View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Digitale Hochschulschriften der LMU

# Verification of Non-Regular Program Properties

Roland Axelsson

München 2010

# Verification of Non-Regular Program Properties

**Roland Axelsson**

Dissertation
am Institut für Informatik
Ludwig–Maximilians–Universität
München

vorgelegt von
Roland Axelsson

München, den 27.4.2010

Erstgutachter: Prof. Dr. Martin Lange

Zweitgutachter: Prof. Dr. Thomas Wilke

Tag der mündlichen Prüfung: 25.6.2010

# Abstract

Most temporal logics which have been introduced and studied in the past decades can be embedded into the modal $\mathcal{L}_\mu$. This is the case for e.g. PDL, CTL, CTL$^*$, ECTL, LTL, etc. and entails that these logics cannot express non-regular program properties. In recent years, some novel approaches towards an increase in expressive power have been made: Fixpoint Logic with Chop enriches $\mathcal{L}_\mu$ with a sequential composition operator and thereby allows to characterise context-free processes. The Modal Iteration Calculus uses inflationary fixpoints to exceed the expressive power of $\mathcal{L}_\mu$. Higher-Order Fixpoint Logic (HFL) incorporates a simply typed $\lambda$-calculus into a setting with extremal fixpoint operators and even exceeds the expressive power of Fixpoint Logic with Chop. But also PDL has been equipped with context-free programs instead of regular ones.

In terms of expressivity there is a natural demand for richer frameworks since program property specifications are simply not limited to the regular sphere. Expressivity however usually comes at the price of an increased computational complexity of logic-related decision problems. For instance are the satisfiability problems for the above mentioned logics undecidable. We investigate in this work the model checking problem of three different logics which are capable of expressing non-regular program properties and aim at identifying fragments with feasible model checking complexity.

Firstly, we develop a generic method for determining the complexity of model checking PDL over arbitrary classes of programs and show that the border to undecidability runs between PDL over indexed languages and PDL over context-sensitive languages. It is however still in PTIME for PDL over linear indexed languages and in EXPTIME for PDL over indexed languages. We present concrete algorithms which allow implementations of model checkers for these two fragments.

We then introduce an extension of CTL in which the *until*- and *release*- operators are adorned with formal languages. These are interpreted over labeled paths and restrict the moments on such a path at which the operators are satisfied. The *until*-operator

is for instance satisfied if some path prefix forms a word in the language it is adorned with (besides the usual requirement that until that moment some property has to hold and at that very moment some other property must hold). Again, we determine the computational complexities of the model checking problems for varying classes of allowed languages in either operator. It turns out that either enabling context-sensitive languages in the *until* or context-free languages in the *release*- operator renders the model checking problem undecidable while it is EXPTIME-complete for indexed languages in the *until* and visibly pushdown languages in the *release*- operator. PTIME-completeness is a result of allowing linear indexed languages in the *until* and deterministic context-free languages in the *release*. We do also give concrete model checking algorithms for several interesting fragments of these logics.

Finally, we turn our attention to the model checking problem of HFL which we have already studied in previous works. On finite state models it is $k$EXPTIME-complete for HFL$^k$, the fragment of HFL obtained by restricting functions in the $\lambda$-calculus to order $k$. Novel in this work is however the generalisation (from the first-order case to the case for functions of arbitrary order) of an idea to improve the best and average case behaviour of a model checking algorithm by using partial functions during the fixpoint iteration guided by the neededness of arguments. This is possible, because the semantics of a closed HFL formula is not a total function but the value of a function at some argument. Again, we give a concrete algorithm for such an improved model checker and argue that despite the very high model checking complexity this improvement is very useful in practice and gives feasible results for HFL with lower order fuctions, backed up by a statistical analysis of the number of needed arguments on a concrete example.

Furthermore, we show how HFL can be used as a tool for the development of algorithms. Its high expressivity allows to encode a wide variety of problems as instances of model checking already in the first-order fragment. The rather unintuitive – yet very succinct – problem encoding together with an analysis of the behaviour of the above sketched optimisation may give deep insights into the problem. We demonstrate this on the example of the universality problem for nondeterministic finite automata, where a slight variation of the optimised model checking algorithm yields one of the best known methods so far which was only discovered recently.

We do also investigate typical model-theoretic properties for each of these logics and compare them with respect to expressive power.

# Zuasmmenfassung

Die meisten Temporallogiken, welche in den vergangenen Jahrzehnten eingeführt und von der Forschung berücksichtigt wurden, lassen sich in den modalen $\mu$-Kalkül einbetten. Dies betrifft z.B. PDL, CTL, CTL$^*$, ECTL, LTL, etc. und beinhaltet, dass diese Logiken nicht dazu in der Lage sind, nicht-reguläre Programmeigenschaften auszudrücken.

In den letzten Jahren wurden allerdings eine Reihe ausdrucksstärkerer Logiken entwickelt: Fixpoint Logic with Chop erweitert den $\mu$-Kalkül um einen Operator für sequentielle Komposition und erlaubt es dadurch, logische Charakterisierungen von kontextfreien Prozessen anzugeben. Im Modal Iteration Calculus führen inflationäre Fixpunkte dazu, dass seine Ausdrucksstärke diejenige des $\mu$-Kalküls übersteigt. Higher-Order Fixpoint Logic (HFL) vereint in sich einen einfach getypten $\lambda$-Kalkül sowie kleinste und grösste Fixpunktquantoren und ist damit sogar noch ausdrucksstärker als Fixpoint Logic with Chop. Selbst PDL wurde in der Vergangenheit bereits mit kontextfreien anstelle von regulären Programmen untersucht.

Da Spezifikationen von Programmeigenschaften nicht auf Regularität beschränkt sind, ergibt sich ein natürlicher Bedarf an ausdrucksstärkeren Spezifikationsformalismen. Grössere Ausdrucksstärke ist jedoch üblicherweise mit einem Ansteigen der Komplexität der im Zusammenhang mit der Logik stehenden Entscheidungsprobleme verbunden. Beispielsweise sind die Erfüllbarkeitsprobleme für jede der oben genannten Logiken unentscheidbar. Die vorliegende Arbeit untersucht die Model Checking Probleme von drei verschiedenen Logiken, welche im Stande sind, nicht-reguläre Eigenschaften auszudrücken und gibt Fragmente von ihnen an, welche eine in der Praxis noch verwertbare Komplexität in Bezug auf das Model Checking Problem besitzen.

Zunächst wird eine generische Methode entwickelt, um die Komplexität des Model Checking Problems von PDL über beliebigen Klassen von Programmen zu bestimmen. Es wird gezeigt, dass die Grenze zur Unentscheidbarkeit zwischen PDL über indexierten Sprachen und PDL über kontextsensitiven Sprachen verläuft. Für PDL über linear indexierten

Sprachen ist das Problem noch immer in PTIME und für PDL über indexierten Sprachen in EXPTIME. Wir geben für diese beiden Fragmente konkrete Algorithmen für eine Implementierung an.

Im Anschluss führen wir eine Erweiterung von CTL ein, in welcher die *until*- und *release*-Operatoren mit formalen Sprachen ausgestattet sind. Diese Sprachen werden über beschrifteten Pfaden interpretiert und kennzeichnen die Momente entlang solcher Pfade in welchen die Operatoren erfüllt sein müssen. So ist beispielsweise der *until*-Operator erfüllt, falls es einen Pfadpräfix gibt, welcher ein Wort in der Sprache bildet, mit der der Operator ausgestattet ist (und die übliche *until*-Bedingung gilt, nämlich, dass eine bestimmte Eigenschaft in jedem Zustand bis zu diesem Zeitpunkt gegolten hat, sowie dass eine andere in genau jenem Zeitpunkt gilt).

Wie im Fall von PDL, bestimmen wir die Komplexität des Model Checking Problems für verschiedene Klassen von erlaubten Sprachen im jeweiligen Operator. Es stellt sich heraus, dass sowohl die Klasse der kontextsensitiven Sprachen im *until*- als auch die Klasse der kontextfreien Sprachen im *release*-Operator zu Unentscheidbarkeit des Model Checking Problems führen. Es ist EXPTIME-vollständig für indexierte Sprachen im *until*- und visibly pushdown Sprachen im *release*-Operator. Linear indexierte Sprachen im *until* sowie deterministisch kontextfreie Sprachen im *release* führen zu einem PTIME-vollständigen Model Checking Problem. Wir geben ebenfalls wieder konkrete Model Checking Algorithmen für ausgewählte Fragmente dieser Logiken an.

Schliesslich wenden wir uns dem Model Checking Problem für HFL zu, welches wir bereits in vorangegangenen Arbeiten untersucht haben. Auf endlichen Modellen ist es $k$EXPTIME-vollständig für $\mathrm{HFL}^k$ (das Fragment von HFL, welches man erhält, wenn man die Ordnung der Funktionen im $\lambda$-Kalkül auf $k$ beschränkt). Neu ist jedoch die Verallgemeinerung einer Idee welche für $\mathrm{HFL}^1$ entwickelt wurde und nun auf das gesamte HFL ausgeweitet wird, um das Verhalten des Model Checkers im besten bzw. durchschnittlichen Fall zu verbessern, indem partielle anstelle von totalen Funktionen während der Fixpunktapproximation in Abhängigkeit von den benötigten Argumentstellen berechnet werden. Dies ist deshalb möglich, weil die Semantik einer geschlossenen HFL Formel selbst keine totale Funktion ist, sondern der Wert einer Funktion an einer bestimmten Argumentstelle.

Wir geben wieder einen konkerten Algorithmus für diesen optimierten Model Checker an und vertreten die Ansicht, dass die Optimierung trotz der hohen Komplexität im schlechtesten Fall brachbare Ergebnisse in der Praxis zeitigen kann, zumindest für HFL mit Funktionen niedriger Ordnung. Wir belegen diese Ansicht durch eine statistische Auswertung

der Anzahl benötigter Argumente anhand eines konkreten Beispiels.
Desweiteren zeigen wir, wie HFL als Instrument zur Entwicklung von Algorithmen verwendet werden kann. Die grosse Ausdrucksstärke erlaubt es, eine Vielzahl von Problemen als Instanzen des Model Checking Problems zu kodieren und zwar bereits in HFL[1]. Die eher wenig intuitive Kodierung in Kombination mit einer Analyse des Verhaltens des optimierten Model Checking Algorithmus auf diesen Problemen kann tiefere Einsicht in das Problem selbst gewähren. Wir demonstrieren dies am Beispiels des Universalitätsproblems für nichtdeterministische endliche Automaten, wo eine leichte Veränderung des optimierten Model Checking Algorithmus zu einer der besten bisher bekannten Methoden dafür führt, welche erst kürzlich beschrieben wurde.
Desweiteren untersuchen wir die typischen modelltheoretischen Eigenschaften jeder dieser Logiken und vergleichen sie untereinander in Bezug auf ihre Ausdrucksstärke.

# Contents

**Abstract** v

**Zusammenfassung** vii

**1 Introduction** 1

**2 Preliminaries** 9

2.1 Formal Languages and Automata . . . 9

2.1.1 The Chomsky Hierarchy . . . 10

2.1.2 Visibly Pushdown Languages . . . 13

2.1.3 Indexed Languages . . . 17

2.1.4 Linear Indexed Languages . . . 20

2.1.5 Alternating Context-Free Languages . . . 23

2.2 Temporal Logics . . . 26

2.2.1 Labeled Transition Systems . . . 26

2.2.2 Logic and Program Verification . . . 28

2.2.3 Computational Complexity . . . 29

2.2.4 Properties of Temporal Logics . . . 30

2.2.5 Expressivity . . . 31

2.2.6 Propositional Dynamic Logic . . . 32

2.2.7 Computation Tree Logic . . . 34

2.2.8 The Modal $\mu$-Calculus . . . 36

2.2.9 Non-Regular Logics . . . 39

**3 Non-Regular Propositional Dynamic Logic** 41

3.1 Syntax and Semantics . . . 41

3.2 Examples . . . 44

3.3 Properties . . . . . 45
3.4 Expressivity . . . . . 47
3.5 Model Checking . . . . . 52
3.5.1 A Generic Method . . . . . 52
3.5.2 A Model Checking Algorithm for PDL over IL . . . . . 61
3.5.3 A Model Checking Algorithm for PDL over MCSL . . . . . 72

**4 Non-Regular Computation Tree Logic 79**
4.1 Syntax and Semantics . . . . . 80
4.2 Examples . . . . . 82
4.3 Properties . . . . . 83
4.4 Expressivity . . . . . 84
4.5 Model Checking . . . . . 90
4.5.1 Model checking EU[PDA] . . . . . 101
4.5.2 Model checking ER[DPDA] . . . . . 105

**5 Higher-Order Fixpoint Logic 109**
5.1 Syntax and Semantics . . . . . 110
5.2 Examples . . . . . 116
5.3 Properties . . . . . 117
5.4 Expressivity . . . . . 117
5.5 Model Checking . . . . . 122
5.5.1 A Standard Fixpoint-Approximation Algorithm . . . . . 123
5.5.2 A Model Checker Using Neededness Analysis . . . . . 124
5.5.3 Soundness and Completeness . . . . . 130
5.5.4 Applications and Evaluation in Practice . . . . . 133

**6 Further Work 143**
Bibliography . . . . . 145

**Acknowledgment 151**

# List of Figures


2.1 Example ACFG derivation. . . . . . . . . . . . . . . . . . . . . . . . . . . 25
2.2 Expressive power of some regular logics. . . . . . . . . . . . . . . . . . . . 38

3.1 Complexity of satisfiability for PDL[$\mathcal{L}$]. . . . . . . . . . . . . . . . . . . . . . . 47
3.2 Expressive power of PDL[$\mathcal{L}$]. . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
3.3 REG-intersection and emptiness for some language classes. . . . . . . . . . 58
3.4 Complexity of SAT vs. model checking PDL[$\mathcal{L}$]. . . . . . . . . . . . . . . . 60

4.1 Complexity of satisfiability for CTL[$\mathcal{L}$] . . . . . . . . . . . . . . . . . . . . 84
4.2 Expressive power of CTL[$\mathcal{L}$] . . . . . . . . . . . . . . . . . . . . . . . . . . . 91
4.3 Complexity of model checking CTL[$\mathfrak{A}, \mathfrak{B}$]. . . . . . . . . . . . . . . . . . . 100

5.1 Type inference rules for HFL. . . . . . . . . . . . . . . . . . . . . . . . . . 111
5.2 Example type derivation of a HFL formula. . . . . . . . . . . . . . . . . . . 112
5.3 Expressive power of PDL[$\mathcal{L}$], CTL[$\mathcal{L}$] and HFL. . . . . . . . . . . . . . . . . 121
5.4 A model checking algorithm for HFL. . . . . . . . . . . . . . . . . . . . . . 125
5.5 Algorithm MC-HFL running on NFA-UNIV. . . . . . . . . . . . . . . . . . . 129
5.6 Counting arguments in the function table of MC-HFL. . . . . . . . . . . . 135
5.7 A transition system representation of a QBF formula. . . . . . . . . . . . . 138
5.8 A transition system representation of a formula in modal logic $K$. . . . . . 140

# Chapter 1

# Introduction

**A Quick Survey on Temporal Logics.** In order to reason about program behaviour and in particular about the stepwise execution of processes, a notion of time and a formalism which enables to describe the changes over time is required.

First efforts included Hoare-style program verification [Hoa69], where valid statements about a typically sequential program are derived by applying inference rules to each program statement and the currently valid conditions. Following classical proof systems, application of the inference rules required human ingenuity and made verification of larger programs extremely tedious, because manual intervention was inevitable.

The idea to interpret modal logic in the context of temporal succession goes back to A. Prior in 1957 [Pri57] and has since then evolved into a large and productive research field. Its importance is rooted in the emerging industrial need for safe hard- and software systems over the years. Techniques such as model checking, i.e. an algorithmic solution to the question whether a given model of a system satisfies its specification provide a solid mathematical basis to ideally guarantee that a set of properties holds for a system. The greatest benefit herein lies in the fact that (sufficient computing power granted) the model checking process is designed for full automation.

Many logics have since Prior been invented to formally reason about time and program behaviour; most of them still have in common the modal foundation but otherwise differ a lot in the machinery of temporal operators provided. Kripke's possible world semantics [Kri63] in general becomes a transition system in the context of program reasoning and the meanings of possibility and necessity shift to *"there is a (direct) successor in time"* and *"for all (direct) successors in time"*.

In the most simple temporal logics like e.g. *Hennessy-Milner-Logic* [HM80], reasoning about

paths in a model can only be done by explicitly declaring the path depth, i.e. there is no recursion device which automatically applies a certain specification scheme. Not until the introduction of fixpoints into this setting is it possible to make a (finite) statement like e.g. *"along every path proposition $q$ eventually holds"*.

The 1977 landmark paper by Pnueli [Pnu77] introduced temporal logic, today known as *Linear Temporal Logic* (LTL), in which (implicitly) universally quantified statements about the runs of a system are possible. The universal quantification allows to merge all system runs into models with a linear concept of time succession – hence the later adopted name prefix. LTL has basic temporal operators: $\mathtt{X}$ (next-time), $\mathtt{F}$ (sometime), $\mathtt{G}$ (always) and $\mathtt{U}$ (until). A formula $\mathtt{X}\varphi$ requires the next moment in time to satisfy $\varphi$, $\mathtt{F}\varphi$ says that $\varphi$ eventually holds in the future, $\mathtt{G}\varphi$ says that $\varphi$ holds from now on forever and the binary operator $\varphi\mathtt{U}\psi$ is satisfied if there exists a future state at which $\psi$ holds and until that moment $\varphi$ must hold. Consider for example the formula $\mathtt{FG}p$ stating *"(on all system runs) sometime in the future, $p$ will always hold"*.

Another widely used temporal logic is *Computation Tree Logic* (CTL)[BAMP81] which keeps different runs of a system apart by modelling the succession of time seperately for every run, thus arriving at branching time models. CTL models explicitly incorporate non-determinism by allowing the time to split up whenever different system behaviour opens up a new branch of possibilites. These models preserve more information about the system behaviour than linear ones and therefore allow a richer variety of specification formalisms. Note that a tree model can be translated into a linear model but the converse translation fails. In particular, CTL is – unlike LTL – capable of specifying properties regarding single system runs, i.e. CTL allows existential quantification in addition to universal quantification over runs.

Temporal formulas of CTL consist of the same temporal operators as LTL with a similar meaning but must occur in the scope of a path quantifier ($\mathtt{A}$ and $\mathtt{E}$ for universal and existential path quantification). Furthermore, no interleaving of temporal operators is allowed unless each temporal operator is guarded by a path quantifier. So, e.g. $\mathtt{AGEF}q$ states liveness of property $q$: *"on all paths it is true everywhere that there exists a path along which eventually $q$ holds"*. For details on the logic, see Sec. 2.2.7.

Both logics are mutually incomparable which is witnessed by the above mentioned formulas. On the one hand, the interleaving of temporal operators $\mathtt{F}$ and $\mathtt{G}$ allowed in LTL cannot be expressed in CTL and on the other hand there is no existential path quantification in LTL. These features are impossible to express by other means of the respective logical languages.

The logic which unifies LTL and CTL is CTL$^*$ which combines the nesting of temporal operators with universal and existential path quantification.

Fischer and Ladner's *Propositional Dynamic Logic* (PDL) [FL79] is yet another branching time temporal logic in which the moments in time at which certain properties should hold are specified by regular expressions, also called programs. These programs $R$ are syntactically embedded into the modalities $\langle R \rangle$ and $[R]$ which correspond to existential and universal quantification over paths labeled with elements of $R$. Both LTL and CTL are interpreted over unlabeled models while using labels inside the specification formalism provides additional means to model program behaviour. For instance, the formula $\langle (aba)^* \rangle \mathtt{tt}$ states *"there exists a path labeled with a word from the language* $(aba)^*$*"*. Some of the properties of unlabeled models can be simulated in PDL however: the CTL expression $\mathtt{AG}q$ corresponds to $[\Sigma^*]q$ for instance. In general however, PDL and CTL are incomparable. For details on PDL, see Sec. 2.2.6.

Another extensively studied logic is Kozen's *modal $\mu$-calculus* [Koz82]. It is equipped with single letter modalities $\langle a \rangle$ and $[a]$ and uses extremal fixpoint constructs as recursion devices to combine these to path properties. For instance, the expression $\nu x.p \wedge \langle a \rangle \langle b \rangle x$ states *"there exists an* $(ab)^\omega$*-labeled path along which p holds in between each ab"*, where $\nu x$ is the greatest fixpoint operator. What gives $\mu$-calculus its expressive power and (seemingly) is the reason for the computational complexity of its model checking problem is the fixpoint alternation. Fixpoint alternation is the mutual dependency of fixpoints and the measure of the dependency complexity is called the alternation depth of a formula (c.f.[BS06]). The best known model checking algorithms for the $\mu$-calculus are exponential in the alternation depth.

Having a fundus of different logics with a (mostly) common base, the question of expressive power naturally arises. Interestingly, almost all well-established temporal logics can be embedded in the $\mu$-calculus, like e.g. LTL, CTL, CTL$^*$ and PDL, since the modal and temporal operators can be expressed as least and greatest fixpoints of a certain form. In fact they can even (except for CTL$^*$) be embedded into the alternation-free fragment of the $\mu$-calculus. For details on syntax and semantics, see Sec. 2.2.8.

**Regularity and Logic.** The $\mu$-calculus plays a very important role, because via its tight relationship with *Monadic Second-Order Logic* (MSO), it connects temporal logic with automata theory.

The works of Büchi and Elgot have shown that MSO (interpreted over finite words) and

finite automata have the same expressive power and are effectively translatable into each other [Büc60, Elg61]. This result was later extended to finite automata over infinite words and trees [McN66, Rab69], namely Büchi-automata and tree-automata with a Rabin-acceptance condition.

On the other hand, the $\mu$-calculus and the bisimulation-invariant fragment of MSO do also have the same expressive power [JW96] and this finally links the theory of finite automata and $\mu$-calculus. It is in this sense that the term *"regular logic"* applies to $\mu$-calculus although it was originally coined in the context of finite automata and formal language theory.

**Temporal Logics Beyond Regularity.** Although formulas specified in the $\mu$-calculus are usually considered hard to understand (at least with increasing alternation depth), they still correspond to the least expressive fragment of the Chomsky hierarchy. Regular languages are very limited in reflecting structural complexity in comparison to the context-free and context-sensitive languages – an observation which also transfers to properties expressible in the $\mu$-calculus.

A demand for richer logical description and recognition frameworks is natural because computer processes are not restricted to regularity and hence have structural properties which cannot be expressed with regular means.

This was for instance the motivation behind the design of *Fixpoint Logic with Chop* (FLC) where a logical characterisation for a class of processes called *context-free* or BPA (Basic Process Algebra) processes was sought [MO99]. It turned out that it was sufficient to add sequential composition to the modal $\mu$-calculus to achieve this in the following way.

Formulas in branching time logics are usually interpreted as sets of states, namely those which satisfy the formula, i.e. predicates on the total state set. In FLC, the semantics is lifted to functions $[\![\cdot]\!] : 2^{\mathcal{S}} \to 2^{\mathcal{S}}$ where $\mathcal{S}$ is the state space; i.e. a formula is basically a predicate transformer. Sequential composition of formulas $\phi; \psi$ is now interpreted as function composition $([\![\phi]\!] \circ [\![\psi]\!])(x)$. This is possible because the set of all monotone functions $2^{\mathcal{S}} \to 2^{\mathcal{S}}$ forms a complete lattice with pointwise inclusion ordering which guarantees the existence of least and greatest fixpoints. As an example property (inexpressible in $\mu$-calculus) consider the formula $\mu x.\tau \wedge \langle a \rangle; x; \langle b \rangle$ stating *"there exists a path labeled with a word $w \in \{a^n b^n \mid n \geq 0\}$"*, where $\tau$ simply is the identity function needed for technical reasons regarding the FLC semantics.

The idea of formulas as functions was consequently generalized by M. and R. Viswanathan

by omitting the restriction of functions to first order in *Higher-Order Fixpoint Logic* (HFL) [VV04]. Here, the $\mu$-calculus is enriched with a simply typed $\lambda$-calculus and fixpoints range over higher-order functions instead of just first-order functions. This is well-defined because higher-order functions form a complete lattice with a pointwise inclusion order on the function values. For formula examples, see Sec. 5.2.

FLC turns out to be easily embeddable into the first-order fragment of HFL, (even restricted to arity 1) but a diagonalisation argument shows that HFL is strictly more expressive than FLC [VV04]. The question whether FLC is equivalent to the first-order fragment of HFL is still open. If we denote by $\mathrm{HFL}^k$ the fragment of HFL which is restricted to functions of order $k$, then $\mathrm{HFL}^0$, the fragment without functions is equivalent (even syntactically) to the $\mu$-calculus.

Regarding the typical decision problems for logics, matters are more or less the same than with FLC, i.e. satisfiability is undecidable and model checking is decidable on finite models only. HFL model checking is already very hard for arbitrarily small models: we have shown that the problem is $k$EXPTIME-complete for $\mathrm{HFL}^k$ even on transition systems of size 1 [ALS07]. A direct consequence of this result is that there is also a strict hierarchy of expressiveness with increasing order of the functions.

This may seem little encouraging, however in this work we show that the higher-order functions which are responsible for the $k$EXPTIME-hardness, are not needed as total but as partial functions on average. Only in worst case scenarios is the computation of the values at all arguments necessary for solving the model checking problem. The leeway between average and worst case can be exploited in practice and shown experimentally to be sufficiently large for feasible employment at least for lower-order functions.

Another aspect we consider is that for a logic as expressive as HFL, various surprisingly different general logical problems can be encoded into the model checking problem, e.g. : satisfiability of modal logic K or universality of non-deterministic finite automata (NFA). This enables a re-evaluation of known algorithms for these problems, since they are expressed in the rather unintuitive way as a fixpoint of a function. This may even lead to better ones. For instance, with a few optimisations the model checking algorithm on the NFA-universality problem turns out to be the same as the antichain method by Henzinger et al. [WDHR06] which is one of the best currently known and only discovered recently.

**Tailored Expressivity.** It is clear that expressive power comes at the price of increased computational complexity. This work discusses several non-regular logics which all have

in common that they are parametric in some sense which directly affects their expressive power. We will for instance investigate HFL, where the parameter which regulates the expressive power is the order $k$ of functions allowed. Every restriction of the order magnitude by one immediately pays off by exponentially lesser cost of model checking.

Another approach to achieve modularity in terms of expressive power is to directly incorporate formal languages into the logic as it is the case in PDL. Although the original work investigated PDL for regular programs only, it is clearly designed as a parametric logical framework over varying classes of programs. But since the focus of attention at the time was on decidability and it was very early conceived by Ladner that PDL equipped with context-free programs is undecidable c.f. [HPS83], the range of considered classes has so far been limited to those located in between the regular and context-free ones.

Interestingly, there is an enormous complexity gap between satisfiability and model checking: while PDL over context-free programs is undecidable, model checking is still in P [Lan05]. Hence it seems worthwhile to extend the range of language classes for the latter problem. In this work, we examine the model checking problem of PDL over arbitrary classes of formal languages and derive complexity bounds for the model checking problem w.r.t. the expressivity of the language class parameter. It turns out that the borderline to undecidability of model checking lies somewhere in between the indexed languages and the context-sensitive.

Here, the advantage of parametric frameworks becomes apparent: it is comparatively easy to determine an adequate formal language class in which a path property can be expressed, while the correspondence between least required function order to express such a property in HFL is unclear.

Parametric PDL mainly draws its expressive power from the language class assigned to it while the inherent logical machinery is still rather weak. It does for instance not feature CTL's *release*-operator. From this circumstance came the idea for a non-regular CTL version which we have proposed in [ALL$^+$b]. It combines the modularity of expressive power with the ease of CTL-specification.

We consider an equally parametric framework for CTL over arbitrary classes of formal languages and the corresponding model checking problems. CTL operators equipped with a formal language constrain the moments in time at which subformulas are required to hold. For instance, the formula $\texttt{EG}^L p$ states *"there is a path on which at every moment where the current path prefix forms a word in L, p holds"*.

Since it turns out that the model checking complexity of such language-adorned temporal

operators differs for $\mathtt{U}^L$ and $\mathtt{R}^L$, we discuss parametric CTL w.r.t. two language class parameters, each of them restricting the use of languages for one of them which adds further granularity to the possibilities of choice in the desired logical expressivity.

**Chapter Overview.** The preliminary chapter recalls definitions of formal language and automata theory as well as the temporal logics PDL, CTL and $\mu$-calculus which form the basis of subsequent chapters. We focus on non-standard notions from the literature and clarify the notational conventions used throughout the thesis.
In chapters 3– 5 we introduce Parametric PDL and CTL as well as HFL. The overall structure of each of these chapters is

- Syntax and Semantics
- Examples
- Properties
- Expressivity
- Model Checking.

After defining syntax and semantics of a logic and giving examples of properties expressible, in the "Properties"-section, we investigate some typical properties: the finite model and tree model property, bisimulation invariance and decidability.
Subsequently, the expressive power of the logics PDL[$\mathcal{L}$], CTL[$\mathcal{L}$] and HFL is compared and delineated against regular logics.
The main results are usually to be found in the section concerned with model checking.
Starting with the simplest – PDL[$\mathcal{L}$]– we interreduce its model checking problem to the non-emptiness problem for $\mathcal{L}$-intersections with regular languages and show the close relationship to graph reachability problems. The transfer of results from these areas allows to derive computational bounds for model checking PDL[$\mathcal{L}$] and a borderline to undecidability for language classes exceeding the context-sensitive. We then develop concrete model checking algorithms for PDL[IL] and PDL[MCSL] which are the most expressive of these logics which retain decidability and give detailed soundness and completeness proofs.
Chapter 4 deals with the verification of CTL[$\mathcal{L}$] . We give computational bounds of the model checking problem and, again, draw the border to undecidability w.r.t. $\mathcal{L}$. Here, we consider the fragments obtained by restricting the expressive power of the language class

parameters in either the *until* and *release* path quantifiers, since the resulting complexity highly depends on it. We also investigate the differences arising from deterministic and non-deterministic variants of the input automata.

In Chapter 5 the highly expressive fixpoint logic HFL is turned attention to. We generalise the model checking algorithm developed in [AL07] for its first-order fragment to the whole of HFL. The optimisations to the straight-forward algorithm enable us to reduce best- and average case complexities. We give statistical evidence that this indeed enhances the performance dramatically and leaves hope for practical feasibility despite the extremely high worst-case complexity which is a consequence of its expressiveness. We propagate the use of HFL as an extremely succinct "programming language" for all kinds of problems – from universality of non-deterministic finite automata to satisfiability checking of modal logic $K$ – to the purpose of deriving ideas for new algorithms due to the usually rather unintuitive problem formulation form, namely as a fixpoint of a higher-order function. This is backed up by the coincidence of the behaviour of our model-checker on a formula encoding universality of non-deterministic finite automata with one of the fastest methods known so far.

The final chapter summarises the achievements of this thesis and points out the directions of further work on the topics contained within.

# Chapter 2

# Preliminaries

## 2.1 Formal Languages and Automata

Formal languages and automata form the well-known dualism of language generation and language recognition. Formal languages are given as grammars which define a set of rules to derive the words of which a language consists. Their counterpart is the concept of an automaton: given a word, an automaton decides according to a set of rules whether it accepts or rejects the word as part of its language. We start with the well known notion of *transitive closure*.

**Definition 1 (Transitive Closure)** Let $R, S$ be binary relations on a universe $U$. Define $RS = \{(x, y) \in U \times U \mid \text{ exists } z \in U \text{ s.t. } xRz \text{ and } zSy\}$. The following inductive definitions for $n, i \in \mathbb{N}$ are standard:

- $R^0 := \{(x, x) \mid x \in U\}$.
- $R^{n+1} := RR^n$.
- $R^* := \bigcup_{i \geq 0} R^i$.
- $R^+ := \bigcup_{i \geq 1} R^i$.

**Definition 2 (Grammar)** A *grammar* is a 4-tuple $G = (N, \Sigma, P, S)$, where $N$ is a finite set of *nonterminal symbols*, $\Sigma$ is a finite set of *terminal symbols* – also called *alphabet* sometimes or *set of actions* in the context of logics – with $N \cap \Sigma = \emptyset$, $S \in N$ is the *starting symbol* and $P \subsetneq (N \cup \Sigma)^+ \times (N \cup \Sigma)^*$ is a finite set of *production rules*.

We use infix notation $\alpha \to \beta$ to denote $(\alpha, \beta) \in P$. An element $\alpha \in (N \cup \Sigma)^*$ is called a *sentential form* and its length $|\alpha|$ is defined as the sum of symbol occurrences from $N$ and $\Sigma$ in $\alpha$. If the length of some sentential form is 0, we call it the *empty word* and denote it by $\epsilon$.

**Definition 3 (Derivation)** Let $G$ be a grammar and $\alpha, \beta, \gamma, \gamma' \in (N \cup \Sigma)^*$. We define the *derivation relation* $\Rightarrow_G \subseteq (N \cup \Sigma)^* \times (N \cup \Sigma)^*$ as

$$\alpha\gamma\beta \Rightarrow_G \alpha\gamma'\beta \quad \text{iff} \quad \gamma \to \gamma'.$$

If it is clear to which grammar a derivation refers to, we often omit the index and simply write $\Rightarrow$ instead of $\Rightarrow_G$.

**Definition 4 (Formal Language)** The *language* of a grammar $G = (N, \Sigma, P, S)$ is defined as

$$\mathcal{L}(G) = \{w \in \Sigma^* \mid S \Rightarrow^+ w\}.$$

Typical decision problems regarding formal languages are the following:
Let $w \in \Sigma^*$ for an alphabet $\Sigma$ and let $L, L'$ be formal languages.

- *word problem*: is $w \in L$ the case?
- *emptiness problem*: is $L = \emptyset$ the case?
- *intersection problem*: is $L \cap L' = \emptyset$ the case?

### 2.1.1 The Chomsky Hierarchy

Faced with the fact that the computational complexity of solving any of the language-related decision problems for different languages varies from trivial to undecidable it seems natural to classify them according to the properties responsible for this.
The *Chomsky hierarchy* is a well-studied classification system dividing grammars (and the languages they define) into four different classes which form an inclusion hierarchy.

**Definition 5 (Chomsky Hierarchy)** Let $G = (N, \Sigma, P, S)$ be a grammar.

- $G$ is of type 0 or *recursively enumerable*.
- $G$ is of type 1 or *context-sensitive*, if $|\alpha| \leq |\beta|$ for all $\alpha \to \beta$.

- $G$ is of type 2 or *context-free*, if $\alpha \in N$ for all $\alpha \to \beta$.
- $G$ is of type 3 or *regular*, if it is context-free and $\beta \in \Sigma \cup \Sigma N$ for all $\alpha \to \beta$.

Abbreviations used throughout this text for context-free and context-sensitive grammars are CFG and CSG, respectively. We adopt the classification for formal languages and may therefore say that a language is recursively enumerable or context-free, etc. if a grammar of the corresponding type exists which generates the language. Let REG, CFL, CSL and RE denote the classes of *regular, context-free, context-sensitive* and *recursively enumerable* languages.

**Definition 6 (Finite Automaton)** A *(nondeterministic) finite automaton* (FA) is a 5-tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, where $Q \cap \Sigma = \emptyset$ and

- $Q$ is a finite set of states,
- $\Sigma$ is a finite set of terminal symbols,
- $\delta : Q \times \Sigma \to 2^Q$ is the transition function,
- $q_0 \in Q$ is the starting state,
- $F \subseteq Q$ is the set of final states.

For reasons of better readability, we may write $q \xrightarrow{a} q'$ instead of $q' \in \delta(q, a)$. We call a finite automaton *deterministic* if $|\delta(q, a)| = 1$ for all $q \in Q, a \in \Sigma$. A *run* of $\mathcal{A}$ on a word $w = a_1a_2 \dots a_n \in \Sigma^*$ is a sequence of states $q_0, q_1, \dots, q_n$ s.t. $q_0$ is the starting state and $q_i \xrightarrow{a_{i+1}} q_{i+1}$ for all $i \geq 0$. We call such a run *accepting* if $q_n \in F$.

**Theorem 1 (Myhill-Nerode, c.f. [HU79])** Let $L$ be a regular language over $\Sigma$ and define $\sim \subseteq \Sigma^* \times \Sigma^*$ as $x \sim y$ iff for all $z \in \Sigma^* : xz \in L \Leftrightarrow yz \in L$. Then $\sim$ is an equivalence relation and the number of equivalence classes is finite.

**Definition 7 (Pushdown Automaton)** A *pushdown automaton* (PDA) is a 6-tuple $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, q_0, F)$, where $Q, \Sigma, q_0$ and $F$ are defined exactly as for an NFA and

- $\Gamma$ is a finite set of stack symbols,
- $\delta : Q \times (\Sigma \cup \{\perp\}) \times \Gamma \to 2^{Q \times \Gamma^*}$ is the transition function.

Again, we may write $(q, \gamma) \xrightarrow{a} (q', \gamma')$ instead of $(q', \gamma') \in \delta(q, a, \gamma)$. We call a pushdown automaton *deterministic* if $|\delta(q, a, \gamma)| = 1$ for all $q \in Q, a \in \Sigma \cup \{\epsilon\}, \gamma \in \Gamma$.
A *configuration* of $\mathcal{A}$ is an element of $Q \times \Gamma^*$ and we denote the set of all its configurations by $\mathsf{Conf}(\mathcal{A})$. In a configuration, the second component is called the current *stack* of $\mathcal{A}$. The *starting configuration* is $(q_0, \bot)$, where $\bot$ denotes a special stack symbol $\bot \notin \Gamma$.
A *run* of $\mathcal{A}$ on $w = a_1 \dots a_n$ is a sequence of configurations $C_0, \dots, C_n$ s.t. $C_0$ is the starting configuration and for all $C_i = (q_i, \sigma_i)$ with $0 \leq i < n$, the following holds: there exist $\gamma \in \Gamma \cup \{\bot\}$ and $\gamma', \sigma \in \Gamma^*$ s.t. $\sigma_i = \gamma\sigma$ and $\sigma_{i+1} = \gamma'\sigma$ and $(q_i, \gamma) \xrightarrow{a_i} (q_{i+1}, \gamma')$. A run is accepting, if $q_n \in F$. Note that $\bot$ does always remain at the bottom of the stack.
It is clear that the transition function $\delta$ can equivalently be given as a relation $\delta' \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \times Q \times \Gamma^*$, where $(q, a, \gamma, q', \gamma') \in \delta'$ iff $(q', \gamma') \in \delta(q, a, \gamma)$. We may occasionally use this syntax for reasons of convenience.
Furthermore, we assume that $\delta$ has the restriction that the current stack is modified by a single application of $\delta$ exactly in one of the following three ways:

- the top stack symbol is deleted (called *pop*),
- the stack is left untouched (called *nop*),
- a stack symbol is placed on top of the (otherwise unchanged) stack (called *push*).

Note that by this restriction the stack height changes at most by one and at most the top stack symbol changes. Clearly, every $\delta$ can be transformed into this normal form by splitting up greater changes into several steps of the above form.

**Definition 8 (Language Recognition)** The language accepted by an NFA (PDA) $\mathcal{A}$ is defined as

$$\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* \mid \text{there exists an accepting run of } \mathcal{A} \text{ on } w\}.$$

**Definition 9 (Pushdown System)** A *pushdown system* (PDS) is the configuration graph of a PDA $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, q_0, F)$, i.e. an LTS $\mathcal{T} = (Q \times \Gamma^*, \rightarrow, \ell)$ with $(q, \gamma v) \xrightarrow{a} (q', wv)$ for some $v \in \Gamma^*$ if $(q', w) \in \delta(q, a, \gamma)$.

For a definition of an LTS see Def. 14. Note that PDS are infinite state systems in general. The standard theory defines at least two more kinds of automata, namely the *linear bounded automaton* (LBA) and the *Turing machine* (TM). But since we do never use these concepts explicitly in this work, we omit their definitions and do just rely on their existence. The reader is referred to [HU79] for further details.

The following theorem manifests the dualism between the concepts of grammar and automaton.

**Theorem 2 (c.f. [HU79])** For any formal language $L$,

- $L \in$ REG iff there exists an NFA $\mathcal{A}$, s.t. $\mathcal{L}(\mathcal{A}) = L$.
- $L \in$ CFL iff there exists a PDA $\mathcal{A}$, s.t. $\mathcal{L}(\mathcal{A}) = L$.
- $L \in$ CSL iff there exists an LBA $\mathcal{A}$, s.t. $\mathcal{L}(\mathcal{A}) = L$.
- $L \in$ RE iff there exists a TM $\mathcal{A}$, s.t. $\mathcal{L}(\mathcal{A}) = L$.

The abbreviations of the *deterministic* versions of the various automata types are preceeded by a "D", i.e. DFA, DPDA, DLBA and DTM. By convention, we use the acronym NFA instead of FA to make the nondeterminism explicit. In the same manner we denote the language classes recognised by the corresponding deterministic machine model with a "D" prefix, i.e. DREG, DCFL, DCSL and DRE.

**Theorem 3 (c.f. [HU79])**

$$\text{DREG} = \text{REG} \subsetneq \text{DCFL} \subsetneq \text{CFL} \subsetneq \text{DCSL} \subseteq \text{CSL} \subsetneq \text{DRE} = \text{RE}.$$

This section covers the standard theory of formal languages and automata used in this work. In the following sections some non-standard language classes are introduced.

### 2.1.2 Visibly Pushdown Languages

*Visibly pushdown automata* (VPA) were introduced by Alur and Madhusudan [AM04] in 2004 as a robust subclass of PDA which is still capable of modelling recursive program behaviour such as nested method calls and returns. Historically, they are generalisations of *simple-minded automata* (SMA) and *semi-simple-minded automata* (SSMA) which were defined in [HR93, HK99]. These classes of automata are all obtained by limiting the functionality of PDA. The definitions of SMA and SSMA were motivated by a search for classes of languages which could be used as recursive programs in PDL (see Sec. 2.2.6) specifications without rendering it undecidable. We refer to SML, SSML and VPL for the classes of languages recognisable by SMA, SSMA and VPA respectively.

The strongest restrictions are imposed by SMA, where every action of the automaton is completely determined by the input symbol, that is: the type of the operation performed

(*push*, *pop* or *nop*), the stack symbol placed on top of the stack upon a push operation and the next control state. SSMA generalise SMA by permitting a nondeterministic choice of the next control state and VPA finally do only choose the type of operation according to the input symbol.
This is achieved by partitioning the set of actions $\Sigma$ into three disjoint sets $\Sigma_{\mathtt{c}}, \Sigma_{\mathtt{i}}$ and $\Sigma_{\mathtt{r}}$ according to a **c**all, **i**nternal or **r**eturn action and performing a corresponding push, nop or pop operation on the stack.

**Definition 10 (Visibly Pushdown Automaton)** A *visibly pushdown automaton* (VPA) is a PDA $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, q_0, F)$, where

- $Q \cap \Gamma = \emptyset$,
- $\bot \in \Gamma$ is a distinguished symbol, called *stack bottom symbol*,
- $\Sigma = \Sigma_{\mathtt{c}} \dot\cup \Sigma_{\mathtt{i}} \dot\cup \Sigma_{\mathtt{r}}$,
- $\delta = \delta_{\mathtt{c}} \cup \delta_{\mathtt{i}} \cup \delta_{\mathtt{r}}$ with

$$\begin{array}{rcl} \delta_{\mathtt{c}} & \subseteq & Q \times \Sigma_{\mathtt{c}} \times (\Gamma \setminus \{\bot\}) \times Q, \\ \delta_{\mathtt{i}} & \subseteq & Q \times \Sigma_{\mathtt{i}} \times Q, \\ \delta_{\mathtt{r}} & \subseteq & Q \times \Sigma_{\mathtt{r}} \times \Gamma \times Q. \end{array} .$$

It is important to note that in contrast to a PDA, a VPA contains no $\epsilon$-transitions.

A VPA $\mathcal{A}$ is called *deterministic* (or a DVPA) if for all $q \in Q$, $a \in \Sigma$, $\gamma \in \Gamma$ we have $|\{(q', \gamma') : (q, a, \gamma', q') \in \delta_{\mathtt{c}}\}| = |\{q' : (q, a, q') \in \delta_{\mathtt{i}}\}| = |\{q' : (q, a, \gamma, q') \in \delta_{\mathtt{r}}\}| = 1$.
A *run* of $\mathcal{A}$ on a finite word $w = a_1 \ldots a_n$ is a sequence of configurations $C_0, C_1, \ldots, C_n$ with $C_i \in Q \times \Gamma^+$ for all $i = 0, \ldots, n$, s.t. $C_0 = (q_0, \bot)$ and for all $C_i = (q_i, \sigma_i)$ the following holds:

- If $a_i \in \Sigma_{\mathtt{c}}$ then there is a $\gamma$ s.t. $(q_i, a_i, \gamma, q_{i+1}) \in \delta_{\mathtt{c}}$ and $\sigma_{i+1} = \gamma\sigma_i$.
- If $a_i \in \Sigma_{\mathtt{i}}$ then $(q_i, a_i, q_{i+1}) \in \delta_{\mathtt{i}}$ and $\sigma_{i+1} = \sigma_i$.
- If $a_i \in \Sigma_{\mathtt{r}}$ then $(q_i, a_i, \bot, q_{i+1}) \in \delta_{\mathtt{r}}$ and $\sigma_{i+1} = \sigma_i = \bot$, or there is a $\gamma$ s.t. $(q_i, a_i, \gamma, q_{i+1}) \in \delta_{\mathtt{r}}$ and $\sigma_i = \gamma\sigma_{i+1}$.

Note that this definition entails that $\bot$ cannot be popped from the stack. It is however read and can be used to indicate that the stack is empty.

The definitions of accepting run and accepted language are identical to those definitions for a PDA. A *visibly pushdown language* (VPL) is a language which is accepted by some VPA.

**Example 1** The language $L = \{a^n b^n \mid n > 0\}$ is a VPL. Let $\Sigma_{\mathtt{c}} = \{a\}$, $\Sigma_{\mathtt{r}} = \{b\}$ and $\Sigma_{\mathtt{i}} = \emptyset$. Consider the VPA $\mathcal{A} = (\{q_0, q_1, q_2, q_3\}, \Sigma, \{A, \#\}, \delta, q_0, \{q_3\})$, where

$$\begin{aligned}\delta_{\mathtt{c}} &= \{(q_0, a, \#, q_1), (q_1, a, A, q_1)\},\\ \delta_{\mathtt{r}} &= \{(q_1, b, A, q_2), (q_2, b, A, q_2), (q_2, b, \#, q_3), (q_1, b, \#, q_3)\},\\ \delta_{\mathtt{i}} &= \emptyset.\end{aligned}$$

The automaton works as follows: on an input word $w \in L$, it first parses the $a$-sequence of length $n > 0$ and thereby produces the stack $A \dots A\#$, since every $a$ requires a push-operation. The $A \dots A$-prefix has length $n - 1$. Note that the kind of symbol which is pushed on the stack via $\delta_{\mathtt{c}}$-operations only depends on the control state and the input symbol.
After reading the first $b$, the control state changes to $q_2$, pops the top $A$ and repeats this as long as further $b$ are seen and the top stack symbol remains $A$. On the last $b$ finally the symbol # appears on top of the stack since it matches the first $a$ and after popping it, the automaton is in the final state $q_3$.
Note that if the input word $w$ is not in $L$ then the automaton eventually gets stuck which is very easily verified, because the automaton is deterministic.
In this fashion all kinds of Dyck-languages such as XML can be parsed. The opening tags are pushed on top of the stack while on closing tags the opening tags are popped.

**Example 2** Let $\Sigma = \{p, c, r\}$ with $p \in \Sigma_{\mathtt{c}}$, $c \in \Sigma_{\mathtt{r}}$, and $r \in \Sigma_{\mathtt{i}}$. Define a VPA $\mathcal{A} = (\{q_0, q_1, q_2\}, \Sigma, \{\bot, \gamma_0, \gamma\}, q_0, \delta, \{q_0\})$, where

$$\begin{aligned}\delta_{\mathtt{c}} &= \{(q_0, p, \gamma_0, q_1), (q_1, p, \gamma, q_1)\},\\ \delta_{\mathtt{r}} &= \{(q_0, c, \bot, q_2), (q_1, c, \gamma_0, q_0), (q_1, c, \gamma, q_1)\},\\ \delta_{\mathtt{i}} &= \{(q_0, r, q_0), (q_1, r, q_1)\}.\end{aligned}$$

Interpret $p$ as a *produce* action, $c$ as a *consume* and $r$ as a *request* in the setting of an automated production line. It is only legal to *consume* goods which have already been produced. The automaton specifies correct behaviour in this sense and rejects words which represent a violation (i.e. a stack underflow). It counts the *produce* actions by placing symbols onto the stack: a $\gamma_0$ for the first *produce* encountered and a $\gamma$ for the remaining

ones. On a *consume* action it removes $\gamma$ or $\gamma_0$ from the stack, the latter indicating that only one more *consume* is possible. If it sees a *consume* action and the stack is empty it switches into a non-final state which it never leaves again. We allow *request* actions anywhere between valid prefixes of words w.r.t. the stack underflow property. Hence, we have $L(\mathcal{A}) = \{w \in \Sigma^* \mid \ |w|_c = |w|_p \text{ and } |v|_c \leq |v|_p \text{ for all } v \preceq w\}$, where $\prec$ means the prefix relation.

VPL are capable of expressing many of the typical context-free languages, e.g. all kinds of Dyck-languages, but have a distinct advantage over CFL, namely their robustness. The following theorems substantiate the fact that VPL over finite words retain all the nice closure and determinisation properties from the regular languages.

**Theorem 4 (VPL Closure Properties, [AM04])** Let $L_1$ and $L_2$ be VPL w.r.t. a partitioned set of actions $\Sigma$ and let $R$ be a regular language. Then the following languages are VPL:

$$L_1 \cup L_2, \qquad L_1 \cap L_2, \qquad L_1 L_2, \qquad L_1^*, \qquad \overline{L_1}, \qquad L_1 \cap R.$$

**Theorem 5 (SML and SSML Closure Properties)** The classes SML and SSML are closed under intersections with regular languages.

Proof This can be shown by a simple product construction between a DFA and an SMA or SSMA, respectively. □

**Theorem 6 (Determinisation, [AM04])** Let $\mathcal{A}_1 = (Q, \Sigma, \Gamma, q_0, \delta, F)$ be a VPA w.r.t. a partitioned set of actions $\Sigma$. Then there exists a deterministic VPA $\mathcal{A}_2$, s.t. $L(\mathcal{A}_1) = L(\mathcal{A}_2)$ and $\mathcal{A}_2$ has at most $2^{|Q|^2}$ states and $2^{|Q|^2} \cdot |\Sigma_c|$ stack symbols.

From the above follows that the class of VPL can be embedded into the Chomsky Hierarchy as follows.

**Theorem 7**

$$\begin{array}{ccccccccccc} & & \text{SML} & & & & & & & & \\ & & \cap\!\!\!\!\!\!\nmid & & & & & & & & \\ \text{REG} & \subsetneq & \text{SSML} & \subsetneq & \text{VPL} & = & \text{DVPL} & \subsetneq & \text{DCFL} & \subsetneq & \text{CFL}. \end{array}$$

Proof Any DFA is clearly an SSMA without stack operations. Since DFA = NFA, we have REG $\subseteq$ SSML. Strictness of the inclusion follows from the fact that $\{a^n b^n \mid n > 0\}$ is an SML [HR93] (and the SML are included in SSML) but not a regular language cf. [HU79].
An SSMA is a generalisation of an SMA, hence SML $\subseteq$ SSML [HK99]. Strict inclusion follows from the property stated in [HK99] that there are only finitely many different SML over any given alphabet $\Sigma$ but infinitely many different REG and hence SSML. Intuitively, a DFA is *not* an SMA, because even its next state is solely determined by the input symbol and not by input symbol *and* current state. This makes in fact the expressivity of REG and SML incomparable.
The strict inclusion of SSML in VPL is stated in [LLS07].
Finally, since VPL are closed under determinisation by Thm. 6, they are all contained in DCFL. Strictness is witnessed by the language $\{a^n ba^n \mid n \geq 0\}$ which is easily seen to be a DCFL but is *not* an SML [HR93]. Note that the first $n$ $a$-symbols require a push-operation while the $a$s occurring behind the $b$ require pop-operations.
That DCFL is strictly included in CFL is a well-known standard theorem in formal language theory cf. [HU79]. □

**Theorem 8 (VPL Emptiness)** The emptiness-problem for VPL is PTIME-complete.

Proof Inclusion in PTIME is a consequence of the fact that the emptiness problem for CFL is in PTIME (c.f. [HU79]) and that VPL is included in CFL. A hardness proof can be found in [Lan10]. □

Since SML and SSML are both included in VPL, their emptiness problems are obviously also in PTIME.

**Corollary 1 (SML and SSML Emptiness)** The emptiness problem for SML and SSML is in PTIME.

### 2.1.3 Indexed Languages

The class of *indexed languages* (IL) was proposed in 1968 by Aho as a result of an increased interest in specification devices for all of the syntactic structures found in modern programming languages of that time – in particular ALGOL is mentioned – for which the CFL were too weak and the CSL were too powerful [Aho68]. Indeed, IL is located strictly

in between CFL and CSL and furthermore enjoys nice closure properties. IL are equally definable by a certain class of automata called *nested stack automata* as well as by a certain class of grammars called *indexed grammars* (IG). A nested stack automaton is a kind of pushdown automaton where the memory consists of nested stacks, i.e. the objects pushed and popped from the stack are stacks themselves. In addition, the automaton may read the contents of all of the stacks nested within itself.
We do only introduce in detail the latter characterisation via grammars since it is the one used in the following chapters explicitly. For further information on nested stack automata, the reader is referred to [Aho69].
The main difference to CFG is that nonterminals are equipped with a stack in an IG. This allows to constrain derivation rules according to the top stack symbol additionally. The stack symbols are called *indices*.

**Definition 11 (Indexed Grammar)** An IG is a 5-tuple $G = (N, \Sigma, I, P, S)$ where

- $N$ is a finite set of nonterminals,
- $\Sigma$ is a finite alphabet,
- $I$ is a finite set of index symbols,
- $S \in N$ is a distinguished starting symbol,
- $P$ is a finite set of productions of which there are the following four different types:

$$\begin{array}{lcl} \text{terminal productions} & : & A \to a,\ A \to \epsilon, \\ \text{composite productions} & : & A \to BC, \\ \text{push productions} & : & A \to B[f], \\ \text{pop productions} & : & A[f] \to B. \end{array}$$

Hence, $P \subseteq N \cup (N \times \Sigma) \cup (N \times N^2) \cup (N \times N \times I) \cup (N \times I \times N)$.

The three symbol sets $N$, $\Sigma$, and $I$ must be mutually disjoint.

In fact, the production rules in this definition are already in a normal form given by Aho (called *reduced form* there). However, every indexed grammar in Aho's original form can be transformed into one in normal form incurring a linear blow-up at most.
An *indexed nonterminal* is an element of $N \times I^*$, written $A[f_n \dots f_1]$ for example. The *index* $f_n \dots f_1$ forms a stack with its top on the left. The empty stack is allowed, i.e. $A[]$ is also an indexed nonterminal which we usually simply write as $A$.

A *sentential form* for an indexed grammar is a word over the alphabet $(N \times I^*) \cup \Sigma$, i.e. we have indexed nonterminals instead of arbitrary nonterminals, and index symbols may only occur in an index of a nonterminal.
The derivation relation $\Rightarrow$ on sentential forms of an indexed grammar is the least relation that satisfies the following for all sentential forms $\alpha, \beta, \gamma$, all indices $\delta \in I^*$, all index symbols $f \in I$, all nonterminals $A$, and all terminals $a$:

$$
\begin{array}{rcll}
A & \Rightarrow & \epsilon & ,\text{if } A \to \epsilon \\
A & \Rightarrow & a & ,\text{if } A \to a \\
A[\delta] & \Rightarrow & B[\delta]C[\delta] & ,\text{if } A \to BC \\
A[\delta] & \Rightarrow & B[f\delta] & ,\text{if } A \to B[f] \\
A[f\delta] & \Rightarrow & B[\delta] & ,\text{if } A[f] \to B \\
\alpha A[\delta]\beta & \Rightarrow & \alpha\gamma\beta & ,\text{if } A[\delta] \Rightarrow \gamma.
\end{array}
$$

It is important to observe that a nonterminal passes its index to anything that is derived from it in one step. Furthermore, terminal symbols cannot have indices. In principle one may regard an indexed grammar as a context-free grammar with an unbounded number of nonterminals, namely indexed nonterminals. The rules, however, can only distinguish finitely many different indexed nonterminals by operating on the top symbol of the index stack only.
As usual, $\Rightarrow^+$ and $\Rightarrow^*$ denote the transitive, resp. transitive-reflexive closure of the binary relation $\Rightarrow$, and $\Rightarrow^n$ for some $n \in \mathbb{N}$ denotes its $n$-fold self-composition. The language of an indexed grammar $G = (N, \Sigma, I, P, S)$ is, as usual $\mathcal{L}(G) = \{w \in \Sigma^* \mid S \Rightarrow^+ w\}$, where the stack of $S$ is empty.

**Example 3** Consider the language $L = \{a^{2^n} \mid n \geq 1\}$. It is generated by the indexed grammar $G = (\{A, S, T\}, \{a\}, \{\#, f\}, P, S)$ with $P$ given as

$$
\begin{array}{rclrclrcl}
S & \to & T[\#], & T & \to & T[f] \mid A, & A[f] & \to & AA, \\
A[\#] & \to & B, & B & \to & a. & & &
\end{array}
$$

A derivation of the word $a^8$ is:

$$
\begin{array}{lllll}
S & \Rightarrow T[\#] \Rightarrow T[n\#] & \Rightarrow & T[nn\#] \Rightarrow T[nnn\#] & \Rightarrow \\
A[nnn\#] & \Rightarrow A[nn\#]A[nn\#] & \Rightarrow^2 & A[n\#]A[n\#]A[n\#]A[n\#] & \Rightarrow^4 \\
\multicolumn{2}{l}{A[\#]A[\#]A[\#]A[\#]A[\#]A[\#]A[\#]A[\#]} & \Rightarrow^{16} & a^8. &
\end{array}
$$

Further examples can be seen in Sec. 2.1.4.

**Theorem 9 (Closure Properties, [Aho68])** Let $L_1$ and $L_2$ be IL and $R$ be a regular language. Then the following languages are IL:

$$L_1 \cup L_2, \qquad L_1 L_2, \qquad L_1^*, \qquad L_1 \cap R.$$

The class of IL is *not* closed under intersection and complement.

**Theorem 10 (Emptiness, [Aho68, TK07])** The emptiness-problem for IG is EXPTIME-complete.

### 2.1.4 Linear Indexed Languages

A *linear indexed grammar* (LIG) (originally defined by Gazdar [Gaz88]) is similar to an IG, but restricts the number of stacks propagated to the next sentential form during a derivation to one. In every production rule righthand side, one nonterminal is appointed to carry over the stack from the nonterminal on the lefthand side.

**Definition 12 (Linear Indexed Grammar)** A LIG is a 5-tuple $G = (N, \Sigma, I, P, S)$ in which all parts are defined identically to an IG, except for the composite production rules in $P$, where the stack inheritant on the righthand side is indicated by a marker. We use here a hat $\widehat{A}$ on top of the nonterminal $A$ to identify the stack inheritant. Hence, productions in a linear indexed grammar are of the following form:
Let $A, B, C \in N$, $a \in \Sigma$ and $f \in I$.

$$\begin{array}{lcl} \text{terminal productions} & : & A \to a,\ A \to \epsilon, \\ \text{composite productions} & : & A \to \widehat{B}C,\ A \to B\widehat{C}, \\ \text{push productions} & : & A \to B[f], \\ \text{pop productions} & : & A[f] \to B. \end{array}$$

A *marked (indexed) nonterminal* is a $\widehat{A[\delta]}$ for some $A \in N$ and some $\delta \in I^*$. An indexed nonterminal is, as above, a $A[\delta]$, and we write $A$ instead of $A[]$ again.
A *sentential form* of a linear indexed grammar is a sentential form in the usual sense, i.e. a word consisting of terminal symbols and indexed nonterminals, with the additional restriction, that at most one (indexed) nonterminal is marked.
The relation $\Rightarrow$ on such sentential forms is the least relation that satisfies the following for all sentential forms $\alpha, \beta, \gamma$, all indices $\delta \in I^*$, all index symbols $f \in I$, all nonterminals

$A, B, C$, and all terminal symbols $a$.

$$
\begin{array}{rclcrclc}
A & \Rightarrow & \epsilon & , & \widehat{A} & \Rightarrow & \epsilon & \text{,if } A \to \epsilon \\
A & \Rightarrow & a & , & \widehat{A} & \Rightarrow & a & \text{,if } A \to a \\
A[\delta] & \Rightarrow & B[\delta]C & , & \widehat{A[\delta]} & \Rightarrow & \widehat{B[\delta]}C & \text{,if } A \to \widehat{B}C \\
A[\delta] & \Rightarrow & BC[\delta] & , & \widehat{A[\delta]} & \Rightarrow & B\widehat{C[\delta]} & \text{,if } A \to B\widehat{C} \\
A[\delta] & \Rightarrow & B[f\delta] & , & \widehat{A[\delta]} & \Rightarrow & \widehat{B[f\delta]} & \text{,if } A \to B[f] \\
A[f\delta] & \Rightarrow & B[\delta] & , & \widehat{A[f\delta]} & \Rightarrow & \widehat{B[\delta]} & \text{,if } A[f] \to B \\
\alpha A[\delta]\beta & \Rightarrow & \alpha\gamma\beta & & & & & \text{,if } A[\delta] \Rightarrow \gamma \\
\alpha\widehat{A[\delta]}\beta & \Rightarrow & \alpha\gamma\beta & & & & & \text{,if } \widehat{A[\delta]} \Rightarrow \gamma.
\end{array}
$$

The last two rules are of course only applicable if $\alpha\gamma\beta$ is a valid sentential form again, i.e. contains at most one marked (indexed) nonterminal.

We remark that the definition of the derivation relation deviates from the original one in [Gaz88] insofar as it uses marked nonterminals simultaneously to unmarked ones. The original definition uses no markers. The use of markers is solely for technical reasons since some theorems later on need to track the stack inheritance from nonterminal to nonterminal through a derivation and to make this explicit. Note that by this definition there is for every derivation using markers a corresponding one without and vice versa but they do not get mixed up in the sense that either the currently derived sentential form has a marker on some nonterminal during every derivation step or during none. Note that in a derivation step $\alpha \Rightarrow \beta$, it is impossible for $\beta$ to contain a marked nonterminal while $\alpha$ does not. Hence, if $\widehat{S} \Rightarrow^+ w$ then $S$ can derive $w$ without markers in the derivation. If markers are present, however, then they trace the inheritance of a stack through sentential forms. In order to understand the language derivation mechanism of LIG it suffices to take the definition without markers (which corresponds to the one in [Gaz88]).

The language of a LIG $G$ is $L(G) := \{w \in \Sigma^* \mid S \Rightarrow^+ w\}$. By the above remark this means that the markers on indexed nonterminals in sentential forms are irrelevant for the language derived by a grammar.

**Example 4** Consider the language $L = \{a^n b^n c^n \mid n \geq 1\}$. It is generated by the linear indexed grammar

$$G = (\{S, S_{AC}, S_B, S_C, A, B, C, D\}, \{a, b, c\}, \{f\}, P, S),$$

where $P$ is given as

$$\begin{array}{rclrclrcl} S & \rightarrow & S_{AC}[f], & S_{AC} & \rightarrow & A\widehat{S_C}, & S_C & \rightarrow & \widehat{S}C \mid \widehat{S_B}C, \\ S_B[f] & \rightarrow & D, & D & \rightarrow & \widehat{S_B}B, & S_B & \rightarrow & \epsilon, \\ A & \rightarrow & a, & B & \rightarrow & b, & C & \rightarrow & c. \end{array}$$

A derivation of the word $a^2b^2c^2$ is:

$$\begin{array}{lclclclc} S & \Rightarrow & S_{AC}[f] & \Rightarrow & AS_C[f] & \Rightarrow & AS[f]C & \Rightarrow \\ AS_{AC}[ff]C & \Rightarrow & AAS_C[ff]C & \Rightarrow & AAS_B[ff]CC & \Rightarrow & AAD[f]CC & \Rightarrow \\ AAS_B[f]BCC & \Rightarrow & AADBCC & \Rightarrow & AAS_BBBCC & \Rightarrow^7 & aabbcc. & \end{array}$$

Again, there is a corresponding derivation $\widehat{S} \Rightarrow a^2b^2c^2$ but it exists solely for technical reasons and has no implications on the language derived by $G$.

LIL belong to the *mildly context-sensitive languages* (MCSL) and are equivalent to several on first glance very different grammar formalisms, namely *head grammars* (HG), *tree adjoining grammars* (TAG) and *combinatory categorical grammars* (CCG), giving rise to the language classes HL,TAL and CCL respectively [VsW94]. The following theorem shows their embedding into the Chomsky hierarchy.

**Theorem 11**

$$\text{CFL} \subsetneq \text{LIL} = \text{HL} = \text{TAL} = \text{CCL} \subsetneq \text{IL} \subsetneq \text{CSL}.$$

PROOF CFL are LIL with empty stacks and the strictness of the inclusion is witnessed by e.g. the language $\{a^nb^nc^n \mid n \geq 1\}$ which is a LIL but not a CFL [HU79]. As mentioned before, the equivalence of the four mildly context-sensitive formalisms is proved in [VsW94]. Their inclusion in IL is given by a rather simple translation: note that the composite production rules of LIL are the only ones in which LIL differ from IL. Now, in a production rule of the form $A \rightarrow \widehat{B}C$, $C$ is substituted by a fresh dummy nonterminal $C'$ (and of course the marker is erased). It is clear that we can add further production rules in which the stack content of $C'$ is popped until it is empty and further rules which transform $C'$ back to $C$ but with an empty stack now. This has the effect that the only way of eliminating $C'$ in a sentential form during a derivation is by emptying its stack and transforming it back into $C$ which exactly simulates the behaviour of the original LIL rule. The same holds of course for rules of the form $A \rightarrow B\widehat{C}$. Strictness is witnessed by the language $\{a^{2^i} \mid i \geq 0\}$ which is not a LIL but an IL [Aho68, Gaz88].
Finally, the strict inclusion of the class IL in CSL is shown again in [Aho68]. □

**Theorem 12 (Closure Properties, [VsW94])** Let $L_1$ and $L_2$ be LIL and $R$ be a regular language. Then the following languages are LIL:

$$L_1 \cup L_2, \qquad L_1 L_2, \qquad L_1^*, \qquad L_1 \cap R.$$

**Theorem 13 (Emptiness, [Bou96])** The emptiness-problem for LIL is PTIME-complete.

### 2.1.5 Alternating Context-Free Languages

Lange and Okhotin have independently defined two language generation devices called *alternating context-free grammar* (ACFG) [Lan02] and *conjunctive grammar* (CG) [Okh01], respectively, which have been proven equivalent [Okh01, ALLa]. For this reason we do only present one of them here. It should also be noted that the homonymous formalism defined by Moriya in [Mor89] is to be strictly distinguished from Lange's. Okhotin notes that CL are strictly included in Moriya's ACFL and hence so are Lange's ACFL.
Syntactically, ACFG and CG are exactly the same. They extend ordinary context-free grammars by partitioning their set of nonterminal symbols into existential and universal ones. The underlying idea states that a (sub-)word is derived from an existential nonterminal if *some* of its productions yield the word whereas it is derived from a universal nonterminal if *all* of its productions yield this word.
The two proposals contained different semantics for such grammars, though. Okhotin has explained the meaning of a conjunctive grammar by extending the derivation relation $\Rightarrow^*$ for context-free languages incorporating parallelism in order to implement the idea of universal productions. Lange has chosen a semantics for alternating context-free grammars that is an extension of the well-known parse tree formalism for context-free grammars.

**Definition 13 (Alternating Context-Free Grammar)** An ACFG is a tuple $G = (N, \Sigma, S, P, \lambda)$ where $N$ is a finite set of non-terminal symbols, $\Sigma$ is an alphabet disjoint from $N$, $S \in N$ is a designated starting symbol, and $P \subseteq N \times (N \cup \Sigma)^*$ is a finite set of production rules. Finally, $\lambda : N \to \{\exists, \forall\}$ labels the non-terminals as either existential or universal.

Let $\vdash_G$ be the smallest relation $\vdash_G \subseteq (N \cup \Sigma)^* \times \Sigma^*$ which is characterised by the following rules.

$$(\texttt{Ax})\ \frac{}{w \vdash_G w} \qquad (\texttt{And})\ \frac{\alpha_1 \vdash_G w \quad \dots \quad \alpha_n \vdash_G w}{A \vdash_G w} \ \text{ if } A \to \alpha_1 \,\&\, \dots \,\&\, \alpha_n$$

$$(\texttt{Or})\ \frac{\alpha_i \vdash_G w}{A \vdash_G w}\ \text{ if } A \to \alpha_1 \mid \ldots \mid \alpha_n \qquad (\texttt{Comp})\ \frac{\beta \vdash_G u \qquad \gamma \vdash_G v}{\beta\gamma \vdash_G uv}$$

The language derived from such a grammar is $\mathcal{L}(G) = \{w \in \Sigma^* \mid S \vdash_G w\}$.

**Example 5 (Okhotin [Okh01])** The grammar given by the following rules derives the language $\{wcw \mid w \in \{a,b\}^*\}$ over the alphabet $\Sigma = \{a,b,c\}$.

$$\begin{array}{ll}
S \to C \,\&\, D, & C \to aCa \mid aCb \mid bCa \mid bCb \mid c, \\
E \to aE \mid bE \mid \epsilon, & D \to aA \,\&\, aD \mid bB \,\&\, bD \mid cE, \\
A \to aAa \mid aAb \mid bAa \mid bAb \mid cEa, & B \to aBa \mid aBb \mid bBa \mid bBb \mid cEb.
\end{array}$$

Intuitively, $S$ derives the intersection of the languages derived by $C$ and $D$. $C$ generates $\{xcy \mid x,y \in \{a,b\}^*, |x| = |y|\}$. $D$ has the purpose to ensure that indeed every $a$ or $b$ positioned on the left of $c$ corresponds to the same terminal to the right of $c$ in the correct order. Note that $A$ and $B$ enforce an $a$ or $b$ respectively right of the $c$. The recursive intersection of $aA \,\&\, aD$ and $bB \,\&\, bD$ takes care of the positions in which the $a$'s and $b$'s occur. Formally, $D$ derives the language $\{wcxw \mid w,x \in \{a,b\}^*\}$ whose intersection with the language of $C$ indeed results in $\{wcw \mid w \in \{a,b\}^*\}$.
The derivation of the word $abcab$ is shown in Fig. 2.1.

**Theorem 14 (Closure Properties, [Okh01])** Let $L_1$ and $L_2$ be ACFL and $R$ be a regular language. Then the following languages are ACFL:

$$L_1 \cup L_2, \qquad L_1 \cap L_2, \qquad L_1L_2, \qquad L_1^*, \qquad L_1 \cap R.$$

It is currently not known whether ACFL are closed under complement.

The closure of ACFL under finite intersections with CFL can trivially be proved since ACFL have a direct means for intersection at hand. From this of course follows as a corollary that ACFL are closed under intersections with REG.

**Theorem 15 (Emptiness, [Okh01])** The emptiness-problem for ACFL is undecidable.

$$
\dfrac{\dfrac{}{a \vdash_G a}\,\texttt{(Ax)} \quad \dfrac{\dfrac{\dfrac{}{b \vdash_G b}\,\texttt{(Ax)} \quad \dfrac{\dfrac{\dfrac{}{c \vdash_G c}\,\texttt{(Ax)} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{a \vdash_G a}\,\texttt{(Ax)} \quad \dfrac{\dfrac{}{\epsilon \vdash_G \epsilon}\,\texttt{(Ax)}}{E \vdash_G \epsilon}\,\texttt{(Or)}}{aE \vdash_G a}\,\texttt{(Comp)}}{E \vdash_G a}\,\texttt{(Or)} \quad \dfrac{}{b \vdash_G b}\,\texttt{(Ax)}}{Eb \vdash_G ab}\,\texttt{(Comp)}}{cEb \vdash_G cab}\,\texttt{(Comp)}}{B \vdash_G cab}\,\texttt{(Or)}}{bB \vdash_G bcab}\,\texttt{(Comp)} \quad \dfrac{\dfrac{}{b \vdash_G b}\,\texttt{(Ax)} \quad \dfrac{\dfrac{\dfrac{}{c \vdash_G c}\,\texttt{(Ax)} \quad \dfrac{\dfrac{\dfrac{}{a \vdash_G a}\,\texttt{(Ax)} \quad \dfrac{\dfrac{\dfrac{}{b \vdash_G b}\,\texttt{(Ax)} \quad \dfrac{\dfrac{}{\epsilon \vdash_G \epsilon}\,\texttt{(Ax)}}{E \vdash_G \epsilon}\,\texttt{(Or)}}{bE \vdash_G b}\,\texttt{(Comp)}}{E \vdash_G b}\,\texttt{(Or)}}{aE \vdash_G ab}\,\texttt{(Comp)}}{E \vdash_G ab}\,\texttt{(Or)}}{cE \vdash_G cab}\,\texttt{(Comp)}}{D \vdash_G cab}\,\texttt{(Or)}}{bD \vdash_G bcab}\,\texttt{(Comp)}}{D \vdash_G bcab}\,\texttt{(And)}}{\text{①} \quad aD \vdash_G abcab}\,\texttt{(Comp)}
$$

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{a \vdash_G a}\,\texttt{(Ax)} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{b \vdash_G b}\,\texttt{(Ax)} \quad \dfrac{\dfrac{\dfrac{}{c \vdash_G c}\,\texttt{(Ax)}}{C \vdash_G c}\,\texttt{(Or)} \quad \dfrac{}{a \vdash_G a}\,\texttt{(Ax)}}{Ca \vdash_G ca}\,\texttt{(Comp)}}{bCa \vdash_G bca}\,\texttt{(Comp)}}{C \vdash_G bca}\,\texttt{(Or)} \quad \dfrac{}{b \vdash_G b}\,\texttt{(Ax)}}{Cb \vdash_G bcab}\,\texttt{(Comp)}}{aCb \vdash_G abcab}\,\texttt{(Comp)}}{C \vdash_G abcab}\,\texttt{(Or)} \quad \dfrac{\dfrac{\dfrac{}{a \vdash_G a}\,\texttt{(Ax)} \quad \dfrac{\dfrac{\dfrac{}{b \vdash_G b}\,\texttt{(Ax)} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{c \vdash_G c}\,\texttt{(Ax)} \quad \dfrac{\dfrac{\dfrac{}{\epsilon \vdash_G \epsilon}\,\texttt{(Ax)}}{E \vdash_G \epsilon}\,\texttt{(Or)} \quad \dfrac{}{a \vdash_G a}\,\texttt{(Ax)}}{Ea \vdash_G a}\,\texttt{(Comp)}}{cEa \vdash_G ca}\,\texttt{(Comp)}}{A \vdash_G ca}\,\texttt{(Or)} \quad \dfrac{}{b \vdash_G b}\,\texttt{(Ax)}}{Ab \vdash_G cab}\,\texttt{(Comp)}}{bAb \vdash_G bcab}\,\texttt{(Comp)}}{A \vdash_G bcab}\,\texttt{(Or)}}{aA \vdash_G abcab}\,\texttt{(Comp)} \quad \text{①}}{D \vdash_G abcab}\,\texttt{(And)}}{S \vdash_G abcab}\,\texttt{(And)}
$$

Figure 2.1: ACFG derivation of *abcab* for the grammar in Ex. 5.

## 2.2 Temporal Logics

### 2.2.1 Labeled Transition Systems

Temporal logics are often interpreted over *finite structures* which reflect *infinite behaviour*. Such a structure represents an abstract model of a program and describes its possible configurations and the computational steps leading from one configuration to another. By behaviour we mean the possible sequences of configurations and the computational steps between them. Since programs need not terminate and may run forever, this behaviour might be an infinite object. But because the behaviour is obtained by some form of unfolding of the structure it usually offers enough regularity to maintain decidability of verification tasks.

On the other hand there is system behaviour which cannot be described by finite structures in general, e.g. *pushdown systems* c.f. [BEM97]. These systems do necessarily display infinite behaviour and thereby increase the difficulty of maintaining decidability of verification. The existence of *finite representations* of such infinite structures remains however a minimum requirement for any verification task.

In compliance with the above requirements, we adopt here the standard definition of a *Labeled Transition System* (LTS) which serves as structure for all temporal logics discussed in this work.

**Definition 14 (Labeled Transition System)** Let $\Sigma$ be a finite set of actions and $\mathcal{P}$ be a finite set of atomic propositions. An LTS is a triple $\mathcal{T} = (\mathcal{S}, \rightarrow, \ell)$, where

- $\mathcal{S}$ is a set of states,
- $\rightarrow \subseteq \mathcal{S} \times \Sigma \times \mathcal{S}$ is called *transition relation*,
- $\ell : \mathcal{P} \rightarrow 2^{\mathcal{S}}$ is called *labeling function*.

Instead of writing $(s, a, t) \in \rightarrow$, we use infix notation $s \xrightarrow{a} t$. By abuse of notation, the transition relation $\rightarrow$ is extended to action sequences $\rightarrow \subseteq \mathcal{S} \times \Sigma^* \times \mathcal{S}$ inductively as

$$\begin{aligned} s \xrightarrow{\epsilon} t \quad &\text{iff} \quad s = t, \\ s \xrightarrow{aw} t \quad &\text{iff} \quad \exists u \in \mathcal{S} \text{ with } s \xrightarrow{a} u \text{ and } u \xrightarrow{w} t, \end{aligned}$$

where $\epsilon$ is the empty word and $w \in \Sigma^*$.

A *path* in an LTS $\mathcal{T} = (\mathcal{S}, \rightarrow, \ell)$ is a finite or infinite sequence of alternating states and actions $s_0, a_1, s_1, a_2, s_2, \ldots$, s.t. $s_i \xrightarrow{a_{i+1}} s_{i+1}$ for all $i \in \mathbb{N}$. We similarly write paths as $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \ldots$ A path $\pi$ is *maximal* if it is infinite or it ends in a state $s_n$, s.t. there is no $a \in \Sigma$ and $t \in \mathcal{S}$ with $s_n \xrightarrow{a} t$. The length of a finite path $\pi = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \ldots \xrightarrow{a_n} s_n$ is $|\pi| = n$. If $\pi$ is infinite we denote its length by $|\pi| = \infty$. Depending on the focus of interest, we may from time to time omit the states in a path and call the projection on the sequence of labels a path anyway or just project onto the sequence of states.

The *size* of an LTS $\mathcal{T}$, usually written $|\mathcal{T}|$, is defined as the number of states $|\mathcal{S}|$ of $\mathcal{T}$. If $\mathcal{T}$ has infinitely many states then we write $|\mathcal{T}| = \infty$.

A state of an LTS – or more precisely, the propositions which hold in it – represents a configuration of a program during execution while a transition between states marks an execution step. For instance, states may hold the program variable assignments and transitions be labeled with program statements if this is the desired level of abstraction.

The behaviour of a program is captured by paths through the LTS which represent single lines of possible executions from some given starting state. Note that we hereby implicitly have introduced a non-deterministic computational model.

**Definition 15 (Bisimulation)** A *bisimulation* on an LTS $\mathcal{T} = (\mathcal{S}, \rightarrow, \ell)$ is a symmetric binary relation $R \subseteq \mathcal{S} \times \mathcal{S}$ s.t. for all $(s, t) \in R$:

- $s \in \ell(p)$ iff $t \in \ell(p)$ for all $p \in \mathcal{P}$, and
- if there is an $a \in \Sigma$ and an $s' \in \mathcal{S}$ s.t. $s \xrightarrow{a} s'$ then there is a $t' \in \mathcal{S}$ s.t. $t \xrightarrow{a} t'$ and $(t, t') \in R$.

Two states $s, t$ are bisimilar, written $s \sim t$, iff there exists a bisimulation $R$ with $(s, t) \in R$.

We may also speak of bisimilar states w.r.t. two LTS $\mathcal{T}$ and $\mathcal{T}'$, with the obvious adjustments to the bisimulation relation. Given two root or starting states $s, s'$ of $\mathcal{T}$ and $\mathcal{T}'$, we may even say that two LTS are bisimilar if $s$ and $s'$ are bisimilar.

It is commonly agreed that the notion of observational behaviour of programs is equally captured by bisimilar program models. Hence, it is a desirable property of temporal logics *not* to distinguish between bisimilar models. See Def. 21 for a formal definition of this property.

### 2.2.2 Logic and Program Verification

In order to reason about program properties a specification language is needed in which such properties can be expressed. A *temporal logic* $\mathcal{L}$ is a formal language, i.e. a set of sentences called *formulas*. Any formula $\varphi \in \mathcal{L}$ describes a property of an LTS $\mathcal{T} = (\mathcal{S}, \rightarrow, \ell)$ in terms of the states in which the property holds. Thus the semantics of $\varphi$ is a subset of $\mathcal{S}$.

We will use two different kinds of formalisms to state that $\varphi$ holds in a state $s \in \mathcal{S}$ (that is $s$ *satisfies* $\varphi$). For variable-free logics we define a *satisfaction relation* $\models_{\mathcal{T}} \subseteq \mathcal{S} \times \mathcal{L}$ over states and formulas w.r.t. an LTS $\mathcal{T}$.

In case a logic has variables it is common practice to define a *semantics function* $[\![\cdot]\!]^{\mathcal{T}}_{\eta} : \mathcal{L} \rightarrow 2^{\mathcal{S}}$ instead, where $\eta$ is a function which interprets the free variables occurring in the formula. The semantics function $[\![\cdot]\!]^{\mathcal{T}}_{\eta}$ maps a formula to exactly those states in which it holds w.r.t. $\eta$. If it is clear which LTS is meant, we usually omit it and simply write $\models$ and $[\![\cdot]\!]_{\eta}$. For closed formulas (i.e. formulas in which no free variables occur), we may also omit $\eta$. The formalisms are interchangeable on closed formulas since we demand

$$s \models \varphi \quad \text{iff} \quad s \in [\![\varphi]\!],$$

from which follows

$$[\![\varphi]\!] \;=\; \{s \in \mathcal{S} \mid s \models \varphi\}.$$

We will occasionally use the symbol $\not\models$ to indicate that the relation $\models$ does *not* hold.

There are a series of desirable standard properties and decision problems regarding temporal logics. From a historical perspective, modal logicians were mostly interested in axiomatising a logic and hence in the validity problem. Since a formula of modal logic is valid iff its negation is unsatisfiable this equally attracts notice to the *satisfiability problem.* But also in the context of e.g. program synthesis – the automatic generation of executable computer programs from specifications of their behaviour – *decidability* of a logic is the main requirement.

**Definition 16 (Satisfiability)** A formula $\varphi$ of some temporal logic $\mathcal{L}$ is *satisfiable* iff there exists a model $\mathcal{T} = (\mathcal{S}, \rightarrow, \ell)$ and a state $s \in \mathcal{S}$ s.t. $s \models \varphi$.

**Definition 17 (Decidability)** A logic $\mathcal{L}$ is *decidable* iff its satisfiability problem is decidable.

With the dedication of logics as tools for computer system and program verification and the thereby triggered automatisation process of these tasks, the *model checking problem* became more and more important while for earlier and less expressive logics the problem was usually considered too trivial.

**Definition 18 (Model Checking Problem)** By the model checking problem, we mean the question whether given an LTS $\mathcal{T} = (\mathcal{S}, \rightarrow, \ell)$ a state $s \in \mathcal{S}$ and a formula $\varphi$ the statement $s \models \varphi$ indeed holds.

Note that model checking is usually easier to solve than validity or satisfiability, because for most temporal logics, model checking can be reduced to validity by describing the model with a succinct formula [Sch02].
Model checking is in this sense a synonym for *program verification*, since a program specification in the form of a logical formula is being verified on an abstract version of a program (given as an LTS). Decidability of a logic does also have an application in this area, namely to prove the consistency of a system specification: if a formula is unsatisfiable, it contains a contradiction and hence cannot have an implementation.
In this work, we are going to focus on model checking but also mention results on decidability, where known.

### 2.2.3 Computational Complexity

One of the most important questions related to the typical decision problems of a logic – such as the model checking and satisfiability problems – is about their computational complexity: determine a measure of used computational resources for solving the problem in terms of a function on the size of the input.
We assume familiarity with the concept of computational complexity and just recall a few very basic notional conventions. See [HU79] for details.

**Definition 19** Let $f(n)$ be a function. $\mathrm{DTIME}(f(n))$, $\mathrm{NTIME}(f(n))$, $\mathrm{DSPACE}(f(n))$ and $\mathrm{NSPACE}(f(n))$ denote the classes of languages that can be recognised by a deterministic, resp. non-deterministic Turing Machine in time, resp. space $f(n)$. This naturally lifts to classes $F$ of functions:

$$\mathrm{DTIME}(F) \;:=\; \bigcup_{f \in F} \mathrm{DTIME}(f),$$

$$\begin{aligned}
\text{NTIME}(F) &:= \bigcup_{f\in F} \text{NTIME}(f),\\
\text{DSPACE}(F) &:= \bigcup_{f\in F} \text{DSPACE}(f),\\
\text{NSPACE}(F) &:= \bigcup_{f\in F} \text{NSPACE}(f).
\end{aligned}$$

Let $2_0^{f(n)} = f(n)$ and $2_{k+1}^{f(n)} = 2^{2_k^{f(n)}}$. Define some important complexity classes mentioned in the following as

$$\begin{aligned}
k\text{EXPTIME} &:= \text{DTIME}(\{2_k^{p(n)} \mid p(n) \text{ polynomial}\}),\\
\text{EXPTIME} &:= 1\text{EXPTIME},\\
\text{PTIME} &:= 0\text{EXPTIME},\\
\text{LINTIME} &:= \text{DTIME}(\{c \cdot n \mid c \text{ constant}\}),\\
\text{NPTIME} &:= \text{NTIME}(\{p(n) \mid p(n) \text{ polynomial}\}),\\
\text{co} - \text{NPTIME} &:= \{L \mid \overline{L} \in \text{NPTIME}\},\\
\text{PSPACE} &:= \text{DSPACE}(\{p(n) \mid p(n) \text{ polynomial}\}),\\
\text{ELEMENTARY} &:= \bigcup_{k\in\mathbb{N}} k\text{EXPTIME},
\end{aligned}$$

for any $k \in \mathbb{N}$.

**Theorem 16 (cf. [HU79])**

LINTIME $\subsetneq$ PTIME $\subseteq$ PSPACE $\subseteq$ EXPTIME $\subsetneq$ 2EXPTIME $\subsetneq \ldots \subsetneq$ ELEMENTARY.

It is not known which of the inclusions between PTIME and EXPTIME is strict, only that PTIME $\subsetneq$ EXPTIME.

### 2.2.4 Properties of Temporal Logics

Regarding the above decision problems, there are some useful properties and problems related which will be investigated for all of the logics occurring here.

**Definition 20 (Finite Model Property)** A logic $\mathcal{L}$ has the *finite model property* iff for all $\varphi \in \mathcal{L}$ we have that if $\varphi$ is satisfiable then there exists a finite model for $\varphi$.

Settling the question whether a logic has the finite model property allows to use techniques such as filtration in order to establish decidability. Note that if a logic has the finite model property, its model checking problem is decidable and it is bounded w.r.t. the formula then decidability is entailed, because it suffices to check all models up to the size of the boundary.

**Definition 21 (Bisimulation-invariance)** Let $\mathcal{T} = (\mathcal{S}, \rightarrow, \ell)$ and $\mathcal{T}' = (\mathcal{S}', \rightarrow', \ell')$ be LTS, $s \in \mathcal{S}$ and $s' \in \mathcal{S}'$ such that $s \sim s'$ (see Def. 15). That a logic $\mathcal{L}$ is *bisimulation-invariant* means that for any $\varphi \in \mathcal{L}$, we have $s \models \varphi$ iff $s' \models \varphi$.

Most modal and temporal logics are bisimulation-invariant and therefore do not distinguish models which are equivalent in this sense. This is of course a reasonable assumption in the context of program verification, since it comprises exactly the kind of abstraction which makes modal logics so attractive for specifying program behaviour: state-basedness and control flow simulation.
Another important aspect is that bisimulation-invariance entails the *tree model property.*

**Definition 22 (Tree Model Property)** A logic $\mathcal{L}$ has the *tree model property* iff for all $\varphi \in \mathcal{L}$ we have that if $\varphi$ is satisfiable then there exists a tree model for $\varphi$.

**Theorem 17** Any bisimulation-invariant logic does also exhibit the tree model property.

For a proof see cf. [Ott06]. A very useful application of the tree model property is that it allows to combine the theory of tree automata with program reasoning, see c.f. [VW86].

### 2.2.5 Expressivity

Given two different logics $\mathcal{L}_1$ and $\mathcal{L}_2$ it is natural to ask whether all properties expressible in $\mathcal{L}_1$ are also expressible in $\mathcal{L}_2$ and vice versa.

**Definition 23 (Expressivity Order)** Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be logics. $\mathcal{L}_2$ is said to be at least as expressive as $\mathcal{L}_1$, written $\mathcal{L}_1 \leq \mathcal{L}_2$ if there exists a $\psi \in \mathcal{L}_2$ such that for all LTS $\mathcal{T} = (\mathcal{S}, \rightarrow, \ell)$, $s \in \mathcal{S}$ and $\varphi \in \mathcal{L}_1$ we have $s \models_{\mathcal{T}} \varphi$ iff $s \models_{\mathcal{T}} \psi$. We write

$$\begin{aligned}
\mathcal{L}_1 &\equiv \mathcal{L}_2 && \text{,if } \mathcal{L}_1 \leq \mathcal{L}_2 \text{ and } \mathcal{L}_2 \leq \mathcal{L}_1, \\
\mathcal{L}_1 &\not\leq \mathcal{L}_2 && \text{,if not } \mathcal{L}_1 \leq \mathcal{L}_2, \\
\mathcal{L}_1 &\lneq \mathcal{L}_2 && \text{,if } \mathcal{L}_1 \leq \mathcal{L}_2 \text{ and } \mathcal{L}_2 \not\leq \mathcal{L}_1.
\end{aligned}$$

In order to emphasize on the size of the translation, we sometimes write $\varphi \leq_{f(x)} \psi$, $\varphi \equiv_{f(x)} \psi$, etc. to additionally require that $|\psi| \leq f(|\varphi|)$. If we are only concerned with the asymptotic behaviour, we write $\mathsf{lin}, \mathsf{exp}$, etc. instead of $f(n)$.

### 2.2.6 Propositional Dynamic Logic

*Propositional Dynamic Logic* (PDL) was originally introduced by Fischer and Ladner [FL79] in order to allow reasoning about programs. It describes the interactions of programs and logical propositions independently of the computation domain. PDL allows, for example, to make assertions of the kind "after executing program $\alpha$ in a state satisfying $\varphi$, property $\psi$ necessarily holds". Programs are built from atomic ones using the operations composition, nondeterministic choice and iteration. They are denoted by regular expressions. This makes the original PDL in effect a *PDL over regular programs.*

**Definition 24 (Propositional Dynamic Logic)** Let $\mathcal{P}$ be a finite set of propositions and $\Sigma$ be a finite set of actions. Formulas and programs of PDL are defined mutually recursive as the least sets $\mathsf{Form}$ and $\mathsf{Prog}$ respectively, satisfying the following conditions:

- $\mathcal{P} \subseteq \mathsf{Form}$.
- If $\varphi \in \mathsf{Form}$ then $\neg\varphi \in \mathsf{Form}$.
- If $\varphi, \psi \in \mathsf{Form}$ then $\varphi \vee \psi \in \mathsf{Form}$.
- If $\varphi \in \mathsf{Form}$ and $\alpha \in \mathsf{Prog}$ then $\langle\alpha\rangle\varphi \in \mathsf{Form}$.
- $\Sigma \subseteq \mathsf{Prog}$.
- If $\alpha, \beta \in \mathsf{Prog}$ then $\alpha\,\beta, \alpha \cup \beta$ and $\alpha^* \in \mathsf{Prog}$.
- If $\varphi \in \mathsf{Form}$ then $\varphi? \in \mathsf{Prog}$.

For notational convenience, we use the following standard abbreviations:

$$\begin{aligned}
\mathtt{tt} &:= q \vee \neg q, \\
\mathtt{ff} &:= \neg\mathtt{tt}, \\
\varphi \wedge \psi &:= \neg(\neg\varphi \vee \neg\psi), \\
\varphi \rightarrow \psi &:= \neg\varphi \vee \psi, \\
\varphi \leftrightarrow \psi &:= (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi), \\
[\alpha]\varphi &:= \neg\langle\alpha\rangle\neg\varphi.
\end{aligned}$$

All these abbreviations except the last are standard from *propositional logic* and will be referred to as *boolean* or *propositional* formulas. $\langle\cdot\rangle$ and $[\cdot]$ will be called *modal operators* or *modalities*. We call any program $\varphi?$ with $\varphi \in \mathsf{Form}$ a *test*.
PDL formulas and programs are interpreted over LTS models. The semantics of a PDL formula and a PDL program is given by simultaneous induction on the structure of the formula and the program: Let $\mathcal{T} = (\mathcal{S}, \rightarrow, \ell)$ be an LTS, $s, t \in \mathcal{S}$, $q \in \mathcal{P}$, $a \in \Sigma$, $\alpha, \beta \in \mathsf{Prog}$, and $\varphi, \psi \in \mathsf{Form}$. By abuse of notation we define

$$
\begin{aligned}
s \xrightarrow{\alpha\,\beta} t &\quad \text{iff} \quad \text{there exists } u \in \mathcal{S} \text{ s.t. } s \xrightarrow{\alpha} u \text{ and } u \xrightarrow{\beta} t,\\
s \xrightarrow{\alpha\cup\beta} t &\quad \text{iff} \quad s \xrightarrow{\alpha} t \text{ or } s \xrightarrow{\beta} t,\\
s \xrightarrow{\alpha^*} t &\quad \text{iff} \quad \text{there exists } n \in \mathbb{N}, u_0, \ldots, u_n \in \mathcal{S} \text{ s.t.}\\
&\quad\phantom{\text{iff}}\quad u_0 = s \text{ and } u_n = t \text{ and } u_i \xrightarrow{\alpha} u_{i+1} \text{ for all } 0 \le i < n,\\
s \xrightarrow{\varphi?} t &\quad \text{iff} \quad s = t \text{ and } s \models \varphi,
\end{aligned}
$$

$$
\begin{aligned}
s \models q &\quad \text{iff} \quad q \in \ell(s),\\
s \models \neg\varphi &\quad \text{iff} \quad s \not\models \varphi,\\
s \models \psi \vee \varphi &\quad \text{iff} \quad s \models \psi \text{ or } s \models \varphi,\\
s \models \langle\alpha\rangle\varphi &\quad \text{iff} \quad \text{exists } t \in \mathcal{S} \text{ s.t. } t \models \varphi \text{ and } s \xrightarrow{\alpha} t.
\end{aligned}
$$

**Example 6** The formula $\langle(\varphi?;\alpha) \cup ((\neg\varphi)?;\beta)\rangle\mathtt{tt}$ is satisfied in some state $s$ if either $\varphi$ holds in $s$ and a path labeled with program $\alpha$ exists or if $\varphi$ does not hold in $s$ and a path labeled with program $\beta$ exists.
Therefore the program used in the modality can be used to model conditional branching `if` $\varphi$ `then` $\alpha$ `else` $\beta$.

**Example 7** Consider the formula $[\alpha]p \leftrightarrow [\beta]p$ for two programs $\alpha$ and $\beta$ and a proposition $p$. This formula states the equivalence of the programs $\alpha$ and $\beta$ on a given structure. If this formula holds independently of the structure then clearly $\alpha \equiv \beta$.

**Theorem 18 (c.f. [HS96])** PDL exhibits the following properties:

- finite model property,
- bisimulation-invariance,

- tree model property.

**Theorem 19 ([FL79, Pra80])** The satisfiability problem for PDL is EXPTIME-complete.

**Theorem 20 ([FL79])** The model checking problem for PDL is PTIME-complete.

### 2.2.7 Computation Tree Logic

*Computation Tree Logic* (CTL) by Emerson and Clarke [CE81] is a widely used branching time logic which emerged from a proposal of Ben-Ari, Manna and Pnueli in 1981 called Unified Branching Time Logic and essentially is CTL without binary temporal operators but just EF and AG instead [BAMP81]. CTL has shown itself to be very useful in the design, specification and automatic verification of reactive and concurrent systems [MP92]. It has a distinct advantage over PDL, since it is capable of expressing a typical correctness specification statement like "all executions of a program will eventually reach a state in which property $\varphi$ holds" which is impossible in PDL.

**Definition 25 (Computation Tree Logic)** Let $\mathcal{P}$ be a countably infinite set of propositions. CTL is the following set of formulas:

$$\varphi \quad ::= \quad q \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathtt{EX}\varphi \mid \mathtt{E}(\varphi \mathtt{U} \varphi) \mid \mathtt{E}(\varphi \mathtt{R} \varphi)$$

where $q \in \mathcal{P}$.

Standard abbreviations include the propositional abbreviations $\mathtt{tt}, \mathtt{ff}, \wedge, \rightarrow, \leftrightarrow$ defined precisely as for PDL and the following:

$$\begin{array}{lcl}
\mathtt{A}(\varphi \mathtt{U} \psi) & := & \neg\mathtt{E}(\neg\varphi \mathtt{R} \neg\psi), \\
\mathtt{A}(\varphi \mathtt{R} \psi) & := & \neg\mathtt{E}(\neg\varphi \mathtt{U} \neg\psi), \\
\mathtt{AX}\varphi & := & \neg\mathtt{EX}\neg\varphi, \\
\mathtt{EF}\varphi & := & \mathtt{E}(\mathtt{tt} \mathtt{U} \varphi), \\
\mathtt{AF}\varphi & := & \mathtt{A}(\mathtt{tt} \mathtt{U} \varphi), \\
\mathtt{EG}\varphi & := & \mathtt{E}(\mathtt{ff} \mathtt{R} \varphi), \\
\mathtt{AG}\varphi & := & \mathtt{A}(\mathtt{ff} \mathtt{R} \varphi).
\end{array}$$

CTL formulas are interpreted in states of an LTS $\mathcal{T} = (\mathcal{S}, \rightarrow, \ell)$ as follows:

$$
\begin{array}{lll}
s \models q & \text{iff} & q \in \ell(s), \\
s \models \neg\varphi & \text{iff} & s \not\models \varphi, \\
s \models \varphi \vee \psi & \text{iff} & s \models \varphi \text{ or } s \models \psi, \\
s \models \mathtt{EX}\varphi & \text{iff} & \text{there exist } a \in \Sigma, t \in \mathcal{S} \text{ s.t. } s \xrightarrow{a} t \text{ and } t \models \varphi, \\
s \models \mathtt{E}(\varphi\mathtt{U}\psi) & \text{iff} & \text{there exists a path } \pi = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} s_n \\
& & \text{s.t. } s_0 = s \text{ and } s_n \models \psi \text{ and for all } i < n : s_i \models \varphi, \\
s \models \mathtt{E}(\varphi\mathtt{R}\psi) & \text{iff} & \text{there exists a maximal path } \pi = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots \\
& & \text{s.t. } s_0 = s \text{ and for all } i \leq |\pi| : \\
& & s_i \models \psi \text{ or there exists } j < i \text{ s.t. } s_j \models \varphi.
\end{array}
$$

Note that the semantics of CTL formulas is usually given over unlabeled transition systems since the labels are ignored anyway and that it is usually required that the transition system is total. We have chosen our definitions under the aspect of comparability between different kinds of logics and therefore wish to have a common and most general semantical base for both modal and temporal logics. This is important in particular with regard to the later on introduced non-regular variants of CTL which do respect the labels. It is important to note that on total transition systems, our definition of CTL semantics coincides with the classical one, i.e. formulas hold in exactly the same states. The same is true if a property is satisfied in a finite prefix of a path, i.e. for all $\mathtt{EU}$ formulas and also for those $\mathtt{E}(\varphi\mathtt{R}\psi)$-formulas which are satisfied because $\varphi$ holds somewhere along the path. The crucial case is the remaining one: what if there exists a finite path along which $\psi$ holds everywhere, but $\varphi$ nowhere? This case is undefined in classical CTL.

Since the main interest here is that the $\mathtt{R}$-operator is the dual to $\mathtt{U}$, we chose to define that such a finite path satisfies $\mathtt{E}(\varphi\mathtt{R}\psi)$. Another reason is that in order to ensure complete agreement between this version of CTL and the classical one, it suffices to add the formula $\mathtt{AGEXtt}$ as a conjunct to each formula, because it will render each formula to $\mathtt{ff}$ on a non-total LTS.

CTL has enrichments such as CTL* [EH86] which allow free mixing of path operators and quantifiers: for example, $\mathtt{A}(p\mathtt{UG}q)$ is a CTL* formula but not a CTL formula, because the $\mathtt{G}$ is not immediately preceeded by a path quantifier. In fact, CTL* unifies CTL and Pnueli's well-known linear time temporal logic LTL.

**Example 8** Typical CTL definable properties include *liveness* of property $\psi$, expressed as $\varphi = \mathtt{AGEF}\psi$. The formula $\varphi$ states "on all paths at any moment there exists a path on which $\psi$ eventually holds".

**Example 9** Dualising the path quantifiers and temporal operators yields the formula $\varphi = \mathtt{EFAG}\psi$ which states "there exists a path on which eventually on all paths at every moment $\psi$ holds".

**Theorem 21 ([EH85])** CTL exhibits the following properties:

- finite model property,
- bisimulation-invariance,
- tree model property.

Regarding the decision problems for CTL we have the following:

**Theorem 22 ([FL79],[EH85])** The satisfiability problem for CTL is EXPTIME-complete.

**Theorem 23 (c.f. [Sch02])** The model checking problem for CTL is PTIME-complete.

Comparing the expressivity of PDL and CTL it can easily be seen that they are mutually incomparable, because CTL is blind to transition labels on the one hand and PDL cannot express the `EG`-operator for instance.

**Theorem 24**

$$\text{PDL} \not\leq \text{CTL} \qquad \text{and} \qquad \text{CTL} \not\leq \text{PDL}.$$

For a proof see Thm. 48.

### 2.2.8 The Modal $\mu$-Calculus

Kozen's modal $\mu$-calculus ($\mathcal{L}_\mu$) [Koz82] extends modal logic with extremal fixpoint quantifiers. Regarding expressivity, it subsumes most of the commonly used modal and temporal logics.

**Definition 26 (Modal $\boldsymbol{\mu}$-Calculus)** Let $\mathcal{P}$ be a countably infinite set of propositions, $\Sigma$ be a finite set of actions and $\mathcal{V}$ be a countably infinite set of monadic second-order variables. Formulas of $\mathcal{L}_\mu$ are given by the following grammar.

$$\varphi \ ::= \ q \mid X \mid \neg\varphi \mid \varphi \vee \varphi \mid \langle a \rangle \varphi \mid \mu X.\varphi$$

where $a \in \Sigma$, $q \in \mathcal{P}$ and $X \in \mathcal{V}$ and the positivity requirement holds: in every subformula of $\mu X.\varphi$, every occurrence of $X$ must be under an even number of negation symbols.

The positivity requirement has the purpose of ensuring the existence of the fixpoint. We write $\varphi[\psi/X]$ for the formula produced by replacing every free occurrence of the variable $X$ in $\varphi$ with $\psi$.

Standard abbreviations include the propositional abbreviations $\mathtt{tt}, \mathtt{ff}, \wedge, \rightarrow, \leftrightarrow$ defined precisely as for PDL and the following:

$$\begin{array}{lcl} \langle - \rangle\varphi & := & \bigvee_{a\in\Sigma}\langle a\rangle\varphi, \\ [-]\varphi & := & \neg\langle - \rangle\neg\varphi, \\ [a]\varphi & := & \neg\langle a\rangle\neg\varphi, \\ \nu X.\varphi & := & \neg\mu X.\neg\varphi[\neg X/X]. \end{array}$$

The replacement of $X$ with $\neg X$ in the definition of $\nu X.\varphi$ ensures that $X$ occurs under the same number of negation symbols in the resulting formula.

The semantics of a $\mathcal{L}_\mu$ formula in a transition system $\mathcal{T} = (\mathcal{S}, \rightarrow, \ell)$ is a subset of $\mathcal{S}$, intuitively those states in which $\varphi$ holds. It is defined inductively using an environment $\rho : \mathcal{V} \rightarrow 2^{\mathcal{S}}$ that interprets free variables in a formula. We write $\rho[X \mapsto T]$ for the environment that maps the variable $X$ to the state set $T$ and behaves like $\rho$ otherwise.

$$\begin{array}{rcl} [\![q]\!]^{\mathcal{T}}_{\rho} & := & \{s \in \mathcal{S} \mid q \in \ell(s)\}, \\ [\![X]\!]^{\mathcal{T}}_{\rho} & := & \rho(X), \\ [\![\neg\varphi]\!]^{\mathcal{T}}_{\rho} & := & \mathcal{S} \setminus [\![\varphi]\!]^{\mathcal{T}}_{\rho}, \\ [\![\varphi \vee \psi]\!]^{\mathcal{T}}_{\rho} & := & [\![\varphi]\!]^{\mathcal{T}}_{\rho} \cup [\![\psi]\!]^{\mathcal{T}}_{\rho}, \\ [\![\langle a\rangle\varphi]\!]^{\mathcal{T}}_{\rho} & := & \{s \in \mathcal{S} \mid \exists t \in \mathcal{S}. s \xrightarrow{a} t \text{ and } t \in [\![\varphi]\!]^{\mathcal{T}}_{\rho}\}, \\ [\![\mu X.\varphi]\!]^{\mathcal{T}}_{\rho} & := & \bigcap\{T \subseteq \mathcal{S} \mid [\![\varphi]\!]^{\mathcal{T}}_{\rho[X\mapsto T]} \subseteq T\}. \end{array}$$

**Example 10** Consider the CTL formulas $\mathtt{E}(p\mathtt{U}q)$ and $\mathtt{E}(p\mathtt{R}q)$ for propositions $p, q \in \mathcal{P}$. They are expressed in $\mathcal{L}_\mu$ as

$$\begin{array}{l} \mu X.q \vee (p \wedge \langle - \rangle X) \qquad \text{and} \\ \nu X.q \wedge (p \vee \langle - \rangle X \vee [-]\mathtt{ff}) \end{array}$$

respectively. Note that this scheme in principle suffices to translate CTL to $\mathcal{L}_\mu$ as follows:

$$\begin{array}{rcl} \mathsf{tr}(q) & = & q, \\ \mathsf{tr}(\neg\varphi) & = & \neg\mathsf{tr}(\varphi), \end{array}$$

$$\begin{aligned}
\mathtt{tr}(\varphi \vee \psi) &= \mathtt{tr}(\varphi) \vee \mathtt{tr}(\psi), \\
\mathtt{tr}(\mathtt{EX}\varphi) &= \langle - \rangle \mathtt{tr}(\varphi), \\
\mathtt{tr}(\mathtt{E}(\varphi \mathtt{U} \psi) &= \mu X.\mathtt{tr}(\psi) \vee (\mathtt{tr}(\varphi) \wedge \langle - \rangle X), \\
\mathtt{tr}(\mathtt{E}(\varphi \mathtt{R} \psi) &= \nu X.\mathtt{tr}(\psi) \wedge (\mathtt{tr}(\varphi) \vee \langle - \rangle X \vee [-]\mathtt{ff}).
\end{aligned}$$

**Theorem 25 ([Koz88], c.f. [BS06])** $\mathcal{L}_\mu$ exhibits the following properties:

- finite model property,
- bisimulation-invariance,
- tree model property.

**Theorem 26 ([FL79],[EJ00])** The satisfiability problem for $\mathcal{L}_\mu$ is EXPTIME-complete.

The lower bound in Thm. 26 is a consequence of the EXPTIME-hardness of PDL satisfiability and the fact that PDL is a fragment of $\mathcal{L}_\mu$.

**Theorem 27 ([EJ88])** The model checking problem for $\mathcal{L}_\mu$ is PTIME-hard and included in NPTIME $\cap$ co $-$ NPTIME.

As stated in the introduction, the importance of $\mathcal{L}_\mu$ for this work is that it expresses exactly the regular properties on words and trees modulo bisimilarity and therefore separates the notions of regular and non-regular logics.
See Fig. 2.2 for an overview of the expressivity results for $\mathcal{L}_\mu$ and some of the most common temporal logics. A dotted line from a lower positioned logic $\mathcal{L}_1$ to a higher positioned one $\mathcal{L}_2$ stands for $\mathcal{L}_1 \lneq \mathcal{L}_2$. MSO/bis is used for the bisimulation-invariant fragment of MSO.

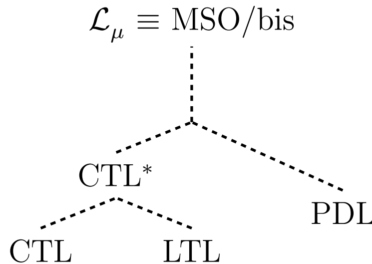



Figure 2.2: Expressive power of some regular logics.

### 2.2.9 Non-Regular Logics

$\mathcal{L}_\mu$ is exactly as expressive as the bisimulation-invariant fragment of *Monadic Second-Order Logic* (MSO) over trees or LTS [JW96]. MSO is the fragment of Second-Order Logic which restricts the use of second-order variables to arity 1, thus allowing to reason about sets of elements of some kind, e.g. states.

Since MSO and Rabin tree automata are also equivalent [Rab69], every property that is expressible in $\mathcal{L}_\mu$ (or one of its fragments PDL, CTL, CTL*, etc.) can also be checked by a finite Rabin tree automaton. The class of languages recognisable by finite automata are the regular languages – or $\omega$-regular languages in case the considered structures are infinite. It is in this sense that $\mathcal{L}_\mu$-definable properties are *regular* and the reason why we call $\mathcal{L}_\mu$ and its sublogics *regular logics*.

The classification of a temporal logic as regular is a statement about its expressive power and refers to the structurally least complex class of formal languages of the Chomsky hierarchy. Clearly, there is a large, almost unexplored space above $\mathcal{L}_\mu$ in terms of non-regular definable properties dual to the space above the regular languages in the Chomsky hierarchy. Non-regular program properties arise naturally in the context of unbounded data structures: for instance can the absence of buffer underflows not be expressed in $\mathcal{L}_\mu$ for unbounded buffers. Also any kind of counting properties like "at any point during the execution of a protocol there have never been more send- than receive-actions" are non-regular. Further examples include Emerson's uniform inevitability stating "some event occurs globally at the same time in all possible runs" [Eme87] or properties making structural assertions about their models like being bisimilar to a balanced tree or word.

This work contains numerous examples of such properties. We will introduce several logics that are capable of expressing such properties, establish basic properties about them, compare them by expressive power and – most important here – determine the computational complexity of their model checking problems.

# Chapter 3

# Non-Regular Propositional Dynamic Logic

The clear distinction between logic and programs in PDL comprises an appealing modularity for the purpose of defining non-regular program properties, namely by enriching the class of allowed programs in modal formulas. This idea is not new altogether: already the earliest works on PDL have dealt with questions regarding such extensions. They were, however, mostly concerned with decidability issues which is probably the reason why the range of considered classes has so far been limited to those located in between the regular and context-free ones, since this is where the borderline to undecidability runs.

## 3.1 Syntax and Semantics

In the following, we define PDL over different classes of formal languages $\mathcal{L}$, or PDL[$\mathcal{L}$] for short. The basic building mechanism of formulas in PDL[$\mathcal{L}$] is very similar to that of PDL over regular programs, except that the programs allowed in the modalities are not restricted to regular expressions but instead to languages $L \in \mathcal{L}$. This raises the question about the representation of such languages.

We do not want to artificially restrict the use of specification formalisms for formal languages of which there are numerous: e.g. automata, grammars, algebraic expressions, systems of equations, etc. On the other hand we may not omit all restrictions since our results do not hold for every kind of language representation, e.g. for extensional or otherwise infinite representations or cryptographically encrypted languages. The least restrictive format we identify in order to ensure the validity of our results is to assume a size measure $|L|$ for

any representation of a language $L$ which is a finite value, even though $L$ may of course contain infinitely many words. We identify any class of languages $\mathcal{L}$ with the class of a certain kind of finite representations of its members. For instance the class REG may be identified with the class NFA since NFA=REG and nondeterministic finite automata are finite representations of regular languages.
We make another very reasonable assumption on each $L$: given an $L \in \mathcal{L}$, its alphabet must be computable in time $\mathcal{O}(|L|)$. This is not a very strong assumption since it holds for virtually all formalisms typically used in this context and in particular for those mentioned above. But it does prevent the use of inadequate language representations such as encrypted languages.
PDL over regular programs is defined using tests. A test is a special kind of program in which a predicate on the set of states occurs. Programs and formulas are defined mutually recursive and therefore allow arbitrary PDL formulas as test predicates. In order to extend this definition to non-regular PDL, we have to extend the language alphabet with tests $\varphi?$ for any formula $\varphi$. Tests are allowed to occur at arbitrary positions in a word $w \in L(\mathcal{A})$.

**Definition 27 (Non-Regular PDL with Tests)** Let $\mathcal{P}$ be a finite set of propositions, $\Sigma$ be a finite set of actions and $\mathcal{L}$ be a class of formal languages over $\Sigma$. Formulas and programs of PDL[$\mathcal{L}$] are defined mutually recursive as the least sets $\mathsf{Form}$ and $\mathsf{Prog}$ respectively, satisfying the following conditions:

- $\mathcal{P} \subseteq \mathsf{Form}$.
- If $\varphi \in \mathsf{Form}$ then $\neg\varphi \in \mathsf{Form}$.
- If $\varphi, \psi \in \mathsf{Form}$ then $\varphi \vee \psi \in \mathsf{Form}$.
- If $\varphi \in \mathsf{Form}$ and $L \in \mathsf{Prog}$ then $\langle L \rangle \varphi \in \mathsf{Form}$.
- If $L \in \mathcal{L}$ then $L^? \in \mathsf{Prog}$, where $L^? = \{w \in (\Sigma \cup \{\varphi? \mid \varphi \in \mathsf{Form}\})^* \mid w_{|\Sigma} \in L\}$.

In the last clause, $w_{|\Sigma}$ defines an operation on $w$ which deletes all tokens except those occurring in $\Sigma$. Hence the clause indeed defines programs as languages $L \in \mathcal{L}$ in which tests may occur at arbitrary positions.
Note that the alphabet $\Sigma \cup \{\varphi? \mid \varphi \in \mathsf{Form}\}$ for each $L^?$ in every step of the induction is *finite*, because $\mathsf{Form}$ contains only finitely many formulas in each step. This is important regarding finite representations of $L^?$ in e.g. automata, where the set of input symbols consists of exactly this alphabet at a certain finite stage of the induction. It would no

longer be the case if $\mathcal{P}$ was chosen to be infinite, as it is usually assumed in the context of temporal logics. This however is no limitation for the undertaking of model checking: the input formula for a model checking routine is finite and therefore does only contain finitely many different propositions to be considered.

Sometimes we may want to reason about PDL[$\mathcal{L}$] without the test operators and distinguish this fragment by calling it PDL$\not?$[$\mathcal{L}$]. Formulas of PDL$\not?$[$\mathcal{L}$] are obtained from the above definition by omitting the last clause. It is clear that PDL$\not?$[$\mathcal{L}$] is a proper syntactical fragment of PDL[$\mathcal{L}$].

Standard abbreviations $\mathtt{tt}, \mathtt{ff}, \wedge, \rightarrow, \leftrightarrow, [L]$ are defined as for PDL, except of course that $L$ is not necessarily a regular expression but in general a formal language.

For every $\varphi \in$ PDL, we define the set of all its subformulas, $\mathsf{sub}(\varphi)$ inductively as follows:

$$\begin{aligned}
\mathsf{sub}(q) &= \{q\}, \\
\mathsf{sub}(\neg\psi) &= \{\neg\psi\} \cup \mathsf{sub}(\psi), \\
\mathsf{sub}(\psi_1 \vee \psi_2) &= \{\psi_1 \vee \psi_2\} \cup \mathsf{sub}(\psi_1) \cup \mathsf{sub}(\psi_2), \\
\mathsf{sub}(\langle L\rangle\psi) &= \{\langle L\rangle\psi\} \cup \mathsf{sub}(\psi).
\end{aligned}$$

This gives rise to a measure of the size of a formula $\varphi$, defined as $|\varphi| = |\mathsf{sub}(\varphi)|$.

Before we give the semantics of PDL[$\mathcal{L}$] formulas, we need a function which extracts test predicates from formulas.

**Definition 28 (Test Extraction)** Let $\varphi$ be a formula of PDL[$\mathcal{L}$] for some class of formal languages $\mathcal{L}$. The set of tests occurring in $\varphi$ is inductively defined as follows:

$$\begin{aligned}
\mathsf{tests}(q) &= \emptyset, \\
\mathsf{tests}(\neg\varphi) &= \mathsf{tests}(\varphi), \\
\mathsf{tests}(\varphi \vee \psi) &= \mathsf{tests}(\varphi) \cup \mathsf{tests}(\psi), \\
\mathsf{tests}(\langle L\rangle\varphi) &= \{\varphi? \in \Sigma \mid \Sigma \text{ is the least set, s.t. } L \subseteq \Sigma^*\} \cup \mathsf{tests}(\varphi).
\end{aligned}$$

Since we require that the alphabet of a language $L$ used as a program in a formula $\varphi$ is parsable in linear time, this holds for the computation of $\mathsf{tests}(\varphi)$, too.

A formula $\varphi$ of PDL[$\mathcal{L}$] (and PDL$\not?$[$\mathcal{L}$] respectively) is interpreted over an LTS $\mathcal{T} = (\mathcal{S}, \rightarrow, \ell)$ as follows. For every $\psi? \in \mathsf{tests}(\varphi)$, we extend the transition relation $\rightarrow$ by adding $\psi?$-labeled self-loops on any state $s \in \mathcal{S}$ for which $s \models \psi$ holds. Formally, we define

$$\xrightarrow{?} \;:=\; \rightarrow \cup \{(s, \psi?, s) \mid s \models \psi \text{ and } \psi? \in \mathsf{tests}(\varphi)\}$$

and interpret $\varphi$ on the obtained LTS $\mathcal{T}' = (\mathcal{S}, \xrightarrow{?}, \ell)$.

**Definition 29 (Semantics of PDL[$\mathcal{L}$])** Let $\mathcal{T}' = (\mathcal{S}, \xrightarrow{?}, \ell)$ be an LTS as described above and $s \in \mathcal{S}$ be a state. Semantics of a PDL[$\mathcal{L}$] formula is given inductively by

$$
\begin{array}{lll}
s \models q & \text{iff} & s \in \ell(q), \\
s \models \neg\varphi & \text{iff} & s \not\models \varphi, \\
s \models \varphi \vee \psi & \text{iff} & s \models \varphi \text{ or } s \models \psi, \\
s \models \langle L \rangle \varphi & \text{iff} & \text{there are } w \in L \text{ and } t \in \mathcal{S} \text{ s.t.} \\
 & & s \xrightarrow{w} t \text{ and } t \models \varphi.
\end{array}
$$

Note that in definition of the case $s \models \langle L \rangle \varphi$, the transition relation now refers to $\xrightarrow{?}$. It is obvious that for formulas of PDL$\not?$[$\mathcal{L}$], the extended transition relation $\xrightarrow{?}$ is identical to $\rightarrow$ and hence models need not be modified for such formulas.

## 3.2 Examples

**Example 11 (Verification of Programs with Stack Inspection in PDL[IL])** In order to detect access violations in safety critical routines, inspection of the call stack may become necessary, e.g. in case of nested calls, where the initial call came from a method without the required permission. This has been implemented for instance in the runtime access control mechanism of JDK 1.2. In [NST01], such programs are modeled as the set of possible sequences of the call stack w.r.t. the program flow, called traces. The set of possible traces $L_{\mathsf{tr}}$ is an indexed language.
The specification of safe traces in which no access violations occur is given as a regular language $L_{\mathsf{safe}}$ and hence an LTS $\mathcal{T}_{\mathsf{unsafe}}$ resembling the NFA for $\overline{L_{\mathsf{safe}}}$ can be built (see Sec. 3.5.1) which contains the set of unsafe paths. The verification itself can be performed by model checking the formula $\langle L_{\mathsf{tr}} \rangle \mathtt{tt}$ on $\mathcal{T}_{\mathsf{unsafe}}$. If the state $s$ representing the starting configuration of the program satisfies $\langle L_{\mathsf{tr}} \rangle \mathtt{tt}$ this means that there exists an unsafe path which is labeled with a word in $L_{\mathsf{tr}}$ and hence that the program has access violating runs.

**Example 12 (Model Checking PDL[CFL] in Abstract Interpretation)** Consider the system of mutually recursive functions in the left table below, where + denotes nondeterministic choice and ; sequential composition. The function $\mathsf{f}_0$ is the entry point of the system. Supposed we were interested in detecting whether on all possible system executions the call of $\mathsf{f}_3$ is preceded by a successful return of $\mathsf{f}_1$ (security check). Note that the stack behaviour, i.e. the sequences of function calls and returns is non-regular in general (for a

non-fixed number of functions). We state the property we wish to verify as the regular expression $L_{\mathsf{safe}} = \Sigma^* c_1 \Sigma^* r_1 \Sigma^* c_3 \Sigma^*$, where a call of function $\mathsf{f_i}$ is indicated by $c_i$, a return by $r_i$ respectively. It is possible to use abstract interpretation and overapproximate the system of recursive functions into a one-state transition system with looping transitions for all elements in $\Sigma$. In order to restrict this overapproximation to non-spurious runs one can consider the context-free grammar $G$ on the right below which is straight-forwardly derived from the recursive functions. Safety of the system is then established by checking the PDL[CFL] property $\varphi_{\mathsf{safe}} = \neg\langle L(G) \cap \overline{L_{\mathsf{safe}}}\rangle\mathtt{tt}$.

$$
\begin{array}{lcl}
\mathsf{f}_0 & := & \mathsf{f}_2;\mathsf{f}_3 + \mathsf{f}_2;\mathsf{f}_1 \\
\mathsf{f}_1 & := & \mathsf{f}_3;\mathsf{f}_1 + \mathsf{f}_2;\mathsf{f}_3 + \mathsf{f}_1;\mathsf{f}_3 \\
\mathsf{f}_2 & := & \mathsf{f}_1;\mathsf{f}_2 + \mathsf{f}_2;\mathsf{f}_3 + \mathsf{term} \\
\mathsf{f}_3 & := & \mathsf{f}_1;\mathsf{f}_1 + \mathsf{term}
\end{array}
\qquad
\begin{array}{lcl}
F_0 & \to & c_0F_2F_3r_0 \mid c_0F_2F_1r_0, \\
F_1 & \to & c_1F_3F_1r_1 \mid c_1F_2F_3r_1 \mid c_1F_1F_3r_1, \\
F_2 & \to & c_2F_1F_2r_2 \mid c_2F_2F_3r_2 \mid c_2r_2, \\
F_3 & \to & c_3F_1F_1r_3 \mid c_3r_3.
\end{array}
$$

It is easy to see that the only state $s$ does not satisfy $\varphi_{\mathsf{safe}}$: $F_0 \Rightarrow c_0F_2F_1r_0 \Rightarrow^* c_0c_2r_2c_3r_3F_1r_0$. Every derivation continuing from this point will end in a violation of $L_{\mathsf{safe}}$, because every derivation from $F_1$ will be prefixed by $c_1$.

## 3.3 Properties

Unlike for PDL, not every satisfiable PDL[$\mathcal{L}$] formula is satisfied in a finite model if $\mathcal{L}$ contains non-regular languages. This result is proved by exhibiting a PDL[VPL] formula, showing that it is satisfiable and that any model must have infinitely many states.

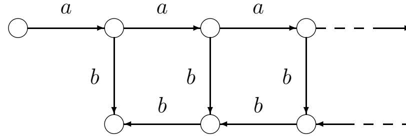
**Theorem 28 (Finite Model Property Absence)** PDL[VPL] does not exhibit the finite model property.

Proof Let $L = \{a^nb^n \mid n \in \mathbb{N}\}$. As shown in Ex. 1, $L$ is a VPL and since by Thm. 4, VPL are closed under negation, so is $\overline{L}$.
Consider the formula

$$\varphi \;:=\; \underbrace{[a^*]\langle a\rangle\mathtt{tt} \;\wedge\; [a^*b^+a]\mathtt{ff}}_{\varphi_1} \;\wedge\; \underbrace{[L][a \cup b]\mathtt{ff}}_{\varphi_2} \;\wedge\; \underbrace{[\overline{L}]\langle b\rangle\mathtt{tt}}_{\varphi_3}$$

where we use regular expressions in the modalities besides languages. Suppose $\mathcal{T}$ is a model of $\varphi$. Because of the first conjunct of $\varphi_1$, it must have an infinite $a$-path, and because of the second conjunct, all other maximal paths must be of the form $a^*b^*$ or $a^*b^\omega$. The latter is

however impossible because of $\varphi_2$ which states that every $a^n b^n$-path is a dead end. Finally, $\varphi_3$ craves the existence of a path with label $a^n b^n$ for any $n \in \mathbb{N}$, because any path in $\overline{L}$ must have a $b$-successor and this holds in particular for every state along the infinite $a$-path. Note that $\varphi$ is satisfiable, for instance by the following infinite model.




It is easy to see that every model of $\varphi$ must be of infinite size. Let $s_0 \xrightarrow{a} s_1 \xrightarrow{a} s_2 \ldots$ be the infinite $a$-path which needs to exist because of $\varphi_1$. Because of $\varphi_2$ and $\varphi_3$, for every $i \in \mathbb{N}$ there must be a path $s_i \xrightarrow{b} t_{i-1} \xrightarrow{b} \ldots \xrightarrow{b} t_0$ having label $b^i$ and ending in a state with no successors. This cannot exist in a finite model of size $n$ for some $n \in \mathbb{N}$ because the $b$-path from $s_n$ would have to contain a loop, but $\mathcal{T}$ cannot contain an infinite $b$-path because of $\varphi_2$. □

**Theorem 29** PDL[$\mathcal{L}$] is bisimulation-invariant and therefore has the tree model property for any $\mathcal{L}$.

Proof For PDL$\not?$[$\mathcal{L}$] this follows from bisimulation-invariance of CTL[$\mathcal{L}$] proved in Thm. 42 and the fact that PDL$\not?$[$\mathcal{L}$] is a sublogic of CTL[$\mathcal{L}$] as proved in Thm. 48. Adding tests poses no difficulties here. □

Given the parametric nature of PDL[$\mathcal{L}$], an immediate question arising regards the correlation between the expressive power of the language class $\mathcal{L}$ and the resulting complexity of program verification.

Early works have only considered decidability of PDL. Fischer and Ladner have shown that PDL is decidable in nondeterministic exponential time and established a deterministic exponential time lower bound [FL79]. The gap was then closed by Pratt who proved decidability in deterministic exponential time [Pra80]. Ladner concluded very early that PDL[CFL] must be undecidable since the validity problem of the formula $\langle L_1 \rangle p \leftrightarrow \langle L_2 \rangle p$ for two context-free languages $L_1, L_2$ amounts to the equivalence problem of CFL which is undecidable [HU79].

A wide study of fragments of PDL[CFL] obtained by restricting the use of context-free languages set off in the 1980ies. Harel et al. refined the previous result by showing that satisfiability of PDL[CFL] is complete for the existential side $\Sigma^1_1$ of the first level

of the analytical hierarchy and established the fact that the borderline to undecidability runs very close to REG: already PDL augmented with the single context-free program $\{a^n ba^n \mid n \in \mathbb{N}\}$ leads to undecidability [HPS83]. Surprisingly, given the similarity of the languages, PDL equipped with the language $\{a^n b^n \mid n \in \mathbb{N}\}$ remains decidable [KP83].
This observation led to the identification of larger fragments of CFL over which PDL is decidable, namely SML, SSML and finally VPL as the most general of them [HR93, HK99, LLS07].
The following table sums up the results and cites the originators.

| | Satisfiability |
|---|---|
| PDL[REG] | EXPTIME-complete [FL79, Pra80] |
| PDL[SML] | 2EXPTIME-complete |
| PDL[SSML] | 2EXPTIME-complete |
| PDL[VPL] | 2EXPTIME-complete [LLS07] |
| PDL[CFL] | undecidable [FL79, HPS83] |

Figure 3.1: Complexity of satisfiability for PDL[$\mathcal{L}$].

Upper bounds for PDL[SML] and PDL[SSML] follow from their inclusion in PDL[VPL] and 2EXPTIME-hardness for decidability of PDL[SML] transfers from [LLS07], where the language used to show the lower bound of PDL[VPL] is actually a SML and hence also a SSML.

## 3.4 Expressivity

Our first and rather obvious observation is that classical PDL indeed coincides with PDL[REG].

**Theorem 30**

$$\text{PDL} = \text{PDL[REG]}.$$

Proof It is well known that regular expressions and NFAs both characterise the class REG and are convertible into each other. Since PDL programs are regular expressions over some $\Sigma$ in which tests may be included, automata over the same alphabet and tests characterising the same languages $L \in \text{REG}$ do exist and vice versa. But then PDL and

PDL[REG] formulas have identical semantics, if the corresponding automata and regular expressions are exchanged. □

Already in [HPS83] the term “non-regular” is applied to the logic PDL[CFL] and it can easily be shown that there are indeed formulas in PDL[$\mathcal{L}$] which are not expressible in $\mathcal{L}_\mu$. However, both logics are in fact incomparable w.r.t. expressivity.

**Lemma 1** Let $L = \{a^n b^n \mid n \in \mathbb{N}\} \in \mathcal{L}$ for some language class $\mathcal{L}$. Then

$$\begin{aligned} \mathrm{PDL}[\mathcal{L}] &\not\leq \mathcal{L}_\mu \text{ and} \\ \mathcal{L}_\mu &\not\leq \mathrm{PDL}\not{?}[\mathcal{L}]. \end{aligned}$$

Proof PDL[$\mathcal{L}$] $\not\leq \mathcal{L}_\mu$: Consider the formula $\varphi = \langle \mathcal{A} \rangle \mathtt{tt}$, where $\mathcal{A}$ is an automaton with $\mathcal{L}(\mathcal{A}) = L$. The formula $\varphi$ is not expressible in $\mathcal{L}_\mu$. This can already be shown for finite word models. A finite word model is an LTS s.t. its states can be arranged to a finite sequence $s_0 \dots s_n$ with exactly one transition $a_{i+1}$ between each pair of adjacent states $s_i$ and $s_{i+1}$ for all $0 \leq i < n$. The concatenation of transition labels forms a finite word $w = a_1 \dots a_n$. Let $W$ be a finite word model of some $w$. Then we have $s_0 \models_W \varphi$ iff $w \in \mathcal{L}(\mathcal{A})$ immediately from the definition.
Hence the set of (all words obtained from) all word models which satisfy $\varphi$ coincides with $L$. It is well-known that $L \notin$ REG. But any formula of $\mathcal{L}_\mu$ translates into a formula of the bisimulation-invariant fragment of MSO and from there into an NFA. Hence there is no formula which is satisfied by the same set of word models.
$\mathcal{L}_\mu \not\leq \mathrm{PDL}\not{?}[\mathcal{L}]$: The proof anticipates the definition of the logic CTL[$\mathcal{L}$] from Chapter 4 and the result that $\mathrm{PDL}\not{?}[\mathcal{L}]$ is equivalent to the CTL[$\mathcal{L}$] fragment EF[$\mathcal{L}$]. In [ALL$^+$b] it is shown that the CTL$^*$ formula $\mathtt{EGF}q$ is not equivalent to any formula in CTL[$\mathcal{L}$] . Since CTL$^* \lneq \mathcal{L}_\mu$ this entails that there is a $\mathcal{L}_\mu$-formula which is not equivalent to any CTL[$\mathcal{L}$] formula and in particular not to any EF[$\mathcal{L}$] formula from which the claim follows. □

We strongly suspect that the result can be extended to $\mathcal{L}_\mu \not\leq$ PDL[$\mathcal{L}$], but have no proof. The above lemma entails that PDL over all language classes in the Chomsky hierarchy which subsume SML are indeed non-regular.
PDL[$\mathcal{L}$] receives its expressive power from the interplay between the intrinsic logical machinery common to all PDL[$\mathcal{L}$] variants and the externally supplied expressive power from the language class parameter $\mathcal{L}$. It is immediately seen that for any of the language classes

REG, SML, SSML, VPL, MVPL, CFL, MSCL, IL, CSL, RE the $\subseteq$-relation is inherited to the logics equipped with the corresponding powers.

**Theorem 31** For all $\mathcal{L}, \mathcal{L}' \in \{$REG, SML, SSML, VPL, MVPL, CFL, MSCL, IL, CSL, RE$\}$, if $\mathcal{L} \subseteq \mathcal{L}'$ then

$$\begin{aligned} \mathrm{PDL}\!\not{?}[\mathcal{L}] &\leq \mathrm{PDL}\!\not{?}[\mathcal{L}']. \\ \mathrm{PDL}\!\not{?}[\mathcal{L}] &\leq \mathrm{PDL}[\mathcal{L}']. \\ \mathrm{PDL}[\mathcal{L}] &\leq \mathrm{PDL}[\mathcal{L}']. \end{aligned}$$

Proof Follows from syntactic inclusion: for any $\varphi \in \mathrm{PDL}\!\not{?}[\mathcal{L}]$, we have $\varphi \in \mathrm{PDL}\!\not{?}[\mathcal{L}']$ and $\varphi \in \mathrm{PDL}[\mathcal{L}']$ and for any $\varphi \in \mathrm{PDL}[\mathcal{L}]$ we have $\varphi \in \mathrm{PDL}[\mathcal{L}']$. □

It is however not obvious at all whether these inclusions are strict or not. Some of the above statements however can be strengthened to strict results.

**Theorem 32**

$$\begin{aligned} \mathrm{PDL}\!\not{?}[\mathrm{REG}] &\lneq \mathrm{PDL}\!\not{?}[\mathrm{SSML}]. \\ \mathrm{PDL}\!\not{?}[\mathrm{VPL}] &\lneq \mathrm{PDL}\!\not{?}[\mathrm{DCFL}]. \end{aligned}$$

Proof The separation of $\mathrm{PDL}\!\not{?}[\mathrm{REG}]$ and $\mathrm{PDL}\!\not{?}[\mathrm{SML}]$ is a consequence of the fact that $\mathrm{PDL}\!\not{?}[\mathrm{REG}]$ is contained in $\mathcal{L}_\mu$ while by Thm. 1 we have that $\mathrm{PDL}\!\not{?}[\mathrm{SML}]$ contains formulas inexpressible in $\mathcal{L}_\mu$.
The second result is obtained by an inspection of a proof in [ALL$^+$b] where CTL[VPL] is separated from CTL[DCFL] (see Sec. 4 for a definition of these logics). The CTL[DCFL] formula shown to be inexpressible in CTL[VPL] is in fact already a formula of the fragment EF[DCFL] and hence by Thm. 47 expressible in $\mathrm{PDL}\!\not{?}[\mathrm{DCFL}]$. Since by the same theorem $\mathrm{PDL}\!\not{?}[\mathrm{VPL}] \equiv \mathrm{EF}[\mathrm{VPL}]$ we have that $\mathrm{PDL}\!\not{?}[\mathrm{VPL}]$ is included in CTL[VPL] and obtain the result. □

The following result states that up to the context-free language classes, parametric PDL without tests is strictly weaker than PDL with tests.

**Theorem 33**

$$\mathrm{PDL}\!\not{?}[\mathrm{CFL}] \lneq \mathrm{PDL}[\mathrm{CFL}].$$

Proof A proof of the version of this theorem for PDL[REG] can be found in [BP81]. The proof idea there is as follows: consider the PDL[REG] formula $\varphi = \langle (p?; a)^*; \neg p?; a; p?\rangle \mathtt{tt}$ for a proposition $p$ and an action $a$. The program can be seen as an encoding of the statement "`until` $\neg p$ `do` $a$" followed by the execution of yet another $a$ and $p?$. Furthermore consider a family of ring-shaped models $\mathcal{T}_m$ connected by unidirectional $a$-transitions, each of length $2m + 1$ for all $m > 0$. Name the states $s_0, \ldots s_{2m}$. In every state the proposition $p$ holds except in $s_0, s_{m-1}, s_{2m-1}$ and $s_{2m}$, where $\neg p$ holds, i.e. if one thinks of the states aligned in a linear sequence then only the first, the one preceding the middle and the last two states do not satisfy $p$. Clearly, $s_0 \models_{\mathcal{T}_m} \varphi$ iff $s_m \not\models_{\mathcal{T}_m} \varphi$ for all $m > 0$.

Now one shows that in test-free PDL there is no formula which can distinguish $s_0$ and $s_m$ on such models under two further conditions: $2m + 1$ is a prime and $m - 1$ is greater than the number of $\langle a\rangle$ occurring in such a formula. Because all regular expressions on one-letter alphabets have a normal form $R \cup R'; (a^n)^*$ for star-free and possibly empty expressions $R, R'$ and $n \geq 1$, there is a corresponding normal form for PDL formulas over one-letter regular programs, s.t. the only subformulas which may occur are of the form $\langle A\rangle\psi$, with $A = a$ or $A = (a^n)^*$.

Clearly, any formula distinguishing $s_0$ and $s_m$ on such models must do so in states in which the propositions differ. That is, in order to claim that there exists a test-free formula $\psi$, s.t. $s_0 \models_{\mathcal{T}_m} \psi$ iff $s_m \not\models_{\mathcal{T}_m} \psi$ holds, $\psi$ must say that only states in which $p$ holds are reachable from $s_0$ and simultaneously that only states in which $\neg p$ holds are reachable from $s_m$ or vice versa.

However, by construction, the number of $\langle a\rangle$ occurrences does not suffice to "reach" a state further away than $m-2$ $a$-transitions. Note that along the way equally for $s_0$ and $s_m$, only $p$ holds. Hence, these formulas do not distinguish $s_0$ and $s_m$.

Regarding subformulas of type $\langle (a^n)^*\rangle$, we have two cases: either $n = 2m+1$ or not. If $n = 2m + 1$, every iteration of $n$ $a$-steps returns at the starting point and hence simultaneously reaches $s_0$ and $s_m$ in which the same proposition holds. If $n \neq 2m + 1$, since $2m + 1$ is prime, both $s_0$ and $s_m$ reach every other state in $\mathcal{T}_m$ and hence not only states with homogenous propositions. As a consequence, test-free PDL cannot distinguish $s_0$ and $s_m$ in $\mathcal{T}_m$ for sufficiently large $m$.

This proof can be extended to PDL[CFL] as follows. Since by Thm. 31, PDL[REG] $\leq$ PDL[CFL] the above mentioned formula $\varphi$ is expressible in PDL[CFL]. Furthermore, it is known that every CFL over one-letter alphabets (denoted by CFL-1) is a regular language [HU79]. Clearly, any formula of PDL$\not?$[CFL] using languages over an $n$-letter alphabet,

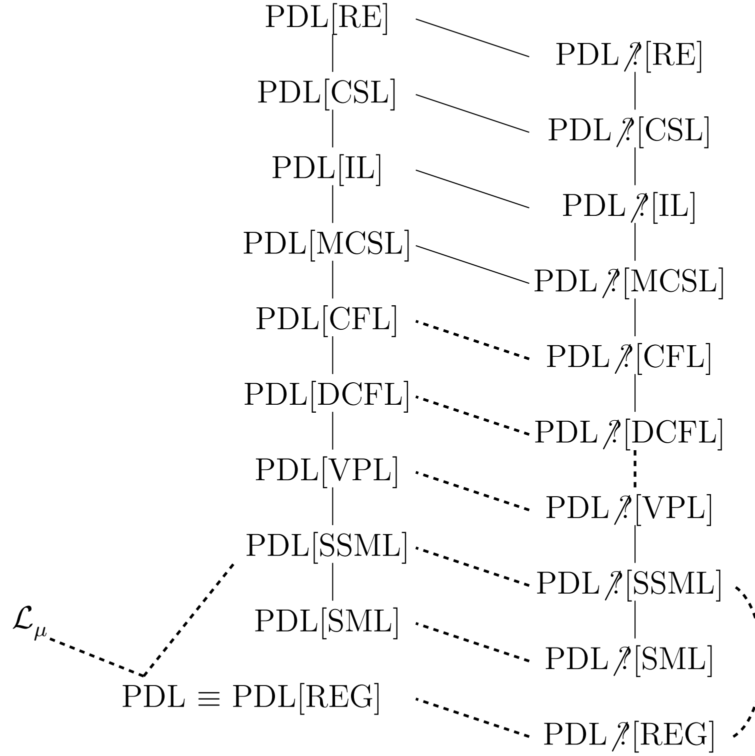



Figure 3.2: Expressive power of PDL[$\mathcal{L}$].

where $n > 1$ cannot do more in terms of expressivity on the type of models described above than a formula using one-letter languages. Thus, all relevant, i.e. PDL$\not?$[CFL − 1] formulas, translate to PDL$\not?$[REG] and hence cannot distinguish $s_0$ and $s_m$ in $\mathcal{T}_m$ either for sufficiently large $m$.

Note that this argument is equally valid for PDL[SSML], PDL[VPL] and PDL[DCFL]. □

**Corollary 2** Let $\mathcal{L} \in \{$SSML, VPL, DCFL$\}$.

$$\text{PDL}\not?[\mathcal{L}] \lneq \text{PDL}[\mathcal{L}].$$

Fig. 3.2 summarises the expressivity results on PDL[$\mathcal{L}$]. A line from a lower positioned item to a higher positioned item denotes inclusion of the former in the latter. If it is dashed this means that the inclusion is strict.

## 3.5 Model Checking

While the complexity and decidability of the satisfiability problem for PDL w.r.t. the class of featured programs is well understood by now, there are still some open questions regarding decidability and complexity of the corresponding model checking problems. The range of language classes that is interesting for the satisfiability problem, namely classes between the regular and the context-free ones, is entirely model checkable in polynomial time [Lan05]. Therefore it is reasonable to extend the scope of considered language classes for the model checking problem beyond the context-free.

The only formula type in which PDL[$\mathcal{L}$] and propositional logic differ is the modal expression scheme $\langle L \rangle \varphi$. Insofar it is the only formula type which poses difficulties for model checking relative to the rather easily solved model checking of propositional logic. There is however an observation which allows to reduce the model checking problem for this formula type to well-studied problems of formal language theory: intuitively, solving the problem $s \models \langle L \rangle \varphi$ amounts to synchronously finding a $w \in L$ and a $w$-labeled path in the model starting in $s$ and ending in a state satisfying $\varphi$. Clearly, it is possible to regard the model as a language consisting of all paths starting in $s$ or – more precisely – the concatenation of their labels. The apparent similarity of an LTS and an NFA suggests that this path language is regular and brings up the conjecture that the synchrony can be captured by intersecting $L$ and the language induced by the LTS. Checking the resulting language for non-emptiness should then solve the model checking problem, since any witness would be a member of $L$ and correspond to an LTS path from $s$, provided that $\varphi$ holds in the target state.

The following section will develop this reduction formally, work out a generic method for model checking PDL[$\mathcal{L}$] and transfer the complexity results accordingly. We thereafter turn our attention to the logics resulting from the largest classes of formal languages for which we have deduced decidability of model checking and develop concrete model checking routines which can be implemented straight-forwardly. We also prove soundness and completeness of these algorithms.

### 3.5.1 A Generic Method

The goal of this section is to carve out the territory of formal language classes $\mathcal{L}$ over which the model checking problem for PDL[$\mathcal{L}$] remains decidable and to show that the method we develop can be used generically to determine its complexity with respect to the language

parameter.

The *non-emptiness problem* for a class $\mathcal{L}$ of formal languages is the following: given a finitely represented $L \in \mathcal{L}$, decide whether or not $L \neq \emptyset$. Furthermore, a class $\mathcal{L}$ is *closed under intersections with regular languages* if for every $L \in \mathcal{L}$ and every regular language $R$ we have $L \cap R \in \mathcal{L}$.

**Definition 30 (REG-Intersection Problem)** The problem of *non-emptiness of intersection with a regular language – REG-intersection problem* for short – for $\mathcal{L}$ is the following: given a finitely represented $L \in \mathcal{L}$ and an NFA $\mathcal{A}$ over a set of terminal symbols $\Sigma$, decide whether or not $L \cap L(\mathcal{A}) \neq \emptyset$.

Clearly, if a class of languages is closed under intersections with regular languages and has a decidable non-emptiness problem, then its REG-intersection problem is decidable, too. Furthermore, if a class of languages is closed under intersections with regular languages but has an undecidable non-emptiness problem then its REG-intersection problem is also undecidable.

We start by showing the close relationship between the REG-intersection problem for $\mathcal{L}$ and the graph-reachability problem for $\mathcal{L}$.

**Definition 31 ($\mathcal{L}$-reachability Problem)** Let $\mathcal{L}$ be a class of languages. The *$\mathcal{L}$-reachability problem* is the following: given an LTS $\mathcal{T} = (\mathcal{S}, \rightarrow, \ell)$, a state $s \in \mathcal{S}$, a set of states $T \subseteq \mathcal{S}$ and a finitely represented $L \in \mathcal{L}$, decide whether or not there is a $w \in L$ and a $t \in T$ s.t. $s \xrightarrow{w} t$.

**Lemma 2** The problem of non-emptiness of intersections with a regular language for $\mathcal{L}$ reduces in linear time to the $\mathcal{L}$-reachability problem.

PROOF Let $L \in \mathcal{L}$ and $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ be an NFA. Take a fixed proposition $f$ and define an LTS $\mathcal{T}_\mathcal{A} := (Q, \rightarrow, \ell)$ with $s \xrightarrow{a} t$ iff $t \in \delta(s, a)$ for any $s, t \in Q$, and $\ell(s) := \{f\}$ if $s \in F$ and $\ell(s) := \emptyset$ otherwise.

Now, $L \cap L(\mathcal{A}) \neq \emptyset$ iff there exists a $w := a_1 a_2 \ldots a_n$ for some $n \in \mathbb{N}$ s.t. $w \in L$ and $w \in L(\mathcal{A})$. The latter is the case iff there are states $q_0, q_1, \ldots q_n$ s.t. $q_{i+1} \in \delta(q_i, a_{i+1})$ for all $i < n$ and $q_n \in F$. This holds by construction of $\mathcal{T}_\mathcal{A}$ iff $q_0 \xrightarrow{w} q_n$ and $q_n \in \ell(f)$. Clearly, $\mathcal{T}_\mathcal{A}$ can be constructed in $\mathcal{O}(|\mathcal{A}|)$. From this follows the claim. $\square$

**Lemma 3** The $\mathcal{L}$-reachability problem reduces in linear time to the problem of non-emptiness of intersections with a regular language for $\mathcal{L}$.

Proof Let $\mathcal{T} = (\mathcal{S}, \rightarrow, \ell)$ be an LTS, $s \in \mathcal{S}$, $T \subseteq \mathcal{S}$, and $L \in \mathcal{L}$. Define an NFA $\mathcal{A}_{\mathcal{T},s,T} := (\mathcal{S}, \Sigma, \delta, s, T)$ s.t. for all $t \in \mathcal{S}$ and all $a \in \Sigma$: $\delta(t,a) := \{u \mid t \xrightarrow{a} u\}$. Note that $\mathcal{A}_{\mathcal{T},s,T}$ can be constructed in $\mathcal{O}(|\mathcal{T}|)$.
Now there is a $w \in L$ and a $t \in T$ with $s \xrightarrow{w} t$ iff there is a path in $\mathcal{T}$ from $s$ to some $t \in T$ s.t. the transition labels along that path form the word $w$. This is the case iff $w \in L(\mathcal{A}_{\mathcal{T},s,T}) \cap L$. Hence, there is such a $w$ iff $L \cap L(\mathcal{A}_{\mathcal{T},s,T}) \neq \emptyset$. □

In order to be able to transfer lower complexity bounds from the REG-intersection problem to the model checking problem for PDL[$\mathcal{L}$], we now show that the $\mathcal{L}$-reachability problem reduces in linear time to model checking PDL[$\mathcal{L}$].

**Lemma 4** Let $\mathcal{L}$ be any class of languages. The $\mathcal{L}$-reachability problem reduces in linear time to the model checking problem for PDL[$\mathcal{L}$].

Proof Let $L \in \mathcal{L}$ be a language over the alphabet $\Sigma$, $\mathcal{T} = (\mathcal{S}, \rightarrow, \ell)$ be an LTS, $s \in \mathcal{S}$ and $T \subseteq \mathcal{S}$. Let $q_T$ be a proposition. Define $\mathcal{T}' = (\mathcal{S}, \rightarrow, \ell')$ s.t. for all $u \in \mathcal{S}$:

$$\ell'(u) \;:=\; \begin{cases} \{q_T\} & \text{, if } u \in T \\ \emptyset & \text{, otherwise.} \end{cases}$$

Now, for any $L \in \mathcal{L}$, there is a $w \in L$ and a $t \in T$ with $s \xrightarrow{w} t$ iff $\mathcal{T}', s \models \langle L \rangle q_T$. Furthermore, both $\mathcal{T}'$ and $\langle L \rangle q_T$ can be constructed in time linear in $\mathcal{T}$ and a representation of $L$. □

It seems however unlikely that also the reverse reduction is possible, because of a lack of direct means to encode the propositional operators of PDL[$\mathcal{L}$] into the reachability or REG-intersection problem.
But having at hand an algorithm solving the $\mathcal{L}$-reachability problem, we can construct a model checker for PDL[$\mathcal{L}$] rather easily. Let $\texttt{reach}(s, L, T)$ be an algorithm which solves the $\mathcal{L}$-reachability problem and takes as arguments a state $s \in \mathcal{S}$, an appropriately represented language $L \in \mathcal{L}$ and a set $T \subseteq \mathcal{S}$ . We assume here that the $\mathcal{L}$-reachability problem is decidable and will later on show for which $\mathcal{L}$ this is the case. Clearly, we can construct a procedure $\texttt{reach}(L, T)$ with $L$ and $T$ as before which returns the set of states $U = \{s \in \mathcal{S} \mid \texttt{reach}(s, L, T) = \mathit{true}\}$ by calling procedure $\texttt{reach}(s, L, T)$ for each $s \in \mathcal{S}$.
Consider now the following algorithm for model checking PDL[$\mathcal{L}$].

```
MC-PDL(𝒯, φ) =
    let (𝒮, →, ℓ) = 𝒯 in
    case φ of
        q       : ℓ(q)
        ¬ψ      : 𝒮 \ MC-PDL(𝒯, ψ)
        ψ1 ∨ ψ2 : MC-PDL(𝒯, ψ1) ∪ MC-PDL(𝒯, ψ2)
        ⟨L⟩ψ    : let {ψ1?, ..., ψn?} = tests(L) in
                  →' := →
                  for i = 1, ..., n do
                      let U = MC-PDL(𝒯,ψi) in
                      for each u ∈ U do
                          →' := →'  ∪  (u, ψi?, u)
                      done
                  done
                  let V = MC-PDL((𝒮, →', ℓ),ψ) in
                  reach(L, V)
```

MC-PDL takes an LTS $\mathcal{T}$ and a formula $\varphi$ and computes the set of states in $\mathcal{T}$ which satisfy $\varphi$. It uses an oracle `reach` which differs depending on the class of languages used in the modal formulas. In case it encounters a modal formula $\langle L\rangle\psi$ it first extracts the tests occurring in the representation of $L$ with the subroutine `tests`, then computes separately for each test $\psi_i$? all states $u$ in which $\psi_i$ holds and finally transforms the transition relation with a $\psi_i$?-self-transition on $u$ accordingly. Finally it computes the set of states in which $\psi$ holds (on the transformed LTS) and uses these states as targets for the $L$-reachability problem in the oracle `reach`.

Soundness and completeness are proved by a straight-forward structural induction on $\varphi$. The only difficulty arises from the fact that the algorithm modifies $\mathcal{T}$ in order to be able to deal with potential tests contained within $\varphi$ in case $\varphi$ is a modal formula. The computation of $\xrightarrow{?}$ from Def. 29 has to be performed w.r.t. each formula or, more precisely, the set of tests occurring in each formula, because there are infinitely many tests in general.

MC-PDL however does this computation on-the-fly and for each modal subformula $\langle L\rangle\psi$ separately. At first, the set of tests is determined in the subroutine `tests`$(L)$ in the corresponding recursion step. After computing for each test $\psi_i$? in `tests`$(L)$ the set of states $U$ in which $\psi_i$ holds (by recursively calling MC-PDL on $\psi_i$), the transition relation is updated with a $\psi_i$?-self-loop for all states in $U$.

Hence, whenever the call of $\texttt{reach}(L, U)$ is reached on any recursion level, it is ensured that for all tests $\psi_i? \in \texttt{tests}(L)$ and for all $u \in \mathcal{S}$ we have $u \models \psi_i$ iff $u \xrightarrow{\psi_i?} u$, of course under the assumption that the MC-PDL computation of $\psi_i$ is sound and complete. This means that just after completion of the double `for`-loop on any recursion level, the current modification of $\rightarrow'$ coincides with $\xrightarrow{?}$ as defined for the current subformula $\langle L\rangle\psi$ and the tests contained within. On the level of the input formula $\varphi$, we therefore have $\rightarrow' = \xrightarrow{?}$ after the double `for`-loop.

**Theorem 34 (Soundness and Completeness)** For all LTS $\mathcal{T} = (\mathcal{S}, \rightarrow, \ell)$, $s \in \mathcal{S}$ and $\varphi \in \text{PDL}[\mathcal{L}]$ we have

$$s \models \varphi \quad \text{iff} \quad s \in \text{ MC-PDL}(\mathcal{T}, \varphi).$$

Proof From the preconsiderations above, it remains to show that the semantics computation on the modified LTS is sound and complete.

**Soundness.** We assume $s \in$ MC-PDL$(\mathcal{T}, \varphi)$ and prove the claim by a structural induction on $\varphi$. Algorithm MC-PDL treats propositional operators as expected and their proof is entirely trivial.

In case $\varphi$ is of the form $\langle L\rangle\psi$, we may assume that $\xrightarrow{?}$ has been computed correctly. The actual model checking of $\langle L\rangle\psi$ is performed via calling the procedure $\texttt{reach}(L, U)$, where $U$ is the set of recursively computed target states in which $\psi$ holds.

By I.H. for any $t \in U$ we have $t \models \psi$. Clearly, a call of $\texttt{reach}(L, U)$ on the modified LTS then returns exactly the set of states $P$ from which there is a path to some state in $U$ labeled with a $w \in L$. But then, if $s \in P$ we have $s \models \varphi$.

**Completeness.** Assume $s \models \varphi$. Again, we show the claim by a structural induction on $\varphi$. Propositional cases are trivial. If $s \models \langle L\rangle\psi$ then there is a $t \in \mathcal{S}$ and a $w \in L$, s.t. $s \xrightarrow{w} t$ and $t \models \psi$. By I.H. we have that $U =$ MC-PDL$(\mathcal{T}, \psi)$ contains $t$. Since the LTS transition relation modification faithfully reflects $\xrightarrow{?}$, procedure $\texttt{reach}(L, U)$ returns a set containing $s$. □

Note that the running time of MC-PDL depends on the running time of $\texttt{tests}(L)$ which in turn depends on the representation of $L$. As argued before, it is easy to construct cases in which the set of tests is hard to detect and may influence the running time significantly. The tests could for instance be encrypted and be hard to decrypt.

Since most of the following results use MC-PDL as a basis for a complexity analysis of model checking PDL$[\mathcal{L}]$, it is essential that the computation of $\texttt{tests}(L)$ for any $L \in \mathcal{L}$ does not affect its asymptotic complexity. We emphasise once more that we make the

implicit assumption of a reasonable representation of $L$, in particular that it is finite and its alphabet is computable in linear time (and therefore also $\texttt{tests}(L)$).

**Lemma 5** The model checking problem for PDL[$\mathcal{L}$] Turing-reduces to the $\mathcal{L}$-reachability problem in time $\mathcal{O}(|\mathcal{T}| \cdot |\varphi|)$.

PROOF It is not hard to see that algorithm MC-PDL can be made to run in time $\mathcal{O}(|\mathcal{T}|\cdot|\varphi|)$ not counting the time complexity of the oracle procedure $\texttt{reach}(L,U)$. Using a dynamic programming approach one can restrict the numbers of recursive calls to one per subformula or test occurring in the input formula. Also, set operations and updates of the labeling function can be made to run in time $\mathcal{O}(|\mathcal{T}|)$. □

The following diagram summarises the conclusions drawn so far:

$$\begin{array}{ccc} \mathcal{L}-\text{reachability} & \overset{\mathcal{O}(|\mathcal{T}||\varphi|)}{\underset{\mathcal{O}(|\mathcal{T}|)}{\Longleftarrow\!\!\longrightarrow}} & \text{model checking PDL}[\mathcal{L}] \\ \mathcal{O}(|\mathcal{T}|)\downarrow\uparrow\mathcal{O}(|\mathcal{A}|) & & \\ \text{REG-intersection for } \mathcal{L} & & \end{array}$$

A single line from $X$ to $Y$ denotes a many-one reduction from $X$ into $Y$ transfering lower bounds along the arrow and upper bounds in the opposite direction. A double line denotes a Turing reduction transferring only an upper bound down the arrow but not a lower bound up the arrow. Taken together, these results allow to transfer lower bounds on the complexity of PDL[$\mathcal{L}$] model checking from either of the other problems.

Concerning the transfer of upper bounds, we have shown that PDL[$\mathcal{L}$] Turing-reduces to $\mathcal{L}$-reachability in quadratic time. Note that the number of $\texttt{reach}(L,U)$ calls of MC-PDL is bounded by the number of $\langle L\rangle$ occurrences in $\varphi$. Remember that every call of $\texttt{reach}(L,U)$ is realised by $|\mathcal{S}|$ calls of $\texttt{reach}(s,L,T)$. Putting this together, we have $\mathcal{O}(|\mathcal{S}| \cdot |\varphi|)$ calls of an oracle $\texttt{reach}(s,L,T)$.

This means that we may transfer upper bounds in terms of complexity classes from either of the problems as long as they are at least PTIME, because at this point the $\mathcal{O}(|\mathcal{S}| \cdot |\varphi|)$ complexity of the reduction gets absorbed by the complexity of the other problems.

**Theorem 35** The model checking problem for PDL[$\mathcal{L}$] is equivalent under polynomial-time Turing reductions to the problem of non-emptiness of intersections with a regular language for $\mathcal{L}$.

Proof Immediately from Lemmas 3–4.

This theorem allows to transfer many known results from the theory of formal languages to the model checking theory of PDL[$\mathcal{L}$]. For example, regular languages are closed under intersections and have a decidable non-emptiness problem. Hence, their problem of non-emptiness of intersections of a regular language is decidable, too. In fact, it is decidable in linear time which then yields polynomial time decidability of the model checking problem for PDL[REG]. It is also known that CFL is closed under intersections with regular languages and has a non-emptiness problem that is decidable in polynomial time. Hence, Thm. 35 reproves that model checking for PDL[CFL] is PTIME-complete.
Regarding language classes $\mathcal{L}$, for which the complexity of model checking PDL[$\mathcal{L}$] is unknown, the following table sums up the results from formal language theory.

| Language class | Closed under intersection with REG | Non-emptiness |
|---|---|---|
| REG | ✓ | $\in$ LINTIME |
| SML | ✓ | $\in$ PTIME |
| SSML | ✓ | $\in$ PTIME |
| VPL | ✓ | PTIME-complete |
| CFL | ✓ | PTIME-complete |
| MCSL | ✓ | PTIME-complete |
| IL | ✓ | EXPTIME-complete |
| ACFL | ✓ | undecidable |
| CSL | ✓ | undecidable |

Figure 3.3: REG-intersection and emptiness for some language classes.

In all of the above classes, the intersection with REG causes at most polynomial blow-up. From Thm. 35 and the above table, the borderline to undecidability of model checking PDL[$\mathcal{L}$] can by now be drawn. The class CSL of context-sensitive languages is closed under intersections with regular languages but its non-emptiness problem is undecidable. Hence, the problem of non-emptiness of intersections with a regular language must be

undecidable, too. The same holds for the class ACFL. The exact correspondence of ACFL and CSL is not known.

**Corollary 3** The model checking problems for PDL[CSL] and PDL[ACFL] are undecidable.

Note that the non-emptiness problem for context-sensitive languages is r.e. because the word problem is decidable. However, since the reduction in Lemma 5 is only a Turing-reduction, recursive enumerability does not extend to the model checking problem.
Accordingly, the border to undecidability runs somewhere between the context-free and the context-sensitive languages. The largest language class in this area which fulfills the required conditions is IL: it is closed under intersections with regular languages (with polynomial blow-ups only) and its non-emptiness problem is EXPTIME-complete [Aho68, TK07]. From this follows that its REG-intersection problem also is.

**Corollary 4** The model checking problem for PDL[IL] is EXPTIME-complete.

Other classes which contain CFL, have decidable non-emptiness problems and are closed under intersections with regular languages are the MCSL. Again, they are closed under intersections with regular languages and their non-emptiness problem is decidable – even in polynomial time. Since the blow-up in the construction of the intersection of a linear-indexed grammar with a regular language is polynomial, their REG-intersection problem is in PTIME as well. Thm. 35 then transfers the upper bound to the corresponding model checking. A matching lower bound follows trivially from the PTIME-hardness of the model checking problem for PDL[CFL].

**Corollary 5** The model checking problems for PDL[LIL], PDL[HL], PDL[CCL], and PDL[TAL] are PTIME-complete.

Since we are not aware of any hardness results for the emptiness problem of SML and SSML, we may only transfer upper bounds from the REG-intersection problem.

**Corollary 6** The model checking problems for PDL[SML] and PDL[SSML] are in PTIME.

For a comparison of the complexities of satisfiability and model checking parametric PDL, see Fig. 3.5.1. Note that some of the lower and upper bound results follow from the expressivity results of the logics as stated in Thm. 31. Hence, for any two logics PDL[$\mathcal{L}$] and PDL[$\mathcal{L}'$], where $\mathcal{L} \leq \mathcal{L}'$ and a complexity class $\mathcal{C}$, if either problem is $\mathcal{C}$-hard in PDL[$\mathcal{L}$] then it is also $\mathcal{C}$-hard in PDL[$\mathcal{L}'$] and vice versa for upper bounds.

| | | satisfiability | model checking |
|---|---|---|---|
| PDL[REG] | ∈ / hard | EXPTIME [FL79, Pra80] | LINTIME[CS92] |
| PDL[SML] | ∈ / hard | 2EXPTIME | PTIME |
| PDL[SSML] | ∈ / hard | 2EXPTIME | PTIME |
| PDL[VPL] | ∈ / hard | 2EXPTIME [LLS07] | PTIME |
| PDL[CFL] | ∈ / hard | undec. [FL79, HPS83] | PTIME [Lan05] |
| PDL[MCSL] | ∈ / hard | undec. | PTIME |
| PDL[IL] | ∈ / hard | undec. | EXPTIME |
| PDL[CSL] | ∈ / hard | undec. | undec. |

Figure 3.4: Complexity of SAT vs. model checking PDL[$\mathcal{L}$].

### 3.5.2 A Model Checking Algorithm for PDL over IL

In this section we present an explicit model checking procedure for PDL[IL] that runs in deterministic exponential time and can be implemented straight-forwardly. We focus on the difficulties imposed by the language part. A model checker is then easily obtained by using the procedure sketched in the proof of Lemma 5.
Later, in the soundness proofs, we will need the following important properties of derivations in indexed grammars.

**Lemma 6 (Stack Distribution Property)** For all $A, B_1, \ldots, B_k \in N$ and all $\delta \in I^*$:
**a)** If $A \Rightarrow^* B_1 \ldots B_k$ and no terminal productions are being used in this derivation then $A[\delta] \Rightarrow^* B_1[\delta] \ldots B_k[\delta]$.
**b)** If $A[\delta] \Rightarrow^* B_1[\delta] \ldots B_k[\delta]$ and no terminal productions are being used and for all indexed nonterminals $X[\delta']$ occurring during the derivation, $\delta' = \gamma\delta$ for some $\gamma \in I^*$, then $A \Rightarrow^* B_1 \ldots B_k$.

Proof Both parts follow easily from the following three observations. Let $A, B, C \in N$, $\delta \in I^*$, $f \in I$:

- $A \Rightarrow BC$ iff $A[\delta] \Rightarrow B[\delta]C[\delta]$,
- $A \Rightarrow B[f]$ iff $A[\delta] \Rightarrow B[f\delta]$,
- $A[f] \Rightarrow B$ iff $A[f\delta] \Rightarrow B[\delta]$.

We exemplarily show the first of these equivalences. The two others are analogous. Suppose $A \Rightarrow BC$. According to the definition of $\Rightarrow$, we must have $A \rightarrow BC$ and, hence, $A[\delta] \Rightarrow B[\delta]C[\delta]$ according to the definition of $\Rightarrow$ again. The converse direction is proved in the same way.
For part (a) suppose $A \Rightarrow^* B_1 \ldots B_k$. By successively applying the "if" parts of the three observations above it is easy to construct a derivation which shows $A[\delta] \Rightarrow^* B_1[\delta] \ldots B_k[\delta]$.
For part (b) suppose $A[\delta] \Rightarrow^* B_1[\delta] \ldots B_k[\delta]$ s.t. no terminal productions occur during the derivation and every nonterminal in every intermediate sentential form has an index $\delta' = \gamma\delta$ for some $\gamma \in I^*$. Then one can successively apply the "only if" parts of the three observations above in order to construct a derivation which shows $A \Rightarrow^* B_1 \ldots B_k$. Note that this would not necessarily be possible if some occurring nonterminal had an index which is not of the required form: the proof relies on a simulation of the derivation steps of $A[\delta] \Rightarrow^* B_1[\delta] \ldots B_k[\delta]$ on $A$ with an empty stack. This is possible, as long as $\delta$ is

left untouched at the bottom of all indexed nonterminals in intermediate sentential forms. Performing a pop-production on some intermittent indexed nonterminal $X[\delta]$ cannot be simulated on $X$ with empty stack because the operation is not defined. □

**Lemma 7 (Commutation Lemma)** For all sentential forms $\alpha, \beta, \gamma_1, \gamma_2, \gamma_3$, all $A, B \in N$ and all $\delta, \delta' \in I^*$ the following holds:

$$\begin{array}{rl} & \gamma_1 A[\delta]\gamma_2 B[\delta']\gamma_3 \;\Rightarrow\; \gamma_1\alpha\gamma_2 B[\delta']\gamma_3 \;\Rightarrow\; \gamma_1\alpha\gamma_2\beta\gamma_3 \\ \text{iff} & \gamma_1 A[\delta]\gamma_2 B[\delta']\gamma_3 \;\Rightarrow\; \gamma_1 A[\delta]\gamma_2\beta\gamma_3 \;\Rightarrow\; \gamma_1\alpha\gamma_2\beta\gamma_3. \end{array}$$

Proof This follows immediately from the definition of $\Rightarrow$. □

**Corollary 7** Let $A \in N$ and $w \in \Sigma^*$ s.t. $A \Rightarrow^+ w$. Then there are sentential forms $\alpha_0, \ldots, \alpha_n$ for some $n \in \mathbb{N}$ all of which do not contain terminal symbols, s.t. $\alpha_0 = A$, $\alpha_{i-1} \Rightarrow \alpha_i$ for all $i = 1, \ldots, n$, and $\alpha_n \Rightarrow^m w$ where $m$ is the number of indexed nonterminals in $\alpha_n$.

Proof Suppose $A \Rightarrow \beta_1 \Rightarrow \ldots \Rightarrow \beta_m = w$ for some $\beta_i$. Consider the least $i$ s.t. $\beta_i$ contains a terminal symbol. If every production rule applied to the right of $\beta_i$ is a terminal production then the claim holds. Assume this is not the case. Lemma 7 allows to hold back the applied terminal production rule and instead to first apply the production rule for $\beta_{i+1}$. Repetitive application of this procedure allows to postpone all applications of terminal production rules to the very last. Now note that it takes $m$ steps to replace $m$ indexed nonterminals by $\epsilon$ or a terminal symbol each. □

For the remainder of this section fix an indexed grammar $G = (N, \Sigma, P, I, S)$ and a Kripke structure $\mathcal{T} = (\mathcal{S}, \rightarrow, \ell)$.

**Definition 32 (Annotated Nonterminal)** An *annotated nonterminal* is a triple $(s, A, t)$, where $s, t \in \mathcal{S}$ and $A \in N$. Let $\mathcal{N}$ denote the set of all annotated nonterminals (over $G$ and $\mathcal{T}$), i.e. $\mathcal{N} := \mathcal{S} \times N \times \mathcal{S}$. We say that an annotated nonterminal $(s, A, t)$ *left-matches* another $(u, B, v)$, if $t = u$.

We define a new relation between two states $s, t$, a sentential form $E_1 \ldots E_k$ consisting of unindexed nonterminals only, and a set $\mathcal{B}$ of annotated nonterminals. Intuitively, $s \xrightarrow[\mathcal{B}]{E_1 \ldots E_k} t$ holds iff $\mathcal{B}$ can be rearranged to a sequence of annotated nonterminals in which each left-matches its right neighbour s.t. that sequence starts with $s$, ends in $t$ and the

projection onto its nonterminal symbols yields the sequence $E_1 \dots E_k$. Annotated nonterminals in $\mathcal{B}$ can be used more than once in this sequence, but each of them has to be used at least once. We also call such a sequence on *open path* from $s$ to $t$ because it represents a path from $s$ to $t$ via intermediate states $s_0, \dots, s_k$ s.t. $s_0 = s$, $s_k = t$ and between each $s_{i-1}$ and $s_i$ there is a *hole* which, intuitively, should be closed by a proper path from $s_{i-1}$ to $s_i$ whose label is derivable from $E_i$.

**Definition 33 (Open Path)** Let $k \in \mathbb{N}$, $s, t \in \mathcal{S}$, $D_1, \dots, D_k \in N$, and $\mathcal{B} \subseteq \mathcal{N}$.

$$s \xrightarrow[\mathcal{B}]{D_1 \dots D_k} t \quad \text{iff} \quad \text{there are } s_0, \dots, s_k \in \mathcal{S} \text{ s.t. } s_0 = s, s_k = t$$
$$\text{and } \mathcal{B} = \{(s_{i-1}, D_i, s_i) \mid i = 1, \dots, k\}.$$

Note that the set equality in this definition does not only constrain the available nonterminals which can be used in order to construct an open path from $s$ to $t$. It particularly also demands that every annotated nonterminal in this set is being used in the construction. The left-matching property is hidden in the second conjunct.

**Example 13** Let $\mathcal{S} := \{s, t\}$ and $\mathcal{B} := \{(s, A, t), (t, B, s), (t, C, s)\}$. Then for instance $s \xrightarrow[\mathcal{B}]{ABAC} u$ holds because there is a sequence of left-matching annotated nonterminals corresponding to $ABAC$ which (as a set) forms $\mathcal{B}$, here namely $(s, A, t), (t, B, s), (s, A, t), (t, C, s)$. On the other hand, $s \xrightarrow[\mathcal{B}]{AB} t$ does not hold since the annotated nonterminal $(t, C, s)$ is not being used in this open path. Furthermore, $s \xrightarrow[\mathcal{B}]{ABC} t$ also does not hold, because $(t, B, s)$ does not left-match $(t, C, s)$.

**Definition 34** Let $\mathcal{C}, \mathcal{D}$ be sets of annotated nonterminals and $f \in I$. Define $\mathcal{D}[f] \rightsquigarrow \mathcal{C}$ iff

- for all $(u, C, v) \in \mathcal{C}$ exists $(u, D, v) \in \mathcal{D}$, s.t. $D[f] \to C$ and
- for all $(u, D, v) \in \mathcal{D}$ exists $(u, C, v) \in \mathcal{C}$, s.t. $D[f] \to C$.

The next lemma states some properties of the open path relation. We omit the proof since all parts follow easily from Def. 33.

**Lemma 8** For all $s, t \in \mathit{States}$, all $\mathcal{B}, \mathcal{C}, \mathcal{D}, \mathcal{B}_1, \dots, \mathcal{B}_k \subseteq \mathcal{N}$, all $C_1, \dots, C_k$, $D_1, \dots, D_k \in N$, all $f \in I$, and all $\beta, \gamma, \alpha_1, \alpha_2, \dots \in N^+$ we have the following.

**a)** If $s \xrightarrow[\mathcal{B}]{\beta} u$ and $u \xrightarrow[\mathcal{C}]{\gamma} t$ then $s \xrightarrow[\mathcal{B} \cup \mathcal{C}]{\beta\gamma} t$.

**b)** If there exists $f \in I$, s.t. $\mathcal{D}[f] \rightsquigarrow \mathcal{C}$ and $s \xrightarrow[\mathcal{D}]{D_1 \dots D_k} t$ then $s \xrightarrow[\mathcal{C}]{C_1 \dots C_k} t$.

**c)** If $\mathcal{B} = \{(u_1, C_1, v_1), \dots, (u_k, C_k, v_k)\}$ and $\{i_1, \dots, i_m\} = \{1, \dots, k\}$ for some $m \in \mathbb{N}$ and $s \xrightarrow[\mathcal{B}]{C_{i_1} \dots C_{i_m}} t$ and $u_i \xrightarrow[\mathcal{B}_i]{\alpha_i} v_i$ for all $i = 1, \dots, k$ then $s \xrightarrow[\mathcal{B}_1 \cup \dots \cup \mathcal{B}_k]{\alpha_{i_1} \dots \alpha_{i_m}} t$.

**d)** If $s \xrightarrow[\mathcal{B}]{\beta\gamma} t$ then there are $u \in \mathcal{S}$ and $\mathcal{B}_1, \mathcal{B}_2 \subseteq \mathcal{N}$ s.t. $\mathcal{B} = \mathcal{B}_1 \cup \mathcal{B}_2$ and $s \xrightarrow[\mathcal{B}_1]{\beta} u$ and $u \xrightarrow[\mathcal{B}_2]{\gamma} t$.

**e)** If $s \xrightarrow[\mathcal{B}]{\alpha} t$ and for every $(u, D, v) \in \mathcal{B}$ it holds that $u = v$ and $D \to \epsilon$, or $u \xrightarrow{a} v$ and $D \to a$ for some $a \in \Sigma$, then there is a $w \in \Sigma^*$ s.t. $\alpha \Rightarrow^+ w$ and $s \xrightarrow{w} t$.

### Approximating IL-reachability

In order to solve the IL-reachability problem we are interested in tuples of states $s, t$ and nonterminals $A$ s.t. there is a path from $s$ to $t$ whose label (of terminals) is derivable from $A$. In order to compute these tuples for every nonterminal $A$ we need to consider sets of open paths first. These will be represented by a triple $\langle s, \mathcal{B}, t \rangle \in \mathcal{S} \times 2^{\mathcal{N}} \times \mathcal{S}$, intuitively describing that there is an open path from $s$ to $t$ which uses all elements in $\mathcal{B}$. We use $\langle\rangle$-brackets to distinguish such triples from annotated nonterminals.

**Definition 35** For each $A \in N$, define:

$$\overrightarrow{A} \ := \ \{ \ \langle s, \mathcal{B}, t \rangle \mid \text{there is } \alpha \in N^+ \text{ with } A \Rightarrow^* \alpha \text{ and } s \xrightarrow[\mathcal{B}]{\alpha} t \}.$$

Next we describe a method for computing $\overrightarrow{A}$. We simultaneously define, for any $A \in N$, a sequence $\overrightarrow{A}^0 \subseteq \overrightarrow{A}^1 \subseteq \overrightarrow{A}^2 \subseteq \dots$ that approximates $\overrightarrow{A}$ from below. We will show that $\bigcup_{j \in \mathbb{N}} \overrightarrow{A}^j = \overrightarrow{A}$. Since each of them is a subset of a finite set, it is clear that the chain has to have a maximal element.

We start by defining the initial sets $\overrightarrow{A}^0$ for an $A \in N$:

$$\overrightarrow{A}^0 \ := \ \{ \ \langle s, \{(s, A, t)\}, t \rangle \mid s, t \in \mathcal{S} \}.$$

Intuitively, it is always possible to find a path from any state $s$ to any state $t$ that is labeled with something derivable from $A$ if one is allowed to leave a hole between $s$ and $t$ that should be closed by anything derivable from $A$. Note that $A \Rightarrow^* A$.

Now let $j > 0$. Define $\overrightarrow{A}^j$ as the union of four sets.

$$\overrightarrow{A}^j \ := \ \overrightarrow{A}^{j-1} \ \cup \ \overrightarrow{A}^{j,\text{conc}} \ \cup \ \overrightarrow{A}^{j,\text{push}} \ \cup \ \overrightarrow{A}^{j,\text{ins}}.$$

Hence, anything at level $j-1$ is preserved into level $j$. Open paths at level $j$ can be constructed by concatenating two open paths at level $j-1$. The label of the resulting path is of course only derivable if this is matched by a composition rule in the indexed grammar $G$. Note that the holes in the resulting open path are the union of the holes in both parts.

$$\begin{aligned}\overrightarrow{A}^{j,\text{conc}} \;:=\; \{\, & \langle s, \mathcal{B} \cup \mathcal{C}, t\rangle \mid \text{there are } B, C \in N \text{ and } u \in \mathcal{S} \text{ with } A \to BC \text{ and} \\ & \langle s, \mathcal{B}, u\rangle \in \overrightarrow{B}^{j-1} \text{ and } \langle u, \mathcal{C}, t\rangle \in \overrightarrow{C}^{j-1} \,\}.\end{aligned}$$

Another way of obtaining an open path from $s$ to $t$ derivable from some nonterminal $A$ is to start the derivation with a push production. This has to be matched in the end by corresponding pop productions since we are interested in open paths whose labels are unindexed nonterminal symbols.

$$\begin{aligned}\overrightarrow{A}^{j,\text{push}} \;:=\; \{\, & \langle s, \mathcal{C}, t\rangle \mid \text{ there are } B \in N, f \in I, \mathcal{D} \subseteq \mathcal{N} \text{ s.t. } A \to B[f] \text{ and} \\ & \langle s, \mathcal{D}, t\rangle \in \overrightarrow{B}^{j-1} \text{ and if } \mathcal{D} = \{(u_1, D_1, v_1), \ldots, (u_k, D_k, v_k)\} \\ & \text{then } \mathcal{C} = \{(u_1, C_1, v_1), \ldots, (u_k, C_k, v_k)\} \text{ s.t. } D_i[f] \to C_i \\ & \text{for all } i = 1, \ldots, k \,\}.\end{aligned}$$

Finally, an open path on level $j$ with a derivation of a sentential form from some nonterminal $A$ can be obtained by inserting a derivation into the context of another derivation. For technical reasons, namely to ensure completeness, we require that all parts of the open path are being replaced simultaneously.

$$\begin{aligned}\overrightarrow{A}^{j,\text{ins}} \;:=\; \{\, & \langle s, \mathcal{B}_1 \cup \ldots \cup \mathcal{B}_k, t\rangle \mid \text{there is } \mathcal{C} = \{(u_1, C_1, v_1), \ldots, (u_k, C_k, v_k)\} \\ & \text{s.t. } \langle s, \mathcal{C}, t\rangle \in \overrightarrow{A}^{j-1} \text{ and for all } i = 1, \ldots, k : \\ & \langle u_i, \mathcal{B}_i, v_i\rangle \in \overrightarrow{C_i}^{j-1} \,\}.\end{aligned}$$

We proceed by showing that the sequence $\overrightarrow{A}^0, \overrightarrow{A}^1, \ldots$ correctly approximates $\overrightarrow{A}$.

**Lemma 9 (Soundness)** For all $A \in N$ and all $j \in \mathbb{N}$ we have $\overrightarrow{A}^j \subseteq \overrightarrow{A}$.

Proof We prove this simultaneously for all $A \in N$ by induction on $j$. The base case of $j = 0$ is rather simple. Remember that $\overrightarrow{A}^0$ only consists of elements of the form $\langle s, \{(s, A, t)\}, t\rangle$. Now, clearly $s \xrightarrow[\{(s,A,t)\}]{A} t$ and $A \Rightarrow^* A$. Thus, we have $\langle s, \{(s, A, t)\}, t\rangle \in \overrightarrow{A}$.

Now let $j > 0$. Note that $\overrightarrow{A}^j$ is the union of four sets. For each of these we will show that they are contained in $\overrightarrow{A}$.

Case (i), $\overrightarrow{A}^{j-1} \subseteq \overrightarrow{A}$. This is trivially true by hypothesis.

Case (ii), $\overrightarrow{A}^{j,\text{conc}} \subseteq \overrightarrow{A}$. Suppose $\langle s, \mathcal{B} \cup \mathcal{C}, t\rangle \in \overrightarrow{A}^{j,\text{comp}}$. Then $A \to BC$ and there are $\langle s, \mathcal{B}, u\rangle \in \overrightarrow{B}^{j-1}$ and $\langle u, \mathcal{C}, t\rangle \in \overrightarrow{C}^{j-1}$. By hypothesis we have $\langle s, \mathcal{B}, u\rangle \in \overrightarrow{B}$ and $\langle u, \mathcal{C}, t\rangle \in \overrightarrow{C}$, i.e. there are $\beta, \gamma \in N^+$ s.t. $B \Rightarrow^* \beta$, $C \Rightarrow^* \gamma$ and $s \xrightarrow[\mathcal{B}]{\beta} u$ as well as $u \xrightarrow[\mathcal{C}]{\gamma} t$. Then $A \Rightarrow BC \Rightarrow^* \beta\gamma$ and according to Lemma 8 (a) we also have $s \xrightarrow[\mathcal{B}\cup\mathcal{C}]{\beta\gamma} t$. Hence, $\langle s, \mathcal{B} \cup \mathcal{C}, t\rangle \in \overrightarrow{A}$.

Case (iii), $\overrightarrow{A}^{j,\text{push}} \subseteq \overrightarrow{A}$. Suppose $\langle s, \mathcal{C}, t\rangle \in \overrightarrow{A}^{j,\text{push}}$. Then $A \to B[f]$ for some $B \in N$ and $f \in I$, and there is a $\langle s, \mathcal{D}, t\rangle \in \overrightarrow{B}^{j-1}$ s.t. $\mathcal{D} = \{(u_1, D_1, v_i), \ldots, (u_k, D_k, v_k)\}$ and productions $D_i[f] \to C_i$ for $i = 1, \ldots, k$ s.t. $\mathcal{C} = \{(u_1, C_1, v_i), \ldots, (u_k, C_k, v_k)\}$. By hypothesis, $\langle s, \mathcal{D}, t\rangle \in \overrightarrow{B}$, i.e. there is an $\alpha \in N^+$ s.t. $B \Rightarrow^* \alpha$ and $s \xrightarrow[\mathcal{D}]{\alpha} t$. Let $\alpha = D_1 \ldots D_k$. Now we apply part (a) of Lemma 6 and obtain $B[f] \Rightarrow^* D_1[f] \ldots D_k[f]$. Extending this derivation with the rule $A \to B[f]$ at the top and the rules $D_i[f] \to C$ at the bottom yields $A \Rightarrow^* C_1 \ldots C_k$. According to Lemma 8 (b) we have $s \xrightarrow[\mathcal{C}]{C_1 \ldots C_k} t$. But then $\langle s, \mathcal{C}, t\rangle \in \overrightarrow{A}$ which was to be proved.

Case (iv), $\overrightarrow{A}^{j,\text{ins}} \subseteq \overrightarrow{A}$. Suppose $\langle s, \mathcal{B}, t\rangle \in \overrightarrow{A}^{j,\text{ins}}$ s.t. $\mathcal{B}$ is suitable decomposed into $\mathcal{B} = \mathcal{B}_1 \cup \ldots \cup \mathcal{B}_k$. Then there is a $\langle s, \mathcal{C}, t\rangle \in \overrightarrow{A}^{j-1}$ s.t. $\mathcal{C} = \{(u_1, C_1, v_1), \ldots, (u_k, C_k, v_k)\}$ and for all $i = 1, \ldots, k$ we have $\langle u_i, \mathcal{B}_i, v_i\rangle \in \overrightarrow{C_i}^{j-1}$. By hypothesis, $\langle s, \mathcal{C}, t\rangle \in \overrightarrow{A}$, i.e. there is an $\alpha \in N^+$ s.t. $s \xrightarrow[\mathcal{C}]{\alpha} t$, in particular $A \Rightarrow^* \alpha$. Let $\alpha = C_{i_1} \ldots C_{i_m}$ for some $m \in \mathbb{N}$, $i_1, \ldots, i_m \in \{1, \ldots, k\}$. The hypothesis also yields, for every $i = 1, \ldots, k$, that $\langle u_i, \mathcal{B}_i, v_i\rangle \in \overrightarrow{C_i}$, i.e. there are $\alpha_i \in N^+$ s.t. $C_i \Rightarrow^* \alpha_i$ and $u_i \xrightarrow[\mathcal{B}_i]{\alpha_i} v_i$. Hence, $A \Rightarrow^* \alpha_{i_1} \ldots \alpha_{i_m}$, and Lemma 8 (c) yields $s \xrightarrow[\mathcal{B}]{\alpha_{i_1} \ldots \alpha_{i_m}} t$ which shows that $\langle s, \mathcal{B}, t\rangle \in \overrightarrow{A}$. □

Remember that we want to use the sequence $\overrightarrow{A}^0, \overrightarrow{A}^1, \ldots$ in order to compute $\overrightarrow{A}$ for some $A$. The above shows that the sequence approximates it from below. We need to prove completeness, i.e. the fact that the sequence eventually captures $\overrightarrow{A}$. For this, we need directedness of the family of sets $\overrightarrow{A}^j$ which is an immediate consequence of the following lemma.

**Lemma 10 (Monotonicity)** For all $A \in N$ and all $j, j' \in \mathbb{N}$ we have: $j \leq j'$ implies $\overrightarrow{A}^j \subseteq \overrightarrow{A}^{j'}$.

Proof Trivial. □

Now we prove that eventually all open paths for all nonterminals are indeed collected by the approximation.

**Lemma 11 (Completeness)** For all $A \in N$ exists $j \in \mathbb{N}$ s.t. $\vec{A} \subseteq \vec{A}^j$.

Proof Again, we prove this simultaneously for all $A \in N$. First note that $\vec{A}$ is finite, because so are $\mathcal{S}$ and $N$. Hence, using Lemma 10 it suffices to show that for every $\langle s, \mathcal{B}, t\rangle \in \vec{A}$ there is a $j \in \mathbb{N}$ with $\langle s, \mathcal{B}, t\rangle \in \vec{A}^j$. So take some $\langle s, \mathcal{B}, t\rangle \in \vec{A}$. Hence, there is an $\alpha \in N^*$ s.t. $s \xrightarrow[\mathcal{B}]{\alpha} t$ and $A \Rightarrow^* \alpha$. Thus, there is an $n \in \mathbb{N}$ with $A \Rightarrow^n \alpha$. We show the claim by induction on $n$.

First assume $n = 0$. If $A \Rightarrow^0 \alpha$ then $\mathcal{B} = \{(s, A, t)\}$ because $\alpha = A$ and remember that in $s \xrightarrow[\mathcal{B}]{\alpha} t$ all elements of $\mathcal{B}$ are required to contribute to the construction of the open path. But then $\langle s, \mathcal{B}, t\rangle \in \vec{A}^0$.

Now let $n > 0$, i.e. $A \Rightarrow \beta \Rightarrow^{k-1} \alpha$ for some sentential form $\beta$. We need to make a case distinction according to the rule that is applied in the derivation of $\beta$ from $A$. Note that it cannot be a pop production because the index of $A$ is empty. It also cannot be a terminal production because $\alpha \in N^+$. Hence, it can only be a composite production $A \to BC$ (with $\beta = BC$) or a push production $A \to B[f]$ (with $\beta = B[f]$). Note furthermore, that – for the same reason – terminal productions cannot occur anywhere in this derivation.

Case (i), $A \to BC$. By absence of terminal productions we must have $|\alpha| \geq 2$. Hence, there are $\beta, \gamma \in N^+$ s.t. $\alpha = \beta\gamma$ and $B \Rightarrow^{n_1} \beta$ and $C \Rightarrow^{n_2} \gamma$ with $n_1 + n_2 \leq n - 1$. Furthermore, by assumption we have $s \xrightarrow[\mathcal{B}]{\beta\gamma} t$. Lemma 8 (d) yields a $u \in \mathcal{S}$ and a decomposition $\mathcal{B} = \mathcal{B}_1 \cup \mathcal{B}_2$ s.t. $s \xrightarrow[\mathcal{B}_1]{\beta} u$ and $u \xrightarrow[\mathcal{B}_2]{\gamma} t$. Since $n_1 < n$, the hypothesis yields a $j_1$ s.t. $\langle s, \mathcal{B}_1, u\rangle \in \vec{B}^{j_1}$. Equally, since $n_2 < n$ we also have $\langle u, \mathcal{B}_2, t\rangle \in \vec{C}^{j_2}$ for some $j_2$. Let $j = \max\{j_1, j_2\}$. By Lemma 10 we have $\langle s, \mathcal{B}_1, u\rangle \in \vec{B}^j$ and $\langle u, \mathcal{B}_2, t\rangle \in \vec{C}^j$. By construction we then have $\langle s, \mathcal{B}, t\rangle \in \vec{A}^{j+1}$ which was to be shown.

Case (ii), $A \to B[f]$. Let $\alpha = E_1 \dots E_m$. Furthermore, in the derivation $A \Rightarrow^* \alpha$, every $E_i$ must be derived from a nonterminal $C$ s.t. $C$ itself stems from an application of a rule $D[f] \to C$ s.t. the index symbol $f$ is inherited from $B[f]$ at the beginning of the derivation. In other words, for every $E_i$ we consider the first moment that the thread in the derivation from $B[f]$ to $\alpha$ loses the bottom index symbol $f$. We can group $\alpha$ according to that. Two adjacent symbols in $\alpha$ belong to the same group iff they are derived from the same symbol $C$ which in turn is derived from an application of a rule $D[f] \to C$ s.t. all ancestors of $D[f]$ up to $B[f]$ at the top of the derivation have the symbol $f$ at the bottom of their stack. This means we have

$$\alpha \;=\; E_{1,1} \dots E_{1,m_1} E_{2,1} \dots E_{2,m_2} \dots E_{k,1} \dots E_{k,m_k}$$

for some $m_1, \ldots, m_k$ with $m_1 + \ldots + m_k = m$.
Since each group $E_{i,1} \ldots E_{i,m_i}$ in $\alpha$ stems from a $C$ as said above there are nonterminals $C_1, \ldots, C_k, D_1, \ldots, D_k$ and $n', n_1, \ldots, n_k \in \mathbb{N}$ with $n' + (n_1 + 1) + \ldots + (n_k + 1) \leq n - 1$ s.t.

$$D_i[f] \Rightarrow C_i \Rightarrow^{n_i} E_{i,1} \ldots E_{i,m_i}$$

for every $i = 1, \ldots, k$ and

$$B[f] \Rightarrow^{n'} D_1[f] \ldots D_k[f]$$

s.t. in every intermediate sentential form, every nonterminal has the symbol $f$ at the bottom of their index. According to Lemma 6 (b) we also have $B \Rightarrow^{n'} D_1 \ldots D_k$.
Remember that $s \xrightarrow[\mathcal{B}]{E_{1,1} \ldots E_{1,m_1} \ldots E_{k,1} \ldots E_{k,m_k}} t$. Applying Lemma 8 (d) repeatedly yields states $s_0, \ldots, s_k$ and a decomposition $\mathcal{B} = \mathcal{B}_1 \cup \ldots \cup \mathcal{B}_k$ s.t. $s_0 = s$, $s_k = t$, and for all $i = 1, \ldots, k$: $s_{i-1} \xrightarrow[\mathcal{B}_i]{E_{i,1} \ldots E_{i,m_i}} s_i$. Hence, we have, for $i = 1, \ldots, k$: $\langle s_{i-1}, \mathcal{B}_i, s_i \rangle \in \overrightarrow{C_i}$. Since $n_i < n$ for all $i = 1, \ldots, k$, we can use the hypothesis on each of them to get $j_1, \ldots, j_k \in \mathbb{N}$ with $\langle s_{i-1}, \mathcal{B}_i, s_i \rangle \in \overrightarrow{C_i}^{j_i}$.
Now define $\mathcal{D} := \{(s_0, D_1, s_1), \ldots, (s_{k-1}, D_k, s_k)\}$. Note that, just because $s_0 = s$ and $s_k = t$, we have $s \xrightarrow[\mathcal{D}]{D_1 \ldots D_k} t$ and therefore $\langle s, \mathcal{D}, t \rangle \in \overrightarrow{B}$. Since $n' < n$ we can now use the hypothesis to obtain a $j'$ with $\langle s, \mathcal{D}, t \rangle \in \overrightarrow{B}^{j'}$. Let $\mathcal{C} := \{(s_0, C_1, s_1), \ldots, (s_{k-1}, C_k, s_k)\}$. Then, by construction we have $\langle s, \mathcal{C}, t \rangle \in \overrightarrow{A}^{j'+1,\text{push}}$ and therefore $\langle s, \mathcal{C}, t \rangle \in \overrightarrow{A}^{j'+1}$.
Finally, let $j := \max\{j' + 1, j_1, \ldots, j_k\}$. Lemma 10, together with the above, shows that for all $i = 1, \ldots, k$ we have $\langle s_{i-1}, \mathcal{B}_i, s_i \rangle \in \overrightarrow{C_i}^{j}$. By construction, we then have $\langle s, \mathcal{B}, t \rangle \in \overrightarrow{A}^{j+1,\text{ins}}$ which finishes the proof. □

**Theorem 36** For all $A \in N$: $\overrightarrow{A} = \bigcup_{j \in \mathbb{N}} \overrightarrow{A}^j$.

Proof By Lemmas 9 and 11. □

Our next concern is the constructability of $\bigcup_{j \in \mathbb{N}} \overrightarrow{A}^j$ for any $A \in N$. We start be remarking that the number of approximation steps required for the construction is finite.

**Lemma 12 (Termination)** For all $A \in N$ there is a $j \in \mathbb{N}$ s.t. for all $j' > j$ we have $\overrightarrow{A}^{j'} = \overrightarrow{A}^j$. Moreover, $j \leq |\mathcal{S}|^2 \times 2^{|\mathcal{S}|^2 \cdot |N|}$.

Proof This follows from Lemma 10 and the fact that for all $j$, $\overrightarrow{A}^j \subseteq \mathcal{S} \times 2^{\mathcal{S} \times N \times \mathcal{S}} \times \mathcal{S}$. □

**Lemma 13 (Running Time)** For any $A \in N$ it is possible to compute $\overrightarrow{A}$ in time $\mathcal{O}(|G|^2 \cdot |N| \cdot |\mathcal{S}|^6 \cdot 2^{3|\mathcal{S}|^2 \cdot |N|})$.

PROOF According to Thm. 36 and Lemma 12, it suffices to compute $\overset{\dashrightarrow}{A}^0, \overset{\dashrightarrow}{A}^1, \ldots$ until stability is reached. Lemma 12 also states that at most $|\mathcal{S}|^2 \cdot 2^{|\mathcal{S}|^2 \cdot |N|}$ many iterations are needed. Remember though, that this has to be done simultaneously for all $A \in N$, which adds another factor $|N|$ to the running time. Finally, for any $A \in N$ and any $j > 0$, computing

- $\overset{\dashrightarrow}{A}^0$ takes time $\mathcal{O}(|S|^2 \cdot |N|)$,
- $\overset{\dashrightarrow}{A}^{j,\text{conc}}$ takes time $\mathcal{O}(|G| \cdot (|\mathcal{S}|^2 \cdot 2^{|\mathcal{S}|^2 \cdot |N|})^2)$,
- $\overset{\dashrightarrow}{A}^{j,\text{push}}$ takes time $\mathcal{O}(|G| \cdot (|\mathcal{S}|^2 \cdot 2^{|\mathcal{S}|^2 \cdot |N|}) \cdot (|S|^2 \cdot |N| \cdot |G|))$,
- $\overset{\dashrightarrow}{A}^{j,\text{ins}}$ takes time $\mathcal{O}((|\mathcal{S}|^2 \cdot 2^{|\mathcal{S}|^2 \cdot |N|}) \cdot |S|^2 \cdot |N|)$,

assuming that set operations take time $\mathcal{O}(1)$ because sets are represented as boolean arrays for example, and that $\overset{\dashrightarrow}{B}^{j-1}$ have already been computed for all $B \in N$.
Putting these all together amounts to $\mathcal{O}(|G|^2 \cdot |N| \cdot |\mathcal{S}|^6 \cdot 2^{3|\mathcal{S}|^2 \cdot |N|})$. □

Remember that $\overset{\dashrightarrow}{A}$ contains triples $\langle s, \mathcal{B}, t\rangle$ s.t. there is a path from $s$ to $t$ whose label is derivable from $A$ and which is made from elements in $\mathcal{B}$. These are triples of the form $(u, B, v)$ with the intuitive meaning that the path from $s$ to $t$ can use a subpath from $u$ to $v$ if it is possible to find one that can be derived from $B$. In the end we are of course interested in *closed paths* from $s$ to $t$, i.e. those that do not contain holes like the ones between $u$ and $v$ anymore. These holes can be closed by considering terminal productions now. Remember that Cor. 7 showed that in a derivation of a word it is always possible to defer the use of terminal productions to the very end, i.e. if $A \Rightarrow^* w$ for some $w \in \Sigma^*$ then also

$$A \Rightarrow^* E_1 \ldots E_k \Rightarrow^* w$$

for some $k \in \mathbb{N}$ s.t. the first part before $E_1 \ldots E_k$ does not contain terminal productions, and the second part only contains terminal productions. Here we also make use of the fact that a terminal can only be derived from an unindexed nonterminal.
The next definition captures the intuition of closed paths from a state to another.

**Definition 36 (Closed Path)** For each $A \in N$, define:

$$\overrightarrow{A} \;:=\; \{\ (s,t) \mid s,t \in \mathcal{S} \text{ and there is } w \in \Sigma^* \text{ s.t. } A \Rightarrow^+ w \text{ and } s \xrightarrow{w} t\}.$$

Approximating this set of closed paths is easier than the above set of open paths. We can define approximations such that the second of these already captures the entire $\overrightarrow{A}$. We define simultaneously for all $A \in N$:

$$\begin{aligned}
\overrightarrow{A}^0 &:= \{(s,s) \mid s \in \mathcal{S}, A \to \epsilon\} \\
&\cup \{(s,t) \mid s,t \in \mathcal{S} \text{ and there is } a \in \Sigma \text{ s.t. } A \to a \text{ and } s \xrightarrow{a} t\}, \\
\overrightarrow{A}^1 &:= \overrightarrow{A}^0 \\
&\cup \{(s,t) \mid \text{there is } \mathcal{B} \subseteq \mathcal{N} \text{ s.t. } (s,\mathcal{B},t) \in \overset{\dashrightarrow}{A} \text{ and} \\
&\qquad \text{for all } (u,D,v) \in \mathcal{B} : (u,v) \in \overrightarrow{D}^0\}.
\end{aligned}$$

Again, we need to show soundness and completeness w.r.t. $\overrightarrow{A}$.

**Lemma 14 (Soundness)** For all $A \in N$ and we have $\overrightarrow{A}^1 \subseteq \overrightarrow{A}$.

PROOF We show this simultaneously for all $A \in N$. Let $(s,t) \in \overrightarrow{A}^1$. There are two cases. If $(s,t) \in \overrightarrow{A}^0$ then the claim follows immediately. Suppose therefore that there is a $\langle s,\mathcal{B},t\rangle \in \overset{\dashrightarrow}{A}$, i.e. there is an $\alpha \in N^+$ with $A \Rightarrow^* \alpha$ and $s \xrightarrow[\mathcal{B}]{\alpha} t$, and that for every $(u,D,v) \in \mathcal{B}$ we have $(u,v) \in \overrightarrow{D}^0$. Then Lemma 8 (e) yields a $w \in \Sigma^*$ s.t. $\alpha \Rightarrow^+ w$ and $s \xrightarrow{w} t$. Hence, we have $A \Rightarrow^+ w$ and therefore $(s,t) \in \overrightarrow{A}$. □

**Lemma 15 (Completeness)** For all $A \in N$ we have $\overrightarrow{A} \subseteq \overrightarrow{A}^1$.

PROOF Suppose $(s,t) \in \overrightarrow{A}$. Then there is a $w \in \Sigma^*$, s.t. $A \Rightarrow^+ w$ and $s \xrightarrow{w} t$. We consider the derivation of $w$ from $A$. Clearly, every symbol $a$ in $w$ is derived in an application of a rule $A \to a$ s.t. $A$ occurs with empty index in a sentential form in this derivation. Furthermore, there can be applications of rule $A \to \epsilon$, again, on empty index only. Cor. 7 gives us $E_1,\dots,E_k \in N$ s.t.

$$A \Rightarrow^* E_1 \dots E_k \Rightarrow^* w$$

and in the left part no terminal productions are used and in the right part only terminal productions are used. Let $w = a_1 \dots a_m$. Note that we must have $k \geq m$, i.e. some of the $E_i$ can be deleted in applications of the form $E_i \to \epsilon$, but no single occurrence of an $E_i$ can derive more than one terminal symbol $a_j$ in $w$. Hence, each of these nonterminals $E_i$ is either *nulling*, i.e. it is deleted in an application of a rule $E_i \to \epsilon$, otherwise it is *non-nulling* and derives a terminal symbol in $w$. Let $E_{i_1} \dots E_{i_m}$ be the subsequence of $E_1 \dots E_k$ that consists exactly of the non-nulling nonterminals in it.

Since $s \xrightarrow{w} t$ there are $s_0, \dots, s_m \in \mathcal{S}$ s.t. $s_{i-1} \xrightarrow{a_i} s_i$ for every $i = 1, \dots, m$ and $s_0 = s$ and $s_m = t$. Let $\mathcal{C} = \{(s_0, E_{i_1}, s_1), \dots, (s_{m-1}, E_{i_m}, s_m)\}$. Then we have $s \xrightarrow[\mathcal{C}]{E_{i_1} \dots E_{i_m}} t$. Let

$$\mathcal{B} := \mathcal{C} \cup \{(s_i, E_i, s_i) \mid E_i \text{ is nulling in the sequence above }\}$$

First one can repeatedly apply Lemma 8 (d) in order to decompose $s \xrightarrow[\mathcal{C}]{E_{i_1} \dots E_{i_m}} t$ into a sequence $u \xrightarrow[\{(u,E_{i_j},v)\}]{E_{i_j}} v$ for $j = 1, \dots, m$. Then one can use Lemma 8 (a) in order to recompose it into $s \xrightarrow[\mathcal{B}]{E_1 \dots E_k} t$. Hence, we have $\langle s, \mathcal{B}, t \rangle \in \dashrightarrow{A}$. Furthermore, note that for every $(u, E, v) \in \mathcal{B}$ we have $(u, v) \in \overrightarrow{E}^0$: each $u, v$ have been chosen such that either

- $u \xrightarrow{a} v$ and $E \rightarrow a$ for some $a \in \Sigma$, or
- $u = v$ and $E \rightarrow \epsilon$.

By the construction, we then have $(s, t) \in \overrightarrow{A}^1$. □

**Lemma 16 (Running Time)** For all $A \in N$, it is possible to compute $\overrightarrow{A}$ in time $\mathcal{O}(|G|^2 \cdot |N| \cdot |\mathcal{S}|^6 \cdot 2^{3|\mathcal{S}|^2 \cdot |N|})$.

Proof Clearly, computing $\overrightarrow{A}^0$ for an $A \in N$ takes time $\mathcal{O}(|S|^2 \cdot |G|)$, and once this is done for all $A \in N$, $\overrightarrow{A}^1$ can be computed in time $\mathcal{O}(|\mathcal{S}|^2 \cdot 2^{|\mathcal{S}|^2 \cdot |N|} \cdot |\mathcal{S}|^2 \cdot |N|)$. Both are superseded by the time it takes to compute $\dashrightarrow{A}$ for all $A$ which is required in advance anyway. Hence, the result follows from Lemma 13. □

**Theorem 37** The model checking problem for PDL[IL] is in EXPTIME.

Proof We reprove this theorem by showing that the algorithm MC-PDL is in EXPTIME, where the subroutine `reach` is implemented as the computation of the closed paths set as described above. Remember that the worst-case scenario for MC-PDL is that `reach` is called $|\varphi| - 1$ times for some PDL[IL] formula $\varphi$ since the computation of the whole semantics can be decomposed into the subsequent computation of the semantics of subformulas and furthermore the most expensive kind of subformula is $\langle L \rangle \psi$ whose semantics is computed by `reach`.

Remember also that `reach`$(L, V)$ takes as parameters an indexed language $L$ (here given as an indexed grammar) and a set $V$ representing the precomputed set of states in which a subformula $\psi$ holds.

According to Lemma 16, given an indexed grammar $G$ with nonterminals $N$ and starting symbol $S$, and a transition system $\mathcal{T}$ with states $\mathcal{S}$ and a $T \subseteq \mathcal{S}$, one can compute $\overrightarrow{S}$ in

time $\mathcal{O}(|G|^2 \cdot |N| \cdot |\mathcal{S}|^6 \cdot 2^{3|\mathcal{S}|^2 \cdot |N|})$. Then one checks in time $|\mathcal{S}|^2$ for which $s \in \mathcal{S}$ the set $\overrightarrow{S}$ contains an element $(s, t)$ with $t \in V$ and returns those $s$.

As stated above, this is done at most $|\varphi| - 1$ times. Clearly, the time consumed is then exponential in both the grammar and the transition system. □

### 3.5.3 A Model Checking Algorithm for PDL over MCSL

In this section we turn our attention to a concrete implementation of `reach` in the model checking algorithm MC-PDL for PDL[MCSL]. This is particularly interesting, because despite the fact that MCSL can already be considered a rather powerful language class, the model checking problem of PDL[MCSL] is still solvable in PTIME (see Cor. 5) just like the model checking problem for PDL[CFL].

First of all, note that the reason for the exponential model checking for PDL[IL] is the representation of sets of open paths through a triple $\langle s, \mathcal{B}, t\rangle$ in which $\mathcal{B}$ itself is a set of annotated nonterminals of which there are exponentially many. If one could restrict that number to a polynomial in the number of states $\mathcal{S}$ and the size of the underlying grammar $G$ then the result would be a polynomial model checking procedure. In the following, we will show that this is the case for LIL.

For the remainder of this section we fix, again, a LIG $G = (N, \Sigma, I, P, S)$ and a Kripke structure $\mathcal{T} = (\mathcal{S}, \rightarrow, \ell)$.

Before we can proceed with a procedure for the LIL-reachability problem, we need some technical lemmas. First of all, note that Lemma 7 (commutativity of pairwise application of production rules) also holds for the derivation relation in linear indexed grammars.

**Lemma 17** For all $A, B \in N$, all $\delta \in I^*$ and all $w_1, w_2 \in \Sigma^*$:

a) If $\widehat{A} \Rightarrow^* w_1 \widehat{B} w_2$ then $\widehat{A[\delta]} \Rightarrow^* w_1 \widehat{B[\delta]} w_2$.

b) If $\widehat{A[\delta]} \Rightarrow^k w_1 \widehat{B[\delta]} w_2$ for some $k \geq 0$ and for all marked indexed nonterminals $\widehat{X[\delta']}$ occurring during the derivation, $\delta' = \gamma\delta$ for some $\gamma \in I^*$ holds then $\widehat{A} \Rightarrow^k w_1 \widehat{B} w_2$.

c) If $\widehat{A} \Rightarrow^* w_1 \widehat{B} w_2$ then $A \Rightarrow^* w_1 B w_2$.

d) If $A \Rightarrow^* \alpha$ and $\alpha \in N^+$ then there exists $B \in N$, s.t. $\widehat{A} \Rightarrow^* \alpha_1 \widehat{B} \alpha_2$ and $\alpha_1 B \alpha_2 = \alpha$.

e) If $A \Rightarrow^k w$ for a $w \in \Sigma^*$ and $k > 1$ then there exist $v_1, v_2, v_3 \in \Sigma^*$, $k_1 < k$, $k_2 < k$ and $B \in N$, s.t. $\widehat{A} \Rightarrow^{k_1} v_1 \widehat{B} v_3$ and $B \Rightarrow^{k_2} v_2$, s.t. $w = v_1 v_2 v_3$.

PROOF For part (a) note that a simulation of the rules used during the derivation $\widehat{A} \Rightarrow^* w_1\widehat{B}w_2$ can be done on $\widehat{A[\delta]}$, apparently leaving the index $\delta$ untouched at the bottom of all index transformations.

The same holds in part (b), since it is required that the index $\delta$ is always at the bottom and hence no pop productions which go below the empty index can be performed in the simulation of $\widehat{A[\delta]} \Rightarrow^* w_1\widehat{B[\delta]}w_2$.

Part (c) and (d) are straightforward.

For part (e), first notice that since Lemma 7 is applicable for LIG, it is possible to postpone the application of terminal productions in a derivation to the end, s.t. $A \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n \Rightarrow^{|\alpha_n|} w$ for some $n \in \mathbb{N}$ and for all $1 \leq i \leq n$, $\alpha_i \in (N \times I^*)^*$. Hence $\alpha_n \in N^+$. From application of part (d) of this lemma follows that there exist $\beta_1, \beta_2 \in N^*, B \in N$, s.t. $\widehat{A} \Rightarrow^* \beta_1\widehat{B}\beta_2$ and $\beta_1 B \beta_2 = \alpha_n$. Clearly, $w$ may be partitioned into $w_1, w_2, w_3$, s.t. $\beta_1 \Rightarrow^* w_1$, $B \Rightarrow w_2$ and $\beta_2 \Rightarrow^* w_3$. We now have proven $\widehat{A} \Rightarrow^{k_1} w_1\widehat{B}w_3$ and $B \Rightarrow w_2$, hence $k_2 = 1 < k$. But since $k > 1$ and $k = k_1 + k_2$, we have that $k_1 < k$, too. □

### Solving the LIL-Reachability Problem

As for the IL-reachability problem, we will devise a procedure that solves the reachability problem for LIL by characterising, for each nonterminal $A$ the pairs of states $s, t$ for which there is an *open path* from $s$ to $t$ whose label can be derived from $A$. However, since the index of a nonterminal can only be passed on to a single nonterminal in any application of a rule, we can restrict our attention to paths with a single hole only.

For example, an open path from $s$ to $t$ may be characterised by two states $s', t'$ and a nonterminal $B$. The intuitive meaning is the following: there are sentential forms $\beta, \gamma$ s.t. $\widehat{A} \Rightarrow^* \beta\widehat{B}\gamma$ and $\widehat{B}$ would inherit a stack from $\widehat{A}$, and $s \xrightarrow{\beta} s'$, $t' \xrightarrow{\gamma} t$, and between $s'$ and $t'$ there is a hole which has to be closed by something derivable from $B$. The crucial observation now is that nothing in $\beta$ or $\gamma$ does inherit a stack from $\widehat{A}$. Therefore, we can assume these parts to be derived to terminal symbols already. This, however, means that we need to define simultaneously the sets of open and closed paths derivable from a given nonterminal because they mutually depend on each other.

Furthermore, this observation explains the claim of polynomial boundedness of the sets of annotated nonterminals in the introductory part above: for an IL, the set $\mathcal{B}$ representing all legal parts of an open path in a triple $\langle s, \mathcal{B}, t\rangle$ boils down to a singleton set $\{(u, B, v)\}$ now. We therefore write those triples simply as $\langle s, u, B, v, t\rangle$ with $s, t \in \mathcal{S}$ and $(u, B, v) \in \mathcal{N} := \mathcal{S} \times N \times \mathcal{S}$.

**Definition 37 (Open/ Closed Path)** For each $A \in N$, define:

$$
\begin{aligned}
\overset{\dashrightarrow}{A} &:= \{\langle s,u,B,v,t\rangle \mid \text{there are } w_1,w_2 \in \Sigma^*. \text{ s.t. } \widehat{A} \Rightarrow^* w_1\widehat{B}w_2 \text{ and} \\
&\qquad s \xrightarrow{w_1} u \text{ and } v \xrightarrow{w_2} t\}, \\
\overrightarrow{A} &:= \{(s,t) \mid \text{there is } w \in \Sigma^* \text{ s.t. } A \Rightarrow^+ w \text{ and } s \xrightarrow{w} t\}.
\end{aligned}
$$

Next we define, for all $A \in N$, sets $\overset{\dashrightarrow}{A}^0, \overset{\dashrightarrow}{A}^1, \ldots \subseteq \mathcal{S} \times \mathcal{N} \times \mathcal{S}$ and $\overrightarrow{A}^0, \overrightarrow{A}^1, \ldots \subseteq \mathcal{S} \times \mathcal{S}$ for which we will show that they approximate $\overset{\dashrightarrow}{A}$ and, resp., $\overrightarrow{A}$. The two base cases are as follows.

$$
\begin{aligned}
\overset{\dashrightarrow}{A}^0 &:= \{\langle s,s,A,t,t\rangle \mid s,t \in \mathcal{S}\}, \\
\overrightarrow{A}^0 &:= \{(s,s) \mid s \in \mathcal{S}, A \to \epsilon\} \\
&\quad \cup \{(s,t) \mid \text{there is } a \in \Sigma \text{ s.t. } A \to a \text{ and } s \xrightarrow{a} t\}.
\end{aligned}
$$

Now let $j > 0$. As above, the set of open paths at level $j$ includes the set of open paths at level $j-1$ and closes it off under applications of composite and push productions as well as insertion of open paths into the holes of other open paths.

$$
\overset{\dashrightarrow}{A}^j \;:=\; \overset{\dashrightarrow}{A}^{j-1} \;\cup\; \overset{\dashrightarrow}{A}^{j,\text{conc}} \;\cup\; \overset{\dashrightarrow}{A}^{j,\text{push}} \;\cup\; \overset{\dashrightarrow}{A}^{j,\text{ins}}
$$

where

$$
\begin{aligned}
\overset{\dashrightarrow}{A}^{j,\text{conc}} &:= \{\ \langle s,t,D,u,v\rangle \mid \text{there are } B,C \in N, v' \in \mathcal{S} \text{ s.t.} \\
&\qquad A \to \widehat{B}C \text{ and } \langle s,t,D,u,v'\rangle \in \overset{\dashrightarrow}{B}^{j-1} \text{ and } (v',v) \in \overrightarrow{C}^{j-1}, \text{ or} \\
&\qquad A \to B\widehat{C} \text{ and } (s,v') \in \overrightarrow{B}^{j-1} \text{ and } \langle v',t,D,u,v\rangle \in \overset{\dashrightarrow}{C}^{j-1}\ \}, \\
\overset{\dashrightarrow}{A}^{j,\text{push}} &:= \{\ \langle s,t,D,u,v\rangle \mid \text{there are } B,C \in N, f \in I, \text{ s.t. } A \to B[f], \\
&\qquad C[f] \to D \text{ and } \langle s,t,C,u,v\rangle \in \overset{\dashrightarrow}{B}^{j-1}\ \}, \\
\overset{\dashrightarrow}{A}^{j,\text{ins}} &:= \{\ \langle s,t,D,u,v\rangle \mid \text{there is } (t',B,u') \in \mathcal{N} \text{ with } \langle s,t',B,u',v\rangle \in \overset{\dashrightarrow}{A}^{j-1} \\
&\qquad \text{and } \langle t',t,D,u,u'\rangle \in \overset{\dashrightarrow}{B}^{j-1}\ \}.
\end{aligned}
$$

Furthermore,

$$
\begin{aligned}
\overrightarrow{A}^j &:= \overrightarrow{A}^{j-1} \\
&\quad \cup \{(s,t) \mid \text{there is } (u,B,v) \in \mathcal{N} \text{ with } \langle s,u,B,v,t\rangle \in \overset{\dashrightarrow}{A}^{j-1} \text{ and} \\
&\qquad (u,v) \in \overrightarrow{B}^{j-1}\ \}.
\end{aligned}
$$

**Lemma 18 (Soundness)** For all $A \in N$ and for all $j \in \mathbb{N}$ we have

a) $\overrightarrow{A}^j \subseteq \overrightarrow{A}$, and

b) $\overset{\dashrightarrow}{A}^j \subseteq \overset{\dashrightarrow}{A}$.

PROOF We prove both parts by simultaneous induction on $j$ for all $A \in N$. The base case of (a), $\overrightarrow{A}^0 \subseteq \overrightarrow{A}$, is immediate. The base case of (b), $\overset{\dashrightarrow}{A}^0 \subseteq \overset{\dashrightarrow}{A}$, is not much more difficult. Note that $\widehat{A} \Rightarrow^* \widehat{A}$ and $s \xrightarrow{\epsilon} s$.

Now suppose $j > 0$. For part (a) suppose that $(s,t) \in \overrightarrow{A}^j$. Then there are two cases. If $(s,t) \in \overrightarrow{A}^{j-1}$ then we simply have $(s,t) \in \overrightarrow{A}$ by hypothesis. Hence, assume that there is $(u,B,v) \in \mathcal{N}$ s.t. $\langle s,u,B,v,t\rangle \in \overset{\dashrightarrow}{A}^{j-1}$ and $(u,v) \in \overrightarrow{B}^{j-1}$. The hypothesis for (b) yields $\langle s,u,B,v,t\rangle \in \overset{\dashrightarrow}{A}$, i.e. there is are $w_1, w_2 \in \Sigma^*$ s.t. $\widehat{A} \Rightarrow^* w_1 \widehat{B} w_2$ and $s \xrightarrow{w_1} u$ and $v \xrightarrow{w_2} t$. Then by Lemma 17 (c) we also have $A \Rightarrow^* w_1 B w_2$. Furthermore, the hypothesis for (a) yields $(u,v) \in \overrightarrow{B}$, i.e. there is a $w \in \Sigma^*$ s.t. $B \Rightarrow^+ w$ and $u \xrightarrow{w} v$. Hence, we have $A \Rightarrow^+ w_1 w w_2$ and $s \xrightarrow{w_1 w w_2} t$ and therefore $(s,t) \in \overrightarrow{A}$.

For part (b) suppose $\langle s,s',D,t',t\rangle \in \overset{\dashrightarrow}{A}^j$. We need to distinguish four cases. The first case of $\langle s,s',D,t',t\rangle \in \overset{\dashrightarrow}{A}^{j-1}$ trivially follows from the hypothesis.

Case $\langle s,s',D,t',t\rangle \in \overset{\dashrightarrow}{A}^{j,\text{conc}}$. Then there are $B, C \in N$, $u \in \mathcal{S}$ s.t. $A \to \widehat{B}C$ and $\langle s,s',D,t',u\rangle \in \overset{\dashrightarrow}{B}^{j-1}$ and $(u,t) \in \overrightarrow{C}^{j-1}$, or $A \to B\widehat{C}$ and $(s,u) \in \overrightarrow{B}^{j-1}$ and $\langle u,s',D,t',t\rangle \in \overset{\dashrightarrow}{C}^{j-1}$. Suppose the former is the case – the latter is entirely dual, we therefore omit that subcase here. Then, by hypothesis for part (b) we have $\langle s,s',D,t',u\rangle \in \overset{\dashrightarrow}{B}$ which yields $w_1, w_2 \in \Sigma^*$ with $\widehat{B} \Rightarrow^* w_1 \widehat{D} w_2$, $s \xrightarrow{w_1} s'$, and $t' \xrightarrow{w_2} u$. Furthermore, the hypothesis for part (a) yields $(u,t) \in \overrightarrow{C}$, i.e. there is a $w \in \Sigma^*$ s.t. $C \Rightarrow^+ w$ and $u \xrightarrow{w} t$. Putting these together yields $\widehat{A} \Rightarrow \widehat{B}C \Rightarrow^+ w_1 \widehat{D} w_2 w$ and $t' \xrightarrow{w_2 w} t$. Hence, we have $\langle s,s',D,t',t\rangle \in \overset{\dashrightarrow}{A}$.

Case $\langle s,s',D,t',t\rangle \in \overset{\dashrightarrow}{A}^{j,\text{push}}$. Then there are $B, C \in N$ and $f \in I$ s.t. $A \to B[f]$, $C[f] \to D$ and $\langle s,s',C,t',t\rangle \in \overset{\dashrightarrow}{B}^{j-1}$. By hypothesis, we have $\langle s,s',C,t',t\rangle \in \overset{\dashrightarrow}{B}^{j-1}$ i.e. there are $w_1, w_2 \in \Sigma^*$ with $\widehat{B} \Rightarrow^* w_1 \widehat{C} w_2$ and $s \xrightarrow{w_1} s'$ and $t' \xrightarrow{w_2} t$. According to Lemma 17 (a) we also have $\widehat{B[f]} \Rightarrow^* w_1 \widehat{C[f]} w_2$ and therefore

$$\widehat{A} \Rightarrow \widehat{B[f]} \Rightarrow^* w_1 \widehat{C[f]} w_2 \Rightarrow w_1 \widehat{D} w_2$$

which shows that $\langle s,s',D,t',t\rangle \in \overset{\dashrightarrow}{A}$.

Finally, suppose $\langle s, s', D, t', t\rangle \in \overset{\dashrightarrow}{A}^{j,\mathrm{ins}}$. Then there is a $(u, B, v) \in \mathcal{N}$ s.t. $\langle s, u, B, v, t\rangle \in \overset{\dashrightarrow}{A}^{j-1}$ and $\langle u, s', D, t', v\rangle \in \overset{\dashrightarrow}{B}^{j-1}$. Applying the hypothesis twice yields $\langle s, u, B, v, t\rangle \in \overset{\dashrightarrow}{A}$ and $\langle u, s', D, t', v\rangle \in \overset{\dashrightarrow}{B}$, i.e. there are $w_1, w_2, w_1', w_2' \in \Sigma^*$ s.t. $\widehat{A} \Rightarrow^* w_1\widehat{B}w_2$, $s \xrightarrow{w_1} u$, $v \xrightarrow{w_2} t$, and $\widehat{B} \Rightarrow^* w_1'\widehat{D}w_2'$, $u \xrightarrow{w_1'} s'$ and $t' \xrightarrow{w_2'} v$. Putting these together yields $\widehat{A} \Rightarrow^* w_1\widehat{B}w_2 \Rightarrow^* w_1w_1'\widehat{D}w_2'w_2$ and $s \xrightarrow{w_1w_1'} s'$ and $t' \xrightarrow{w_2'w_2} t$. Hence, we have $\langle s, s', D, t', t'\rangle \in \overset{\dashrightarrow}{A}$ which finishes the proof. □

**Lemma 19 (Monotonicity)** For all $A \in N$ and all $j, j' \in \mathbb{N}$ we have: $j \leq j'$ implies $\overset{\dashrightarrow}{A}^j \subseteq \overset{\dashrightarrow}{A}^{j'}$ and $\overrightarrow{A}^j \subseteq \overrightarrow{A}^{j'}$.

Proof Trivial. □

**Lemma 20 (Completeness)** For all $A \in N$ exists $j \in \mathbb{N}$ s.t.

a) $\overrightarrow{A} \subseteq \overrightarrow{A}^j$, and

b) $\overset{\dashrightarrow}{A} \subseteq \overset{\dashrightarrow}{A}^j$.

Proof Because of Lemma 19 it suffices to show for every $(s', t') \in \overrightarrow{A}$ that there is a $j \in \mathbb{N}$ with $(s', t') \in \overrightarrow{A}^j$ and likewise for every $\langle s, u, E, v, t\rangle \in \overset{\dashrightarrow}{A}$.
So let $(s', t') \in \overrightarrow{C}$ and $\langle s, u, E, v, t\rangle \in \overset{\dashrightarrow}{A}$ for arbitrary $A, C, E \in N$ and $s, s', t, t', u, v \in \mathcal{S}$. From the definition of these it follows that

- there is a $w \in \Sigma^*$ s.t. $A \Rightarrow^k w$ and $s' \xrightarrow{w} t'$ for some $k \geq 1$, and
- there are $w_1, w_2 \in \Sigma^*$ s.t. $\widehat{A} \Rightarrow^m w_1\widehat{E}w_2$ for some $m \geq 0$ and $s \xrightarrow{w_1} u$ and $v \xrightarrow{w_2} t$.

We prove both parts by simultaneous induction on $k$ and $m$.
In the base case for part (a) we assume that $k = 1$ and hence either $w = a$ or $w = \epsilon$. In the former case we have $A \rightarrow a$ and therefore $(s', t') \in \overrightarrow{A}^0$. In the latter case we have $A \rightarrow \epsilon$ and therefore $s' = t'$ and $(s', s') \in \overrightarrow{A}^0$.
The base case for part (b), where $m = 0$, requires $\widehat{A} = \widehat{E}$ and $w_1 = w_2 = \epsilon$ and therefore $s = u$ and $v = t$. But then $\langle s, s, A, t, t\rangle \in \overset{\dashrightarrow}{A}^0$.
For part (a) we now assume $k > 1$ and have $A \Rightarrow^k w$ and $s' \xrightarrow{w} t'$. Lemma 17 (e) yields $w_1, w_2, w_3 \in \Sigma^*$, $k_1 < k$, $k_2 < k$ and a $D \in N$, s.t. $\widehat{A} \Rightarrow^{k_1} w_1\widehat{D}w_3 \Rightarrow^{k_2} w_1w_2w_3$ and $w = w_1w_2w_3$. From this follows that $D \Rightarrow^{k_2} w_2$. This means there exist $u', v' \in \mathcal{S}$, s.t.

$s' \xrightarrow{w_1} u'$, $u' \xrightarrow{w_2} v'$ and $v' \xrightarrow{w_3} t'$. By hypthesis there are $i, i' \in \mathbb{N}$ with $\langle s', u', D, v', t' \rangle \in \overset{\dashrightarrow}{A}^{i}$ and $(u', v') \in \overrightarrow{D}^{i'}$. Therefore, we have $(s', t') \in \overrightarrow{A}^{1+\max\{i,i'\}}$.

For part (b) we assume $m > 0$, have $\widehat{A} \Rightarrow^m w_1 \widehat{E} w_2$ and $s \xrightarrow{w_1} u$ and $v \xrightarrow{w_2} t$. We make a case distinctions on the type of production that is applied in the first step of this derivation. Since $m \geq 1$ and $D \in N$, it cannot be a terminal production. It cannot be a pop production either, because $A$ has an empty index. Hence, it must either be a composite of a push production.

Case $\widehat{A} \Rightarrow \widehat{B}C \Rightarrow^{m-1} w_1 \widehat{E} w_2$, i.e. $A \rightarrow \widehat{B}C$. Then there are $x_1, x_2 \in \Sigma^*$, s.t. $\widehat{B} \Rightarrow^{m'} w_1 \widehat{E} x_1$ and $C \Rightarrow^{m''} x_2$ with $m', m'' < k$ and $w_2 = x_1 x_2$. Furthermore, there also is a $v' \in \mathcal{S}$, s.t. $u \xrightarrow{x_1} v'$ and $v' \xrightarrow{x_2} v$. Using the hypothesis for both parts (a) and (b) yields $j_1, j_2 \in \mathbb{N}$, s.t. $\langle s, t, E, u, v' \rangle \in \overset{\dashrightarrow}{B}^{j_1}$ and $(v', v) \in \overrightarrow{C}^{j_2}$. Hence, we have $\langle s, t, E, u, v \rangle \in \overset{\dashrightarrow}{A}^{1+\max\{j_1,j_2\},\mathrm{conc}}$, and therefore $\langle s, t, E, u, v \rangle \in \overset{\dashrightarrow}{A}^{1+\max\{j_1,j_2\}}$. The case of $\widehat{A} \Rightarrow B\widehat{C} \Rightarrow^{m-1} w_1 \widehat{E} w_2$ is entirely symmetric.

Case $\widehat{A} \Rightarrow \widehat{B[f]} \Rightarrow^{m-1} w_1 \widehat{E} w_2$. Since the index of $\widehat{E}$ is empty in this sentential form, the index symbol $f$ must have been popped somewhere during the derivation, i.e. there is a $C \in N$ and sentential forms $\alpha, \beta$ s.t.

$$\widehat{A} \Rightarrow \widehat{B[f]} \Rightarrow \ldots \Rightarrow \alpha \widehat{C[f]} \beta \Rightarrow \alpha \widehat{D} \beta \Rightarrow \ldots \Rightarrow w_1 \widehat{E} w_2.$$

Note that at most one index symbol can be popped per derivation step. Let the production $C[f] \rightarrow D$ be the first one in the above derivation that pops the bottom index symbol $f$ from any marked indexed nonterminal. Thus, all intermediate sentential forms occurring between $\widehat{B[f]}$ and $\alpha \widehat{C[f]} \beta$ in the above derivation, are of the form $\alpha' \widehat{X[\delta' f]} \beta'$ for some $X \in N$ and some $\alpha', \beta'$.

Note furthermore that there must exist $w_1', w_1'', w_2', w_2'' \in \Sigma^*$ s.t. $w_1 = w_1' w_1''$ and $w_2 = w_2' w_2''$ and $\alpha \Rightarrow^* w_1'$ and $\beta \Rightarrow^* w_2''$ and $\widehat{D} \Rightarrow^i w_1'' \widehat{E} w_2'$ with $i < k$. This is because the marker $\widehat{\cdot}$ is laways inherited from a nonterminal in the predecessing sentential form, hence $\widehat{E}$ in $w_1 \widehat{E} w_2$ has inherited it from $\widehat{D}$ in $\alpha \widehat{D} \beta$. There also must exist $s', u' \in \mathcal{S}$, s.t. $s \xrightarrow{w_1'} s'$, $s' \xrightarrow{w_1''} t$, and $u \xrightarrow{w_2'} u'$, $u' \xrightarrow{w_2''} t$.

By the Commutation Lemma 7 we have $\widehat{B[f]} \Rightarrow^{i'} w_1' \widehat{C[f]} w_2''$ with $i' < k$ s.t. for all marked indexed nonterminals $X[\delta]$ occurring during this derivation, $\delta = \delta' f$ for some $\delta' \in I^*$. By Lemma 17 (b) we then have $\widehat{B} \Rightarrow^{i'} w_1' \widehat{C} w_2''$ and the hypothesis yields a $j_1 \in \mathbb{N}$ s.t. $\langle s, s', C, u', v \rangle \in \overset{\dashrightarrow}{B}^{j_1}$. But then $\langle s, s', D, u', v \rangle \in \overset{\dashrightarrow}{A}^{j_1+1,\mathrm{push}}$ and therefore $\langle s, s', D, u', v \rangle \in \overset{\dashrightarrow}{A}^{j_1+1}$.

Because $i < k$, the hypothesis also yields a $j_2 \in \mathbb{N}$, s.t. $\langle s', t, E, u, u' \rangle \in \overset{\dashrightarrow}{D}^{j_2}$. Putting these together we have $\langle s, t, E, u, v \rangle \in \overset{\dashrightarrow}{A}^{1+\max\{j_1+1, j_2\}, \text{ins}}$ and therefore $\langle s, t, E, u, v \rangle \in \overset{\dashrightarrow}{A}^{1+\max\{j_1+1, j_2\}}$ which concludes the proof. □

**Theorem 38** For all $A \in N$: $\overset{\dashrightarrow}{A} = \bigcup_{j \in \mathbb{N}} \overset{\dashrightarrow}{A}^j$ and $\overrightarrow{A} = \bigcup_{j \in \mathbb{N}} \overrightarrow{A}^j$.

Proof By Lemmas 18 and 20. □

The following lemma estimates the number of iterations it takes to approximate $\overset{\dashrightarrow}{A}$ and $\overrightarrow{A}$ from below.

**Lemma 21 (Termination)** For all $A \in N$ there are $j, j' \in \mathbb{N}$ s.t. for all $i > j$ and all $i' > j'$ we have $\overset{\dashrightarrow}{A}^i = \overset{\dashrightarrow}{A}^j$ and $\overrightarrow{A}^{i'} = \overrightarrow{A}^{j'}$. Moreover, $j \leq |\mathcal{S}|^4 \times |N|$ and $j' \leq |\mathcal{S}|^2$.

Proof This follows from Lemma 19 and the fact that for all $j$, $\overset{\dashrightarrow}{A}^j \subseteq \mathcal{S} \times (\mathcal{S} \times N \times \mathcal{S}) \times \mathcal{S}$ and $\overrightarrow{A}^j \subseteq \mathcal{S} \times \mathcal{S}$. □

**Lemma 22** For all $A \in N$, it is possible to compute $\overrightarrow{A}$ in time $\mathcal{O}(|G|^2 \cdot |N|^4 \cdot |\mathcal{S}|^{10})$.

Proof According to Lemma 21, at most $|\mathcal{S}|^4 \cdot |N|$ many iterations are necessary. Each iteration requires the computation of $\overset{\dashrightarrow}{A}^j$ and $\overrightarrow{A}^j$ for some $j$ and every $A \in N$. Hence, at most $|\mathcal{S}|^4 \cdot |N|^2$ many computations of an approximation for a single $A \in N$ are needed. It is not difficult to see that each such computation can be done in worst-case time $\mathcal{O}(|G|^2 \cdot |N|^2 \cdot |\mathcal{S}|^6)$. □

As with indexed grammars above, we can use the approximation of open and closed paths in order to solve the diamond problem for linear indexed languages and therefore the model checking problem for PDL[LIL].

**Theorem 39** The model checking problem for PDL[LIL] is in PTIME.

Proof Similarly as for PDL[IL] we reprove this theorem by showing that the algorithm MC-PDL is in PTIME, where the subroutine `reach` is implemented as the computation of the closed paths set as described above.

According to Lemma 22, given a LIG $G$ with nonterminals $N$ and starting symbol $S$, and a transition system $\mathcal{T}$ with states $\mathcal{S}$ and a $T \subseteq \mathcal{S}$, one can compute $\overrightarrow{S}$ in time $\mathcal{O}(|G|^2 \cdot |N|^4 \cdot |\mathcal{S}|^{10})$. Then one checks in time $|\mathcal{S}|^2$ for which $s \in \mathcal{S}$ the set $\overrightarrow{S}$ contains an element $(s, t)$ with $t \in V$ and returns those $s$. Since there are maximally $|\varphi| - 1$ calls of `reach` in the worst case it follows that the time consumed is polynomial in both the grammar and the transition system. □

# Chapter 4

# Non-Regular Computation Tree Logic

One of the reasons why CTL has gained great popularity and is widely used in hardware verification is that in contrast to logics like $\mathcal{L}_\mu$, it features very intuitive operators and is considered easy to understand. The most common specification properties are usually divided into *safety* and *liveness*, meaning that programs are either required to conform with some invariant holding on all runs at any time or that some desired property should eventually hold. These kinds of properties are explicit language constructs of CTL, realised by $\mathtt{AG}$ and $\mathtt{EF}$. Apart from this, the computational complexity of its model checking as well as satisfiability problems lie within reasonable bounds: model checking is PTIME-complete and satisfiability is EXPTIME-complete [CE81, EH85].

CTL is however very limited in expressive power and can be embedded into the alternation-free fragment of $\mathcal{L}_\mu$. One of the motivations for introducing the following extension of CTL is that it enhances the expressive power of CTL without losing its easy comprehensiveness. Instead of quantifying over arbitrary paths, we allow control over the path structure along which some property is required to hold by adorning the $\mathtt{U}$ and $\mathtt{R}$ operators with formal languages $L$, thereby constraining the quantification to paths which correspond to words $w \in L$. The modular style in which CTL operators are enriched resembles the approach taken in PDL[$\mathcal{L}$] and is therefore consequently named CTL[$\mathcal{L}$] . But since the fragments using solely the $\mathtt{U}^L$ or $\mathtt{R}^L$ operators turn out to be fundamentally different regarding the computational complexity of for instance the model checking problem, it is natural to seperate the classes of languages allowed in each construct. We call the resulting logical framework *parametric* CTL and denote it by CTL[$\mathfrak{A}, \mathfrak{B}$] to emphasise the use of different

language classes $\mathfrak{A}$ and $\mathfrak{B}$ in either operator.

## 4.1 Syntax and Semantics

Like for PDL[$\mathcal{L}$], the question of reasonable language representation arises for CTL[$\mathfrak{A}, \mathfrak{B}$]. In principle, we demand the same, i.e. the existence of finite language representations with a linearly parsable alphabet. It turns out however that the quantification structure of the CTL[$\mathfrak{A}, \mathfrak{B}$] semantics is more complex than for PDL[$\mathcal{L}$] and that in particular the formats are not exchangeable as simple as in PDL[$\mathcal{L}$] with regard to the decision procedures introduced. There is for instance an essential difference in the computational complexity of the model checking problem between deterministic and nondeterministic automata representations which is rooted in the incommutativity of alternating quantifiers on paths and automata runs in the semantics of some operators. For more details see Sec. 4.5.

As a compromise, we define CTL[$\mathfrak{A}, \mathfrak{B}$] independently of the language representations and speak of e.g. CTL[REG], CTL[CFL], etc. whenever the chosen representation is irrelevant and use automata everywhere else. It is clear that the results we obtain are transferable to any other format, if polynomial translations to the respective automata classes exist.

**Definition 38 (Parametric CTL)** Let $\mathcal{P}$ be a countably infinite set of propositions, $\Sigma$ a finite set of actions and $\mathfrak{A}$ and $\mathfrak{B}$ be classes of languages over the alphabet $\Sigma$.
CTL[$\mathfrak{A}, \mathfrak{B}$] is the following set of formulas:

$$\varphi \ ::= \ q \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathtt{E}(\varphi \mathtt{U}^{\mathcal{A}} \varphi) \mid \mathtt{E}(\varphi \mathtt{R}^{\mathcal{B}} \varphi)$$

where $q \in \mathcal{P}$, $\mathcal{A} \in \mathfrak{A}$ and $\mathcal{B} \in \mathfrak{B}$.

Subformulas of CTL[$\mathfrak{A}, \mathfrak{B}$] are defined identically as in PDL[$\mathcal{L}$] for propositional formulas and otherwise as follows.

$$\begin{aligned} \mathsf{sub}(\mathtt{E}(\psi_1 \mathtt{U}^{\mathcal{A}} \psi_2)) &= \{\mathtt{E}(\psi_1 \mathtt{U}^{\mathcal{A}} \psi_2)\} \cup \mathsf{sub}(\psi_1) \cup \mathsf{sub}(\psi_2), \\ \mathsf{sub}(\mathtt{E}(\psi_1 \mathtt{R}^{\mathcal{A}} \psi_2)) &= \{\mathtt{E}(\psi_1 \mathtt{R}^{\mathcal{A}} \psi_2)\} \cup \mathsf{sub}(\psi_1) \cup \mathsf{sub}(\psi_2). \end{aligned}$$

The size of a formula $|\varphi|$ is determined by the number of its subformulas $|\mathsf{sub}(\varphi)|$. We permit the propositional abbreviations $\mathtt{tt}, \mathtt{ff}, \wedge, \rightarrow, \leftrightarrow$ (see section 2.2.6), as well as the

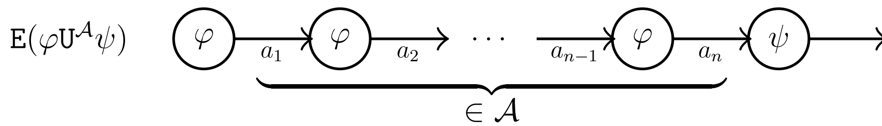
following, where $Q \in \{\mathtt{E}, \mathtt{A}\}$:

$$\begin{array}{lcl}
\mathtt{A}(\varphi \mathtt{U}^{\mathcal{A}} \psi) & := & \neg \mathtt{E}(\neg\varphi \mathtt{R}^{\mathcal{A}} \neg\psi), \\
\mathtt{A}(\varphi \mathtt{R}^{\mathcal{A}} \psi) & := & \neg \mathtt{E}(\neg\varphi \mathtt{U}^{\mathcal{A}} \neg\psi), \\
Q\mathtt{F}^{\mathcal{A}}\varphi & := & Q(\mathtt{tt}\mathtt{U}^{\mathcal{A}}\varphi), \\
Q\mathtt{G}^{\mathcal{A}}\varphi & := & Q(\mathtt{ff}\mathtt{R}^{\mathcal{A}}\varphi), \\
Q\mathtt{F}\varphi & := & Q\mathtt{F}^{\Sigma^*}\varphi, \\
Q\mathtt{G}\varphi & := & Q\mathtt{G}^{\Sigma^*}\varphi, \\
Q\mathtt{X}^{a}\varphi & := & Q\mathtt{F}^{\{a\}}\varphi, \\
Q\mathtt{X}\varphi & := & Q\mathtt{F}^{\Sigma}\varphi.
\end{array}$$

As mentioned above, CTL[$\mathfrak{A}$] is short for CTL[$\mathfrak{A}, \mathfrak{A}$]. Furthermore, we identify the fragments EU[$\mathfrak{A}$], ER[$\mathfrak{A}$], EF[$\mathfrak{A}$] and EG[$\mathfrak{A}$] which are obtained by restricting the use of temporal operators to $\mathtt{E}(\varphi \mathtt{U}^{\mathcal{A}} \varphi)$, $\mathtt{E}(\varphi \mathtt{R}^{\mathcal{A}} \varphi)$, $\mathtt{EF}^{\mathcal{A}}$ and $\mathtt{EG}^{\mathcal{A}}$ respectively for some $\mathcal{A} \in \mathfrak{A}$.
CTL[$\mathfrak{A}, \mathfrak{B}$] formulas are interpreted in states of an LTS $\mathcal{T} = (\mathcal{S}, \rightarrow, \ell)$.

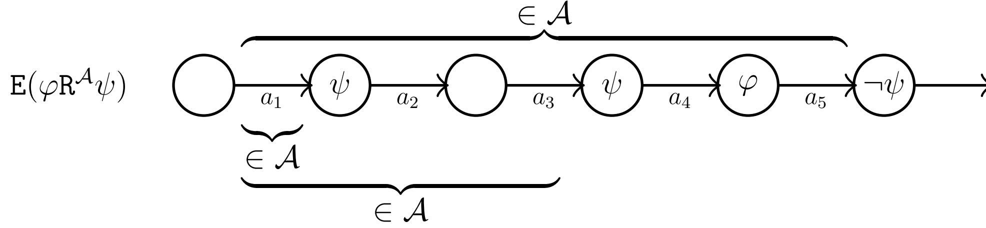
$$\begin{array}{lll}
s \models q & \text{iff} & s \in \ell(q), \\
s \models \neg\varphi & \text{iff} & s \not\models \varphi, \\
s \models \varphi \vee \psi & \text{iff} & s \models \varphi \text{ or } s \models \psi, \\
s \models \mathtt{E}(\varphi \mathtt{U}^{\mathcal{A}} \psi) & \text{iff} & \text{there exists a path } \pi = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_n \\
 & & \text{s.t. } s_0 = s \text{ and } s_n \models \psi \text{ and for all } i < n : \\
 & & s_i \models \varphi \text{ and } a_0 \ldots a_n \in \mathcal{A}, \\
s \models \mathtt{E}(\varphi \mathtt{R}^{\mathcal{B}} \psi) & \text{iff} & \text{there exists a path } \pi = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \\
 & & \text{s.t. } s_0 = s \text{ and for all } i \in \mathbb{N} : \\
 & & \text{if } a_0 \ldots a_i \in \mathcal{B} \text{ then } s_i \models \psi \text{ or there exists } j < i \text{ s.t. } s_j \models \varphi.
\end{array}$$

In order to illustrate the semantics of the $\mathtt{U}^{\mathcal{A}}$ and $\mathtt{R}^{\mathcal{A}}$ operators, consider exemplarily the following models.




Call the leftmost state $s_0$. The formula $\mathtt{E}(\varphi \mathtt{U}^{\mathcal{A}} \psi)$ is satisfied in $s_0$, since there exists a path starting in $s_0$ which is labeled with $w = a_0 a_1 \ldots a_{n-1} a_n$ and $w$ forms a word in $\mathcal{A}$.

Furthermore, this path ends in a state satisfying $\psi$ and along the way, $\varphi$ holds in every state.




Again, call the leftmost state $s_0$. The formula $\mathtt{E}(\varphi \mathtt{R}^{\mathcal{A}} \psi)$ is satisfied in $s_0$. Along the path $a_1 a_2 a_3 a_4 a_5 \ldots$, the prefixes $a_1$, $a_1 a_2 a_3$ and $a_1 a_2 a_3 a_4 a_5$ form words in $\mathcal{A}$. The first two end in a state which satisfies $\psi$. The state in which the latter ends does not satisfy $\psi$, but it is preceded by a state in which $\varphi$ holds. The implication "if $a_0 \ldots a_i \in \mathcal{A}$ then there exists $j < i$ s.t. $s_j \models \varphi$" is now valid for all future states.

## 4.2 Examples

Consider a concurrent producer/consumer scenario, where one process produces objects and places them into a shared buffer. The consumer takes away one such element at a time from the buffer. If the buffer is empty, the consumer process requests a new resource and halts until the producer delivers a new one. Any parallel execution of these processes should obey a non-underflow property (NBU), that is: at any moment the number of produce actions is greater than or equal to the number of consume actions done so far.

Suppose the goal was to formally specify the above scenario including the non-underflow property and on top of that to demand properties like, e.g. "whenever the consumer process sends a request, the buffer is empty".

If the buffer is realised in software it is reasonable to assume that it is unbounded. But then these specifications become non-regular since the NBU property involves unlimited counting of the actions and hence cannot be expressed in, e.g., $\mathcal{L}_\mu$. Let $\Sigma = \{p, c, r\}$, where $p$ stands for *production* of a buffer object, $c$ for *consumation* and $r$ for *requesting* such an object. Formally, the language defining the NBU property is $L_{\mathsf{NBU}} = \{w \in \Sigma^* \mid \ |v|_c \leq |v|_p$ for all $v \preceq w\}$, where $\preceq$ denotes the prefix relation. Emptiness of the buffer is modelled by the language $L_{\mathsf{EMPTY}} = \{w \in \Sigma^* \mid \ |w|_c = |w|_p\}$. Words in $L_{\mathsf{EMPTY}}$ clearly do not respect NBU, so in order to model traces to empty buffers which do respect NBU, we

define $L = L_{\mathsf{EMPTY}} \cap L_{\mathsf{NBU}}$. Note that $L_{\mathsf{NBU}}$ and $L_{\mathsf{EMPTY}}$ are VPL and because VPL are closed under intersection, so is $L$. The desired properties are now expressible as CTL[VPL] formulas:

$$\begin{array}{rcl} \mathtt{AGEX}^p\mathtt{tt} & : & \text{"}\textit{At any time it is possible to produce an object}\text{".} \\ \mathtt{AG}^L(\mathtt{AX}^c\mathtt{ff} \wedge \mathtt{EX}^r\mathtt{tt}) & : & \text{"}\textit{Whenever the buffer is empty, it is impossible to consume} \\ & & \textit{and possible to request}\text{".} \\ \mathtt{AG}^{\overline{L}}(\mathtt{EX}^c\mathtt{tt} \wedge \mathtt{AX}^r\mathtt{ff}) & : & \text{"}\textit{Whenever the buffer is non-empty it is possible to} \\ & & \textit{consume and impossible to request}\text{".} \\ \mathtt{EFEG}^{\overline{c^*}}\mathtt{ff} & : & \text{"}\textit{At some point there is a consume-only path}\text{".} \end{array}$$

The conjunction of the first three properties yields a specification of the producer / consumer scenario described and states that a *request* can only be made if the buffer is empty. Remember that VPL are closed under complement and therefore the third property is indeed a CTL[VPL] property. Every satisfying model gives a raw implementation of the main characteristics of this concurrent process. Note that if it is always possible to *produce* and always possible to *consume* (if the buffer is non-empty), yet impossible to *consume* on an empty buffer, then a straight-forward model with self-loops $p, c$ and $r$ does not satisfy the specification. Instead, a model with infinitely many different $p$ transitions is required. If we strengthen the specification by adding the fourth formula, it becomes unsatisfiable. However, this is not trivial to see and underlines the usefulness of a decidable logic of corresponding expressive power.

## 4.3 Properties

**Theorem 40** CTL[REG] has the finite model property.

Proof This is a consequence of its embedding into $\mathcal{L}_\mu$which has the finite model property (see Thm. 44). □

**Theorem 41** CTL[VPL] does not exhibit the finite model property.

Proof This follows from Thm. 28 in which a PDL$\not?$[VPL] formula serves as witness for the absence of the finite model property. The formula can by Thm. 47 be translated into an equivalent CTL[VPL] formula. Since both formulas are required to hold in exactly the same models, the absence of the finite model property for CTL[VPL] follows. □

**Theorem 42** CTL[$\mathcal{L}$] is bisimulation-invariant and therefore has the tree model property for any $\mathcal{L}$.

Proof We show bisimulation-invariance by induction on the structure of $\varphi$. The base case of $\varphi = q$ for some $q \in \mathcal{P}$ is immediate.
Case $\varphi = \psi_1 \vee \psi_2$. Then we have $s \models \varphi$ iff $s \models \psi_1$ or $s \models \psi_2$ which, by hypothesis, is the case iff $t \models \psi_1$ or $t \models \psi_2$, i.e. $t \models \varphi$. The case of $\varphi = \neg\psi$ is similar.
Case $\varphi = \mathtt{E}(\psi_1 \mathtt{U}^L \psi_2)$. Suppose $s \models \varphi$. Then there is a path $\pi = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \ldots$ s.t. $s_0 = s$ and $\pi \models \psi_1 \mathtt{U}^L \psi_2$. Since $s \xrightarrow{a_1} s_1$ and $s \sim t$ there is a $t_1$ s.t. $t \xrightarrow{a_1} t_1$ and $s_1 \sim t_1$. This can now be iterated, possibly ad infinitum, revealing a path $\pi' = t_0 \xrightarrow{a_1} t_1 \xrightarrow{a_2} t_2 \ldots$ s.t. $t_0 = t$, and $s_i \sim t_i$ for all $i \in \mathbb{N}$.
Now, since $\pi \models \psi_1 \mathtt{U}^L \psi_2$ there is a $k \in \mathbb{N}$ s.t. $s_k \models \psi_2$, $a_1 \ldots a_k \in L$ and $s_j \models \psi_1$ for all $j < k$. By the hypothesis we have $t_k \models \psi_2$, and $t_j \models \psi_1$ for all $j < k$. But then we have $\pi' \models \psi_1 \mathtt{U}^L \psi_2$ and therefore $t \models \varphi$.
The case of $\varphi = \mathtt{E}(\psi_1 \mathtt{R}^L \psi_2)$ is similar. □

The following table presents the computational complexity of the satisfiability problem of CTL[$\mathfrak{A}, \mathfrak{B}$] for the most important classes $\mathfrak{A}$ and $\mathfrak{B}$ [ALL$^+$b].

| | $\mathfrak{B}$ = | DFA | NFA | DVPA | VPA | (D)PDA |
|---|---|---|---|---|---|---|
| CTL[(D)FA, $\mathfrak{B}$] | $\in$ | EXP | 2EXP | 2EXP | 3EXP | undec. |
| | hard | | EXP | | 2EXP | |
| CTL[(D)VPA, $\mathfrak{B}$] | $\in$ | 2EXP | 2EXP | 2EXP | 3EXP | undec. |
| | hard | | | | 2EXP | |
| CTL[(D)PDA, $\mathfrak{B}$] | $\in$ | undec. | undec. | undec. | undec. | undec. |
| | hard | | | | | |

Figure 4.1: Complexity of satisfiability for CTL[$\mathcal{L}$] .

## 4.4 Expressivity

The original CTL ignores path labels and it is therefore easy to give a formula of CTL[REG] already which cannot be expressed in CTL. This is for instance witnessed by the regular language $L = (a(\Sigma \setminus \{a\})^* a)^*$ and the formula $\mathtt{EG}^L p$ stating that there exists a path on which $p$ holds whenever an even number of $a$s is seen. However, CTL translates into

CTL[$\{\Sigma^*, \Sigma\}$] because the universal language $\Sigma^*$ anulls the additional path constraints in the parametric CTL semantics. The language $\Sigma$ is needed in addition for the translation of $\texttt{EX}$. On the other hand, CTL[REG] does not yet exceed regular expressivity.

**Theorem 43**

$$\text{CTL} \;\equiv\; \text{CTL}[\{\Sigma^*, \Sigma\}] \;\lneq\; \text{CTL}[\text{REG}].$$

Proof For the proof of CTL $\equiv$ CTL[$\{\Sigma^*, \Sigma\}$], define inductively a translation function $\overrightarrow{\mathsf{tr}} : \text{CTL} \to \text{CTL}[\{\Sigma^*, \Sigma\}]$ as follows:

$$\begin{aligned}
\overrightarrow{\mathsf{tr}}(p) &= p,\\
\overrightarrow{\mathsf{tr}}(\neg\psi) &= \neg\overrightarrow{\mathsf{tr}}(\psi),\\
\overrightarrow{\mathsf{tr}}(\psi_1 \vee \psi_2) &= \overrightarrow{\mathsf{tr}}(\psi_1) \vee \overrightarrow{\mathsf{tr}}(\psi_2),\\
\overrightarrow{\mathsf{tr}}(\texttt{EX}\psi') &= \texttt{EF}^{\Sigma}\ \overrightarrow{\mathsf{tr}}(\psi'),\\
\overrightarrow{\mathsf{tr}}(\texttt{E}(\psi_1 Q \psi_2)) &= \texttt{E}(\overrightarrow{\mathsf{tr}}(\psi_1) Q^{\Sigma^*} \overrightarrow{\mathsf{tr}}(\psi_2)).
\end{aligned}$$

for $Q \in \{\texttt{U}, \texttt{R}\}$. The translation function for the converse direction $\overleftarrow{\mathsf{tr}}$ is $\overrightarrow{\mathsf{tr}}^{-1}$ but has the additional mappings

$$\begin{aligned}
\overleftarrow{\mathsf{tr}}(\texttt{E}(\psi_1 \texttt{U}^{\Sigma} \psi_2)) &= \overleftarrow{\mathsf{tr}}(\psi_1) \wedge \texttt{EX}\,\overleftarrow{\mathsf{tr}}(\psi_2),\\
\overleftarrow{\mathsf{tr}}(\texttt{E}(\psi_1 \texttt{R}^{\Sigma} \psi_2)) &= \overleftarrow{\mathsf{tr}}(\psi_1) \vee \texttt{EX}\,\overleftarrow{\mathsf{tr}}(\psi_2).
\end{aligned}$$

The proof that these translations are semantically faithful is trivial.
Considering the remaining claim, it is clear that $\Sigma^*$ and $\Sigma$ are regular languages and therefore CTL[$\{\Sigma^*, \Sigma\}$] $\leq$ CTL[REG]. A witness for CTL $\lneq$ CTL[REG] has already been given above by the formula $\texttt{EG}^L p$, where $L = (a(\Sigma \setminus \{a\})^* a)^*$.

**Theorem 44**

$$\text{CTL}[\text{REG}] \;\lneq\; \mathcal{L}_\mu.$$

Proof Consider the following inductively defined translation $\mathsf{tr} : \text{CTL}[\text{REG}] \to \mathcal{L}_\mu$. We assume the regular language adornments of the $\texttt{U}$- and $\texttt{R}$-operators are given as a DFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ with $Q = \{q_0, \ldots, q_n\}$.

$$\begin{aligned}
\mathsf{tr}(p) &= p,\\
\mathsf{tr}(\neg\psi) &= \neg\mathsf{tr}(\psi),
\end{aligned}$$

$$\mathsf{tr}(\psi_1 \vee \psi_2) \quad = \quad \mathsf{tr}(\psi_1) \vee \mathsf{tr}(\psi_2),$$

$$\mathsf{tr}(\mathtt{E}(\psi_1 \mathtt{U}^{\mathcal{A}} \psi_2)) \quad = \quad \mu x_0 . \begin{pmatrix} x_0 & . & \ldots \\ & \vdots & \\ x_i & . & \left\{ \begin{array}{ll} \mathsf{tr}(\psi_2) & \text{,if } q_i \in F \\ \mathtt{ff} & \text{otherwise} \end{array} \right\} \vee \Big( \mathsf{tr}(\psi_1) \wedge \bigvee_{a \in \Sigma} \langle a \rangle ( \bigvee_{q_j = \delta(q_i, a)} x_j ) \Big) \\ & \vdots & \\ x_n & . & \ldots \end{pmatrix},$$

$$\mathsf{tr}(\mathtt{E}(\psi_1 \mathtt{R}^{\mathcal{A}} \psi_2)) \quad = \quad \nu x_0 . \begin{pmatrix} x_0 & . & \ldots \\ & \vdots & \\ x_i & . & \left\{ \begin{array}{ll} \mathsf{tr}(\psi_2) & \text{,if } q_i \in F \\ \mathtt{tt} & \text{otherwise} \end{array} \right\} \wedge \Big( \mathsf{tr}(\psi_1) \vee \bigvee_{a \in \Sigma} \langle a \rangle ( \bigvee_{q_j = \delta(q_i, a)} x_j ) \Big) \\ & \vdots & \\ x_n & . & \ldots \end{pmatrix}.$$

The latter translations use simultaneous fixpoint notation which is explained in Def. 47. Note that the structure of each of the inner fixpoint formulas is in principle the same as in the translation of CTL (see Ex. 10). The difference is that only such successors are considered which correspond to transitions in $\mathcal{A}$ and that the checks of the subformulas $\mathsf{tr}(\psi_2)$ respect final states of $\mathcal{A}$.

Strictness follows from the fact that the alternation hierarchy in $\mathcal{L}_\mu$ is strict and that the formulas resulting from $\mathsf{tr}$ have alternation depth 0. Hence, any formula expressible in CTL[REG] has alternation depth 0, but there exist $\mathcal{L}_\mu$-formulas with alternation depth greater than 0 which cannot be expressed by formulas with lesser alternation depth.

Note that if the language adornment for the $\mathtt{R}$-operator is given as an NFA, the translation is of exponential size, because the NFA has to be translated into a DFA first. The construction is not correct for NFAs in general. See the introductory paragraph of Sec. 4.5 for details. □

For certain language classes $\mathcal{L}$ which are richer than REG, the following theorem confirms that the CTL-related logical frameworks using such languages from $\mathcal{L}$ as adornments are indeed capable of expressing non-regular program properties.

**Theorem 45** Let $\mathcal{L}$ be a class of formal languages s.t. $L = \{a^n b^n \mid n \in \mathbb{N}\} \in \mathcal{L}$. Then for all language classes $\mathfrak{B}$

$$\text{CTL}[\mathcal{L}, \mathfrak{B}] \;\not\leq\; \mathcal{L}_\mu.$$

Proof The formula used to show non-regularity of PDL[$\mathcal{L}$] for any class of languages containing at least $\{a^n b^n \mid n \in \mathbb{N}\}$ in Lemma 1 is – as can easily be seen – from the fragment PDL$\not{?}$[$\mathcal{L}$] without tests. Hence, by Thm. 47, it can be translated into EF[$\mathcal{L}$]. So already the fragment EF[$\mathcal{L}$] contains a formula which has no equivalent in $\mathcal{L}_\mu$. □

**Corollary 8** For all language classes $\mathfrak{B}$,

$$\text{CTL}[\text{SML}, \mathfrak{B}] \;\not\leq\; \mathcal{L}_\mu.$$

Proof This follows from Thm. 45 and the fact that $\{a^n b^n \mid n \in \mathbb{N}\}$ is an SML [HPS83]. □

But just as it is the case with PDL[$\mathcal{L}$], parametric CTL is no extension of $\mathcal{L}_\mu$. This follows from a theorem in [ALL$^+$b] in which it is proved that CTL[$\mathcal{L}$] is an entirely different extension of CTL than CTL$^*$ is. Remember that CTL$^*$ is a strict fragment of $\mathcal{L}_\mu$.

**Theorem 46 ([ALL$^+$b])** For all language classes $\mathfrak{A}, \mathfrak{B}$, we have

$$\text{CTL}[\text{SML}, \mathfrak{B}] \;\not\leq\; \text{CTL}^*.$$
$$\text{CTL}^* \;\not\leq\; \text{CTL}[\mathfrak{A}, \mathfrak{B}].$$

This result is a consequence of the fact that the fragment EF[$\{a^n b^n \mid n \in \mathbb{N}\}$] contains non-regular properties inexpressible in CTL$^*$. On the other hand, fairness is expressible in CTL$^*$ but not in CTL[$\mathfrak{A}, \mathfrak{B}$].
The $\langle L \rangle$-operator of parametric PDL is clearly equivalent to the $\mathtt{EF}^L$-operator in parametric CTL. This observation leads to the following theorem.

**Theorem 47** For all language classes $\mathcal{L}$,

$$\text{EF}[\mathcal{L}] \equiv \text{PDL}\not{?}[\mathcal{L}].$$

Proof Note that both logics do only differ in the $\langle L \rangle$ and $\mathtt{EF}^L$ constructs. Their semantical equivalence is trivial to prove: $s \models \mathtt{EF}^L \varphi$ iff there exists a path $\pi = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_i} s_i$, s.t. $s_0 = s$ and $a_1 \ldots a_i \in L$ and $s_i \models \varphi$ iff $s \models \langle L \rangle \varphi$. □

On the other hand, there seems to be no equivalent in PDL$\not?$[$\mathcal{L}$] for the expression scheme $\mathtt{E}(\psi_1\mathtt{R}^L\psi_2)$ or even $\mathtt{EG}^L\varphi$. We prove this up to CFL with help of the following lemma stating that there exists an EG-formula inexpressible in EF[CFL].

**Lemma 23**

$$\text{EG}[\{\Sigma,\Sigma^*\}] \not\leq \text{EF}[\text{CFL}].$$

PROOF Let $w/L = \{v \in \Sigma^* \mid wv \in L\}$ for any $w \in \Sigma^*$ and any formal language $L$. Define the Fischer-Ladner-closure $\mathsf{Cl}(\varphi)$ for any formula $\varphi \in \text{EF}[\mathcal{L}]$ as the least set satisfying the following:

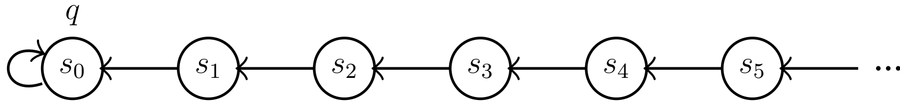
- $\varphi \in \mathsf{Cl}(\varphi)$.
- if $\neg\psi \in \mathsf{Cl}(\varphi)$ then $\psi \in \mathsf{Cl}(\varphi)$.
- if $\psi_1 \vee \psi_2 \in \mathsf{Cl}(\varphi)$ then $\psi_1, \psi_2 \in \mathsf{Cl}(\varphi)$.
- if $\mathtt{EF}^L\psi \in \mathsf{Cl}(\varphi)$ then $\mathtt{EF}^{\{a\}}\mathtt{EF}^{a/L}\psi \in \mathsf{Cl}(\varphi)$ and $\psi \in \mathsf{Cl}(\varphi)$ for all $a \in \Sigma$.

Furthermore, define the quotient of a transition system $\mathcal{T} = (\mathcal{S}, \rightarrow, \ell)$ under a set of formulas $\Phi \subseteq \mathtt{EF}^{\mathcal{L}}$ as $\mathcal{T}/\Phi = (\mathcal{S}/\Phi, \rightarrow, \ell/\Phi)$ with

- $\mathcal{S}/\Phi = \{[s] \mid s \in \mathcal{S}\}$ where $[s] = \{t \in \mathcal{S} \mid s \sim_\Phi t\}$ and $s \sim_\Phi t$ iff $\forall \varphi \in \Phi : s \models_\mathcal{T} \varphi$ iff $t \models_\mathcal{T} \varphi\}$,
- $[s] \xrightarrow{a} [t]$ iff $\exists s', t'$ with $s' \sim_\Phi s$, $t' \sim_\Phi t$, and $s' \xrightarrow{a} t'$,
- $\ell/\Phi([s]) = \ell(s) \cap \Phi$.

We do now show that for all $\mathcal{T}$ with states $s \in \mathcal{S}$ and all $\varphi \in \text{EF}[\mathcal{L}]$ for any $\mathcal{L}$, we have $s \models_\mathcal{T} \varphi$ iff $[s] \models_{\mathcal{T}/\mathsf{Cl}(\varphi)} \varphi$ by induction on the structure of $\varphi$. The propositional cases are entirely trivial. Now assume $s \models_\mathcal{T} \mathtt{EF}^L\psi$ and let $s_0 \xrightarrow{a_1\ldots a_n} s_n$ be a path witnessing this, hence $s = s_0$, $a_1 \ldots a_n \in L$ and $s_n \models_\mathcal{T} \psi$. By definition of $\rightarrow$ the path $[s_0] \xrightarrow{a_1\ldots a_n} [s_n]$ indeed exists and by induction hypothesis we have $[s_n] \models_{\mathcal{T}/\mathsf{Cl}(\varphi)} \psi$.
For the other direction assume $[s] \models_{\mathcal{T}/\mathsf{Cl}(\varphi)} \mathtt{EF}^L\psi$ and let $[s_0] \xrightarrow{a_1\ldots a_n} [s_n]$ be a path witnessing this, hence $s = s_0$, $a_1 \ldots a_n \in L$ and $[s_n] \models_{\mathcal{T}/\mathsf{Cl}(\varphi)} \psi$.
Note that for all $0 \leq i \leq n$ we have $[s_i] \models_\mathcal{T} \mathtt{EF}^{a_1\ldots a_i/L}\psi$. From this follows $t \models_\mathcal{T} \mathtt{EF}^{a_1\ldots a_i/L}\psi$ for all $t \in [s_i]$. But then the path $s_0 \xrightarrow{a_1\ldots a_n} s_n$ exists and by induction hypothesis we have $s_n \models_\mathcal{T} \psi$ which concludes the proof.

We first prove that $\mathrm{EG}[\{\Sigma, \Sigma^*\}] \not\leq \mathrm{EF}[\mathrm{REG}]$. For this we need to prove that $\mathcal{T}/\mathsf{Cl}(\varphi)$ has only finitely many states for any $\varphi \in \mathrm{EF}[\mathrm{REG}]$.
But this follows from Thm. 1 (Myhill-Nerode) and the construction of $\mathsf{Cl}(\varphi)$, because there are only finitely many elements in $\mathsf{Cl}(\varphi)$ which can be distinguished w.r.t. $\models_{\mathcal{T}}$ by any two states.
On the other hand, consider the transition system $\mathcal{T} = (\{s_i \mid i \geq 0\}, \rightarrow, \ell)$ with $s_i \rightarrow s_j$ iff $j = i - 1$ and $s_0 \rightarrow s_0$ (depicted below), and $\ell(q) = \{s_0\}$.




Clearly, we have $s_i \models_{\mathcal{T}} \mathtt{AF}q$ for all $i \in \mathbb{N}$. However, suppose there was an EF[REG] formula $\xi$ equivalent to $\mathtt{AF}q$. By the above, we have $s_i \models_{\mathcal{T}} \mathtt{AF}q$ iff $[s_i] \models_{\mathcal{T}/\mathsf{Cl}(\xi)} \xi$. Since $\mathcal{T}/\mathsf{Cl}(\xi)$ is finite, there must exist a $[s_j]$ with $j > 0$ and $[s_j] \rightarrow [s_j]$. But then $[s_k] \not\models \mathtt{AF}q$ for every $k \geq j$.
That $\mathrm{EG}[\{\Sigma, \Sigma^*\}] \not\leq \mathrm{EF}[\mathrm{CFL}]$ now simply follows from the fact that the model used in the proof above uses no transition labels and that CFL over single-letter alphabets are REG and hence the same proof applies.

**Theorem 48** For all language classes $\mathcal{L} \in \{$REG, SML, SSML, VPL, CFL$\}$,

$$\mathrm{PDL}\,\not{?}[\mathcal{L}] \lneq \mathrm{CTL}[\mathcal{L}, \{\Sigma, \Sigma^*\}].$$

Proof $\leq$ follows from Thm. 47 and the fact that $\mathrm{EF}[\mathcal{L}]$ is syntactically included in $\mathrm{CTL}[\mathcal{L}, \{\Sigma, \Sigma^*\}]$. Strictness follows from Lemma 23 which states that there exists a formula $\varphi$ in $\mathrm{EG}[\{\Sigma, \Sigma^*\}]$ inexpressible in EF[CFL]. □

Regarding a comparison between the expressivity of different $\mathrm{CTL}[\mathcal{L}]$ fragments, the following correspondence holds.

**Theorem 49** For all $\mathcal{L}, \mathcal{L}', \mathcal{N}, \mathcal{N}'$, if $\mathcal{L} \subseteq \mathcal{N}$ and $\mathcal{L}' \subseteq \mathcal{N}'$ then

$$\mathrm{CTL}[\mathcal{L}, \mathcal{L}'] \leq \mathrm{CTL}[\mathcal{N}, \mathcal{N}'].$$

Proof Trivial. More languages at hand cannot decrease the expressive power. $\mathrm{CTL}[\mathcal{L}, \mathcal{L}']$ is a syntactical fragment of $\mathrm{CTL}[\mathcal{N}, \mathcal{N}']$. □

But for certain language classes, we can strengthen the above result to strictness.

**Theorem 50**

$$\mathrm{CTL[REG]} \lneq \mathrm{CTL[VPL]} \lneq \mathrm{CTL[DCFL]}.$$

Proof The containment of CTL[REG] in CTL[VPL] is a consequence of Thm. 49. Strict separation follows from Thm. 44 stating that CTL[REG] is strictly contained in $\mathcal{L}_\mu$ and Cor. 8 stating that CTL[SML] is strictly more expressive than $\mathcal{L}_\mu$. Again, by Thm. 49 CTL[SML] is contained in CTL[VPL] which finishes the proof.
Containment of CTL[VPL] in CTL[DCFL] is again a consequence of Thm. 49 while strictness has been proved in [ALL$^+$b]. The proof uses a theorem which states that every satisfiable CTL[VPL] formula has a model which is a visibly pushdown system. Then a CTL[DCFL] formula is constructed whose models are bisimilar to an LTS which can not even be represented by a pushdown system. □

Fig. 4.2 summarises the expressivity results on CTL[$\mathcal{L}$] . A line from a lower positioned item to a higher positioned item denotes inclusion of the former in the latter. If it is dashed this means that the inclusion is strict.

## 4.5 Model Checking

In this section we intend to determine the computational complexity of model checking CTL[$\mathfrak{A}, \mathfrak{B}$] w.r.t. the automata classes $\mathfrak{A}$ and $\mathfrak{B}$. We focus on robust classes such as NFA, VPA, PDA, etc.
First of all, we observe that the $\mathtt{EU}^\mathcal{A}$ and $\mathtt{ER}^\mathcal{A}$ are two fundamentally different operators. Take some formula of the form $\mathtt{E}(p_1\mathtt{U}^\mathcal{A}p_2)$, where $\mathcal{A}$ is some automaton and $p_1, p_2$ are propositions. Note that the existential path quantification and the existential quantification over runs of $\mathcal{A}$ in the acceptance condition for a nondeterministic automaton $\mathcal{A}$ commute. This allows product constructions of $\mathcal{A}$ and the underlying LTS plus some overhead stemming from the checks of $p_1$ and $p_2$ along the paths. If there is a witness for non-emptiness of the product automaton then it serves simultaneously as a path in the LTS and a word accepted by $\mathcal{A}$, which is the pattern constituting the semantics of $\mathtt{E}(p_1\mathtt{U}^\mathcal{A}p_2)$.
In fact, the task is very similar to model checking a formula $\langle\mathcal{A}\rangle p_2 \in \mathrm{PDL}[\mathcal{L}]$, where a close relationship between the REG-intersection problem for a language class $\mathcal{L}$ and model checking PDL[$\mathcal{L}$] was established. The formula $\mathtt{E}(p_1\mathtt{U}^\mathcal{A}p_2)$ only differs from $\langle\mathcal{A}\rangle p_2$ in the

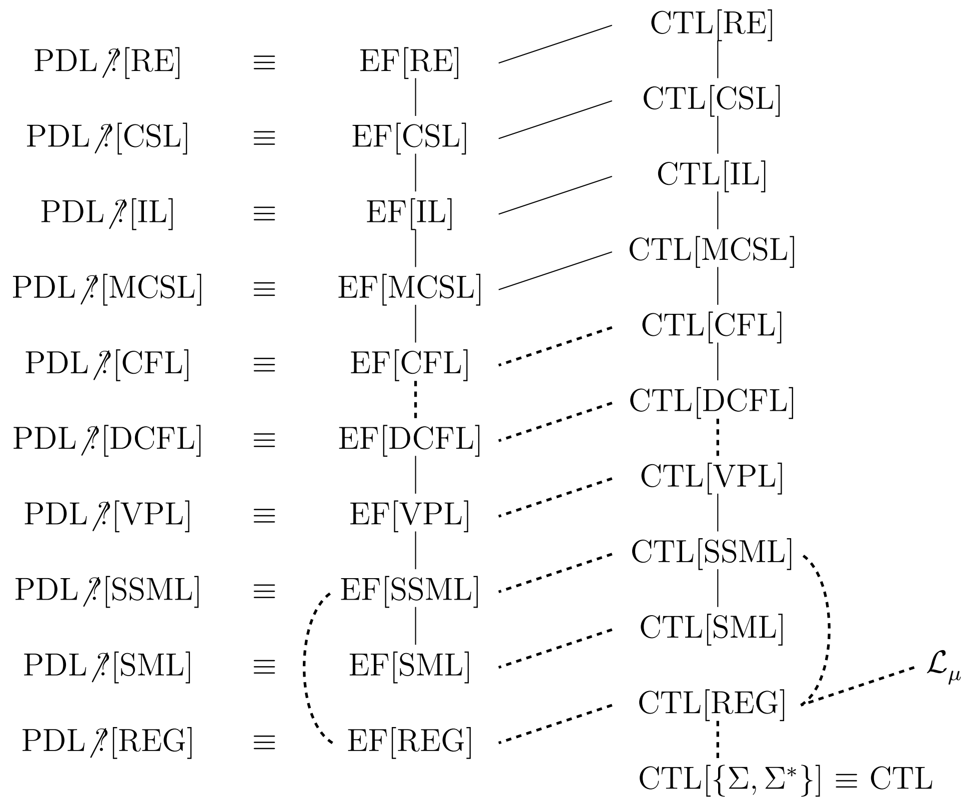



Figure 4.2: Expressive power of $\mathrm{CTL}[\mathcal{L}]$ .

requirement of recurrent propositions $p_1$ along the witnessing path. As has been shown before, the actual equivalent of $\langle \mathcal{A} \rangle p_2$ in CTL$[\mathfrak{A}, \mathfrak{B}]$ is $\mathtt{EF}^{\mathcal{A}} p_2$.

The situation changes when we take a formula of the form $\mathtt{E}(p_1 \mathtt{R}^{\mathcal{A}} p_2)$ for some nondeterministic automaton $\mathcal{A}$. Note that, here, the path is again existentially quantified but the runs of the automaton on any prefix are implicitly universally quantified by the $\mathtt{R}^{\mathcal{A}}$-operator ("on all prefixes it either holds that $\mathcal{A}$ does not accept the prefix or ..."). The quantification does no longer commute and this prevents using product constructions in the same way as for $\mathtt{EU}^{\mathcal{A}}$ formulas, because it requires to keep a protocol of all nondeterministic choices of $\mathcal{A}$ w.r.t. the currently considered path in the LTS, since every such choice might end up in an accepting state. If on the other hand the automaton is deterministic, the problem does not arise because no matter which LTS path is chosen as a witness for satisfaction, the automaton has just one state at every moment while reading the labels along the path. This regains the property of local determinateness and simplifies model checking significantly.

Due to this difference we need to investigate CTL$[\mathfrak{A}, \mathfrak{B}]$ not only w.r.t. the language class parameter for the two different temporal subformula types but also w.r.t. the representing automaton model, i.e. deterministic or nondeterministic.

The following algorithm serves as a general scheme which deals with the common base of all CTL$[\mathfrak{A}, \mathfrak{B}]$ fragments under consideration here. In particular, the treatment of temporal formulas is externalised into subroutines and simplified by preparational steps on the structure of the formula and model.

MC-CTL($\mathcal{T}$, $\varphi$) =
  `let` $(\mathcal{S}, \rightarrow, \ell) = \mathcal{T}$ `in`
  `case` $\varphi$ of
    $q$ : $\ell(q)$
    $\neg\psi$ : $\mathcal{S} \setminus$ MC-CTL($\mathcal{T}$, $\psi$)
    $\psi_1 \vee \psi_2$ : MC-CTL($\mathcal{T}$, $\psi_1$) $\cup$ MC-CTL($\mathcal{T}$, $\psi_2$)
    $\mathtt{E}(\psi_1 Q^{\mathcal{A}} \psi_2)$ : `let` $p_1, p_2$ : *fresh propositions* `in`
      $\ell' := \ell[p_1 \mapsto \text{MC-CTL}(\mathcal{T}, \psi_1)]$;
      $\ell' := \ell'[p_2 \mapsto \text{MC-CTL}(\mathcal{T}, \psi_2)]$;
      `let` $\mathcal{T}' = (\mathcal{S}, \rightarrow, \ell')$ `in`
        `if` $Q = \mathtt{U}$ `then` MC-U($\mathcal{T}'$, $p_1 \mathtt{U}^{\mathcal{A}} p_2$)
        `else` MC-R($\mathcal{T}'$, $p_1 \mathtt{R}^{\mathcal{A}} p_2$)

Algorithm MC-CTL takes an LTS $\mathcal{T}$ and a formula $\varphi \in$ CTL$[\mathfrak{A}, \mathfrak{B}]$ and returns the set of states in which $\varphi$ holds. In its current form, MC-CTL uses oracles MC-U and MC-R

(taking arguments of the same type as MC-CTL) to compute the result set for EU[$\mathfrak{A}$] and ER[$\mathfrak{B}$] formulas. Before a call of the subroutines MC-U or MC-R takes place, the original formula $\varphi$ of the form $\mathtt{E}(\psi_1 Q^{\mathcal{A}} \psi_2)$, where $Q \in \{\mathtt{U}, \mathtt{R}\}$, and the LTS $\mathcal{T}$ are transformed: the subformulas $\psi_1$ and $\psi_2$ are evaluated recursively in a first step and then replaced in $\varphi$ by fresh atomic propositions $p_1$ and $p_2$. The labeling function is updated accordingly, s.t. $\ell'(p_1)$ contains exactly the states which satisfy $\psi_1$ and $\ell'(p_2)$ those which satisfy $\psi_2$.

The proof of soundness and completeness is trivial under the assumption of soundness and completeness of the subroutines MC-U and MC-R. It consists of a straight-forward structural induction on the input formula $\varphi$.

We remark that algorithm MC-CTL has two main benefits for our purposes. First of all, the subroutines MC-U and MC-R are called on flattened versions of the original formula which now contain propositions as nested subformulas only, i.e. only have to deal with restricted fragments of EU[$\mathfrak{A}$] and ER[$\mathfrak{B}$] respectively. This will of course simplify any further analysis of these subroutines. We denote these restricted fragments by $\text{EU}_{\mathcal{P}}[\mathfrak{A}]$ and $\text{ER}_{\mathcal{P}}[\mathfrak{A}]$.

Furthermore, upper bounds on the computational complexity of MC-CTL can be derived from upper bounds on MC-U or MC-R, respectively, depending on which of the corresponding model checking problems for $\text{EU}_{\mathcal{P}}[\mathfrak{A}]$ and $\text{ER}_{\mathcal{P}}[\mathfrak{B}]$ formulas is harder to solve. Note that MC-CTL runs in time $\mathcal{O}(|\varphi|)$ when regarding MC-U and MC-R as oracles.

We now turn our attention to concrete instances of the automata classes $\mathfrak{A}$ and $\mathfrak{B}$ as restricting parameters for $\text{EU}_{\mathcal{P}}[\mathfrak{A}]$ and $\text{ER}_{\mathcal{P}}[\mathfrak{B}]$. As mentioned before, model checking $\mathtt{E}(p_1 \mathtt{U}^{\mathcal{A}} p_2)$ is closely related to model checking the PDL formula $\langle \mathcal{A} \rangle p_2$. Both formulas hold if there is a path in the model which is labeled with a $w \in \mathcal{L}(\mathcal{A})$ and ends in a state labeled with $p_2$. The difference is simply that the $\mathtt{E}(p_1 \mathtt{U}^{\mathcal{A}} p_2)$ operator additionally requires $p_1$ to hold in every state along the path except the last.

Following this observation, it is tempting to try to establish reductions between altered versions of the REG-intersection-, $\mathcal{L}$-reachability- and the model checking problem for $\text{EU}_{\mathcal{P}}[\mathfrak{A}]$ in a similar fashion as for model checking PDL[$\mathcal{L}$]. In fact, a generalisation of the $\mathcal{L}$-reachability problem which takes into account the propositions of the LTS in the way required can easily be found and shown to be equivalent to model checking $\text{EU}_{\mathcal{P}}[\mathfrak{A}]$. However, this little difference destroys the equivalence to the REG-intersection problem and we are not aware of any natural counterpart to cover the discrepancy. Any repair of this defect seems technically cumbersome while upper bounds can be found much easier by directly applying product construction techniques for a reduction on non-emptiness of

certain automata classes as follows.

**Lemma 24** Model checking $\text{EU}_{\mathcal{P}}$[PDA] is in PTIME.

PROOF By reduction to the non-emptiness problem of PDA. Let $\varphi \in \text{EU}_{\mathcal{P}}[\text{PDA}]$, $\mathcal{T} = (\mathcal{S}, \rightarrow, \ell)$ be an LTS and $s \in \mathcal{S}$. Clearly, $\varphi$ is of the form $\mathtt{E}(p_1 \mathtt{U}^{\mathcal{A}} p_2)$, where $p_1, p_2 \in \mathcal{P}$ and $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, q_0, F)$ is a PDA. To solve the question whether $s \models \varphi$, we construct a PDA $\mathcal{A}_{\mathcal{T}} = (Q \times \mathcal{S}, \Sigma, \Gamma, \delta', (q_0, s), F')$, where

- $F' = \{(q, s) \mid q \in F \text{ and } s \in \ell(p_2)\}$,
- $\delta'((q, s), a, \gamma) = \{(q', s') \mid q' \in \delta(q, a, \gamma) \text{ and } s \xrightarrow{a} s' \text{ and } s \in \ell(p_1)\}$.

Note that $|\mathcal{A}_{\mathcal{T}}| = \mathcal{O}(|\mathcal{A}| \cdot |\mathcal{T}|)$.
Now, assume $\mathcal{A}_{\mathcal{T}}$ does not reject every word and let $w \in \Sigma^*$ be a witness for this. Note that any accepting run of $\mathcal{A}_{\mathcal{T}}$ on $w$ simulates an accepting run of $\mathcal{A}$ on $w$ and synchronously follows a $w$-labeled path in $\mathcal{T}$ along which $p_1$ holds in every state except the last. Furthermore, from the requirement on accepting states we have that the last state is in labeled with $p_2$. Hence, $w \in \mathcal{L}(\mathcal{A})$ and there exists a $t \in \mathcal{S}$ s.t. $s \xrightarrow{w} t$ with the required $p_1$ and $p_2$ labels on states. It is well known that the non-emptiness problem for PDA is in PTIME (cf. [HU79]). From this and the fact that the size of $\mathcal{A}_{\mathcal{T}}$ is polynomial in $|\mathcal{A}|$ and $|\mathcal{T}|$ follows the claim. □

**Theorem 51** Model checking $\text{EU}_{\mathcal{P}}$[LIL] is in PTIME.

PROOF We prove this by a linear-time Turing-reduction on the model checking problem for PDL[LIL] which itself is in PTIME by Cor. 5. Let $\varphi \in \text{EU}_{\mathcal{P}}[\text{LIL}]$ and $\mathcal{T} = (\mathcal{S}, \rightarrow, \ell)$ be an LTS. Clearly, $\varphi$ is of the form $\mathtt{E}(p_1 \mathtt{U}^{\mathcal{A}} p_2)$, where $p_1, p_2 \in \mathcal{P}$ and $\mathcal{A}$ is a LIL representation. We compute an LTS $\mathcal{T}'$ obtained from $\mathcal{T}$ by the following steps:

- Remove all states in $\mathcal{T}$ in which $p_1$ does not hold and then remove transitions leading nowhere.
- Create a new state in which proposition $p_2$ holds and add $a$-transitions to this state from all states which sustained step 1 and have an $a$-transition leading to a state in which $p_2$ holds in the original LTS $\mathcal{T}$.

Now, for any state $s \in \mathcal{S}$, we have that $s \models_{\mathcal{T}} \mathtt{E}(p_1 \mathtt{U}^{\mathcal{A}} p_2)$ iff $s \models_{\mathcal{T}'} \langle\mathcal{A}\rangle p_2$. Note that in every state along every path in $\mathcal{T}'$ proposition $p_1$ holds except for possibly the last one in which $p_2$ holds. All labels between such states are conserved from $\mathcal{T}$ and hence all path labels $w \in \mathcal{L}(\mathcal{A})$ between the remaining states are intact. Note also that the potential doubling of a transition which leads to the new state does no harm at all.
The above sketched algorithm runs in time $\mathcal{O}(|\to|)$ and the resulting LTS has only one additional state and $|\Sigma|$ additional transitions in the worst case.
We therefore obtain a PTIME algorithm from this. □

**Theorem 52** Model checking $\text{EU}_{\mathcal{P}}$[IL] is in EXPTIME.

Proof This is proved by a linear-time Turing-reduction on the model checking problem for PDL[IL] in exactly the same way as for Thm. 51. Note that model checking PDL[IL] is in EXPTIME by Thm. 4. □

**Theorem 53** Let $\mathfrak{A} \in \{$DFA, NFA, DVPA, VPA, DPDA, PDA$\}$. Model checking EU[$\mathfrak{A}$] is PTIME-complete.

Proof Let $\mathcal{T} = (\mathcal{S}, \to, \ell)$ be an LTS. In order to show containment within PTIME it suffices to show the statement for the class PDA since DFA, NFA, DVPA, VPA and DPDA are subclasses of PDA. For the logic EU[PDA], algorithm MC-CTL runs in time $\mathcal{O}(|\varphi|)$ for any formula $\varphi$ and does only call the oracle MC-U and never MC-R. The reduction used in the proof of Lemma 24 allows to implement MC-U by calling the emptiness check of the product automaton (which itself is a PTIME procedure) once for each $s \in \mathcal{S}$. Altogether we have $\mathcal{O}(|\varphi| \cdot |\mathcal{S}|)$ calls of a PTIME procedure and therefore established the claim.
Hardness follows from the fact that the logic EF[$\mathfrak{A}$] is a sublogic of EU[$\mathfrak{A}$] for all $\mathfrak{A}$. From Lemma 47 we have that EF[$\mathfrak{A}$] is equi-expressive to PDL[$\mathfrak{A}$].
Therefore hardness results transfer from PDL[$\mathfrak{A}$] which in the cases of NFA, VPA, DPDA and PDA yield PTIME. □

**Theorem 54** Model checking EU[LIL] is PTIME-complete.

Proof A PTIME implementation for MC-U has been given in Thm. 51. The complete algorithm MC-CTL calls MC-U only $\mathcal{O}(|\varphi|)$ times for any formula $\varphi \in$ EU[LIL] (and never MC-R). □

**Theorem 55** Model checking EU[IL] is EXPTIME-complete.

PROOF An EXPTIME implementation for MC-U has been given in Thm. 52. The complete algorithm MC-CTL calls MC-U only $\mathcal{O}(|\varphi|)$ times for any formula $\varphi \in$ EU[IL] (and never MC-R).

**Theorem 56** Model checking EF[CSL] is undecidable.

PROOF From Lemma 47 we have that PDL$\not?$[CSL] $\equiv$ EF[CSL] and that the translation is computable. Cor. 3 states that model checking PDL$\not?$[CSL] is undecidable. □

While the similarities between PDL[$\mathfrak{A}$] and EU[$\mathfrak{A}$] are by now also reflected in the computational complexities of model checking, the situation is very different with the ER[$\mathfrak{A}$] fragment. As has been stated earlier, it is of great importance whether the automata under consideration are deterministic or nondeterministic. We start with deterministic automata.

**Lemma 25** Model checking ER$_{\mathcal{P}}$[DPDA] is in PTIME.

PROOF By a reduction to the problem of model checking a fixed LTL formula on a PDS. Let $\varphi \in$ ER$_{\mathcal{P}}$[DPDA], $\mathcal{T} = (\mathcal{S}, \rightarrow, \ell)$ be an LTS and $s \in \mathcal{S}$. Clearly, $\varphi$ is of the form $\mathtt{E}(p_1 \mathtt{R}^{\mathcal{A}} p_2)$, where $p_1, p_2 \in \mathcal{P}$ and $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, q_0, F)$ is a DPDA. We construct a PDS $\mathcal{T}_{\mathcal{A}} = (Q \times \mathcal{S} \cup \{g, b\}, \Gamma, \Delta, \ell')$, where $\ell' : 2^{\mathcal{P}} \cup \{p_b\} \rightarrow Q \times \mathcal{S} \cup \{g, b\}$ (for a fresh proposition $p_b$) is defined as $\ell'(q) = Q \times \ell(q)$, if $q \in \mathcal{P}$ and $\ell'(p_b) = \{b\}$ otherwise.
Intuitively, $g$ represents "good" and $b$ "bad" states, i.e. dead-end states, in which the property which $\varphi$ expresses has been fulfilled or violated, respectively.
Furthermore, $\Delta$ contains the following transition rules:

$$((q,s),\gamma) \hookrightarrow \begin{cases} (g,\epsilon) & \text{,if } (q,s) \in \ell'(p_1) \text{ and} \\ & (q \in F \text{ implies } (q,s) \in \ell'(p_2)). \\ & \\ (b,\epsilon) & \text{,if } q \in F \text{ and } (q,s) \notin \ell'(p_2). \\ & \\ ((q',s'),w) & \text{,if none of the above match} \\ & \text{and there exists } a \in \Sigma, \text{ s.t.} \\ & s \xrightarrow{a} s' \text{ and } (q',w) \in \delta(q,a,\gamma) \\ & \text{for some } \gamma \in \Gamma, w \in \Gamma^*. \end{cases}$$

Note that $|\mathcal{T}_{\mathcal{A}}| = \mathcal{O}(|\mathcal{T}| \cdot |\mathcal{A}|)$.
Now consider the LTL formula $\mathtt{F}p_b$. We show that $s \not\models_{\mathcal{T}} \mathtt{E}(p_1 \mathtt{R}^{\mathcal{A}} p_2)$ iff $((q_0,s),\epsilon) \models_{\mathcal{T}_{\mathcal{A}}} \mathtt{F}p_b$.

The "only-if" direction: Assume $s \not\models_{\mathcal{T}} \mathtt{E}(p_1\mathtt{R}^{\mathcal{A}}p_2)$. This means that on all paths starting in $s$, $(\neg p_1\mathtt{U}^{\mathcal{A}}\neg p_2)$ holds and hence on all paths a $w = a_1 \ldots a_n \in L(\mathcal{A})$ and $s_0, \ldots, s_n$ exist, s.t. $s_0 \xrightarrow{a_1} s_1 \ldots \xrightarrow{a_n} s_n$, $s = s_0$ and for all $i \leq n$ we have $s_i \models_{\mathcal{T}} \neg p_1$ and $s_n \models_{\mathcal{T}} \neg p_2$.
Since $\mathcal{A}$ is deterministic, every path in the corresponding PDS (starting in $((q_0, s), \epsilon)$) labeled with such a $w$ runs through a state $((q, s_n), v)$, where $q \in F$ and $v \in \Gamma^*$. Since $(q, s_n) \notin \ell'(p_2)$, every such path ends in the next state which is $(b, \epsilon)$, where $p_b$ holds. Therefore the LTL formula $\mathtt{F}p_b$ holds in $\mathcal{T}_{\mathcal{A}}$ with the initial state being $((q_0, s), \epsilon)$.
The "if" direction: Assume $((q_0, s), \epsilon) \models_{\mathcal{T}_{\mathcal{A}}} \mathtt{F}p_b$. Hence, every path $\pi$ starting in $((q_0, s), \epsilon)$ ends in a state in which $p_b$ holds and therefore has to run through a state $((q, t), v)$, where $q \in F$, $(q, t) \notin \ell'(p_2)$ and $v \in \Gamma^*$.
Note that since $(g, \epsilon)$ and $(b, \epsilon)$ are dead-ends in $\mathcal{T}_{\mathcal{A}}$, no state along $\pi$ may satisfy either of the constraints for both transition types leading to such a dead-end and only transitions of the third kind can be taken. Hence, for all states $((q', s'), v')$ along $\pi$, we have that before $((q, t), v)$ is reached, $(q', s) \notin \ell'(p_1)$ must hold.
Therefore on all paths a $w \in \mathcal{L}(\mathcal{A})$ exists, s.t. $s \xrightarrow{w} t$ and along each such path $\neg p_1$ holds until $((q, t), v)$ is reached, where $\neg p_2$ holds. Hence, $s \models_{\mathcal{T}} A(\neg p_1\mathtt{U}^{\mathcal{A}}\neg p_2)$. From this follows clearly that $s \not\models_{\mathcal{T}} E(p_1\mathtt{R}^{\mathcal{A}}p_2)$.
Finally, it is known that model checking a fixed LTL formula on a PDS is in PTIME [BEM97]. Since the size of $\mathcal{T}_{\mathcal{A}}$ is polynomial in $|\mathcal{T}|$ and $|\mathcal{A}|$ the claim follows. □

**Theorem 57** Let $\mathfrak{A} \in \{\text{DFA}, \text{DVPA}, \text{DPDA}\}$. Model checking ER[$\mathfrak{A}$] is PTIME-complete.

Proof Along the same lines as the proof of Thm. 53. Membership in PTIME follows from the PTIME implementation of MC-R in algorithm MC-CTL given in Lemma 25 which is called at most $\mathcal{O}(|\varphi|)$ times for a formula $\varphi \in$ ER[DPDA]. Since DFA and DVPA are subclasses of DPDA, the result transfers to these. PTIME-hardness follows from PTIME-hardness of the corresponding PDL[$\mathfrak{A}$] fragments. □

Regarding nondeterministic machine models, the model checking problem seems to become more difficult. Here, we obtain PSPACE-hardness already for the class NFA.

**Lemma 26** Model checking $\text{ER}_{\mathcal{P}}$[NFA] is in PSPACE.

Proof By a reduction to the problem of model checking a fixed CTL formula on an LTS of exponential size. Let $\varphi \in \text{ER}_{\mathcal{P}}[\text{NFA}]$, $\mathcal{T} = (\mathcal{S}, \rightarrow, \ell)$ be an LTS and $r \in \mathcal{S}$. Clearly, $\varphi$ is of the form $\mathtt{E}(p_1\mathtt{R}^{\mathcal{A}}p_2)$, where $p_1, p_2 \in \mathcal{P}$ and $\mathcal{A}$ is an NFA.

First of all, we construct a DFA $\mathcal{D} = (Q, \Sigma, \delta, q_0, F)$ from $\mathcal{A}$. The size of $\mathcal{D}$ is $\mathcal{O}(2^{|\mathcal{A}|})$. Now we construct an LTS $\mathcal{T}_\mathcal{D} = (Q \times \mathcal{S}, \xrightarrow{\times}, \ell')$ with

- $(q, s) \xrightarrow{\times} (q', s')$, if there exists $a \in \Sigma$ s.t. $q' \in \delta(q, a)$ and $s \xrightarrow{a} s'$.
- $\ell'(p) = Q \times \ell(p)$, if $p \in \mathcal{P}$ and $\ell'(p_f) = F \times \mathcal{S}$ for a fresh proposition $p_f$ otherwise.

Note that the size of $\mathcal{T}_\mathcal{D}$ is $\mathcal{O}(|\mathcal{T}| \cdot 2^{|\mathcal{A}|})$.
Intuitively, the determinisation enables to annotate each model state with a unique indication of the corresponding automaton state for any path leading to this state. If the NFA is not transformed into a DFA, such an annotation is useless since it just reflects an arbitrary run of the NFA and makes no statement about the fact whether the automaton actually could accept the path seen so far in some other run on the same path.
The product construction has eliminated the edge labels from $\mathcal{T}$ and compensates the loss of information by the additional proposition $p_f$ which indicates accepting states of the DFA. It is now possible to model check the CTL formula $\mathtt{E}(p_1 \mathtt{R} (p_f \wedge p_2))$ on the product LTS $\mathcal{T}_\mathcal{D}$ which respects the accepting states. We conclude by showing

$$r \models_\mathcal{T} \mathtt{E}(p_1 \mathtt{R}^\mathcal{A} p_2) \quad \text{iff} \quad (q_0, r) \models_{\mathcal{T}_\mathcal{D}} \mathtt{E}(p_1 \mathtt{R} (p_f \wedge p_2)).$$

The "only-if" direction: Assume $r \models_\mathcal{T} \mathtt{E}(p_1 \mathtt{R}^\mathcal{A} p_2)$ and let $\pi = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots$ be a path in $\mathcal{T}$, where $s_0 = r$ and for all $s_i$ we have that if $a_1 \ldots a_i \in \mathcal{L}(\mathcal{A})$ then $s_i \models_\mathcal{T} p_2$ or there exists $k \leq i$ s.t. $s_k \models_\mathcal{T} p_1$.
Clearly, there is a corresponding path $\pi' = (q_0, s_0) \rightarrow (q_1, s_1) \xrightarrow{a_2} \ldots$ in $\mathcal{T}_\mathcal{D}$ where all states $(q_i, s_i)$ are labeled with $p_f$ if $a_0 \ldots a_i \in \mathcal{L}(\mathcal{A})$. Since the labels are otherwise inherited from $\mathcal{T}$, we have that $\pi'$ is a witness for $(q_0, r) \models_{\mathcal{T}_\mathcal{D}} \mathtt{E}(p_1 \mathtt{R} (p_f \wedge p_2))$.
The "if" direction: The witnessing path is constructed entirely dual to the other direction.
Model checking a fixed CTL formula is well-known to reside in NLOGSPACE. Since the product LTS has size $\mathcal{O}(|\mathcal{T}| \cdot 2^{|\mathcal{A}|})$ we arrive at a compound complexity of PSPACE using Savitch's theorem (NPSPACE = PSPACE). □

**Theorem 58 ([ALL$^+$b])** Model checking ER[NFA] is PSPACE-complete.

Proof The upper bound follows the same lines as the proof of Thm. 53. Membership in PSPACE follows from the PSPACE implementation of MC-R in algorithm MC-CTL given in Lemma 26 which is called at most $\mathcal{O}(|\varphi|)$ times for a formula $\varphi \in$ ER[NFA]. PSPACE-hardness is proved in [ALL$^+$b]. □

The theorem holds already for the fragment EG[NFA] and a fixed transition system of size 1. The proof works by a reduction from the well-known $n$-tiling problem resembling the halting problem of a nondeterministic linear-space bounded Turing Machine. Two aspects are worth noting. First, this result – as opposed to the one for the fragment EF[$\mathfrak{A}$] – heavily depends on the fact that $\mathfrak{A}$ is a class of nondeterministic automata. For $\mathfrak{A}$ = DFA for instance, there is no such lower bound unless PSPACE = PTIME.
The other aspect is the fact that the formulas constructed in this reduction are of the form $\mathtt{EG}^{\mathcal{A}}\mathtt{ff}$, no boolean operators, no multiple temporal operators, and no atomic propositions are needed. The principle is as follows. Tilings, successful or not, can be represented by infinite words over the alphabet of all tiles. This basically concatenates the entire plane row by row. However, unsuccessful tilings must have a finite prefix which is a word that cannot be extended to a successful tiling. The reduction then constructs an automaton $\mathcal{A}$ which recognises the set of all words representing a prefix of a tiling which cannot be extended to become successful. Every possible tiling is represented by a path in a one-state transition system with universal transition relation. The question whether or not a successful tiling is possible then reduces to the question whether or not this single state satisfies the formula $\mathtt{EG}^{\mathcal{A}}\mathtt{ff}$, i.e. whether or not there is a path such that no prefix of that path represents an error in the tiling of the corresponding plane.

**Theorem 59 ([ALL$^+$b])** Model checking ER[VPA] is EXPTIME-complete.

PROOF The upper bound is easily obtained as follows. By Thm. 6 we can construct a DVPA of exponential size from a given VPA. The result then follows from the PTIME upper bound for model checking ER[DVPA] established in Thm. 57.
The lower bound has been proved in [ALL$^+$b] by a reduction from the halting problem for alternating linear-space bounded Turing machines to the model checking problem for EG[VPA]. It does already hold for transition systems of size 1. □

**Theorem 60 ([ALL$^+$b])** Model checking ER[PDA] is undecidable.

PROOF The theorem has been proved in [ALL$^+$b] and holds already for the fragment EG[PDA] and a fixed transition system of size 1. The proof is, again, by a reduction from a tiling problem. This time we consider the octant tiling problem which asks for a successful tiling of the plane which has successively longer rows [vEB97]. The plane can, again, be represented by an $\omega$-word by reading it off row-by-row and, hence, as a path in a one-state transition system. Using PDA it is then possible to link a cell in one row of unbounded

length to the cell in the same column in the following row. Thus, it is then again possible to construct a PDA $\mathcal{A}$ which recognises all prefixes of a word representing a tiling which cannot be made successful, or a word in which successive rows do not grow in length. The tiling problem reduces to model checking the formula $\texttt{EG}^{\mathcal{A}}\texttt{ff}$ again. Since the octant tiling problem resembles the halting problem for a Turing Machine with unbounded space consumption, it is clearly undecidable which carries over to model checking EG[PDA]. □

**Summary** The previous theorems on different fragments of CTL[$\mathfrak{A}, \mathfrak{B}$] cover all cases necessary to give matching upper and lower bounds on model checking the full logics. The following table summarises the computational complexities of each combination of automata classes in either fragment under consideration. If complexity class $\mathcal{C}$ is positioned in row $x$ and column $y$ then the logic CTL[$\mathfrak{A}, \mathfrak{B}$] is $\mathcal{C}$-complete, where $\mathfrak{A}$ occurs leftmost in row $x$ and $\mathfrak{B}$ occurs on top of colum $y$. These results are simple corollaries of the theorems in this section.

<table>
<tr><th></th><th>DFA</th><th>DVPA</th><th>DPDA</th><th>NFA</th><th>VPA</th><th>PDA</th></tr>
<tr><td>REG</td><td colspan="3" rowspan="4">PTIME</td><td rowspan="4">PSPACE</td><td rowspan="4"></td><td rowspan="5"></td></tr>
<tr><td>VPL</td></tr>
<tr><td>CFL</td></tr>
<tr><td>LIL</td></tr>
<tr><td>IL</td><td colspan="5">EXPTIME</td></tr>
<tr><td>CSL</td><td colspan="6">undecidable</td></tr>
</table>

Figure 4.3: Complexity of model checking CTL[$\mathfrak{A}, \mathfrak{B}$].

For the EU[$\mathfrak{A}$] fragment, the representation of formal languages – as long as they fulfill the basic requirements aforementioned – is not relevant. For formulas of ER[$\mathfrak{B}$] it is however relevant in terms of deterministic and nondeterministic automata models. Corresponding results for other representations can be transferred as long as the translation to the adequate automaton class takes at most polynomial time.

Despite the high expressivity in comparison to classical temporal logics, the table shows that there is a wide range of logics with very feasible model checking complexity. Note that formulas of e.g. CTL[LIL, DPDA] are capable of describing path properties even beyond the context-free, yet the model checking problem is solvable in PTIME. But even the greatest fragment of CTL[TM, TM], namely CTL[IL, VPA] is still model checkable in EXPTIME.

### 4.5.1 Model checking EU[PDA]

The reductions in the proofs of the previous sections provide tight bounds on the model checking problems for various logics, they may however not be suitable for ad-hoc implementations. In this section and the following, we give concrete implementations of the subroutines MC-U and MC-R for key classes of automata which complete the algorithm MC-CTL. We start with an abstract version of MC-U for EU[PDA] formulas and explain each subroutine in the following.

$\text{MC-U}(\mathcal{T}', \mathtt{E}(p_1 \mathtt{U}^{\mathcal{A}} p_2)) =$
  $\mathtt{let}\ \mathcal{T} = \mathtt{reduce\text{-}LTS}(\mathcal{T}')\ \mathtt{in}$
  $\mathtt{let}\ \mathcal{A}_{\mathcal{T}} = \mathtt{build\text{-}product}(\mathcal{T}, \mathcal{A})\ \mathtt{in}$
  $\mathtt{let}\ \mathcal{M} = \mathtt{compute\text{-}pre}(\mathcal{A}_{\mathcal{T}})\ \mathtt{in}$
    $\mathtt{extract\text{-}states}(\mathcal{M})$

MC-U gets as arguments an LTS $\mathcal{T}'$ and a formula $\mathtt{E}(p_1 \mathtt{U}^{\mathcal{A}} p_2)$, where $\mathcal{A}$ is a PDA. Regardless of the operations of $\mathcal{A}$, in order to find a path along which $p_1$ holds until $p_2$ holds, we may eliminate all states of $\mathcal{T}'$ in which neither proposition holds. We call this procedure `reduce-LTS` and assume that it takes as argument the LTS $\mathcal{T}' = (\mathcal{S}', \rightarrow', \ell')$ and returns an LTS $\mathcal{T} = (\mathcal{S}, \rightarrow, \ell)$, s.t.

- $\mathcal{S} = \{s \in \mathcal{S}' \mid s \in \ell'(p_1) \text{ or } s \in \ell'(p_2)\}$,
- $\rightarrow = \rightarrow' \ \cap\ \mathcal{S} \times \Sigma \times \mathcal{S}$,
- $\ell : \{p_1, p_2\} \rightarrow 2^{\mathcal{S}}$ is the a function with $\ell(p) = \ell'(p) \setminus \overline{\mathcal{S}}$.

Recall the product PDA constructed in the proof of Lemma 24 and assume it is computed by a procedure `build-product` which takes the reduced LTS $\mathcal{T} = (\mathcal{S}, \rightarrow, \ell)$ and a PDA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ and returns the product automaton $\mathcal{A}_{\mathcal{T}} = (Q \times \mathcal{S}, \Sigma, \Gamma, \delta', (q_0, s), F')$, where

- $F' = \{(q, s) \mid q \in F \text{ and } s \in \ell(p_2)\}$,
- $\delta'((q, s), a, \gamma) = \{(q', s') \mid q' \in \delta(q, a, \gamma) \text{ and } s \xrightarrow{a} s' \text{ and } s \in \ell(p_1)\}$.

for an arbitrary $s \in \mathcal{S}$ in the starting state $(q_0, s)$ of $\mathcal{A}_{\mathcal{T}}$. It is arbitrary, because we will use $\mathcal{A}_{\mathcal{T}}$ rather in the fashion of a pushdown system and compute predecessor configurations in a bottom-up algorithm where the starting state does not matter.

Consider the set of configurations $\mathsf{Conf}(\mathcal{A}_\mathcal{T}) = \{(q,s,w) \mid q \in Q \text{ and } s \in \mathcal{S} \text{ and } w \in \Gamma^*\}$ which $\mathcal{A}_\mathcal{T}$ may take. We define the set of goal configurations $\mathsf{Goal}(\mathcal{A}_\mathcal{T})$ as $F' \times \Gamma^*$ and the set of starting configurations as $\mathsf{Start}(\mathcal{A}_\mathcal{T}) = \{q_0\} \times \mathcal{S} \times \{\epsilon\}$.
Furthermore, define the set of (immediate) predecessors of a set of configurations $C \subseteq \mathsf{Conf}(\mathcal{A}_\mathcal{T})$ as

$$\begin{aligned}\mathsf{Pre}(C) \;=\; \{(q,s,\gamma w) \in \mathsf{Conf}(\mathcal{A}_\mathcal{T}) \mid \;& \text{there exists } (q',s',v'w) \in C \\ & \text{and } a \in \Sigma \text{ s.t. } ((q',s'),v') \in \delta'((q,s),a,\gamma)\}.\end{aligned}$$

**Lemma 27** Let $c_0 = (q_0, s', \epsilon)$ be in $\mathsf{Start}(\mathcal{A}_\mathcal{T})$. Furthermore, let $\mathcal{A}'_\mathcal{T}$ be defined as $\mathcal{A}_\mathcal{T}$, except for the starting state which is $(q_0, s')$.

$$c_0 \in \mathsf{Pre}^*(\mathsf{Goal}(\mathcal{A}_\mathcal{T})) \quad\text{iff}\quad s \models_\mathcal{T} \mathtt{E}(p_1 \mathtt{U}^\mathcal{A} p_2).$$

Proof "only-if-direction": Clearly, $c_g$ is an accepting configuration which is reachable from $c_0$ and hence $\mathcal{L}(\mathcal{A}'_\mathcal{T}) \neq \emptyset$. Since the starting state of $\mathcal{A}'_\mathcal{T}$ has been exchanged to fit the requirements of the proof in Lemma 24, the result is an immediate consequence.
"if-direction": If $s \models_\mathcal{T} \mathtt{E}(p_1 \mathtt{U}^\mathcal{A} p_2)$ then $\mathcal{L}(\mathcal{A}'_\mathcal{T}) \neq \emptyset$. From this again follows the claim. □

This reduces the task of determining the set of states in which $\mathtt{E}(p_1 \mathtt{U}^\mathcal{A} p_2)$ holds to the task of computing $\mathsf{Start}(\mathcal{A}_\mathcal{T}) \cap \mathsf{Pre}^*(\mathsf{Goal}(\mathcal{A}_\mathcal{T}))$ and extracting the model states from the resulting configurations which are exactly those $c_0$ for which Lemma 27 applies.
Our procedure for the computation of $\mathsf{Pre}^*$ is a specialisation of the idea found in [BEM97]. The procedure `compute-pre` takes the product automaton $\mathcal{A}_\mathcal{T}$ and computes the set of its predecessor configurations. The basic data structure on which the procedure operates is called a multi-automaton which resembles an NFA with every state being a starting state.

**Definition 39 (Multi-Automaton)** Let $\mathcal{A}_\mathcal{T} = (Q \times \mathcal{S}, \Sigma, \Gamma, \delta', (q_0, s), F')$ be a PDA. A multi-automaton for $\mathcal{A}_\mathcal{T}$ is a 5-tuple $\mathcal{M} = (Q \times \mathcal{S}, \rightarrow, F')$, where

- $Q \times \mathcal{S}$ is a set of (product) states,
- $F'$ is the set of final states inherited from $\mathcal{A}_\mathcal{T}$,
- $\rightarrow \subseteq (Q \times \mathcal{S}) \times \Gamma \times (Q \times \mathcal{S})$ is the transition relation.

We use infix notation for the transition relation $\rightarrow$ and write $(q,s) \xrightarrow{\gamma} (q',s')$ instead of $((q,s),\gamma,(q',s')) \in \rightarrow$. We also extend $\rightarrow$ to $w \in \Gamma^*$ in the same way as for an LTS.

A multi-automaton accepts a set of configurations $\mathcal{C} = \{(q, s, w) \mid \exists (q', s') \in F' \text{ s.t. } (q, s) \xrightarrow{w} (q', s')\}$.
Intuitively, `compute-pre` builds a multi-automaton which accepts all goal configurations initially and successively adds transitions which enrich the set of accepted configurations to the set of predecessor configurations. The helper routine `build-transitions`$(\mathcal{A}_\mathcal{T})$ in the following is expected to return the initial transition relation of a multi-automaton with self transitions on all final states for all stack symbols: $\rightarrow \; = \{((q, s), \gamma, (q, s)) \mid (q, s) \in F' \text{ and } \gamma \in \Gamma\}$. Hence, the initial set of accepted configurations is $F' \times \Gamma^*$, i.e. the set $\mathsf{Goal}(\mathcal{A}_\mathcal{T})$. In order to distinguish the previously computed transition relation from the current one, we use $\underset{l}{\rightarrow}$ and $\underset{c}{\rightarrow}$.

compute-pre$(\mathcal{A}_\mathcal{T}) =$
  let $(Q \times \mathcal{S}, \Sigma, \Gamma, \delta', (q_0, s), F') = \mathcal{A}_\mathcal{T}$ in
    $\underset{c}{\rightarrow}$ := build-transitions$(\mathcal{A}_\mathcal{T})$
    repeat
      $\underset{l}{\rightarrow} := \underset{c}{\rightarrow}$
      for all $((q, s), \gamma) \hookrightarrow ((q', s'), w) \in \delta$
        if $\exists (q'', s'') \in Q \times \mathcal{S}$ s.t. $(q', s') \xrightarrow{w} (q'', s'')$
        then $\underset{c}{\rightarrow} := \underset{c}{\rightarrow} \cup ((q, s), \gamma, (q'', s''))$
    until $\underset{c}{\rightarrow} = \underset{l}{\rightarrow}$
    return $(Q \times \mathcal{S}, \underset{c}{\rightarrow}, F')$

**Lemma 28 (Termination)** Procedure `compute-pre`$(\mathcal{A}_\mathcal{T})$ runs in time $\mathcal{O}(|Q|^2 \cdot |\mathcal{S}|^2 \cdot |\Gamma| \cdot |\delta|)$ for a PDA $\mathcal{A}_\mathcal{T} = (Q \times \mathcal{S}, \Sigma, \Gamma, \delta', (q_0, s), F')$.

Proof The `repeat`-loop finishes after at most $(|Q|^2 \cdot |\mathcal{S}|^2 \cdot |\Gamma|) - 1$ iterations, because this is the maximum size of $\underset{c}{\rightarrow}$ and after each iteration, at least one additional transition must enter $\underset{c}{\rightarrow}$ to prevent earlier termination. Inside the `repeat`-loop there are $|\delta|$ many checks of the `if`-condition. These can however be reduced to constant time, since it is only necessary to check for $w$-matches of a newly entered transition, because all other checks are redundant. Note that $|w| \leq 2$ in our definition of a PDA (it is 0 for pop-, 2 for push-, and 1 for non-changing stack operations). The costs of `build-transitions`$(\mathcal{A}_\mathcal{T})$ are also constant in the above worst-case scenario, because initially only one transition may occur in $\underset{c}{\rightarrow}$. □

**Lemma 29 (Soundness and Completeness)** Let $\mathcal{M}$ be the multi-automaton computed by $\texttt{compute-pre}(\mathcal{A}_\mathcal{T})$ for the product PDA $\mathcal{A}_\mathcal{T}$. The set of accepted configurations of $\mathcal{M}$ coincides with $\mathsf{Pre}^*(\mathsf{Goal}(\mathcal{A}_\mathcal{T}))$.

Proof The set of accepted configurations of $\mathcal{M}$ at any time during the computation is $\mathcal{C} = \{(q,s,w) \mid \exists (q',s') \in F' \text{ s.t. } (q,s) \xrightarrow[c]{w} (q',s')\}$, but depends on the monotonically growing $\xrightarrow[c]{}$. We start with showing $\mathcal{C} \subseteq \mathsf{Pre}^*(\mathsf{Goal}(\mathcal{A}_\mathcal{T}))$ by induction on the sequence of (different) accepted configurations $\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_n$ of $\mathcal{M}$ during the computation.
Initially, $\mathcal{C}_0$ is clearly a subset of $\mathsf{Pre}^*(\mathsf{Goal}(\mathcal{A}_\mathcal{T}))$, since $\mathcal{C}_0 = \mathsf{Goal}(\mathcal{A}_\mathcal{T})$).
Now assume $((q,s),\gamma,(q'',s''))$ enters $\xrightarrow[c]{}$ during some iteration inside the $\texttt{repeat}$-loop and therefore constitutes some $\mathcal{C}_{i+1}$. We then have that there exists $(q',s')$ and a $w \in \Gamma^*$, s.t. $((q,s),\gamma) \hookrightarrow ((q',s'),w) \in \delta$ and $(q',s') \xrightarrow{w} (q'',s'')$.
Note that every transition added to $\xrightarrow[c]{}$ to constitute $\mathcal{C}_{i+1}$ comes from a state $(q,s)$ and leads to a state $(q',s')$ which is already connected with a state in $F'$. This is due to the fact that in the if-condition $(q',s') \xrightarrow{w} (q'',s'')$ is required and at the beginning only paths to final states exist. Hence, $(q'',s'')$ is either a final state or leads to one.
Therefore, the path $(q',s') \xrightarrow[c]{w} (q'',s'') \xrightarrow[c]{u} f$ exists, where $u \in \Gamma^*$ and $f \in F'$. But then $(q',s',wu) \in \mathcal{C}_i$ and we have by I.H. that $(q',s',wu) \in \mathsf{Pre}^*(\mathsf{Goal}(\mathcal{A}_\mathcal{T}))$. But since clearly $((q,s),\gamma u)$ is an immediate predecessor configuration of $((q',s'),wu)$, we have that $((q,s),\gamma u)$ is contained within $\mathsf{Pre}^*(\mathsf{Goal}(\mathcal{A}_\mathcal{T}))$.
For the direction $\mathcal{C} \supseteq \mathsf{Pre}^*(\mathsf{Goal}(\mathcal{A}_\mathcal{T}))$, let $(q,s,w) \in \mathsf{Pre}^*(\mathsf{Goal}(\mathcal{A}_\mathcal{T}))$. This means that there exists a sequence of configurations $(q_0,s_0,w_0),(q_1,s_1,w_1),\ldots,(q_n,s_n,w_n)$, s.t.

- $(q_n,s_n,w_n) \in F' \times \Gamma^*$ (1)
- $(q_0,s_0,w_0) = (q,s,w)$ (2)
- for all $i \geq 0$ exist $v_{i+1} \in \Gamma^*$ and $\gamma_i \in \Gamma$: $((q_i,s_i),\gamma_i) \hookrightarrow ((q_{i+1},s_{i+1}),v_{i+1})$ and $w_i = \gamma_i w'_i$ and $w_{i+1} = v_{i+1} w'_i$ (3)

It suffices to prove that for all $0 \leq k \leq n$ we have $(q_k,s_k) \xrightarrow[c]{w_k} (q_n,s_n)$, because then clearly for all $k$, $(q_k,s_k,w_k) \in \mathcal{C}$ and in particular $(q,s,w) \in \mathcal{C}$.
For $k = n$, we have that $(q_k,s_k) \in F'$ and since the initial multi-automaton accepts any $w \in \Gamma^*$ on itself for such a state (and in particular $w_k$), the claim follows.
Assume $0 \leq k < n$. By I.H. we have $(q_{k+1},s_{k+1}) \xrightarrow[c]{w_{k+1}} (q_n,s_n)$. Note that from (3) it follows in particular that there exist $\gamma, v$, s.t. $((q_k,s_k),\gamma) \hookrightarrow ((q_{k+1},s_{k+1}),v)$ and $w_k = \gamma w'_k$

and $w_{k+1} = vw'_k$ for some $w'_k$. Hence, $(q_{k+1}, s_{k+1}) \xrightarrow[c]{vw'_k} (q_n, s_n)$ and therefore there clearly exists some $(q'', s'')$, s.t. $(q_{k+1}, s_{k+1}) \xrightarrow[c]{v} (q'', s'')$.
Note that the following conditions are now met:

- $((q_k, s_k), \gamma) \hookrightarrow ((q_{k+1}, s_{k+1}), v)$.
- $\exists (q'', s'')$, s.t. $(q_{k+1}, s_{k+1}) \xrightarrow[c]{v} (q'', s'')$.

Remember that these are exactly the conditions inside the `repeat`-loop for adding the transition $((q_k, s_k), \gamma, (q'', s''))$ to $\underset{c}{\rightarrow}$. We therefore have established the following path: $(q_k, s_k) \xrightarrow[c]{\gamma} (q'', s'') \xrightarrow[c]{w'_k} (q_n, s_n)$. Since $w_k = \gamma w'_k$, the claim follows. □

Finally, the procedure `extract-states` takes the multi-automaton $\mathcal{M}$ computed by the subroutine `compute-pre` and returns $\{s \in \mathcal{S} \mid (q_0, s, \epsilon) \in \mathsf{Start}(\mathcal{A}_\mathcal{T}) \cap \mathcal{C}\}$, where $\mathcal{C}$ is the set of predecessor configurations computed by $\mathcal{M}$. Note that it is easy to determine this set from given $\mathcal{M}$, since all states in $\mathcal{M}$ are starting states and if there is an outgoing edge from any state, then it leads to a final state. Hence this set is equal to $\{s \in \mathcal{S} \mid$ there exists $q' \in Q, s' \in \mathcal{S}, \gamma \in \Gamma$ s.t. $(q_0, s) \xrightarrow[c]{\gamma} (q', s')\}$.

**Theorem 61**

$$s \in \text{MC-U}(\mathcal{T}, \mathtt{E}(p_1 \mathtt{U}^\mathcal{A} p_2) \quad \text{iff} \quad s \models \mathtt{E}(p_1 \mathtt{U}^\mathcal{A} p_2).$$

Proof Follows from Lemmas 27 – 29. □

### 4.5.2 Model checking ER[DPDA]

While formulas in $\text{EU}_\mathcal{P}[\mathcal{L}]$ are always satisfied in the finite, a temporal formula $\varphi = \mathtt{E}(p\mathtt{R}^A q)$ in $\text{ER}_\mathcal{P}[\mathcal{L}]$ may also be satisfied on an infinite path: clearly, a state $s$ satisfies $\varphi$, if along an infinite path starting in $s$, the proposition $p$ is never seen and $q$ holds whenever a prefix of this path forms a word in $\mathcal{L}(\mathcal{A})$.
The general idea of the model checking algorithm for the logic $\text{ER}_\mathcal{P}[\text{DPDA}]$ has been presented in the proof of Lemma 25 already, where a PDS $\mathcal{T}_\mathcal{A} = (Q \times \mathcal{S} \cup \{g, b\}, \Gamma, \Delta, \ell')$ is constructed as a product of an LTS $\mathcal{T} = (\mathcal{S}, \rightarrow, \ell)$ and the DPDA $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, q_0, F)$ occurring in $\varphi$. Model checking $\varphi$ over $\mathcal{T}$ is reduced to model checking the fixed LTL formula $\mathtt{F}p_b$ over $\mathcal{T}_\mathcal{A}$.
We present here a direct implementation which checks the CTL formula $\mathtt{EG}\neg p_b$ instead. Clearly, this formula is dual to the LTL formula such that soundness remains intact.

**Lemma 1 ([BEM97])** Let $C$ be a configuration of a PDS $\mathfrak{P} = (Q, \Gamma, \Delta, \ell)$ and $q \in Q$. The control location $q$ is visited infinitely often along any path of $\mathfrak{P}$ starting in $C$ iff there exist configurations $(p, \gamma), (f, u)$ and $(p, \gamma w)$ with $\gamma \in \Gamma \cup \{\epsilon\}$ and $u, w \in \Gamma^*$, not all three equal, s.t. the following conditions are met:

- $C \in \mathsf{Pre}^*(\{p\} \times \gamma\Gamma^*)$.
- $(p, \gamma) \in \mathsf{Pre}^+((\{f\} \times \Gamma^*) \cap \mathsf{Pre}^*(\{p\} \times \gamma\Gamma^*))$.

The first condition simply claims that some configuration $(p, \gamma v)$ is reachable from $C$, where $v \in \Gamma^*$. Intuitively, this configuration is the starting point of some kind of cyclic behaviour of $\mathfrak{P}$: the second condition requires that from $(p, \gamma)$ a configuration $(f, u)$ is reachable which in turn is a predecessor of some configuration $(p, \gamma w)$. Hence the cycle $(p, \gamma), (p, \gamma w), (p, \gamma ww), \ldots$ can be repeated forever. Taken together, the conditions establish the following infinite configuration path:

$$C \rightsquigarrow (p, \gamma v) \rightsquigarrow (f, uv) \rightsquigarrow (p, \gamma wv) \rightsquigarrow (f, uwv) \rightsquigarrow (p, \gamma wwv) \rightsquigarrow \ldots$$

Note that from this follows that the control state $f$ is visited infinitely often.
Instead of model checking the LTL formula $\mathtt{F}p_b$ we may add a self-transition on the state $g$ in the PDS $\mathcal{T}_\mathcal{A}$ s.t. the only finite paths are those which end in configurations $(b, x)$ for some $x \in \Gamma^*$ and look for the existence of an infinite path.
This leads to the following implementation of algorithm MC-R for the logic ER[DPDA].

MC-R($\mathcal{T}$, $\mathtt{E}(p_1\mathtt{R}^\mathcal{A}p_2)$) =
    `let` $\mathcal{T}_\mathcal{A}$ = `build-PDS`($\mathcal{T}$, $\mathcal{A}$) `in`
        $V := \emptyset$
        `for each` $(p, \gamma) \in ((Q \times \mathcal{S}) \cup \{g\}) \times \Gamma$ `do`
            $\mathcal{M} := \mathsf{Pre}^+((p, \gamma\Gamma^*))$
            `if` $(p, \gamma) \in \mathcal{M}$ `then`
                $V := V \cup$ `extract-states`($\mathsf{Pre}^*((p, \gamma))$)
        `return` $V$

Subroutine `build-PDS` is supposed to return the product PDS $\mathcal{T}_\mathcal{A}$ from the proof of Lemma 25 enriched with self-transitions on the state $g$ in order to have every infinite configuration path satisfy the CTL formula $\mathtt{EG}\neg p_b$ since $p_b$ does only hold in dead-ends. The set of LTS states $V$ is used to store the result set, i.e. the set of states which satisfy $\mathtt{E}(p_1\mathtt{R}^\mathcal{A}p_2)$.

The central loop takes each combination of a PDS state $p$ and a stack symbol $\gamma$ and checks the existence of a cycle starting in the corresponding configuration. This is done by computing a multi-automaton $\mathcal{M}$ which represents the set of predecessor configurations $\mathsf{Pre}^+((p, \gamma\Gamma^*))$ and checking whether $(p, \gamma)$ is a member of this set.

If this is the case, the set of all predecessors of $(p, \gamma)$ is computed in turn, because all these predecessor configurations lead to a cycle. Hence if any such configuration $((q, s)w)$ is a member of $\mathsf{Start}(\mathcal{T}_\mathcal{A})$ then $s$ satisfies $\mathtt{E}(p_1\mathtt{R}^\mathcal{A}p_2)$ and is added to the result set $V$. Remember from Sec. 4.5.1 that `extract-states` extracts the LTS states of the intersection of $\mathsf{Start}(\mathcal{T}_\mathcal{A})$ and the configurations represented by a multi-automaton.

The computation of the relations $\mathsf{Pre}^+$ and $\mathsf{Pre}^*$ is very similar to what the procedure `compute-pre` from the previous section does and we therefore do not give details here and instead refer to [BEM97]. The procedure `compute-pre` is just a problem-optimised version of the general algorithm there.

**Theorem 62**

$$s \in \text{MC-R}(\mathcal{T}, \mathtt{E}(p_1\mathtt{R}^\mathcal{A}p_2) \quad \text{iff} \quad s \models \mathtt{E}(p_1\mathtt{R}^\mathcal{A}p_2).$$

Proof Let $s \in \text{MC-R}(\mathcal{T}, \mathtt{E}(p_1\mathtt{R}^\mathcal{A}p_2)$. Since $s \in V$, there exists $(p, \gamma) \in ((Q \times \mathcal{S}) \cup \{g\}) \times \Gamma$, s.t. $((s, q_0), \epsilon) \in \mathsf{Pre}^*(p, \gamma)$ and $(p, \gamma) \in \mathsf{Pre}^+(p, \gamma v)$ for some $v \in \Gamma^*$. But then there is an infinite path $\pi$ in $\mathcal{T}_\mathcal{A}$ starting in $((s, q_0), \epsilon)$. Since there are no outgoing edges from configurations $(b, x)$ for any $x \in \Gamma^*$ and $b$ is the only state in which proposition $p_b$ holds, no state along the LTS-related component of $\pi$ satisfies $p_b$. Hence $((s, q_0), \epsilon) \models_{\mathcal{T}_\mathcal{A}} \mathtt{EG}\neg p_b$ and by construction of $\mathcal{T}_\mathcal{A}$ we have $s \models \mathtt{E}(p_1\mathtt{R}^\mathcal{A}p_2)$.

Let $s \models \mathtt{E}(p_1\mathtt{R}^\mathcal{A}p_2)$. By construction of $\mathcal{T}_\mathcal{A}$ there exists an infinite path starting in $((s, q_0), \epsilon)$. Any infinite path has a cycle which is detected by the central loop and results in $s$ being stored in $V$. □

# Chapter 5

# Higher-Order Fixpoint Logic

In order to give a logical characterisation of *context-free processes* (CFP) [BK85], Müller-Olm extended $\mathcal{L}_\mu$ with a sequential composition operator and named the resulting logic *Fixpoint Logic with Chop*[1] (FLC) [MO99]. It is capable of expressing many non-regular – and even non-context-free – program properties and thus exceeds the expressivity of the $\mathcal{L}_\mu$ [MO99, LS06]. Given that FLC is capable of expressing characteristic formulas for the simulation of CFP, deciding simulation between CFP can be reduced on model checking FLC. But since this is known to be undecidable, the same holds for model checking FLC [MO99]. On finite state systems, the model checking problem for FLC is however in EXPTIME [MO99, LS02, Lan02].

The semantics of an $\mathcal{L}_\mu$-formula $\varphi$ w.r.t. an LTS is the set of states in which $\varphi$ holds and hence a predicate on the total state set $\mathcal{S}$. In contrast, the semantics of FLC is given as a predicate transformer on states, i.e. a (monotonic) function of type $2^{\mathcal{S}} \to 2^{\mathcal{S}}$. The sequential composition operator ";" is interpreted as function composition, i.e. an FLC formula $\psi_1; \psi_2$ is interpreted as $[\![\psi_1]\!] \circ [\![\psi_2]\!]$.

This idea has been generalised in Mahesh and Ramesh Viswanathan's *Higher Order Fixpoint Logic* (HFL), where $\mathcal{L}_\mu$ was equipped with a *simply typed* $\lambda$*-calculus* s.t. now arbitrary function types based on the primitive type $2^{\mathcal{S}}$ can be built [VV04]. This makes it even more expressive than FLC. It is possible for instance to express *assume-guarantee-properties* in HFL [VV04].

Nevertheless, the model checking problem on finite state systems remains decidable, since all occurring functions operate on finite domains and are thus effectively computable.

This section is organised as follows. After the definition of syntax and semantics, model-

[1] The name is a reference to Interval Logics, where the sequential composition operator is called "chop".

theoretic properties and an expressivity analysis, we give a model checking algorithm for HFL which is a generalisation of the algorithm we presented in [AL07] for the first-order fragment of HFL. Since this algorithm optimises the straight-forward fixpoint approximation for HFL, we give empirical evidence that it indeed enhances the performance vastly in practice. Thereafter, we will argue that the analysis of the behaviour of our optimised model checker can be a valuable tool for the development of new algorithms and demonstrate this on a couple of examples.

## 5.1 Syntax and Semantics

**Definition 40 (Type)** Let $\mathcal{T} = (\mathcal{S}, \rightarrow, \ell)$ be an LTS and a $v \in \{-, +, 0\}$ be called a variance. The set of *HFL types* is the smallest set containing the atomic type $\mathsf{Pr}$ and is closed under function typing with variances, i.e. if $\sigma$ and $\tau$ are HFL types and $v$ is a variance, then $\sigma^v \rightarrow \tau$ is an HFL type.

**Definition 41 (Term)** Let $\mathcal{P}$ be a countably infinite set of atomic propositions, $\Sigma$ be a finite set of action names, $\mathcal{V}$ a countably infinite set of variables. The set of *HFL terms* is given by the following grammar:

$$\varphi \ ::= \ q \mid X \mid \neg\varphi \mid \langle a \rangle\varphi \mid \varphi\ \varphi \mid \lambda(X^v : \tau).\varphi \mid \mu(X : \tau).\varphi$$

where $q \in \mathcal{P}$, $X \in \mathcal{V}$, $a \in \Sigma$, $v$ is a variance and $\tau$ is an HFL type.

We use the following standard abbreviations:

$$\begin{array}{rclrcl}
\mathtt{tt} & := & q \vee \neg q \ \text{ for some } q \in \mathcal{P}, & \mathtt{ff} & := & \neg\mathtt{tt}, \\
\varphi \wedge \psi & := & \neg(\neg\varphi \vee \neg\psi), & \varphi \rightarrow \psi & := & \neg\varphi \vee \psi, \\
\varphi \leftrightarrow \psi & := & (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi), & \nu X.\varphi & := & \neg\mu X.\neg\varphi[\neg X/X], \\
[a]\psi & := & \neg\langle a \rangle\neg\psi, & \langle - \rangle\varphi & := & \bigvee_{a\in\Sigma}\langle a \rangle\varphi, \\
[-]\varphi & := & \bigwedge_{a\in\Sigma}[a]\varphi. & & &
\end{array}$$

where $\varphi[\psi/X]$ denotes the formula that results from $\varphi$ by replacing simultaneously every occurrence of $X$ by $\psi$.

**Definition 42 (Formula)** A sequence $\Gamma$ of the form $X_1^{v_1} : \tau_1, \ldots, X_n^{v_n} : \tau_n$ where $X_i$ are variables, $\tau_i$ are types and $v_i$ are variances is called a *context* (we assume all $X_i$ are distinct). An HFL term $\varphi$ has type $\tau$ in context $\Gamma$ if the statement $\Gamma \vdash \varphi \colon \tau$ can be inferred using the

$$\frac{}{\Gamma \vdash q \colon \mathsf{Pr}} \qquad \frac{v \in \{0,+\}}{\Gamma, X^v \colon \tau \vdash X : \tau} \qquad \frac{\Gamma^- \vdash \varphi \colon \tau}{\Gamma \vdash \neg\varphi \colon \tau}$$

$$\frac{\Gamma \vdash \varphi \colon \mathsf{Pr} \quad \Gamma \vdash \psi \colon \mathsf{Pr}}{\Gamma \vdash \varphi\, \psi \colon \mathsf{Pr}} \qquad \frac{\Gamma \vdash \varphi \colon \mathsf{Pr}}{\Gamma \vdash \langle a \rangle \varphi \colon \mathsf{Pr}} \qquad \frac{\Gamma, X^v \colon \sigma \vdash \varphi \colon \tau}{\Gamma \vdash \lambda(X^v \colon \sigma).\varphi : (\sigma^v \to \tau)}$$

$$\frac{\Gamma \vdash \varphi : (\sigma^+ \to \tau) \quad \Gamma \vdash \psi \colon \sigma}{\Gamma \vdash (\varphi\, \psi) : \tau} \qquad \frac{\Gamma \vdash \varphi : (\sigma^- \to \tau) \quad \Gamma^- \vdash \psi \colon \sigma}{\Gamma \vdash (\varphi\, \psi) : \tau}$$

$$\frac{\Gamma \vdash \varphi : (\sigma^0 \to \tau) \quad \Gamma \vdash \psi \colon \sigma \quad \Gamma^- \vdash \psi \colon \sigma}{\Gamma \vdash (\varphi\, \psi) : \tau} \qquad \frac{\Gamma, X^+ \colon \tau \vdash \varphi \colon \tau}{\Gamma \vdash \mu(X \colon \tau).\varphi : \tau}$$

Figure 5.1: Type inference rules for HFL.

rules of Fig. 5.1. We say that $\varphi$ is *well-formed* if $\Gamma \vdash \varphi \colon \tau$ for some $\Gamma$ and $\tau$. A well-formed HFL term of type $\mathsf{Pr}$ is called a *formula*. For a variance $v$, we define its complement $v^-$ as $+$ if $v = -$, as $-$ if $v = +$, and 0 otherwise. For a context $\Gamma = X_1^{v_1} \colon \tau_1, \ldots, X_n^{v_n} \colon \tau_n$, the complement $\Gamma^-$ is defined as $X_1^{v_1^-} : \tau_1, \ldots, X_n^{v_n^-} : \tau_n$.

The purpose of variances in the typing system is to ensure that in a term $\mu(x : \tau).\varphi$, $\varphi$ is monotonic in $x$ because otherwise the existence of a fixpoint cannot be guaranteed. While in $\mathcal{L}_\mu$ it suffices to require every occurrence of $x$ to appear under an even number of negation symbols, this requirement is too weak in the presence of $\lambda$-abstractions, since the actual negative or positive occurrence may be hidden in nested function abstractions and applications. Consider for instance the following term (taken from [VV04]):

**Example 14**

$$\mu(f : \mathsf{Pr}^- \to \mathsf{Pr}).\lambda(z^- : \mathsf{Pr}).\mu(x : \mathsf{Pr}).f(\neg x) \vee \neg z : \mathsf{Pr}^- \to \mathsf{Pr}$$

Its type derivation is shown in Fig.5.2, where $f$ appears positively and $z$ negatively. The variance of $x$ – seemingly negative – however depends on the variance of its applicator $f$. If $f$ was anti-monotone, $x$ would occur positively.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{f^{+} : \mathsf{Pr}^{-} \to \mathsf{Pr}, z^{-} : \mathsf{Pr}, x^{+} : \mathsf{Pr} \vdash f : \mathsf{Pr} \to \mathsf{Pr} \qquad \dfrac{f^{+} : \mathsf{Pr}^{-} \to \mathsf{Pr}, z^{-} : \mathsf{Pr}, x^{+} : \mathsf{Pr} \vdash x : \mathsf{Pr}}{f^{-} : \mathsf{Pr}^{-} \to \mathsf{Pr}, z^{+} : \mathsf{Pr}, x^{-} : \mathsf{Pr} \vdash \neg x : \mathsf{Pr}}}{f^{+} : \mathsf{Pr}^{-} \to \mathsf{Pr}, z^{-} : \mathsf{Pr}, x^{+} : \mathsf{Pr} \vdash f(\neg x) : \mathsf{Pr}} \qquad \dfrac{f^{-} : \mathsf{Pr}^{-} \to \mathsf{Pr}, z^{+} : \mathsf{Pr}, x^{-} : \mathsf{Pr} \vdash z : \mathsf{Pr}}{f^{+} : \mathsf{Pr}^{-} \to \mathsf{Pr}, z^{-} : \mathsf{Pr}, x^{+} : \mathsf{Pr} \vdash \neg z : \mathsf{Pr}}}{f^{+} : \mathsf{Pr}^{-} \to \mathsf{Pr}, z^{-} : \mathsf{Pr}, x^{+} : \mathsf{Pr} \vdash f(\neg x) \vee \neg z : \mathsf{Pr}^{-} \to \mathsf{Pr}}}{f^{+} : \mathsf{Pr}^{-} \to \mathsf{Pr}, z^{-} : \mathsf{Pr} \vdash \mu(x : \mathsf{Pr}).f(\neg x) \vee \neg z : \mathsf{Pr}^{-} \to \mathsf{Pr}}}{f^{+} : \mathsf{Pr}^{-} \to \mathsf{Pr} \vdash \lambda(z^{-} : \mathsf{Pr}).\mu(x : \mathsf{Pr}).f(\neg x) \vee \neg z : \mathsf{Pr}^{-} \to \mathsf{Pr}}}{\emptyset \vdash \mu(f : \mathsf{Pr}^{-} \to \mathsf{Pr}).\lambda(z^{-} : \mathsf{Pr}).\mu(x : \mathsf{Pr}).f(\neg x) \vee \neg z : \mathsf{Pr}^{-} \to \mathsf{Pr}}
$$

Figure 5.2: Example type derivation of a HFL formula.

Functions which do not occur under the scope of a fixpoint quantifier are not required to be monotonic. The expressivity of HFL would be limited if non-monotonic functions were forbidden in general.

In order to define the size of an HFL formula, we need the following.

**Definition 43** The Fischer-Ladner closure of an HFL formula $\varphi_0$ is the least set $\mathsf{Cl}(\varphi_0)$ that contains $\varphi_0$ and satisfies the following.

- If $\psi_1 \vee \psi_2 \in \mathsf{Cl}(\varphi_0)$ then $\{\psi_1, \psi_2\} \subseteq \mathsf{Cl}(\varphi_0)$.
- If $\neg(\psi_1 \vee \psi_2) \in \mathsf{Cl}(\varphi_0)$ then $\{\neg\psi_1, \neg\psi_2\} \subseteq \mathsf{Cl}(\varphi_0)$.
- If $\langle a \rangle \psi \in \mathsf{Cl}(\varphi_0)$ then $\psi \in \mathsf{Cl}(\varphi_0)$.
- If $\neg\langle a \rangle \psi \in \mathsf{Cl}(\varphi_0)$ then $\neg\psi \in \mathsf{Cl}(\varphi_0)$.
- If $\varphi\ \psi \in \mathsf{Cl}(\varphi_0)$ then $\{\varphi, \psi, \neg\psi\} \subseteq \mathsf{Cl}(\varphi_0)$.
- If $\neg(\varphi\ \psi) \in \mathsf{Cl}(\varphi_0)$ then $\{\neg\varphi, \psi, \neg\psi\} \subseteq \mathsf{Cl}(\varphi_0)$.
- If $\lambda X.\psi \in \mathsf{Cl}(\varphi_0)$ then $\psi \in \mathsf{Cl}(\varphi_0)$.
- If $\neg(\lambda X.\psi) \in \mathsf{Cl}(\varphi_0)$ then $\neg\psi \in \mathsf{Cl}(\varphi_0)$.
- If $\mu X.\psi \in \mathsf{Cl}(\varphi_0)$ then $\psi \in \mathsf{Cl}(\varphi_0)$.
- If $\neg(\mu X.\psi) \in \mathsf{Cl}(\varphi_0)$ then $\neg\psi[\neg X/X] \in \mathsf{Cl}(\varphi_0)$.
- If $\neg\neg\psi \in \mathsf{Cl}(\varphi_0)$ then $\psi \in \mathsf{Cl}(\varphi_0)$.
- If $\neg X \in \mathsf{Cl}(\varphi_0)$ then $X \in \mathsf{Cl}(\varphi_0)$.
- If $\neg q \in \mathsf{Cl}(\varphi_0)$ then $q \in \mathsf{Cl}(\varphi_0)$.

Note that the size of $\mathsf{Cl}(\varphi)$ is at most twice the length of $\varphi$. We define $|\varphi| := |\mathsf{Cl}(\varphi)|$ as the size of $\varphi$.

**Definition 44 (Type Semantics)** The semantics of a type w.r.t. $\mathcal{T}$ is inductively defined as a partially ordered set as follows:

$$\begin{aligned} [\![\mathsf{Pr}]\!]^{\mathcal{T}} &= (2^{\mathcal{S}}, \subseteq), \\ [\![\sigma^v \to \tau]\!]^{\mathcal{T}} &= \big(([\![\sigma]\!]^{\mathcal{T}})^v \to [\![\tau]\!]^{\mathcal{T}}, \sqsubseteq_{\sigma^v \to \tau}\big). \end{aligned}$$

where for two partially ordered sets $(\tau, \sqsubseteq_\tau)$ and $(\sigma, \sqsubseteq_\sigma)$, $\sqsubseteq_{\sigma^v \to \tau}$ denotes the partial order of all monotone functions ordered pointwise:

$$f \sqsubseteq_{\sigma^v \to \tau} g \quad \text{iff} \quad \text{for all } x \in [\![\sigma]\!]^{\mathcal{T}} : f\ x \sqsubseteq_\tau g\ x.$$

Moreover, complements in these partially ordered sets are denoted by $\bar{f}$ and defined on higher levels as $\bar{f}\ x = \overline{f\ x}$.
A positive variance leaves a partial order unchanged, $\bar{\tau}^+ = (\tau, \sqsubseteq_\tau)$, a negative variance turns it upside-down to make antitone functions look well-behaved, $\bar{\tau}^- = (\tau, \sqsupseteq_\tau)$, and a neutral variance flattens it, $\bar{\tau}^0 = (\tau, \sqsubseteq_\tau \cap \sqsupseteq_\tau)$.

**Lemma 30 ([VV04])** For all HFL types $\tau$ and finite LTS $\mathcal{T}$, $[\![\tau]\!]^{\mathcal{T}}$ is a complete lattice.

Although variances may destroy the lattice structure, they do only occur on the left of a typing arrow. The space of monotone functions from a partially ordered set to a complete lattice with pointwise ordering forms a complete lattice again.
By $\bot_\tau$ and $\top_\tau$ we denote the bottom and top elements of $[\![\tau]\!]^{\mathcal{T}}$.

**Definition 45 (HFL Semantics)** Let $\mathcal{T}$ be an LTS. An *environment* $\eta$ is a partial map on the variable set $\mathcal{V}$. For a context $\Gamma = X_1^{v_1} : \tau_1, \ldots, X_n^{v_n} : \tau_n$, we say that $\eta$ respects $\Gamma$, denoted by $\eta \models \Gamma$, if $\eta(X_i) \in [\![\tau_i]\!]^{\mathcal{T}}$ for $i \in \{1, \ldots, n\}$. We write $\eta[X \mapsto f]$ for the environment that maps $X$ to $f$ and otherwise agrees with $\eta$. If $\eta \models \Gamma$ and $f \in [\![\tau]\!]^{\mathcal{T}}$ then $\eta[X \mapsto f] \models \Gamma, X : \tau$, where $X \in \mathcal{V}$ is a variable that does not appear in $\Gamma$.
For any well-formed term $\varphi$ and environment $\eta \models \Gamma$, we define the semantics of $\varphi$ inductively to be an element of $[\![\tau]\!]^{\mathcal{T}}$ as follows:

$$\begin{aligned}
[\![\Gamma \vdash q : \mathsf{Pr}]\!]_\eta^{\mathcal{T}} &= \{s \in \mathcal{S} \mid q \in \ell(s)\}, \\
[\![\Gamma \vdash X : \tau]\!]_\eta^{\mathcal{T}} &= \eta(X), \\
[\![\Gamma \vdash \neg\varphi : \mathsf{Pr}]\!]_\eta^{\mathcal{T}} &= \mathcal{S} \setminus [\![\Gamma^- \vdash \varphi : \mathsf{Pr}]\!]_\eta^{\mathcal{T}}, \\
[\![\Gamma \vdash \neg\varphi : \sigma^v \to \tau]\!]_\eta^{\mathcal{T}} &= f \in [\![\sigma^v \to \tau]\!]^{\mathcal{T}} \text{ s.t. } \bar{f} = [\![\Gamma^- \vdash \varphi : \sigma^v \to \tau]\!]_\eta^{\mathcal{T}}, \\
[\![\Gamma \vdash \varphi \vee \psi : \mathsf{Pr}]\!]_\eta^{\mathcal{T}} &= [\![\Gamma \vdash \varphi : \mathsf{Pr}]\!]_\eta^{\mathcal{T}} \cup [\![\Gamma \vdash \psi : \mathsf{Pr}]\!]_\eta^{\mathcal{T}}, \\
[\![\Gamma \vdash \langle a \rangle\varphi : \mathsf{Pr}]\!]_\eta^{\mathcal{T}} &= \{s \in \mathcal{S} \mid s \xrightarrow{a} t \text{ for some } t \in [\![\Gamma \vdash \varphi : \mathsf{Pr}]\!]_\eta^{\mathcal{T}}\}, \\
[\![\Gamma \vdash \lambda(X^v : \sigma).\varphi : \sigma^v \to \tau]\!]_\eta^{\mathcal{T}} &= f \in [\![\sigma^v \to \tau]\!]^{\mathcal{T}} \text{ s.t. } \forall x \in [\![\sigma]\!]^{\mathcal{T}} \\
&\qquad f\ x = [\![\Gamma, X^v : \sigma \vdash \varphi : \tau]\!]_{\eta[X \mapsto x]}^{\mathcal{T}}, \\
[\![\Gamma \vdash \varphi\ \psi : \tau]\!]_\eta^{\mathcal{T}} &= [\![\Gamma \vdash \varphi : \sigma^v \to \tau]\!]_\eta^{\mathcal{T}}\ [\![\Gamma' \vdash \psi : \sigma]\!]_\eta^{\mathcal{T}}, \\
[\![\Gamma \vdash \mu(X : \tau)\varphi : \tau]\!]_\eta^{\mathcal{T}} &= \bigsqcap\{x \in [\![\tau]\!]^{\mathcal{T}} \mid [\![\Gamma, X^+ : \tau \vdash \varphi : \tau]\!]_{\eta[X \mapsto x]}^{\mathcal{T}} \sqsubseteq_\tau x\}.
\end{aligned}$$

In the clause for function application $(\varphi\ \psi)$ the context $\Gamma'$ is $\Gamma$ if $v \in \{+, 0\}$, and is $\Gamma^-$ if $v = -$.

**Definition 46 (Order, Arity)** We consider fragments of HFL that can be built using restricted types only. Note that because of right-associativity of the function arrow, every HFL type is isomorphic to a $\tau = \tau_1 \to \ldots \to \tau_m \to \mathsf{Pr}$ where $m \in \mathbb{N}$. Clearly, for $m = 0$ we simply have $\tau = \mathsf{Pr}$. We stratify types w.r.t. their *order*, i.e. the degree of using proper functions as arguments to other functions, as well as *maximal arity*, i.e. the number of arguments a function has. Order can be seen as depth, and maximal arity as the width of a type. Both are defined recursively as follows.

$$\begin{aligned} \mathit{ord}(\tau_1 \to \ldots \to \tau_m \to \mathsf{Pr}) &:= \max\{1 + \mathit{ord}(\tau_i) \mid i = 1, \ldots, m\}, \\ \mathit{mar}(\tau_1 \to \ldots \to \tau_m \to \mathsf{Pr}) &:= \max(\{m\} \cup \{\mathit{mar}(\tau_i) \mid i = 1, \ldots, m\}), \end{aligned}$$

where we assume $\max(\emptyset) = 0$. Now let, for $k \geq 1$ and $m \geq 1$,

$$\begin{aligned} \mathrm{HFL}^{k,m} &:= \{\varphi \in \mathrm{HFL} \mid \emptyset \vdash \varphi : \mathsf{Pr} \text{ using types } \tau \text{ with } \mathit{ord}(\tau) \leq k \text{ and } \mathit{mar}(\tau) \leq m \text{ only}\}, \\ \mathrm{HFL}^{k} &:= \bigcup_{m \in \mathbb{N}} \mathrm{HFL}^{k,m}. \end{aligned}$$

Note that no formula can have maximal type order $k > 0$ but maximal type arity $m = 0$. The combination $k = 0$ and $m > 0$ is also impossible. Hence, we define

$$\mathrm{HFL}^0 = \{\ \varphi \in \mathrm{HFL} \mid \emptyset \vdash \varphi : \mathsf{Pr} \text{ using types } \tau \text{ with } \mathit{ord}(\tau) = 0 \text{ only }\}.$$

We extend these measures to formulas in a straightforward way: $\mathit{ord}(\varphi) = k$ and $\mathit{mar}(\varphi) = m$ iff $k$ and $m$ are the least $k'$ and $m'$ s.t. $\varphi$ can be shown to have some type using types $\tau$ with $\mathit{ord}(\tau) \leq k'$ and $\mathit{mar}(\tau) \leq m'$ only.

**Definition 47 (Simultaneous Fixpoint)** When using least fixpoint quantifiers it is often beneficial to recall the Békič principle [Bék84] which states that a simultaneously defined least fixpoint of a monotone function is the same as a parametrised one. We will use this to allow formulas like

$$\varphi \ := \ \mu X_i . \begin{pmatrix} X_1 & . & \varphi_1(X_1, \ldots, X_n) \\ & \vdots & \\ X_n & . & \varphi_n(X_1, \ldots, X_n) \end{pmatrix}$$

in the syntax of HFL. This abbreviates

$$\varphi' \ := \ \mu X_i . \varphi_i(\mu X_1 . \varphi_1(X_1, \mu X_2 . \varphi_2(X_1, X_2, \ldots, X_i, \ldots), \ldots, X_i, \ldots), \mu X_2 \ldots, \ldots, X_i, \ldots).$$

Note that the size of $\varphi'$ can be exponentially bigger than the size of $\varphi$, and this even holds for the number of their subformulas. However, it is only exponential in $n$, not in $|\varphi|$: $|\varphi'| = O(|\varphi| \cdot 2^n)$.

## 5.2 Examples

**Example 15** HFL can express the non-regular (but context-free) property "on any path the number of $\mathsf{out}$'s seen at any time never exceeds the number of $\mathsf{in}$'s seen so far." Let

$$\varphi \; := \; \mu(X : \mathsf{Pr} \to \mathsf{Pr}).(\lambda(Z : \mathsf{Pr}).\langle \mathsf{out} \rangle Z \vee \langle \mathsf{in} \rangle (X \; (X \; Z))) \; \mathtt{tt}.$$

This formula is best understood by comparing it to the CFG $G = (\{X\}, \{\mathsf{in}, \mathsf{out}\}, P, X)$, where $P$ contains the rules

$$X \to \mathsf{out} \mid \mathsf{in} \, X \, X.$$

It generates the language $L$ of all words $w \in \{\mathsf{in}, \mathsf{out}\}^*\{\mathsf{out}\}$ s.t. $|w|_{\mathsf{in}} = |w|_{\mathsf{out}}$ and for all prefixes $v$ of $w$ we have: $|v|_{\mathsf{in}} \geq |v|_{\mathsf{out}}$ which are exactly the prefixes of buffer runs which are violating due to an underflow. Then $s \models \varphi$ iff there is a finite path through $\mathcal{T}$ starting in $s$ that is labeled with a word in $L$, and $\neg\varphi$ consequently describes the property mentioned above. In Section 5.4 we will see that in fact every path specification given by a context-free grammar can be checked by an $\mathrm{HFL}^{1,1}$ formula.

**Example 16** Another property that is easily seen not to be expressible by a finite tree automaton and, hence, not by a formula of $\mathcal{L}_\mu$ either is *bisimilarity to a word.* Note that a transition system $\mathcal{T}$ with starting state $s$ is not bisimilar to a linear word model iff there are two distinct actions $a$ and $b$ s.t. there are two (not necessarily distinct) states $t_1$ and $t_2$ at the same distance from $s$ s.t. $t_1 \xrightarrow{a} t_1'$ and $t_2 \xrightarrow{b} t_2'$ for some $t_1', t_2'$. This is expressed by the HFL formula

$$\neg\Big( \bigvee_{a \neq b} \big(\mu(F : \mathsf{Pr} \to \mathsf{Pr} \to \mathsf{Pr}).\lambda(X : \mathsf{Pr}).\lambda(Y : \mathsf{Pr}).(X \wedge Y) \vee (F \; \langle - \rangle X \; \langle - \rangle Y)\big) \; \langle a \rangle \mathtt{tt} \; \langle b \rangle \mathtt{tt} \Big).$$

This formula is best understood by regarding the least fixpoint definition $F$ as a functional program. It takes two arguments $X$ and $Y$ and checks whether both hold now or calls itself recursively with the arguments being checked in two (possibly different) successors of the state that it is evaluated in.

Note that here, bisimulation does not consider the labels of states but only the actions along transitions. It is not hard to change the formula accordingly to incorporate state labels as well.

**Example 17** Let $\mathbf{2}^n_0 := n$ and $\mathbf{2}^n_{m+1} := 2^{\mathbf{2}^n_m}$. For any $m \in \mathbb{N}$, there is a short HFL formula $\varphi_m$ (linear in $m$) expressing the fact that there is a maximal path of length $\mathbf{2}^1_m$ (number of states on this path) through a transition system. It can be constructed using a typed version of the Church numeral 2. Let $\tau_0 = \mathsf{Pr}$ and $\tau_{i+1} = \tau_i \rightarrow \tau_i$. For $i \geq 1$ define $\psi_i$ of type $\tau_{i+1}$ as $\lambda(F : \tau_i).\lambda(X : \tau_{i-1}).F\ (F\ X)$. Then

$$\varphi_m := \psi_m\ \psi_{m-1}\ \ldots\ \psi_1\ \big(\lambda(X : \mathsf{Pr}).\langle - \rangle X\big)\ [-]\mathtt{ff}\ .$$

Note that for any $m \in \mathbb{N}$, $\varphi_m$ is of size linear in $m$. This indicates that HFL is able to express computations of Turing Machines of arbitrary elementary complexity which has been shown in [ALS07].

## 5.3 Properties

**Theorem 63 (Finite Model Property Absence)** $\mathrm{HFL}^1$ does not exhibit the finite model property.

Proof Like for CTL[$\mathcal{L}$] , this follows from Thm. 28 in which a PDL[VPL] formula serves as witness for the absence of the finite model property. The formula can by Thm. 67 be translated into an equivalent $\mathrm{HFL}^1$ formula. Since both formulas are required to hold in exactly the same models, the absence of the finite model property for $\mathrm{HFL}^1$ follows. □

**Theorem 64 ([VV04])** HFL is bisimulation-invariant and therefore has the tree model property.

**Theorem 65 ([VV04])** HFL is undecidable.

## 5.4 Expressivity

HFL is clearly a much closer relative of $\mathcal{L}_\mu$ than the other logics under consideration here. All of them share a common propositional base but parametric CTL and PDL achieve non-regular expressive power by rather different means than HFL: the former two by a language plug-in mechanism which directly makes use of the expressive power contained within the language parameter, the latter with help of logic-inherent machinery, namely extremal fixpoints on higher-order functions.

HFL and its precursor FLC are merely generalisations of $\mathcal{L}_\mu$, while the relationship of $\mathcal{L}_\mu$ with parametric PDL and CTL is of a mutually non-inclusive form as has been proved in previous chapters.

**Theorem 66 ([MO99],[VV04],[ALS07])**

$$\mathcal{L}_\mu \equiv \text{HFL}^{0,0} \lneq \text{FLC} \equiv \text{HFL}^{1,1} \leq \text{HFL}^1 \lneq \text{HFL}^2 \lneq \text{HFL}^3 \ldots \lneq \text{HFL}.$$

Proof Note that $\mathcal{L}_\mu$ is a syntactical fragment of HFL and that every subformula of a $\mathcal{L}_\mu$ formula has type rank 0 in HFL. On the other hand, any $\text{HFL}^{0,0}$ formula cannot contain a subformula of type rank $\geq 1$, i.e. no $\lambda$-expressions (and hence no function applications) or fixpoint formulas other than of type rank 0. But deleting these two clauses from the definition of HFL's syntax yields exactly the syntax of $\mathcal{L}_\mu$. It is easy to see that the $\text{HFL}^{0,0}$ semantics coincide with the semantics of $\mathcal{L}_\mu$.
The result that $\mathcal{L}_\mu \lneq \text{FLC}$ originates from [MO99]. FLC can express simulations of context-free processes which $\mathcal{L}_\mu$ cannot.
That $\text{FLC} \equiv \text{HFL}^{1,1}$ is immediately seen by comparing the resulting semantics of this HFL restriction with FLC. The fact that $\text{FLC} \lneq \text{HFL}$ has been observed by [VV04].
Finally, the result that the expressive power increases in the hierarchy $\text{HFL}^k \lneq \text{HFL}^{k+1}$ for all $k \in \mathbb{N}$ is a corollary of the $k$EXPTIME-completeness result in Thm. 68 for model checking $\text{HFL}^k$.
For $\text{HFL}^0$, we have already shown that it is strictly lesser expressive than $\text{HFL}^1$, because $\text{HFL}^0 \equiv \mathcal{L}_\mu \lneq \text{FLC} \equiv \text{HFL}^{1,1} \leq \text{HFL}^1$. Now, assume $\varphi \in \text{HFL}^{k+1}$ for some $k \geq 1$ s.t. model checking $\varphi$ over some LTS is $(k+1)$EXPTIME-hard. But then there is no formula in $\text{HFL}^k$ which corresponds to $\varphi$, because model checking $\text{HFL}^k$ is in $k$EXPTIME and $k\text{EXPTIME} \subsetneq (k+1)\text{EXPTIME}$ for all $k \in \mathbb{N}$. □

The conceptual unrelatedness of HFL and the language parametric logics makes a comparison difficult. Clearly, the modal and temporal formulas of parametric PDL and CTL must in fact be expressible as fixpoints in a "unifying" logic in the same manner as for instance $\mathcal{L}_\mu$ or MSO serve as backbones for regular PDL and CTL. But it is not clear whether there exists any suitable candidate capable of simulating the languages used in the modalities and expressing the corresponding fixpoint statements. To our knowledge there is no work in the literature which systematically deals with the correspondence between the expressive power of logics and formal languages *above* the regular sphere.
We are however able to embed PDL[CFG] into HFL. The idea is very similar to the embedding of PDL[CFG] into FLC in [LS06].

**Theorem 67**

$$\text{PDL[CFG]} \lneq \text{HFL}^{1,1}.$$

Proof If $\leq$ holds, strictness is a consequence of $\mathcal{L}_\mu \lneq \text{HFL}^{1,1}$ and the fact that fairness is inexpressible in PDL[$\mathcal{L}$] (independently of $\mathcal{L}$) but expressible in $\mathcal{L}_\mu$. In order to show $\leq$, consider the following translation $\mathsf{tr} : \text{PDL[CFG]} \to \text{HFL}^{1,1}$ with

$$\begin{array}{rcl} \mathsf{tr}(q) & = & q, \\ \mathsf{tr}(\neg\varphi) & = & \neg\mathsf{tr}(\varphi), \\ \mathsf{tr}(\varphi \vee \psi) & = & \mathsf{tr}(\varphi) \vee \mathsf{tr}(\psi), \\ \mathsf{tr}(\langle G \rangle \psi) & = & \mathsf{tr}'(\langle G \rangle)\ \mathsf{tr}(\psi), \end{array}$$

where $\mathsf{tr}'(\langle G \rangle)$ is defined for a CFG $G$ as follows. Let $G = (N, \Sigma, P, S)$. Define the righthand sides of production rules w.r.t. a $X \in N$ as $\mathsf{rhs}(X) = \{\alpha \in (N \cup \Sigma)^* \mid X \to \alpha \in P\}$.

$$\mathsf{tr}'(\langle G \rangle) \;=\; \mu(S : \mathsf{Pr} \to \mathsf{Pr}). \begin{pmatrix} X_1 & . & \lambda(Z_1 : \mathsf{Pr}). \bigvee\limits_{\alpha \in \mathsf{rhs}(X_1)} \widehat{\alpha}\ Z_1 \\ & \vdots & \\ X_n & . & \lambda(Z_n : \mathsf{Pr}). \bigvee\limits_{\alpha \in \mathsf{rhs}(X_n)} \widehat{\alpha}\ Z_n \end{pmatrix}.$$

where $\bigcup\limits_{i=1}^{n} X_i = N$ and

$$\widehat{\alpha} \;=\; \begin{cases} \langle a \rangle \widehat{\beta} & , \text{if } \alpha = a\beta. \\ \mathsf{tr}(\psi) \wedge \widehat{\beta} & , \text{if } \alpha = \psi?\beta. \\ X_i\ \widehat{\beta} & , \text{if } \alpha = X_i\beta. \\ \epsilon & , \text{if } \alpha = \epsilon. \end{cases}$$

for some $a \in \Sigma$, $\varphi \in$ PDL[CFG] and $\epsilon$ denoting a blank.

Let $\mathcal{T} = (\mathcal{S}, \to, \ell)$ be an LTS. We will now show that for all $s \in \mathcal{S}$ and $\varphi \in$ PDL[CFG], we have

$$s \models \varphi \text{ iff } s \models \mathsf{tr}(\varphi).$$

We show this by induction on the structure of $\varphi$. The propositional cases are entirely trivial in both directions and so it remains to show that $s \models \langle G \rangle \psi$ iff $s \models \mathsf{tr}'(\langle G \rangle)\ \mathsf{tr}(\psi)$.

It is well known that $\mathcal{L}(G)$ is the simultaneously defined least fixpoint of an equation system given by the grammar rules and projected onto the starting symbol $S$. The function $\mathsf{tr}'(\langle G \rangle)$ represents exactly this equation system but restricts derivable words in $G$ to paths in $\mathcal{T}$.

Since $\mathsf{tr}'(\langle G\rangle)$ is applied to the set of states which satisfy $\mathsf{tr}(\psi)$, it is additionally required that these paths end in such a state. This establishes the claim. □

Fig. 5.3 summarises all expressivity results obtained in previous chapters.

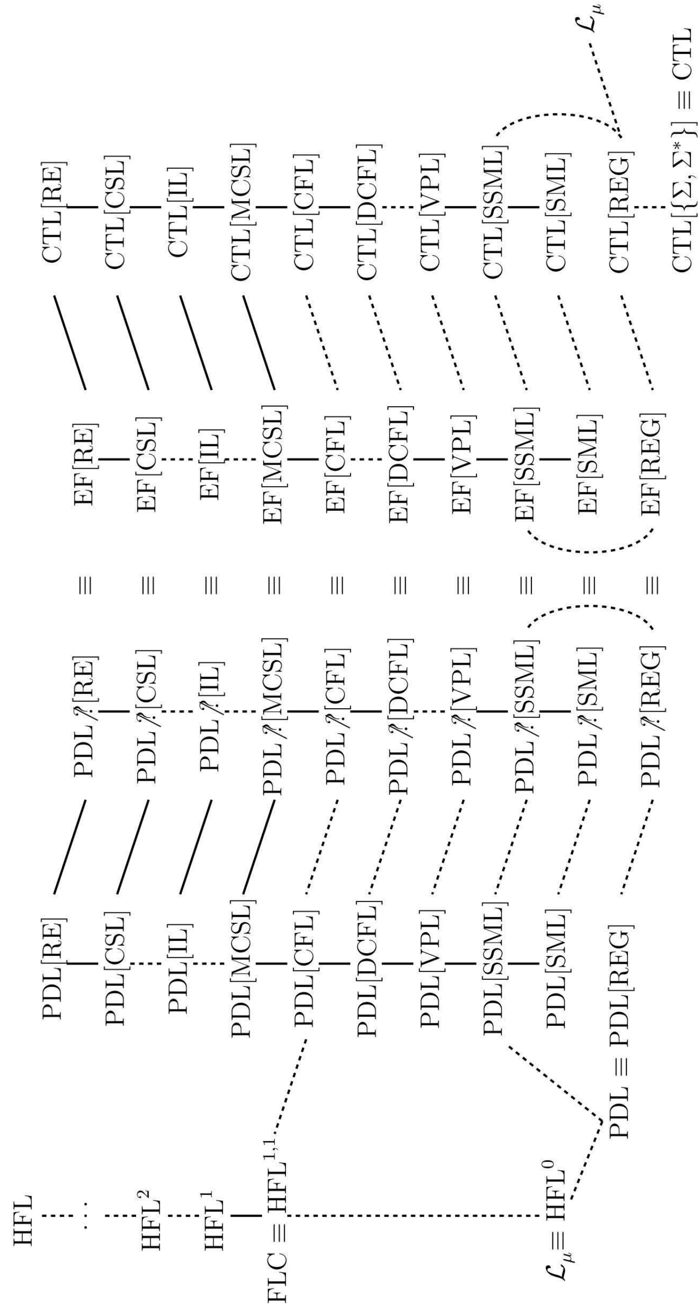



Figure 5.3: Expressive power of PDL[$\mathcal{L}$], CTL[$\mathcal{L}$] and HFL.

## 5.5 Model Checking

In [ALS07], a game-based model checking procedure is being introduced to prove a $k$-EXPTIME upper bound for HFL$^k$. It is however likewise possible to extend standard fixpoint approximation schemes (as known from $\mathcal{L}_\mu$ model checkers) to the higher order case. While the game-based procedure is hardly feasible in practice, we may use an optimisation technique from static analysis called neededness analysis (cf. [Jør94]) inside the fixpoint approximation in order to obtain an algorithm which despite the high complexity has a chance to be useable at least for formulas of lower-order HFL. We present here a generalisation of the technique described in [AL07], where only the first-order case is treated.
For the following, note that because of right-associativity of the function arrow, every HFL-type is isomorphic to a $\tau = \tau_1 \to \ldots \to \tau_m \to \mathsf{Pr}$ for a $m \in \mathbb{N}$.

**Definition 48 (HFL-Fixpoint Approximants)** Let $\sigma x.\varphi$ be an HFL term of type $\tau = \tau_1 \to \ldots \to \mathsf{Pr}$, where $\sigma \in \{\mu, \nu\}$. We define finite approximants of this formula for all $i \in \mathbb{N}$ as follows:

$$\sigma^0 x.\varphi \quad = \quad \lambda(Z_1 : \tau_1).\ldots.\lambda(Z_k : \tau_k).\begin{cases} \mathtt{ff}, & \text{if } \sigma = \mu \\ \mathtt{tt}, & \text{otherwise} \end{cases}$$

$$\sigma^{i+1} x.\varphi \quad = \quad \varphi[\sigma^i x.\varphi / x].$$

**Lemma 31** Let $\mu x.\varphi$ be an HFL term of type $\tau = \tau_1 \to \ldots \to \tau_{k+1}$ and let $h$ be defined as $h(\llbracket \tau_{k+1} \rrbracket^{\mathcal{T}})$. Then for any finite LTS $\mathcal{T}$ we have $\llbracket \mu^h x.\varphi \rrbracket_\eta^{\mathcal{T}} = \llbracket \mu x.\varphi \rrbracket_\eta^{\mathcal{T}}$ for any environment $\eta$.

Proof Note that the underlying LTS is finite. According to Lemma 30, the HFL type semantics forms a complete lattice. Because the types are all finite on finite models, the lattice has also finite height. On the other hand, the type system guarantees that HFL fixpoint terms are exclusively defined on monotone functions. As a consequence, the fixpoint approximation goes through a sequence of lattice elements of which each is greater or equal to the former w.r.t. $\sqsubseteq$. Since this sequence has maximally $h$ many different elements, the claim follows. $\square$

**Lemma 32** [ALS07] For all HFL types $\tau$ and all LTS $\mathcal{T}$ with $n$ states we have:

$$h(\tau) \leq (n+1)(\mathbf{2}_{ord(\tau)}^{n(mar(\tau)+ord(\tau)-1)^{ord(\tau)-1}})^{mar(\tau)}.$$

**Theorem 68 (ALS07)** For any $k, m \geq 1$ the HFL$^{k,m}$ model checking problem is $k$EXPTIME-complete.

### 5.5.1 A Standard Fixpoint-Approximation Algorithm

Consider a model checking algorithm for HFL formulas in which a subroutine FPapprox computes fixpoint approximants as given in Def. 48. Since the least and greatest fixpoint cases are entirely dual, we restrict our attention w.l.o.g. to least fixpoint formulas. FPapprox takes a (not necessarily closed) HFL term $\mu(x : \tau_0 \to \ldots \to \mathsf{Pr}).\varphi$ and an environment $\eta$ which maps free variables to values of the right type and tabulates the fixpoint approximants as shown in the table below, where $a_i^0, \ldots, a_i^{m_i}$ denotes an arbitrary enumeration of the elements in $[\![\tau_i]\!]$ and $h$ its height respectively.
The table is to be read as follows: the rows starting with $arg_i$ entries contain all possible combinations of arguments of type $\tau_0 \to \ldots \to \tau_k$. The rows underneath list the semantics of the fixpoint approximants given as a mapping from each sequence of arguments in the same column to the values in this column as they would successively be computed line-by-line in the routine FPapprox. It is clear that FPapprox could stop any time before the $h$-th approximant is reached, if the last and current approximant were identical in all columns and hence the fixpoint was established earlier.

| $arg_0 : \tau_0$ | $a_0^0$ | $a_0^1$ | $\ldots$ | $a_0^{m_0}$ | $a_0^0$ | $a_0^1$ | $\ldots$ | $a_0^{m_0}$ | $\ldots$ | $a_0^0$ | $a_0^1$ | $\ldots$ | $a_0^{m_0}$ | $\ldots$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $arg_1 : \tau_1$ | $a_1^0$ | $a_1^0$ | $\ldots$ | $a_1^0$ | $a_1^1$ | $a_1^1$ | $\ldots$ | $a_1^1$ | $\ldots$ | $a_1^{m_1}$ | $a_1^{m_1}$ | $\ldots$ | $a_1^{m_1}$ | $\ldots$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| $arg_k : \tau_k$ | $a_k^0$ | $a_k^0$ | $\ldots$ | $a_k^0$ | $a_k^0$ | $a_k^0$ | $\ldots$ | $a_k^0$ | $\ldots$ | $a_k^0$ | $a_k^0$ | $\ldots$ | $a_k^0$ | $\ldots$ |
| $[\![\mu^0 x.\varphi]\!]$ | $\emptyset$ | $\emptyset$ | $\ldots$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\ldots$ | $\emptyset$ | $\ldots$ | $\emptyset$ | $\emptyset$ | $\ldots$ | $\emptyset$ | $\ldots$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| $[\![\mu^h x.\varphi]\!]$ | $v_0$ | $v_1$ | $\ldots$ | $\ldots$ | $\ldots$ | $\ldots$ | $\ldots$ | $\ldots$ | $\ldots$ | $\ldots$ | $\ldots$ | $\ldots$ | $\ldots$ | $\ldots$ |

This is so far a naive extension of standard fixpoint computation techniques as known for instance from $\mathcal{L}_\mu$ model checking algorithms. Note that an HFL-equivalent to an $\mathcal{L}_\mu$-formula needs zero arguments and hence uses only a single column in the above table.
The following improvements to this procedure stem from the observation that every HFL formula is of type $\mathsf{Pr}$ and hence the semantics of higher-order terms (i.e. functions) has to be

broken down by function application in order to make for a formula. This implies that the function semantics is not necessarily needed as a whole, but just at the specific arguments to which it is applied. It does however *not* imply that the computation of the value at a single argument can be performed independently of values at other arguments, simply because values at different arguments might be needed in the presence of recursive function application, as may be the case in fixpoint formulas. The next section develops this idea to the extent that fixpoint approximants are computed as partial functions, where the defined domain is extended on demand, driven by value neededness during computation.

### 5.5.2 A Model Checker Using Neededness Analysis

Consider the recursive procedure MC-HFL as given in Fig. 5.4: it takes as input a typed HFL term $\varphi$, a (possibly empty) list of arguments $[f_1, ..., f_k]$ and an environment function $\eta$ which maps free variables to values of the correct type.

We assume that at the initial call of MC-HFL, $\varphi$ is a well-formed HFL formula of type $\mathsf{Pr}$, the argument list is empty and $\eta$ is entirely undefined for all arguments. The LTS $\mathcal{T} = (\mathcal{S}, \rightarrow, \ell)$ over which $\varphi$ is to be model checked is assumed to be available globally. After termination, MC-HFL is supposed to return the set of LTS states in which $\varphi$ holds. Note that the formulas and terms occurring in the case distinctions reflect the full expressive power of HFL. We omit type annotations where the type is obvious from the definition or irrelevant for the computation. Variances are omitted as well, since the formulas are assumed to be well-formed.

The propositional and modal formulas are handled in a standard way. The difficulties are posed by fixpoint formulas. The idea is in principle that the algorithm maintains a table similar to the one described in the previous section for the standard fixpoint approximation scheme, except that it is empty initially and filled with arguments and values as needed. This means that HFL fixpoint formulas are evaluated to functions which are stored as tables.

**Notation:** A partial function $f : X \rightarrow Y$ is assumed to map any $x \in X$ either to $f(x)$ if $f$ is defined at $x$ and to `undef` otherwise. Furthermore, $dom(f)$ is defined as the function which maps $f$ to the set of arguments on which it is defined, i.e. $dom(f) = \{x \in X \mid f(x) \neq \texttt{undef}\}$. The expression $f\{z \mapsto v\}$ denotes the (partial) function $f'$ which agrees with $f$ on all arguments $x \in X$, except possibly for $z$, where its value is $v$, i.e.

```
MC-HFL(φ, [f_1, ..., f_k], η) =
  case φ of
    q                    : ℓ(q)
    ¬φ                   : S \ MC-HFL(φ, [], η)
    ψ_1 ∨ ψ_2            : MC-HFL(ψ_1, [], η) ∪ MC-HFL(ψ_2, [], η)
    ⟨a⟩ψ                 : {s ∈ S | ∃t ∈ MC-HFL(ψ, [], η) s.t. s --a--> t}
    X                    : return η(X)([f_1, ..., f_k])
    x : σ → τ            : if η(x)([f_1, ..., f_k]) = undef
                           then let v := if fp(x) = μ then ⊥_τ else ⊤_τ
                                    η(x) := η(x){[f_1, ..., f_k] ↦ v}
                           return η(x)([f_1, ..., f_k])
    λ(X : τ).ψ           : if [f_1, ..., f_k] = []
                           then return λ(f : τ).MC-HFL(ψ, [], η{X ↦ f})
                           else return MC-HFL(ψ, [f_2, ..., f_k], η{X ↦ f_1})
    ψ_1 ψ_2              : MC-HFL(ψ_1, [MC-HFL(ψ_2, [], η), f_1, ..., f_k], η)
    σ(x : τ_1 → ... → Pr).ψ : if [f_1, ..., f_k] = [] and type(x) ≠ Pr
                           then return λ(g_1 : τ_1) ... λ(g_k : τ_k).
                           MC-HFL(σ(x : τ_1 → ... → τ_{k+1}).ψ, [g_1, ..., g_k], η)
                           else let v := if fp(x) = μ then ⊥_{τ_n} else ⊤_{τ_n}
                                    η(x) := {[f_1, ..., f_k] ↦ v}
                                    repeat
                                      f := η(x)
                                      for all [f'_1, ..., f'_k] ∈ dom(η(x))
                                        η(x) := η(x){[f'_1, ..., f'_k] ↦ MC-HFL(ψ, [f'_1, ..., f'_k], η)}
                                    until f = η(x)
                           return η(x)([f_1, ..., f_k])
```

Figure 5.4: A model checking algorithm for HFL.

$f'(x) = f(x)$, if $x \neq z$ and $v$ otherwise. Note that the data structures which represent functions have to be available globally.

$\bot_\tau$ and $\top_\tau$ denote the bottom and top elements of type $\tau$, and [] is the empty list. $\lambda$-bound variables are distinguished from $\mu$- and $\nu$-bound variables by upper- and lower-case letters respectively.

**Type Safety:** The return type of algorithm MC-HFL after termination is the data type which represents $\mathsf{Pr}$. However, in several cases e.g. the subcase of $\lambda$-abstraction, where $[f_1, ..., f_k] = []$, an anonymous function (here $\lambda(f : \tau).\text{MC-HFL}(\psi, [], \eta\{X \mapsto f\})$) is returned for the purpose of postponing the current computation to a later moment (see next paragraph for details). The returned $\lambda$-term is not to be confused with a HFL $\lambda$-expression, but should be interpreted as an anonymous function in the implementing programming language. It has to be read as a lazy evaluation of $\text{MC-HFL}(\psi, [], \eta)$ which will only be evaluated in the context of a later function application or maybe even not at all. If it is never touched again and remains unevaluated, this means that it only occurred as an argument in a higher-typed function.
Note however that in a real implementation it has to be type-consistent with the "evaluated" return types of MC-HFL. This problem could for instance be solved by using an abstract data type encapsulating both evaluated and unevaluated return types adequatly. Our algorithm transcipt is a concession to presentation clarity and therefore omits this level of detail.

**Step-by-Step Explanation:**

- (Propositional and modal formulas) The first four cases are concerned with propositional and modal formulas of primitive type. Propositions $q$ are immediately evaluated according to the labels in the LTS, the rest result in recursive evaluations of subformulas w.r.t. the demands of the operators $\neg, \vee, \langle a \rangle$.

- (Function application, $\lambda$-abstractions and $\lambda$-bound variables) Any occurring $\lambda$-bound variable $X$ is assumed to have been bound earlier and its value stored in the environment $\eta$. Its bound value $\eta(X)$ is returned. Function application $\psi_1 \ \psi_2$ is treated by recursive evaluation of $\psi_2$ which is put into the argument list of the recursive MC-HFL-call of $\psi_1$. If in case of a formula $\lambda(X : \tau).\psi$, the list of arguments $[f_1, ..., f_k]$ is empty, its denotation is currently not needed and its computation postponed until arguments are provided. This is expressed by the return value $\lambda(f : \tau).\text{MC-HFL}(\psi, [], \eta\{X \mapsto f\})$, not to be confused with a HFL $\lambda$-expression, but interpreted as an anonymous function in the implementing programming language (see previous paragraph for details). If the list of arguments is not empty, then $X$ is bound to the first argument $f_1$ provided and MC-HFL is called recursively on the body of the expression.

- (Fixpoint computation and $\mu, \nu$-bound variables) The fixpoint computation is localised and performed on needed values only: if no arguments are provided and $x$ is not of primitive type, its denotation is currently not needed and the fixpoint approximation postponed. Otherwise, the first fixpoint approximant is initialised according to least or greatest fixpoint type with $\bot_{\tau_n}$ or $\top_{\tau_n}$, so far realised as the partial function $x$ which is only defined at $[f_1, ..., f_k]$. The `repeat`-loop updates $x$ in the line $\eta(x) := \eta(x)\{[f'_1, ..., f'_k] \mapsto \text{MC-HFL}(\psi, [f'_1, ..., f'_k], \eta)\}$ and computes the approximants on the currently defined domain of $x$. It stops on two conditions: no fresh arguments enter $dom(\eta(x))$ and the approximation stabilises. Then the computed value of $x$ at the original arguments $[f_1, ..., f_k]$ is returned.

  The case of $\mu$- and $\nu$-bound variables $x$ is similar to $\lambda$-bound variables. Either $x$ is defined at the arguments in which case its value is returned, or it is undefined. This is the case, where a fresh argument enters $dom(\eta(x))$ which is initialised with $\bot_{\tau_n}$ or $\top_{\tau_n}$ according to the fixpoint type.

The algorithm MC-HFL improves a naive bottom-up model checker in two ways: by lazy evaluation of functions without arguments and by demand-driven fixpoint computation. We demonstrate both features by an example.

**Example 18** Consider the formula

$$\Big(\lambda(F : (\mathsf{Pr} \to \mathsf{Pr}) \to \mathsf{Pr}).F\ (\lambda(X : \mathsf{Pr} \to \mathsf{Pr}).X)\Big)\ \Big(\mu(y : (\mathsf{Pr} \to \mathsf{Pr}) \to \mathsf{Pr}).\lambda(G : \mathsf{Pr} \to \mathsf{Pr}).y\ G\Big).$$

The formula does not express anything particularly meaningful but serves our purpose. In fact it is also independent of the transition system, because its semantics is `ff` on every model. So let $\mathcal{T}$ be an arbitrary LTS in the following.

The basic structure is that of a function application: the least fixpoint function on the right hand side (representing the function which maps every function of type $\mathsf{Pr} \to \mathsf{Pr}$ to the least set of states on which its $n$-fold application stabilises) is plugged into the function on the left which takes any function of right type and applies it to the identity function.

After $\beta$-reduction, the expression is easily seen to boil down to an application of the fixpoint function on the identity function. However, this is a valid HFL formula and demonstrates the usefulness of lazy evaluation.

For reasons of readability, we omit type annotations in the following and do only hint at the development of the environment $\eta$ (as side-effects) between calls of MC-HFL. Note that $\eta$ contains no bindings initially.

$$\text{MC-HFL}\Big(\big(\lambda F.F\ (\lambda X.X)\big)\ \big(\mu y.\lambda G.y\ G\big), []\Big) = \tag{1}$$

$$\text{MC-HFL}\big(\lambda F.F\ (\lambda X.X), [\text{MC-HFL}(\mu y.\lambda G.y\ G, [])]\big) = \tag{2}$$

$$\text{MC-HFL}\big(\lambda F.F\ (\lambda X.X), [\lambda f.\text{MC-HFL}(\mu y.\lambda G.y\ G, [f])]\big) = \ldots \tag{3}$$

So far, the algorithm has processed the argument of the function $\lambda F.F(\lambda X.X)$ which is a least fixpoint of a second-order function for which no argument has been provided. This leads to a delay of the actual computation of the fixpoint in line 3 where just the anonymous function $\lambda f.\text{MC-HFL}(\mu y.\lambda G.y\ G, [f])$ is returned which passes on its argument to the fixpoint function: it is lazily evaluated and merely serves as a symbolic placeholder.

$$\ldots =$$

$$\text{MC-HFL}\big(F\ (\lambda X.X), []\big) = \tag{4}$$

$$\eta := \eta\{F \mapsto \lambda f.\text{MC-HFL}(\mu y.\lambda G.y\ G, [f])\} \tag{5}$$

$$\text{MC-HFL}\big(F, [\text{MC-HFL}(\lambda X.X)]\big) = \tag{6}$$

$$\text{MC-HFL}\big(F, [\lambda g.\text{MC-HFL}(X, [])]\big) = \tag{7}$$

$$\eta := \eta\{X \mapsto g)\} \ldots \tag{8}$$

In line $4-5$, the variable $F$ is bound to the fixpoint function. The following lines demonstrate yet another lazy evaluation, this time of the identity function which has no arguments either.

$$\ldots =$$

$$\text{MC-HFL}\big(\lambda f.\text{MC-HFL}(\mu y.\lambda G.y\ G, [f]), [\lambda g.\text{MC-HFL}(X, [])]\big) = \tag{9}$$

$$\text{MC-HFL}\big(\text{MC-HFL}(\mu y.\lambda G.y\ G, [\lambda g.\text{MC-HFL}(X, [])]), []\big). \tag{10}$$

Lines $9-10$ perform a $\beta$-reduction on the level of the programming language (as opposed to the level of HFL expressions) and show the whole benefit of the evaluation delay: instead of computing the whole function, we now just have to compute the function at an argument which was formerly hidden in the formula structure. The rule of thumb here is simply that every function with an argument is computed immediately but restricted to that argument while the computation of functions without arguments is delayed. This is justified by the observation that every well-formed HFL formula sooner or later breaks down any higher-order construct to primitive type $\mathsf{Pr}$. We exclude the computation of the fixpoint here since the next example will demonstrate this improvement on a more suitable function. We just state as a fact here that $(\mu y.\lambda G.y\ G)(\lambda g.\text{MC-HFL}(X, [])) = \emptyset$ (the identity function stabilises on every argument after one self-application and the least argument of primitive type on which this happens is $\emptyset$).

| $X$ | $\{3\}$ | $\{2,3\}$ | $\{1,2,3\}$ |
|---|---|---|---|
| 0 | $\emptyset$ | | |
| 1 | $\{3\}$ | $\emptyset$ | |
| 2 | $\{3\}$ | $\{2,3\}$ | $\emptyset$ |
| 3 | $\{2,3\}$ | $\{2,3\}$ | $\{1,2,3\}$ |
| 4 | $\{2,3\}$ | $\{1,2,3\}$ | $\{1,2,3\}$ |
| 5 | $\{1,2,3\}$ | $\{1,2,3\}$ | $\{1,2,3\}$ |
| 6 | $\{1,2,3\}$ | $\{1,2,3\}$ | $\{1,2,3\}$ |

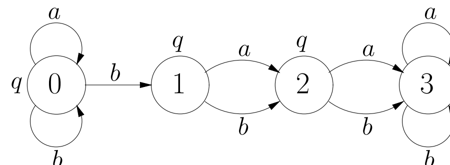



Figure 5.5: Algorithm MC-HFL running on a simple example.

**Example 19** For the demonstration of fixpoint computation on demand, consider the formula

$$\big(\mu(x : \mathsf{Pr} \to \mathsf{Pr}).\lambda(Z : \mathsf{Pr}).Z \vee \bigvee_{a\in\Sigma} x\ [a]Z\big)\ \neg q$$

and the transition system shown on the right side in Fig. 5.5. Intuitively, $\varphi$ asserts that there is a sequence of actions s.t. all paths under that sequence lead to a state not satisfying $q$. States $1, 2, 3$ satisfy this property, state 0 does not. However, the meaning of this formula is irrelevant for the understanding of how it is evaluated by algorithm MC.

The table on the left of Fig. 5.5 shows the successive calculation of the semantics of the fixpoint formula. Although only two rows need to be stored in each iteration step – the current one and the last one for comparison – we depict all stages in this example for the reader to be be able to follow this step-by-step.

At the beginning, the formula $\neg q$ is evaluated to $\{3\}$. This forms the initial argument in the table. It is to be read as follows: time proceeds line by line from left to right. Each row below the arguments contains a snapshot of the current state at the end of an iteration over the current domain. Note that in general fixpoint approximants cannot easily be read off the table since different columns may be at different stages of approximation. As computation proceeds, arguments are added to the list.

Row 6 then represents a partial function that agrees with the total function that is the semantics of the corresponding fixpoint formula. The return value is the one in the first column – the value of the fixpoint function applied to the original argument.

These improvements do of course not affect the worst-case complexity of the HFL model checking problem. Instead, they allow for better best- and average-case complexities which otherwise would just be the same as the worst-case complexity. In Section 5.5.4, we give empirical evidence that the improvements have significant influence on the performance of

the model checking algorithm and make it feasible in practice in the first place (of course only for lower-order fragments of HFL).

### 5.5.3 Soundness and Completeness

We will now prove that MC-HFL correctly computes the semantics of any well-formed formula of HFL. In order to do so, we need to relate the environment used in MC-HFL which maps variables to partially defined functions (which we will call shortly "partial environments") with the environment of HFL term semantics which contains only total functions (which we call "HFL environments").

**Definition 49** Let $f : \tau_0 \to \ldots \to \mathsf{Pr}$ be a partial function on HFL types. Define $\nabla(f)$ as the set of all total functions which agree with $f$ on all arguments on which $f$ is defined, i.e. $g \in \nabla(f)$ iff for all $x \in dom(f)$: $g(x) = f(x)$ and $g$ is total. We overload the $\nabla$-operator to be applicable also for partial environments $\eta$. Its meaning is that if $\eta(X) = f$ then for all $\eta' \in \nabla(\eta)$: $\eta'(X) \in \nabla(f)$.

**Theorem 69** For all transition systems $\mathcal{T}$, all partial environments $\eta$, HFL environments $\eta' \in \nabla(\eta)$ and all well-formed formulas $\varphi \in$ HFL we have: MC-HFL$(\varphi, [], \eta) = [\![\varphi]\!]^{\mathcal{T}}_{\eta'}$.

We cannot prove this theorem directly: the statement is too weak as an inductive invariant because of subformulas of type other than $\mathsf{Pr}$.
We will instead prove the following stronger statement, from which the above theorem follows immediately.

**Lemma 33** For all transition systems $\mathcal{T}$, all partial environments $\eta$, HFL environments $\eta' \in \nabla(\eta)$ all sequences of arguments $[f_1, \ldots, f_k]$ (consisting of valid HFL types) and all (not necessarily closed) well-formed terms $\varphi \in$ HFL we have:

$$\text{MC-HFL}(\varphi, [f_1, \ldots, f_k], \eta) = [\![\varphi]\!]^{\mathcal{T}}_{\eta'}([f_1, \ldots, f_k]).$$

Proof We show the claim by induction on the structure of the formula $\varphi$. Let $\varphi$ be a term, $\eta$ be a partial environment that maps any free variable in $\varphi$ to a (possibly partial) function and $f_i$ be a valid HFL type over a transition system $\mathcal{T}$ for all $1 \leq i \leq k$.

**The propositional and modal part.** The statement is immediately seen to be true for the case of $\varphi = q$ for some $q \in \mathcal{P}$. It also follows directly from the hypothesis in the cases $\varphi = \psi_1 \vee \psi_2$, $\varphi = \langle a \rangle \psi$ and $\varphi = \neg\psi$ . Note that in all these cases, $\psi, \psi_1$ and $\psi_2$ must have type $\mathsf{Pr}$. Hence, the argument list $[f_1, \ldots, f_k]$ must in fact be empty.

**The functional part.** Now consider $\varphi = X$, where $X$ is a $\lambda$-,$\mu$- or $\nu$-bound variable: the call of MC-HFL$(X, [f_1, \dots, f_k], \eta)$ returns in any case $\eta(X)([f_1, \dots, f_k])$ which agrees with $[\![X]\!]^{\mathcal{T}}_{\eta'}([f_1, \dots, f_k])$ by definition of the semantics and the definition of $\nabla$.
Now consider the case $\varphi = \lambda(X : \sigma).\psi$ of type $\sigma \to \tau$. Note that $\varphi$ cannot be of primitive type $\mathsf{Pr}$, i.e. it takes an argument.
We distinguish according to the two cases in MC-HFL, namely that

- an argument is provided in the list. Then MC-HFL$(\varphi, [f_1, \dots, f_k], \eta)$ evaluates to MC-HFL$(\psi, [f_2, \dots, f_k], \eta\{X \mapsto f_1\})$ which by I.H. is $[\![\psi]\!]^{\mathcal{T}}_{\eta\{X \mapsto f_1\}}([f_2, \dots, f_k])$. This is in turn equivalent to $[\![\lambda(X : \sigma).\psi]\!]^{\mathcal{T}}_{\eta'}([f_1, \dots, f_k])$ by a $\beta$-reduction in which $X$ is overridden in $\eta'$ and bound to $f_1$.

- no argument is provided. Then the call is MC-HFL$(\varphi, [], \eta)$ and the return value is $\lambda(y : \sigma)$.MC-HFL$(\psi, [], \eta\{X \mapsto y\})$, i.e. a function which for any argument $y$ of type $[\![\sigma]\!]^{\mathcal{T}}$ yields by I.H. the value $[\![\psi]\!]^{\mathcal{T}}_{\rho\{X \mapsto y\}}([])$ of type $\tau$. But this is exactly $[\![\varphi]\!]^{\mathcal{T}}_{\eta'}([])$. Note again, that $X$ is overridden in $\eta'$.

The case of function application $\varphi = \psi_1\ \psi_2$ is simple:
MC-HFL$(\psi_1\ \psi_2, [f_1, \dots, f_k], \eta) =$ MC-HFL$(\psi_1, [$MC-HFL$(\psi_2, [], \eta), f_1, \dots, f_k], \eta)$. By I.H., we have MC-HFL$(\psi_2, [], \eta) = [\![\psi_2]\!]^{\mathcal{T}}_{\eta'}([])$ and therefore MC-HFL$(\psi_1\psi_2, [f_1, \dots, f_k], \eta) =$ MC-HFL$(\psi_1, [[\![\psi_2]\!]^{\mathcal{T}}_{\eta'}, f_1, \dots, f_k], \eta)$ which by I.H. is $[\![\psi_1]\!]^{\mathcal{T}}_{\eta'}([[\![\psi_2]\!]^{\mathcal{T}}_{\eta'}, f_1, \dots, f_k])$.

The only cases posing difficulties are those of $\varphi = \sigma X.\psi$ for $\sigma \in \{\mu, \nu\}$. Here it is helpful to prove soundness (direction "$\subseteq$") and completeness (direction "$\supseteq$") separately. However, the soundness proof for the $\mu$-case is entirely analogous to the completeness proof of the $\nu$-case and vice-versa. Thus, we only present soundness and completeness of the $\mu$-case here.

**Soundness of the $\mu$-part.** Consider the following call of the model checking algorithm: MC-HFL$(\mu(x : \tau_1 \to \dots \tau_{k+1}).\psi, [f_1, \dots, f_k], \eta)$. Here we have to take into account that the environment may contain partially defined functions. Thus we have to prove the following statement:

$$\forall [f'_1, \dots, f'_k] \in dom(\eta(x)) : \eta(x)([f'_1, \dots, f'_k]) \sqsubseteq [\![\mu(x : \tau_1 \to \dots \tau_{k+1}).\psi]\!]^{\mathcal{T}}_{\eta'}([f'_1, \dots, f'_k]). \quad \text{(I)}$$

The algorithm distinguishes two cases.

- If $[f_1, \dots, f_k] = []$ and the type of $x$ is not $\mathsf{Pr}$, i.e. $x$ is a function with no arguments supplied, the algorithm returns a dummy function and postpones the fixpoint computation until arguments are provided. Formally, after $\beta$-reduction, the returned function is the same as MC-HFL$(\mu(x : \tau_1 \to \dots \tau_{k+1}).\psi, [f_1, \dots, f_k], \eta)$. Note that since $dom(\eta(x)) = \emptyset$, statement (I) trivially holds.

- In case the arguments have been provided, i.e. $[f_1, \dots, f_k] \neq []$ or $x$ is of primitive type $\mathsf{Pr}$, statement (I) is in fact an invariant of the `repeat`-loop in Algorithm MC-HFL. It trivially holds before the loop because $dom(\eta(x)) = \{[f_1, \dots, f_k]\}$ only, and $\eta(x)$ maps this tuple to the bottom element of $\tau_{k+1}$.

  Furthermore, if statement (I) holds at the beginning of one iteration of the `repeat`-loop then it also holds after this iteration. This is simply a consequence of monotonicity, the hypothesis, and the fact that $[\![\mu(x : \tau_1 \to \dots \tau_{k+1}).\psi]\!]^{\mathcal{T}}_{\eta'}$ is a unique fixpoint of $\psi$ w.r.t. $\sqsubseteq$: if we have $\eta(x)([f'_1, \dots, f'_k]) \sqsubseteq [\![\mu(x : \tau_1 \to \dots \tau_{k+1}).\psi]\!]^{\mathcal{T}}_{\eta'}([f'_1, \dots, f'_k])$ for all such tuples then, by monotonicity and the definition of the pointwise inclusion ordering, we also have $[\![\psi]\!]^{\mathcal{T}}_{\eta'\{x \mapsto \eta(x)\}} \sqsubseteq [\![\psi]\!]^{\mathcal{T}}_{\eta'\{x \mapsto [\![\mu(x:\tau_1 \to \dots \tau_{k+1}).\psi]\!]^{\mathcal{T}}_{\eta'}\}}$. Now note that the latter is (because it is a fixpoint) equal to $[\![\mu(x : \tau_1 \to \dots \to \tau_{k+1}).\psi]\!]^{\mathcal{T}}_{\eta'}$.

  And the former is, by hypothesis, the value of $\eta(x)$ on all arguments in $dom(\eta(x))$ at the end of this `repeat`-loop iteration (note that $\eta(x)$ is updated with the value of MC-HFL$(\psi, [f'_1, \dots, f'_{n-1}], \eta)$ for all $[f'_1, \dots, f'_{n-1}] \in dom(\eta(x)))$.

This implicitly shows that – on finite transition systems – the loop eventually terminates. Since $dom(\eta(x))$ at most grows in each iteration, we have $[f_1, \dots, f_k] \in dom(\eta(x))$ at termination point, and the soundness part of Lemma (33) immediately follows from the fact that (I) holds at this point.

**Completeness of the $\mu$-part.** We will prove this part using fixpoint induction. For any two functions $f, g$ of type $\tau_1 \to \dots \to \tau_{k+1}$ and a set $D \subseteq \tau_1 \times \dots \times \tau_k$, we write

$$f \sqsubseteq_D g \text{ iff for all } [a_1, \dots, a_k] \in D : f([a_1, \dots, a_k]) \sqsubseteq g(a_1, \dots, a_k]).$$

Now consider again the call MC-HFL$(\mu x.\psi, [f_1, \dots, f_k], \eta)$. Let $D := dom(\eta(x))$ upon termination of the `repeat`-loop. An immediate consequence of the induction hypothesis for $\psi$ is the following:

$$[\![\psi]\!]^{\mathcal{T}}_{\eta'\{x \mapsto f\}} \sqsubseteq_D \eta(x). \tag{II}$$

for any function $f \in \nabla(\eta(x))$. This is because the `repeat`-loop is iterated on the whole of $D$ until stability is reached, i.e. until MC-HFL$(\psi, [f'_1, \ldots, f'_k], \eta) = \eta(x)([f'_1, \ldots, f'_k])$ holds for all $[f'_1, \ldots, f'_k] \in D$. By I.H. $[\![\psi]\!]^{\mathcal{T}}_{\eta'}([f'_1, \ldots, f'_k]) \sqsubseteq$ MC-HFL$(\psi, [f'_1, \ldots, f'_k], \eta)$ for all $[f'_1, \ldots, f'_k]$. Hence for all $[f'_1, \ldots, f'_k] \in D$ we also have $[\![\psi]\!]^{\mathcal{T}}_{\eta'}([f'_1, \ldots, f'_k]) \sqsubseteq \eta(x)([f'_1, \ldots, f'_k])$ and from this follows the claim by definition of $\sqsubseteq_D$.

We now extend the function $\eta(x)$ to a function $\eta^\top(x)$ in the following way.

$$\eta^\top(x)([f'_1, \ldots, f'_k]) \;:=\; \begin{cases} \eta(x)([f'_1, \ldots, f'_k]) & \text{, if } [f'_1, \ldots, f'_k] \in D. \\ \top_{\tau_{k+1}} & \text{, otherwise.} \end{cases}$$

Now note that we have

$$[\![\psi]\!]^{\mathcal{T}}_{\eta'\{x \mapsto \eta^\top(x)\}} \;\sqsubseteq\; \eta^\top(x).$$

i.e. the function on the right subsumes the one on the left on *all* arguments. For arguments in $D$ this is stated in (II) above. For all other arguments this is trivially true by the construction of $\eta^\top(x)$. But then $\eta^\top(x)$ is a pre-fixpoint of $\psi$ and, hence, we have $[\![\mu x.\psi]\!]^{\mathcal{T}}_{\eta'} \sqsubseteq \eta^\top(x)$. In particular, the inclusion holds for all argument tuples in $D$. Since the domain of $\eta(x)$ at most grows in each iteration of the `repeat`-loop, we have $[f_1, \ldots, f_k] \in D$ and therefore $[\![\mu x.\psi]\!]^{\mathcal{T}}_{\eta'}([f_1, \ldots, f_k]) \sqsubseteq$ MC-HFL$(\mu x.\psi, [f_1, \ldots, f_k], \eta)$ which finishes the proof.□

### 5.5.4 Applications and Evaluation in Practice

The expressive power of HFL allows to encode numerous interesting problems as model checking instances. This section covers the encoding of the following problems: NFA universality (NFA-UNIV), Quantified Boolean Formulas (QBF), Satisfiability of modal logic $K$ ($K$-SAT) and Shortest Common Supersequence (SCS). All of these problems can already be encoded in HFL[1].

A possible benefit of studying such encodings is to extract formerly unknown algorithms for these problems by analysing the behaviour of the optimised model checker. The justification for this potential lies in the unusual, yet very succinct problem formulation which HFL imposes upon the "programmer". It is fair to say that it is not common practice among programmers to think of methods and routines as fixpoints of concrete functions. This however is the only recursion device which is offered by HFL. In this regard we will use HFL as an extremely succinct programming language in this section and demonstrate the validity of the claim that HFL can be a valuable tool for designing new and original algorithms which at least in case of NFA-UNIV and SCS are competitive to known ones.

### NFA Universality

We start by picking NFA-UNIV to demonstrate how encoding a problem as a model checking instance can lead to an efficient solution. In fact, we have already introduced the encoding in Example 19 without mentioning it.

Recall the model in Example 19. If the proposition $q$ is interpreted as a flag for being a final state then the whole model can easily be viewed as an NFA. In this context the formula

$$\varphi_{NFA} \;:=\; \big(\mu(x : \mathsf{Pr} \to \mathsf{Pr}).\lambda(Z : \mathsf{Pr}).Z \vee \bigvee_{a \in \Sigma} x\,[a]Z\big)\, \neg q$$

translates to *"there is a word $w$, s.t. all states reachable under $w$ are non-final"*. NFA-UNIV is solved by checking whether or not the starting state satisfies this formula. This problem suits well to practically evaluate the behaviour of our model checking algorithm since we can easily generate random NFA instances upon which the formula is model checked.

**Local Fixpoint Computation in Practice** We now give empirical evidence of the benefits of local fixpoint computations and demonstrate that the necessity to compute larger fragments of the complete domain rarely occurs. Algorithm MC-HFL has been implemented as a prototype[2] in OCaml and run on the following random model for NFAs (by [TV05]) in order to guarantee a wide spectrum of test cases: two parameters $s$ and $t$ determine the number of randomly chosen final states and transitions in an NFA w.r.t. the total number of states $n$. The ratios $f := \frac{s}{n}$ and $r := \frac{t}{n}$ are called final state density and transition density respectively. To perform the universality tests, we fix $n = 10$ and generate 20 random NFAs for each of 250 pairs $(r, f)$ with $0 \leq r \leq 2.5$ and $0 \leq f \leq 1$.

The average number of arguments needed in the fixpoint computation by algorithm MC-HFL in dependence of $(r, f)$ is depicted in Fig. 5.6. Note that the number of possible arguments $|2^{\mathcal{S}}|$ is 1024 in this case. Fig. 5.6 shows that in all cases the algorithm is far away from exhaustive fixpoint calculation on the full argument set $2^{\mathcal{S}}$. Even for the most difficult instances which in our tests are $f = 0.1$ and $r$ between 1.4 and 1.6, the number of needed arguments never gets anywhere near that. The average number of arguments distributed over all 5000 tests is just 13.2 and the highest number of arguments ever measured during the tests is 109.

[2]see `http://www2.tcs.ifi.lmu.de/~axelsson/veri_non_reg/mchfl_tool_doc.html`

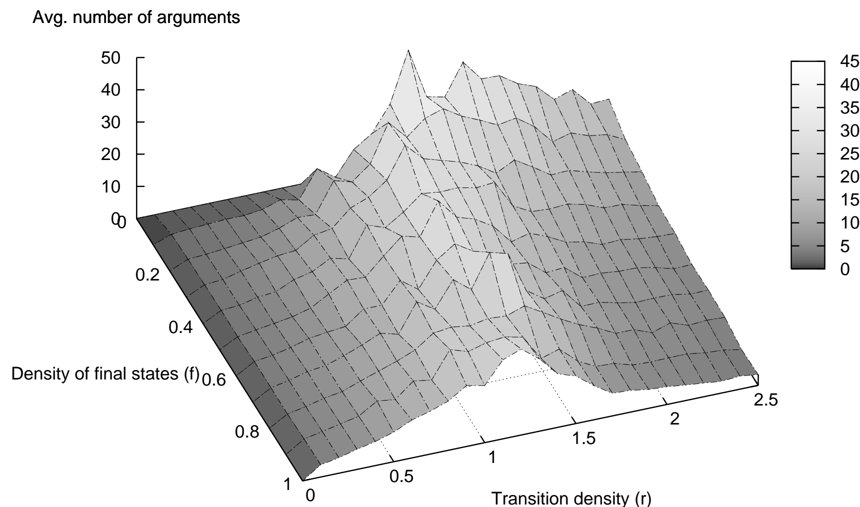



Figure 5.6: Number of arguments in function table ($n = 10$).

It is reasonable to assume that the approach of guiding the fixpoint iterations locally through neededness analysis also proves to be successful in other cases (on different formulas) unless the underlying models have been constructed pathologically to enforce an exponential behaviour.

**Optimising Algorithm MC-HFL w.r.t. a Fixed Formula** There are still several standard performance enhancements available, e.g. acceleration of the fixpoint computation by exploiting monotonicity, in order to optimise this algorithm.

However, we need to observe that algorithm MC-HFL will be used on fixed formulas in most cases. In many verification tasks the property to be checked is fixed while the models change. This holds especially for non-regular properties since non-regularity often eliminates dependence on model sizes, etc. It is therefore much more beneficial to regard MC-HFL as a *template* for specialised cases rather than a general algorithm for all kinds of verification purposes. Model checking a fixed formula bears a higher potential for algorithm optimisations which possibly cannot be achieved for varying formulas.

Consider the algorithm's behaviour on the formula of Ex. 19 as depicted in the table there. If we follow the succession of the fixpoint iteration closely, a simple pattern can be observed: the iterated function $\lambda Y.Y \vee \bigvee_{a\in\Sigma} X\ [a]Y$ takes an argument (initially the set $[\![\neg q]\!]^{\mathcal{A}}$) and returns its union with the set of its recursive $[a]$-predecessors for all $a \in \Sigma$. But this set is exactly the union of the elements of $dom(x)$, each of them the result of a single $[a]Y$

computation step. So the return value does not provide any additional information if the set of needed arguments is known. Furthermore, since only a union operation is performed, it suffices to keep track of $\subseteq$-maximal sets of arguments. This insight immediately leads to an optimisation by discarding all redundant information. It is obviously not necessary to protocol all these values in the fixpoint iterations – when in the end all we want to know is whether or not the initial automaton state is included in the union over all arguments. It suffices to iterate this schema until no more arguments enter the table, and then to form their unions. This, however, means that, by monotonicity of the $[a]$-operators, one can always discard the larger of two arguments that are comparable w.r.t. $\subseteq$ which leads to the idea of storing $dom(x)$ as an *antichain*.

An antichain over an NFA $\mathcal{A}$ is a set $\mathcal{C}$ of pairwise incomparable (w.r.t. set inclusion) sets of states of $\mathcal{A}$. These antichains form a complete lattice when equipped with the following order:

$$\mathcal{C} \sqsubseteq \mathcal{C}' \quad \text{iff} \quad \forall C \in \mathcal{C}\ \exists C' \in \mathcal{C}' \text{ s.t. } C \subseteq C'.$$

This naturally induces a notion of supremum $\mathcal{C} \sqcup \mathcal{C}'$ as the smallest antichain (w.r.t. $\sqsubseteq$) which contains both $\mathcal{C}$ and $\mathcal{C}'$.

The basic principle of the optimization is to populate an antichain with sets of states which uphold the possibility of generating a word that is not included in the language of the automaton. This can be achieved by loosely speaking applying the modal $[a]$-operator (for all $a \in \Sigma$) to its elements and minimizing the resulting set to an antichain. More formally, define the following monotone operation on antichains:

$$CPre(\mathcal{C}) \;:=\; \lceil \{S \subseteq Q \mid \exists T \in \mathcal{C}\ \exists a \in \Sigma \text{ s.t. } S = [\![ [a]X ]\!]^{\mathcal{A}}_{\{X \mapsto T\}}\} \rceil$$

where the $\lceil \cdot \rceil$ operator discards all sets which are subsumed by another set in this set of sets – i.e. it makes an antichain of the expression on the right-hand side.

This is exactly the idea which Henzinger et al. have in mind when they characterise NFA-UNIV using least fixpoints in antichain lattices in [WDHR06].

**Lemma 34 ([WDHR06])** Let $\mathcal{A}$ be an NFA over the alphabet $\Sigma$ with state set $Q$, initial state $q_0$ and final states $F$. Then

$$L(\mathcal{A}) \neq \Sigma^* \ \text{ iff } \ \{\{q_0\}\} \sqsubseteq \bigsqcap \{\mathcal{C} \mid CPre(\mathcal{C}) \sqcup \{Q \setminus F\} \sqsubseteq \mathcal{C}\}.$$

Of course, the least fixpoint can be computed by a straight-forward fixpoint iteration: Define $\mathcal{C}^0 := \{\emptyset\}$ and $\mathcal{C}^i := CPre(\mathcal{C}^{i-1}) \sqcup \{Q \setminus F\}$. The following table compares in parallel

two runs of MC-HFL and the antichain method on Ex. 19:

| $X$ | $\{3\}$ | $\{2,3\}$ | $\{1,2,3\}$ |
|---|---|---|---|
| 0 | $\emptyset$ | | |
| 1 | $\{3\}$ | $\emptyset$ | |
| 2 | $\{3\}$ | $\{2,3\}$ | $\emptyset$ |
| 3 | $\{2,3\}$ | $\{2,3\}$ | $\{1,2,3\}$ |
| 4 | $\{2,3\}$ | $\{1,2,3\}$ | $\{1,2,3\}$ |
| 5 | $\{1,2,3\}$ | $\{1,2,3\}$ | $\{1,2,3\}$ |
| 6 | $\{1,2,3\}$ | $\{1,2,3\}$ | $\{1,2,3\}$ |

$$\begin{array}{lllll}
\mathcal{C}^0 & := & \{\emptyset\} & & \\
\mathcal{C}^1 & := & CPre(\mathcal{C}^0) \sqcup \{Q \setminus F\} & = & \{\{3\}\} \\
\mathcal{C}^2 & := & CPre(\mathcal{C}^1) \sqcup \{Q \setminus F\} & = & \{\{2,3\}\} \\
\mathcal{C}^3 & := & CPre(\mathcal{C}^2) \sqcup \{Q \setminus F\} & = & \{\{1,2,3\}\} \\
\mathcal{C}^4 & := & CPre(\mathcal{C}^3) \sqcup \{Q \setminus F\} & = & \{\{1,2,3\}\}
\end{array}$$

The cost reduction of the antichain method is established by the fact that it simply computes $\lceil dom(x) \rceil$, i.e. the antichain of the currently present arguments. One can show that $\lceil dom(x^i) \rceil = \mathcal{C}^{i+1}$, where $dom(x^i)$ is the currently needed domain of the $i$th fixpoint approximant w.r.t. a given argument and a partial evaluation according to MC-HFL.
It turns out that the result of this optimisation is exactly the method devised by Henzinger et al. in [WDHR06]. Their tool shows a very good performance on the universality test for NFAs and does apparently outperform the classical powerset construction by several orders of magnitude.

### Quantified Boolean Formulas

By not just restricting the term "model checking" to a method used in automatic program verification but understanding it as a general logic problem we can obtain algorithms for various other problems as well. Note that NFA-UNIV is PSPACE-complete, and it is therefore reasonable to try to encode the standard PSPACE-complete problem QBF as an HFL$^1$ model checking problem.
It is well-known that every quantified Boolean formula can be put into prenex CNF normal form $Q_1 x_1 \ldots . Q_n x_n . \bigwedge_i \bigvee_{j_i} l_{i,j_i}$ with the $Q_k \in \{\exists, \forall\}$, and the $l_{i,j_i}$ literals over the variables $x_1, \ldots, x_n$. The problem QBF is to decide whether or not such a formula evaluates to 1 under the usual interpretation of the Boolean operators and the quantifiers over the domain $\{0, 1\}$.
With each QBF formula $\Phi$ we associate a loop-free transition system $\mathcal{T}_\Phi$ which is exemplarily shown in Fig. 5.7 for $\Phi = \exists x_1 . \forall x_2 . \exists x_3 . \forall x_4 . (x_2 \vee \neg x_4) \wedge (x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee x_3)$. It uses atomic propositions $\exists$, $\forall$ to mark the type of quantification over a variable, $c$ to

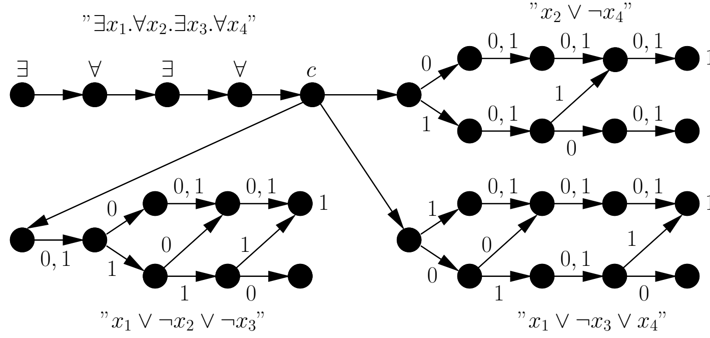



Figure 5.7: A transition system representation of a QBF formula.

indicate the branching into the different clauses, and 1 to mark the value of a clause under an assignment valuation given by a path through each clause's component. Its actions are 0 and 1 for representing variable values, and an anonymous one for branching into different clauses and for separating the quantifiers in the prefix.

Evaluation to 1 of $\Phi$ can now be expressed in $\mathrm{HFL}^1$ as follows.

$$\varphi_{QBF} \;:=\; \Big(\mu(x:\mathsf{Pr}\to\mathsf{Pr}).\lambda(Z:\mathsf{Pr}).\big(c\to[-]Z\big)\wedge\big(\exists \;\to\; \langle -\rangle(x\,\langle 0\rangle Z)\vee\langle -\rangle(x\,\langle 1\rangle Z)\big)\wedge \big(\forall \;\to\; \langle -\rangle(x\,\langle 0\rangle Z)\wedge\langle -\rangle(x\,\langle 1\rangle Z)\big)\Big)\;1$$

Again, $\varphi_{QBF}$ does not depend on the underlying QBF formula $\Phi$. It is therefore possible to obtain a QBF solver by analysing the behaviour of algorithm MC-HFL on $\varphi_{QBF}$ and specialised transition systems $\mathcal{T}_\Phi$. For example, it is not hard to see that the fixpoint iteration always terminates after a number of steps given by the length of the quantifier prefix. It can therefore be made explicit through a `for`-loop. Furthermore, antichains can also be used to replace the arguments of the function table. Preliminary results show that this is far away from yielding a competitive QBF solver. However, it may be interesting to investigate combinations of this bottom-up approach with existing solvers that mostly work top-down.

### Encoding the Satisfiability Problem for Modal Logic $K$

Another important problem that $\mathrm{HFL}^1$ can express and that therefore can be solved using algorithm MC-HFL is the satisfiability problem for modal logic $K$, extending propositional logic with the modal operators $\Diamond$ and $\Box$. For technical reasons and simplicity we assume modal formulas to be in positive normal form and only consider the uni-modal case.

A *tableau* for a modal formula $\Phi$ is a finite tree whose nodes are labeled with subsets of $\mathsf{sub}(\Phi)$, called *sequents*, s.t. each inner node is an instance of one of the following rules, and each leaf is consistent, i.e. it does not contain an atomic proposition $q$ and its complement $\bar{q}$.

$$(\wedge)\ \frac{\psi_1, \psi_2, \Gamma}{\psi_1 \wedge \psi_2, \Gamma} \qquad (\vee)\ \frac{\psi_i, \Gamma}{\psi_1 \vee \psi_2, \Gamma}\ i \in \{1,2\}$$

$$(\Diamond)\ \frac{\varphi_1, \psi_1, \ldots, \psi_m \qquad \ldots \qquad \varphi_n, \psi_1, \ldots, \psi_m}{\Diamond\varphi_1, \ldots, \Diamond\varphi_n, \Box\psi_1, \ldots, \Box\psi_m, l_1, \ldots, l_k}$$

where $\{l_1, \ldots, l_k\}$ must be a consistent set of literals.

We will show that K-SAT, the satisfiability problem for $K$ can be encoded as a model checking problem for HFL[1]. With a formula $\Phi \in K$ we associate a transition system $\mathcal{T}_\Phi$ with states $\mathsf{sub}(\Phi)$, the subformulas of $\Phi$. There are five accessibility relations:

- $\xrightarrow{l}$ and $\xrightarrow{r}$ connect each subformula to its immediate superformula marking it as its left, resp. right argument assuming that the modal operators only have a right one,
- $\xrightarrow{s}$ (for "select") introduces a linear order on $\mathsf{sub}(\Phi)$ with $\Phi$ being the maximal element,
- $\xrightarrow{c}$ (for "conflict") connects all propositions $q$ to their complements $\bar{q}$ and vice-versa,
- $\xrightarrow{t}$ (for "test") connects $\Phi$ to every other subformula.

Each subformula is labeled with one of $p_\wedge, p_\vee, p_\Diamond, p_\Box, \mathit{prop}$ according to the type of the subformula. Finally, $\Phi$ is also labeled with *init*.

A $\Gamma \subseteq \mathsf{sub}(\Phi)$, i.e. a sequent in a tableau, can be represented naturally by an object of type $\mathsf{Pr}$. The existence of a tableau for $\Phi$ can then be encoded by a function of type $\mathsf{Pr} \to \mathsf{Pr}$ that takes the current sequent, decides which rule to apply and continues recursively with the corresponding premisses. The relations $\xrightarrow{l}$ and $\xrightarrow{r}$ are used to model subformula replacement in an application of a tableau rule, and relation $\xrightarrow{s}$ is used to select the principal formula of the next rule application, i.e. the one determining which rule to apply. The transition representation of the modal formula $\Phi = \Diamond(q \wedge \Box\bar{q}) \wedge \Box(\bar{q} \vee \Diamond q)$ is given in Fig. 5.8. To avoid clutter we do not show the relation $\xrightarrow{t}$ which simply has arcs from the leftmost state to each other including itself.

Now consider the following formula $\varphi_{KSAT}$:

$$\Big(\mu Z^{\tau_1}.\lambda X.\ [t](X \to [c]\neg X)$$

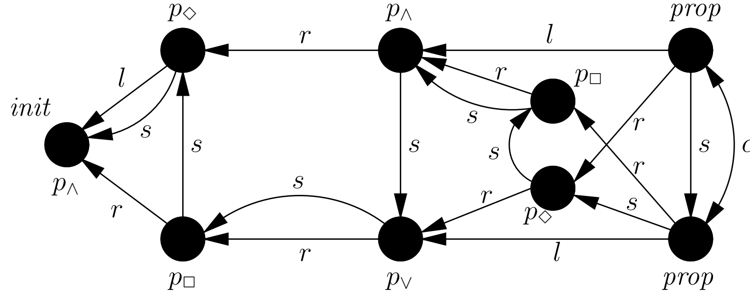



Figure 5.8: A transition system representation of a formula in modal logic $K$.

$$
\begin{aligned}
\wedge \Big( \; & [t](X \to prop) \\
\vee \Big( & [t]\big((X \wedge \neg prop) \to (p_\Diamond \vee p_\Box)\big) \wedge \\
& \quad \Big(\nu Y^{\tau_1}.\lambda V.\Big([t](V \to X \wedge p_\Diamond) \to \big(Z\ (\langle r\rangle V \vee \langle r\rangle (X \wedge p_\Box))\big)\Big) \\
& \qquad \wedge \big([t]\neg V \vee (Y\ \langle s\rangle V)\big)\Big)\ init\Big) \\
\vee \Big( & \mu Y^{\tau_1}.\lambda V.\ [t]\neg V \\
& \quad \vee \Big([t](V \to X \wedge p_\wedge) \wedge \big(Z\ ((X \wedge \neg V) \vee \langle l\rangle V \vee \langle r\rangle V)\big)\Big) \\
& \quad \vee \Big([t](V \to X \wedge p_\vee) \wedge \big((Z\ ((X \wedge \neg V) \vee \langle l\rangle V))\vee \\
& \qquad\qquad (Z\ ((X \wedge \neg V) \vee \langle r\rangle V))\big)\Big) \\
& \quad \vee (Y\ \langle s\rangle V)\Big)\Big)\ init\Big)\Big)\ init.
\end{aligned}
$$

This formula will be evaluated in state $\Phi$ of $\mathcal{T}_\Phi$. The outer least fixpoint recursion through variable $Z$ finds a tableau. Variable $X$ represents a sequent in this tableau starting with $\Phi$, the only node satisfying *init*. The first line assures that $X$ represents a propositionally consistent sequent. This is the case iff no element of $X$ has a $c$-successor in $X$. Note that here we use the relation $\xrightarrow{t}$ in order to test in state $\Phi$ whether or not something holds in all states.

Then there are three disjuncts. The first one applies if $X$ consists of propositions only, hence, a tableau leaf is found. The second disjunct applies if $X$ consists of literals and $\Diamond$- and $\Box$-formulas only. Hence, rule $(\Diamond)$ needs to be modeled. The inner fixpoint recursion traverses through the entire set of subformulas starting with $\Phi$. In each iteration, variable

$V$ contains a single node only because the relation $\xrightarrow{s}$ is deterministic. It then checks whether $V$ consists of a $\Diamond$-formula in the current sequent $X$. If this is the case, it calls the tableau building function $Y$ again and passes it, as the new sequent, the argument of that $\Diamond$-formula as well as the arguments of all $\Box$-formulas in $X$.

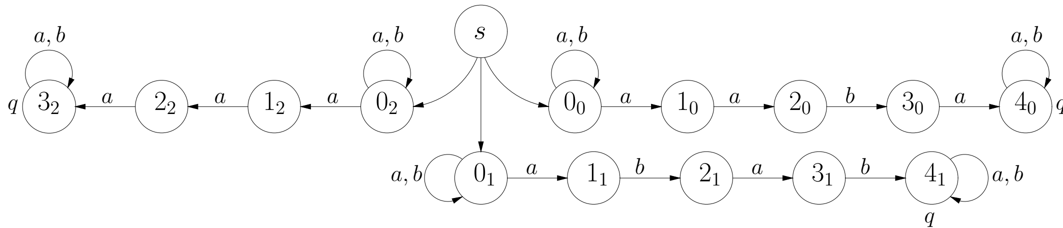
Finally, the third disjunct models applications of rules ($\wedge$) and ($\vee$). Similar to the case above, there is an inner fixpoint function which recursively selects a Boolean subformula of the current sequent. This is stored in $V$. If $V$ consists of a conjunction it gets replaced by its conjuncts according to rule ($\wedge$). This is modeled by calling $Y$ again on the argument consisting of everything in $X$ without the node in $V$ but adding the $r$- and $l$-predecessor of $V$. A similar construction applies to model rule ($\vee$) for disjunctions. Note that this rule is nondeterministic, hence, we call $Y$ with either of two arguments including either of the two disjuncts.

Then we have, for any formula $\Phi \in K$: $\mathcal{T}_\Phi, \Phi \models \varphi_{KSAT}$ iff $\Phi$ is satisfiable.

### Shortest Common Supersequence

Some optimisation problems that require more than a yes/no answer can also be dealt with using an extension of algorithm MC-HFL that keeps track of parts of the solution to be computed. We sketch a new algorithm for the Shortest Common Supersequence problem (SCS): given a set $\{w_1, \ldots, w_n\}$ of finite words of some alphabet $\Sigma$, find a shortest $v \in \Sigma^*$ that contains all $w_i$ as subwords. The algorithm is obtained from the template MC-HFL using an antichain optimisation as in the case of NFA-UNIV.

The first step consists of building a transition system $\mathcal{T}$, here depicted for the words $\{aaba, abab, aaa\}$.




Next, consider the HFL[1] formula

$$\varphi_{SCS} := \big(\mu(x : \mathsf{Pr} \to \mathsf{Pr}).\lambda(Z : \mathsf{Pr}).[-]Z \vee \bigvee_{a \in \Sigma} x\ \langle a \rangle Z\big)\ q.$$

Each state in $\mathcal{T}$ satisfies $\varphi_{SCS}$ which only reflects the fact that for every finite set of words there is a word containing all of them. However, suppose the arguments in the table for the

fixpoint iteration in this formula are annotated in the following way: the initial argument receives the annotation $\epsilon$, and if an argument $Z$ with annotation $w$ causes another argument to be created in the table through the recursive call of $X\ \langle a \rangle Z$ then the new argument receives the annotation $aw$.

Now note the apparent similarity of this formula with the one from Ex. 19 expressing NFA-UNIV. In both cases the subformulas $X\ \psi(Z)$ only occur under a disjunction. Hence, the argument row of the function table can again be optimised into an antichain, and the evaluation of the formula can be regarded as a fixpoint iteration in an antichain lattice. It terminates when the topmost state of $\mathcal{T}$ occurs in an element of the current antichain, and that element's annotation is the solution to the SCS problem.

The computation of the solution $aaabab$ using annotated antichains is found as follows. Let $I := \{4_0, 4_1, 3_2\}$. For a set $S$ we write $S_I^w$ to abbreviate $(S \cup I)^w$ where the superscript simply denotes the word annotation of this set.

$$\begin{aligned}
\mathcal{C}_0 &:= \{I^\epsilon\} \\
\mathcal{C}_1 &:= \{\{2_2, 3_0\}_I^a, \{3_1\}_I^b\} \\
\mathcal{C}_2 &:= \{\{2_2, 2_1, 3_0\}_I^{ab}, \{2_2, 1_2, 3_0\}_I^{aa}, \{3_1, 2_0\}_I^{ba}\} \\
\mathcal{C}_3 &:= \{\{2_2, 2_1, 3_0, 1_0\}_I^{aba}, \{2_2, 1_2, 0_2, 3_0\}_I^{aaa}, \{3_1, 1_1, 2_0\}_I^{bab}\} \\
\mathcal{C}_4 &:= \{\{2_2, 2_1, 0_1, 3_0, 1_0\}_I^{abab}, \{2_2, 1_2, 0_2, 3_0\}_I^{aaaa}, \{2_2, 1_2, 3_0, 0_0\}_I^{aaba}, \\
&\qquad \{0_2, 3_1, 2_0\}_I^{baaa}, \{3_1, 1_1, 2_0\}_I^{baba}\} \\
\mathcal{C}_5 &:= \{\{\ldots\}_I^{ababa}, \{\ldots\}_I^{abaaa}, \{\ldots\}_I^{aaaaa}, \{2_2, 1_2, 0_1, 3_0, 0_0\}_I^{aabab}, \\
&\qquad \{\ldots\}_I^{baaba}, \{\ldots\}_I^{baaaa}, \{\ldots\}_I^{babab}\} \\
\mathcal{C}_6 &:= \{\ldots, \{2_2, 1_2, 0_2, 0_0, 0_1\}_I^{aaabab}, \ldots\}
\end{aligned}$$

Finally, since a set containing $\{0_0, 0_1, 0_2\}$ has been found, $s$ is included in the next iteration, and the solution is the annotation of this witnessing set.

# Chapter 6

# Further Work

We have investigated the model-theoretic properties, expressivity and model checking problem of PDL[$\mathcal{L}$] for arbitrary classes of formal languages $\mathcal{L}$. Some questions regarding its expressivity are however still open. For instance the question whether the result that PDL[$\mathcal{L}$] gains additional power from the test operator up to the context-free languages extends to PDL over more expressive language classes or if the test operator can somehow be simulated in these fragments.

Clearly, one could also extend PDL[$\mathcal{L}$] with additional operators such as *converse* or $\Delta$ as defined in [Str81]. In fact, we have compared the latter to CTL[$\mathcal{L}$] in [ALL$^+$b]. It turns out that PDL[$\mathcal{L}$] with a $\Delta$-operator is strictly more expressive than CTL[$\mathcal{L}$] for deterministic automata models. Strictness is merely a consequence of the fact that CTL[$\mathcal{L}$] is not capable of expressing fairness while PDL[$\mathcal{L}$] with $\Delta$ is. The embedding is otherwise straight-forward. Nondeterministic automata classes are however not generically embeddable, except when the automaton class is closed under determinisation, of course. For instance are CTL[CFL] and PDL[CFL] with $\Delta$ mutually incomparable.

Regarding model checking, the correspondence to the emptiness problem should extend to PDL[$\mathcal{L}$] with $\Delta$, except that automata models with a Büchi acceptance condition need to be considered instead of normal ones, since that is basically what the $\Delta$-operator amounts to.

There are also some open problems regarding the expressivity of CTL[$\mathcal{L}$] . In particular, we do not know whether CTL[DCFL] $\lneq$ CTL[CFL] holds.

Another idea is that in a similar manner as parametric CTL operators have been adorned with formal languages, one can think of such extensions for CTL$^*$. It would be very interesting to analyse the interplay between logical machinery and formal languages in

such a setting.

On a more general level, we are interested in a unifying logic for all three logical frameworks presented in this work. Some attempts were made to embed PDL[IL] into HFL but all of them failed in the end. The problematic case is of course the diamond formula scheme where the task is to simulate a derivation resulting in a word which coincides with a path in the model. A direct approach which simulates the derivation relation by a simultaneous fixpoint using nonterminals as variables in the way demonstrated by the embedding of PDL[CFL] fails here, because the only way we could see to encode the stack of each nonterminal was as a list of arguments in some function of the $\lambda$-calculus. However, the encoding of lists in the simply typed $\lambda$-calculus does not support the deletion of elements which corresponds to pop-operations on stacks and hence the whole construction fails.

Another approach was to try to encode the language derivation part of the algorithm used for the computation of closed paths in HFL. The reasons why this failed were similar and raise the question whether this is an inherent weakness of HFL. If so, then the question immediately arises what kind of feature a logic has to support in order to be able to simulate such behaviour. Or, more generally speaking, to serve as a unifying logic which links automata classes and logics like MSO links $\mathcal{L}_\mu$ and finite automata. The correspondence between temporal logic and automata which exceed the regular or context-free has to our knowledge never been analysed systematically.

On a more practically oriented level it might be interesting to follow up the matter of algorithm development via encoding problems as model checking instances of HFL and to observe the behaviour of the fixpoint approximation in order to gain insight into the problem and to develop optimised algorithms from this.

# Bibliography


[Aho68] Alfred V. Aho. Indexed grammars - an extension of context-free grammars. *Journal of the Association for Computing Machinery*, 15(4):647–671, 1968.

[Aho69] Alfred V. Aho. Nested stack automata. *Journal of the Association for Computing Machinery*, 16(3):383–406, 1969.

[AL07] Roland Axelsson and Martin Lange. Model checking the first order fragment of higher-order fixpoint logic. In N. Dershowitz and A. Voronkov, editors, *Proc. 14th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR'07*, volume 4790 of *LNCS*, pages 62–76, Yerevan, Armenia, 2007. Springer.

[ALLa] Roland Axelsson, Martin Lange, and Markus Latte. Alternating context-free grammars are conjuctive grammars and vice versa. Submitted for publication 2010.

[ALL+b] Roland Axelsson, Martin Lange, Markus Latte, Matthew Hague, and Stephan Kreutzer. Extended computation tree logic. Submitted for publication 2010.

[ALS07] Roland Axelsson, Martin Lange, and Rafal Somla. The complexity of model checking higher order fixpoint logic. *Logical Methods in Computer Science*, 3(2:7):1–33, 2007.

[AM04] Rajeev Alur and P. Madhusudan. Visibly pushdown languages. In *Proc. of the thirty-sixth annual ACM symposium on Theory of computing*, pages 202–211, New York, NY, USA, 2004. ACM.

[BAMP81] Mordechai Ben-Ari, Zohar Manna, and Amir Pnueli. The temporal logic of branching time. In *Proc. of the 8th ACM SIGPLAN-SIGACT symposium on*

*Principles of programming languages*, pages 164–176, New York, NY, USA, 1981. ACM.

[Bék84] H. Békiċ. *Programming Languages and Their Definition, Selected Papers*, volume 177 of *LNCS*. Springer, 1984.

[BEM97] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *Proc. 8th Int. Conf. on Concurrency Theory, CONCUR'97*, volume 1243 of *LNCS*, pages 135–150. Springer, 1997.

[BK85] Jan A. Bergstra and Jan Willem Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77–121, 1985.

[Bou96] Pierre Boullier. Another facet of lig parsing. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, pages 87–94, Morristown, NJ, USA, 1996. Association for Computational Linguistics.

[BP81] Francine Berman and Mike Paterson. Propositional dynamic logic is weaker without tests. *Theoretical Computer Science*, 16:321–328, 1981.

[BS06] J. Bradfield and C. Stirling. Modal mu-calculi. In *Handbook of Modal Logic*. Elsevier, 2006.

[Büc60] J. R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. of the Int. Congress on Logic, Methodology and Philosophy of Science*, pages 1–11. Stanford University Press, 1960.

[CE81] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logics of Programs: Workshop*, volume 131 of *LNCS*, pages 52–71, Yorktown Heights, New York, 1981. Springer.

[CS92] Rance Cleaveland and Bernhard Steffen. A linear-time model-checking algorithm for the alternation-free modal mu-calculus. In *Proc. of the 3rd International Workshop on Computer Aided Verification*, pages 48–58, London, UK, 1992. Springer-Verlag.

[EH85] E. A. Emerson and J. Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Computer and System Sciences*, 30:1–24, 1985.

[EH86] E. Allen Emerson and Joseph Y. Halpern. "sometimes" and "not never" revisited: on branching versus linear time temporal logic. *Journal of the Association for Computing Machinery*, 33(1):151–178, 1986.

[EJ88] E. Allen Emerson and Charanjit S. Jutla. The complexity of tree automata and logics of programs (extended abstract). In *29th Annual Symposium on Foundations of Computer Science*, pages 328–337. IEEE, 1988.

[EJ00] E. Allen Emerson and Charanjit S. Jutla. The complexity of tree automata and logics of programs. *SIAM Journal on Computing*, 29(1):132–158, 2000.

[Elg61] C.C. Elgot. Decision problems of finite automata design and related arithmetics. *Transactions of the American Mathematical Society*, 98:21–52, 1961.

[Eme87] E. Allen Emerson. Uniform inevitability is tree automaton ineffable. *Information Processing Letters*, 24(2):77–79, 1987.

[FL79] M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18:194–211, 1979.

[Gaz88] G. Gazdar. Applicability of indexed grammars to natural languages. In U. Reyle and C. Rohrer, editors, *Natural Language Parsing and Linguistic Theories*, pages 69–94. Reidel, Dordrecht, 1988.

[HK99] David Harel and Moshe Kaminsky. Strengthened results on nonregular pdl. Technical Report MCS99-13, Weizmann Institute of Science, Dept. of Computer Science and Applied Mathmatics, 1999.

[HM80] M. C. B. Hennessy and R. Milner. On observing nondeterminism and concurrency. In J. W. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming, 7th Colloquium*, volume 85 of *LNCS*, pages 299–309, Noordweijkerhout, Netherlands, 1980. Springer-Verlag.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the Association for Computing Machinery*, 12(10):576–580, 1969.

[HPS83] David Harel, Amir Pnueli, and Jonathan Stavi. Propositional dynamic logic of nonregular programs. *Journal of Computer Systems and Science*, 26(2):222–243, 1983.

[HR93] David Harel and Danny Raz. Deciding properties of nonregular programs. *SIAM Journal on Computing*, 22(4):857–874, 1993.

[HS96] David Harel and Eli Singerman. More on nonregular pdl: Finite models and fibonacci-like programs. *Information and Computation*, 128(2):109–118, 1996.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[Jør94] N. Jørgensen. Finding fixpoints in finite function spaces using neededness analysis and chaotic iteration. In *Proc. 1st Int. Static Analysis Symposium, SAS'94*, volume 864 of *LNCS*, pages 329–345. Springer, 1994.

[JW96] D. Janin and I. Walukiewicz. On the expressive completeness of the propositional mu-calculus with respect to monadic second order logic. In *7th Int. Conf. on Concurrency Theory CONCUR '96*, pages 263–277, Pisa, Italy, 1996.

[Koz82] Dexter Kozen. Results on the propositional $\mu$-calculus. In *9th Int. Colloquium on Automata, Languages and Programming*, volume 140 of *Lecture Notes in Computer Science*, pages 348–359. Springer, 1982.

[Koz88] Dexter Kozen. A finite model theorem for the propositional -calculus. *Studia Logica*, 47:233–241, 1988.

[KP83] T. Koren and A. Pnueli. There exist decidable context free propositional dynamic logics. In *Logic of Programs*, volume 164 of *Lecture Notes in Computer Science*, pages 290–312. Springer, 1983.

[Kri63] S.A. Kripke. Semantical analysis of modal logic i - normal modal propositional calculi. *Zeitschrift fr mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963.

[Lan02] Martin Lange. Local model checking games for fixed point logic with chop. In Lubos Brim, Petr Jancar, Mojmír Kretínský, and Antonín Kucera, editors, *13th International Conference on Concurrency Theory*, volume 2421 of *Lecture Notes in Computer Science*, pages 240–254. Springer, 2002.

[Lan05] M. Lange. Model checking propositional dynamic logic with all extras. *Journal of Applied Logic*, 4(1):39–49, 2005.

[Lan10] Martin Lange. A ptime-hardness proof for emptiness of visibly pushdown languages. http://www2.tcs.ifi.lmu.de/~mlange/papers/emptinessvpl.pdf, 2010.

[LLS07] C. Löding, C. Lutz, and O. Serre. Propositional dynamic logic with recursive programs. *Journal of Logic and Algebraic Programming*, 73(1-2):51–69, 2007.

[LS02] Martin Lange and Colin Stirling. Model checking fixed point logic with chop. In *5th International Conference on Foundations of Software Science and Computation Structures*, volume 2303 of *Lecture Notes in Computer Science*, pages 250–263. Springer, 2002.

[LS06] Martin Lange and Rafal Somla. Propositional dynamic logic of context-free programs and fixpoint logic with chop. *Information Processing Letters*, 100(2):72–75, 2006.

[McN66] Robert McNaughton. Testing and generating infinite sequences by a finite automaton. *Information and Control*, 9(5):521–530, 1966.

[MO99] M. Mueller-Olm. A modal fixpoint logic with chop. In Christoph Meinel and Sophie Tison, editors, *Proc. 16th. Symposium on Theoretical Aspects in Computer Science, STACS'99*, volume 1563 of *LNCS*, pages 510–520, Trier, Germany, 1999. Springer.

[Mor89] E. Moriya. A grammatical characterization of alternating pushdown automata. *Theoretical Computer Science*, 67(1):75–85, 1989.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems Specification.* Springer, 1992.

[NST01] Naoya Nitta, Hiroyuki Seki, and Yoshiaki Takata. Security verification of programs with stack inspection. In *SACMAT*, pages 31–40, 2001.

[Okh01] A. Okhotin. Conjunctive grammars. *Journal of Automata, Languages and Combinatorics*, 6(4):519–535, 2001.

[Ott06] Martin Otto. Bisimulation invariance and finite models. January 2006.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[Pra80] Vaughan R. Pratt. A near-optimal method for reasoning about action. *Journal of Computer and System Sciences*, 20(2):231–254, 1980.

[Pri57] A.N. Prior. Time and modality, 1957.

[Rab69] Michael O. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 141:1–35, 1969.

[Sch02] Ph. Schnoebelen. The complexity of temporal logic model checking. In *4th Conference on Advances in Modal logic*, pages 393–436. King's College Publications, 2002.

[Str81] Robert S. Streett. Propositional dynamic logic of looping and converse. In *Proc. of the Thirteenth Annual ACM Symposium on Theory of Computation*, pages 375–383. ACM, 1981.

[TK07] S. Tanaka and T. Kasai. The emptiness problem for indexed language is exponential-time complete. *Systems and Computers in Japan*, 17(9):29–37, 2007.

[TV05] D. Tabakov and M. Y. Vardi. Experimental evaluation of classical automata constructions. In *Proc. 12th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning*, volume 3835 of *LNCS*, pages 396–411. Springer, 2005.

[vEB97] P. van Emde Boas. The convenience of tilings. In A. Sorbi, editor, *Complexity, Logic, and Recursion Theory*, volume 187 of *Lecture notes in pure and applied mathematics*, pages 331–363. Marcel Dekker, Inc., 1997.

[VsW94] K. Vijay-shanker and D. J. Weir. The equivalence of four extensions of context-free grammars. *Mathematical Systems Theory*, 27:27–511, 1994.

[VV04] Mahesh Viswanathan and Ramesh Viswanathan. A higher order modal fixed point logic. In Philippa Gardner and Nobuko Yoshida, editors, *Proc. 15th Int. Conference on Concurrency Theory, CONCUR'04*, volume 3170 of *LNCS*, pages 512–528, London, UK, 2004. Springer.

[VW86] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Symposium on Logic in Computer Science*, pages 332–344. IEEE Computer Society, 1986.

[WDHR06] M. De Wulf, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Antichains: A new algorithm for checking universality of finite automata. In *Proc. 18th Int. Conf. on Computer Aided Verification, CAV'06*, volume 4144 of *LNCS*, pages 17–30. Springer, 2006.

# Acknowledgment

First of all, I would like to thank my supervisor Martin Lange for his generosity in terms of time spent with me in the past few years in discussions at the whiteboard and wherever else. From this I have incredibly profited and it was also great fun. I do also think it was infinitely more successful than our attempts at fishing from the shore in Denmark which in fact is quite a weak statement.
Also, he offered me the opportunity to come around in the academic as well as the physical world from the very beginning, be it conferences, visits or extended stays abroad. Thanks in this context also to his wife Becky and the kids who made me feel very comfortable and welcome during my visits in all of the homes they inhabited since I started to work with Martin.
Thanks to Martin Hofmann for initially making it possible for me to take up a position at his chair and to get the extra funding after the expiry of the project. Thanks also for trusting me with the work as an assistant in his lectures. His universal interest in all kinds of disciplines has always been very inspiring. Sigrid Roden should also be named for her manifold virtues as the secretary of the chair.
Mogens Nielsen at BRICS is to be thanked for offering me an office during my stay there and for his help to provide me with a travel grant. Also his invaluable secretary Lene Kjeldsten did help me in every possible way to get a place to live there and whatever else was necessary. Thanks to the Friday café and all the nice people there just for doing what they do every friday.
I would also like to thank Thomas Wilke for agreeing to be the external reviewer of this thesis.
Thanks to everybody whom I forgot to thank for or did not mention, because I wish to keep this list within reasonable bounds.
Finally, thanks to Caro, who encouraged me to take up the position in the first place despite the fact that I was going to leave to Denmark for quite a while.