is given by the following set of steps:

1. For each production rule $F\vec{\varphi}\vec{x} = r$ in $\mathcal{R}$, we hereditarily η-expand every subterm of r. Note that we do this even for ground type terms occurring in the operand position; for example the ground type term e becomes $\lambda.e$. The use of these dummy lambdas (for ground type terms) will become obvious later.

2. After η-expanding each rule, we consider every subterm of the form $Dt_1 \cdots t_n : o$ where D is a non-terminal and replace this with $(((@_D D)t_1) \cdots t_n)$, where $@_D$ is a new constant of type $(A_D, A_{t_1}, \cdots, A_{t_n}, o)$, where A_X is the type of the term X.

3. Lastly, we rename bound variables or formal parameters afresh so that any two variables that occur in distinct rules have distinct names.

This completes our first transformation. We now have a level-2 recursion scheme[3] $R' = \langle N, \Sigma \cup \{@_D : D \in N\}, V \cup Z, S, \mathcal{R}' \rangle$, where Z represents any new variables we may have introduced through the η-expansion or variable renaming. $\mathcal{R}'$ is as dictated above.

Example 4.1.11. Consider the following level-2 recursion scheme R with $N = \{S : o, F : ((o,o),o,o,o), G : (o,o)\}$, $\Sigma = \{f : (o,o,o), g : (o,o), a : o, b : o\}$, $V = \{\varphi : (o,o), x : o\}$ and the following production rules:

$$\begin{aligned} S &= Fgab \\ F\varphi xy &= f(F(F\varphi x)yx)(G(\varphi y)) \\ Gx &= g(Gx) \end{aligned}$$

Applying the above transformation results in the following production rules:

$$\begin{aligned} S &= @_F F(\lambda z_1.gz_1)(\lambda.a)(\lambda.b) \\ F\varphi xy &= f(\lambda.@_F F(\lambda w_1.@_F F(\lambda w_2.\varphi(\lambda.w_2))(\lambda.x)(\lambda.w_1))(\lambda.y)(\lambda.x))(\lambda.y)(\lambda.x))(\lambda.@_G G(\lambda.\varphi(\lambda.y))) \\ Gu &= g(\lambda.@_G G(\lambda.u)) \end{aligned}$$

Note the renaming of the formal parameter of G.

Now we define the next transformation. Given $R' = \langle N, \Sigma \cup \{@_D : D \in N\}, V \cup Z, S, \mathcal{R}' \rangle$, we define a level-0 grammar $R^0 = \langle N^0, \Sigma^0, V^0, S^0, \mathcal{R}^0 \rangle$ where:

1. $N^0 = \{F^0 : o \mid F \in N\}$;
2. $V^0 = \emptyset$;
3. Σ^0 consists of:
   (a) If $f \in \Sigma$ with type $o^n \to o$, then $f^0 : o^n \to o \in \Sigma^0$ and $Dir(f^0) = \{1, 2, \cdots, n\}$.
   (b) If $v : A$ is a variable in $V \cup Z$, then $v^0 : A$ is in Σ^0.
   (c) For each explicit "apply symbol" $@_F : A$ where $A = (A_0, A_1, \cdots, A_n, o)$ we add $@^0_n : o^{n+1} \to o$ to Σ^0 with directions $Dir(@_n) = \{0, 1, \cdots, n\}$.
   (d) Finally, we have a set of lambda nodes:
   $$\{\lambda\vec{x} : (o,o) \mid \vec{x} \subseteq V \cup Z\}$$
   Note that we also allow for the case where $\vec{x} = \emptyset$, we write this as simply λ.
4. Intuitively, the set of production rules $\mathcal{R}^0$ is given by currying each rule in $\mathcal{R}'$ and then treating every symbol occurring on the right hand side (minus, of course, the non-terminals) as a terminal symbol. Formally, however, we first define a transformation α from λ-terms with constants in $\Sigma \cup N \cup \{@_D : D \in N\}$ into typed applicative terms over $\Sigma^0 \cup N^0$. The transformation α is defined inductively:
   (a) If $r = f : A$ for $f \in \Sigma$, then $\alpha(f) = f^0 : A$;
   (b) If $r = v : A$ for $v \in V \cup Z$, then $\alpha(v) = v^0 : A$;

[3]This is not a scheme in the strictest sense as it contains λ-abstractions and signature constants of level greater than 1. However, this is not a serious problem as this is only an intermediary structure.

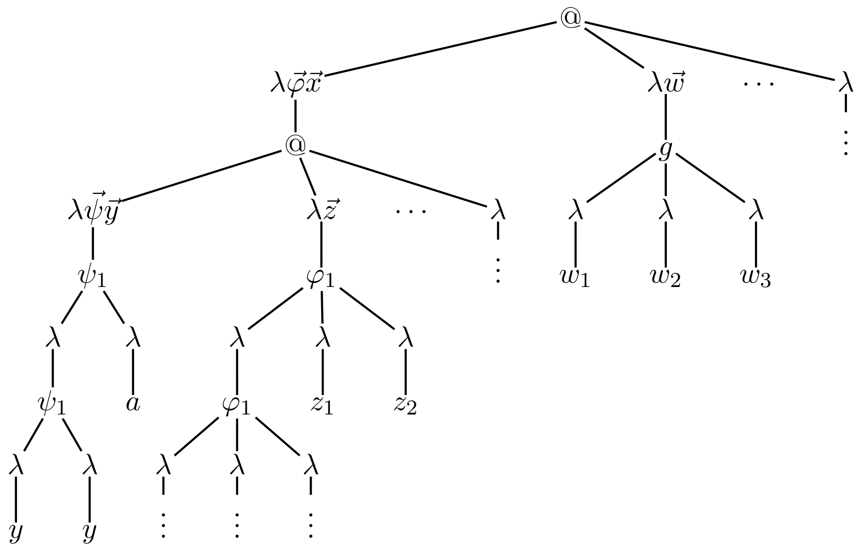


Figure 4.1: The computation tree for a recursion scheme.

(c) If $r = F : A$ for $F \in N$, then $\alpha(F) = F^0 : o$;

(d) If $r = @_F : (A_0, A_1, \cdots, A_n, o)$ then $\alpha(@_F) = @^0_n : o^{n+1} \to o$;

(e) If $r = \lambda\vec{z}.s : o$ where $\vec{z}$ is possibly empty and s is not a λ-abstraction, then:

$$\alpha(\lambda\vec{z}.s) = (\lambda\vec{z})(\alpha(s)) : o$$

where $(\lambda\vec{z})$ on the right hand side is regarded as a single symbol of type (o, o).

(f) Finally, in the case where we have an applicative term $st_1 \cdots t_n : o$, then $\alpha(st_1 \cdots t_n) = \alpha(s)\alpha(t_1) \cdots \alpha(t_n) : o$.

Given a production rule $F\vec{v} = r$ in $\mathcal{R}'$, this now becomes $F^0 = \lambda\vec{v}.\alpha(r)$.

Definition 4.1.12. The ***computation tree*** of a recursion scheme R is given by the tree $[\![R^0]\!]$.

We include an example of the computation tree of a recursion scheme in Fig. 4.1.

Remark 4.1.13. Although we have defined the alphabet of R^0, Σ^0, to be non-intersecting with that of R, there is actually no need to enforce this. For example, there is no need to rename $f \in \Sigma$ to f^0; this was done so merely to aid clarity in the above transformation. The same applies to non-terminals and variables in R. In fact, for the remainder of this chapter we remove all the 0 superscripts for these symbols in Σ^0 when convenient. In future we will also often omit the subscript of @ nodes when it is obvious from the context.

Definition 4.1.14. Let $[\![R^0]\!]$ be the computation tree associated with the recursion scheme R. Using the notation defined earlier, this is a Σ^0-tree. If $\alpha \in Dom([\![R^0]\!])$ has label $\lambda\vec{v}$ for some $\vec{v}$, we say that α is a ***lambda node***. Otherwise, we say it is a ***non-lambda node***.

Definition 4.1.15. We define a binary relation over the node-set of the computation tree $[\![R^0]\!]$, called the ***enabling*** relation, which we write as follows $m \vdash_i m'$, to be read as m i-enables m'.

1. A lambda-node l is i-enabled by its parent node m in t, where l is the ith child of m.
2. A variable node with label ξ_i is i-enabled by its binder. This is defined to be the closest (en route to the root) node occurrence with label $\lambda\xi_1 \cdots \xi_n$.

It is important to point out that any node $\alpha \in [\![R^0]\!]$ labeled by $@_n$ has $n+1$ children and they are $\alpha 0, \alpha 1, \cdots, \alpha n$. For any other symbol $s \in \Sigma^0$, if s has rank n then $Dir(s) = \{1, 2, \cdots, n\}$.

Remark 4.1.16. When it is convenient and clear from the context we will often refer to a node by its label.

Canonical Traversal Trees and their Annotated Counterparts

The reader familiar with Knapik et al.'s proof of the MSO decidability of safe level-2 recursion schemes will recall the notion of a "derived tree" (Section 2.2. of [KNU01]). Briefly, given an infinite λ-term (such as the one described by a computation tree), the derived tree corresponds to its β-normal form. Knapik et al. define an entirely deterministic procedure for traversing the λ-term, starting from the root, in such a way that the β-normal form is extracted. The canonical traversal tree is analogous to this deterministic procedure in the sense that it embodies a description of how to traverse $[\![R^0]\!]$ in such a way to extract $[\![R]\!]$.

There are several ways to define the canonical traversal tree and the way we have chosen to do so here is via a game semantics approach. Whilst we have not formally introduced game semantics in this chapter nor previous ones, we justify this by our very limited use of it. Namely, we will only need the concepts of justified sequences (adapted for trees), P-visibility and P-views and these will be introduced on an "as-needed" basis. All of these are basic concepts of game semantics and can be found in, for example, [HO00].

We will now introduce the canonical traversal tree, followed by the annotated canonical traversal tree relative to a non-deterministic tree automaton N. Finally, we prove the correspondence between an accepting annotated canonical traversal tree relative to N and accepting runs of N over $[\![R]\!]$. This correspondence is crucial to our result.

In the following, let $R = \langle N, \Sigma, V, S, \mathcal{R} \rangle$ be a level-2 recursion scheme and let $[\![R^0]\!]$ be the computation tree induced by this scheme.

Definition 4.1.17. A ***traversal*** over $[\![R^0]\!]$ is a (finite) sequence

$$\alpha_1 \alpha_2 \cdots \alpha_n$$

where $\alpha_i \in Dom([\![R^0]\!])$. Note that it is possible for $\alpha_i = \alpha_j$ for $i \neq j$, however they are still distinct *nodes occurrences* of the same node in the traversal.

Remark 4.1.18. The reader familiar with Luke Ong's results on the same topic will notice a disparity between some of our definitions. Traversal is one such term. Here, a traversal is a generalisation of a path and is an arbitrary sequence of nodes with no particular structure.

Example 4.1.19. We consider the (finite) computation tree in Figure 4.2. Note that we have superscripted each node with a unique number for easy reference.

If we assume that these numbers form the domain of the computation tree, then the following is a valid traversal

1 2 5 8 12 3 6 11 15 17 19

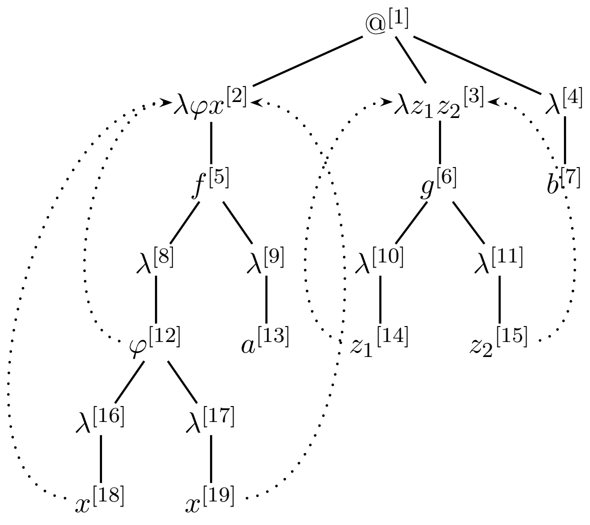


Figure 4.2: A finite computation tree. The enabling relation for non-lambda nodes is shown by the dotted lines. Recall that lambda nodes are always enabled by their parent.

Note also that any path from any node α in the computation tree is, of course, a traversal. Thus, every path is a traversal but the converse is generally false.

Remark 4.1.20. When convenient we will identify a traversal over $[\![R^0]\!]$ with the sequence of labels of the nodes occurring on that traversal. Given the traversal in the preceding example we can identify this with:

$$@\ \lambda\varphi x\ f\ \lambda\ \varphi\ \lambda z_1 z_2\ g\ \lambda\ z_2\ \lambda\ x$$

It must be understood that writing a traversal in the above manner is merely to aid our understanding and is not a unique representation of a traversal. For example, the above also corresponds to the labels on the traversal:

1 2 5 9 12 3 6 9 15 9 19

Definition 4.1.21. A ***traversal with pointers*** over $[\![R^0]\!]$ is a pair

$$(\alpha_1\alpha_2\cdots\alpha_n, P)$$

where $\alpha_1\alpha_2\cdots\alpha_n$ is a traversal over $[\![R^0]\!]$ as above, and $P : [n] \longrightarrow \omega\times[n]$ is a *partial function.* P defines the pointer relation for the traversal; in particular if $(k, i, l) \in P$ this means that there exists an i-pointer from the kth node in the sequence to the lth one. As P is a function, there is at most one pointer emanating from each node occurrence.

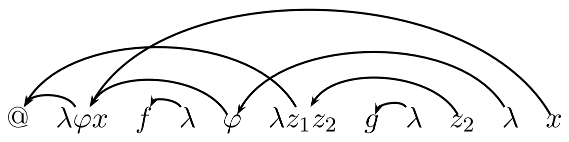
Example 4.1.22. Let us consider the traversal from the previous example; repeated below.

1	2	5	8	12	3	6	11	15	17	19	traversal
@	$\lambda\varphi x$	f	λ	φ	$\lambda z_1 z_2$	g	λ	z_2	λ	x	corresponding labels
1	2	3	4	5	6	7	8	9	10	11	index in sequence

Based on this traversal we will create a traversal with pointers (α, P) where P is given by the following (in the form of a relation):

$$P = \left\{ \begin{array}{c} (11,2,2) \\ (10,2,5) \\ (9,2,6) \\ (8,2,7) \\ (6,1,1) \\ \underline{(5,1,2)} \\ (4,1,3) \\ (2,0,1) \end{array} \right\}$$

In particular, the underlined tuple indicates that there is a 1-pointer from the 5th node in the sequence to the 2nd. There are no pointers emanating from the 7th, nor 3rd not 1st nodes in the sequence. This is represented pictorially as follows:



Definition 4.1.23. We say that a traversal with pointers (α, P) over $[\![R^0]\!]$ is ***justified*** if and only if the following holds:

1. α is a sequence of alternating lambda and non-lambda nodes in $[\![R^0]\!]$.
2. It satisfies the ***pointer condition***: every node α_j that occurs in α (except those labelled by a @ or a signature constant) has an i-pointer to some earlier node-occurrence α_k in the sequence ($k < j$). Furthermore, $\alpha_k \vdash_i \alpha_j$ in $[\![R^0]\!]$. We say that the node occurrence α_j is ***justified*** by the node occurrence α_k.

Example 4.1.24. It is easy to verify that the traversal with pointers in the preceding example is a justified traversal.

Definition 4.1.25. Let $\pi = \pi_1 \pi_2 \cdots \pi_n$ be a finite path (from the root) in $[\![R^0]\!]$. We define the traversal with pointers induced by the path π to be (π, P^π) where $P^\pi : [n] \longrightarrow \omega \times [n]$ is the pointer relation where

$$(m, i, n) \in P^\pi \text{ if and only if } \pi_n \vdash_i \pi_m.$$

In other words, P^π is the enabling relation of the computation tree but *restricted* to those nodes occurring on this path.

Example 4.1.26. Let π be a finite path (from the root) in $[\![R^0]\!]$. It is easy to verify that (π, P^π) is a justified traversal over $[\![R^0]\!]$.

Definition 4.1.27. Let (α, P) be a traversal where $\alpha = \alpha_1 \alpha_2 \cdots \alpha_n$. Suppose that $k \leq n$, then $(\alpha, P) \upharpoonright_k$ is defined to be the traversal:

$$(\alpha_1 \alpha_2 \cdots \alpha_k, P \upharpoonright_{[k]})$$

where $P\restriction_{[k]}$ is the restriction of the function P to the domain $[k]$.

Definition 4.1.28. Let (α, P) be a justified traversal over $[\![R^0]\!]$, we define the P-view of (α, P) recursively below. In the following we will represent (α, P) pictorially as we did at the end of Example 4.1.22. We let n range over non-lambda nodes.

$$\begin{aligned}
\ulcorner @ \urcorner &= @ \\
\ulcorner f \urcorner &= f \text{ where } f \text{ is a signature constant} \\
\ulcorner t\; n \cdots \lambda\vec{v} \urcorner &= \ulcorner t \urcorner\; n\, \lambda\vec{v} \\
\ulcorner t\; \lambda\vec{v}\; n \urcorner &= \ulcorner t\; \lambda\vec{v} \urcorner n
\end{aligned}$$

In the third case, if n points to a node occurrence in t, say l, then if l appears in $\ulcorner t \urcorner$, then n is defined to point to l in the resulting sequence, otherwise n has no pointer in $\ulcorner t \urcorner$. Similarly for the fourth case: if n points to a node occurrence l in $t\ \lambda\vec{v}$, then if l appears in $\ulcorner t\ \lambda\vec{v} \urcorner$, n is defined to point to l in the resulting sequence, otherwise n has no pointer.

Example 4.1.29. We denote the justified traversal from Example 4.1.22 by (α, P). The P-view is given by:

$$\begin{tikzcd}
@ & \lambda\varphi x \arrow[l, bend right] & f & \lambda \arrow[l, bend right] & \varphi \arrow[lll, bend right] & \lambda \arrow[l, bend right] & x \arrow[lllll, bend right]
\end{tikzcd}$$

As another example, the justified traversal $(\alpha, P)\restriction_7$ is given by:

$$\begin{tikzcd}
@ & \lambda\varphi x \arrow[l, bend right] & f & \lambda \arrow[l, bend right] & \varphi \arrow[lll, bend right] & \lambda z_1 z_2 \arrow[lllll, bend right] & g
\end{tikzcd}$$

Its P-view is given below:

$$\begin{tikzcd}
@ & \lambda z_1 z_2 \arrow[l, bend right] & g
\end{tikzcd}$$

Definition 4.1.30. Let $(\alpha_1\alpha_2\cdots\alpha_n, P)$ be a justified traversal over $[\![R^0]\!]$. It is said to satisfy ***P-visibility*** if and only if for each prefix $(\alpha, P)\restriction_k$ (where $k \leq n$), if α_k has a pointer to some node α_l $(l < k)$, then α_k retains this pointer in $\ulcorner(\alpha, P)\restriction_k\urcorner$.

Lemma 4.1.31. *Let π be a path in $[\![R^0]\!]$. Then $\ulcorner(\pi, P^\pi)\urcorner$ satisfies P-visibility and furthermore*

$$\ulcorner(\pi, P^\pi)\urcorner = (\pi, P^\pi).$$

Proof. We will show by induction on the length of π that $\ulcorner(\pi, P^\pi)\urcorner = (\pi, P^\pi)$ (P-visibility then follows automatically).

1. $|\pi| = 1$. Then this path consists of only the root of the computation tree which must be labeled by @ or $f \in \Sigma$. The result holds trivially.

2. Suppose now that $|\pi| > n + 1$ and that the hypothesis is true for all paths of length $\leq n$. Let $\pi = \pi_1\pi_2\cdots\pi_n\pi_{n+1}$. We perform a case analysis on the label of π_{n+1} (let m range over non-lambda nodes):

   (a) Suppose that π_{n+1} is a non-lambda node, then:

   $$\ulcorner t\ m \urcorner = \ulcorner t \urcorner\, m$$

   The result now follows by the induction hypothesis.

(b) On the other hand, suppose that π_{n+1} is a lambda node, then:

$$\ulcorner t\ \overset{\frown}{m \cdots \lambda\vec{v}}\urcorner = \ulcorner t \urcorner\ \overset{\frown}{m\,\lambda\vec{v}}$$

The result now holds trivially by observing that as this is a path then m occurs directly before $\lambda\vec{v}$ in the traversal (i.e. the $\cdots$'s are empty) and the rest now follows by the induction hypothesis.

$\square$

We extend the above notions to *trees* with pointers. These are simply trees where each node of the tree may have a back-edge to one of its ancestors in the tree.

Definition 4.1.32. A ***tree with pointers*** over $[\![R^0]\!]$ is a pair

$$(t : T \to Dom([\![R^0]\!]), P)$$

where $t : T \to Dom([\![R^0]\!])$ is a tree and $P : T \longrightarrow \omega \times T$ is a partial function that denotes the pointer relation. In particular, if $(a, i, b) \in P$ then the node $a \in T$ has an i-pointer to the node $b \in T$. Again, note that it is a function so that each node in the tree has at most one pointer to another node.

Definition 4.1.33. Let $(t : T \to Dom([\![R^0]\!]), P)$ be a tree with pointers and let π be a finite path (from the root) in $t : T \to [\![R^0]\!]$, we denote by

$$(t : T \to Dom([\![R^0]\!]), P) \upharpoonright_\pi$$

the following traversal with pointers:

$$(t(\pi_1)t(\pi_2)\cdots t(\pi_n), P \upharpoonright_\pi)$$

where $P \upharpoonright_\pi$ is defined so that $(k, i, l) \in P \upharpoonright_\pi$ if and only if π_k has an i-pointer to π_l in P for $l, k \leq n$. Note that this is nothing more than the P relation restricted to the path π.

Definition 4.1.34. We say that a tree with pointers $(t : T \longrightarrow Dom([\![R^0]\!]), P)$ over $[\![R^0]\!]$ is ***justified*** if and only if every finite *path* of the tree (including the pointer information relevant to that path) is a justified traversal over $[\![R^0]\!]$. Formally, for every finite path π from the root of the tree t we have that $(t, P) \upharpoonright_\pi$ is a justified traversal over $[\![R^0]\!]$.

We are now in a position to define the canonical traversal tree for a recursion scheme R. It is a tree with pointers that is constructed from $[\![R^0]\!]$ in an entirely deterministic fashion described below. We will see that this tree possesses many good properties that are vital to our result.

Definition 4.1.35. Given a computation tree $[\![R^0]\!]$, the ***canonical traversal tree*** for $[\![R^0]\!]$ is the tree with pointers $c_{[\![R^0]\!]} = (c : T \longrightarrow Dom([\![R^0]\!]), P)$, defined inductively as follows:

1. $c(\epsilon) = \epsilon$.

2. If $c(\alpha) = z$, we perform a case analysis on $[\![R^0]\!](z)$:

(a) *(App)* $[\![R^0]\!](z) = @$. Then we define $c(\alpha 0) = z0$, and we add $(\alpha 0, 0, \alpha) \in P$. Pictorially, we have created the following 0-pointer:

$$\begin{tikzcd}
\underbrace{\alpha}_{z} & \underbrace{\alpha 0}_{z0} \arrow[l, bend right]
\end{tikzcd}$$

(b) *(Sig)* $[\![R^0]\!](z) = f$, then we define $c(\alpha i) = zi$ for $1 \leq i \leq n$ where $\varrho(f) = n$. Furthermore, we have that $(\alpha i, i, \alpha) \in P$ for $1 \leq i \leq n$. Pictorially, we have the following:

$$\begin{tikzcd}
 & & \alpha & & \\
\alpha 1 \arrow[urr, dash] \arrow[urr, bend left] & \alpha 2 \arrow[ur, dash] \arrow[ur, bend right] & & \cdots & \alpha n \arrow[ull, dash] \arrow[ull, bend right]
\end{tikzcd}$$

(c) *(Lambda)* If $[\![R^0]\!](z) = \lambda \vec{v}$ where $\vec{v}$ is possibly empty, then we define $c(\alpha 1) = z1$. If $[\![R^0]\!](z1)$ is a variable node, then we must also add a pointer to P. To determine this pointer we consider the path from the root of c to α, which we denote by π. Next we consider $\ulcorner c_{[\![R^0]\!]} \upharpoonright_{\pi} \urcorner$ – i.e. the P-view of this path. We will see (later) that there exists precisely one node occurrence in this P-view, w, such that $w \vdash_i z1$ for some i. Let β be the node in on the path π that gives rise to this node occurrence of w. We add a pointer $(\alpha 1, i, \beta)$.

(d) *(Var)* Let $[\![R^0]\!](z) = v$ where v is a variable, and suppose we have the following situation in $c_{[\![R^0]\!]}$:

$$\begin{tikzcd}
\underbrace{\beta}_{y} & \underbrace{\beta 1}_{w} & \cdots & \underbrace{\alpha}_{z} \arrow[ll, bend right]
\end{tikzcd}$$

so that α has an i-pointer to $\beta 1$. Then we set $c(\alpha 1) = yi$ and include an i-pointer from $\alpha 1$ to β (i.e. $(\alpha 1, i, \beta) \in P$) so that we arrive at the following:

$$\begin{tikzcd}
\underbrace{\beta}_{y} & \underbrace{\beta 1}_{w} & \cdots & \underbrace{\alpha}_{z} \arrow[ll, bend right] & \underbrace{\alpha 1}_{yi} \arrow[llll, bend right]
\end{tikzcd}$$

Before we prove that the canonical traversal tree is a well-defined construction, let us assume for a moment that it is in order to introduce some notation.

Let $c_{[\![R^0]\!]} = (c : T \longrightarrow Dom([\![R^0]\!]), P)$ be the canonical traversal tree associated with the computation tree $[\![R^0]\!]$. If $\alpha \in T$ and $[\![R^0]\!](c(\alpha)) = \lambda \vec{v}$, then we refer to α as a ***lambda*** node. Otherwise, it is a ***non-lambda*** node. This is consistent with the definition of lambda and non-lambda nodes in a computation tree.

To show that the construction of the canonical traversal tree $c_{[\![R^0]\!]}$ is well-defined, we actually prove the following stronger lemma:

Lemma 4.1.36. *Suppose that $c_{[\![R^0]\!]} = (c, P)$ is defined up to some node $\alpha \in Dom(c)$. Let us denote by π the path from the root of c to α, and let $\pi = \pi_1 \pi_2 \cdots \pi_n$. The following hold for π:*

1. *If for some j we have that:*

$$[\![R^0]\!](c(\pi_j)) = \lambda \vec{v}$$

and $|\vec{v}| = n$, where $n > 0$, then

$$c(\pi_{j-1})$$

is a node in the computation tree that has (at least) children in directions $1, 2, \cdots, n$.

2. $(\pi, P\upharpoonright_\pi)$ *is a justified traversal over* $[\![R^0]\!]$ *satisfying P-visibility.*

3. $\ulcorner(\pi, P\upharpoonright_\pi)\urcorner$ *is a path in the computation tree* $[\![R^0]\!]$.

Proof. We proceed by induction on the length of the path π. For a path of length 1, the result holds trivially as this consists of solely the root of the canonical traversal tree, which is by construction the root of the computation tree.

Let π be a path of length $n+1$ and suppose that the hypothesis is true for all paths with length $\leq n$. We perform a case analysis on the rule applied to the nth node in the path to construct the $n+1$th. There are only two non-trivial cases (in both we let m denote a non-lambda node):

1. (*Var*) By the induction hypothesis we have the following scenario:

$$\begin{tikzcd}[row sep=tiny]
\pi_1 & \pi_2 & \cdots & \pi_{j-1} & \pi_j & \pi_{j+1} & \cdots & \pi_n \arrow[lll, bend right] & \text{The path } \pi \\
z_1 & z_2 & \cdots & z_{j-1} & z_j & z_{j+1} & \cdots & z_n & \text{where } z_i = c(\pi_i) \\
? & ? & \cdots & ? & ? & ? & \cdots & v & \text{corresponding labels in } [\![R^0]\!]
\end{tikzcd}$$

By items (2) and (3) of the induction hypothesis, the P-view of the above results in a path of the computation tree of the following shape:

$$\begin{tikzcd}[column sep=tiny]
\ulcorner\pi_1\cdots\pi_{j-1} & \pi_j\urcorner\cdots & \pi_n \arrow[l, bend right]
\end{tikzcd}$$

Therefore, it must be the case that z_j is the binder of z_n in the computation tree, and therefore must be of the form $\lambda\vec{v}$ such that $v \in \vec{v}$. By item (1) of the induction hypothesis, z_{j-1} in the computation tree must have (at least) the children $1, 2, \cdots, |\vec{v}|$. Furthermore, $[\![R^0]\!](z_{j-1})$ must be, by the induction hypothesis, a non-lambda node, say m. We can now refine the picture to produce the following:

$$\begin{tikzcd}[row sep=tiny]
\pi_1 & \pi_2 & \cdots & \pi_{j-1} & \pi_j & \pi_{j+1} & \cdots & \pi_n \arrow[lll, bend right] \\
z_1 & z_2 & \cdots & z_{j-1} & z_j & z_{j+1} & \cdots & z_n \\
? & ? & \cdots & m & \lambda\vec{v} & ? & \cdots & v
\end{tikzcd}$$

As v must be i-enabled by $\lambda\vec{v}$ for $i \leq n$, it is legitimate for us to extend the above path as follows :

$$\begin{tikzcd}[row sep=tiny]
\pi_1 & \pi_2 & \cdots & \pi_{j-1} & \pi_j & \pi_{j+1} & \cdots & \pi_n \arrow[lll, bend right] & \pi_{n+1} \arrow[lllll, bend right] \\
z_1 & z_2 & \cdots & z_{j-1} & z_j & z_{j+1} & \cdots & z_n & z_{j-1}i \\
? & ? & ? & m & \lambda\vec{v} & ? & \cdots & v & \lambda\vec{z}
\end{tikzcd}$$

Note that $c(\pi_{n+1})$ is the ith child of z_{j-1} and this must, of course, be a lambda node as a path in the computation tree is a justified traversal. We must now check that each of the items of the induction hypothesis remain true. It is easy to verify that (1) remains true, by construction of the computation tree (by case analysis on the level of v). As for (2) and (3) we note that the above traversal is clearly justified and that the P-view of the traversal induced by this path is (using shorthand only):

$$\begin{tikzcd}[column sep=tiny] \ulcorner t \urcorner & m & \lambda\vec{z}. \arrow[l, bend right] \end{tikzcd}$$

where t is the path up to π_{j-2}. This is clearly a path in $[\![R^0]\!]$.

2. (Lambda) Now suppose that the path (up to length n) has the following form:

$$\begin{array}{ccc} \pi_1 & \cdots & \pi_n \\ z_1 & \cdots & z_n \\ ? & \cdots & \lambda\vec{v} \end{array}$$

By construction, the $n+1$th node in this path must give rise to the following:

$$\begin{array}{cccc} \pi_1 & \cdots & \pi_n & \pi_{n+1} \\ z_1 & \cdots & z_n & z_n1 \\ ? & \cdots & \lambda\vec{v} & m \end{array}$$

By the induction hypothesis $\ulcorner ? \ \cdots \ \lambda\vec{v} \urcorner$ is a path in $[\![R^0]\!]$, so clearly $\ulcorner ? \ \cdots \ \lambda\vec{v} \urcorner\, m$ is as well. Thus, there must exists a unique π_j the *occurs in the P-view* such that z_j is the unique binder of z_n1 (in the case where $[\![R^0]\!](z_n1)$ is a variable). By construction of the canonical traversal tree, π_{n+1} is thus defined to point to π_j. We arrive at:

$$\begin{tikzcd}[row sep=small, column sep=small] \pi_1 & \cdots & \pi_j & \cdots & \pi_n & \pi_{n+1} \arrow[llll, bend right] \\ z_1 & \cdots & z_j & \cdots & z_n & z_n1 \\ ? & \cdots & \lambda\vec{z} & \cdots & \lambda\vec{v} & m \end{tikzcd}$$

It is now trivial to verify that items (1), (2) and (3) still hold.

□

Example 4.1.37. Given the computation tree in Figure 4.2, the corresponding canonical traversal tree is shown in Figure 4.3. Recall that the canonical traversal tree is a tree with labels in $Dom([\![R^0]\!])$; for expository purposes our tree in Figure 4.3 is such that a node α is not shown to have label $c(\alpha)$ but instead is identified with $[\![R^0]\!](c(\alpha))$ – for ease of reading.

Before proceeding any further, let us pause for a moment to describe the intuition behind the canonical traversal tree.

Example 4.1.38. Let us consider the following recursion scheme R with non-terminals $\{S : o, F : ((o,o),o,o)\}$. The variables and signature constants can now be deduced from the following production rules:

$$\begin{array}{rcl} S & = & Fgb \\ F\varphi x & = & f(\varphi xx)a \end{array}$$

This is a very simple recursion scheme, indeed, $[\![R]\!] = f(gbb)a$. Its computation tree has already been introduced earlier in this section, Figure 4.2. Correspondingly, its canonical traversal tree can be found in Figure 4.3.

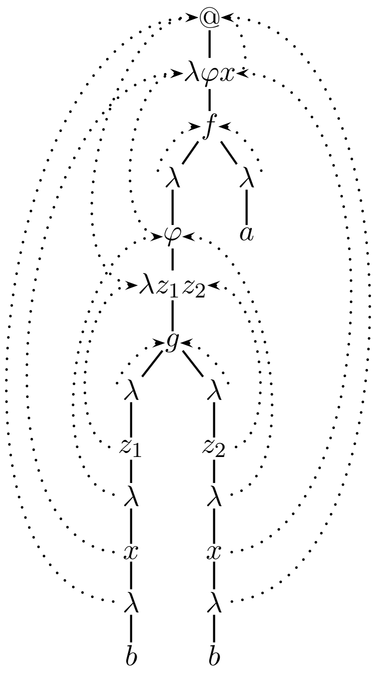


Figure 4.3: The canonical traversal tree induced by the computation tree from Figure 4.2.

The canonical traversal tree details how to traverse the computation tree in such a way that β-reductions are performed on an "as-needed" basis. In other words, these are local β-reductions [DR93]. Perhaps the easiest way to understand what we are trying to achieve is for the reader to assume the role of an automaton whose goal is to traverse the computation tree in such a way it extracts the β-normal form.

To follow this example, the reader will need to refer to Figure 4.2. The automaton starts at the root of the computation tree, denoted by node [1]. By construction of the computation tree, a node labeled by @ is indicative of a β-redex, and the leftmost child (the 0th one) is the λ-abstraction and the remaining children are what it is applied to. Therefore, it seems reasonable to enter the leftmost child as the body of the abstraction will surely prescribe the "shape" of normal form. We have moved from [1] to [2]. From [2] we simply keep progressing towards the body of the abstraction, taking us to [5]; the root of the body of the abstraction. In this example, [5] is labeled with a terminal symbol and it is thus clear that our normal form will be of the form $f \cdots$. We thus spawn two copies of the automaton, one in the direction of each child (we must find out what both are in the normal form). The rightmost child is quite unproblematic and leads us to the entire normal form of the right child. However, the leftmost child [8] is more complicated. When we reach [12] the automaton has, unfortunately, hit upon a variable φ. This is clearly undesirable because it is very probable that in β-normal form φ would have been substituted by another term. The automaton must therefore "evaluate" φ before progressing any further. From *our* point of view, we know exactly what to substitute for φ and this is the subtree rooted at [3]. We assume for a moment that the automaton has a way of knowing this and thus jumps to location [3]. [3] is labeled by a λ-abstraction, therefore as before, we progress down to its body, taking us to [6]. We spawn two automata now; one in the direction of each child. Both children lead to the same problem, the leftmost terminating in a variable z_1 and the other terminating in z_2, both of which are variables. At this point, the automaton must recall that the current subtree in which it is in (that rooted at [3]) is actually the substitution for φ. Therefore z_1, being the first parameter of $\lambda z_1 z_2$ refers to the first child of φ, and z_2 being the second parameter refers to the second child of φ. Each spawned automaton must return to the respective child of φ.

The above should hopefully illustrate the tall order that is placed on an automaton that wishes to extract the β-normal form from a computation tree. However, our construction of the canonical traversal tree captures *exactly* what such an automaton is attempting to do. Furthermore, at each point where the automaton had to make a "jump" to evaluate a variable, all the information required to successfully execute this jump is given by the pointers of the computation tree and the pointers of the canonical traversal tree.

To understand better how this achieved, we examine the canonical traversal tree in Figure 4.4. Note that we have left out the pointers, but we have outlined certain "regions" of the tree.

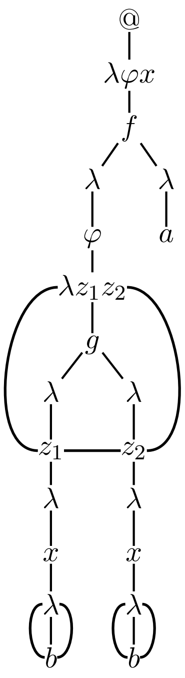


Figure 4.4: The canonical traversal tree induced by the computation tree from Figure 4.2.

In particular, if we look at the region outlined directly below φ, we have *exactly* what we wanted. The canonical traversal tree has effectively transplanted what should be substituted for φ in the β-normal form. Furthermore, the information contained within the pointers has enabled us to stitch the "exit points" (namely the occurrences of variables z_1 and z_2) to the correct children of φ as well. Similarly, we have outlined two regions (below the two occurrences of the node x), and these are, not surprisingly, the terms that must be substituted for x.

Given a tree $t : T \longrightarrow \Sigma \cup C$ where each node in T labelled by an element of C has precisely one child, we recall the definition of the collapse of t onto Σ from Chapter 2, denoted by t^{Σ}. Now we note that $c_{[\![R^0]\!]} = (c, P)$ where c is a tree over $\Sigma \cup C$, where C denotes the symbols that represent variables, λ-abstractions and so forth. The important thing to note is that for all the symbols in C, a node in c labeled by one such item has precisely one child.

Theorem 4.1.39. *Let $c_{[\![R^0]\!]} = (c, P)$ be the canonical traversal tree induced by the level-2 recursion scheme R. Then:*

$$c^{\Sigma} = [\![R]\!]$$

Proof. This is the formalisation of the intuition presented earlier. In particular, one shows that there is a 1-1 correspondence between paths in the value tree and paths in the canonical traversal tree. In particular, using game semantics, it is shown that a play in the strategy-denotation of the recursion scheme corresponds exactly to a path in the value tree and a path in the canonical traversal tree is an uncovering of such a play. A straightforward proof of this can be found here [Ong06a]. □

The above result is key to our analysis. Intuitively, it says that if we travel in a top-down fashion starting from the root of the canonical traversal tree, then, provided we ignore any non-Σ symbols, what we are traversing is $[\![R]\!]$. This leads to our next definition.

Definition 4.1.40. Let $N = \langle Q, \Sigma, \delta, q_0, Acc \rangle$ be a non-deterministic term tree automaton and let R be a level-2 recursion scheme (with signature Σ) with canonical traversal tree $c_{[\![R^0]\!]} = (c : T \longrightarrow Dom([\![R^0]\!]), P)$. An ***annotated canonical traversal tree with respect to*** N is any term tree with pointers $c^N_{[\![R^0]\!]} = (c^N : T \longrightarrow Q \times Dom([\![R^0]\!]), P)$ such that the following hold.

1. $c^N(\epsilon) = (q_0, c(\epsilon))$
2. If $c^N(\alpha) = (q, z)$ and $[\![R^0]\!](z) \notin \Sigma$ then $c^N(\alpha i) = (q, c(\alpha i))$ if $c(\alpha i)$ is defined.
3. If $c^N(\alpha) = (q, z)$ and $[\![R^0]\!](z) = f \in \Sigma$ and $c^N(\alpha i) = (q_i, c(\alpha i))$ then, $(q_1, \cdots, q_{\varrho(f)}) \in \delta(q, f)$

Note that c^N and c have the same domain and the same pointer relation P. It is merely in the labeling that c^N differs from c.

Thus, an annotated canonical traversal tree *reflects* a run of the automaton N on $[\![R]\!]$. We can imagine superimposing the automaton N on our canonical traversal tree: as we trace out the underlying tree (by ignoring any non-Σ symbols), we permit the automaton N to act on the terminals. Accordingly, we only change state when visiting a terminal symbol (in which case we reference δ).

Definition 4.1.41. Let N be a non-deterministic tree automaton as above. An annotated canonical traversal tree relative to N is ***accepting*** if and only if each maximal path satisfies the acceptance condition Acc.

Example 4.1.42. See Figure 4.5 for an annotated canonical tree based on the automaton with transition function below:

$$\begin{array}{rcl} (q_0, f) & = & \{(q_1, q_2), (q_2, q_2)\} \\ (q_1, g) & = & \{(q_2, q_2)\} \\ (q_2, g) & = & \{(q_1, q_1)\} \end{array}$$

and acceptance condition given by $Parity \cup Fin$, where $Fin = \{(q_2, a), (q_1, a)\}$.

Note that we only ever change state immediately after visiting a Σ-symbol. The annotated canonical traversal tree in Figure 4.5 is *not* accepting. In fact, in this case, it easy to verify that there are no accepting canonical traversal trees. Neither are there any accepting runs over $[\![R]\!]$.

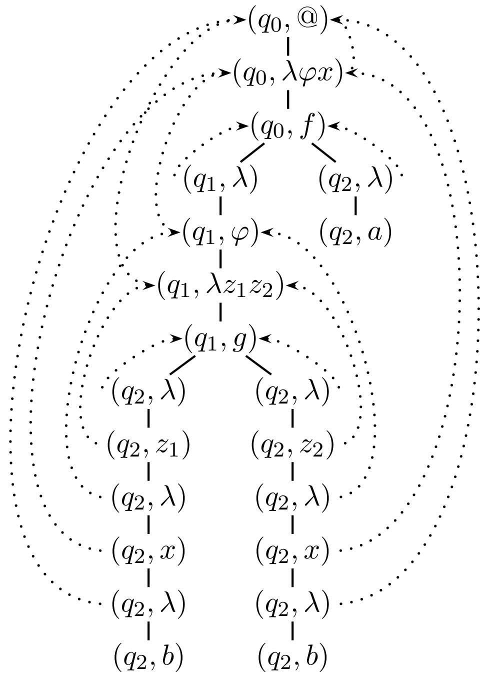


Figure 4.5: The *annotated* canonical traversal tree induced by the computation tree from Figure 4.2 relative the the automaton in Example 4.1.42.

Theorem 4.1.43. *$N \models [\![R]\!]$ if and only if there exists an accepting annotated canonical traversal tree over $[\![R^0]\!]$ with respect to N.*

Proof. This follows immediately from the definition of the annotated canonical traversal and Theorem 4.1.39. □

We conclude this section with a technical aside. In addition to P-views of finite paths, we must also introduce the notion of a P-view of an infinite path. This is a non-standard construction and the definition and related results are given below (with respect to a canonical traversal tree). On a first reading this definition may be safely skipped as we will make use of it only towards the very end of the chapter.

Definition 4.1.44. Let $c_{[\![R^0]\!]} = (c, P)$ be the canonical traversal tree induced by the computation tree $[\![R^0]\!]$. Let $\pi = \pi_1\pi_2\cdots$ be an *infinite* path in c. We define the ***infinite P-view*** of this path π. It is denoted by $c_{[\![R^0]\!]} \upharpoonright_\pi$.

In the following, we will define a subsequence of π (along with pointers) and this will constitute the nodes occurring in the P-view. The subsequence is given by $\pi_{i_1}\pi_{i_2}\cdots$: our aim is to define the sequence of indices $i_1, i_2, \cdots$. At each point if i_j is defined and we wish to define i_{j+1}, then if $\pi_{i_{j+1}}$ has a pointer to π_{i_k} for $k \leq j$ in $c_{[\![R^0]\!]}$, then this pointer persists in the P-view, otherwise if there is no such k then $\pi_{i_{j+1}}$ has no pointer.

1. We set $i_1 = 1$;
2. Given i_j, where $c(\pi_{i_j}) = z$ we define i_{j+1} by case analysis on $[\![R^0]\!](z)$:
   (a) $[\![R^0]\!](z) = f$, then $i_{j+1} = i_j + 1$.
   (b) $[\![R^0]\!](z) = \lambda\vec{v}$, then $i_{j+1} = i_j + 1$.
   (c) $[\![R^0]\!](z) = @$, then there are two possibilities:
      i. @ has infinitely many pointers pointing back to it. In this case, we set $i_{j+1} = i_j + 1$.
      ii. @ has a finite number of pointers pointing back to it. Suppose the one which is furthest away is given by a node π_k where $c(\pi_k) = z$ and $[\![R^0]\!](z) = \lambda\vec{v}$ (where $\vec{v}$) is possibly empty. Then, there are two possibilities:
         A. π_k is not pointed to by anything. In which case we set $i_{j+1} = k$.
         B. Otherwise, we set $i_{j+1} = i_j + 1$.
   (d) Finally, if $[\![R^0]\!](z) = v$ for v a variable, then we set $i_{j+1} = k$, where π_k is the nearest node in the path that points to π_j. If no such node exists[4] then we set $i_{j+1} = i_j + 1$.

Claim 4.1.45. *Let $\pi_{i_1}\pi_{i_2}\cdots$ be the infinite P-view of the the path $\pi_1\pi_2\cdots$ in the canonical traversal tree. Furthermore, let $\sigma = \epsilon \to \pi_{i_k}$, in other words, σ is the path in the canonical traversal tree from the root to π_{i_k}. Then:*

$$\ulcorner c_{[\![R^0]\!]} \upharpoonright_{\sigma} \urcorner = \pi_{i_1}\pi_{i_2}\cdots\pi_{i_k}$$

Proof. A straightforward induction on the length of the path from the root to π_{i_k}. □

4.1.3 Definition of the Simulating Automaton

Let φ be an MSO sentence that we wish to verify for a given level-2 recursion scheme R. As we know, there exists a non-deterministic tree automaton $N_\varphi = \langle Q, \Sigma, \delta, q_0, Acc \rangle$ where $Acc = Parity \cup Fin$ such that $N_\varphi \models t$ if and only if $t \models \varphi$.

From the previous section, we have shown that $N_\varphi \models [\![R]\!]$ if and only if there exists an annotated canonical traversal tree relative to N_φ that is accepting. Our decision problem is thus reduced to identifying whether or not there exists an accepting annotated canonical traversal tree relative to N_φ.

To achieve this we construct an alternating parity tree automaton, S_{N_φ} that works on $[\![R^0]\!]$, i.e. the computation tree. The automaton *simulates* the construction of the canonical traversal tree and at the same time guesses an appropriate annotation relative to N_φ. In particular, S_{N_φ} is constructed so that it has an accepting run over $[\![R^0]\!]$ if and only if there exists an annotated canonical traversal tree $c^{N_\varphi}_{[\![R^0]\!]}$ that is accepting. In the following we

[4]It is in fact easy to prove that there always exists a *unique* node in the path that points to π_j.

abbreviate $c^{N_\varphi}_{[\![R^0]\!]}$ to simply c'. To summarise:

$$\begin{array}{ll} & \varphi \models [\![R]\!] \\ \iff & \text{Definition of non-deterministic tree automaton} \\ & N_\varphi \text{ has an accepting run over } [\![R]\!] \\ \iff & \text{by Theorem 4.1.39} \\ & \exists c' \text{ such that } c' \text{ is an accepting canonical traversal tree relative to } N_\varphi \\ \iff & \text{by Theorem 4.1.68 (to be proved)} \\ & S_{N_\varphi} \text{ has an accepting run over } [\![R^0]\!] \end{array}$$

For simplicity, we will refer to S_{N_φ} as simply S_N for the remainder of this chapter. $S_N = \langle\, Q \times \mathsf{Env}, \Sigma^0, \delta', (q_0, \emptyset), Acc' \,\rangle$ and we describe it shortly. We begin, however, with an informal account of the behaviour of S_N.

Informal Explanation

We attempt to traverse the computation tree $[\![R^0]\!]$ with an alternating term tree automaton in such a way that we simulate the construction of the annotated canonical traversal tree. Recall the annotated canonical traversal tree is built in exactly the same way as the canonical traversal tree. However, in addition, it has another automaton, denoted N_φ, "piggy-backing" along and inspecting all the signature constants. In the following we abbreviate N_φ to N.

A state of S_N will consist of a pair (q_N, p) where q_N is the current state of N that we are simulating and p is another state responsible for directing the operation of the automaton. Intuitively, the second component of the state will guide our traversal over the computation tree in such a way that we simulate the construction of the canonical traversal tree. The first component only comes into play upon seeing signature constants.

Our first attempt will be the following: at each node of the computation tree we mimic the construction of the annotated canonical traversal tree given in Definition 4.1.35. For the most part, this is straightforward. If the current node is @ then we visit the leftmost child. If the current node is $\lambda\vec{v}$, then we visit the only child. If the current node is a signature constant, we send one automaton per child. However, if the current node is a variable, φ, our progression is less clear. There are two issues that must be dealt with here:

1. To evaluate φ we must visit another subtree of the computation tree which is *not* a child of the current node.
2. After we have evaluated φ, how do we return to inspect φ's children?

We consider as a concrete example the computation tree in Figure 4.2. Suppose we are an automaton at node [12] and we must evaluate φ. Then to address item (1) we must find some way of entering the subtree rooted at [3]. To address item (2) after exploring the subtree rooted at [3] we must have a way to handle the jump from the two exit points [14] and [15] to [16] and [17] respectively.

The first item can be handled easily if we take advantage of the ability of the 2-way alternating term tree automaton to spawn copies of the automaton "up" the tree. The regular nature of the computation tree is sufficient to guide this spawned copy to the correct subtree. So the first item is not an issue at all.

Unfortunately, the second item is more difficult. Any attempt of a "direct" approach and the reader will quickly find we are in wanting of a structure such as a stack to keep track of the locations of level-1 variables we have seen so far.

Thus, to get around this we will require a more sophisticated approach. For the greater part, we proceed as before for lambda nodes, @-nodes, and signature nodes. However, upon meeting a variable such as φ we perform the following:

1. First, we non-deterministically guess what the evaluation of φ will "look like." For example, we might say:

   "We will make a detour to evaluate φ and in doing there will be a line of computation where we will visit precisely the states $\{q_1, q_4, q_5\}$ (of the automaton N) and after seeing these states we will return here looking for the first child of φ and we will, at this point, be in state q_5. There will be another line of computation in our evaluation of φ, where we will visit states $\{q_1, q_3\}$ but will return here looking for the second child of φ and we will, at this point, be in state q_1."

   Thus, such a description will consist of a set of triples in $Q \times 2^Q \times [k]$, where the arity of φ is k. Thus there is clearly a bound on how large such a description can be. Effectively, we are describing how the evaluation of φ *interfaces* with φ's children.

   For example, the above verbose description can be represented as the set consisting of the two elements $(q_5, \{q_1, q_4, q_5\}, 1)$ and $(q_1, \{q_1, q_3\}, 2)$.

2. After guessing such a finite description we now perform two actions:

   (a) We consider each triple in the description in turn. We take $(q_1, \{q_1, q_3\}, 2)$ from the above example. For this tuple we send one copy of the automaton in the direction of the first child of φ in state q_1 and *echoing* the states $\{q_1, q_3\}$. In other words, we "pretend" we have returned from the detour by mimicking the states seen (q_1 and q_3) and switching to the appropriate state (q_1). We spawn one such copy of the automaton in the appropriate direction for each item in the description.

   (b) Next, we spawn one single automaton upwards towards the subtree that should be substituted for φ. Along with the automaton we encapsulate the above description (by encoding it as part of the state). The automaton will enter the correct subtree and in visiting this subtree it will actively *verify* that this description is not violated. In particular, using the example of the subtree rooted at [3], when we reach an exit point such as [15], we know this signals a return to the 2nd child of φ – we check that we have seen precisely the states prescribed and that we are in the correct simulating state. If this is true, then were merely stop in an accepting state safe in the knowledge that what happens next is already being verified on a parallel line of computation. If, however, it is false, then we abort.

The idea is somewhat more complex than this, but the above informal explanation captures the essence of S_N: the two main behaviours of S_N consist of (1) guessing and (2) verifying.

In actual fact, we *pre-empt* the guessing. For example, in the sketch above we performed all our guesses *upon* visiting a variable node. Instead, we will do this at the @-nodes[5]. At such an @-node we will guess exactly which variables of the abstraction will be visited and the appropriate detours that we will take for each one.

[5]As these always indicate a β-redex, the leftmost child being a λ-abstraction and the remaining children being the operands.

Overview of the Set of States

A state of S_N is given by a pair (q, ρ) where q is the current state we are simulating from N_φ and ρ is the environment. In particular, if $[\![R^0]\!]$ is a computation tree, and we have (q, ρ) sitting on a node $z \in Dom([\![R^0]\!])$, then ρ gives a semantics to the free variables in the subtree rooted at z. In particular, we have that $\rho \in \mathsf{Env}$, where Env is given by:

$$\begin{aligned} \mathsf{Env} &= 2^{(Q \times V \times 2^Q \times \mathsf{IC})} \\ \mathsf{IC} &= 2^{(Q \times 2^Q \times [m])} \end{aligned}$$

where m is determined by the underlying grammar. The set IC is referred to as the set of ***interfacing configurations***. For example, suppose that we have state (q, ρ) with the following:

$$\rho = \left\{ \begin{array}{l} (q_3, z, \{q_1, q_2, q_3\}, \emptyset), \\ (q_3, \varphi, \{q_3\}, \{(q_2, \{q_1, q_2\}, 1), (q_1, \{q_1, q_3\}, 1)\}), \\ (q_1, \varphi, \{q_1, q_3\}, \emptyset) \end{array} \right\}$$

then this means that we will visit an occurrence of the free variable z (at least once) in state q_3 after having seen the states $\{q_1, q_2, q_3\}$ from the current node (inclusive).

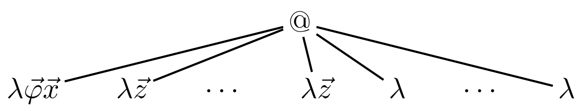
Similarly, we will visit an occurrence of the free variable φ in state q_3 after having seen precisely the states $\{q_3\}$ from the current node (inclusive). It also says that when we are done "expanding" φ, we will come back (at least once) asking for the first argument in state q_2 after having seen states $\{q_1, q_2\}$ in the expansion. Similarly, we will come back (at least once) asking for the first argument but this time in state q_1 after having seen states $\{q_1, q_3\}$.

Thus, as one might be able to deduce, the interfacing configuration details what happens when φ is expanded. More specifically, however, it details whether we ever come back from the expansion and what was seen during the expansion. In the event that we do not return from the expansion then the fourth component of the tuple will be the empty set.

The Rules; δ'

In the following let us assume that the current state is (q, ρ) and that we are sitting on a node $z \in Dom([\![R^0]\!])$. We perform a case analysis on $[\![R^0]\!](z)$. The initial configuration is $(q_0, \emptyset)$.

1. $[\![R^0]\!](z) = @$. The first two levels of the subtree rooted at $[\![R^0]\!](z)$ are depicted below:



   (a) First, for the jth child of @ (for $j > 0$), we guess how it will be accessed through calls to v_j from inside the subtree rooted at 0, where v_j is the jth variable in the vector $\vec{\varphi}\vec{x}$. Suppose we guess it will be accessed in the following ways:

   $$\{(q_{j,k}, R_{j,k}, c_{j,k}, \rho_{j,k}) : k \in [K_j]\}$$

   Note that one tuple $(q_{j,k}, R_{j,k}, c_{j,k}, \rho_{j,k})$ in the above set does not necessarily indicate that we enter the jth child precisely once in this fashion; but *at least* once.

An element $(q_{j,k}, R_{j,k}, c_{j,k}, \rho_{j,k})$ communicates the following. It says that we will call variable v_j from the body of the abstraction rooted at the 0th child of @, in state $q_{j,k}$ such that $R_{j,k}$ is the set of states (of the automaton N_φ) that we will have seen from this moment (inclusive) to the moment we visit this occurrence (inclusive). It also says that on expanding the variable, the appropriate environment to use is $\rho_{j,k}$ with interfacing configuration $c_{j,k}$.

(b) We then need to ensure that our guesses are consistent with the current environment ρ. In particular, we need to check the following:

$$\bigcup_{1\le j\le n, 1\le k\le K_j} \rho_{j,k}^{R_{j,k}} \supseteq \rho$$

where given an environment σ and given a set of states R, the environment σ^R is defined as follows:

$$(q, v, P, c) \in \sigma \leftrightarrow (q, v, P \cup R, c) \in \sigma^R$$

If such a consistency check fails, then the automaton must abort.

(c) We are now ready to launch the automata.

- For $1 \le j \le n$ and for each $1 \le k \le K_j$ we send an automaton in direction j in state $(q_{j,k}, \rho_{j,k})$ first ***echoing*** states $R_{j,k}$.
- We send an automaton in direction 0 (i.e. the body of the abstraction) in state (q, ρ') where:
$$\rho' := \bigcup_{1\le j\le n, 1\le k\le K_j} \{(q_{j,k}, v_j, R_{j,k}, c_{j,k})\}$$

2. $[\![R^0]\!](z) = f$ where $\varrho(f) = n$. Suppose that $(q_1, \cdots, q_n) \in \delta(q, f)$, then we guess, for each $1 \le i \le n$ a new environment ρ_i, such that the following holds:

$$\bigcup_i \rho_i^{\{q\}} = \rho$$

So, for each i, we spawn an automaton in direction i in state (q_i, ρ_i).

In the case where $\varrho(f) = 0$, if $\rho \ne \emptyset$, then the above equation is unsatisfiable and we must abort.

3. $[\![R^0]\!](z) = v$ and $v : o^n \to o$. Then we need to check that $(q, v, \{q\}, c) \in \rho$ for some c. If such a tuple does not exist, then we abort – i.e. we have guessed incorrectly. However, if such a tuple does exist, we fix one and for each $(p, k, R) \in c$ we guess a family of environments:
$$F_{(p,k,R)} := \{\rho_{i,(p,k,R)} : i \in I_{(p,k,R)}\}$$

Finally we check that one of the following holds:

$$\bigcup_{(p,k,R)\in c, i\leq I_{p,k,R}} \rho^R_{i,(p,k,R)} = \rho - \{(q,v,\{q\},c)\}$$

or

$$\bigcup_{(p,k,R)\in c, i\leq I_{p,k,R}} \rho^R_{i,(p,k,R)} = \rho$$

Note that in the case where $c = \emptyset$ and $|\rho| > 1$, neither of the above equations will ever be satisfied so we must abort.

Thus, for each $(p,k,R) \in c$, we spawn an automaton for each $i \in I_{p,k,R}$ in direction k in state $(p, \rho_{i,(p,k,R)})$ by first echoing the states R.

4. $[\![R^0]\!](z) = \lambda\vec{w}$. Then we spawn one automaton in direction 1 with the same state, i.e. (q,ρ).

Remark 4.1.46. Note that for the case where $[\![R^0]\!](z) = @$, we are making an implicit assumption. Namely, that we know what the jth child of @ "looks like"; more specifically, that if the jth child has label $\lambda w_1 \cdots w_n$; then we know that the variables bound to this λ-node are $w_1, \cdots, w_n$. Note that this can easily be achieved by decorating each @ with a finite and bounded amount of information.

Acceptance Condition for the Simulating Automaton

Let $d : Q \to \omega$ be the colouring function which gives us the parity condition for the automaton N. Then, the colouring function d' for the automaton S_N is given by:

$$d'((q,\rho)) = d(q)$$

The acceptance condition for leaf nodes is implicit in the definition of δ'.

However, our definition of the automaton S_{N_φ} only works if we assume that every infinite path in the canonical traversal tree has an infinite number of signature constants. What about the case where this does not hold – i.e., in the event of an infinite reduction sequence with no terminal output after some point? As we know this results in a terminal node $\perp$ in the value tree.

Suppose that (using the transition rules described above) we are at a node $z \in Dom([\![R^0]\!])$ with label f in state (q,ρ) and suppose that we spawn an automaton towards the ith child in state (q_i,ρ_i). Instead of this, we can perform the following sequence of actions:

1. From state (q,ρ) we remain at the same node and transition into state (q',ρ) – note the primed state.

2. From (q',ρ) we then spawn an automaton towards the ith child in state (q_i,ρ_i) as before. Note the unprimed states.

Thus we use the brief transition into a primed state to signal a visit to a signature constant. The parity condition can thus be adapted to only consider infinite sequences of ***primed*** states. If we have an infinite sequence of unprimed states, then this is treated as a $\perp$.

4.1.4 Direction 1: From Annotated Canonical Traversal Trees to Run Trees

Environments and Interfacing Configurations

The proof of the correspondence between an accepting run of S_N over $[\![R^0]\!]$ and an accepting annotated canonical traversal tree relative to N requires a detailed analysis of annotated canonical traversal trees. In the following let $c'_{[\![R^0]\!]} = (c : T \longrightarrow Q \times Dom([\![R^0]\!]), P)$ be an annotated canonical traversal tree with respect to N, where Q is the set of states for N.

Remark 4.1.47. Recall that given a finite path π (from the root) of $c'_{[\![R^0]\!]}$, $\ulcorner \pi \urcorner$ denotes the P-view of this path. Given any node $\alpha \in T$, then α defines the unique path from the root of c to the node α. Thus, we will often write, by an abuse of notation, $\ulcorner \alpha \urcorner$ to mean $\ulcorner \epsilon \to \alpha \urcorner$.

Definition 4.1.48. Let α in T, we define the colour of α, denoted by $col(\alpha)$ to be:

$$col(\alpha) = \pi_1 \circ c(\alpha)$$

Let π be a path in T (not necessarily from the root), we define the colour of π, denoted by $col(\pi)$ be be:

$$col(\pi) = \{col(\pi_i) : \pi = \pi_1 \pi_2 \cdots \}$$

Note that the latter is the *set*[6] of states seen along this path.

Definition 4.1.49. Let $\alpha \in T$. The set of ***interfacing nodes*** for α is denoted by the following set of nodes:

$$\mathsf{ic}(\alpha) = \{\beta \in T : \ \beta \text{ has a pointer to } \alpha\}$$

For each node β in $\mathsf{ic}(\alpha)$ we associate a finite description $\mathsf{i}^\alpha(\beta)$, where:

$$\mathsf{i}^\alpha(\beta) = (col(\beta), col(\alpha \to \beta), k)$$

where $(\beta, k, \alpha) \in P$.

Definition 4.1.50. The ***interfacing configuration*** for α is given by the following set of tuples

$$\overline{\mathsf{ic}(\alpha)} = \{\mathsf{i}^\alpha(\beta) : \beta \in \mathsf{ic}(\alpha)\}$$

Definition 4.1.51. The set of ***environment nodes*** for $\alpha \in T$ is denoted by the following set of nodes:

$$\mathsf{env}(\alpha) = \{\beta \in T : \alpha \in \ulcorner \beta \urcorner \text{ and } \beta \text{ is a variable with pointer to a node in } \ulcorner \alpha \urcorner.\}$$

For each β in $\mathsf{env}(\alpha)$ we associate a finite description β, given by $\mathsf{e}^\alpha(\beta)$ where:

$$\mathsf{e}^\alpha(\beta) = (col(\beta), [\![R^0]\!](\pi_2 \circ c(\beta)), col(\alpha \to \beta), \overline{\mathsf{ic}(\beta 1)})$$

Definition 4.1.52. Given α, the ***environment*** of α is given by the following set of tuples:

$$\overline{\mathsf{env}(\alpha)} = \{\mathsf{e}^\alpha(\beta) : \beta \in \mathsf{env}(\alpha)\}$$

Remark 4.1.53. In both the definitions of interfacing nodes and environment nodes, the sets are potentially infinite. However, the corresponding sets $\overline{\mathsf{ic}(\cdot)}$ and $\overline{\mathsf{env}(\cdot)}$ are clearly finite. The barred sets therefore serve as finite descriptions of these sets.

[6]It is *not* a sequence!

Lemma 4.1.54. *Let $[\![R^0]\!](\pi_2 \circ c(\alpha)) = \varphi$ where φ is a level-1 variable. The following holds:*

$$\mathsf{env}(\alpha) = \{\alpha\} \cup \bigcup_{\beta \in \mathsf{ic}(\alpha 1)} \mathsf{env}(\beta 1)$$

Proof. From the construction of the canonical traversal tree, we must have the following situation:

$$\cdots\ n\ \lambda\vec{\varphi}\vec{x}\ \cdots\ \underbrace{\varphi}_{\alpha}\ \underbrace{\lambda\vec{z}}_{\alpha 1}$$

for some non-empty $\vec{z}$. We consider a node $\beta \in \mathsf{ic}(\alpha 1)$ giving us:

$$\cdots\ n\ \lambda\vec{\varphi}\vec{x}\ \cdots\ \underbrace{\varphi}_{\alpha}\ \underbrace{\lambda\vec{z}}_{\alpha 1}\ \cdots\ \underbrace{z_i}_{\beta}\ \underbrace{\lambda}_{\beta 1}$$

From the construction, we must have $\beta 1$ is a lambda node that is k-enabled by α as indicated above. The P-view at $\beta 1$ must be $\cdots \alpha\ \beta 1$. Thus, if $\delta \in \mathsf{env}(\beta 1)$ then as $\beta 1$ is in its P-view α must be in it as well. This shows the the right hand side is included in the left hand side.

To so the converse, we consider any node $\delta \neq \alpha$[7], where $\delta \in \mathsf{env}(\alpha)$, we must show there exists some $\beta \in \mathsf{ic}(\alpha)$ where $\delta \in \mathsf{env}(\beta 1)$. By assumption (and the fact that the P-view must be a path in $[\![R^0]\!]$):

$$\ulcorner \delta \urcorner = \cdots\ \varphi\ \lambda\ \cdots\ \delta$$

where δ has a pointer to some node occurrence in the $\cdots$ to the left of φ.

By the construction of the canonical traversal tree, the above can only arise in the case when the path from the root of c to δ has the following structure:

$$\cdots\ \varphi\ \lambda\vec{u}\ \cdots\ u_i\ \lambda\ \cdots\ \delta$$

where λ is in the P-view of δ, as required. ☐

Lemma 4.1.55. *Let $[\![R^0]\!](\pi_2 \circ c(\alpha)) = @$, then:*

$$\mathsf{env}(\alpha) \subseteq \bigcup_{\beta \in \mathsf{env}(\alpha 0)} \mathsf{env}(\beta 1)$$

Proof. Suppose that $\delta \in \mathsf{env}(\alpha)$ and has label w, where w must be a variable. By definition of $\mathsf{env}(\alpha)$ and the fact that the P-view always results in a path in $[\![R^0]\!]$ the following must hold:

$$\ulcorner \delta \urcorner = \cdots \underbrace{@}_{\alpha}\ \lambda\vec{v}\ \cdots\ \underbrace{w}_{\delta}$$

where $\vec{v}$ consists of either level-0 variables or is empty. It cannot contain both level-1 and level-0 variables as this would indicate that δ is not bound by a predecessor of α.
Analysing the construction of the canonical traversal tree, the above case can only happen if

[7]In the case where $\delta = \alpha$ the analysis is trivial.

the path from ϵ to δ in $c'_{[\![R^0]\!]}$ takes the following form:

$$\begin{tikzcd}[column sep=small] \cdots & \underbrace{@}_{\alpha} & \underbrace{\lambda\vec{\varphi}\vec{x}}_{\alpha 0} & \cdots & \underbrace{u}_{\beta} \arrow[ll, bend right] & \underbrace{\lambda\vec{v}}_{\beta 1} \arrow[llll, bend right] & \cdots & \underbrace{w}_{\delta} \end{tikzcd}$$

for some $u \in \vec{\varphi}\vec{x}$. By definition β is in $\mathsf{env}(\alpha 0)$, and the result holds. □

Recall that for the set $\mathsf{ic}(\alpha)$, we can associate with any node β in $\mathsf{ic}(()\alpha)$ a finite description $\mathsf{i}^\alpha(\beta)$. Similarly, we can associate with any node β in $\mathsf{env}(\alpha)$ a finite description $\mathsf{e}^\alpha(\beta)$.

We define the following equivalence relations between nodes. Let β and β' be elements of $\mathsf{ic}(\alpha)$, then we write:

$$\begin{array}{lcl} \beta \sim_i^\alpha \beta' & \leftrightarrow & \mathsf{i}^\alpha(\beta) = \mathsf{i}^\alpha(\beta') \\ \beta \approx_i^\alpha \beta' & \leftrightarrow & \beta \sim_i^\alpha \beta' \text{ and } \overline{\mathsf{env}(\beta 1)} = \overline{\mathsf{env}(\beta' 1)} \end{array}$$

Let β and β' be elements of $\mathsf{env}(\alpha)$, then we write:

$$\begin{array}{lcl} \beta \sim_e^\alpha \beta' & \leftrightarrow & \mathsf{e}^\alpha(\beta) = \mathsf{e}^\alpha(\beta') \\ \beta \approx_e^\alpha \beta' & \leftrightarrow & \beta \sim_e^\alpha \beta' \text{ and } \overline{\mathsf{env}(\beta 1)} = \overline{\mathsf{env}(\beta' 1)} \end{array}$$

Given an equivalence relation $\sim$ and a set S, we will write $S/_\sim$ for the set of equivalence classes. Also if $\beta \in S$, we write $[\beta]_\sim$ for the equivalence class of β.

Remark 4.1.56. Note that $\mathsf{env}(\alpha)/_{\sim_e^\alpha}$ is isomorphic to $\overline{\mathsf{env}(\alpha)}$. Similarly, $\mathsf{ic}(\alpha)/_{\sim_i^\alpha}$ is isomorphic to $\overline{\mathsf{ic}(\alpha)}$.

Remark 4.1.57. Often, we will say "... let $\beta \in S/_\sim$" where $\sim$ is an equivalence relation. We interpret this to mean that β is the *representative member* of an equivalence class in S under the equivalence $\sim$. In the case where $[\beta]_\sim$ has more than one member, we use the axiom of choice to select a representative member. Hence the number of elements in $S/_\sim$ is equal to the number of equivalence classes.

The Extraction

Given an accepting canonical traversal tree $c'_{[\![R^0]\!]} = (c : T \longrightarrow Q \times Dom([\![R^0]\!]), P)$ relative to non-deterministic tree automaton N, we will construct a tree

$$r : T_r \longrightarrow T \times Q \times \mathsf{Env} \times Dom([\![R^0]\!])$$

where T_r is defined inductively below and r has the property that if we project it onto its second, third and fourth components (viewing the second and third as a pair) then the result is an accepting run tree of S_N over $[\![R^0]\!]$.

Intuitively, r is thus nothing more than a run tree of S_N over $[\![R^0]\!]$ with some additional annotations. We will build r by examining nodes in $c'_{[\![R^0]\!]}$ and the first component of a node in r reflects the node in T that we have used as a reference.

Remark 4.1.58. In particular, if $r(a) = (\alpha, q, \rho, z)$, then this will mean that the state (q, ρ) of this particular *copy* of the automaton on the node z is *derived* from the node α in T. We will see that it will be the case that if $r(a) = (\alpha, q, \rho, z)$ then $c(\alpha) = (q, z)$ and furthermore $\rho = \overline{\mathsf{env}(\alpha)}$.

1. We set $r(\epsilon) = (\epsilon, q_0, \emptyset, \epsilon)$.
2. If $r(a)$ is defined and $r(a) = (\alpha, q, \rho, z)$, we perform a case analysis on $[\![R^0]\!](z)$.
   (a) $[\![R^0]\!](z) \in \Sigma^o \cup V^o$. Stop.
   (b) $[\![R^0]\!](z) \in \Sigma^{o^n \to o}$ for $n \geq 1$. We set
   $$r(ai) = (\alpha i, q_i, \rho_i, z_i)$$
   where $c(\alpha i) = (q_i, z_i)$ and $\overline{env(\alpha i)} = \rho_i$.
   (c) $[\![R^0]\!](z) \in V^{o^n \to o}$ for $n \geq 1$. Consider $\alpha 1$ (the expansion of the variable) and in particular $\mathsf{ic}(\alpha 1)/_{\approx_i^{\alpha 1}}$. For each equivalence class $[\beta]_{\approx_i^{\alpha 1}}$ we create a child:
   $$r(a) \xrightarrow{R} (\beta 1, \underbrace{\pi_1 \circ c(\beta 1)}_{\text{state}}, \overline{\mathsf{env}(\beta 1)}, \underbrace{\pi_2 \circ c(\beta 1)}_{\text{node in } [\![R^0]\!]})$$
   where $R = col(\alpha \to \beta 1)$.
   (d) $[\![R^0]\!](z) = @$. Consider $\mathsf{env}(\alpha 0)/_{\approx_e^{\alpha 0}}$; for each equivalence class $[\beta]/_{\approx_e^{\alpha 0}}$ we create a child:
   $$r(a) \xrightarrow{R} (\beta 1, \pi_1 \circ c(\beta 1), \overline{\mathsf{env}(\beta 1)}, \pi_2 \circ c(\beta 1)).$$
   where $R = col(\alpha \to \beta 1)$. Also, we send one automaton:
   $$r(a) \longrightarrow (\alpha 0, q, \overline{\mathsf{env}(\alpha 0)}, \pi_2 \circ c(\alpha 0))$$
   (e) Finally, in the case of lambda nodes, we send one automaton in the direction of the only child as follows:
   $$r(a) \longrightarrow r(a1) = (\alpha 1, q, \overline{\mathsf{env}(\alpha 1)}, \pi_2 \circ c(\alpha 1))$$

Proof of Correctness

Let r be the run tree extracted from $c'_{[\![R^0]\!]}$ as defined above. In the following we will continue to take P-views of paths in $c'_{[\![R^0]\!]}$, however, we deliberately choose not to represent pointer information to avoid cluttering.

Lemma 4.1.59. *Suppose $r(a) = (\alpha, q, \rho, z)$, then $c(\alpha) = (q, z)$ and $\rho = \overline{\mathsf{env}(\alpha)}$.*

Proof. By construction. □

Lemma 4.1.60. *Let $\sigma_1 \cdots \sigma_n$ be a prefix of a path in r. If $r(\sigma_n) = (\alpha, q, \rho, z)$, then $\ulcorner \alpha \urcorner = \pi_1 \circ r(\sigma_1) \cdots \pi_1 \circ r(\sigma_n)$.*

Proof. This follows easily by induction if we can show that for any node $b = (\alpha, q, \rho, z)$ in r, then its child $b' = (\alpha', q', \rho', z')$ has the property that $\ulcorner \alpha' \urcorner = \cdots \alpha\ \alpha'$.

This is a straightforward induction on the length of the path with a case analysis on the last rule used; therefore we only illustrate one case for expository purposes. Suppose that $b = (\alpha, q, \rho, z)$ where $[\![R^0]\!](z) = @$. From the construction of r defined above, there are two possibilities for the child b':

1. $r(b') = (\alpha 0, q, \overline{\mathsf{env}(\alpha 0)}, \pi_2 \circ c_t(\alpha 0))$. The result holds trivially from the game semantics and definition of the canonical traversal tree.

2. $r(b') = (\beta 1, \pi_1 \circ c'_{[\![R^0]\!]}(\beta 1), \overline{\mathsf{env}(\beta 1)}, \pi_2 \circ c'_{[\![R^0]\!]}(\beta 1))$ where β is a node in $\mathsf{env}(\alpha 0)$; in particular we thus have:

$$\begin{tikzcd}
@ & \lambda\vec{\varphi}\vec{x} & \cdots & v \arrow[ll, bend right] & \lambda\vec{y} \arrow[llll, bend right] \\
\alpha & \alpha 0 & \cdots & \beta & \beta 1
\end{tikzcd}$$

   where $v \in \vec{\varphi}\vec{x}$ and $\vec{y}$ is a possibly empty set of level-0 variables. It follows from the construction of the canonical traversal tree that $\beta 1$ has a pointer to α and the rest now follows from the game semantics.

The cases for variables, signature constants and lambda nodes may be argued similarly. □

Lemma 4.1.61. *If $(\alpha, q, \rho, z) \xrightarrow{R} (\alpha', q', \rho', z')$ then there exists a path from α to α' in c'_t. Furthermore, $R \cup \{q\} \cup \{q'\} = col(\alpha \to \alpha')$.*

Proof. Again, this follows by construction. □

Lemma 4.1.62. *r, under suitable projections, is a run-tree of the automaton S_N over the tree $[\![R^0]\!]$.*

Proof. We need to verify that that our extraction complies with the transition function of the automaton. For each point in the run tree, we consider whether the next step complies with the transition function. The cases for signature constants, lambda nodes and leaf nodes are straightforward to show. For the cases of level-1 variables and @ nodes we make use of Lemma 4.1.54 and Lemma 4.1.55 respectively. □

Proposition 4.1.63. *Let $c'_{[\![R^0]\!]}$ be an accepting canonical traversal. Then r (under suitable projections) is an accepting run tree for S_N over $[\![R^0]\!]$.*

Proof. From the above, we know that r is a run tree, we must now verify that it is accepting. We must check that the acceptance condition is met for both finite and infinite paths.

1. Let $a_1 a_2 a_3 \cdots$ be an infinite path. The result follows from Lemma 4.1.61 and the fact that $c'_{[\![R^0]\!]}$ is accepting.

2. Let $a_1 a_2 \cdots a_n$ be a finite terminal path, we must verify that $r(a_n) = (\alpha, q, \rho, z)$ corresponds to an accepting leaf node. First we note, by Lemma 4.1.59, that $\rho = \overline{\mathsf{env}(\alpha)}$. Furthermore, by construction we have that $c_{[\![R^0]\!]}(\alpha) = (q, z)$ and hence z is a variable of level 0 or a signature constant of level 0. It follows therefore (respectively) that either $\mathsf{env}(\alpha) = \{\alpha\}$ and $\overline{\mathsf{env}(\alpha)} = \emptyset$. In the case of a variable, $(q, \overline{\mathsf{env}(\alpha)})$ satisfies the acceptance condition by definition of the simulating automaton. In the case of a signature constant, the result follows immediately.

□

Theorem 4.1.64. *If there exists an accepting annotated canonical traversal tree over the computation tree $[\![R^0]\!]$ relative to non-deterministic parity tree automaton N, then there exists an accepting run of S_N over $[\![R^0]\!]$.*

4.1.5 Direction 2: From Run Trees to Annotated Canonical Traversal Trees

Extracting an annotated canonical traversal tree from a run-tree r

Given the canonical traversal tree $c_{[\![R^0]\!]} = (c : T \longrightarrow Dom([\![R^0]\!]), P)$, we want to annotate it with information we have derived from r, where r is a run-tree of the simulating automaton S_N over $[\![R^0]\!]$.

Recall that each node of the run tree is represented as a pair (q, z) and can be thought to represent a copy of the automaton in state q sitting on node $z \in Dom([\![R^0]\!])$. However, for our purposes, we will express $r : T_r \to Q \times \mathsf{Env} \times Dom([\![R^0]\!])$, as each state of S_N is represented as a pair $Q \times \mathsf{Env}$.

We construct a new tree

$$c^r : T \longrightarrow Dom(r) \times \underbrace{Q \times \mathsf{Env}}_{\text{state of } S_N} \times Dom([\![R^0]\!])$$

where c^r has the same domain as c, but its nodes have more elaborates labels. However, we will see that $\pi_4 \circ c^r(\alpha) = c(\alpha)$.

Note that analogous to our construction of a run tree from a canonical traversal tree, we have an extra component (the first one) indicating which node in r has provided us with information to aid us in annotating our tree.

1. Set $c^r(\epsilon) = (\epsilon, \pi_1 \circ r(\epsilon), \pi_2 \circ r(\epsilon), \pi_3 \circ r(\epsilon))$.

2. Now for the inductive step. Suppose we have the following:

$$c^r(\alpha) = (a, q, \rho, z)$$

   and $r(a) = (q, \rho, z)$. We perform a case analysis on $[\![R^0]\!](z)$.

   (a) $[\![R^0]\!](z) = f$ for $f \in \Sigma$. Then we simply set $c^r(\alpha i) = (ai, \pi_1 \circ r(ai), \pi_2 \circ r(ai), \pi_3 \circ r(ai))$ for $1 \leq i \leq \mathsf{ar}(f)$.

   (b) $[\![R^0]\!](z) = @$. We set $c^r(\alpha 0) = (a0, \pi_1 \circ r(a0), \pi_2 \circ r(a0), \pi_3 \circ r(a0))$

   (c) $[\![R^0]\!](z) = \lambda\vec{v}$ ($\vec{v}$ is possibly empty). Set $c^r(\alpha 1) = (a1, \pi_1 \circ r(a1), \pi_2 \circ r(a1), \pi_3 \circ r(a1))$.

   (d) $[\![R^0]\!](z) = v$ for v a variable. There are two possibilities:

      i. In the case of a level-1 variable we have the following scenario for α in $c_{[\![R^0]\!]}$:

$$\begin{tikzcd}[column sep=small] \underbrace{@}_{\beta} & \underbrace{\lambda\vec{\varphi}\vec{x}}_{\beta 1} & \cdots & \underbrace{\varphi_i}_{\alpha} \arrow[ll, bend right] & \underbrace{\lambda\vec{w}}_{\alpha 1} \arrow[llll, bend right] \end{tikzcd}$$

      Furthermore, suppose that $c^r(\beta) = (b, q_b, \rho_b, z_b)$ where $r(b) = (q_b, \rho_b, z_b)$. We therefore set:

$$c^r(\alpha 1) = (bl, q, \rho', z_b i)$$

      where $l \in \omega$ and $b \xrightarrow{R} bl$ in r with $r(bl) = (q, \rho', z_b i)$ for some i such that the following hold:

- R is the set of states[8] seen along the path $\underbrace{\lambda\vec{\varphi}\vec{x}}_{\beta_1}\cdots\underbrace{\varphi_i}_{\alpha}$;
- Furthermore,
$$\rho' - \{(q_v, v, Q_v, c_v) : v \notin \vec{w}\} = S$$
where
$$S := \{(p, w_i, Q, \emptyset) : r(a) \xrightarrow{Q} r(ak) \text{ where } r(ak) = (p, Q, zi)\}$$

In other words, the environment of $r(bl)$ must satisfy the condition that the variable profiles pertaining to $\vec{w}$ match up with how we exit φ_i in the run-tree. In other words, interfaces match up.
Note that there may be several l's that satisfy the above criteria. In such cases, we use the ***axiom of choice***.

ii. In the case of a level-0 variable we have the following scenario for α:

$$\begin{tikzcd}
\underbrace{n}_{\beta} & \underbrace{\lambda\vec{\varphi}\vec{x}}_{\beta 1} & \cdots & \underbrace{x_i}_{\alpha} \arrow[ll, bend right] & \underbrace{\lambda}_{\alpha 1} \arrow[llll, bend right]
\end{tikzcd}$$

where $\vec{\varphi}$ is possibly empty. Furthermore, suppose that $c^r(\beta) = b$ where $r(b) = (q_b, \rho_b, z_b)$. We therefore set:

$$c^r(\alpha 1) = (bl, q, \rho', z_b i)$$

where $l \in \omega$ and $b \xrightarrow{R} bl$ in r where $r(bl) = (q, \rho', z_b i)$ such that R is the set of states seen from $\underbrace{\lambda\vec{\varphi}\vec{x}}_{\beta_1}\cdots\underbrace{\varphi_i}_{\alpha}$.
Note that there may be several l's that satisfy the above criteria. In such cases, we use the ***axiom of choice***.

Proof of correctness

Recall that we started with the canonical traversal $c_{[\![R^0]\!]} = (c, P)$, then with the aid of an accepting run tree r created a new tree c^r. We now show that the tree with pointers:

$$c^r_{[\![R^0]\!]} = (c^r, P)$$

(under a suitable projection) is an accepting annotated canonical traversal tree. As with the proof of correctness for the other direction, we will refrain from explicitly representing pointers.

Lemma 4.1.65. *Let $\alpha \in c^r$ and consider its P-view $\alpha_1\alpha_2\cdots\alpha_n = \ulcorner\alpha\urcorner$. Then for each $i = 1\cdots n-1$, we have the following:*

1. $\pi_1 \circ c^r(\alpha_i) \xrightarrow{R} \pi_1 \circ c^r(\alpha_{i+1})$ in r for some R;

2. The set:
$$\{\pi_2 \circ c^r(\beta) : \beta \textit{ is in the path from } \alpha_i \textit{ to } \alpha_{i+1} \textit{ in } c^r\}$$

[8]Of the automaton N.

is equal to $R \cup \{\pi_2 \circ c^r(\alpha_i)\} \cup \{\pi_2 \circ c^r(\alpha_{i+1})\}$.

Proof. The above proof amounts to saying that for any node $\alpha \in c^r$, if we take the P-view at this point and consider it predecessor in the P-view, say, α' then we will find the following desirable property: $c^r(\alpha) = (a, q, \rho, z)$ and $c^r(\alpha') = (a', q', \rho', z')$ are such that a' is the direct predecessor of a in the run tree, and furthermore, all the states echoed from a to a' are precisely those states seen from α' to α in c^r.

This follows by a tedious but very straightforward induction over the rules in the construction of $c^r_{[\![R^0]\!]}$. □

Corollary 4.1.66. *A corollary of the above is that for a given infinite path in* $c^r_{[\![R^0]\!]}$*, its infinite P-view is an infinite path in* r *satisfying properties (1) and (2) of Lemma 4.1.65.*

Theorem 4.1.67. *If* r *is accepting then the projection of the second, third and fourth components of* $c^r_{[\![R^0]\!]}$ *is an accepting canonical traversal tree.*

4.1.6 Putting it all together

Theorem 4.1.68. *Combining Theorems 4.1.64 and 4.1.67 we see that* $S_N \models [\![R^0]\!]$ *if and only there exists an accepting canonical traversal tree over* $[\![R^0]\!]$ *relative to* N.

Finally, all that remains to be shown is that deciding $S_N \models [\![R^0]\!]$ is decidable. This follows from the fact that $[\![R^0]\!]$ is defined as a level-0 recursion scheme and hence gives rise to a regular tree. The acceptance problem for regular trees by alternating parity tree automata is decidable [EJ91].

4.2 Related Work and Concluding Remarks

In this chapter we have shown that given any level-2 recursion scheme (potentially unsafe), the term tree it produces possesses a decidable monadic second-order theory. This result was a first step towards answering Knapik et al.'s question of whether their result for safe term trees could be extended to the unsafe case.

Shortly after the publication of the result contained in this chapter, Knapik et al. [KNUW05] provided an independent proof of the result [KA04]. We discuss their proof methodology below. In addition to an alternative proof for the level-2 case, they also provided something we did not – namely a complexity analysis. They succeeded in showing that the model-checking problem for such a term tree with respect to a μ-calculus formula is 2EXPTIME-complete. Note that the μ-calculus is equi-expressive with monadic second-order logic over trees [JW96].

4.2.1 An Independent Proof of the Level-2 Result

Knapik et al. reduce the the problem of deciding whether a level-2 recursion scheme R satisfies an MSO property φ into a suitable parity game over 2-trees. The result then follows from the decidability of the winner of such games [Wal00]. The reduction of the former problem to the latter proceeds by a sequence of transformations and we briefly describe these below.

We recall from the preceding chapter the notion of a panic automaton introduced by Knapik et al. [KNUW05] as a machine characterisation of potentially unsafe level-2 recursion schemes. Thus, for a given recursion scheme R there exists a panic automaton A that accepts precisely $[\![R]\!]$. We fix R and the corresponding panic automaton A for the remainder of this section. Associated with A is $Tr(A)$, the computation tree of A – distinct from the computation tree defined in this chapter. Intuitively, $Tr(A)$ is nothing more than the unique run tree detailing each move of the panic automaton; it has the property that the collapse of the tree to Σ (as defined in Chapter 2) gives us $[\![R]\!]$. This means that $[\![R]\!]$ is MSO-definable in $Tr(A)$ (this should be straightforward to see). The problem can therefore be reduced to MSO decidability of $Tr(A)$. In fact, instead of considering $Tr(A)$ one considers $\hat{Tr}(A)$: each node in $Tr(A)$ is either a signature constant or is of the form (q, s) where s is 2-store; $\hat{Tr}(A)$ projects each node (q, s) onto $(q, \mathsf{top}_1(s))$. Note that, of course, the collapsed tree of $\hat{Tr}(A)$ is still $[\![R]\!]$. Knapik et al. also make use of the fact that a *given* MSO property φ can be characterised by a suitable non-deterministic parity tree automaton B. Let φ be a property of $\hat{Tr}(A)$. We thus wish to know whether B has an accepting run over $\hat{Tr}(A)$.

To determine whether B has an accepting run over $\hat{Tr}(A)$, Knapik et al. define a new panic automaton C which is essentially the product construction of B (a non-deterministic parity tree automaton) and A (a panic automaton). The result is C, where C is a panic automaton but with a colouring function and a parity acceptance condition. C generates $[\![R]\!]$ in the *exact* same way as A does, and in doing so it can check whether B holds (recall that B is a property of $\hat{Tr}(A)$). The problem is now reduced to checking whether C has an accepting run, or, equivalently determining the winner of the parity game defined by such an automaton. Knapik et al. refer to the game induced by C as $Gr(C)$.

The remainder of the proof now hinges on showing that determining the winner of $Gr(C)$ is decidable. This is done by giving a reduction from $Gr(C)$ to $Game(C)$ where the latter is a game over an arena that is defined by a 2-tree. Before we define a 2-tree, we must consider the $\star$-operation considered in [Wal96] (via [She75, Stu75, Sem84]). Given a relational structure $M = \langle\, D, r_1, \cdots, r_n \,\rangle$ where D is the domain and r_i are relations over D, the $\star$-operation creates a new logical structure:

$$M^\star = \langle\, D^*, son, clone, r_1^\star, \cdots, r_n^\star \,\rangle$$

where $son(w, wd)$ holds for all $w \in D^*$ and $d \in D$, and similarly for $clone(wd, wdd)$. Let us illustrate the power of the $\star$-operation by considering, for example, a simple pushdown tree automaton $\langle\, Q, \Sigma, \Gamma, \delta, q_0, \perp \,\rangle$. It is easy to show that the configuration graph of such an automaton is MSO-definable in $M^\star$ where $M = \langle\, (Q \cup \Gamma) \,\rangle$, i.e. M consists of just the domain and no relations. A configuration of the automaton is represented as a string over $\Gamma^* Q$. For example, if $\delta(q, a) = (q', \mathsf{push}(a'))$ then we describe an edge from waq to $waa'q$ for all $w \in \Gamma^*$. For a transition $\delta(q, a) = (q', pop)$ we put an edge from waq to wq'. All of these edges are MSO-definable in $M^\star$ thanks to the operations of *son* (*clone* is not used here).

If we now let M define a tree, then the $\star$-operation gives us a tree of trees, and these are referred to as 2-trees. It is then easy to show that the graphs of ordinary 2-pushdown automaton can be MSO defined in these trees and furthermore that these trees have a decidable MSO theory[9]. The idea is thus to extend this to panic automaton: ordinary 2-pushdown automaton with the panic operation. The proof method is quite intricate and introduces an element of non-determinism; and this is where the similarities between our two approaches becomes apparent. To describe edges that correspond to panic actions seems impossible to encode directly therefore Knapik et al. get around this by making use of non-deterministic predicting when panic actions occur and also verifying these predictions.

[9]See [Cac03] for an excellent exposition of how this is done.

Let us recall how panic actions arise. Recall that each time the level-2 pushdown store is headed by a variable $\varphi x_1 \cdots x_n$ where φ is of level-1, then we perform a push_2: enabling us to save the current configuration (on the stack below) and the newly created stack signals the start of our evaluation of φ. Subsequently, we may eventually reach a point, after evaluating φ (and possibly other variables) that we must return to our calling site, and this achieved by the panic action. Knapik et al. handle this by, upon entering a configuration where the topmost symbol of the panic automaton is given by $\varphi x_1 \cdots x_n$ for some level-1 variable φ, they both (1) predict exactly all the ways in which we will return to this configuration via a panic action in the future (2) generate a "certificate" describing these ways. The computation then proceeds by (1) for each predicted way, we assume that the we have just returned from a panic action (without executing it) and check that the run from here is accepting and (2) proceed as usual with the certificate in hand. In (2) when we are about the perform a panic action we do not perform it at all and merely check that we comply with the certificate. This is very similar to the notions captured by our simulating automaton of guessing and verifying. As with out approach, Knapik et al. define the new game to handle abortive computations in the case that certificates are violated and so forth. The end result is the proof that the game $Gr(C)$ is MSO-definable in $Game(C)$, where the latter is a game over 2-tree over a suitable alphabet that allows for the guessing and verifying detailed above.

We can see that while many of the devices and the terminology used by Knapik et al. may seem orthogonal to our own, there are very deep similarities in what is trying to be achieved. In both cases, we use "simpler structures" to capture the behaviour of a panic automaton and a canonical traversal tree (respectively). These structures are the 2-tree and the computation tree respectively. These "simpler" structures both have the problem with level-1 variables: there is an element of a detour with the possibility of return and both must be overcome the hurdle of handling the fact that we cannot follow the detour without losing information about where we started from. Both proceed by a process of guessing a finite description of the detour (1) verifying it and (2) simultaneously continuing the execution of the panic automaton (resp. canonical traversal tree) assuming the detour has taken place.

4.2.2 Extensions to Higher Levels

Shortly before completing this thesis, both Klaus Aehlig and Luke Ong independently made significant advances in answering the question posed by this chapter. For example, Aehlig extended the results of this chapter to all levels [Aeh06] for MSO properties that result in automata *with the trivial acceptance condition*; namely that there exists an infinite run. Although this is still only a partial answer the methodology is quite distinct from those used in this chapter and could potentially have further applications. Given a fixed term tree automaton, Aehlig proceeds by defining a new finitary semantics for simply-typed λ-terms based on the notion of whether or not an infinite run exists for the tree denoted by this term. This semantics can then be used to divide infinite trees into finitely many classes, according to which state has an infinite run. Aehlig then presents a calculus based on this semantics that is both sound and complete and the result therefore follows as an application of the adequacy of the semantics.

On the other hand, Luke Ong [Ong06c] extended this result to all levels of the hierarchy.

Specifically, he shows that the term tree defined by any level-n recursion scheme[10] possesses a decidable MSO theory – note the restriction of homogeneity has been dropped. In addition, he shows that for level-n, the modal mu-calculus model-checking problem for trees generated by recursion schemes is n-EXPTIME complete.

This seminal result turns out to be a relatively straightforward extension of the concepts introduced in this chapter. In fact, many of the concepts extend to higher-levels for free. As an example, the construction of a canonical traversal tree for a level-n recursion scheme carries over without any modification to its definition.

However, for the construction of the "traversal-simulating automaton", or S_N as we have called it (c.f. Section 4.1.3), the concepts of environments and interfacing configurations must be extended higher-levels. Ong achieves this by introducing the concept of a variable profile for a variable of type A where:

$$\begin{aligned} VP(o) &= V^o \times Q \times 2^Q \times 2^{\emptyset} \\ VP(A_1, \cdots, A_n, o) &= V^A \times Q \times 2^Q \times 2^{\bigcup_{i=1}^{n} VP(A_i)} \end{aligned}$$

where V^A is the set of variables[11] of type A that occur in the computation tree for a recursion scheme R[12]. An environment then consists of a set of variable profiles. Note that if we limit ourselves to the level-2 case, this is isomorphic to our definition of an environment. Interfacing configurations are extended to higher-levels in an analogous fashion.

In this way, the "essence" of the definition of the traversal-simulating automaton and the proofs of both directions remain very much the same as described for the level-2 case in this chapter. It is quite clear, however, from reading Luke Ong's extension to the whole hierarchy that the prodigious use of game semantics has been quite integral to the result. However, all the tools and game semantic notions used in the extension are exactly those introduced here. Although, there are a number of places (such as the definition of a traversal) where Ong's result and the one presented here differ in terms of terminology.

Lastly, we comment on the complexity result. The n-EXPTIME hardness for level-n follows from Cachat's result [Cac03] for level-n safe trees (looking at it from the point of view of pushdown trees). For completeness, Ong proceeds by recasting the question of "does the traversal-simulating automaton have an accepting run?" into deciding the winner of an appropriate parity game. The arena of the parity game results from, effectively, taking the product of the finite automaton that defines $[\![R^0]\!]$ and the traversal-simulating automaton. One can then employ the following standard complexity result:

Theorem 4.2.1. *[Jur00] The winning region of Eloise and her winning strategy in a parity*

[10]Whether safe or not and whether homogeneous or not.

[11]Rather, representations of variables.

[12]In Ong's presentation, the component 2^Q, corresponding to the set of states seen from the current node to the variable occurrence, is actually replaced by an integer in $[k]$, where k is the maximal priority. The integer represents the maximal priority seen from the current node to the variable occurrence. Note that this retains all the information we really need. It is not necessary to know precisely which nodes were seen; all that is actually relevant to the analysis is the highest priority seen – indeed, we could have adjusted our own definitions accordingly.

game with $|V|$ vertices and $|E|$ edges and $p \geq 2$ priorities can be computed in time:

$$O\left(p \cdot |E| \cdot \left(\frac{|V|}{\lfloor p/2 \rfloor}\right)^{\lfloor p/2 \rfloor}\right)$$

Unfortunately, the direct application of this result does not render the correct bound – and this is ultimately due to the huge branching factor of run-trees for the traversal-simulating automaton as defined here. To complete the proof one must be able to show that the traversal simulating automaton can be made more economical. We illustrate the problem using our level-2 setting. For example, recall the situation where the simulating automaton S_N is faced with an @-node. At this point S_N must consider all the possible ways in which it may visit φ for some φ occurring in the body of the abstraction rooted at the 0th child of @. Suppose that it will visit one such occurrence after seeing states Q and arriving in state q and interfacing configuration c. Even then we must guess a *family* of environments to verify the corresponding child of @. However, Ong's analysis shows that one can, in fact, always guess precisely one environment for each triple (q, Q, c) that we guess for a particular φ. This renders the desired complexity result.

4.2.3 Concluding Remarks

To summarise, this chapter has shown that the MSO theory of a term tree produced by any level-2 recursion scheme R (whether safe or not) is decidable. The result ultimately involved transferring the analysis from the value tree to the corresponding computation tree and exploiting the relationship between the two, encapsulated by the canonical traversal tree.

Though the proof of this result was both long and involved, the concepts and ideas of this result have proved to be fruitful. This can be evidenced by the recent extension of this result to recursion schemes of all levels (due to Luke Ong). Ong's result can, objectively, be said to have closed the book on the original problem posed at the beginning of this chapter by not only proving the result for the whole hierarchy but also providing an exact complexity bound. Thus, unlike the previous chapter where there was still much work to do to answer the original question, the future directions relevant to this would indeed be deviations from the original aim. For this reason we include them and other directions in the discussion of the final chapter.

Chapter 5

Conclusion

With this fifth and final chapter we conclude the thesis. We provide a summary of contributions and consider possible avenues for future work.

5.1 Summary of Contributions

Let G be a level-n grammar. By definition, G defines a language of (finite) term trees, and, as illustrated in Chapter 2 one can configure G to generate different structures, such as word languages, term tree languages, single infinite trees, forests of infinite trees and so forth. We sought to answer the following two questions:

1. Let $[\![G]\!]$ be the structure generated by G where G is a (potentially unsafe) level-n grammar. Does there exist a level-n safe grammar G' such that $[\![G']\!] = [\![G]\!]$?

2. Focusing our attention to the case where G is a recursion scheme and where G is unsafe: does $[\![G]\!]$, when represented by an appropriate logical structure, have a decidable monadic-second order theory?

Both questions were justified by the complete lack of knowledge concerning the relationship between safe and unsafe grammars. The structure of this thesis, as can be evidenced, has been constructed to reflect the above two questions: with Chapter 3 focusing on expressibility and Chapter 4 on decidability.

In Chapter 3 we showed that for word grammars of level-2, the safety restriction is a spurious constraint and that every level-2 unsafe word grammar can be converted into a safe one. This is the key contribution of this chapter and was published in FoSSaCS 2005 [AdMOb]. The proof proceeded by giving an effective construction of a non-deterministic level-2 pushdown automaton accepting the same language. The proof proceeded in two parts: first showing that the pointer machine of a level-2 unsafe word grammar can be simulated by a level-2 pushdown automaton with links; and then showing that the latter can be simulated by a level-2 non-deterministic pushdown automaton. We consider the development and definition of the level-2 pushdown automaton with links as a charactersation of level-2 unsafe grammars to be another contribution of this thesis.

The use of non-determinism in the second part of the above proof was integral to the result and the pursuit of understanding why this was so led us to the next set of results.

Namely, we introduced a definition of a deterministic word-grammar and showed the equivalence with deterministic level-n pushdown automaton. Furthermore, we showed that finding a deterministic unsafe level-2 word grammar that cannot be generated by a deterministic safe one would imply the existence of a level-2 unsafe recursion scheme whose term tree cannot be produced by a level-2 safe one. This collection of results on non-determinism are perhaps not surprising given that recursion schemes are, effectively, deterministic grammars. However, this is the first account of such correspondences in the literature and it is also interesting to see, once again, the relationships between word languages and term tree languages.

Lastly, a final contribution of this chapter is the alternative decomposition of Urzyczyn's language. Although this is a minor aside, it does provide for a particularly simple and intuitive proof of its acceptance by a level-2 non-deterministic pushdown automaton.

It is clear from the above that our efforts to analyse the effects of safety on the expressibility of higher-order grammars has only taken us as far as level-2 word grammars (implicitly OI). Our key result can thus be considered an extension of Damm and Goerdt's [DG86] result, who showed that every level-n safe word grammar can be converted into a level-n pushdown automaton and vice versa. We have shown that at level-2 unsafe word grammars can also be converted into level-2 pushdown automata. There is thus much room for future work. Many of the proposed directions were detailed in Chapter 3 itself as they were seen to be directly relevant to the chapter.

As for decidability, we have been able to show in Chapter 4 that all level-2 recursion schemes produce term trees with a decidable monadic second-order theory. This result was presented in TLCA 2005 [AdMOa]. This result was recently extended by Luke Ong [Ong06c, Ong06b] to all levels of the hierarchy – his proof method is a direct extension of the one presented here.

5.2 Future Work

A number of suggestions for future work have been discussed in earlier chapters. In this section we briefly summarise these and propose a few others.

Expressibility

Most of the proposed avenues for investigating the impact of safety on the expressive power of higher-order grammars have been discussed in depth in Chapter 3. We refer the reader back to chapter; however we recall that obvious directions involve extending our level-2 results to all levels of word grammars, and also to higher-order grammars more generally. However, there is growing suspicion that for recursion schemes safety *is* a genuine constraint and indeed, we have even suggested a possible candidate (c.f. Conjecture 3.5.13).

Decidability

Due to the work of Luke Ong, the original question posed in Chapter 4 can be said to be answered fully and this is even complete with a complexity bound. However, there are, of course, possibilities to explore other methods, for example: building on ideas by Aehlig or Knapik et al. Ong's method (and indeed our own) is very technically involved, so it would be of interest to contrast this with other approaches should such methods ever result in fruition.

Alternative Characterisations of Higher-order Grammars

As indicated in Chapter 2 the relationship between safe higher-order grammars and pushdown automata is well-established. The same, however, cannot be said for unrestricted higher-order grammars. Throughout the course of this thesis we have introduced (1) pointer machines (2) pushdown automata with links and (3) panic automata as viable machine characterisations of potentially unsafe higher-order grammars. This is still a relatively young area though, with (2) and (3) having only been defined up to level-2 and (1) serving as an implementation rather than a full machine characterisation. Clearly, their usefulness in our level-2 result for word languages and the use of panic automata in [KNUW05] are indications that understanding and developing these characterisations further may help in understanding the safety constraint and its impact on the structures generated. In fact, it appears that this is already underway.

In a yet unpublished paper, Murawski and Ong [MO06] have recently considered an extension of the level-2 pushdown automaton with links (and therefore of the panic automaton). They introduce higher-order collapsible pushdown automaton (CPDA) and show that they are equivalent to higher-order recursion schemes as generators of node-labelled trees. The authors mention that these are a variation of the pushdown automaton with links introduced in Chapter 2 and they retain the feature of being able to choose when to create a link and the ability to follow a link (now referred to as the "collapse" operation). Apearancewise they are thus close to the pushdown automaton with links. Murawski and Ong show that every level-n recursion scheme (not assumed to be safe, nor homogeneously typed) can be transformed into a CPDA of the same level – here the CPDA computes traversals of the computation tree and therefore, paths in of the value tree (using notation from Chapter 4). In a second unpublished paper, Ong [Ong06d] considers and defines parity games over the configuration graphs collapsible pushdown automaton and shows that they are solvable. The proof proceeds by reducing the problem to a suitable model-checking problem for higher-order recursion schemes.

The Safe Lambda Calculus

The safe lambda calculus was introduced as an alternative characterisation of the safety constraint as defined by Knapik et al. While we did not make use of this characterisation in any of our results, this characterisation made explicit an interesting algorithmic property of safe lambda terms; namely that one never needs to worry about variable capture. This is indeed the reason by Knapik et al.'s construction in [KNU01] is quoted (in other words) not to extend to unsafe grammars. Potential developments in this area are to consider models of this calculus and potentially a corresponding Curry-Howard-type isomorphism with an appropriate logic.

Recent work by Blum [Blu06b, Blu06a] has already shown considerable steps have been taken to better understanding and investigating the properties of the safe lambda calculus. In his technical report [Blu06b], Blum reviews the definition of the safe λ-calculus as it first appeared in [AdMO04] and produces a new calculus : one without the homogeneity constraint. It still retains the property that one need not worry about variable capture when performing (simultaneous) substitutions. Blum also develops the notion of safe-β reduction to capture the set of multi-step beta-reductions required to achieve the effect of such a simultaneous substitution.

Another contribution by Blum is a game-semantic characterisation of the safe lambda calculus. Whilst this may seem incompatible at first, given that safety is a syntactic constraint and game semantics is, in essence, syntax-free, Blum achieves this novel result by making explicit a correspondence between a syntactical representation of a term and its game denotation. He ultimately goes on to prove that the pointers in the game semantics of a safe simply-typed term (using his non-homogeneous calculus) can be uniquely reconstructed from the underlying sequence of moves.

Model-checking in Practice

Although the complexity of the model-checking problem is non-elementary and therefore impractical for implementation, the fact that it is decidable with respect to *MSO* opens many doors. Logics such as LTL, CTL, alternation-free μ-calculus are automatically decidable for such recursion schemes. An obvious line of investigation is to determine whether any of these weaker logic give rise to more tractable complexity bounds. Should this prove successful one may then even derive model-checkers based on recursion schemes and these weaker logics. Alternatively, one can focus on problems such as reachability or termination analysis.

From Trees to Graphs

The definition of a higher-order grammar is very much tree-based. One may ask whether such a concept can be extended to graphs and hypergraphs. In the event of a successful definition of a graph-generating higher-order grammar, perhaps we will be able to compare safe such graphs to the Caucal graph hierarchy and the status of unsafe graphs.

Equivalence

Given higher-order grammars of level n, G and G' are $[\![G]\!]$ and $[\![G']\!]$ equal? A decision procedure for unrestricted G and G' seems unlikely. Indeed, in the case where they both define context-free languages the answer is already undecidable. However, one may consider limiting oneself to the deterministic case: for example, if G and G' are level-1 word grammars and deterministic then a decision procedure exists [S9́7, Sti01a]. Perhaps this can be extended to the whole hierarchy of deterministic OI word grammars.

Pumping Lemmas and Normal Forms

Similar to the above or in aid of the above, one may also be interested in computing normal-forms for a general higher-order grammar G. For example, in Damm and Goerdt's proof the equivalence between OI safe word grammars and pushdown automaton required an inspired use of normal forms in one direction. Other language tools such as pumping Lemmas, or general tools for helping to show whether a particular word or tree is in $[\![G]\!]$ are also in short supply. Some progresses in this area were touched upon in Chapter 3.

As can be evidenced there is much exciting work to be done in this field.

Bibliography

[AdMOa] 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 *TLCA '05*, volume 3641 of *LNCS*.

[AdMOb] K. Aehlig, J. G. de Miranda, and C. H. L. Ong. Safety is not a restriction at level 2 for string languages. In *FoSSaCS '05*, volume 3441 of *LNCS*.

[AdMO04] K. Aehlig, J. G. de Miranda, and C. H. L. Ong. Safety is not a restriction at level 2 for string languages. Technical Report PRG-RR-04-23, OUCL, 2004.

[Aeh06] K. Aehlig. A finite semantics of simply-typed lambda terms for infinite runs of automata. In *CSL*, LNCS. Springer-Verlag, 2006.

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

[AN98] C.A. Albayrak and T. Noll. The WHILE hierarchy of program schemes is infinite. In M. Nivat, editor, *Proceedings of Foundations of Software Science and Computation Structures (FoSSaCS'98)*, volume 1378, pages 35–47. Springer Verlag, 1998.

[Bar97] K. Barthelmann. On equational simple graphs. Technical Report 9, Univerität Mainz, Institut für Informatik, 1997.

[BBDEL96] I. Beer, S. Ben-David, C. Eisner, and A. Landver. Rulebase: An industry-oriented formal verification tool. In *Design Automation Conference*, pages 655–660, 1996.

[BG04] A. Blumensath and Erich Gradel. Finite presentations of infinite structures: Automata and interpretations. In *Theory of Comput. Systems*, 2004.

[Blu01] A. Blumensath. Prefix-recognisable graphs and monadic second-order logic. Technical Report AIB-2001-06, RWTH Aachen, 2001.

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

[Blu06a] W. Blum. Game semantics of the safe λ-calculus. PRG Student Conference, 2006.

[Blu06b] W. Blum. Transfer thesis. Oxford University, Programming Research Group (Untitled), 2006.

[BS01] J. Bradfield and C. Stirling. Modal logics and mu-calculi. In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebra*, pages 293–332. Elsevier, North-Holland, 2001.

[Cac01] T. Cachat. Two-way tree automata solving pushdown games. In *Automata, Logics, and Infinite Games*, pages 303–317, 2001.

[Cac03] T. Cachat. *Games on Pushdown Graphs and Extensions.* PhD thesis, 2003.

[Cau92] D. Caucal. On the regular structure of prefix rewriting. *Theor. Comput. Sci.*, 106(1):61–86, 1992.

[Cau96] D. Caucal. On infinite transition graphs having a decidable monadic theory. In *Proceedings 23rd ICALP*, pages 194–205. Springer, 1996. LNCS Vol. 1099.

[Cau02a] D. Caucal. On infinite graphs having a decidable monadic theory. In *Proceedings 27th MFCS*, pages 1165–176. Springer, 2002. LNCS Vol. 2420.

[Cau02b] D. Caucal. On infinite terms having a decidable monadic theory. In *Proceedings 27th MFCS, Warszawa, 2002.* Springer, 2002.

[CC03] A. Carayol and T. Colcombet. On equivalent representations of infinite structures. In *ICALP*, pages 599–610, 2003.

[CGP99] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.

[CK02] B. Courcelle and T. Knapik. The evaluation of first-order substitution is monadic second-order compatible. *Theoretical Computer Science*, 281, 2002.

[CN78] B. Courcelle and M. Nivat. The algebraic semantics of recursive program schemes. In *MFCS*, pages 16–30, 1978.

[Cou76] B. Courcelle. *Sur les ensembles algébriques d'arbres et les langages déterministes, quelques applications à la théorie des schemas de programmes.* Thèse, Université de Paris-7, 1976.

[Cou78a] B. Courcelle. A representation of trees by languages i. *Theor. Comput. Sci.*, 6:255–279, 1978.

[Cou78b] B. Courcelle. A representation of trees by languages ii. *Theor. Comput. Sci.*, 7:25–55, 1978.

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

[Cou89] B. Courcelle. The monadic second-order logic of graphs II: infinite graphs of bounded width. *Mathematical Systems Theory*, 21:187–221, 1989.

[Cou90] B. Courcelle. Graph rewriting: An algebraic and logic approach. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 193–242. 1990.

[Cou94] B. Courcelle. Monadic second-order definable graph transductions: A survey. *Theor. Comput. Sci.*, 126(1):53–75, 1994.

[Cou95] B. Courcelle. The monadic second-order logic of graphs IX: machines and their behaviours. *Theoretical Computer Science*, 151:125–162, 1995.

[CS92] R. Cleaveland and B. Steffen. A linear–time model–checking algorithm for the alternation–free modal mu–calculus. In Kim G. Larsen and Arne Skou, editors, *Proceedings of Computer Aided Verification (CAV '91)*, volume 575, pages 48–58, Berlin, Germany, 1992. Springer.

[CW98] B. Courcelle and I. Walukiewicz. Monadic second-order logic, graph coverings and unfoldings of transition systems. *Ann. Pure Appl. Logic*, 92(1):35–62, 1998.

[CW03] A. Carayol and S. Wöhrle. The Caucal hierarchy of infinite graphs in terms of logic and higher-order pushdown automata. In *Proceedings of the 23rd Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2003*, pages 112–123. Springer-Verlag, 2003. LNCS 2914.

[Dam77a] W. Damm. Higher type program schemes and their tree languages. In *Theoretical Computer Science*, pages 51–72, 1977.

[Dam77b] W. Damm. Languages defined by higher type program schemes. In *ICALP*, pages 164–179, 1977.

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

[dBS69] J. W. de Bakker and D. Scoot. A theory of programs. unpublished report, August 1969.

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

[DR93] V. Danos and L. Regnier. Local and asynchronous beta-reduction (an analysis of girard's execution formula). In *LICS*, pages 296–306, 1993.

[EJ91] E. A. Emerson and C. Jutla. Tree automata, μ-calculus and determinacy. In *FOCS*, volume 32, pages 368–377, October 1991.

[EJS01] E. A. Emerson, C. S. Jutla, and A. P. Sistla. On model checking for the -calculus and its fragments. *Theor. Comput. Sci.*, 258(1-2):491–522, 2001.

[EMC86] A. P. Sistla E. M. Clarke, E. A. Emerson. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Transaction on Programming Languages and Systems*, 8(2):244–263, 1986.

[Eng74] J. Engelfriet. Simple program schemes and formal languages. *Lecture Notes in Computer Science*, 20, 1974.

[Eng83] J. Engelfriet. Iterated pushdown automata and complexity classes. In *STOC '83: Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 365–373, New York, NY, USA, 1983. ACM Press.

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

[ES77] J. Engelfriet and E. M. Schmidt. Io and oi. i. *J. Comput. Syst. Sci.*, 15(3):328–353, 1977.

[ES78] J. Engelfriet and E. M. Schmidt. Io and oi. ii. *J. Comput. Syst. Sci.*, 16(1):67–99, 1978.

[EV87] J. Engelfriet and H. Vogler. Look-ahead on pushdowns. *Inf. Comput.*, 73(3):245–279, 1987.

[Gal81] J. Gallier. Dpda's in 'atomic normal form' and applications to equivalence problems. *Theoretical Computer Science*, 14:155–186, 1981.

[Gil96] R. H. Gilman. A shrinking lemma for indexed languages. *Theoretical Computer Science*, 163:277–281, 1996.

[GL73] S. J. Garland and D. C. Luckham. Program schemes, recursion schemes, and formal languages. *J. Comput. Syst. Sci.*, 7(2):119–160, 1973.

[Gre69] S. A. Greibach. Full afls and nested iterated substitution. In *FOCS*, pages 222–230, 1969.

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

[Gue79] I. Guessarian. Program transformations and algebraic semantics. *TCS*, 9(1):39–65, 1979.

[Hay83] T. Hayashi. On derivation trees of indexed grammars: An extension of the uvwxy-theorem. *Publ. RIMS Kyoto Univ.*, 9:61–92, 1983.

[HMU01] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, second edition, 2001.

[HO00] J. M. E. Hyland and C.-H. L. Ong. On Full Abstraction for PCF: I. Models, observables and the full abstraction problem, II. Dialogue games and innocent strategies, III. A fully abstract and universal game model. *Information and Computation*, 163:285–408, 2000.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[Hun99] H. Hungar. Model checking and higher-order recursion. In *Mathematical Foundations of Computer Science*, pages 149–159, 1999.

[IDM94] S. Ben-David I. Beer, R. Gewirtzman D. Geist, and M. Yoeli. Methodology and system for practical formal verification of reactive hardware. In David L. Dill, editor, *Proceedings of the sixth International Conference on Computer-Aided Verification CAV*, volume 818, pages 182–193, Standford, California, USA, 1994. Springer-Verlag.

[Ind76] K. Indermark. Schemes with recursion on higher types. In *MFCS*, pages 352–358, 1976.

[JAGW77] E. G. Wagner J. A. Goguen, J. W. Thatcher and J. B. Wright. Initial algebra semantics and continuous algebras. *JACM*, 24, 1977.

[Jur00] M. Jurdziński. Small progress measures for solving parity games. In *STACS*, volume LNCS 1770, pages 290–301, 2000.

[JW96] D. Janin and I. Walukiewicz. On the expressive completeness of the propositional mu-calculus with respect to monadic second order logic. In *CONCUR*, pages 263–277, 1996.

[KA04] J. G. de Miranda D. Niwiński L. Ong P. Urzyczyn I. Walukiewicz K. Aehlig, T. Knapik. [types] paper announcements: Mso theory of hyperalgebraic trees is decidable. http://lists.seas.upenn.edu/pipermail/typeslist/2004/000429.html, October 2004.

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

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

[KNUW05] T. Knapik, D. Niwiński, P. Urzyczyn, and I. Walukiewicz. Unsafe grammars, panic automata, and decidability. 25 October, 2005.

[KU88] A. Kfoury and P. Urzyczyn. Finitely typed functional programs, part ii: comparisons to imperative languages. *Report, Boston University*, 1988.

[KV00] O. Kupferman and M. Y. Vardi. An automata-theoretic approach to reasoning about infinite-state systems. In E. A. Emerson and A. P. Sistla, editors, *Proceedings of the 12th International Conference on Computer-Aided Verification (CAV'00)*, volume 1855. Springer, 2000.

[Mai74] T. S. E. Maibaum. A generalized approach to formal languages. *J. Comput. Syst. Sci.*, 8(3):409–439, 1974.

[Man73] Z. Manna. Program schema. In A. V. Aho, editor, *Currents in the Theory of Computing*. Prentice-Hall, 1973.

[Mas74] A. N. Maslov. The hierarchy of indexed languages of an arbitrary level. *Soviet Math. Dokl.*, 15:1170–1174, 1974.

[Mas75] A. N. Maslov. Indexed grammars and wijngaarden grammars. *Problemy Peredachi Informatsii*, 11(3):81–89, 1975.

[Mas76] A. N. Maslov. Multilevel stack automata. *Problemy Peredachi Informatssi*, 12:55–62, 1976.

[Mey05] A. Meyer. *Finitely Presented Infinite Graphs.* PhD thesis, 2005.

[MO06] A. Murawski and C.-H. L. Ong. Collapsible pushdown automata and recursion schemes. Oxford University, Programming Research Group, 2006.

[MS85] D. E. Muller and P. E. Schupp. The theory of ends, pushdown automata, and second-order logic. *Theoretical Computer Science*, 37:51–75, 1985.

[MS87] J. Mitchell and P. J. Scott. Typed lambda models and cartesian closed categories. In *Categories in Computer Science and Logic*. AMS, Providence, RI, 1987. Proc. Boulder.

[Niv72] M. Nivat. On the interpretation of recursive program schemes. In *Symposia Matematica*, 1972.

[Ong06a] C.-H. L. Ong. Local computation of beta-reduction by game semantics. Preprint, 2006.

[Ong06b] C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. Preprint, 2006.

[Ong06c] C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes (extended abstract). In *Proceedings of IEEE Symposium on Logic in Computer Science*. Computer Society Press, 2006.

[Ong06d] C.-H. L. Ong. Solving games over collapsible pushdown graphs. Oxford University, Programming Research Group, 2006.

[PDS80a] R. Parchmann, J. Duske, and J. Specht. On deterministic indexed languages. *Information and Control*, 45:48–67, 1980.

[PDS80b] R. Parchmann, J. Duske, and J. Specht. On deterministic indexed languages. *Information and Control*, 46:200–218, 1980.

[QS81] J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in cesar. *Proc. 5th Int. Symp. in Programming*, 60, 1981.

[Rab69] M. O. Rabin. Decidability of second-order theories and automata on infinite trees. *Trans. Amer. Maths. Soc*, 141:1–35, 1969.

[Rab72] M. O. Rabin. Automata on infinite objects and church's problem. *Amer. Math. Soc., Providence, RI*, 1972.

[Ś97] G. Sénizergues. The equivalence problem for deterministic pushdown automata is decidable. In *Proceedings of ICALP'97*, pages 671–681, 1997. LNCS Volume 1256.

[Sem84] A. Semenov. Decidability of monadic theories. In *MFCS*, volume 176 of *LNCS*, pages 162–175. Springer-Verlag, 1984.

[She75] S. Shelah. The monadic second order theory of order. *Annals of Mathematics*, 102:379–419, 1975.

[Sti00] C. Stirling. Schema revisited. In *CSL: 14th Workshop on Computer Science Logic*. LNCS, Springer-Verlag, 2000.

[Sti01a] C. Stirling. Decidability of DPDA equivalence. *Theoretical Computer Science*, 255:1–31, 2001.

[Sti01b] C. Stirling. *Modal and Temporal Properties of Processes*. Texts in Computer Science. Springer-Verlag, 2001.

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

[Stu75] J. Stupp. The lattice-model is recursive in the original model. Institute of Mathematics, The Hebrew University, Jerusalem, January 1975.

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

[Tho01] W. Thomas. A short introduction to infinite automata. In *Developments in Language Theory*, pages 130–144, 2001.

[Tho03] W. Thomas. Constructing infinite graphs with a decidable MSO-theory. In *Proceedings of the 28th International Symposium on Mathematical Foundations of Computer Science*. Springer-Verlag, 2003. LNCS.

[Tur36] A. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936.

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

[Var98] M. Vardi. Reasoning about the past with two-way automata. *Lecture Notes in Computer Science*, 1443:628+, 1998.

[vD83] D. van Dalen. *Logic and Structure*. Springer-Verlag, second edition, 1983. Universitext.

[Wal96] I. Walukiewicz. Monadic second order logic on tree-like structures. In *STACS*, pages 401–413, 1996.

[Wal00] I. Walukiewicz. Pushdown processes: games and model-checking. *Information and Computation*, 157:234–263, 2000.

[Wan74] M. Wand. An algebraic formulation of the chomsky hierarchy. In *Category Theory Applied to Computation and Control*, pages 209–213, 1974.

[Wöh05] S. Wöhrle. *Decision Problems over Infinite Graphs : Higher-order Pushdown Systems And Synchronised Products*. PhD thesis, 2005.