Numéro National de Thèse : 2021LYSEN024

Thèse de Doctorat de l'Université de Lyon

opérée par

l'École Normale Supérieure de Lyon

École doctorale InfoMaths N° 512
École doctorale en Informatique et Mathématiques de Lyon

Spécialité de doctorat : Informatique

Soutenue publiquement en Juillet 2021 par

Laureline Pinault

From automata to cyclic proofs: equivalence algorithms and descriptive complexity

Des automates aux preuves cycliques : algorithmes d'équivalence et complexité descriptive

Devant le jury composé de :

David Baelde	Maître de conférences, ENS Cachan	Rapporteur
Luigi Santocanale	Professeur des Universités, Université Aix-Marseille	Rapporteur
Delia Kesner	Professeur des Universités, Université Paris-Diderot	Examinatrice
Damien Pous	Directeur de Recherche, ENS de Lyon	Directeur de thèse
Denis Kuperberg	Chargé de Recherche, ENS de Lyon	Encadrant de thèse

Résumé

Les modèles de calcul permettent d'abstraire le fonctionnement des programmes afin de raisonner sur ceux-ci. Selon le problème étudié il est important de choisir un modèle approprié en terme d'expressivité mais aussi de propriétés. Cette thèse étudie différents modèles de calcul afin de développer des outils pour la conception et l'analyse de programmes.

Dans un premier temps nous nous intéressons au problème d'équivalence d'automates, qui a de nombreuses applications notamment dans le domaine de la vérification de programmes. Le point de départ est un algorithme coinductif exploitant les techniques up-to développé par Bonchi et Pous pour comparer des automates finis. Nous redonnons cet algorithme dans un cadre légèrement plus général afin de l'étendre aux automates de Büchi. Nous donnons aussi une version linéaire d'une sous-routine de l'algorithme initial et présentons un cadre de test utilisant de l'apprentissage d'automates pour comparer de tels algorithmes d'équivalence.

Dans un deuxième temps nous explorons le contenu calculatoire d'un système de preuves cycliques en le comparant à des modèles de calculs préexistants (automates à plusieurs têtes de lecture et système T de Gödel). Ce système de preuve correspond à un système de type cyclique pour des programmes fonctionnels. Nous montrons que si l'on se restreint aux fonctions des mots dans les booléens on obtient les langages réguliers pour le système affine et LogSpace en présence de la contraction. Sans cette restriction, le système coïncide avec système T sur les fonctions d'entiers naturels : les fonctions primitives récursives dans le cas affine et les fonctions Peano définissables en présence de la contraction.

Abstract

Computational models allow us to reason about programs by abstracting their operating process. The choice of a suitable model for a given situation takes into account both its expressivity and properties such as the complexity of the associated problems. This thesis studies computational models in order to develop tools for conception and analysis of programs.

First we consider the language equivalence problem for automata, which has various applications especially in the formal verification field. Our starting point is a coinductive algorithm HKC developed by Bonchi and Pous that exploits up-to techniques to compare finite automata. We present a version of this algorithm that works in a slighty extended setting, so we can adapt it to Büchi automata. We also give a linear version of a test used as a subroutine in HKC and a framework based on automata learning to evaluate the efficiency of equivalence algorithms.

Secondly we explore the expressivity of a cyclic proof system seen as a calculation device by comparing it with existing computational models (multiheads automata and Gödel's System T). This proof system corresponds to a type system for functional programs. If we restrict ourselves to functions from words to boolean, we get exactly the regular languages in the affine subsystem, and Logspace if we add the contraction rule. Without this restriction the system coincides with System T on natural functions: the primitive recursive functions in the affine case et the Peano definable functions with the contraction.

Contents

Introduction

We all rely on software and automated systems in various aspects of our everyday lives: communications, online transactions, transportation, finance, *etc.* Ensuring their correct behavior is crucial due to the potential human and monetary cost of an error. Moreover we often want the programs to run either as fast as possible and without using much memory (*e.g.*, mobile phone applications) or within a fixed amount of time (*e.g.*, in avionics).

These two concerns respectively correspond to computer science notions of correction and complexity. The correction of a program consists in making sure that it does what it is meant to do. To ensure it, we might, for instance, run the program with a variety of inputs. This might help identifying bugs but it cannot guarantee their absence as there often exist infinitely many possible executions of a program. As for complexity, it acts as a measure of a program's efficiency, either in terms of the time it takes to run, or in terms of the additional memory it requires. They are respectively called time and space complexities. These notions are intertwined. For instance, any memory used by the program requires at least the time it takes to write it during execution.

For the last fifty years, computer scientists have developed formal methods that can be used at different stages of the programming process in order to write reliable code: specification, development and verification. This thesis takes place in this context.

Two ingredients play an important role in the field of formal methods: models of computation that provide a framework to describe how programs operate, and logics to express the expected behaviors. We give a brief introduction to the theory of computational models and present how it interacts with logics in model checking and typed programming. Then we present the contributions of this thesis.

Models of Computation

In order to reason about programs we abstract their behavior via models of computation. We discuss two important classes of models below: abstract state machines (also called automata) and functional models.

Automata (on finite words)

In plain language, the word automaton refers to a small mechanism that can act by itself. In theoretical computer science, an automaton is an abstract machine that represents the inner operation of a process through states and transitions between them.

Even before there were computers, in the 1930's, Alan Turing developed an abstract model to study what a computing device could do and what it could not: the Turing machine. His model and conclusions still hold today. In the 1940's and 1950's, researchers considered a simpler version of it, that we nowadays call finite automata. They were originally intended to model brain functions but turned out to be extremely useful for a variety of applications ranging from finding a pattern in a text to the lexical analysis of a compiler or the design of circuits.

A *finite automaton* takes as input a *word*–a finite sequence of symbols from a given fixed alphabet–and decides whether it is accepted or not. To do so it reads the word from left to right, and navigates from states to states. The purpose of a state is to remember the relevant pieces of information gathered from computations on the input read so far, so that the automaton can decide to accept or reject the word when reaching its end. The set of input words that are accepted by an automaton is called the *language* of this automaton.

Example. A finite automaton is depicted in Figure 1. It takes a binary number as an input and accepts it if and only if it is divisible by 3. When reading a number–for instance 1001 which is the binary representation of 9–it begins in the state labeled 0. This is the initial state, which is depicted with an input arrow. Then, for each digit, reading from left to right, it follows the corresponding transition. Thus, in the case of 1001 it goes from state 0 to state 1, from state 1 to state 2, then back to state 1 and finally to state 0.

State 0 is circled twice in the picture. This is used to indicate the final states (sometimes they are also called accepting sates). So the automaton accepts the input 1001 which is indeed a multiple of 3. If the input had been 1000–binary for 8–the automaton would have ended up in state 2 and it would have been rejected.

State i, for i being 0, 1 or 2, remembers the rest of the euclidean division by 3 of the number read so far. This piece of information is sufficient to determine where the following transition should lead since reading a 0 corresponds to multiplying the number by two, and reading a 1 to multiplying it by two and adding 1.

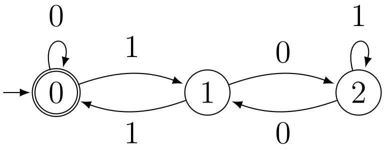


Figure 1: An automaton that recognizes the multiples of 3 given as binary numbers.

Rabin and Scott introduced non-deterministic automata: a generalization in which several transitions with the same label can originate from a given state [74]. An input is accepted if there exists a way to read it that ends up in a final state. In the case of finite automata on finite words the non-deterministic case is equivalent to the deterministic one, *i.e.* one can always find a deterministic automaton that accepts the same set of inputs as a given non-deterministic automaton. Nevertheless this model allows for far more concise representations. Indeed, for some non-deterministic automata, all their deterministic counterparts are of exponential size. This is why non-deterministic automata are the favored model in numerous applications.

In this context, the question of whether two automata are equivalent, meaning that they accept the same set of inputs, arises naturally. Figure 2 exhibits two non-deterministic automata that accept an input if and only if it contains the pattern ab (on the alphabet $\{a, b\}$). Note that both automata have 3 states, but they are not identical. For instance, the one on the right-hand side is deterministic while the one on the left-hand side is not. Checking that they nevertheless recognize the same set of words is not difficult here. The general problem, namely language equivalence of finite automata, is a classical problem in computer science with numerous applications such as compiler construction or model checking. It is also used in some algorithms that perform automata learning.

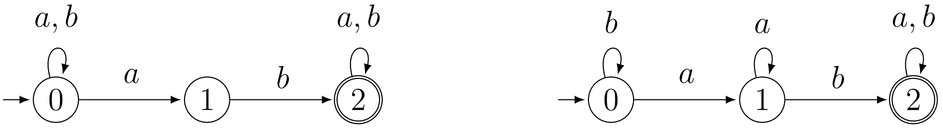


Figure 2: Two different automata that both recognize the words containing ab.

The model of finite automata has huge limitations due to the fact that it has no memory apart from a finite and determined number of possible states. For example, this makes it impossible to recognize the language of words for which there exists an integer n so they are of the shape a^nb^n (n a's followed by n b's). This would indeed require the automaton to remember how many a's it has seen in order to check that these are followed by the same number of b's. So finite automata are enough to model computing devices that operate on a very limited set of resources such as hardware circuits. But, since we sometimes need to describe more languages, we have to extend the model with additional features.

For instance, we might use a stack as a way to store information. In this model, called *pushdown automaton*, during a transition the machine can pop or push some elements to or from an auxiliary memory. Such a device can recognize the language of words of the shape a^nb^n. To do so we build an automaton that pushes a a on top of the stack as long as it reads an a on the input. Then, whenever it begins to read some b's it will makes sure that it reads the same number of b's by popping an a from the stack for each b. If at some point it either reads a a after reading some b or reads a b whereas the stack is empty, the automaton rejects the word. When reaching the end of the input, the word is accepted if and only if everything has been popped from the stack. The set of languages that are recognized by pushdown automata is called the set of *context-free languages*; it is strictly larger than the set of languages recognized by finite automata, also called the *regular languages*. Contrary to the finite automata, for this model non-determinism add actual expressive power.

Another way to gain some expressivity is to use several heads that read the input at different positions, in parallel. With this model, the *multihead automata*, it is also possible to recognize the words of the shape a^nb^n: one of the heads advances to the first b, and then they check in parallel that while the first head reads an a, the second one reads a b. These reading heads can also be allowed to read the word not only from the left to the right, but also to make transitions where they go backward in the input word: this is the *two-way multihead automata*. With this model it is possible to recognize for instance the palindromes (words that read the same forward and backward). More generally this model recognizes exactly the LogSpace languages (*i.e.* the languages that can be recognized by a Turing machine with logarithmic computing space). As for the previous model, non-deterministic multihead automata are more expressive than deterministic ones.

The most complex automata model has one head that can read the input both ways and additional memory in the form of a potentially infinite tape: it is the Turing machine. It is said to be the most complex because despite being a rather simple model it can simulate any algorithm. Other computational models, for instance a non-deterministic Turing machine with multiple heads, a random access memory-based model closer to actual computers, or some functional models that we will see in the next section, may compute faster or use less memory, but they cannot compute a function that cannot be computed by a Turing machine. This idea is called the Church-Turing thesis, after the names of the mathematicians Alonzo Church and Alan Turing. It states that a function on the natural numbers can be calculated by an effective method if and only if it is computable by a Turing machine.

Nevertheless, some problems cannot be solved even by a Turing machine, which means that there exists no algorithm that can compute a solution to the problem. The most famous one is the halting problem: there is no general algorithm that takes as input an arbitrary program and an entry for this program and decides whether the program terminates on this entry or loops forever. We say that those problems are undecidable.

One may wonder why we bother with multiple computational models when there is one that is expressive enough to simulate all others. In fact, simpler models are often easier to study and a lot of problems on automata quickly become undecidable. For instance the equivalence problem mentioned above is undecidable for pushdown automata and even the emptiness problem ("is the language of an automaton empty?") is undecidable for multihead automata.

Let us finally mention that the automaton model has been further generalized in several directions, *e.g.* probabilistic automata, weighted automata, automata on infinite trees, transducers,...). It is the subject of ongoing research to find appropriate notions of automata, and analogues of regular languages, in various settings.

Functional models

Programs are often used in order to compute *functions*. Some programming languages actually put a specific emphasis on functions: Caml, Haskell, Lisp, Erlang... are called *functional languages*. However a simple cardinality argument shows that not all functions can be represented by a program: as a program is a finite sequence of symbols, there is a countable number of programs, yet there are uncountably many functions, even when considering only functions on natural numbers.

A first attempt to capture the notion of effectively computable functions lies in the primitive recursive functions. The idea is to have some basic functions and some way to assemble them in order to create new functions. More formally, the set of primitive recursive functions is the smallest set of functions from tuples of integers to integers (each such function has a type $\mathbb{N}^d \to \mathbb{N}$ for some d) containing the constant functions (which associate to any argument a fixed number k), the projection functions (which return the i-th component of a tuple of natural numbers), and the successor (which adds 1 to its argument), and closed under the composition scheme (roughly speaking if f and g are two primitive recursive functions, then $f \circ g$ is also one) and the recursion scheme (to define a function on a natural number it suffices to express its base case and its recursive case with primitive recursive functions). For instance we can define the addition of two integers this way: the addition of m and n equals n when m is zero–so the base case is the identity (a special case of projection)–and the successor of the addition of $m - 1$ and n otherwise. So by the recursion scheme the addition is primitive recursive.

The initial functions are certainly computable and the two operations that allow to create more primitive functions can also be encoded quite directly in any programming language. Most functions that come to the mind are primitive recursive (addition, multiplication, division, minimum, maximum, factorial, exponentiation, logarithm, ...). However, the set of primitive recursive functions does not contain all possible computable functions. A famous example of such a function which is computable but not primitive recursive is Ackermann's function, whose rather simple recursive definition is given below.

$$\begin{aligned} A(0,k) &= k+1 \\ A(n,0) &= A(n-1,1) \\ A(n,k) &= A(n-1,A(n,k-1)) \end{aligned}$$

In fact primitive recursive functions are exactly the functions that can be implemented in a programming language that requires each loop to be a `for` loop: the number of times a loop runs must be specified before it begins to run. The set of primitive recursive functions can be expanded with a so-called minimization operator called μ and define the new class of general recursive functions, or μ-recursive functions. Intuitively the minimization operator searches for the smallest argument that causes a given function to return 0. The general recursive functions are exactly the functions that are computable by a Turing machines (this is one of the results that support the Church-Turing thesis).

Another equivalent class of functions are the functions definable via the λ-calculus formalism. It was introduced by Alonzo Church in 1930 in his research towards the foundation of mathematics, was studied by a number of his contemporaries including Turing, Curry, and Kleene, and is still widely used nowadays, with a lot a variants that have been developed. The main difference with the recursive functions presented above is that λ-calculus is a higher-order model: every object is a function, and functions can take other functions as arguments. Formally a *term* in λ-calculus is either a variable (noted x, y, z,...), an abstraction $\lambda x.M$ (this term represents a function that takes x as input and returns the term M), or an application $M\ N$ (this term is the application of the function M to the term N).

Example. We can define in λ-calculus various functions such as the identity and the first projection presented below. We can then manipulate them as we want, we can for instance apply one to the other.

$$id : \lambda x.x \qquad proj_1^2 : \lambda z.\lambda t.z \qquad id\ proj_1^2 : (\lambda x.x)(\lambda z.\lambda t.z)$$

A notion of computation appears naturally when we apply functions to their arguments: if we apply a function $\lambda x.M$ to a term N we want it to return the term M where the occurrences of x have been replaced by N. We say that $(\lambda x.M)N$ reduces to $M[x := N]$. In the example above, the third term $id\ proj_1^2$ reduces to $x[x := proj_1^2] = proj_1^2$. However it is possible to define terms whose reduction loops indefinitely. For instance consider the λ-term $\Omega = (\lambda x.xx)(\lambda x.xx)$. It reduces to itself, yielding an infinite sequence of reduction. Those terms correspond to non-halting Turing machines.

The integers can be encoded in λ-calculus by representing n as the function that iterates n times a given function over some variable. Then all recursive functions can be defined. It is also possible to encode the booleans thanks to the projections and then to have a function that encodes the standard `if...then ...else ...` statement. We can also add to this simple calculus some features such as types, recursors, ... This flexibility explains the success of λ-calculus in theoretical computer science.

Summary

Figure 3 sums up the different computational models we have seen so far and exhibits their relative expressiveness.

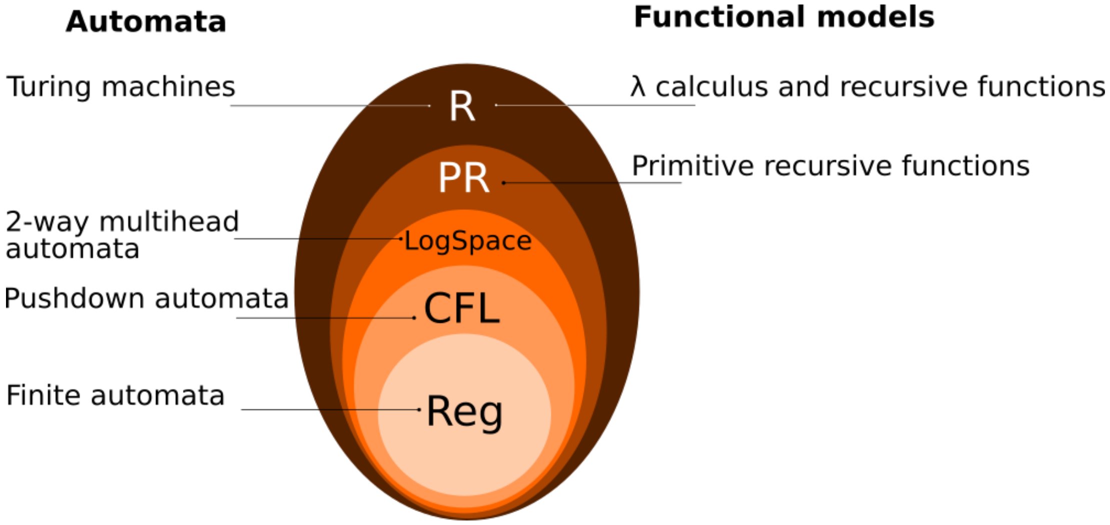


Figure 3: Relative expressiveness between some computational models.

Formal verification and model checking

Formal verification is a collection of methods and techniques to prove that certain programs behave as expected. In contrast with tests or simulations, one often uses static analyses that do not need to execute the program. Model checking is one of these methods. Given a suitable model of the program it checks that the latter meets some requirement. This process can be completely automated, which makes it appealing. Moreover usual model checking algorithms exhibit a wrong behavior when the program does not respect the specification.

Reactive systems and Büchi automata

When we think about programs, we think about computations. Computational programs run over an input in order to produce a final result, upon termination. For instance, the program which takes an integer n as input and returns $n!$ or the program which takes a graph as input and returns a minimum spanning tree of the graph, are computational programs. These programs can be modeled as functions, as seen above.

A contrario, reactive programs maintain an ongoing interaction with their environment. Those programs, whose main aim is to interact rather than to compute a value, are omnipresent: operating systems, drivers, CPUs, car controllers. . . To model these programs, we may consider that they operate on infinite inputs, which represent the infinite sequences of interactions with the environment.

Example. Figure 4 represents a simple model for a dispenser of tea and coffee. The inner states of the machine are represented as circles, and the actions the user can perform (like pressing a button, inserting a coin or taking the cup) label the edges. In state 1 the dispenser waits for the user to order either coffee or tea, and goes to state 2, or 3 accordingly. In state 2 and 3, the machine expects that some money is inserted and then moves on to either state 4 or 5 and starts pouring coffee or tea. When the cup is taken the dispenser goes back to state 1: it is ready to take another order. Whenever in state 1, 2 or 3 (not when pouring!), the user can press the off button and the machine shuts down. This is represented by state 0.

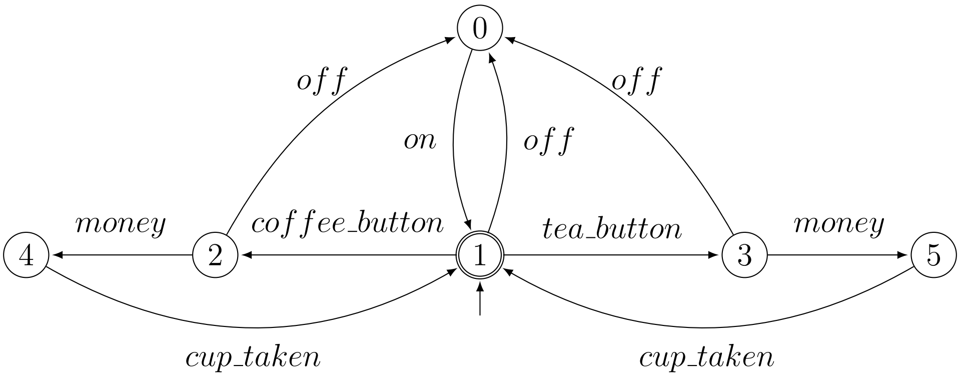


Figure 4: A simple coffee/tea dispenser model.

Examining the finite traces of these systems is often not enough as we want to be able to check properties of the shape “there will be this action at some point”. Thus, to model those systems we must adapt the automata concept to work on infinite words. The study of automata on infinite inputs was first developed by Büchi in relation with certain decision problems arising in logic. Later a strong connection has been discovered between those automata and temporal logics and nowadays they play a fundamental role in the field of formal verification.

A Büchi automaton has the same formalism and functioning as a finite automata except that it operates on infinite sequences. An input is accepted if and only if when reading it the automaton goes infinitely many often through an accepting state (represented double circled). Figure 5 shows two examples of such automata. The first one visits an accepting state when the letter a is read, so its input is accepted if and only if it contains infinitely many a's. The second one can go to an accepting state only if there are no b's left in the input so it accepts it if and only if the input contains finitely many b's. While the first automaton is deterministic, the second one is not and there exists no deterministic Büchi automaton that recognizes the same set of inputs. The fact that non-determinism adds expressivity to the model makes the usual constructions like complementation more intricate and the study of this class of automata more involved.

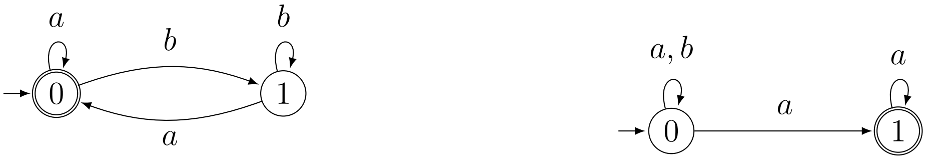


Figure 5: Two Büchi automata. On the left a deterministic one that accepts the infinite words with infinitely many a's. On the right a non-deterministic one that accepts the infinite words with finitely may b's.

As for the finite sequences case, the problem of deciding the equivalence of two given automata is a natural question. One could hope to reuse the existing algorithms for automata on finite inputs but Figures 6 and 7 exhibit two counter-examples. On Figure 6 there are two automata that are equivalent if seen as automata with finite inputs–they both recognize the words of the shape a^n for some $n \geq 1$, but the one on the left accepts no infinite sequence while the one on the right accepts the word $aaaa\ldots$ where a is repeated ad infinitum. A contrario on Figure 7 there are two automata that are equivalent if seen as automata with infinite inputs–they both accept only the word $aaaa\ldots$, but the one on the left accepts the finite sequences with an even number of a's while the one on the right accepts words with an odd number of a's.

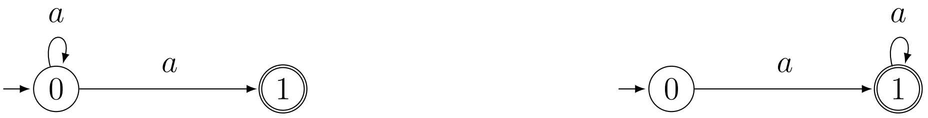


Figure 6: Two automata that are equivalent on finite inputs but not on infinite ones.

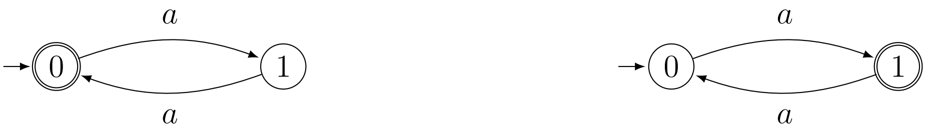


Figure 7: Two automata that are equivalent on infinite inputs but not on finite ones.

The set of languages that can be recognized by a (non-deterministic) Büchi automaton is called the set of *ω-regular languages*. As deterministic Büchi automata cannot recognize all ω-regular languages, other models of automata whose deterministic and non-deterministic counterparts both recognize the ω-regular languages have been developed by strengthening the acceptance criterion: Muller, Streett, Rabin and parity automata. In this thesis we shall focus on non-deterministic Büchi automata.

Specification checking

A specification describes the expected behavior of a system. For instance when studying the modelization of a crossroad one may require that the traffic lights cannot be green at the same time, or that there is no traffic light that eventually always stays red. Then an infinite execution may or may not respect the property. The typical properties we want to express are request-response (some action from the environment prompts some response from the system), safety (a "bad" behavior never happens), liveness (a "good" behavior eventually happens) and fairness (an event occurs repeatedly).

In order to allow for a rigorous verification, the properties need to be enunciated precisely and unambiguously through a specification language. Temporal logics have been designed to express these kind of properties: they make it possible to describe sequences of events. Besides the standard logical connectives (*not*, *and*, *or*), these logics have temporal connectives such as *next* ($\bigcirc$), *eventually* ($\Diamond$), *always* ($\Box$) and *until* ($\bigcup$). The *next* operator expresses the fact that something happens in the next temporal step; the *eventually* operator that something will happen in the future; the *always* operator that some property will always hold from now on, and the *until* operator that some property holds until a moment in the future where another property will hold.

For instance for the above example of a coffee/tea dispenser it is possible with this kind of logic to express that if the money is inserted at some point then there will be be a cup to take in the future by saying that *always* the action *money* implies that *eventually* there will be the action *cup_taken*: $\Box(\mathit{money} \rightarrow \Diamond \mathit{cup\_taken})$.

Now that we have seen how to model a program with an automaton and how to express the property we want to check thanks to temporal logic, we need a way to check that the model satisfies the property, *i.e.* to make sure that all the possible executions of the program respect the property. It turns out that there is an effective procedure transforming temporal logic formulae into Büchi automata that recognize exactly the words that respect the property. Figure 8 shows such an automaton for the formula above.

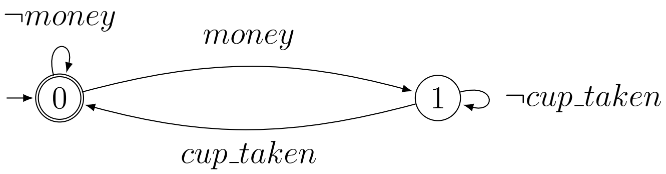


Figure 8: A Büchi automaton accepting the words that respect the property $\Box(\mathit{money} \rightarrow \Diamond \mathit{cup\_taken})$. The symbol $\neg a$ stands for 'anything but a".

Then the specification checking problem reduces to the language containment between the two automata–the one for the model and the one for the formula. This inclusion can be checked by computing the intersection between one automaton and the complement of the other. It is also closely related to the equivalence problem mentioned before, since on one hand the double inclusion ensures the equivalence ($\mathcal{A} \equiv B$ if and only if $A \subseteq B$ and $B \subseteq A$) and on the other hand the equivalence between the union of the two automata and the supposedly larger automaton holds if and only if the inclusion between the two automata holds ($A \cup B \equiv B$ if and only if $A \subseteq B$).

Remark. It is not possible to write all the desired properties with the method exposed above. For instance, one cannot express that at any point of the execution of the tea/coffee dispenser it is possible to shutdown the machine (*i.e.* take the off action). This requires to look at all possible executions at once. For this, we represent the possible executions as an infinite tree (the unfolding of the automaton) and have a logic that takes into account the various paths: CTL which stands for Computation Tree Logic. As the objects we consider are different, the checking algorithms are also different. Moreover those logics are not comparable in term of expressivity. We already saw that some properties are not expressible in the (linear) temporal logic described previously, and the converse is also true. For instance properties of the form "eventually always something" which express a form of stability cannot be expressed in CTL.

Correction a priori: Typed programming and Curry-Howard isomorphism

When we develop the program independently of its specification and verify the latter afterwards, there is a high chance that the program does not operate as it ought to. Then the verification and debugging phase can be long, navigating back and forth between the programmer and the verification module. Another approach to this problem consists in trying to directly develop it in such a way that it must behave according to its specification.

A deep connection between proof systems found in the field of formal logic and typed calculi developed in computational theory, namely the Curry-Howard correspondence, has been the theoretical ground for progress in this direction. Tools coming from this theory are currently implemented into software.

(Functional) Typed programming

If one uses a typed language, the syntax itself prevents the programmer to write code that would result in a type error in the execution. A similar untyped program could be proved to share the same property, but it has to be done individually for each program while with a typed language it is certified for every code that could be written.

The notion of type can be added to the λ-calculus we have seen in previous sections. A type is either a base type belonging to a fix set, or constructed from other types with an arrow such as $\tau \to \sigma$ where τ and σ represent arbitrary types. The type $\tau \to \sigma$ is the type of a function that takes an argument of type τ and returns an argument of type σ.

For example $id = \lambda x.x$ has the type $\tau \to \tau$ with τ representing an arbitrary type and $proj_1^2 = \lambda z.\lambda t.z$ has the type $\tau \to \sigma \to \tau$ (the same as $\tau \to (\sigma \to \tau)$) with τ and σ being arbitrary types because if it takes an argument of type τ and then an argument of type σ it returns the first one of type τ.

$$\mathit{id}\,\frac{}{\Gamma, x:\tau \vdash x:\tau} \qquad \to\text{-}i\,\frac{\Gamma, x:\tau\vdash M:\sigma}{\Gamma\vdash \lambda x.M:\tau\to\sigma} \qquad \to\text{-}e\,\frac{\Gamma\vdash M:\tau\to\sigma \quad \Gamma\vdash N:\tau}{\Gamma\vdash MN:\sigma}$$

Figure 9: Typing rules for simply typed λ-calculus.

In order to establish the type of more complex terms we use the inference rules that are shown in Figure 9. They all have the following shape: a horizontal line separates the conclusion (below the line) form the set of premises (above the line). Conclusions and premises read the same way: to the left of $\vdash$ there is the context, *i.e.* a list of typing assignations that we assume to be known, and to the right the typing assignation we want to assert. The first rule has no premise and means that from the context where a variable has some type, we can assert that this variable has this type, it will be used as an axiom. The second rule has one premise and means that if when assuming that a variable has some type τ we can assert that a term has some type σ in some context, then the function that takes this variable and returns this term has type $\tau \to \sigma$ in this context. Lastly the third rule has two premises and means that if in some context a term has some arrow type $\tau \to \sigma$ and another term has the type τ then we can apply the first to the second one and the result has type σ in this context.

Then we can stack those rules and build what we call a typing derivation for a λ-term in the form of a tree. Figure 10 shows how we can use it to derive for example the type of $\mathit{id}\ \mathit{proj}_1^2$. We find that is it indeed the type of proj_1^2.

$$\to\text{-}e\ \frac{\to\text{-}i\ \dfrac{\mathit{id}\ \dfrac{}{x:\tau\to\sigma\to\tau\vdash x:\tau\to\sigma\to\tau}}{\vdash \lambda x.x:(\tau\to\sigma\to\tau)\to(\tau\to\sigma\to\tau)} \qquad \to\text{-}i\ \dfrac{\to\text{-}i\ \dfrac{\mathit{id}\ \dfrac{}{z:\tau,y:\sigma\vdash z:\tau}}{z:\tau\vdash\lambda y.z:\sigma\to\tau}}{\vdash\lambda z.\lambda y.z:\tau\to\sigma\to\tau}}{\vdash(\lambda x.x)(\lambda z.\lambda y.z):\tau\to\sigma\to\tau}$$

Figure 10: Typing derivation of the λ-term $\mathit{id}\ \mathit{proj}_1^2$.

This type system prevents from writing typable terms whose reduction would loop indefinitely. For example the λ-term $\Omega = (\lambda x.xx)(\lambda x.xx)$ seen above cannot be typed. This result is known as the strong normalization of the simply typed λ-calculus: any evaluation of a typed λ-term is a *finite* sequence of reduction, meaning that the programs they represent all terminate. The simply typed λ-calculus is thus a proper fragment of the general λ-calculus.

The simply typed λ-calculus can be extended with new base types, such as an integer type, or new ways of combining types, such as the product of two types. Those new types come with new constructors for the λ-terms and new typing rules. For instance *System T* is a formal system defined for simply typed λ-calculus by adding a type for natural numbers and a recursion constructor over it.

Formal systems in logic and Curry-Howard isomorphism

Logic is the science that studies the correctness of arguments, *i.e.* the relations that lead to the acceptance of a proposition (the conclusion) on the basis of hypotheses (the premises). In everyday life we often use words as *hence*, *so*, *therefore* and so on to point out the steps of our reasoning. In mathematics and computer science there is a need to be more formal to assess the soundness of an argument, and this is where we use the notions of formal logics and proof systems. One may think that there exists only one "true" logic. However, not only there exist many formalisms for one reasoning system, but there also exist different reasoning systems. First there exist no formal systems that would encompass all possible human reasoning on everything because it would be too dense to be studied. Then different logics can be developed to talk about different things. For instance we have seen in the previous sections the temporal logics that are useful to describe facts about infinite sequences. Secondly the classical logic principles, even if easily acceptable by our intuition and suited for most everyday life and mathematics deductive patterns, are not the only possible principles and it can be argued that for some contexts, especially in computer science, they are not well-suited. For instance the principle saying that for any proposition P, P or not P holds (known as the excluded middle principle) raises some issues. It allows for example to prove the fact that there exists two irrational numbers x and y such that x^y is rational: either $\sqrt{2}^{\sqrt{2}}$ is rational and we can take $x = y = \sqrt{2}$ or it is not and we can take $x = \sqrt{2}^{\sqrt{2}}$ and $y = \sqrt{2}$. However the problem with this proof is that we do not know which of the two numbers is rational: it is a non-constructive proof. Logic that reject this principle are often called intuitionistic or constructive and have immediate links with computation as we will see below. Another interesting example is linear logic. Linear logic is obtained by forbidding structural rules (contraction and weakening): meaning that to prove a result one has to use each of its hypotheses exactly once. As intuitionistic logic, it is constructive. It can be seen as a logic manipulating resources that cannot be duplicated or thrown away at will and as such it has numerous applications in fields such as programming languages, game semantics, quantum physics and linguistics.

Formal logics are defined by a language to write the propositions and a proof system to deduce the validity of the propositions. The language is usually given by a set of constructors. For instance the classical and intuitionistic logics both use the connectives *or* ($\vee$), *and* ($\wedge$), *not* ($\neg$) and implication ($\rightarrow$). A proof system is a collection of inference rules, that describe atomic steps of reasoning, and a way to combine them. For instance the rules in Figure 11 are inference rules. Their formalism is close to the one of the typing rules from previous section: the horizontal line separates the conclusion (below the line) from the premises (above the line); conclusion and premises contain on the left of the $\vdash$ symbol a set of propositions that are assumed to be *true* (the context), and on its right, the proposition to prove. The first rule has no premise and means that when we assume some proposition P to hold, we can prove that P holds. The second rule has one premiss and means that if when assuming P to be *true* we can prove Q to be *true* as well, then we can prove that P implies Q. Lastly the third rule has two premises and means that if in some context in some context we can prove that P holds and that P implies Q, then in this context we can prove Q. In general for each logical connective there are one or several introduction rules and one or several elimination rules. The second and third rules of Figure 11 are respectively the introduction and elimination rules for the implication. The system can also include some structural rules. As for typing systems those rules can be stacked to form a proof tree as shown in Figure 12.

$$ {\scriptstyle ax}\,\dfrac{}{\Gamma, P \vdash P} \qquad\qquad {\scriptstyle \to\text{-}i}\,\dfrac{\Gamma, P \vdash Q}{\Gamma \vdash P \to Q} \qquad\qquad {\scriptstyle \to\text{-}e}\,\dfrac{\Gamma \vdash P \to Q \quad \Gamma \vdash P}{\Gamma \vdash Q} $$

Figure 11: Some inference rules of a proof system.

The similarity between proofs systems and type systems extends beyond a mere coincidence. In fact, if we take the typing derivation for $id\ proj_1^2$ (Figure 10), “forget” about the λ-terms and keep only the types we get exactly a proof of the proposition $P \to Q \to P$, as pictured in the left of Figure 12 (the types τ and σ have respectively been renamed P and Q). Despite being valid, this proof is not the simplest one. The detours in such a proof come from the fact that some introduction rules are directly followed by the corresponding elimination rule. We can rewrite the proof by eliminating them: this is called proof normalization. When applying this method on the proof tree on the left of Figure 12, we obtain the proof tree on the right. This is an alternative proof of $P \to Q \to P$ and it corresponds exactly to the typing derivation of $proj_1^2$ that has the same type as $id\ proj_1^2$.

$$ {\scriptstyle \to\text{-}e}\,\dfrac{{\scriptstyle \to\text{-}i}\,\dfrac{{\scriptstyle id}\,\dfrac{}{P \to Q \to P \vdash P \to Q \to P}}{\vdash (P \to Q \to P) \to (P \to Q \to P)} \quad {\scriptstyle \to\text{-}i}\,\dfrac{{\scriptstyle \to\text{-}i}\,\dfrac{{\scriptstyle id}\,\dfrac{}{P, Q \vdash P}}{P \vdash Q \to P}}{\vdash P \to Q \to P}}{\vdash P \to Q \to P} \qquad {\scriptstyle \to\text{-}i}\,\dfrac{{\scriptstyle \to\text{-}i}\,\dfrac{{\scriptstyle ax}\,\dfrac{}{P, Q \vdash P}}{P \vdash Q \to P}}{\vdash P \to Q \to P} $$

Figure 12: Two different proofs of $P \to Q \to P$.

The Curry-Howard isomorphism, also known as the ”propositions-as-types” paradigm, formalizes this link. It comes from the observation that an implication $A \to B$ corresponds to a type of function from A to B because on one hand applying a given function to an element of A to get an element of B resembles to applying an assumption $A \to B$ with the hypothesis A to prove B and on the other hand an application from A to B can be viewed as a procedure that transform proofs of A into proofs of B. Similarly, the conjunction corresponds to the Cartesian product, the disjunction to the union and the negation to the complement. Thus formulae correspond to types and proofs correspond to programs. Yet this correspondence is not simply a bijection between proofs and programs but a real isomorphism because the computational notions existing on both sides (proof normalization and program reduction) correspond as well.

This correspondence comes from an observation of Curry in 1934 and has been formalized by Howard in 1969 in the context of intuitionistic natural deduction and simply typed λ-calculus. Since, it has been extended to other frameworks. Virtually all proof-related concepts can be interpreted in terms of computations and vice versa. For example different proofs formalisms (Hilbert style, natural deduction, sequent calculi) are mirrored by different computational models (combinatory logic, λ-calculus, explicit substitution calculi). Moreover, by enriching the logic we enrich as well the type system. For instance first order logic with quantification corresponds to dependent types, second order logic corresponds to polymorphic types, and proofs by contradiction in classical logic to control operators such as exceptions.

Over the years, the Curry-Howard isomorphism evolved from a theoretical tool to analyze proofs and programs to a practical way of developing software systems supporting program verification and computer-assisted reasoning. For instance, the Coq proof assistant builds on this correspondence.

Fixed point logics and cyclic proof systems

A fixed point of a function f is an element x such that $f(x) = x$. A function can have zero, one, or several fixed points. In the context of a (partially) order set, the notion of least fixed point (a fixed point that is lower than any other fixed point) and greatest fixed point (a fixed point that is greater than any other fixed point) may prove useful. If they exist they are unique.

In particular, when f is a monotone function over a complete lattice–an partially ordered set in which all subsets have a supremum (and an infimum), then f admits a least and a greatest fixed point (this result is part of Knaster-Tarski's theorem).

Least and greatest fixed points can be used to model respectively inductive and coinductive datatypes.

For instance if we want to define a list of elements from a given set A we will say that it is either an empty list (represented as a constant, usually named `Nil`) or a list built from an element from A (the head) and another list (the tail). To define this list type in an OCaml-like syntax we could write:

```
type list= Nil | Cons of A * list
```

Let us consider the operator f that maps a set X to $\{\texttt{Nil}\} \cup \{\texttt{Cons}(a, l) \mid a \in A,\ l \in X\}$. Then the set of lists that we aim to define is a fixed point of this operator. More precisely it is the least fixed point of this operator. In terms of type we will note $\texttt{list} = \mu x.(1 + A \cdot x)$. Here 1 represent is a type of a singleton, that represent the constant `Nil`, $+$ represents disjoint union, $\cdot$ Cartesian product and x is a type variable. The operator μ is the least fixed point operator: $\mu x.(1 + A \cdot x)$ designates the least fixed point of $x \mapsto 1 + A \cdot x$.

Dually we can define coinductive datatypes with greatest fixed points. For instance, the set of streams over the set A (*i.e.* the set of infinite sequences of elements of A) corresponds to the greatest fixed point $\nu x.A \cdot x$.

Those inductive and coinductive datatypes could also be given as primitive types of the language. For instance a lot of programming languages and computational models, such as Gödel's system T mentioned mentioned, consider the type of natural numbers as a primitive type although it could be defined with a least fixed point (a natural number is either 0 or the successor of a natural number). In such a setting, the language will then be restricted to those inductive and coinductive datatypes that are predefined. On the other hand, considering the possibility to define new types with fixed points the same way we can define new types with union or Cartesian products offers more flexibility.

Moreover it allows to naturally define recursive functions over those datatypes. For instance to define the `append` function that concatenates two lists, we can proceed as follow: either the first list is empty and we return the second list, or it contains a least one element (the head) and we build the result from this element and the list returned by a recursive call of `append` on the tail and the second list. In Ocaml-like syntax it could be written as:

```
let rec append list1 list2 = match list1 with
  | Nil -> list2
  | Cons x tail -> Cons x (append tail list2)
```

As discussed in the previous section, everything that can be defined on types can also be defined in logic. In particular we can consider logics enriched with fixed points.

For instance, if we look back at the temporal logic LTL defined previously in this introduction, we can notice that we can express some of the connectors in terms of fixed points. For the "eventually" operator $\Diamond$, we can say that a property is eventually true if either it is true at this moment or it is eventually true at the next step. We can express this with a least a fixed point: $\Diamond A = \mu X.(A \vee \bigcirc X)$. Dually, a property is always true if it is true at this moment and will still be always true in the next step: $\Box A = \nu X.(A \wedge \bigcirc X)$.

As with types and programming, logics with fixed points allow for more expressivity than their counterparts with predefined connectors. For instance modal μ-calculus is an extension of propositional logic with modalities and fixed points, and it encompasses many temporal logics including linear temporal logic (LTL) and computational tree logic (CTL). Fixed points have also been investigated in the context of linear logic, giving rise to μMALL.

Now the issue is to design inference rules to handle such constructors in proofs. The first systems designed to reason about formulae with fixed points were finite proof systems. The inference rules for least of greatest fixed points were respectively reflecting the induction and coinduction principles. The drawback is that, with those principles, to eliminate a least fixed point or introduce a greatest fixed point we need to provide an explicit invariant, much in the same way we need an invariant to prove the correctness of a recursive function. This invariant is a formula that has to be expressed in the same language (which is not always possible) but can be syntactically unrelated to the formula we want to prove. This breaks the *subformula* property: this property ensures that if there is a proof of a result then there exists a proof of this result such that every formula in a conclusion or a premiss of an inference rule is a subformula of the result. Such a property is usually sought in the design of a proof system because it is a key point to automatize proof search.

Remark. The fact that such systems do not respect the subformula property is closely linked with the fact that the induction and coinduction rules hide a *cut* rule that cannot be eliminated.

For these reasons, infinite proofs have been introduced to study formulae with fixed points. The inference rules for least and greatest fixed points are basically an unfolding of the fixed point, and they can be stacked an infinite number of times, giving rise to an infinite proof tree. These proofs have two problems.

The first problem is that infinite proofs are, infinite objects. In order to have a way to represent them and handle them algorithmically we have to restrict the system to a fragment of infinite proofs that can be finitely represented: regular (or cyclic, or circular) proofs. The regular proofs are the proof trees that contain only finitely many distinct subtrees. They can be represented as graphs with backpointers rather than trees, hence the name of cyclic or circular proofs.

The second problem is that not every infinite proof tree built from the rules is a valid proof. Intuitively it is possible to just add rules that ”do nothing” ad infinitum and derive false conclusions. To fix this problem, those systems introduce a validity criterion: a proof is an infinite tree of inference rules that respect some property. In general, this property requires that for each infinite branch of the tree, there exists a formula that goes through an infinite number of unfoldings. Such criteria are non-local, yet they are syntactic and decidable for regular proofs (there is an algorithm that can decide whether a regular infinite proof tree is a valid proof or not).

An example of such a cyclic proof is depicted in Figure 13. The formula A^* corresponds to the type of the lists over A: $A^* = \mu X.1 + A \cdot X$ where 1 is the type of the empty list. The rule labeled with $*$-l is the unfolding of this fixed point: a list is either empty, or composed of an element from A and another list. The rule $*$-$r_{::}$ corresponds to a constructor for this type: if we have an element and a list we can construct a list from them. Lastly, the *cut* rule corresponds to the composition of functions.

On this example, there is only one infinite branch in the tree we obtain by unfolding the finite representation of the proof: the branch that always takes the backpointer. On this branch, the A^* formula that goes infinitely many often through an unfolding is depicted with a purple highlight.

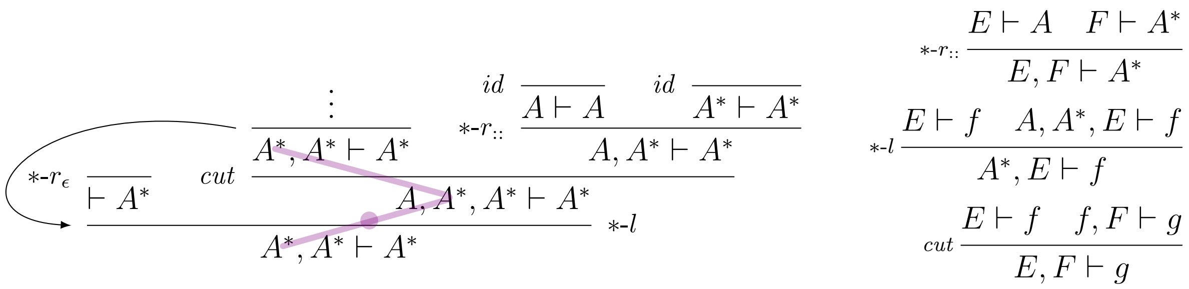
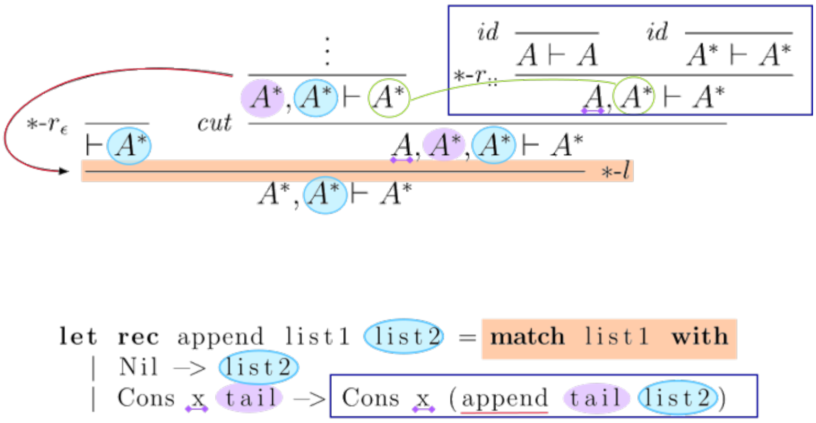


Figure 13: Example of a cyclic proof.

The formalism of this proof has many similarities with the formalism for the definition of the list type in a functional language. In fact, if we see this proof as a computing device in a Curry-Howard manner, we can remark that it corresponds exactly to the `append` function defined above. The Figure 14 highlights the corresponding parts between the proof and the code of the function.



```
let rec append list1 list2 = match list1 with
   | Nil -> list2
   | Cons x tail -> Cons x (append tail list2)
```

Figure 14: Correspondence between the cyclic proof of Figure 13 and the code of the `append` function. The mirrored element are highlighted in the same color in order to visualize the relationship between them.

These links between cyclic proofs and recursive functions will be studied in Chapter 4, as a first step towards extending the Curry-Howard framework to cyclic proof systems.

Bibliographic notes

We gave in this introductory part a brief introduction to various subjects that are studied in computer science. For readers that would like to develop their knowledge on these topics we recommend the following reference books that have been used as support to write this introduction:

- For the theory of finite automata on finite words, “Introduction to Automata Theory, Languages, and Computation” by Hopcroft and Ullman [53] and “Elements of the theory of computation” by Lewis and Papadimitriou [65].

- For the primitive recursive and μ-recursive functions, “Computability and logic” by Boolos, Burgess and Jeffrey [15], especially chapter 6.

- For the model checking techniques, “Principles of Model Checking” by Baier and Katoen [9].

- For the theory of automata over infinite structures, especially infinite words, “Infinite words : automata, semigroups, logic and games” by Perrin and Pin [71], “Automata on Infinite Objects” by Thomas [85], and “Automata: From Logics to Algorithms” by Vardi and Wilke [89].

- For the λ-calculus, with or without types, and its links with formal proof systems, “Lectures on the Curry-Howard Isomorphism” by Sorensen and Urzyczyn [79], “Proofs and Types” by Girard [43], and “Type Theory and Functional Programming” by Thompson [86].

Contributions and outline

In the first chapter (Chapter 1), we recall the coinductive algorithm defined by Bonchi and Pous to check equivalence of finite automata on finite words [14], and we propose a linear algorithm for the congruence check required by this algorithm. We also propose a technique to test such equivalence algorithms using the automata learning framework of Angluin [6].

In the second chapter (Chapter 2), we show how to adapt the aforementioned algorithm to non-deterministic Büchi automata, i.e., automata on infinite words. This is a non-trivial task and we build on a construction due to Calbrix Nivat and Podelski [21] making it possible to reduce the problem to the case of finite word automata. This work was presented at DLT'19 [59].

Then we move to cyclic proofs. The starting point is a proof system inspired from linear logic (IMALL) [27], that makes it possible to reason about *Kleene star* in a cyclic way rather than by induction (Kleene star is an iteration operation closely related to while loops in programming languages and reflexive transitive closure in mathematics, which is by essence inductive). In this thesis, we study the computational content of such systems.

We first study a specific fragment making it possible to represent formal languages (Chapter 3). This fragment is *cut-free*: the cut rule intuitively corresponding to modus ponens is absent. In the general case, we show that we obtain exactly the DLogSpace languages, by establishing an equivalence with a model of jumping multihead automata: automata with several reading heads and the ability to relocate heads. If we further remove the *contraction* rule from the system (a rule making it possible to reuse hypotheses), we show that we obtain exactly the regular languages. This work was presented at FSTTCS'19 [60].

We finally consider the whole system (Chapter 4). We establish a correspondence with Gödel's System T [45] (a simply-typed lambda-calculus with an iterator for natural numbers or lists). In the affine case, i.e., without contraction, we obtain precisely the primitive recursive functions. In the general case we obtain precisely the first-order functions of system T. In order to obtain prove those theorems, we use tools from reverse mathematics. This work was presented at PoPL'21 [61].

Notation

We write $\mathbb{N}$ for the set $\{0, 1, 2, \dots\}$ of natural numbers. If $i \leq j$ are natural numbers, we write $[i; j]$ for the set $\{i, i+1, \dots, j\}$ and $[i; j[$ for the set $[i; j-1]$.

We denote sets by capital letters $X, Y, S, T \dots$ and functions by lower case letters $f, g, \dots$ Given sets X and Y, $X \times Y$ is their Cartesian product, $X \uplus Y$ is the disjoint union, X^Y is the set of functions $f \colon Y \to X$. The collection of subsets of S is denoted by $\mathcal{P}(S)$. We note $\langle x, y, z \rangle$ for tuples, and $Im(f)$ for the image of a function f. We write 1 for the singleton set $\{\langle\rangle\}$.

We write $\mathbb{B}$ for the set $\{0, 1\}$ (Booleans). The truth values 1 and 0 will also be sometimes noted **tt** and **ff** for *true* and *false* respectively.

The collection of relations on S is denoted by $Rel(S) = \mathcal{P}(S^2)$. Given a relation $R \in Rel(X)$, we write $x \mathrel{R} y$ for $\langle x, y \rangle \in R$.

Given a finite alphabet A, we write A^* for the set of all finite words over A; ϵ for the empty word; w_1w_2 for the concatenation of words $w_1, w_2 \in A^*$; $|w|$ for the length of a word w, and w_i for its i^{th} letter (when $i < |w|$). We write A^+ for the set of non-empty words over A and A^ω for the set of infinite words over A. We usually let a, b range over the letters of such an alphabet A.

We also use the notation X^* for sequences over an arbitrary, possibly infinite, set X. In this case, if l is such a sequence, we sometimes write $l(i)$ for its i^{th} element. We use commas to denote concatenation of both sequences and tuples, ϵ to denote the empty sequence, and $x :: q$ to denote the addition of an element x in front of a sequence q.

Chapter 1

Equivalence algorithms for finite automata

1.1 Introduction

Checking language inclusion or language equivalence of finite automata is a classical problem in computer science with numerous applications such as compiler construction or model checking.

For deterministic automata this problem can be solved either for all pairs of states of the automata at once via minimization [51] or for a given pair of state with Hopcroft and Karp's algorithm [52]. In this chapter and the following we focus on the second problem. Indeed non-deterministic (Chapter 1) and Büchi (Chapter 2) automata have no canonical notion of minimal automata, so the first approach is not available..

Two families of algorithms were discovered for non-deterministic automata on finite words, which drastically improved over the pre-existing ones in practice: antichain-based algorithms [91, 3, 35] and algorithms based on bisimulations up to congruence [14]. In both cases, those algorithms explore the starting automata by resolving non-determinism on the fly through the powerset construction, and they exploit subsumption techniques to avoid the need to explore all reachable states (which can be exponentially many). The algorithms based on bisimulations up to congruence improve over those based on antichains by strengthening the coinduction principle used implicitly by antichain algorithms with the older technique for deterministic automata, due to Hopcroft and Karp. They however require a membership test in the congruence closure of a relation at each step. This test is performed in quadratic time in the size of the relation in [14].

Note that both families of algorithms require exponential space (and time) in worst-case complexity, for a problem which is only PSpace. However the theoretical comparison of the complexity of the algorithm is not very informative and in practice both families perform better than existing PSpace algorithms, because the latter require exponential time even for best cases.

We first recall the algorithms from [14] for checking equivalence of automata on finite words (Section 1.2). We extend the presentation of the algorithms to Moore machines that generalize automata with non-boolean outputs in order to prepare the ground for chapter 2. Then we present a variant of Dowling and Gallier's algorithm that carries out the congruence test in linear time in the size of the relation (Section 1.3). We also give some leads to improve further this test. Lastly we present a framework based on automata learning in order to evaluate and compare the efficiency of the algorithms that check language equivalence of finite automata (Section 1.4).

1.2 Coinductive algorithms for finite automata

We will work with *Moore machines*, which generalize finite automata by allowing output values in an arbitrary set rather than Booleans: they recognize *weighted languages.* We keep the standard automata terminology for the sake of readability.

A deterministic finite automaton (DFA) over the alphabet A and with outputs in O is a triple $\langle S, o, t\rangle$, where S is a finite set of states, $o\colon S \to O$ is the output function, and $t\colon A \times S \to S$ is the transition function which returns, for each letter $a \in A$ and for each state x, the next state $t_a(x)$. Note that we do not specify an initial state in the definition of DFA: rather than comparing two DFAs, we shall compare two states in a single DFA (obtained as disjoint union if necessary).

Every DFA $\mathcal{A} = \langle S, o, t\rangle$ induces a function $[\cdot]_{\mathcal{A}} : S \to O^{A^*}$, mapping each state to a weighted language with weights in O. This function is defined by $[x]_{\mathcal{A}}(\epsilon) = o(x)$ for the empty word, and $[x]_{\mathcal{A}}(aw) = [t_a(x)]_{\mathcal{A}}(w)$ otherwise. We shall omit the subscript $\mathcal{A}$ when it is clear from the context. For a state x of a DFA, $[x]$ is called the language accepted by x. The languages accepted by some state in a DFA with Boolean outputs are the *rational languages.*

1.2.1 Deterministic automata: Hopcroft and Karp's algorithm

We fix a DFA $\langle S, o, t\rangle$. Coinductive algorithms for checking language equivalence proceed by trying to find a *bisimulation* relating the given starting states.

Definition 1.1 (Bisimulation)**.** Let $g\colon \mathit{Rel}(S) \to \mathit{Rel}(S)$ be the function on relations defined as

$$g(R) = \{\langle x, y\rangle \mid o(x) = o(y) \text{ and } \forall a \in A,\ t_a(x)\ R\ t_a(y)\}$$

A *bisimulation* is a relation R such that $R \subseteq g(R)$.

The above function g being monotone (i.e., it preserves the inclusion ordering), it admits the union of all bisimulations as a greatest fixpoint, by Knaster-Tarski's theorem [56, 84]. This greatest-fixpoint is actually language equivalence:

Theorem 1.1. *For all $x, y \in S$, $[x] = [y]$ iff there is a bisimulation R with $x\ R\ y$.*

This theorem yields two families of algorithms: on the one hand, backward algorithms like partition-refinement [51] make it possible to compute the largest bisimulation, and thus to minimize DFAs; on the other hand, forward algorithms make it possible to compute the smallest bisimulation containing a given pair of states (if any), and thus to check language equivalence locally, between two states [52]. The latter problem is the one we are interested in here. (Unlike with languages of finite words, there is no canonical notion of minimal automaton for Büchi automata.) For deterministic automata on finite words

```
input  : A DFA $\mathcal{A} = \langle S, o, t \rangle$ and two states $x, y \in S$
output : true if $[x]_{\mathcal{A}} = [y]_{\mathcal{A}}$; false otherwise

1 $R := \emptyset$; $todo := \{\langle x, y \rangle\}$;
2 while $todo \neq \emptyset$ do
      // invariant: $\langle x, y \rangle \in R \cup todo \wedge R \subseteq g(f(R \cup todo))$
3     extract $\langle x', y' \rangle$ from $todo$;
4     if $o(x') \neq o(y')$ then return $false$;
5     if $\langle x', y' \rangle \in f(R \cup todo)$ then skip; // back to the beginning of the
       loop
6     forall $a \in A$ do
7         insert $\langle t_a(x'), t_a(y') \rangle$ in $todo$;
8     insert $\langle x', y' \rangle$ in $R$;
9 return $true$; // because: $\langle x, y \rangle \in R \subseteq g(f(R))$
```

Figure 1.1: Coinductive algorithm for language equivalence in a DFA; the function f on line 5 ranges over the identity for the naive algorithm ($\texttt{Naive}(\mathcal{A}, x, y)$) or e for Hopcroft & Karp's algorithm ($\texttt{HK}(\mathcal{A}, x, y)$).

this problem is slightly easier complexity-wise: when the starting automaton has size n, minimization can be performed in time $o(n\ln(n))$ while language equivalence of two given states can be tested in almost linear time [83].

A preliminary algorithm for checking language equivalence of two states $x, y \in S$ is obtained as follows: try to complete the relation $\{\langle x, y \rangle\}$ into a bisimulation, by adding the successors along all letters and checking that o agrees on all inserted pairs. This algorithm is described in Figure 1.1; it is quadratic in the worst case since a pair of states is added to the relation R at each iteration. The standard and almost linear algorithm by Hopcroft and Karp [52, 83], can be seen as an improvement of this naive algorithm where one searches for bisimulations up to equivalence rather than plain bisimulations:

Definition 1.2. Let $e \colon Rel(S) \to Rel(S)$ be the function mapping a relation R to the least equivalence relation containing R. A *bisimulation up to equivalence* is a relation R such that $R \subseteq g(e(R))$.

This coarser notion makes it possible to take advantage of the fact that language equivalence is indeed an equivalence relation, so that one can skip pairs of states whose equivalence follows by transitivity from the previously visited pairs. The soundness of this technique is established by the following Proposition:

Proposition 1.1 ([14, Thm. 1])**.** If R is a bisimulation up to equivalence, then $e(R)$ is a bisimulation.

Complexity-wise, when looking for bisimulations up to equivalence in a DFA with n states, at most n pairs can be inserted in R in the algorithm in Figure 1.1: at the beginning, $e(R)$ corresponds to a discrete partition with n equivalence classes; at each iteration, two classes of $e(R)$ are merged.

Note that Hopcroft and Karp's algorithm proceeds forward and computes the smallest bisimulation up to equivalence containing the starting pair of states, if any. As mentioned above, this contrasts with partition-refinement algorithms [51], which proceed backward: they start with a coarse partition (accepting v.s. non-accepting states), which they refine by reading transitions backward.

1.2.2 Non-deterministic automata: HKC

Definition 1.3. A semilattice is a tuple $\langle O, +, 0\rangle$ where O is a set of elements, $+\colon O^2 \to O$ is an associative, commutative and idempotent binary operation, and $0 \in O$ is a neutral element for $+$. For instance, $\langle \mathbb{B}, \max, 0\rangle$ is a semilattice. More generally $\langle \mathcal{P}(X), \cup, \emptyset\rangle$ is a semi-lattice for every set X.

A *non-deterministic finite automaton* (NFA) over the alphabet A and with outputs in O is a triple $\langle S, o, t\rangle$, where S is a finite set of states, $o\colon S \to O$ is the output function, and $t\colon A \times S \to \mathcal{P}(S)$ is the transition function which returns, for each letter $a \in A$ and for each state x, a set $t_a(x)$ of potential successors. Like for DFA, we do not specify a set of initial states in the definition of NFA.

We fix an NFA $\langle S, o, t\rangle$ in this section and we assume that the set O of outputs is a semilattice. Under this assumption, an NFA $\mathcal{A} = \langle S, o, t\rangle$ can be transformed into a DFA $\mathcal{A}^\# = \langle \mathcal{P}(S), o^\#, t^\#\rangle$ using the well-known powerset construction:

$$o^\#(X) = \sum_{x\in X} o(x) \qquad\qquad t_a^\#(X) = \bigcup_{x\in X} t_a(x)$$

This construction makes it possible to extend the function $[\cdot]$ into a function from sets of states of a given NFA to weighted languages. It also gives immediately algorithms to decide language equivalence in NFA: just use algorithms for DFA on the resulting automaton. Note that when doing so, it is not always necessary to compute the whole determinized automaton beforehand. For instance, with coinductive algorithms like in Figure 1.1, the determinized automaton can be explored on the fly. This is useful since this DFA can have exponentially many states, even when restricting to reachable subsets.

Formally, when doing so, the function g is defined as in Section 1.2.1, but with respect to the determinized DFA with state space $\mathcal{P}(S)$, so its type is $\mathit{Rel}(\mathcal{P}(S)) \to \mathit{Rel}(\mathcal{P}(S))$:

$$g(R) = \left\{\langle X, Y\rangle \mid o^\#(X) = o^\#(Y) \text{ and } \forall a \in A,\ t_a^\#(X)\ R\ t_a^\#(Y)\right\}$$

The key idea behind the HKC algorithm [14] is that one can actually do better than Hopcroft and Karp's algorithm by exploiting the semilattice structure of the state-space of automata determinized through the powerset construction. This is done using *bisimulations up to congruence.*

Definition 1.4. Let $R \in \mathit{Rel}(\mathcal{P}(S))$. We say that R is a *congruence relation* if it is an equivalence relation such that XRY and $X'RY'$ entail $(X \cup X')R(Y \cup Y')$ for all $X, X', Y, Y' \in \mathcal{P}(S)$.

Definition 1.5. Let $c\colon \mathit{Rel}(\mathcal{P}(S)) \to \mathit{Rel}(\mathcal{P}(S))$ be the function mapping a relation R to the least congruence relation containing R. A *bisimulation up to congruence* is a relation R such that $R \subseteq g(c(R))$.

Proposition 1.2 ([14, Thm. 2])**.** If R is a bisimulation up to congruence, then $c(R)$ is a bisimulation.

```
input  : A NFA $\mathcal{A} = \langle S, o, t\rangle$ and two sets of states $X, Y \subseteq S$
output : a relation $R$ such that $[X] = [Y]$ iff $\forall\langle X', Y'\rangle \in R,\ o^{\#}(X') = o^{\#}(Y')$

1 $R := \emptyset$; $todo := \{\langle X, Y\rangle\}$;
2 while $todo \neq \emptyset$ do
      // invariant:  $\langle X, Y\rangle \in R \cup todo \wedge R \subseteq g'(c(R \cup todo))$
3     extract $\langle X', Y'\rangle$ from $todo$;
4     if $\langle X', Y'\rangle \in c(R \cup todo)$ then skip; // back to the beginning of the
       loop
5     forall $a \in A$ do
6         insert $\langle t_a^{\#}(X'), t_a^{\#}(Y')\rangle$ in $todo$;
7     insert $\langle X', Y'\rangle$ in $R$;
8 return $R$;
```

Figure 1.2: HKC'$(\mathcal{A}, X, Y)$: computing a pre-bisimulation up to congruence in a NFA.

Checking whether a pair of sets belongs to the congruence closure of a finite relation can be done algorithmically (see Section 1.3). The algorithm HKC [14] is obtained by running the algorithm from Figure 1.1 on $\mathcal{A}^{\#}$, replacing the function f on line 5 with the congruence closure function c. We provide a variant of this algorithm in Figure 1.2, where we prepare the ground for the algorithms we will propose for Büchi automata in Chapter 2. There, we only explore the transitions of the determinized automaton, leaving aside the verification that the output function agrees on all pairs (the test on line 4 in Figure 1.1 has been removed). This corresponds to using a function g' instead of g, defined as

$$g'(R) = \left\{\langle X, Y\rangle \mid \forall a \in A,\ t_a^{\#}(X)\ R\ t_a^{\#}(Y)\right\}$$

Indeed, while this verification step is usually done on the fly in order to fail faster when a counter-example is found (as in Figure 1.1, line 4), it will be useful later to perform this step separately.

As mentioned in the Introduction, the advantage of HKC over HK is that in practice it often makes it possible to skip reachable subsets from the determinized automaton, even when the algorithm answers positively, thus achieving substantial gains in terms of performance: there are families of examples where it answers positively in polynomial time even though the underlying minimal DFA has exponential size. Actually it can also improve exponentially over the more recent antichain-based algorithms [14, Section 4]. These latter gains can be explained by the fact that we focus on language equivalence rather than language inclusion: while the two problems are inter-reducible (e.g., $[X] \subseteq [Y]$ iff $[X \cup Y] = [Y]$), working with equivalence relations makes it possible to strengthen the coinductive argument used implicitly by both algorithms.

1.3 Computing the congruence closure

For the algorithm to be effective, we need a way to check whether some pairs belong to the congruence closure of some relation over the set of states (line 4 in Figure 1.2). Generating the congruence closure explicitly is intractable. Even when the relation is empty its congruence closure contains a pair for each possible set by reflexivity. Moreover it is not necessary because during the execution of HKC we only need to know if some pairs belong to the congruence closure. Thus, we are interested in the problem of membership within the congruence closure of a given relation. In [14], the authors present a quadratic solution based on a rewriting system. Below we present another solution based on the satisfiability of a propositional formula that is linear in the size of the relation. Such a method was used in [24], where the authors extend HKC to alternating automata. Their notion of congruence is slightly different and results in a membership problem that is NP-complete. We show below a variant of Dowling and Gallier's algorithm [32, 76] that solves the problem in linear time. We also give some leads to improve the test at the end of this subsection.

1.3.1 Logical closure

Given a relation $R \in \mathit{Rel}(\mathcal{P}(S))$, we define the following formula over $|S|$ boolean variables $\{x_s \mid s \in S\}$

$$\varphi_R = \bigwedge_{\langle X,Y\rangle \in R} \lfloor X \rfloor \;\Leftrightarrow\; \lfloor Y \rfloor \qquad\qquad \text{where } \lfloor X \rfloor = \bigvee_{s \in X} x_s.$$

Definition 1.6. We define the *logical closure* of a relation to be the following function on $\mathit{Rel}(\mathcal{P}(S))$:

$$\mathit{cl}(R) = \{\langle X_0, Y_0\rangle \mid \varphi_R \Rightarrow (\lfloor X_0 \rfloor \Leftrightarrow \lfloor Y_0 \rfloor) \text{ is valid}\}$$

Proposition 1.3. For all $R \in \mathit{Rel}(\mathcal{P}(S))$, we have $c(R) = \mathit{cl}(R)$.

Proof. For all $R \in \mathit{Rel}(\mathcal{P}(S))$, $\mathit{cl}(R)$ is a congruence relation that contains R, so that $c(R) \subseteq \mathit{cl}(R)$.

For the converse inclusion, consider $\langle X_0, Y_0\rangle \notin c(R)$. To show that $\langle X_0, Y_0\rangle \notin \mathit{cl}(R)$ we will construct an assignment such that φ_R evaluates to true but $\lfloor X_0 \rfloor \Leftrightarrow \lfloor Y_0 \rfloor$ evaluates to false. As $c(R)$ is an equivalence relation compatible with the union, we can turn $\mathcal{P}(S)/c(R)$ into a semilattice where we lift the union operation: $\lfloor X \rfloor \sqcup \lfloor Y \rfloor = \lfloor X \cup Y \rfloor$. We obtain a quotient semilattice $\langle \mathcal{P}(S)/c(R), \sqcup, [\emptyset]\rangle$ with partial order: $T \sqsubseteq T' \Leftrightarrow T \sqcup T' = T'$. The equivalence classes $[X_0]$ and $[Y_0]$ are distinct in the quotient semilattice. Without loss of generality assume that $[X_0] \not\sqsubseteq [Y_0]$. Since S is finite, $[X_0]$ is the join of a finite set of join-irreducible elements. Therefore, there must exist join-irreducible Z such that $Z \sqsubseteq [X_0]$ and $Z \not\sqsubseteq [Y_0]$.

Now we can construct $f : \mathcal{P}(S) \to \{0, 1\}$ defined by $f(X) = 1$ if and only if $Z \sqsubseteq [X]$. Since Z is join-irreducible, for any X and Y, $f(X \cup Y) = 1$ if and only if $f(X) = 1$ or $f(Y) = 1$ so f restricted to singletons is an assignment and the evaluation of the formula $\lfloor X \rfloor$ for this assignment is exactly $f(X)$. For any $\langle X, Y\rangle \in R$, $f(X) = f(Y)$ because $[X] = [Y]$ so φ_R evaluates to 1 under this assignment but $f(X_0) = 1 \neq f(Y_0) = 0$ by construction of Z so $\lfloor X_0 \rfloor \Leftrightarrow \lfloor Y_0 \rfloor$ evaluates to false under this assignment. In the end, $\langle X_0, Y_0\rangle \notin \mathit{cl}(R)$. □

Thus the membership problem of a pair of sets of states in the congruence closure of a given relation reduces to the problem of validity of a propositional formula. This problem is in general coNP-complete but the shape of the formula, which is very close to a Horn formula, makes it possible to use a variant of Dowling and Gallier's algorithm [32, 76] and solve the problem in linear time.

Definition 1.7. A *literal* is either a propositional variable (a positive literal) or the negation of a propositional variable (a negative literal).

A *Horn clause* is a disjunction of literals that contains at most one positive literal. Alternatively a Horn clause can be seen as an implication which left side is a (possibly empty) conjunction of positive literals and which right side is empty or reduced to a single positive literal. A *Horn formula* is a conjunction of Horn clauses.

We call *factorized Horn clause* a disjunction of negative literals and *a* (possibly empty) conjunction of positive literals. It is a natural extension of Horn clauses in the sense that it can be seen as an implication between (possibly empty) conjunctions of positive literals. If we expand a factorized Horn clause according to the conjunction of positive literals we get several Horn clauses, hence the name. A factorized Horn clause will be said *proper* if it contains positive and negative literals. A *factorized Horn formula* is a conjunction of factorized Horn clauses.

Example 1.1. $\varphi = (x \vee \neg y \vee \neg z) \wedge (u \vee \neg y \vee \neg z)$ is a Horn formula with two Horn clauses. It is equivalent to the factorized Horn formula $\varphi' = (x \wedge u) \vee \neg y \vee \neg z$ composed of a unique factorized Horn clause. φ and φ' can also respectively be written $(y \wedge z \Rightarrow x) \wedge (y \wedge z \Rightarrow u)$ and $(y \wedge z) \Rightarrow (x \wedge u)$.

Factorized Horn clauses that are not proper, *i.e.* that are reduced to a conjunction of positive literal or a disjunction of negative literals can also be written with implications. For instance $x \wedge u$ and $\neg y \vee \neg z$ correspond respectively to $\top \Rightarrow (x \wedge u)$ and $(y \wedge z) \Rightarrow \bot$.

Proposition 1.4. The problem of membership in the congruence closure of a relation reduces linearly to the satisfiability of two factorized Horn formulae and quadratically to the satisfiability of two Horn formulae.

Proof. The size of the problem is the sum of the size of all the considered sets (the ones in R and the ones we are testing).

Let us recall the meaning of $\lfloor X \rfloor$ and introduce $\lceil X \rceil$ its symmetrical notation:

$$\lfloor X \rfloor = \bigvee_{s \in X} x_s \qquad\qquad \lceil X \rceil = \bigwedge_{s \in X} x_s$$

By Proposition 1.3 the problem to know whether some pair $\langle X_0, Y_0 \rangle$ belongs to the congruence closure of some relation R reduces to the validity of a formula

$$\varphi = \Big(\bigwedge_{\langle X,Y \rangle \in R} (\lfloor X \rfloor \ \Leftrightarrow \ \lfloor Y \rfloor) \Big) \Rightarrow (\lfloor X_0 \rfloor \Leftrightarrow \lfloor Y_0 \rfloor)\,.$$

As $\lfloor X \rfloor$ is a disjunction over the elements of X, the size of φ is exactly the size of the problem. With a factor 4 we get as an equivalent problem the validity of the two formulae

$$\begin{aligned}
\varphi_1' &= \Big(\bigwedge_{\langle X,Y\rangle \in sym(R)} (\lfloor X\rfloor \ \Rightarrow\ \lfloor Y\rfloor)\Big) \Rightarrow (\lfloor X_0\rfloor \Rightarrow \lfloor Y_0\rfloor) \\
\text{and } \varphi_2' &= \Big(\bigwedge_{\langle X,Y\rangle \in sym(R)} (\lfloor X\rfloor \ \Rightarrow\ \lfloor Y\rfloor)\Big) \Rightarrow (\lfloor Y_0\rfloor \Rightarrow \lfloor X_0\rfloor) \\
\text{where } sym(R) &= R \cup \{\langle Y,X\rangle \mid \langle X,Y\rangle \in R\}\,.
\end{aligned}$$

As the two formulae have exactly the same shape we will only treat φ_1' in what follow. Such a formula is valid if and only if $\neg\varphi_1'$ is not satisfiable, where:

$$\begin{aligned}
\neg\varphi_1' &= \neg\left(\Big(\bigwedge_{\langle X,Y\rangle \in sym(R)} (\lfloor X\rfloor \ \Rightarrow\ \lfloor Y\rfloor)\Big) \Rightarrow (\lfloor X_0\rfloor \Rightarrow \lfloor Y_0\rfloor)\right) \\
&= \bigwedge_{\langle X,Y\rangle \in sym(R)} (\lfloor X\rfloor \Rightarrow \lfloor Y\rfloor) \wedge \neg(\lfloor X_0\rfloor \Rightarrow \lfloor Y_0\rfloor) \\
&= \bigwedge_{\langle X,Y\rangle \in sym(R)} (\neg\lfloor Y\rfloor \Rightarrow \neg\lfloor X\rfloor) \wedge \neg\lfloor Y_0\rfloor \wedge \lfloor X_0\rfloor.
\end{aligned}$$

Without changing the satisfiability problem we can invert all positive and negative literals and we obtain a factorized Horn formula

$$\begin{aligned}
\psi_1 &= \bigwedge_{\langle X,Y\rangle \in sym(R)} (\lceil Y\rceil \Rightarrow \lceil X\rceil) \wedge \lceil Y_0\rceil \wedge \bigvee_{s\in X_0} \neg x_s \\
&= \bigwedge_{\langle X,Y\rangle \in sym(R)} (\lceil Y\rceil \Rightarrow \lceil X\rceil) \wedge (\top \Rightarrow \lceil Y_0\rceil) \wedge (\lceil X_0\rceil \Rightarrow \bot).
\end{aligned}$$

Then the pair $\langle X_0, Y_0\rangle$ belongs in the congruence closure of R if and only if the two factorized Horn formulae ψ_1 and ψ_2 (obtained from φ_2' the same way), whose sizes are linear in the size of the original problem, are both non-satisfiable.

By expanding ψ_1 and ψ_2 with respect to $\lceil X\rceil = \bigwedge_{s\in X} x_s$ we obtain Horn formulae whose sizes are quadratic in the original problem. □

Example 1.2. Let us consider the relation $R = \{\langle\{x\},\{u\}\rangle, \langle\{y,z\},\{u\}\rangle\}$ over $\mathcal{P}(\{x,y,z,u\})$ and the pair $\langle\{x,y\},\{u\}\rangle$.

The formula that encodes the relation is $\varphi_R = [x \Leftrightarrow u] \wedge [(y \vee z) \Leftrightarrow u]$.

The pair $\langle\{x,y\},\{u\}\rangle$ belongs to the congruence closure of the relation if and only if the formulae

$$\begin{aligned}
\psi_1 &= C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge (\top \Rightarrow x \wedge y) \wedge (u \Rightarrow \bot) \\
\text{and } \psi_2 &= C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge (x \wedge y \Rightarrow \bot) \wedge (\top \Rightarrow u)
\end{aligned}$$

are both non-satisfiable, where C_1, C_2, C_3 and C_4 are four factorized Horn clauses coming from the relation:

$$C_1 : x \Rightarrow u \qquad C_2 : u \Rightarrow x \qquad C_3 : (y \wedge z) \Rightarrow u \qquad C_4 : u \Rightarrow (y \wedge z).$$

Note that the disjunction $(y \vee z)$ in φ_R gave rise to conjunction $(y \wedge z)$ in ψ_1 and ψ_2. Note also that only the clauses coming from the relation are proper factorized Horn clauses whereas the clauses coming from the pair are degenerated.

The pair belongs to $c(R)$. Indeed, by transitivity and reflexivity we respectively know that $\langle \{x\}, \{y, z\} \rangle$ and $\langle \{y\}, \{y\} \rangle$ belongs to $c(R)$, so by union $\langle \{x, y\}, \{y, z\} \rangle \in c(R)$ and we can conclude by transitivity again that $\langle \{x, y\}, \{u\} \rangle \in c(R)$. ψ_1 and ψ_2 are indeed unsatisfiable. To further reduce to Horn formula it suffices to expand C_4 as $(u \Rightarrow y) \wedge (u \Rightarrow z)$.

1.3.2 A variant of Dowling and Gallier's algorithm

Dowling and Gallier presented two algorithms that solve the satisfiability problem of Horn formulae in linear time [32]. The idea is that because the clauses contain at most one positive literal, the constraints on the assignments of variables are stronger and can be represented as a graph. The difference between the two algorithms lies in the strategy used to resolve the data flow problem on the graph, either bottom-up or top-down. We present below their first algorithm and explain how to adapt it in order to handle factorized Horn formulae. The presentation differs slightly from the original paper [32] in order to prepare the ground for the variant we propose.

The algorithm (called DG in what follows) is listed in Figure 1.3. The idea will be to identify the variables that need to be assigned to *true* and propagate the information looking to Horn clauses as propagation rules. To illustrate how the algorithm works let us take the formula ψ_1 from Ex.1.2. Initially the variables x and y are assigned to *true* because of the clause $\top \Rightarrow x \wedge y$. With the clause $C_1 : x \Rightarrow u$ we deduce that u should also be assigned to *true* which conflicts with the clause $u \Rightarrow \bot$.

DG starts with a preliminary assignment *val* where all variables evaluate to $false$ (line 3). At any point of the algorithm, the current assignment is given by the array *val*. Then the algorithm will identify the variables that have to be assigned to *true* and propagate the information. The list *todo* contain the variables that have to be assigned to *true* but which consequences have not been treated yet[1]. It is initialized with the variables belonging to some clause without negative literal (line 5). Then whenever we have a Horn clause $\bigwedge x_i \Rightarrow y$ whose negative literals $x_i's$ have already all be assigned to *true*, y is added to *todo* (lines 14-15). Those consequences are derived from the clauses until there is no more or it reaches a contradiction, that is a clause without positive literal such that all its negative literals have been assigned to *true* (lines 12-13).

It remains to explain how the algorithm proceeds to achieve this in linear time in the size of the formula. The cleverness of the algorithm lies in the pre-computation of the list of clauses where the variable appears negatively (*clauses*, line 2) and the maintenance of a counter *nb_neg*. When a variable is assigned to *true* during the execution of the algorithm, the array *clauses* points to the clauses whose *nb_neg* has to be decreased. A clause 'activates' when its *nb_neg* goes down to zero and its positive literal is assigned to *true* if there is one, the algorithm returns false otherwise.

Proposition 1.5. The algorithm DG runs in linear time in the size of the input formula, *i.e.* in the total number of literals that appear in the input formula.

[1] In [32], clauses are added to this list, here we directly handle the positive variables of the clauses.

```
input  : A Horn formula $F = C_1 \wedge \cdots \wedge C_k$ over the set of variables $\{x_1, \ldots, x_n\}$
         $neg(C_j)$ is the set of negative variables of a clause
         $pos(C_j)$ is the positive variables of a clause if any, $\perp$ otherwise
output : true if $F$ is satisfiable; false otherwise

// pre-treatment
forall $i \in [\![1; n]\!]$ do
    $clauses[i] := \{j \mid x_i \in neg(C_j)\}$;

// algorithm
forall $i \in [\![1; n]\!]$ do $val[i] := false$;
forall $j \in [\![1; k]\!]$ do $nb\_neg[j] := |neg(C_j)|$;
$todo := \{pos(C_j) \mid nb\_neg[j] = 0\}$;
while $todo \neq \emptyset$ do
    // invariant: $C_j$ evaluates to $false$ under $val$ if and only if $\exists i$
        s.t. $x_i = pos(C_j)$, $val[i] = false$ and $i \in todo$.
    extract $i$ from $todo$;
    if $val[i] = true$ then skip;
    $val[i] := true$;
    forall $j \in clauses[i]$ do
        $nb\_neg[j] := nb\_neg[j] - 1$;
        if $nb\_neg[j] = 0$ and $pos(C_j) = \perp$ then
            return $false$;
        if $nb\_neg[j] = 0$ then
            insert $pos(C_j)$ in $todo$;
return $true$;
```

Figure 1.3: DG (F): satisfiability of Horn formulae in linear time.

Proof. The pre-treatment is linear because it suffices to go through each clause and update the arrays *clauses*, *nb_neg* and *pos* accordingly.

A variable is inserted in *todo* only when $nb\_neg[j]$ goes down to 0 for some clause C_j, which can only happen once per clause in the algorithm, so the while loop is executed at most k times, where k is the number of clauses. During the whole execution of the algorithm, a given clause C_j can see its counter $nb\_neg[j]$ decremented at most n_j times where n_j is the number of negative literals of C_j. Indeed it can only happen when one of its negative literals has its value set to *true*, which happens at most once for each variable during the whole execution. In total the forall loop is executed at most $\sum_{j \in [\![1,k]\!]} |C_j| = |F|$ times. □

By Proposition 1.4 we can use the algorithm DG to solve the problem of membership in the congruence closure in quadratic time. We explain below how to slightly modify this algorithm in order to handle directly factorized Horn formulae, still in linear time. The idea is that when all the negative literals x_i of a factorized Horn clause $\bigwedge x_i \Rightarrow \bigwedge y_i$ are true, then all its positive literals y_i are assigned to true. The difference with DG lies in the fact that we have only one clause with one counter where DG would have dealt with

```
input  : A factorized Horn formula $F = C_1 \wedge \cdots \wedge C_k$ over the set of variables
         $\{x_1, \ldots, x_n\}$
         $neg(C_j)$ and $pos(C_j)$ are the set of negative and positive variables of a clause
output : true if $F$ is satisfiable; false otherwise

   // pre-treatment
1  forall $i \in [\![1, n]\!]$ do
2      $clauses[i] := \{j \mid x_i \in neg(C_j)\}$;

   // algorithm
3  forall $i \in [\![1, n]\!]$ do $val[i] := false$;
4  forall $j \in [\![1, k]\!]$ do $nb\_neg[j] := |neg(C_j)|$;
5  $todo := \bigcup\{pos(C_j) \mid nb\_neg[j] = 0\}$;
6  while $todo \neq \emptyset$ do
       // invariant: $C_j$ evaluates to $false$ under $val$ if and only if $\exists i$
           s.t. $x_i \in pos(C_j)$, $val[i] = false$ and $i \in todo$.
7      extract $i$ from $todo$;
8      if $val[i] = true$ then skip;
9      $val[i] := true$;
10     forall $j \in clauses[i]$ do
11         $nb\_neg[j] := nb\_neg[j] - 1$;
12         if $nb\_neg[j] = 0$ and $pos(C_j) = \emptyset$ then
13             return $false$;
14         if $nb\_neg[j] = 0$ then
15             forall $i' \in pos(C_j)$ do insert $i'$ in $todo$;
16 return $true$;
```

Figure 1.4: DG' (F): satisfiability of factorized Horn formulae in linear time.

several Horn clauses (one for each y_i), every one with its own counter.

To illustrate this, let us take the formula ψ_2 from Ex.1.2. Initially the variable u is assigned to *true* because of the clause $\top \Rightarrow u$. With the clause $C_2 : u \Rightarrow x$ we deduce that x should also be assigned to *true*; with the clause $C_4 : u \Rightarrow (y \wedge z)$ we deduce that both y and z have to be assigned to *true*; this eventually conflicts with the clause $(x \wedge y) \Rightarrow \bot$.

The modified algorithm DG' is listed in Figure 1.4. Its input is a factorized Horn formula F consisting of a disjunction of factorized Horn clauses $C_1, \ldots, C_k$. The list *todo* is initialized with the union of the sets of the positive variables of clauses that do not contain negative literals (line 5). When the counter of a clause reaches 0, we insert in *todo* all its positive variables that have not already be assigned to *true* (lines 14-15).

Proposition 1.6. The algorithm DG' is correct and runs in linear time in the size of the input formula.

Proof. If we expand the formula F to get a Horn formula we get for each factorized Horn clause C_j and each x_i positive literal of C_j a Horn clause C_j^i. For any j all the C_j^i share the same negative literals so when we run DG on the expanded formula they are all inserted in *todo* at the same time, as does `DG'` in line 17.

The same analysis as for `DG` in the proof of Proposition 1.5 shows that `DG'` runs in linear time in the size of the input formula. □

We presented in this subsection an algorithm that carries out the membership test in the congruence closure of a relation in linear time in the size of the relation. A lead to improve the test is to add a union-find structure that takes care of the test of membership in the equivalence closure of the relation in quasi-constant amortized time before moving to the congruence test, if necessary. This structure could in addition help accelerate the congruence test itself by adding to the rules used by the modified Dowling and Gallier algorithm the ones corresponding to the equivalence closure of the relation. Yet this structure has a counterpart in terms of complexity for the algorithm because it has to be maintained, and increases the size of the input of `DG'`, so the gain is not obvious and may depend on the cases.

Moreover we can take advantage of the fact that this test is a subroutine of another algorithm, `HKC`. During the execution of the whole algorithm this membership test is carried out each time a new pair of set of states is encountered. The pairs are *a priori* different each time but the relation itself changes very little between each step: either the precedent pair has been skipped and it is identical, or it has not and it contains one additional pair. A first optimization is to modify the pre-treatment part in order to update the array *clauses* rather than re-computing it from scratch. Besides, with ideas similar to incremental SAT solving [50, 90, 36], we can reuse information from the previous solving steps to test membership of a pair in the relation. We can see the algorithm `DG'` as taking a set of variables that have to be *true* and deriving the consequences according to some rules until it reaches a contradiction according to a given set of variables that have to be $false$ or there are no more consequences to derive. The set of variables that have to be *true* corresponds to factorized Horn clauses containing no negative literal; the rules corresponds to the clauses containing positive and negative literals (proper clauses); the set of variables that have to be $false$ corresponds to factorized Horn clauses containing no positive literal. The factorized Horn formulae obtained from a pair and a relation always have the same structure: each pair of the relation corresponds to two rules – *i.e.* two clauses with positive and negative literals; the sets of the pair correspond to the sets of variables that have to be *true* or $false$. Each time the algorithm tests a new pair it derives a new rule: the set of variable that are assigned to *true* at the end of the algorithm is a consequence derived from the set of variables initially assigned to *true*. This rule can be added so part of future work might be skipped. However this increases the number of clauses of the formula to look at, making each step possibly longer.

1.4 Evaluating the efficiency of equivalence algorithms via automata learning

In this section we explore a way to assess the efficiency of algorithms that compare automata. As this problem is PSpace-complete the worst case complexity is well known but might not be representative of the most frequent cases. It is also complicated to analyze and compare the average-case theoretical complexity because there is no natural notion of random distribution over automata nor pairs of automata. Nevertheless we can try to evaluate the efficiency of the algorithms through executions on collections of automata. As collections of real automata – obtained for instance from model checking problems – are quite rare, those benchmarks are usually reinforced by tests on randomly generated automata. However, even if there exist some good models to generate a random automaton with a non-trivial language [80], two randomly generated NFAs have high chances not to be equivalent and to be separated by a short counter-example. In this context most equivalence algorithms stop quickly the exploration when finding the counter-example and return it. The differences between algorithms can more often be observed on pairs of equivalent automata or "close" ones. To circumvent this difficulty the authors of [14] have generated automata without accepting states so that they all recognize the empty language. Yet this solution is a bit artificial and not satisfactory. We can indeed expect two automata that recognize the same language to have some hidden structural links that some of the algorithms may be able to exploit, but with this solution the automata are *a priori* completely independent. We present here another solution based on algorithms that learn automata.

Automata learning has received much focus these last years. It consists in trying to guess an automaton that would recognize an hidden and unknown regular language which can only be accessed through some queries. In [6] Angluin develops $\mathtt{L}^\star$, an algorithm that learns the minimal DFA for a regular language by interacting with an oracle that can answer membership and equivalence queries. Several extensions build upon this algorithm, such as Bollig et al. that adapts it to learn a non-deterministic automaton [13]. Those algorithms start with basic membership queries (are the empty word and the letters in the language?) to construct a first simple hypothesis automaton; then they alternate equivalence queries (is the current hypothesis correct?) with phases of membership queries. When the oracle answers negatively to an equivalence query it also gives a counter example, the algorithms use the counter example alongside some additional membership queries to correct and update their current hypothesis. The membership and equivalence queries are considered as blackboxes by the algorithms. If the membership queries can usually be implemented easily, it is not the case for equivalence queries.

In the case where the target language is given as an automaton, any equivalence algorithm can be used to answer to the equivalences queries. Then the given equivalence algorithm is tested on a sequence of pairs of automata whose languages are intuitively closer and closer, until being the same. It has the advantage of testing the algorithms on a wide range of pairs of automata in term of closeness of recognized language. Moreover one of the main applications of equivalence algorithms is model checking, and in this context automata that are compared come from different modelings (one from the program and one from the logical specification) so they may look very different, but they usually recognize "close" languages when they are not equivalent because we search for non-trivial bugs. Here by "close" we mean that witnesses of non-inclusion are difficult to build. So this sequence of pairs of automata produced by the learning framework allows to test on more realistic cases than totally random pairs.

However as the automata that are produced by the learning process are not arbitrary but have some particular structure due to the way they are generated, this method can also introduce some biases. We present below the learning framework for finite automata, either deterministic or non-deterministic and for each case we explain how it can interact with equivalence algorithms.

1.4.1 Deterministic automata: $\mathtt{L}^\star$

We fix $\mathcal{L}$ a regular language to learn, over alphabet Σ. A residual of $\mathcal{L}$ is a language that can be written $u\backslash\mathcal{L} = \{v \in \Sigma^* \mid uv \in \mathcal{L}\}$ for some $u \in \Sigma^*$. By the Myhill-Nerode theorem, the number of residuals of $\mathcal{L}$ is finite and there is a natural bijection between the minimal DFA that recognize $\mathcal{L}$ and its residuals.

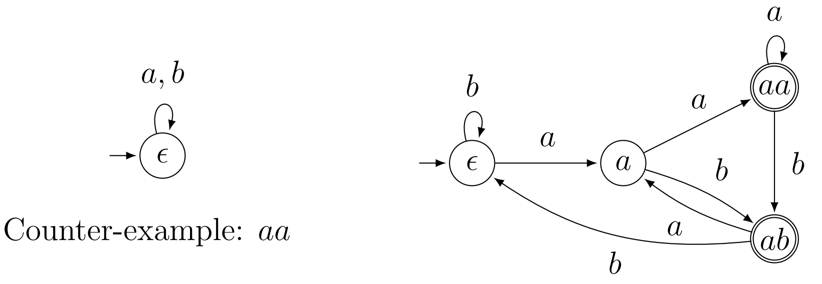
The algorithm $\mathtt{L}^\star$ developed by Angluin in [6] aims at building this minimal automaton by determining those residuals. It begins with the residual corresponding to the empty word ε which will be the initial state of the automaton. It asks membership queries about ε and the different letters so it can establish the acceptance status of this state and the out-going transitions and then construct a first hypothesis automaton. When a counter-example is given by the oracle in response to an equivalence query, the algorithm distinguishes some residuals thanks to the prefixes of the counter-example. Then it fills up the transition table of the updated automaton with additional membership queries. Note that by construction, this algorithm builds the minimal automaton because the states corresponds to the residuals of the target language. We present below an example, for the more technical details we refer the reader to [6].

Example 1.3. Let us consider the language $\mathcal{L} = \Sigma^* a \Sigma$ over the alphabet $\Sigma = \{a, b\}$. At the beginning of the algorithm, the hypothesis automaton contains only one state that corresponds to the residual $\epsilon^{-1}\mathcal{L}$. As ϵ is not in the language, this state is not accepting. In order to define the outgoing transitions, the algorithm asks the membership status of the words a and b which are both rejected. So the algorithm identifies the residuals $\epsilon\backslash\mathcal{L}$, $a\backslash\mathcal{L}$ and $b\backslash\mathcal{L}$ and submits the automaton depicted in Figure 1.5a.

The hypothesis automaton does not recognize the target language and the oracle gives aa as a counter example: it should be accepted but it is not by the hypothesis automaton. This counter-example distinguishes two residuals: a belongs to $a\backslash\mathcal{L}$ but not to $\epsilon\backslash\mathcal{L}$. Then the algorithm launches an iterative process of multiple membership queries in order to complete the table in Figure 1.5d; each time it finds a new residual that must be distinguished from others, it determines outgoing transitions from the corresponding state. At the end of this subroutine we obtain the table in Figure 1.5d and the equations in Figure 1.5c where the equalities corresponds to an equality of the columns in Figure 1.5d. From the table and the equations we deduce the second hypothesis automaton depicted in Figure 1.5b: one column of the table corresponds to a state and the transitions follow from the equations. On this example the algorithm stops here because the second guessed automaton recognizes the target language.

If the target language would have been $\mathcal{L}' = \Sigma^* a \Sigma + bbb$ for instance, all membership queries would have been answered in the same way so the algorithm would have submitted the same hypothesis automata but it would not have stopped after the second equivalence query, and would have carried on processing the counter example bbb.

By taking a random NFA as a model for the target regular language, we can use any equivalence algorithm to answer the equivalence queries. However this would always test the algorithms in a very restricted case since one of the automata would always be



(a) First hypothesis automaton.

(b) Second and final hypothesis automaton.

$$\begin{aligned} b\backslash\mathcal{L} &= \epsilon\backslash\mathcal{L} \\ aaa\backslash\mathcal{L} &= aa\backslash\mathcal{L} \\ aab\backslash\mathcal{L} &= ab\backslash\mathcal{L} \\ aba\backslash\mathcal{L} &= a\backslash\mathcal{L} \\ abb\backslash\mathcal{L} &= \epsilon\backslash\mathcal{L} \end{aligned}$$

(c) Identification of residuals.

	$\epsilon\backslash\mathcal{L}$	$b\backslash\mathcal{L}$	$abb\backslash\mathcal{L}$	$a\backslash\mathcal{L}$	$aba\backslash\mathcal{L}$	$aa\backslash\mathcal{L}$	$aaa\backslash\mathcal{L}$	$ab\backslash\mathcal{L}$	$aab\backslash\mathcal{L}$
ϵ	*no*	*no*	*no*	*no*	*no*	*yes*	*yes*	*yes*	*yes*
a	*no*	*no*	*no*	*yes*	*yes*	*yes*	*yes*	*no*	*no*
aa	*yes*	*yes*	*yes*	*yes*	*yes*	*yes*	*yes*	*yes*	*yes*

(d) Membership in residual languages.

Figure 1.5: $\mathtt{L}^\star$ learning $\mathcal{L} = \Sigma^* a \Sigma$

deterministic. This is why we investigate the variant proposed by Bollig et al. [13] in the next subsection.

We can furthermore note that this usage of $\mathtt{L}^\star$ and an equivalence algorithm together gives rise to a way to find the equivalent minimal DFA for a given NFA without going through the powerset construction.

1.4.2 Non-deterministic automata: $\mathtt{NL}^\star$

In [13] Bollig et al. adapt $\mathtt{L}^\star$ in order to learn non deterministic automata; they call their algorithm $\mathtt{NL}^\star$. The global process is the same but the hypothesis automata are NFAs. In order to achieve this, the authors note that some states of the minimal DFA can be seen as the union of other states. We illustrate this idea on the example of the previous subsection.

Example 1.4. Let us consider again the language $\mathcal{L} = \Sigma^* a \Sigma$ from Example 1.3. If we look at the table of Figure 1.5d we can note that the column $aa\backslash\mathcal{L}$ is the union of the column $\epsilon\backslash\mathcal{L}$, $a\backslash\mathcal{L}$ and $ab\backslash\mathcal{L}$. In this case $\mathtt{NL}^\star$ will not consider $aa\backslash\mathcal{L}$ as a new state. In fact, to construct an NFA, $\mathtt{NL}^\star$ will take as states the residuals that are *prime* i.e., that are not union of others. Transitions can be recovered via column inclusion. For instance, the columns $\epsilon\backslash\mathcal{L}$ and $a\backslash\mathcal{L}$ are included in column $aba\backslash\mathcal{L}$ so by reading a from the state ab we go to the states ϵ and a. The NFA constructed from the table in Figure 1.5d is depicted in Figure 1.6; it recognizes the target language.

With this variant we can test the algorithms on a pair of two non-deterministic automata. However there is still a bias because $\mathtt{NL}^\star$ produces a particular automaton. In fact the learned automaton is in a subclass of NFAs called *Residual Finite State Automata* (RFSAs). An RFSA is a non deterministic finite automaton whose states accept residual languages of the language of the automaton. This is due to the way the learning algorithm processes. Even if NFAs in general have no notion of canonical minimal automaton, this

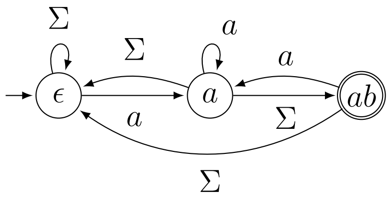


Figure 1.6: Automaton guessed by $\texttt{NL}^\star$ with target language $\mathcal{L} = \Sigma^* a \Sigma$

subclass does admit canonical representatives, which is the one generated by $\texttt{NL}^\star$. There exist some sequences of automata such that this minimal RFSA is exponentially smaller than the minimal DFA, but there also exist sequences of NFAs such that the minimal RFSA is of exponential size compared to the given NFA.

During the whole learning process, $\texttt{NL}^\star$ generates only minimal RFSAs for languages that eventually converge to the target language. Thus in this framework, the equivalence algorithms are tested on pairs consisting of a random NFA and a minimal RFSA. Yet it is not clear to us that it could have an impact on standards algorithms like HKC or antichain-based ones.

1.5 Conclusion and future work

We reformulated the algorithms HK and HKC with Moore machines. We will use this presentation to extend those algorithms to Büchi automata in Chapter 2.

This process led us to study more in depth the congruence problem: given a relation over sets of states and a pair of sets of states, is the pair in the congruence closure of the relation? This problem is solved repeatedly by a subroutine of HKC but it is done suboptimally. We presented a reduction to the problem of satisfiability of a formula and a variant of Dowling and Gallier's algorithm for HornSat to solve the problem in linear time in the size of the relation.

Lastly we presented a framework based on automata learning to evaluate in practice the efficiency of equivalence algorithms.

Chapter 2

Comparison algorithms for Büchi automata

2.1 Introduction

Büchi automata are machines which make it possible to recognize sets of infinite words. They form a natural counterpart to finite automata, which operate on finite words. They play a crucial role in logic for their links with monadic second order logic (MSO) [18], and in program verification. For instance, they are widely used in model-checking tools, in order to check whether a given program satisfies a linear temporal logic (LTL) formula [88, 41].

A key algorithmic property of Büchi automata is that checking whether two automata recognize the same language is decidable, and in fact PSPACE-complete, like in the finite case with non-deterministic finite automata. This is how one obtains model-checking algorithms. Several algorithms have been proposed in the literature [18, 44, 1, 55] and implemented in various tools [49, 87, 68].

The antichain-based algorithms could be adapted to Büchi automata by exploiting constructions to compute the complement of a Büchi automaton, either Ramsey-based [37, 38] or rank-based [34, 35]. Unfortunately, those complementation operations do not make it possible to adapt the algorithms based on bisimulations up to congruence: these algorithms require a proper powerset construction for determinization, which is not available for Büchi automata. Here we propose to circumvent this difficulty using a construction by Calbrix, Nivat, and Podelski [21], which makes it possible to reduce the problem of checking Büchi automata equivalence to that of checking equivalence of automata on finite words.

The first observation, which is used implicitly in the so-called Ramsey-based algorithms from the literature [37, 38, 1], is that it suffices to consider ultimately periodic words: if the languages of two Büchi automata differ, then they must differ on an ultimately periodic word. The second observation is that the set of ultimately periodic words accepted by a Büchi automaton can be faithfully represented as a rational language of finite words, for which Calbrix et al. give an explicit non-deterministic finite automaton. This automaton contains two layers: one for the prefixes of the ultimately periodic words, and one for their periods. We show that algorithms like `HKC` [14] can readily be used to reason about the prefix layer, without systematically determinizing it. The period layer requires more work in order to avoid paying a doubly exponential price. We show how to analyze it to compute *discriminating sets* that summarize the periodic behavior of the automaton, and suffice to check language equivalence.

We first revisit the construction of Calbrix et al. to make their use of the Büchi transition monoid [72] explicit (Section 2.2). We define the new algorithm $\mathtt{HKC}^\omega$ in Section 2.3. We discuss more advanced refinements of the algorithm in Section 2.4. We conclude with directions for future work in Section 2.5.

2.2 From Büchi automata to finite words automata

Let $\mathfrak{3}$ be the set $\{0, 1, \star\}$. A *(non-deterministic) Büchi automaton* (NBW) over the alphabet A is a tuple $\langle S, T\rangle$ where S is a finite set of states, and $T \colon A \to \mathfrak{3}^{S^2}$ is the transition function. We work with Büchi automata with Büchi transitions rather than Büchi states, hence the type of T (the two models are equivalent and the one we chose is slightly more succinct). The 0 represents the absence of a transition, the 1 a non-Büchi transition, and the $\star$ a Büchi transition. We write T_a for $T(a)$, $x \xrightarrow{a} x'$ when $T_a(x, x') \neq 0$, and $x \overset{a}{\Rightarrow} x'$ when $T_a(x, x') = \star$; the latter denote Büchi transitions, that should be fired infinitely often in order to accept an infinite word. Like for DFA and NFA, we do not include a set of initial states in the definition.

Given a NBW $\mathcal{A} = \langle S, T\rangle$ and $w \in A^\omega$ an infinite word, we say that a sequence of states $\chi \in S^\omega$ accepts w if the sequence $(T_{w_i}(\chi_i, \chi_{i+1}))_{i\in\mathbb{N}}$ contains infinitely many $\star$ and no 0. The ω-language $[X]_\mathcal{A}$ of a set of states $X \subseteq S$ is the set of infinite words accepted by a sequence χ such that $\chi_0 \in X$. The ω-languages accepted by some set of states in a NBW are the *rational ω-languages* [18].

Given a finite word $u \in A^*$ and a finite non-empty word $v \in A^+$, write uv^ω for the infinite word $w \in A^\omega$ defined by $w_i = u_i$ if $i < |u|$ and $w_i = v_{(i-|u|) mod |v|}$ otherwise. *Ultimately periodic words* are (infinite) words of the form uv^ω for some $u, v \in A^* \times A^+$. Given an ω-language $L \subseteq A^\omega$, we set

$$UP(L) = \{uv^\omega \mid uv^\omega \in L\} \qquad\qquad L^\$ = \{u\$v \mid uv^\omega \in L\}$$

$UP(L)$ is a ω-language over A while $L^\$$ is a language of finite words over the alphabet $A^\$ = A \uplus \{\$\}$. The first key observation is that the ultimately periodic words of a rational ω-language fully characterize it:

Proposition 2.1 ([21, Fact 1])**.** For all rational ω-languages L, L', we have that $UP(L) = UP(L')$ entails $L = L'$.

Proof. Consequence of the closure of rational ω-languages under Boolean operations [18], and the fact that every non-empty rational ω-language contains at least one ultimately periodic word. □

As a consequence, to compare the ω-languages of two sets of states in a NBW, it suffices to compare the ω-languages of ultimately periodic words they accept. Calbrix et al. show that these ω-languages can be faithfully represented as rational languages (of finite words):

Proposition 2.2 ([21, Proposition 4])**.** If $L \subseteq A^\omega$ is ω-regular, then $L^\$$ is regular.

To prove it, Calbrix et al. construct a NFA for $L^\$$ from a NBW $\mathcal{A}$ for L. The constructed NFA has two layers. The first layer recognizes the prefixes (the u in uv^ω). This is a copy of the NBW for L (without accepting states, and where the Büchi status of the transitions is ignored). This layer guesses non-deterministically when to read the \$ symbol and then jumps into the second layer, whose role is to recognize the period (the v in uv^ω). We depart from [21] here, by using the notion of *(Büchi) transition monoid* [72], which makes the presentation easier and eventually makes it possible to propose our algorithm.

Consider the set $\mathfrak{3}$ as an idempotent semiring, using the following operations:

$+$	0	1	$\star$
0	0	1	$\star$
1	1	1	$\star$
$\star$	$\star$	$\star$	$\star$

$\cdot$	0	1	$\star$
0	0	0	0
1	0	1	$\star$
$\star$	0	$\star$	$\star$

Let $\mathcal{A} = \langle S, T\rangle$ be an NBW and write $\mathscr{M} = \mathfrak{3}^{S^2}$ for the set of square matrices over $\mathfrak{3}$ indexed by S; it forms a Kleene algebra [22, 58] and in particular a semiring. Let I denote the identity matrix of $\mathscr{M}$. The transition function of $\mathcal{A}$ has type $A \to \mathscr{M}$; we extend it to finite words by setting $T_\epsilon = I$ and $T_{u_1 \dots u_n} = T_{u_1} \cdot \dots \cdot T_{u_n}$. We have that $T_u(x, x')$ is $\star$ if there is a path along u from x to x' visiting an accepting transition, 0 if there is no path from x to x' along u, and 1 otherwise. We extend the notations $x \xrightarrow{u} x'$ and $x \xRightarrow{u} x'$ to words accordingly.

The *Kleene star* M^* of a matrix $M \in \mathscr{M}$ is defined by $M^* := \sum_{i \in \mathbb{N}} M^i$, where the sum is defined componentwise with respect to the $+$ operation defined above on $\mathfrak{3}$. As before, the coefficient $M^*(x, x')$ represents the type of the "best" available path of any length from x to x': it is $\star$ if there is a path containing a Büchi transition, 1 if there is a path but not one with a Büchi transition, and 0 if there is no path at all. Using a pumping argument, we can remark that it is enough to consider paths with at most $2|S|$ transitions, so $M^* = \sum_{0 \leq i \leq 2|S|} M^i$.

A periodic word v^ω is accepted from a state x in $\mathcal{A}$ if and only if there is a *lasso* for v starting from x: a state y and two natural numbers n, m such that $x \xrightarrow{v^n} y \xRightarrow{v^m} y$. This information can be computed from the matrix T_v: given a matrix M, compute[1] its Kleene star M^*, and set:

$$\omega(M) = \{x \in S \mid \exists y \in S,\ M^*(x, y) \neq 0 \wedge M^*(y, y) = \star\} \quad . \tag{†}$$

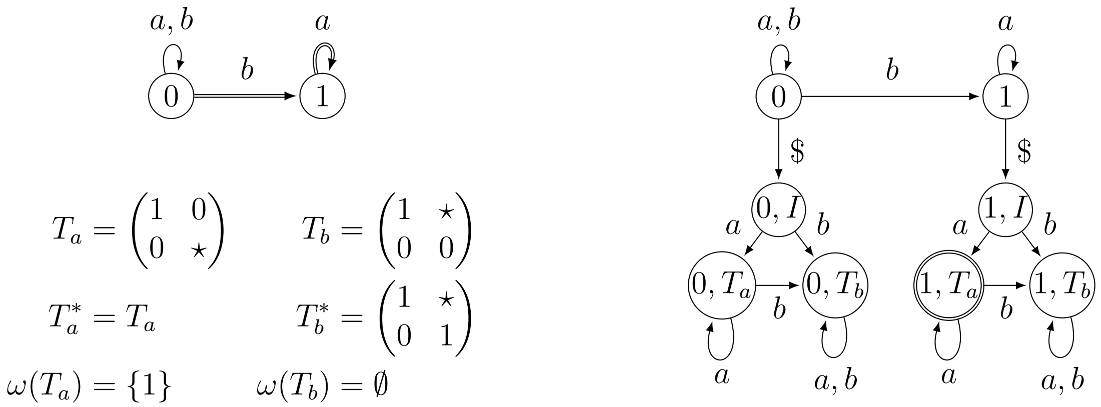
At this point, one can notice that with the previously defined operations, matrices and subsets form the *Wilke algebra* associated to the NBW $\mathcal{A}$, as in [72].

Lemma 1. *For all words v, v^ω is accepted from a state x iff $x \in \omega(T_v)$.*

We can now formally define the desired NFA: set $\mathcal{A}^\$ = \langle S^\$, o^\$, T^\$\rangle$, where $S^\$ = S \uplus S \times \mathscr{M}$ is the disjoint union of S and $|S|$ copies of $\mathscr{M}$, and

$$\begin{cases} T^\$_a(x) = \{x' \mid T_a(x, x') \neq 0\} \\ T^\$_a(\langle x, M\rangle) = \{\langle x, M \cdot T_a\rangle\} \end{cases} \qquad \begin{cases} T^\$_\$(x) = \{\langle x, I\rangle\} \\ T^\$_\$(\langle x, M\rangle) = \emptyset \end{cases} \qquad \begin{cases} o^\$(x) = 0 \\ o^\$(\langle x, M\rangle) = x \in \omega(M) \end{cases}$$

[1] To compute M^*, one can use the fact that $M^* = (I + M)^{2n}$ with $n = |S|$, and use iterated squaring.



$$T_a = \begin{pmatrix} 1 & 0 \\ 0 & \star \end{pmatrix} \qquad T_b = \begin{pmatrix} 1 & \star \\ 0 & 0 \end{pmatrix}$$

$$T_a^* = T_a \qquad T_b^* = \begin{pmatrix} 1 & \star \\ 0 & 1 \end{pmatrix}$$

$$\omega(T_a) = \{1\} \qquad \omega(T_b) = \emptyset$$

Figure 2.1: A NBW $\mathcal{A}$ (left) and the reachable part of its associated NFA $\mathcal{A}^{\$}$ (right).

The set $\mathscr{M}$ can be replaced here by its accessible part $\mathscr{M}' = \{T_u \mid u \in A^*\}$. The main difference with the construction from [21] is that we use deterministic automata in the second layer, which enable a streamlined presentation in terms of matrices—which are not mentioned explicitly in [21]. The construction of $\mathcal{A}^{\$}$ preserves the semantics of all sets of states, up to $L \mapsto L^{\$}$:

Theorem 2.1. *For all sets X of states from $\mathcal{A}$, we have $[X]_{\mathcal{A}^{\$}} = ([X]_{\mathcal{A}})^{\$}$.*

Example 2.1. To illustrate this construction, consider the NBW depicted on the left in Figure 2.1, where double lines represent Büchi transitions. The state 0 accepts the words with a finite but non-zero number of b's; the state 1 only accepts the word a^ω. Accordingly, we have $[0]_{\mathcal{A}}^{\$} = (a+b)^*ba^*\$a^+$ and $[1]_{\mathcal{A}}^{\$} = a^*\$a^+$.

The corresponding NFA $\mathcal{A}^{\$}$ is depicted on the right. Its states 0 and 1 form the first layer; they respectively recognize the two previous rational languages. The second layer is reached from those states when reading the letter \$. We only depicted the reachable part of the second layer: those states consisting of matrices of the form T_u for some word u. There are only three such matrices in this example since $T_a \cdot T_b = T_b \cdot T_a = T_b \cdot T_b = T_b$ and $T_a \cdot T_a = T_a$.

By definition, the second layer consists of several blocks (here, two) whose transitions are identical, and which differ only by the accepting status of their states. Given that the first block has no accepting state in this example, it might seem interesting to prune $\mathcal{A}^{\$}$ so that all states may reach an accepting state. We restrain ourselves from doing so because we want to exploit the fact that all blocks share the same structure.

Note that since the second layer of $\mathcal{A}^{\$}$ is already deterministic, one can determinize $\mathcal{A}^{\$}$ into a DFA with at most $2^n + 2^n 3^{n^2}$ states, where n is the number of states of $\mathcal{A}$. This is slightly better than the $2^n + 2^{2n^2+n}$ bound obtained in [21].

We summarize the operations defined so far on languages and automata in Figure 2.2; we define the operations in the right-most column in the following section.

2.3 HKC for Büchi automata

By Proposition 2.1 and Theorem 2.1, given two sets of states X, Y of a NBW $\mathcal{A}$, we have $[X]_{\mathcal{A}} = [Y]_{\mathcal{A}}$ iff $[X]_{\mathcal{A}^{\$}} = [Y]_{\mathcal{A}^{\$}}$. One can thus use any algorithm for language equivalence

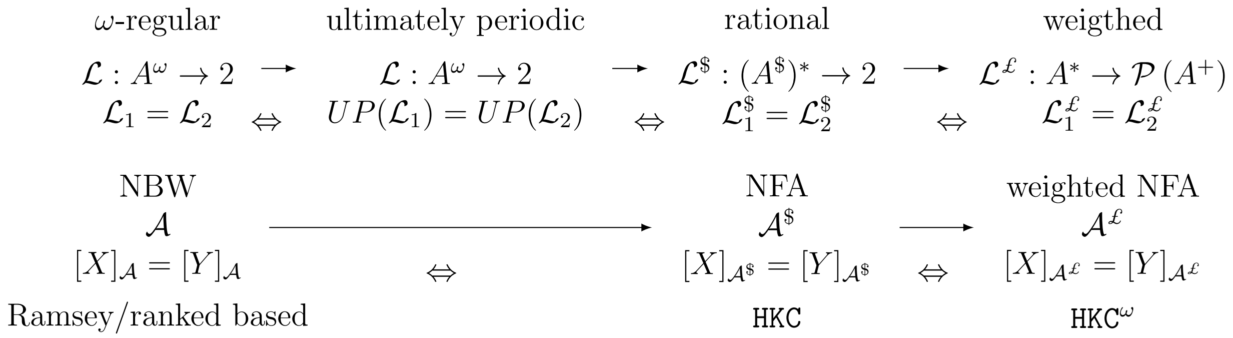

ω-regular
ultimately periodic
rational
weigthed
$\mathcal{L} : A^\omega \to 2$
$\mathcal{L} : A^\omega \to 2$
$\mathcal{L}^\$: (A^\$)^* \to 2$
$\mathcal{L}^{\pounds} : A^* \to \mathcal{P}(A^+)$
$\mathcal{L}_1 = \mathcal{L}_2$
$\Leftrightarrow$
$UP(\mathcal{L}_1) = UP(\mathcal{L}_2)$
$\Leftrightarrow$
$\mathcal{L}_1^\$ = \mathcal{L}_2^\$$
$\Leftrightarrow$
$\mathcal{L}_1^{\pounds} = \mathcal{L}_2^{\pounds}$
NBW
$\mathcal{A}$
NFA
$\mathcal{A}^\$$
weighted NFA
$\mathcal{A}^{\pounds}$
$[X]_{\mathcal{A}} = [Y]_{\mathcal{A}}$
$\Leftrightarrow$
$[X]_{\mathcal{A}^\$} = [Y]_{\mathcal{A}^\$}$
$\Leftrightarrow$
$[X]_{\mathcal{A}^{\pounds}} = [Y]_{\mathcal{A}^{\pounds}}$
Ramsey/ranked based
HKC
HKC$^\omega$

Figure 2.2: Summary of the operations and algorithms on languages and automata.

on NFA to solve language equivalence on NBW. Given the structure (and size) of $\mathcal{A}^\$$, this would however be inefficient: each time the letter \$ is read, the algorithm would explore one of the blocks of the second layer, without ever realizing that the transition structure of those sub-automata is always the same, only the accepting status of their states differ. We show in this section that we can do better, by using a weighted automaton.

Given an ω-language L, the language $L^\$$ can be seen as a weighted language $L^{\pounds} \colon A^* \to \mathcal{P}(A^+)$ with weights in the semilattice $\langle \mathcal{P}(A^+), \cup, \emptyset \rangle$:

$$L^{\pounds} : u \mapsto \left\{ v \in A^+ \mid uv^\omega \in L \right\}$$

Given a NBW $\mathcal{A} = \langle S, T \rangle$, one can immediately construct a NFA $\mathcal{A}^{\pounds} = \langle S^{\pounds}, T^{\pounds}, o^{\pounds} \rangle$ such that for every set of states X, $[X]_{\mathcal{A}}^{\pounds} = [X]_{\mathcal{A}^{\pounds}}$. This is just the first layer from the previous construction: set $S^{\pounds} = S$ and

$$T_a^{\pounds}(x) = \{x' \mid T_a(x, x') \neq 0\} \qquad\qquad o^{\pounds}(x) = \left\{ v \in A^+ \mid v^\omega \in [x]_{\mathcal{A}} \right\}$$

Let $\mathcal{A}^{\pounds\#}$ be the powerset automaton of $\mathcal{A}^{\pounds}$. To use algorithms such as HKC on $\mathcal{A}^{\pounds}$, it suffices to be able to compare the outputs of any two states of $\mathcal{A}^{\pounds\#}$, i.e., compare the languages $o^{\pounds\#}(X)$ and $o^{\pounds\#}(Y)$ for any two sets $X, Y \subseteq S$. Since those languages are rational (using the second layer of the previous construction), it might be tempting to use algorithms such as `HK` or `HKC` to perform this task. We proceed differently in order to exploit the shared structure of those languages.

Lemma 2. *For all states $x \in S$ and sets $X \subseteq S$, we have*

$$o^{\pounds}(x) = \left\{ v \in A^+ \mid x \in \omega(T_v) \right\}$$
$$o^{\pounds\#}(X) = \left\{ v \in A^+ \mid X \cap \omega(T_v) \neq \emptyset \right\}$$

Proof. Immediate consequence of Lemma 1 and the definitions of $o^{\pounds}$ and $o^{\pounds\#}$. □

Note that allowing empty v would not change the statement since $\omega(T_\epsilon) = \omega(I) = \emptyset$.

Proposition 2.3. For all sets $X, Y \subseteq S$,

$$o^{\pounds\#}(X) = o^{\pounds\#}(Y) \quad \text{iff} \quad \text{for all } v \in A^+,\ X \cap \omega(T_v) = \emptyset \Leftrightarrow Y \cap \omega(T_v) = \emptyset.$$

```
input  : A NBW $\mathcal{A} = \langle S, T \rangle$
output : The set of discriminating sets $\mathcal{D} = \{\omega(T_v) \mid v \in A^*\}$

$\mathcal{D} := \emptyset$; $\mathcal{M} := \emptyset$; $todo := \{I\}$;
while $todo \neq \emptyset$ do
    extract $M$ from $todo$;
    if $M \in \mathcal{M}$ then skip;
    forall $a \in A$ do
        insert $M \cdot T_a$ in $todo$;
    insert $M$ in $\mathcal{M}$; insert $\omega(M)$ in $\mathcal{D}$;
return $\mathcal{D}$;
```

Figure 2.3: `Discr`($\mathcal{A}$): exploring the transition monoid of a NBW $\mathcal{A}$ to compute discriminating sets.

```
input  : A NBW $\mathcal{A} = \langle S, T \rangle$ and two sets $X, Y \subseteq S$
output : true if $[X]_{\mathcal{A}} = [Y]_{\mathcal{A}}$; false otherwise

1 $R := \mathtt{HKC}'(\mathcal{A}^{\pounds}, X, Y)$     ||     $\mathcal{D} := \mathtt{Discr}(\mathcal{A})$;
2 forall $\langle X', Y' \rangle \in R,\ D \in \mathcal{D}$ do
3     if $X' \cap D = \emptyset \not\Leftrightarrow Y' \cap D = \emptyset$ then return false;
4 return true;
```

Figure 2.4: $\mathtt{HKC}^{\omega}(\mathcal{A}, X, Y)$: checking language equivalence in a NBW using bisimulations up to congruence.

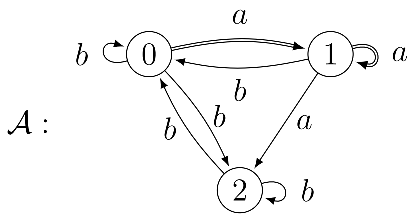
This result shows that an explicit computation of $o^{\pounds\#}$ is not necessary, as the knowledge of $\{\omega(T_v), v \in A^+\}$ is enough to assess whether X and Y have same output. Let $\mathcal{D} = \{\omega(T_v) \mid v \in A^+\}$. We call the sets in $\mathcal{D}$ *discriminating sets*. Again, allowing empty v here would make no difference: the discriminating set $\emptyset$ is useless to distinguish two sets $X, Y \subseteq S$. As subsets of S, there are at most $2^{|S|}$ discriminating sets. Those can be enumerated since the T_v range over finitely many matrices (at most $3^{|S|^2}$). This is what is done in the algorithm from Figure 2.3.

We finally obtain the algorithm in Figure 2.4 for language equivalence in a NBW: we compute the discriminating sets ($\mathcal{D}$) and a relation (R) which is almost a bisimulation up to congruence: the outputs of its pairs must be checked against the discriminating sets, which we achieve with a simple loop (lines 2-4).

Example 2.2. We execute $\mathtt{HKC}^{\omega}$ on the NBW on the left of Figure 2.5, starting with states $\{0\}$ and $\{1\}$.

The transition monoid has 13 elements, which are listed with their discriminating sets in Figure 2.6. In fact the exploration of the monoid by the algorithm `Discr` stops because of the following equations:

$$\begin{array}{ccccc}
T_{aaa} = T_{aa} & T_{bba} = T_{ba} & T_{aaba} = T_{aba} & T_{abaa} = T_{aa} & T_{baba} = T_{ba} \\
T_{abb} = T_{aab} & T_{bbb} = T_{bb} & T_{aabb} = T_{aab} & T_{baaa} = T_{baa} & T_{babb} = T_{baab}
\end{array}$$



$$R = \mathtt{HKC}'(\mathcal{A}^{\pounds}, \{0\}, \{1\}) = \{\langle\{0\}, \{1\}\rangle, \langle\{1\}, \{1,2\}\rangle\}$$
$$\mathcal{D} = \mathtt{Discr}(\mathcal{A}) = \{\emptyset, \{0,1\}, \{0,1,2\}\}$$

Figure 2.5: An example run of $\mathtt{HKC}^\omega$

u	ϵ	a	b	aa	ab	ba	bb
T_u	$\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix}$	$\begin{pmatrix}0&\star&0\\0&\star&1\\0&0&0\end{pmatrix}$	$\begin{pmatrix}1&0&1\\1&0&0\\1&0&1\end{pmatrix}$	$\begin{pmatrix}0&\star&\star\\0&\star&\star\\0&0&0\end{pmatrix}$	$\begin{pmatrix}\star&0&0\\\star&0&1\\0&0&0\end{pmatrix}$	$\begin{pmatrix}0&\star&0\\0&\star&0\\0&\star&0\end{pmatrix}$	$\begin{pmatrix}1&0&1\\1&0&1\\1&0&1\end{pmatrix}$
T_u^*	$\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix}$	$\begin{pmatrix}1&\star&\star\\0&\star&\star\\0&0&1\end{pmatrix}$	$\begin{pmatrix}1&0&1\\1&1&1\\1&0&1\end{pmatrix}$	$\begin{pmatrix}1&\star&\star\\0&\star&\star\\0&0&1\end{pmatrix}$	$\begin{pmatrix}\star&0&0\\\star&1&1\\0&0&1\end{pmatrix}$	$\begin{pmatrix}1&\star&0\\0&\star&0\\0&\star&1\end{pmatrix}$	$\begin{pmatrix}1&0&1\\1&1&1\\1&0&1\end{pmatrix}$
$\omega(T_u)$	$\varnothing$	$\{0,1\}$	$\varnothing$	$\{0,1\}$	$\{0,1\}$	$\{0,1,2\}$	$\varnothing$

u	aab	aba	baa	bab	$abab$	$baab$
T_u	$\begin{pmatrix}\star&0&\star\\\star&0&\star\\0&0&0\end{pmatrix}$	$\begin{pmatrix}0&\star&0\\0&\star&0\\0&0&0\end{pmatrix}$	$\begin{pmatrix}0&\star&\star\\0&\star&\star\\0&\star&\star\end{pmatrix}$	$\begin{pmatrix}\star&0&0\\\star&0&0\\\star&0&0\end{pmatrix}$	$\begin{pmatrix}\star&0&0\\\star&0&0\\0&0&0\end{pmatrix}$	$\begin{pmatrix}\star&0&\star\\\star&0&\star\\\star&0&\star\end{pmatrix}$
T_u^*	$\begin{pmatrix}\star&0&\star\\\star&1&\star\\0&0&1\end{pmatrix}$	$\begin{pmatrix}1&\star&0\\0&\star&0\\0&0&1\end{pmatrix}$	$\begin{pmatrix}1&\star&\star\\0&\star&\star\\0&\star&\star\end{pmatrix}$	$\begin{pmatrix}\star&0&0\\\star&1&0\\\star&0&1\end{pmatrix}$	$\begin{pmatrix}\star&0&0\\\star&1&0\\0&0&1\end{pmatrix}$	$\begin{pmatrix}\star&0&\star\\\star&1&\star\\\star&0&\star\end{pmatrix}$
$\omega(T_u)$	$\{0,1\}$	$\{0,1\}$	$\{0,1,2\}$	$\{0,1,2\}$	$\{0,1\}$	$\{0,1,2\}$

Figure 2.6: Reachable part of the transition monoid of Example 2.2. The discriminating sets are calculated using the formula (†) on page 38.

It then returns three different discriminating sets: $\emptyset$, $\{0,1\}$, and $\{0,1,2\}$, which arise for instance from the matrices T_b, T_a and T_{ba}.

On the other hand $\mathtt{HKC}$' returns the relation $R = \{\langle\{0\}, \{1\}\rangle, \langle\{1\}, \{1,2\}\rangle\}$, which contains only two pairs. The pairs $\langle\{0,2\}, \{0\}\rangle$, $\langle\{1,2\}, \{1,2\}\rangle$, and $\langle\{0\}, \{0,2\}\rangle$, which are reachable from $\langle\{0\}, \{1\}\rangle$ by reading the words b, aa, and ab, are skipped thanks to the up to congruence technique. For instance the pair $\langle\{0,2\}, \{0\}\rangle$ belongs to the congruence closure of R thanks to the following argument: starting from $\langle\{0\}, \{1\}\rangle$ and $\langle\{1\}, \{1,2\}\rangle$ we can obtain $\langle\{0\}, \{1,2\}\rangle$ by transitivity, from which we deduce $\langle\{0,2\}, \{1,2\}\rangle$ by union with $\langle\{2\}, \{2\}\rangle$; we finally obtain $\langle\{0,2\}, \{0\}\rangle$ by transitivity and symmetry.

The two pairs of R cannot be told apart using the three discriminating sets and $\mathtt{HKC}^\omega$ returns *true*. States 0 and 1 are indeed equivalent: they accept the words with infinitely many a's. If instead we start $\mathtt{HKC}^\omega$ from sets $\{0\}$ and $\{2\}$, it returns *false*: the discriminating set $\{0,1\}$ distinguishes $\{0\}$ and $\{2\}$. Indeed, the state 2 recognizes the words starting with b and with infinitely many a's.

Note that $\mathtt{HKC}^\omega$ can be instrumented to return a counterexample in case of failure: it suffices to record the finite word u leading to each pair in R as well as the finite word v leading to each discriminating set in $\mathcal{D}$: if the check on line 3 fails, the corresponding word uv^ω is a counter-example to language equivalence.

Also note that $\mathtt{HKC}^\omega$ is intrinsically parallel: the computations of $\mathcal{D}$ and R can be done in parallel, and the checks in lines 2-4 can be performed using a producer-consumer pattern where they are triggered whenever new values are inserted in $\mathcal{D}$ or R. Alternatively, those checks can be delegated to a SAT solver. Indeed, given a discriminating set D, define the following formula with $2|D|$ variables $\{x_d \mid d \in D\} \cup \{y_d \mid d \in D\}$:

$$\varphi_D = \bigvee_{d \in D} x_d \;\Leftrightarrow\; \bigvee_{d \in D} y_d$$

For all sets $X, Y \subseteq S$, we have $X \cap D = \emptyset \Leftrightarrow Y \cap D = \emptyset$ iff φ_D evaluates to *true* under the assignment $x_d \mapsto d \in X$ and $y_d \mapsto d \in Y$. Given the set of discriminating sets $\mathcal{D}$, it thus suffices to build the formula $\varphi_{\mathcal{D}} = \bigwedge_{D \in \mathcal{D}} \varphi_D$ with $2|S|$ variables, and to evaluate it on all pairs from the relation R returned by `HKC`'. The main advantage of proceeding this way is that the SAT solver might be able to represent $\varphi_{\mathcal{D}}$ in a compact and efficient way. If we moreover use an incremental SAT solver, this formula can be built incrementally, thus avoiding the need to store explicitly the set $\mathcal{D}$.

One can also use a (incremental) SAT solver in a symmetrical way: Given a pair of sets $\langle X, Y \rangle \in S^2$, define the following formula with $|S|$ variables $\{x_s \mid s \in S\}$:

$$\psi_{\langle X,Y \rangle} = \bigvee_{s \in X} x_s \;\Leftrightarrow\; \bigvee_{s \in Y} x_s$$

For all sets D, we have $X \cap D = \emptyset \Leftrightarrow Y \cap D = \emptyset$ iff $\psi_{\langle X,Y \rangle}$ evaluates to *true* under the assignment $x_s \mapsto s \in D$. Like previously, one can thus construct incrementally the formula $\psi_R = \bigwedge_{p \in R} \psi_p$ before evaluating it on all discriminating sets.

2.4 Further refinements

A weakness of the algorithm `HKC`$^\omega$ is that it must fully explore the transition monoid of the starting NBW, which may contain up to 3^{n^2} elements when starting with a NBW with n states. Since the goal of this exploration is to obtain discriminating sets, we would like to isolate parts of the transition monoid that can safely be skipped: for instance because they will lead to discriminating sets which have already been encountered, or which are subsumed by previously encountered ones. This leads us to optimizations which are similar in spirit to those brought by `HKC` for the analysis of the prefix automaton.

To make this idea precise, given a set of sets of states $\mathcal{E}$, define the following equivalence relation on sets of states:

$$X \sim_{\mathcal{E}} Y \quad \text{if} \quad \forall D \in \mathcal{E}, X \cap D = \emptyset \Leftrightarrow Y \cap D = \emptyset$$

By Proposition 2.3, we can replace the sub-algorithm `Discr` (Figure 2.3) by any algorithm returning a subset $\mathcal{D}'$ of $\mathcal{D}$ such that $\sim_{\mathcal{D}'} = \sim_{\mathcal{D}}$.

This sub-algorithm basically computes the least solution to an equation (the least set of matrices containing the identity and closed under multiplication on the right by the transition matrices of the starting NBW), and computes a set of discriminating sets out of this solution. We can improve it by weakening the equation to be satisfied, in the very same way `HKC` improves over `HK` by allowing to look for bisimulations up to congruence rather than bisimulations up to equivalence. We shall use the following abstract lemma about partial orders to prove the correctness of such improvements: This lemma is inspired by the theory of coinduction up-to [73], where the *compatibility* condition $f \circ r \leq r \circ f$ plays a central role. We explain how we will instantiate this lemma below.

Lemma 3. *Let $\mathcal{X}, \mathcal{Y}$ be two partial orders. Let $r, f\colon \mathcal{X} \to \mathcal{X}$ and $s\colon \mathcal{X} \to \mathcal{Y}$ be three monotone functions such that $f \circ r \leq r \circ f$; $id \leq f$; $f \circ f \leq f$; and $s \circ f \leq s$. Fix $x_0 \in \mathcal{X}$, suppose that x is a least element of $\mathcal{X}$ such that $x_0 \leq x \leq r(x)$, and assume that x' is an element such that $x_0 \leq x' \leq r(f(x'))$ and $x' \leq x$. Then we have $s(x) = s(x')$.*

Proof. Since $f \circ r \leq r \circ f$ and $f \circ f \leq f$, we have $f(x') \leq f(r(f(x'))) \leq r(f(f(x'))) \leq r(f(x'))$. Since $id \leq f$, we also have $x_0 \leq x' \leq f(x')$, so that $x \leq f(x')$ by minimality of x. We deduce $s(x') \leq s(x) \leq s(f(x')) \leq s(x')$ by monotonicity of s and $s \circ f \leq s$. □

Given a set $\mathcal{M}$ of matrices, set $d(\mathcal{M}) = \{\omega(M) \mid M \in \mathcal{M}\}$. We can apply the above lemma by choosing $\mathcal{X} = \langle \mathcal{P}(\mathscr{M}), \subseteq \rangle$, $\mathcal{Y} = \langle Rel(\mathcal{P}(S)), \supseteq \rangle$, $x_0 = \{I\}$, and

$$r(\mathcal{M}) = \{M \mid \forall a \in A, M \cdot T_a \in \mathcal{M}\} \qquad\qquad s(\mathcal{M}) = \sim_{d(\mathcal{M})}$$

We have $\mathcal{M} \subseteq r(\mathcal{M})$ if and only if $\mathcal{M}$ is closed under multiplication on the right by the $(T_a)_{a \in A}$. Accordingly, the x from the statement of the lemma is the set of matrices $\mathcal{M} = \{T_u \mid u \in A^*\}$ obtained at the end of the execution of `Discr`. It follows that $d(x)$ is the returned set $\mathcal{D}$ of discriminating sets, and $s(x)$ is the equivalence relation $\sim_{\mathcal{D}}$.

We will show how to instantiate the function f from the lemma in the following sections. Intuitively, a function f satisfying the other requirements of the lemma can be used as an up-to technique, in order to skip elements from the transition monoid. Indeed, we can obtain an algorithm $\texttt{Discr}_f$ by replacing line 4 from `Discr` (Figure 2.3) with

4' | **if** $M \in f(\mathcal{M} \cup todo)$ **then** skip;

This algorithm terminates with a subset $\mathcal{M}' \subseteq \mathcal{M}$ of matrices corresponding to the x' from the statement of the lemma, and returns a set $\mathcal{D}'$ of discriminating sets for which the lemma guarantees that we have $\sim_{\mathcal{D}'} = \sim_{\mathcal{D}}$, as required.

Such techniques can drastically improve performances: when an element is skipped thanks to the up-to technique, all elements which were reachable only through this element virtually disappear. We give two examples of such techniques in the sequel.

2.4.1 Working up to unions

A first property which we can exploit in order to cut-down the exploration of the transition monoid is the following: if two discriminating sets D, D' have been discovered, then their union $D \cup D'$ is not useful as a discriminating set. Formally, for all $\mathcal{D} \subseteq \mathcal{P}(S)$, if $D, D' \in \mathcal{D}$ then $\sim_{\{D \cup D'\} \cup \mathcal{D}} = \sim_{\mathcal{D}}$.

One could think that this should allow us to skip matrices from the transition monoid when they can be written as sums of already visited matrices. This is however wrong, because the discriminating set of a sum is in general *not* the union of the underlying discriminating sets. For instance, we have:

$$\omega\begin{pmatrix} 0 & \star \\ 1 & 0 \end{pmatrix} = \{0, 1\} \neq \emptyset \cup \emptyset = \omega\begin{pmatrix} 0 & \star \\ 0 & 0 \end{pmatrix} \cup \omega\begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$$

In order to find an operation on matrices which corresponds to unions when taking discriminating sets, we need to slightly generalize the notion of matrix.

Say that a matrix is a *vector* if it contains at most one non-zero coefficient per line. Let $\mathscr{V}$ denote the set of vectors. The three matrices above are vectors. A *generalized matrix* is a set of vectors. We write $\mathscr{M}'$ for the set of generalized matrices. We order vectors and matrices pointwise using the order $0 < 1 < \star$. Given a matrix M, we write $\underline{M}$ for the generalized matrix $\{V \in \mathscr{V} \mid V \leq M\}$. While the map $M \mapsto \underline{M}$ is injective, it is not surjective: there are generalized matrices which cannot be represented using a single matrix. We equip $\mathscr{M}'$ with a sum $\oplus$, a mixed product $\bullet$, and an operation ω as follows ($\mathbf{M}, \mathbf{N}$ range over generalized matrices, N ranges over matrices, V ranges vectors):

$$\mathbf{M} \oplus \mathbf{N} = \mathbf{M} \cup \mathbf{N} \qquad \mathbf{M} \bullet N = \bigcup_{V \in \mathbf{M}} \underline{V \cdot N} \qquad \omega(\mathbf{M}) = \bigcup_{V \in \mathbf{M}} \omega(V)$$

By definition, the mixed product is distributive on the left, and the function ω is a homomorphism of semilattices:

Lemma 4. *For all generalized matrices $\mathbf{M}, \mathbf{N} \in \mathscr{M}'$ and matrix $O \in \mathscr{M}$, we have*

$$(i)\ (\mathbf{M} \oplus \mathbf{N}) \bullet O = (\mathbf{M} \bullet O) \oplus (\mathbf{N} \bullet O) \qquad (ii)\ \omega(\mathbf{M} \oplus \mathbf{N}) = \omega(\mathbf{M}) \cup \omega(\mathbf{N})$$

The counter-example above shows that in general, $\underline{M} \oplus \underline{N} \neq \underline{M + N}$. However, we do have:

Lemma 5. *For all matrices $M, N \in \mathscr{M}$, we have:*

$$(i)\ \underline{M} \bullet N = \underline{M \cdot N} \qquad (ii)\ \omega(\underline{M}) = \omega(M)$$

Proof. (i) We have

$$\underline{M} \bullet N = \bigcup_{V \leq M} \underline{V \cdot N} \qquad \underline{M \cdot N} = \{U \in \mathscr{V} \mid U \leq M \cdot N\}$$

The direct inclusion comes from monotonicity of matrix multiplication: if $U \leq V \cdot N$ for some vector $V \leq M$, then $V \cdot N \leq M \cdot N$, whence $U \leq M \cdot N$. For the other inclusion, assume a vector $U \leq M \cdot N$. The non-empty elements of U can be described by a function δ associating to each row i the column $\delta(i)$ of the non-empty coefficient of that row (or an arbitrary one if the row is all zeros). For all row i, we have $U_{i,\delta(i)} \leq \Sigma_k M_{i,k} N_{k,\delta(i)}$, and we can find an index $\delta'(i)$ such that $\Sigma_k M_{i,k} N_{k,\delta(i)} = M_{i,\delta'(i)} N_{\delta'(i),\delta(i)}$. δ' determines a vector $V \leq M$ such that $U \leq V \cdot N$, as required.

(ii) The fact that $x \in \omega(M)$ is witnessed by an accepting lasso in M, and such a lasso can be assumed to be simple (i.e., every state is visited at most once, except the last visited state which is visited twice). Such a simple lasso yields $x \in \omega(V)$ for a vector $V \leq M$ (just select in M those transitions that are required by the simple lasso).

□

Now lift the functions r, d, s we defined after Lemma 3 to sets of generalized matrices $\mathcal{P}(\mathscr{M}')$:

$$r(E) = \{\mathbf{M} \mid \forall a \in A,\ \mathbf{M} \bullet T_a \in E\} \qquad d(E) = \{\omega(\mathbf{M}) \mid \mathbf{M} \in E\} \qquad s(E) = \sim_{d(E)}$$

Finally define the up-to technique as the following function $u\colon \mathcal{P}(\mathscr{M}') \to \mathcal{P}(\mathscr{M}')$:

$$u(E) = \{\mathbf{M}_1 \oplus \cdots \oplus \mathbf{M}_n \mid n \in \mathbb{N},\ \forall i \leq n,\ \mathbf{M}_i \in E\}$$

Intuitively, this function allows us to cut-down the exploration on the transition monoid whenever we encounter a matrix which can be written as a union (in the sense of $\oplus$) of already encountered matrices.

Proposition 2.4. The functions r, u and s satisfy the requirements of Lemma 3 (taking u for f).

Proof. For compatibility of u w.r.t. r ($u \circ r \leq r \circ u$), let $\mathbf{M}_1 \oplus \cdots \oplus \mathbf{M}_n$ with $\forall a, i \leq n,\ \mathbf{M}_i \bullet T_a \in E$ be an element of $u(r(E))$ for some E. We have to show that this sum belongs to $r(u(e))$, i.e., $\forall a, (\mathbf{M}_1 \oplus \cdots \oplus \mathbf{M}_n) \bullet T_a \in u(E)$. This follows directly from distributivity of $\bullet$ over $\oplus$ (Lemma 4(i)).

The function u is obviously extensive (id $\leq u$) and idempotent ($u \circ u = u$).

The last requirement ($s \circ u \leq s$) follows from Lemma 4(ii) and the observation at the beginning of Section 2.4.1. □

Generalized matrices are not convenient to use in practice: many matrices expand into generalized matrices of exponential size. However, we use them only to establish the correctness of the optimization: thanks to Lemma 5, the version of the algorithm `Discr` where we use the function u to cut down the search-space only manipulates generalized matrices of the form $\underline{M}$, which can thus be represented as plain matrices.

It remains to check that we can implement the refined check on line 4'. The following lemma shows that this is relatively expensive (at least theoretically, since state-of-the art SAT solvers tend to be efficient in practice).

Proposition 2.5. Given a set $\mathcal{M}$ of matrices and a matrix N, the problem of deciding if $\underline{N} \in u(\{\underline{M} \mid M \in \mathcal{M}\})$ is coNP-complete.

Proof. Let's first detail what it means for $\underline{N}$ to be in $u(\{\underline{M} \mid M \in \mathcal{M}\}$:

$$\begin{aligned}
\underline{N} \in u(\{\underline{M} \mid M \in \mathcal{M}\}) &\Leftrightarrow \underline{N} \in \cup_{\{M \in \mathcal{M} \mid M \leq N\}} \underline{M} \\
&\Leftrightarrow \forall V \in \underline{N},\ V \in \cup_{\{M \in \mathcal{M} \mid M \leq N\}} \underline{M} \\
&\Leftrightarrow \forall V \in \underline{N},\ \exists M \in \mathcal{M},\ M \leq N \text{ and } \exists V' \in \underline{M} \text{ s.t. } V \leq V' \\
&\Leftrightarrow \forall V \in \underline{N},\ \exists M \in \mathcal{M},\ M \leq N \text{ and } V \in \underline{M}
\end{aligned}$$

The existential subformula can be checked in polynomial time, hence the membership in coNP.

To show that the problem is coNP-Hard we will show that its complementary problem is NP-Hard via a reduction from 3-SAT. Let $\mathcal{I} = C_1 \wedge \cdots \wedge C_k$ be an instance of 3-SAT. We note $x_1, \ldots, x_n$ the Boolean variables of $\mathcal{I}$. We construct an instance of our problem as:

$$N = \begin{pmatrix} 1 & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & 1 & 0 & \cdots & 0 \end{pmatrix} \quad \mathcal{M} = \left\{ M_i : \begin{pmatrix} y_1^i & 0 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ y_n^i & 0 & \cdots & 0 \end{pmatrix} \right\}_{1 \leq i \leq k} \quad y_j^i = \begin{cases} 1\ 0 \text{ if } \overline{x_j} \in C_i \\ 0\ 1 \text{ if } x_j \in C_i \\ 1\ 1 \text{ if } \overline{x_j}, x_j \notin C_i \end{cases}$$

We can do some observations on this instance:

1. For all $M_i \in \mathcal{M}$, $M_i \leq N$.
2. There is a bijection between the set of truth value distribution of $x_1, \ldots, x_n$ and $\underline{N}$ via the function:

$$f : \begin{cases} 2^n & \to \underline{N} \\ \delta & \mapsto V_\delta \end{cases} \qquad\qquad (V_\delta)_j = \begin{cases} 1\ 0\ 0 \cdots 0 \text{ iff } \delta(x_j) = \top \\ 0\ 1\ 0 \cdots 0 \text{ iff } \delta(x_j) = \bot \end{cases}$$

3. For any $M_i \in \mathcal{M}$, $V_\delta \in \underline{M_i}$ if and only if δ does not satisfy the clause C_i.

Then:

$$\begin{aligned} \mathcal{I} \text{ is not satisfiable } &\Leftrightarrow \forall \delta \in 2^n,\ \exists C_i \text{ s.t. } \delta \text{ does not satisfy } C_i \\ &\Leftrightarrow \forall V_\delta \in f(2^n),\ \exists M_i \in \mathcal{M} \text{ s.t. } V_\delta \in \underline{M_i} \\ &\Leftrightarrow \forall V \in \underline{N},\ \exists M_i \in \mathcal{M} \text{ s.t. } V \in \underline{M_i} \\ &\Leftrightarrow \underline{N} \in u(\{\underline{M} \mid M \in \mathcal{M}\}) \end{aligned}$$

We have shown than $\mathcal{I}$ is satisfiable if and only if $\underline{N} \notin u(\{\underline{M} \mid M \in \mathcal{M}\})$.
The initial problem is thus coNP-hard. □

Example 2.3. When running this refined version of $\texttt{HKC}^\omega$ on the NBW on the top right in Figure 2.7, the up-to-union technique makes it possible to explore only 11 matrices of the monoid, although it contains 17 elements. Indeed, 4 matrices are skipped, being recognized as sums of previously encountered matrices, and 2 matrices are not even computed because they are reachable only through the 4 previous matrices.

The explored part of the transition monoid of the NBW is detailed in Figure 2.7, together with the discriminating sets associated to its elements and the justification for the elements skipped thanks to the up-to-union technique. Note that $T_{ca} = T_{bc} + T_{ccc}$ but $\underline{T_{ca}} \neq \underline{T_{bc}} \oplus \underline{T_{ccc}}$.

2.4.2 Working up to equivalence

There is also room for improvement when we start with a disjoint union of NBWs: the starting NBWs most probably contain loops, and the transition monoid of the disjoint union will need to unfold those loops until they 'synchronize'. Take for instance the two NBWs $\mathcal{A}_1$ and $\mathcal{A}_2$ over a single letter a, defined by the two matrices on the left in Figure 2.8, whose disjoint $\mathcal{A}$ union can be represented by the diagonal block matrix on the right.

We have $T_{1(aa)a} = T_{1a}$: the transition monoid of $\mathcal{A}_1$ has size 3 (including I); we have $T_{2(aaa)a} = T_{2a}$: the transition monoid of $\mathcal{A}_2$ has size 4; and we have $T_{(aaaaaa)a} = T_a$: the transition monoid of $\mathcal{A}$ has size 7. Generalizing 2 and 3 into n and m in the example, the transition monoid of the disjoint union contains $\texttt{lcm}(n, m) + 1$ matrices. By designing an up-to-equivalence technique reminiscent of the one used in Hopcroft and Karp's algorithm, we will obtain an algorithm that explores at most the first $n + m + 1$ matrices. (On this specific example all matrices but I give rise to the same discriminating set, so that we could stop even earlier; but there is no generic argument behind this observation.)

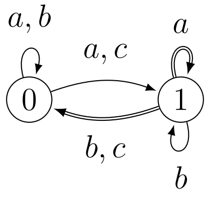
Kept matrices

u	T_u	$\omega(T_u)$
ϵ	$\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}$	$\varnothing$
a	$\begin{pmatrix}1 & 1\\ 0 & \star\end{pmatrix}$	$\{0,1\}$
b	$\begin{pmatrix}1 & 0\\ \star & 1\end{pmatrix}$	$\varnothing$
c	$\begin{pmatrix}0 & 1\\ \star & 0\end{pmatrix}$	$\{0,1\}$
aa	$\begin{pmatrix}1 & \star\\ 0 & \star\end{pmatrix}$	$\{0,1\}$
ac	$\begin{pmatrix}\star & 1\\ \star & 0\end{pmatrix}$	$\{0,1\}$
bc	$\begin{pmatrix}0 & 1\\ \star & \star\end{pmatrix}$	$\{0,1\}$
ca	$\begin{pmatrix}0 & \star\\ \star & \star\end{pmatrix}$	$\{0,1\}$
cc	$\begin{pmatrix}\star & 0\\ 0 & \star\end{pmatrix}$	$\{0,1\}$
bcc	$\begin{pmatrix}\star & 0\\ \star & \star\end{pmatrix}$	$\{0,1\}$
ccc	$\begin{pmatrix}0 & \star\\ \star & 0\end{pmatrix}$	$\{0,1\}$

Equations

$$\begin{aligned}
T_{bb} &= T_b\\
T_{cb} &= T_{ac}\\
T_{acb} &= T_{ac}\\
T_{aaa} &= T_{aa}\\
T_{aab} &= T_{aca}\\
T_{aac} &= T_{ac}\\
T_{bca} &= T_{ca}\\
T_{bcb} &= T_{ab}\\
T_{caa} &= T_{ca}\\
T_{cab} &= T_{aab}\\
T_{cac} &= T_{bcc}\\
T_{cca} &= T_{acc}\\
T_{ccb} &= T_{bcc}\\
T_{bccc} &= T_{ca}
\end{aligned}$$

Automaton



Skipped matrices (by union)

$$\begin{aligned}
T_{ab} &= \begin{pmatrix}\star & 1\\ \star & \star\end{pmatrix} &= T_{ac} \oplus T_{cc} \oplus T_{bc}\\
T_{ba} &= \begin{pmatrix}1 & 1\\ \star & \star\end{pmatrix} &= T_{b} \oplus T_{bc} \oplus T_{a}\\
T_{acc} &= \begin{pmatrix}\star & \star\\ 0 & \star\end{pmatrix} &= T_{aa} \oplus T_{cc}\\
T_{aca} &= \begin{pmatrix}\star & \star\\ \star & \star\end{pmatrix} &= T_{ab} \oplus T_{ca}
\end{aligned}$$

Accessible matrices not generated

$$\begin{aligned}
T_{baa} &= \begin{pmatrix}1 & \star\\ \star & \star\end{pmatrix} \quad ; \quad \omega(T_{baa}) = \{0,1\}\\
T_{accc} &= \begin{pmatrix}\star & \star\\ \star & 0\end{pmatrix} \quad ; \quad \omega(T_{accc}) = \{0,1\}
\end{aligned}$$

Figure 2.7: Exploration of a transition monoid with up-to-union technique. To alleviate notations, we identified matrices M with their associated generalized matrices $\underline{M}$.

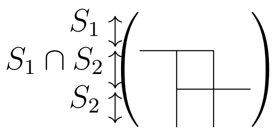
$$T_{1a} = \begin{pmatrix}0 & \star\\ \star & 0\end{pmatrix} \qquad T_{2a} = \begin{pmatrix}0 & \star & 0\\ 0 & 0 & \star\\ \star & 0 & 0\end{pmatrix} \qquad T_a = \begin{pmatrix}0 & \star & & & \\ \star & 0 & & & \\ & & 0 & \star & 0\\ & & 0 & 0 & \star\\ & & \star & 0 & 0\end{pmatrix}$$

Figure 2.8: Motivating example for up-to-equivalence technique.

We fix in the sequel a NBW $\mathcal{A} = \langle S, T\rangle$ and two subsets $S_1, S_2 \subseteq S$. Let $\mathscr{M}_d$ be the set of matrices M such that:

$$\forall i \in \{1,2\}\,,\ \forall x, y \in S,\ x \in S_i\ \wedge\ M(x,y) \neq 0 \Rightarrow y \in S_i$$

Such matrices look like the picture on the right. We require $T_a \in \mathscr{M}_d$ for all $a \in A$: states from S_i should only reach states from S_i. Since $\mathscr{M}_d$ is closed under products (it actually forms a sub-semiring of $\mathscr{M}$), we deduce $T_u \in \mathscr{M}_d$ for all $u \in A^*$.



If $S_1 = S_2 = S$ then the requirement is void, as well as the optimization to be described below; if $S_1 \cap S_2 = \emptyset$ and $S_1 \cup S_2 = S$ then this corresponds to the case where $\mathcal{A}$ is a disjoint union of two NBWs. Intermediate cases are allowed. In practice if we want to test the equivalence between starting sets X and Y we will take as S_1 (resp. S_2) the set of accessible states from X (resp. Y).

For $i = 1, 2$, let $\mathscr{M}_i$ be the set of matrices indexed by S_i and let $\pi_i \colon \mathscr{M}_d \to \mathscr{M}_i$ be the obvious surjective semiring homomorphism. For all $M \in \mathscr{M}_d$, we have $\omega(M) \cap S_i = \omega(\pi_i(M))$. Define the following function $e' \colon \mathcal{P}(\mathscr{M}_d) \to \mathcal{P}(\mathscr{M}_d)$:

$$e'(\mathcal{M}) = \{N \mid \langle \pi_1(N), \pi_2(N) \rangle \in e(\{\langle \pi_1(M), \pi_2(M) \rangle \mid M \in \mathcal{M}\})\}$$

where $e(R)$ denotes the equivalence closure of a relation R, here for relations on $\mathscr{M}_1 \uplus \mathscr{M}_2$.

Like in the previous section, we will show by using Lemma 3, that when $\texttt{HKC}^\omega$ is restricted to starting sets $\langle X, Y \rangle \in \mathcal{P}(S_1) \times \mathcal{P}(S_2)$, it remains correct when using e' as an up-to technique on line 4 from Figure 2.3.

We need to work in a larger structure than sets of matrices. Moreover, we need to turn the set of discriminating sets into a relation. Set $U = \{1\} \times S_1 \cup \{2\} \times S_2$. Given a relation $\mathcal{E} \in \mathit{Rel}(U)$, define the following relation between $\mathcal{P}(S_1)$ and $\mathcal{P}(S_2)$:

$$X_1 \approx_{\mathcal{E}} X_2 \quad \text{if} \quad \forall \langle \langle i, D \rangle, \langle j, D' \rangle \rangle \in \mathcal{E},\ X_i \cap D = \emptyset \Leftrightarrow X_j \cap D' = \emptyset$$

(We define $\approx_{\mathcal{E}}$ as a relation between $\mathcal{P}(S_1)$ and $\mathcal{P}(S_2)$ because when starting with sets $X_1 \subseteq S_1$ and $X_2 \subseteq S_2$, $\texttt{HKC}$' will return such a relation.)

Set $\mathscr{M}'' = (\{1\} \times \mathscr{M}_1 \ \cup\ \{2\} \times \mathscr{M}_2)^2$, write iM for the pair $\langle i, M \rangle \in \{i\} \times \mathscr{M}_i$ and define a mixed product operation $\cdot \colon \mathscr{M}'' \times \mathscr{M}_d \to \mathscr{M}''$ by setting:

$$\langle iM, jN \rangle \cdot O = \langle i(M \cdot \pi_i(O)), j(N \cdot \pi_j(O)) \rangle$$

Now lift the functions r, s we defined after Lemma 3 to work on $\mathcal{P}(\mathscr{M}'')$:

$$\begin{aligned} r(R) &= \{\mathbf{M} \in \mathscr{M}'' \mid \forall a \in A,\ \mathbf{M} \cdot T_a \in R\} \\ d(R) &= \{\langle i\omega(M), j\omega(N) \rangle \mid \langle iM, jN \rangle \in R\} \\ s(R) &= \approx_{d(R)} \end{aligned}$$

Recall that e is the function taking the equivalence closure of a relation.

Proposition 2.6. The functions r, e and s satisfy the requirements of Lemma 3 (taking e for f).

Proof. For compatibility of e w.r.t. r ($e \circ r \leq r \circ e$), assume $\langle i_1 M_1, i_n M_n \rangle \in e(r(R))$. There are $(i_k, M_k)_{k \in [2..n[}$ such that for all $k < n$, either $\langle i_k M_k, i_{k+1} M_{k+1} \rangle \in r(R)$ or $\langle i_{k+1} M_{k+1}, i_k M_k \rangle \in r(R)$. We need to show that $\langle i_1 M_1, i_n M_n \rangle \in r(e(R))$. Let $a \in A$; for all $k < n$, either $\langle i_k M_k, i_{k+1} M_{k+1} \rangle \cdot T_a \in R$ or $\langle i_{k+1} M_{k+1}, i_k M_k \rangle \cdot T_a \in R$, which means by definition that $\langle i_k M_k \cdot \pi_k(T_a), i_{k+1} M_{k+1} \cdot \pi_{k+1}(T_a) \rangle \in R$ or $\langle i_{k+1} M_{k+1} \cdot \pi_{k+1}(T_a), i_k M_k \cdot \pi_k(T_a) \rangle \in R$.

Therefore, for all $a \in A$, $\langle i_1M_1{\cdot}\pi_1(T_a), i_nM_n{\cdot}\pi_n(T_a)\rangle \in e(R)$, which means $\langle i_1M_1, i_nM_n\rangle \in r(e(R))$, as required.

The function e is obviously extensive and idempotent, so that it only remains to show that $s \circ e \leq s$, i.e., for all R, $s(R) \subseteq s(e(R))$ (recall that we take reverse inclusions for the partial order $\mathcal{Y}$). Suppose $\langle X_1, X_2\rangle \in s(R)$, i.e., $X_1 \approx_{d(R)} X_2$, we have to show $\langle X_1, X_2\rangle \in s(e(R))$, i.e., $X_1 \approx_{d(e(R))} X_2$. Let $\langle i_1D_1, i_nD_n\rangle \in d(e(R))$. There are M_1, M_n such that $D_1 = \omega(M_1)$, $D_n = \omega(M_n)$, and $(i_k, M_k)_{k\in[2..n[}$ such that for all $k < n$, either $\langle i_kM_k, i_{k+1}M_{k+1}\rangle \in R$ or $\langle i_{k+1}M_{k+1}, i_kM_k\rangle \in R$. Since $X_1 \approx_{d(R)} X_2$, we deduce that for all $k < n$, either $X_{i_k} \cap \omega(M_k) = \emptyset \Leftrightarrow X_{i_{k+1}} \cap \omega(M_{k+1}) = 0$, or $X_{i_{k+1}} \cap \omega(M_{k+1}) = \emptyset \Leftrightarrow X_{i_k} \cap \omega(M_k) = 0$, which is just the same. By transitivity of logical equivalence, we deduce that $X_{i_1} \cap \omega(M_1) = \emptyset \Leftrightarrow X_{i_n} \cap \omega(M_n) = 0$, as required. □

Overloading the notation from Section 2.4.1, given a matrix $M \in \mathscr{M}_d$, write

$$\underline{M} = \langle 1\pi_1(M), 2\pi_2(M)\rangle \in \mathscr{M}''.$$

Then we have:

$$(1)\ \forall\ M, N \in \mathscr{M}_d,\ \underline{M \cdot N} = \underline{M} \cdot N \quad (2)\ \forall\ X_1 \subseteq S_1, X_2 \subseteq S_2,\ X_1 \sim_{\omega(M)} X_2 \text{ iff } X_1 \approx_{d(\underline{M})} X_2$$

The first property guarantees that when taking $x_0 = \{\underline{I}\}$, the x from Lemma 3 is the set $\left\{\underline{T_u} \mid u \in A^*\right\}$. The second property ensures that $s(x)$ properly discriminates the pairs provided by `HKC`' (R). By Lemma 3, so does $s(x')$, which can easily be shown to correspond to the computation with the optimized algorithm $\texttt{Discr}_{e'}$, where we use the up-to-equivalence technique to skip redundant matrices.

As in Hopcroft and Karp's algorithm [52], one can implement the up-to-equivalence test efficiently using an appropriate union-find data structure.

Example 2.4. When running this refined version of $\texttt{HKC}^\omega$ on the NBW over a single letter defined by the matrix T_a in Figure 2.8, the up-to-equivalence technique makes it possible to retain only 5 matrices of the monoid, although it contains 7 elements. Indeed, the matrix T_{a^5} is skipped because the pair of its components is in the equivalence closure of the set of pairs of components of already explored matrices. The explored part of the transition monoid and the skipped matrices are detailed in Figure 2.9 Note that T_{a^6} and T_{a^7} do not even need to be generated.

2.5 Conclusion and future work

We presented an algorithm for checking language equivalence of non-deterministic Büchi automata. This algorithm exploits advanced coinductive techniques to analyze the finite prefixes of the considered languages, through bisimulations up to congruence, as in the algorithm `HKC` for NFA. The periodic part of the considered languages is also analyzed coinductively, in order to compute the discriminating sets. Those sets make it possible to classify the periodic words accepted by the various states of the starting automaton, thus providing all the necessary information together with the analysis of the finite prefixes. The coinductive framework makes it possible to develop up-to techniques similar to the

Kept matrices

$$\begin{array}{cccccc}
u = & \epsilon & a & aa & aaa & aaaa \\
T_u = &
\begin{pmatrix} 1 & 0 & & & \\ 0 & 1 & & & \\ & & 1 & 0 & 0 \\ & & 0 & 1 & 0 \\ & & 0 & 0 & 1 \end{pmatrix} &
\begin{pmatrix} 0 & \star & & & \\ \star & 0 & & & \\ & & 0 & \star & 0 \\ & & 0 & 0 & \star \\ & & \star & 0 & 0 \end{pmatrix} &
\begin{pmatrix} \star & 0 & & & \\ 0 & \star & & & \\ & & 0 & 0 & \star \\ & & \star & 0 & 0 \\ & & 0 & \star & 0 \end{pmatrix} &
\begin{pmatrix} 0 & \star & & & \\ \star & 0 & & & \\ & & \star & 0 & 0 \\ & & 0 & \star & 0 \\ & & 0 & 0 & \star \end{pmatrix} &
\begin{pmatrix} \star & 0 & & & \\ 0 & \star & & & \\ & & 0 & \star & 0 \\ & & 0 & 0 & \star \\ & & \star & 0 & 0 \end{pmatrix}
\end{array}$$

Skipped matrix

$$T_{aaaaa} = \begin{pmatrix} 0 & \star & & & \\ \star & 0 & & & \\ & & 0 & 0 & \star \\ & & \star & 0 & 0 \\ & & 0 & \star & 0 \end{pmatrix} \quad \text{because} \quad \begin{pmatrix} 0 & \star \\ \star & 0 \end{pmatrix} \sim \begin{pmatrix} 0 & \star & 0 \\ 0 & 0 & \star \\ \star & 0 & 0 \end{pmatrix} \sim \begin{pmatrix} \star & 0 \\ 0 & \star \end{pmatrix} \sim \begin{pmatrix} 0 & 0 & \star \\ \star & 0 & 0 \\ 0 & \star & 0 \end{pmatrix}$$

Figure 2.9: Exploration of a transition monoid with up-to-equivalence technique.

ones used in HKC in order to compute the discriminating sets more efficiently. We provide two such techniques, namely coinduction up to unions (Section 2.4.1) and coinduction up to equivalence (Section 2.4.2). It is not clear to us whether the two techniques can be used at the same time.

We also want to investigate how to exploit techniques using simulation relations, which were successfully used in [35, 1, 2, 68] and which tend to nicely fit in the coinductive framework we exploit here [14, Section 5].

Our algorithm stems from the construction of Calbrix et al. [21], which we revisited using notions from [72] in Section 2.2. $\texttt{HKC}^\omega$ is rather close to *Ramsey-based* algorithms [37, 1] (as opposed to *rank-based* ones [63, 33, 34, 35]). In particular, our matrices are often called *super-graphs* in Ramsey-based algorithms. A key difference is that we focus on language equivalence, thus enabling stronger coinductive proof principles.

A prototype implementation is available at https://framagit.org/dpous/hkcw; it makes it possible to test several combinations of up-to techniques.

Chapter 3

Cyclic proofs and jumping automata

Abstract

We consider a fragment of a cyclic sequent proof system for Kleene algebra, and we see it as a computational device for recognizing languages of words. The starting proof system is linear and we show that it captures precisely the regular languages. When adding the standard contraction rule, the expressivity raises significantly: the system captures exactly the class of deterministic logspace languages. We prove this result by introducing as an intermediary model a new notion of multihead finite automata where heads can jump.

3.1 Introduction

In recent years there has been a surge of interest in the theory of non-wellfounded proofs. This is an approach to infinitary proof theory where proofs remain finitely branching but are permitted to be infinitely deep. A correctness criterion is usually required to guarantee consistency, typically some ω-regular condition on the infinite branches. Proofs whose graphs are regular trees are known as *cyclic* proofs; being finite objects, they can be communicated and checked, thus playing the role of traditional *inductive* proofs.

Such systems have been proposed for instance by Brotherston and Simpson in the context of first order logic with inductive predicates [17], as an alternative to the standard induction schemes. A natural question is whether specific cyclic and inductive proof systems have the same logical strength. The infinite descent principles associated to cyclic proofs are in general at least as powerful as the standard induction schemes and inductive proofs can usually be translated easily into cyclic ones (see, e.g., [17]), while the converse problem is a delicate problem. It was proven only recently that it holds in certain cases [12, 77], and that there are also cases where cyclic proofs are strictly more expressive [11].

Cyclic proof systems have also been used in the context of the μ-calculus [29], where we have inductive predicates (least fixpoints), but also coinductive predicates (greatest fixpoints), and alternation of those. Proof theoretical aspects such as cut-elimination were studied from the linear logic point of view [39, 31], and these systems were recently used to obtain constructive proofs of completeness for Kozen's axiomatisation [30, 5].

Building on these works, Das and Pous considered the simpler setting of Kleene algebra, and proposed a cyclic proof system for regular expression containments [27]. The key observation is that regular expressions can be seen as μ-calculus formulas using only a single form of fixpoint: the definition of Kleene star as a least fixpoint ($e^* = \mu x.1 + e \cdot x$). Their system is based on a non-commutative version of μMALL [31], and it is such that a sequent $e \vdash f$ is derivable if and only if the language of e is contained in that of f. This work eventually led to an alternative proof of left-handed completeness for Kleene algebra [26].

In the latter works, it is natural to consider regular expressions as datatypes [40], and proofs of language containments as total functions between those datatypes [46]. Such a computational interpretation of cyclic proofs was exploited to prove cut-elimination in [28].

We follow the same approach here, focusing on an even simpler setting: our sequents essentially have the shape $A^* \vdash \mathbb{B}$, where A is a finite alphabet and $\mathbb{B}$ is a type (or formula) for Boolean values. Cyclic proofs no longer correspond to language containments: they give rise to functions from words to Booleans, i.e., formal languages. We characterize the class of languages that arise from such proofs. In chapter 4 we will consider an enriched type system and more general sequents.

If we keep a purely linear proof system, as in [27, 28], we show that we obtain exactly the regular languages. In contrast, if we allow the contraction rule, we can express non-regular languages. We show that in this case, we obtain exactly the deterministic logarithmic space languages (DLogSpace). This is done by introducing a new class of automata, which we call *jumping multihead automata*[1]. Intuitively, when reading a word, a multihead automaton may only move its heads forward, letter by letter, while a jumping multihead automaton also has the possibility to let a given head jump to the position of another head. This gives the opportunity to record positions in the word, and to repeatedly analyze the suffixes starting from those positions. Cyclic proofs translate naturally into this new model that is in fact equivalent to the two-way multihead automata that were studied in the literature [48] and characterize DLogSpace.

Outline. We define our cyclic proof system and its computational interpretation in Section 3.2. Then we define jumping multihead automata and show they define the same languages as two-way multihead automata in Section 3.3. We prove the equivalence between the two models in Section 3.4 (Theorem 3.2), from which the characterizations of DLogSpace and regular languages follow. We discuss directions for future work in Section 3.5.

3.2 Infinite proofs and their semantics

We let a, b range over the letters of a fixed and finite alphabet A. We work with only three *types* (or *formulas*): the type $\mathbb{B}$ of Boolean values, the type A of letters, and the

[1] This new class should not be confused with the *jumping finite automata* introduced by Meduna and Zemek [69], which are not multihead.

$$w\ \frac{E,F\vdash\mathbb{B}}{E,e,F\vdash\mathbb{B}} \qquad c\ \frac{E,e,e,F\vdash\mathbb{B}}{E,e,F\vdash\mathbb{B}} \qquad t\ \frac{}{\vdash\mathbb{B}} \qquad f\ \frac{}{\vdash\mathbb{B}}$$

$$A\ \frac{(E,F\vdash\mathbb{B})_{a\in A}}{E,A,F\vdash\mathbb{B}} \qquad *\ \frac{E,F\vdash\mathbb{B} \qquad E,A,A^*,F\vdash\mathbb{B}}{E,A^*,F\vdash\mathbb{B}}$$

Figure 3.1: The rules of C for formal languages.

type A^* of words. We let e, f range over types and we let $[e]$ denote the expected set associated to a type e. We let E, F range over finite sequences of types. Given such a sequence $E = e_1, \dots, e_n$, we write $[E]$ for the Cartesian product $[e_1] \times \cdots \times [e_n]$.

We define a sequent proof system, where sequents have the shape $E \vdash \mathbb{B}$, and where proofs of such sequents denote functions from $[E]$ to $\mathbb{B}$, i.e., subsets of $[E]$.

3.2.1 Infinite proofs

We now define the cyclic proof system whose six inference rules are given in Figure 3.1. In addition to two *structural rules* (weakening and contraction), we have a left introduction rule for each type, and two right introduction rules for Boolean constants. Note that there is no exchange rule, which explains why the structural and left introduction rules use two sequences E and F rather than a single one.

The left introduction rule for type A^* corresponds to an unfolding rule, looking at A^* as the least fixpoint expression $\mu X.(1 + A \times X)$ (e.g., from μ-calculus). The left premiss intuitively corresponds to the case of an empty list, while the right premiss covers the case of a non-empty list. Except from weakening and contraction, those rules form a very small fragment of those used for Kleene algebra in [28] (interpreting A as a sum $1 + \cdots + 1$ with $|A|$ elements and $\mathbb{B}$ as the binary sum $1 + 1$).

Note that we are not interested in *provability* in this chapter: every sequent can be derived trivially, using weakenings and one of the two right introduction rules. The objects of interest are the proofs themselves; this explains why we have two axioms for proving the sequent $\vdash \mathbb{B}$: they correspond to two different proofs.

We set $B = A \uplus \{0, 1\}$. A (possibly infinite) *tree* is a non-empty and prefix-closed subset of B^*, which we view with the root, ϵ, at the bottom. We let v range over elements of B^*, which we call *addresses*.

Definition 3.1. A *preproof* is a labeling π of a tree by sequents such that, for every node v with children $v_1, \dots v_n$, the expression $\dfrac{\pi(v_1) \ \cdots \ \pi(v_n)}{\pi(v)}$ is an instance of a rule from Figure 3.1. A preproof is *regular* if it has finitely many distinct subtrees, i.e., it can be viewed as the unfolding of a finite graph. A preproof is *affine* if it does not use the c-rule.

If π is a preproof, we note $Addr(\pi)$ its set of addresses, i.e., its underlying tree. The formulas appearing in lists E, F of any rule instance are called *auxiliary formulas.* The non auxiliary formula appearing in the conclusion of a rule is called the *principal formula.*

A $*$ *address* in a preproof π is an address v which is the conclusion of a $*$ rule in π.

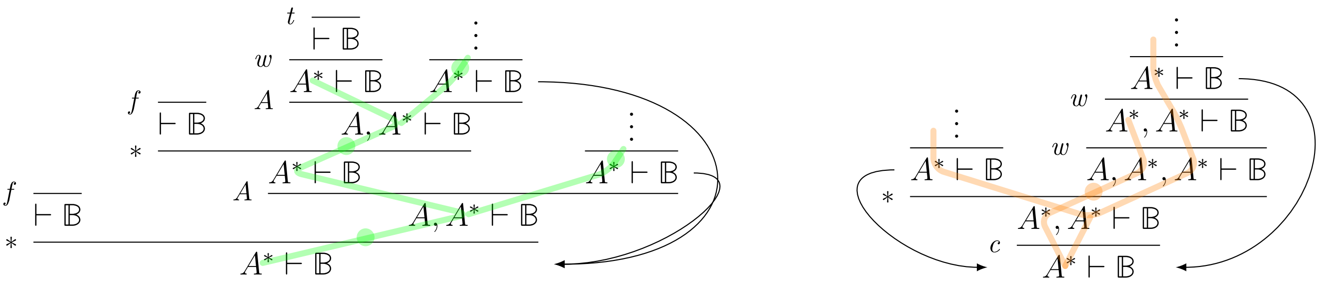


Figure 3.2: Two regular preproofs; only the one on the left is valid.

Two examples of regular preproofs are depicted in Figure 3.2. The alphabet A is assumed to have exactly two elements, so that the A rule is binary. Backpointers are used to denote circularity: the actual preproofs are obtained by unfolding the graphs. The preproof on the right might look suspicious: it never uses the axioms t or f. In fact, only the one on the left satisfies the validity criterion which we define below. Before doing so, we need to define a notion of thread, which are the branches of the shaded trees depicted on the preproofs. Intuitively a thread follows a star formula occurrence along a branch of the proof. First we need to define parentship and ancestor relations.

Definition 3.2. A *position* in a preproof π is a pair $\langle v, i\rangle$ consisting of an address v and an index $i \in [0; |E|-1]$, where $\pi(v) = E \vdash \mathbb{B}$ and E_i is a star formula. A position $\langle v, i\rangle$ is the *parent* of a position $\langle w, j\rangle$ if $|v| = |w| + 1$ and, looking at the rule applied at address w the two positions point at the same place in the lists E, F of auxiliary formulas, or at the formula e when this is the contraction rule, or at the principal formula A^* when this is the $*$ rule and $v = w1$. We write $\langle v, i\rangle \lhd \langle w, j\rangle$ in the former cases, and $\langle v, i\rangle \mathrel{\dot{\lhd}} \langle w, j\rangle$ in the latter case. We say that $\langle v, i\rangle$ is an *ancestor* of $\langle w, j\rangle$ when those positions are related by the transitive closure of the parentship relation.

The graph of the parentship relation is depicted in Figure 3.2 using shaded thick lines and an additional bullet to indicate when we pass principal star steps ($\dot{\lhd}$). Note that in the $*$ rule, the occurrence of A in the right premiss is not a parent of A^* in the conclusion.

Remark 3.1. When working with trees, it is common in computer science to have the convention that a node in a tree has at most one parent, and many children: the root of a tree is the ancestor of all its leaves. Be careful that we use the opposite convention in the present thesis, following the tradition in proof theory [20]: the ancestors are to be found towards the leaves. This convention also matches the intuition from family trees.

We can finally define threads and the validity criterion.

Definition 3.3. A *thread* is a possibly infinite branch of the ancestry graph. A thread is *principal* when it visits a $*$ rule through its principal formula, *spectator* if it is never principal and *valid* if it is principal infinitely often.

In the first preproof of Figure 3.2, the infinite green thread

$$\langle \epsilon, 0\rangle \mathrel{\dot{\rhd}} \langle 1, 1\rangle \rhd \langle 11, 0\rangle \mathrel{\dot{\rhd}} \langle 111, 1\rangle \rhd \langle 1111, 0\rangle \dots$$

is valid, as well as every other infinite thread. There is no valid thread in the second preproof: taking a principal step forces the thread to terminate and infinite threads along the infinite branches are all spectator.

Definition 3.4. A preproof is *valid* if every infinite branch contains a valid thread. A *proof* is a valid preproof. We write $\pi : E \vdash \mathbb{B}$ when π is a proof whose root is labeled by $E \vdash \mathbb{B}$.

In the examples from Figure 3.2, only the preproof on the left is valid, thanks to the infinite green thread. The second preproof is invalid: infinite threads along the (infinitely many) infinite branches are never principal.

This validity criterion is essentially the same as for the system LKA [28], which in turn is an instance of the one used for μMALL [31]: we just had to extend the notion of ancestry to cover the contraction rule. Note however that the presence of this rule induces some subtleties. For instance, while in the cut-free fragment of LKA, a preproof is valid if and only if it is *fair* (i.e., every infinite branch contains infinitely many $*$ steps [28, Proposition 8]), this is no longer true with contraction: the second preproof from Figure 3.2 is fair and invalid.

In the affine case, due to the fragment we consider here, and since we do not include cut, the situation is actually trivial:

Proposition 3.1. Every affine preproof is valid.

Proof. Except for the contraction rule and the right premiss of the $*$ rule, the length of a sequent strictly decreases when moving from the conclusion of a rule to one of its premisses. Therefore, every infinite branch of an affine preproof must pass through the right of a $*$ rule infinitely often. By the subformula property, the principal (star) formulas of these steps must be ancestors of star formulas in the conclusion of the preproof. Since they are finitely many, at least one of the star formulas from the conclusion gives rise to a valid thread. □

3.2.2 Computational interpretation of infinite proofs

We now show how to interpret a proof $\pi : E \vdash \mathbb{B}$ as a function $[\pi] : [E] \to \mathbb{B}$. Since proofs are not well-founded, we cannot reason directly by induction on proofs. We use instead the following relation on computations, which we prove to be well-founded thanks to the validity criterion.

Definition 3.5. A *computation* in a fixed proof π is a pair $\langle v, s\rangle$ consisting of an address v of π with $\pi(v) = E \vdash \mathbb{B}$, and a value $s \in [E]$

Given two computations, we write $\langle v, s\rangle \prec \langle w, t\rangle$ when

1. $|v| = |w| + 1$,

2. for every i, j such that $\langle v, i\rangle \lhd \langle w, j\rangle$, we have $s_i = t_j$, and

3. for every i, j such that $\langle v, i\rangle \mathrel{\ooalign{$\lhd$\cr\hidewidth$\cdot$\hidewidth}} \langle w, j\rangle$, we have $|s_i| < |t_j|$.

The first condition states that the subproof at address v should be one of the premisses of the subproof at w; the second condition states that the values assigned to star formulas should remain the same along auxiliary steps; the third condition ensures that they actually decrease in length along principal steps.

Lemma 6. *The relation $\prec$ on computations is well-founded.*

Proof. Suppose by contradiction that there exists an infinite descending sequence. By condition 1/, this sequence corresponds to an infinite branch of π. By validity, this branch must contain a thread which is principal infinitely many times. This thread contradicts conditions 2/ and 3/ since we would obtain an infinite sequence of lists of decreasing length. □

Definition 3.6. The *return value* $[v](s)$ of a computation $\langle v, s\rangle$ with $\pi(v) = E \vdash \mathbb{B}$ is a Boolean defined by well-founded induction on $\prec$ and case analysis on the rule used at address v.

$$\begin{array}{rl} w : & [v](s,x,t) \triangleq [v0](s,t) \\ c : & [v](s,x,t) \triangleq [v0](s,x,x,t) \\ t : & [v]() \triangleq \mathbf{tt} \\ f : & [v]() \triangleq \mathbf{ff} \end{array}$$

$$\begin{array}{rl} A : & [v](s,a,t) \triangleq [va](s,t) \\ * : & [v](s,l,t) \text{ is defined by case analysis on } l\text{:} \end{array}$$

- $[v](s,\epsilon,t) \triangleq [v0](s,t)$
- $[v](s,x :: q,t) \triangleq [v1](s,x,q,t)$

In each case, the recursive calls are made on strictly smaller computations: they occur on direct subproofs, the values associated to auxiliary formulas are left unchanged, and in the second subcase of the $*$ case, the length of the list associated to the principal formula decreases by one.

Definition 3.7. The semantics of a proof $\pi : E \vdash \mathbb{B}$ is the function $[\pi] \colon s \mapsto [\epsilon](s)$.

(Note that we could give a simpler definition of the semantics for affine proofs by reasoning on the total size of the arguments; such an approach however breaks in presence of contraction.)

Let us compute the semantics of the first (and only) proof in Figure 3.2. Recall that A has two elements in this example, so set $A = \{a, b\}$ (and thus $B = \{0, 1, a, b\}$), and let us use a (resp. b) to navigate to the left (resp. right) premiss of the A rule. Starting from words ab and aab, we get the two computations on the left below:

$$\begin{array}{ll} & [\epsilon](ab) \\ = & [1](a,b) \\ = & [1a](b) \\ = & [1a1](b,\epsilon) \\ = & [1a1b](\epsilon) \\ = & [1a1b0]() \\ = & \mathbf{ff} \end{array} \qquad \begin{array}{ll} & [\epsilon](aab) \\ = & [1](a,ab) \\ = & [1a](ab) \\ = & [1a1](a,b) \\ = & [1a1a](b) \\ = & [1a1a0]() \\ = & \mathbf{tt} \end{array} \qquad \begin{array}{rl} [\epsilon](\epsilon) = & \mathbf{ff} \\ [\epsilon](au) = & [\epsilon](u) \\ [\epsilon](bu) = & [1a](u) \\ & \\ [1a](\epsilon) = & \mathbf{ff} \\ [1a](au) = & \mathbf{tt} \\ [1a](bu) = & [\epsilon](u) \end{array}$$

Using the fact that the subproofs at addresses ϵ, $1a$ and $1a1b$ are identical, we can also deduce the equations displayed on the right, which almost correspond to the transition

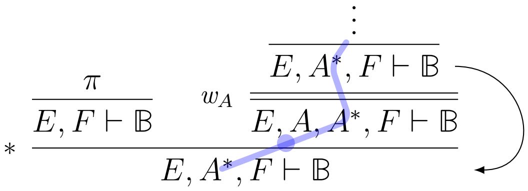

⋮
π
w_A
$E, A^*, F \vdash \mathbb{B}$
$E, F \vdash \mathbb{B}$
$E, A, A^*, F \vdash \mathbb{B}$
$*$
$E, A^*, F \vdash \mathbb{B}$

Figure 3.3: Weakening stars (Proposition 3.2).

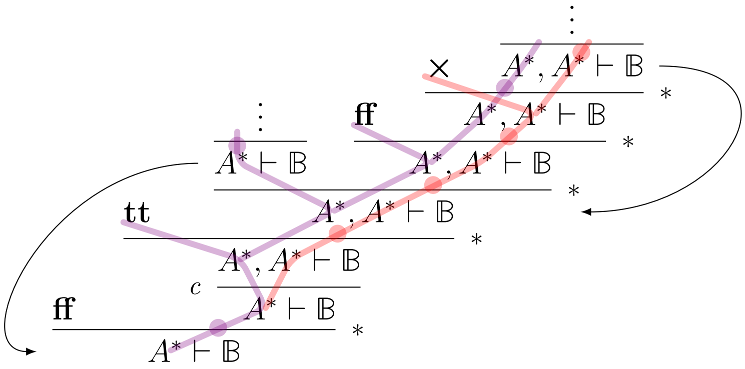

⋮
×
$A^*, A^* \vdash \mathbb{B}$
$*$
⋮
ff
$A^*, A^* \vdash \mathbb{B}$
$*$
$A^* \vdash \mathbb{B}$
$A^*, A^* \vdash \mathbb{B}$
$*$
tt
$A^*, A^* \vdash \mathbb{B}$
$*$
$A^*, A^* \vdash \mathbb{B}$
c
ff
$A^* \vdash \mathbb{B}$
$*$
$A^* \vdash \mathbb{B}$

Figure 3.4: A regular proof for $\left\{a^{2^n} \mid n \in \mathbb{N}\right\}$.

table of a deterministic automaton with two states ϵ and $1a$. This is not strictly speaking a deterministic automaton because of the fifth line: when reading an a, the state $1a$ decides to accept immediately, whatever the remainder of the word. We can nevertheless deduce from those equations that ϵ recognizes the language A^*aaA^*.

Trying to perform such computations on the invalid preproof on the right in Figure 3.2 gives rise to non-terminating behaviors, e.g., $[\epsilon](\epsilon) \rightsquigarrow [0](\epsilon,\epsilon) \rightsquigarrow [00](\epsilon) \rightsquigarrow \dots$ and $[\epsilon](x :: q) \rightsquigarrow [0](x :: q, x :: q) \rightsquigarrow [01](x, q, x :: q) \rightsquigarrow [010](q, x :: q) \rightsquigarrow [0100](x :: q) \rightsquigarrow \dots$.

Before studying a more involved example, we prove the following property:

Proposition 3.2. The weakening rule (w) is derivable in a way that respects regularity, affinity, existing threads, and the semantics.

Proof. When the weakened formula is A, it suffices to apply the A rule and to use the starting proof $|A|$ times. When the weakened formula is A^*, assuming a proof $\pi : E, F \vdash \mathbb{B}$, we construct the proof in Figure 3.3. The step marked with w_A is the previously derived weakening on A. The preproof is valid because this step does preserve the blue thread. □

As a consequence, the full proof system is equivalent to the one without weakening. We shall see that the system would remain equally expressive with the addition of an exchange rule (see Remark 3.3 below), but that the contraction rule instead plays a crucial role and changes the expressive power, going from regular languages to DLogSpace languages.

Let us conclude this section with an example beyond regular languages: we give in Figure 3.4 a proof whose semantics is the language of words over a single letter alphabet, whose length is a power of two (a language which is not even context-free). Since the alphabet has a single letter, the A rule becomes a form of weakening, and we apply it implicitly after each $*$ step. We also abbreviate subproofs consisting of a sequence of weakenings followed by one of the two axioms by **tt**, **ff**, or just × when it does not matter whether we return true or false.

Writing n for the word of length n and executing the proof on small numbers, we observe

$$
\begin{aligned}
&[\epsilon](0) = [0]() = \mathbf{ff}\\
&[\epsilon](1) = [1](0) = [10](0,0) = [100](0) = \mathbf{tt}\\
&[\epsilon](2) = [1](1) = [10](1,1) = [101](1,0) = [1010](1) = [\epsilon](1) = \mathbf{tt}\\
&[\epsilon](3) = [1](2) = [10](2,2) = [101](2,1) = [1011](2,0) = \mathbf{ff}\\
&[\epsilon](4) = [1](3) = [10](3,3) = [101](3,2) = [1011](3,1) = [10111](3,0) = [101111](2,0)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad = [101](2,0) = [1010](2) = [\epsilon](2) = \mathbf{tt}
\end{aligned}
$$

More generally, the idea consists in checking that the given number can be divided by two repeatedly, until we get 1. To divide a number represented in unary notation by two, we copy that number using the contraction rule, and we consume one of the copies twice as fast as the other one (through the three instances of the $*$ rule used at addresses 101, 1011, and 10111); if we reach the end of one copy, then the number was even, the other copy precisely contains its half, and we can proceed recursively (through the backpointer on the left), otherwise the number was odd and we can reject. The subproof at address 101110 is never explored: we would be in a situation where the slowly consumed copy gets empty before the other one.

Finally note that every (even undecidable) language can be represented using an infinite (in general non regular) proof: apply the left introduction rules eagerly, and fill in the left premisses of the $*$ rules using the appropriate axiom.

3.3 Jumping multihead automata

Now we introduce the model of Jumping Multihead Automata (JMA) and establish its equivalence with the two-way multihead automata model [48]. We will give direct translations between JMA and cyclic proofs in Section 3.4.

3.3.1 Definition and semantics of JMAs

Let A be a finite alphabet and $\lhd \notin A$ be a fresh symbol. We note $A_\lhd = A \uplus \{\lhd\}$.

Definition 3.8. A jumping multihead automaton (JMA) is a tuple $\mathcal{M} = \langle S, k, s_0, s_{acc}, s_{rej}, \delta\rangle$ where:

- S is a finite set of states;
- $k \in \mathbb{N}$ is the number of heads;
- $s_0 \in S$ is the initial state;
- $s_{acc} \in S$ and $s_{rej} \in S$ are final states, respectively accepting and rejecting;
- $\delta : S_{trans} \times (A_\lhd)^k \longrightarrow S \times Act^k$ is the deterministic transition function, where $S_{trans} \triangleq S \backslash \{s_{acc}, s_{rej}\}$ is the set of non-final states, and $Act \triangleq \{\text{◌}, \blacktriangleright\!|\} \uplus \{J_1, J_2, \dots, J_k\}$.

In the transition function, symbols ◌ and ⧐ stand for "stay in place" and "move forward" respectively, and action J_i stands for "jump to the position of head number i". Intuitively, if the machine is in state s, each head j reads letter $\vec{a}(j)$, and $\delta(s, \vec{a}) = (s', \alpha)$, then the machine goes to state s' and each head j performs the action $\alpha(j)$. Accordingly, to guarantee that the automaton does not try to go beyond the end marker of the word, we require that if $\delta(s, \vec{a}) = (s', \alpha)$, then for all $j \in [\![1, k]\!]$ with $\vec{a}(j) = \triangleleft$ we have $\alpha(j) \neq$ ⧐.

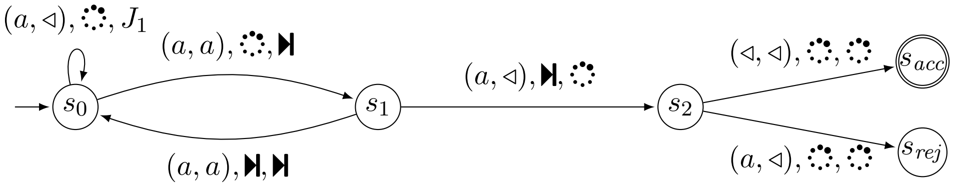
A *configuration* of a JMA $\mathcal{M} = \langle S, k, s_0, s_{acc}, s_{rej}, \delta \rangle$ is a triple $c = (w, s, p)$ where w is the input word, $s \in S$ is the current state, and $p = (p_1, \ldots, p_k) \in [\![0, |w|]\!]^k$ gives the current head positions. If the position p_i is $|w|$ then the head i is scanning the symbol $\triangleleft$.

The initial configuration on an input word w is $(w, s_0, (0, \ldots, 0))$. Let $w = a_0 a_1 \ldots a_{n-1}$ be the input and $a_n = \triangleleft$. Let $(w, s, (p_1, \ldots, p_k))$ be a configuration with $s \in S_{trans}$, and $(s', (x_1, \ldots, x_k)) = \delta(s, (a_{p_0}, \ldots a_{p_k}))$ be given by the transition function. Then the successor configuration is defined by $(w, s', (p'_1, \ldots, p'_k))$, where for all $i \in [\![1, k]\!]$ p'_i depends on x_i in the following way:

(1) $p'_i = p_i$ if $x_i =$ ◌ (2) $p'_i = p_i + 1$ if $x_i =$ ⧐ (3) $p'_i = p_j$ if $x_i = J_j$

A configuration (w, s, p) is *final* if $s \in \{s_{acc}, s_{rej}\}$. It is accepting (resp., rejecting) if $s = s_{acc}$ (resp., $s = s_{rej}$). A *run* of a JMA $\mathcal{M}$ on w is a sequence of configurations $c_0, c_1, \ldots, c_r$ on w where c_0 is the initial configuration, and c_{i+1} is the successor configuration of c_i for all i. If c_r is rejecting (resp. accepting), we say that the run is rejecting (resp. accepting). We say that $\mathcal{M}$ *terminates* on w if there is a (maximal, finite) run of $\mathcal{M}$ on w ending in a final configuration (either accepting or rejecting). The *language* of $\mathcal{M}$, noted $L(\mathcal{M})$, is the set of finite words leading to an accepting run in $\mathcal{M}$.

Example 3.1. The language $L = \{a^{2^n} \mid n \in \mathbb{N}\}$ can be recognized by the following JMA with two heads. (Missing transitions all go to the rejecting final state.)


$(a, \triangleleft)$, ◌, J_1
(a, a), ◌, ⧐
$(a, \triangleleft)$, ⧐, ◌
$(\triangleleft, \triangleleft)$, ◌, ◌
s_0
s_1
s_2
s_{acc}
s_{rej}
(a, a), ⧐, ⧐
$(a, \triangleleft)$, ◌, ◌

The idea behind the automaton is similar to one for the proof given in Figure 3.4: one head advances at twice the speed of the other. When the fast head reaches the end of the word, it either rejects if the length is odd and at least 2, or jumps to the position of the slow head located in the middle of the word. From there, the automaton proceeds recursively.

Notice that on an input word u, three scenarios are possible: the automaton accepts by reaching s_{acc}, rejects by reaching s_{rej}, or rejects by looping forever. In order to translate JMAs into cyclic proofs, whose validity criterion ensures termination, it is convenient to forbid the last scenario. We ensure such a property by a syntactic restriction on the transition structure of JMAs.

Definition 3.9. The *transition graph* of a JMA $\mathcal{M} = \langle S, k, s_0, s_{acc}, s_{rej}, \delta \rangle$ is the labeled graph $G_{\mathcal{M}} = (S, E)$, where the vertices are states S, and the set of edges is $E \subseteq S \times S \times Act^k$, defined by $E = \{(s, s', \alpha) \mid \exists \vec{a} \in (A_\triangleleft)^k, \delta(s, \vec{a}) = (s', \alpha)\}$.

A JMA $\mathcal{M}$ is *progressing* if for every cycle $e_1 e_2 \dots e_l$ in its transition graph, where $e_i = (s_i, s_{i+1}, \alpha_i)$ for each $i \in [\![1, l]\!]$ and $s_{l+1} = s_1$, there exists a head $j \in [\![1, k]\!]$ with $\alpha_1(j)\alpha_2(j) \dots \alpha_l(j) \in (◌^* \cdot ⧐ \cdot ◌^*)^+$.

(Intuitively we require that for every loop, one of the heads does not jump during this loop and moves forward at least once.)

The JMA from Example 3.1 always terminates, but it is not progressing due to the loop on the initial state. It could easily be modified into a progressing JMA by introducing a new intermediary state instead of looping on s_0. In fact, even in cases where a JMA can indefinitely loop on some inputs, one can always turn it into a progressing one recognizing the same language. Hence all JMAs are assumed to be progressing from now on.

Lemma 7. *Every JMA can be converted into a progressing JMA with the same language.*

Proof. Let $\mathcal{M} = \langle S, k, s_0, s_{acc}, s_{rej}, \delta \rangle$ be a JMA. We want to construct a progressing JMA $\mathcal{M}'$ such that $L(\mathcal{M}) = L(\mathcal{M}')$. We use the fact that the number of possible configurations on a given word w is bounded polynomially in the length of w. We add heads to the JMA that just advance counting up until this bound, making the JMA progressing.

For all w such that $\mathcal{M}$ terminates on w, the run $c_0, c_1, \dots, c_r$ of $\mathcal{M}$ on w is such that $r < |S||w|^k$. Indeed, for a given word w, there are only $|S||w|^k$ distinct configurations. So if there is an accepting run $c_0, c_1, \dots, c_r$ on w of length greater than $|S||w|^k$ then necessarily there exists $i \neq j$ such that $c_i = c_j$, meaning the automaton $\mathcal{M}$ has entered a loop. Since $\mathcal{M}$ is deterministic, it will stay in this loop forever, which contradicts the fact that $\mathcal{M}$ terminates on w.

We construct $\mathcal{M}'$ by adding $k + 1$ heads to $\mathcal{M}$. The $(k + 1)^{th}$ head stays at the beginning of the word to allow the other heads to jump back to it. The first added head reads the word letter by letter; when it reaches the final marker $\triangleleft$, it jumps back to the beginning and starts again. Then for all $i < k$, the $(i + 1)^{th}$ head advances each time the i^{th} reads the end symbol $\triangleleft$ and jumps back to the beginning every time it reaches the end. Note that the i^{th} head takes exactly $|w|^i$ steps to read the whole word. The state space S' of $\mathcal{M}'$ is defined as $S \times [\![1, |S|]\!]$, with initial state $s'_0 = (s_0, 1)$. The second component of S' will be called the *counter*. Each time the k^{th} head reads the end symbol, we increment the counter in addition to jumping back to the beginning. If the counter reaches $|S|$ and needs to be incremented, the automaton $\mathcal{M}'$ enters state s_{rej} and rejects the input.

$\mathcal{M}'$ is progressing. Indeed if there is a loop $e_1 e_2 \dots e_l$ with $e_i = (s_i, s_{i+1}, \alpha_i)$ in $G_{\mathcal{M}'}$ then it corresponds to a loop of $\mathcal{M}$ with a fixed counter value, so the k^{th} head never jumps back. Let $i = max\{j \in [\![1, k]\!] \mid \exists t \; \alpha_t(j) = ⧐\}$. Then the i^{th} head never jumps back. In fact if $i < k$ and the i^{th} head had jumped back then the $(i + 1)^{th}$ head would have advanced, which contradicts the maximality of i.

$\mathcal{M}'$ recognizes exactly $L(\mathcal{M})$. Indeed if $w \in L(\mathcal{M})$ then there exists an accepting run of $\mathcal{M}$ of length less than $|S||w|^k$, this run also exists in $\mathcal{M}'$ and so $w \in L(\mathcal{M}')$. If $w \notin L(\mathcal{M})$ then either $\mathcal{M}$ rejects it in less than $|S||w|^k$ steps, in which case $\mathcal{M}'$ also rejects it, or $\mathcal{M}'$ will reject the word after $|S||w|^k + 1$ steps. □

Lemma 8. *Given a JMA $\mathcal{M}$, we can check in* NL *whether $\mathcal{M}$ is progressing. If $\mathcal{M}$ is progressing, then it terminates on all words.*

Proof. To witness that an input JMA $\mathcal{M}$ is not progressing, it suffices to guess on-the-fly a loop in the transition graph violating the condition defining progressing automata. Notice that the memory needed to verify that the loop violates the condition is $\Theta(k)$, since for each head, one needs to remember whether it has already violated the condition, and if not whether it has already moved to the right. The transition table of $\mathcal{M}$ is of size exponential in k, so this memory of $\Theta(k)$ is indeed logarithmic in the input size. Since NL $=$ coNL, this yields a NL algorithm to verify that a JMA is progressing.

We now show that if $\mathcal{M}$ is a progressing JMA, it terminates on all words. Assume by contradiction that there is a word w of length n such that $\mathcal{M}$ does not terminate on w. Since its transition function is total, it means that $\mathcal{M}$ has an infinite run ρ on w. Let $I \subseteq [\![1, k]\!]$ be the set of heads that advance infinitely many times in ρ. We can choose a factor τ of ρ such that each head from I advances at least $n+2$ times in τ, heads not in I do not advance in τ, and additionally the first and last state of τ are identical. Since τ corresponds to a loop in the transition graph of $\mathcal{M}$, and $\mathcal{M}$ is progressing, there must be a head $j \in I$ that does not jump during τ. This means this head j advances $n+2$ times without jumping, which is impossible on the word w of length n. We reached a contradiction, thereby proving that a progressing JMA must terminate on all words. □

3.3.2 Expressive power of JMAs

Write $\mathit{JMA}(k)$ for the set of languages expressible by a progressing JMA with k heads. JMAs encode precisely the DLogSpace languages; one-head JMAs capture exactly the regular languages.

Lemma 9. *$\mathit{JMA}(1) = \textsc{Reg}$.*

Proof. Every deterministic automaton translates directly into a JMA with a single head. Conversely, a progressing JMA with a single head cannot jump or stay in place, so that it suffices to extend the transition table to make the two final states sink states. (Moreover note that a JMA with a single head can be transformed into a progressing one without adding new heads: it suffices to add a sink state.) □

Theorem 3.1. $\bigcup_{k \geq 1} \mathit{JMA}(k) = \textsc{DLogSpace}$

The forward direction of Theorem 3.1 is relatively easy:

Lemma 10. $\bigcup_{k \geq 1} \mathit{JMA}(k) \subseteq \textsc{DLogSpace}$.

Proof. It is straightforward to translate a JMA with k heads into a Turing machine using space $O(\log^k(n))$, by remembering the position of the heads. □

To obtain the other direction of Theorem 3.1, we go through a notion of (non-jumping) multihead automata that has already been investigated in the literature [48]. They consist of automata with a fixed number of heads (k) that can either only go from left to right, (one-way automata, $1\mathit{DFA}(k)$), or in both directions (two-way automata, $2\mathit{DFA}(k)$). We briefly compare JMAs to those automata, starting with the one-way case.

First of all, it is clear that for all $k \geq 1$, $1\mathit{DFA}(k) \subseteq \mathit{JMA}(k)$ (in particular, because $1\mathit{DFA}$s can be assumed to be progressing without increasing the number of heads).

Remark 3.2. Since emptiness, universality, regularity, inclusion and equivalence are undecidable for $1\mathit{DFA}$s with 2 heads [48], these problems are also undecidable for JMAs with 2 heads.

Concerning two-way automata ($2DFA$) it is known that $\bigcup_{k\geq 1} 2DFA(k) = \text{DLogSpace}$ [48], so that by Lemma 10 every JMA can be translated into a deterministic multihead two-way automaton, not necessarily preserving the number of heads. We prove the converse direction with a direct transformation of a two-way multihead automaton into a jumping multihead one. We start by giving an example of a seemingly 2-way behavior that can be simulated by JMAs.

Example 3.2. The palindrom language $\mathcal{L} = \{w \mid w \in A^* \wedge w = w^R\}$ where w^R denotes the reverse of w is recognizable by a JMA with 4 heads. Let us call these 4 heads h_0, h_{lin}, $h_{\overline{lin}}$ and h_{temp} in the following. The head h_0, will always stay at the beginning of the word to allow other heads to jump back to this position. The head h_{lin} will linearly read the word from left to right. The head $h_{\overline{lin}}$ will simulate the linear reading of the word from right to left by doing multiple jumps. The head h_{temp} will help $h_{\overline{lin}}$ to find the position of its next move.

In the initial configuration of the automaton, all the reading heads are locating on the first letter of the input word. Then, each time h_{lin} is reading the i-th letter we do as follow:

– $h_{\overline{lin}}$ and h_{temp} jump respectively on h_0 and h_{lin} and then h_{temp} moves one step right. They are exactly i letters apart.

– $h_{\overline{lin}}$ and h_{temp} move right synchronously until h_{temp} is reading the end symbol of the word $\triangleleft$ (*i.e.* the $(n+1)$-th letter). Then $h_{\overline{lin}}$ is reading the $(n+1-i)$-th letter.

The automaton compares the letters read by h_{lin} and $h_{\overline{lin}}$. If they are different it halts and rejects the word. Otherwise h_{lin} moves right. If it reaches the end symbol of the word $\triangleleft$ then the automaton halts and accepts, otherwise we do the same process again.

The automaton recognizing the palindrom language works by simulating a head that reads the word from the right to the left thanks to an auxiliary head and multiple jumps. We can generalize this process to simulate any two-way multihead automaton.

Lemma 11. *Let $\mathcal{M}$ be a two-way (deterministic) multihead automaton with k heads. Then there exists $\mathcal{M}'$ a JMA with $2k+2$ heads such that $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{M}')$.*

Proof. We construct $\mathcal{M}'$ as follows.

For each head h of $\mathcal{M}$ we add another reading head h' that will be always kept at the symmetric position of h (*i.e.* when h will be reading the i-th letter, h' would be reading the $n-i+1$-th letter). Moreover we add a head h_0 that will always stay at the beginning of the word and a head h_{temp} that we will use to make a given head move one letter left.

Each time a head h moves (left or right) in $\mathcal{M}$, in $\mathcal{M}'$ the corresponding head h will do the same move and its symmetric head h' the inverse move. We always begin by doing the right move and then use the fact that we have a head at the symmetric position to find the position where the second head should move with h_{temp}. For simplicity let's assume that h moves right so that h' should move left. If h is reading the i-th letter then we have to find the $n-i+1$-th letter. We proceed as follow:

– h' and h_{temp} jump respectively on h_0 and h and then h_{temp} moves one step right. They are exactly i letters apart.

- h' and h_{temp} move right synchronously until h_{temp} is reading the end symbol of the word $\lhd$ (*i.e.* the $n+1$-th letter). Then h' is reading the $n+1-i$-th letter.

□

Since it was shown in [92] that 2*DFA*s are strictly more expressive than 1*DFA*s, it is also the case that JMAs are strictly more expressive than 1*DFA*s. We refine this result here, and show that JMAs with 2 heads can already compute languages that are not computable by 1*DFA*s with any number of heads.

Proposition 3.3. For all $k \geq 1$, $JMA(2) \nsubseteq 1DFA(k)$.

Proof. It is proven in [92] that $(1DFA(k))_{k\in\mathbb{N}}$ forms a strict hierarchy, by defining a language L_b that is recognizable by a 1*DFA* with k heads if and only if $b < \binom{k}{2}$. We slightly modify these languages so that they become expressible with a two-head JMA while keeping the previous characterization for 1*DFA*.

We define a language L_{b+} that is recognizable by a JMA with only 2 heads while it can be recognized by a 1*DFA* with k heads if and only if $b < \binom{k}{2}$. To do so, we start from the language defined in [92] to prove the hierarchy theorem for 1*DFA*s. We slightly modify this language in order to add some information that helps an automaton with jumping heads but not one with only multiple heads.

In [92], it is proven that the language $L_b = \{w_1\$\ldots\$w_{2b} \mid \forall i, w_i = w_{2b+1-i} \in A^*\}$ is recognizable by a 1FA with k heads if and only if $b < \binom{k}{2}$ (with \$ a fresh letter not in A). Let us define the language L_{b+} in the following way:

$$L_{b+} = \{(+)^b w_1\$(+)^{b-1}\$\ldots\$+w_b\$w_{b+1}\$\ldots\$w_{2b}\$ \mid \forall i, w_i \in A^* \wedge w_i = w_{2b+1-i}\}$$

(Again, with + another fresh letter not in A.) The proof from [92] can easily be adapted to prove that the language L_{b+} is recognizable by a 1*DFA* with k heads if and only if $b < \binom{k}{2}$: it suffices to define type[2] of a configuration according to the number of + that have been added to the words.

On the other hand we can recognize L_{b+} with a JMA with only 2 heads. This automaton works in two steps. The first one serves to verify that the word is of the shape $(+)^b w_1\$(+)^{b-1}\$\ldots\$+w_b\$w_{b+1}\$\ldots\$w_{2b}\$$ with $w_i \in A^*$ for all i. The second step serves to verify that $w_i = w_{2b+1-i}$ for all i. For the first step, one of the heads makes sure the word consists of b sequences of + of length $b, b-1, \ldots, 1$, each followed by a word of A^* and a \$ separator and then b words of A^* followed by a \$ separator. Since b is a constant of the language this is feasible even if it requires many states. Then the head jumps to the beginning of the word (using the second head). For the second step, one head linearly reads the word. During this process the second head does the following. Each time the first head reads a \$ separator, the second head jumps to the same position if it is also on a \$ separator and rejects otherwise. Then, for each + read by the first head, the second head goes through the word until it has skipped one \$ separator if it is the first + of a sequence and two separators otherwise. Thereby when the first head begins to read the word w_i, it has just read $\$(+)^{b+1-i}$ (or $(+)^b$ for w_1) and thus the second head has skipped $2(b+1-i)-1$ separators and is reading the word w_{2b+1-i}. When the first head begins to read one of the w_i, the second head begins to read at the same place as the first head

[2] see [92].

until the first head reaches a separator. If at some point the two letters being read differ then the automaton rejects the word. When a separator is reached (simultaneously by the two heads), the second head jumps back to the first one and the process continues. If at some point, after reading a separator, the first head does not read a + then the process stops and the word is accepted. □

3.4 Equivalence between JMAs and cyclic proofs

We now turn to proving the following equivalence.

Theorem 3.2. *The languages recognized by JMAs are those recognized by regular proofs.*

Together with Theorem 3.1 we deduce that regular proofs recognize exactly the DLogSpace languages. We prove the theorem in the next two subsections, by providing effective translations between the two models. Notice that by Remark 3.2, the theorem implies that for regular proofs π, emptiness and other basic properties of $[\pi]$ are undecidable.

3.4.1 From JMAs to cyclic proofs

Let $\mathcal{M} = \langle S, k, s_0, s_{acc}, s_{rej}, \delta \rangle$ be a jumping multihead automaton. We want to build a regular proof $\pi_{\mathcal{M}}$ of $A^* \vdash \mathbb{B}$ such that $[\pi_{\mathcal{M}}] = L(\mathcal{M})$. A difficulty is that heads in the automaton may stay in place, thus reading the same letter during several steps. In contrast the letters are read only once by cyclic proofs, so that we have to remember this information. We do so by labeling the sequents of the produced proof π_M with extra information describing the current state of the automaton. If $k' \in \mathbb{N}$, let $\mathcal{F}_{k'}$ be the set of injective functions $[\![1, k']\!] \to [\![1, k]\!]$. A *labeled sequent* is a sequent of the form $(A^*)^{k'} \vdash \mathbb{B}$ together with an extra label in $S \times \mathcal{F}_{k'} \times (A \uplus \{\square, \triangleleft\})^k$.

The intuitive meaning of a label $(s, f, \vec{y})$ is the following: s is the current state of the automaton, f maps each formula A^* of the sequent to a head of the automaton, and $\vec{y}$ stores the letter that is currently processed by each head. Symbol $\square$ is used if this letter is unknown, and the head is scheduled to process this letter and move to the right. The values intuitively provided to each A^* formula of the sequent are the suffixes to the right of the corresponding heads of the automaton. On the examples, labels will be written in gray below the sequents.

It will always be the case that if the label of $(A^*)^{k'}$ is $(s, f, \vec{y})$, then $Im(f) \subseteq \{i \mid y_i \neq \triangleleft\}$, i.e., all heads reading symbols from $A \uplus \{\square\}$ correspond to a formula A^* in the sequent. We say that a sequent is *fully labeled* if its label does not contain $\square$.

The construction of $\pi_{\mathcal{M}}$ will proceed by building gadgets in the form of proof trees, each one (apart from the initial gadget) connecting a labeled sequent in the conclusion to a finite set of labeled sequents in the hypotheses. If some labeled sequents in the hypotheses have already been encountered, we simply put back pointers to their previous occurrence. Since the number of labeled sequents is finite, this process eventually terminates and yields a description of $\pi_{\mathcal{M}}$.

When describing those gadgets we abbreviate sequences of inference steps or standalone proofs using double bars labeled with the involved rule names.

Initial gadget. The role of the initial gadget is to reach the first labeled sequent from the conclusion $A^* \vdash \mathbb{B}$. It simply creates k identical copies of A^*. This expresses the fact that the initial configuration is $\langle w, s_0, (0, 0, ...0)\rangle$. We note id_k the identity function on $[\![1, k]\!]$. The initial labeled sequent is $(A^*)^k \vdash \mathbb{B}$ together with label $(s_0, id_k, (\square, \ldots, \square))$.

The initial gadget is as follows:
$$c, \ldots, c\ \frac{\underset{s_0,id_k,(\square,\ldots,\square)}{(A^*)^k \vdash \mathbb{B}}}{A^* \vdash \mathbb{B}}$$

Reading gadget. Every time the label $(s, f, \vec{y})$ of the current address is not fully labeled, we use the gadget $read_i$, where $i = \min\{j \mid \vec{y}(j) = \square\}$ to process the first unknown letter.

We note $i' = f^{-1}(i)$ the position of the A^* formula corresponding to head i and define the gadget $read_i$ as follows:

$$\frac{\underset{s,f',(y_1,\ldots,y_{i-1},\lhd,\ldots,y_k)}{(A^*)^{k'-1} \vdash \mathbb{B}} \qquad A\ \frac{\left(\underset{s,f,(y_1,\ldots,y_{i-1},a,\ldots,y_k)}{(A^*)^{k'} \vdash \mathbb{B}}\right)_{a\in A}}{(A^*)^{i'-1}, A, A^*, (A^*)^{k'-i'} \vdash \mathbb{B}}}{\underset{s,f,(y_1,\ldots,y_{i-1},\square,\ldots,y_k)}{(A^*)^{k'} \vdash \mathbb{B}}}\ * \qquad \text{where } f'(x) = \begin{cases} f(x) & \text{if } 1 \le x < i' \\ f(x+1) & \text{if } i' \le x \le k'-1 \end{cases}$$

Transition gadget. Thanks to the $read_i$ gadgets, we can now assume we reach a fully labeled sequent, with label of the form $(s, f, (y_1, \ldots, y_k))$. If $s \notin \{s_{acc}, s_{rej}\}$, we use a *transition gadget*, whose general shape is as on the right below, with $(s', \alpha) = \delta(s, (y_1, \ldots, y_k))$:

This gadget is designed such that for all $i \in [\![1, k]\!]$:

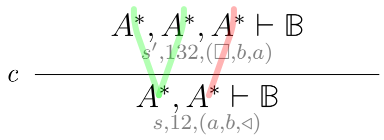
- if $\alpha(i) = ◌$ then $z_i = y_i$
- if $\alpha(i) = ⧐$ then $z_i = \square$,
- if $\alpha(i) = J_j$ then $z_i = y_j$.

$$\delta\ \frac{\underset{s',f',(z_1,\ldots,z_k)}{(A^*)^{k''} \vdash \mathbb{B}}}{\underset{s,f,(y_1,\ldots,y_k)}{(A^*)^{k'} \vdash \mathbb{B}}}$$

In the last case, a contraction is used to duplicate the A^* formula corresponding to head j, and the function f' maps this new formula to head i. The occurrence of A^* corresponding to y_i is weakened (possibly after having been duplicated if another head jumped to i).

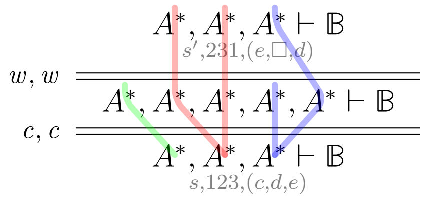
We describe this gadget on two examples below. An element $f : [\![1, k']\!] \to [\![1, k]\!]$ is simply represented by $f(1)f(2)\ldots f(k')$.

$\delta(s, (a, b, \lhd)) = (s', (⧐, ◌, J_1))$



$A^*, A^*, A^* \vdash \mathbb{B}$
$s', 132, (\square, b, a)$
c
$A^*, A^* \vdash \mathbb{B}$
$s, 12, (a, b, \lhd)$

$\delta(s, (c, d, e)) = (s', (J_3, ⧐, J_2))$



$A^*, A^*, A^* \vdash \mathbb{B}$
$s', 231, (e, \square, d)$
w, w
$A^*, A^*, A^*, A^*, A^* \vdash \mathbb{B}$
c, c
$A^*, A^*, A^* \vdash \mathbb{B}$
$s, 123, (c, d, e)$

Notice that it is also possible to avoid unnecessary contractions, in order to bound the number of A^* formulas in a sequent by k. The symbol $\square$ means that the formula A^* is scheduled for a $*$ rule, and will be immediately processed thanks to the gadget $read_i$ as described above.

Final gadget. It remains to describe what happens if the current sequent is fully labeled with $s \in \{s_{acc}, s_{rej}\}$. In this case, we simply conclude with a $(\mathbf{tt})$ axiom if $s = s_{acc}$ or with a $(\mathbf{ff})$ axiom if $s = s_{rej}$.

This achieves the description of the preproof $\pi_{\mathcal{M}}$. The following lemma expresses its correctness.

Lemma 12. *If $\mathcal{M}$ is a progressing JMA, the preproof $\pi_{\mathcal{M}}$ is valid, and $[\pi_{\mathcal{M}}] = L(\mathcal{M})$.*

Proof. Assume $\pi_{\mathcal{M}}$ is not valid, i.e., there exists an infinite branch ρ without validating thread. By considering a sufficiently long prefix of ρ, we can find addresses v, w in ρ such that

- v and w correspond to the same subtree, and contain a fully labeled sequent.
- there is no thread visiting a position from v to w.

The path from v to w witnesses a loop in the transition graph of $\mathcal{M}$. Moreover, since positions in $\pi_{\mathcal{M}}$ are only encountered when a head advances, and a thread is cut only when the corresponding head jumps, this loop does not verify the progressing condition, i.e., there is no head that advances without jumping, otherwise it would yield a thread from v to w visiting a position. We obtain a contradiction, thereby proving that $\pi_{\mathcal{M}}$ is valid.

To show that $[\pi_{\mathcal{M}}] = L(\mathcal{M})$, we can analyze the computation of the proof $\pi_{\mathcal{M}}$ on a word $u \in A^*$. We proceed by induction, and show the computation of $\pi_{\mathcal{M}}$ closely follows the computation of $\mathcal{M}$ in the following way:

- each sequence of steps of the evaluation of $[\pi_{\mathcal{M}}](u)$ between two fully labeled sequents corresponds to a transition of $\mathcal{M}$
- when reaching address v with sequent $(A^*)^{k'}$ fully labeled by $(s, f, \vec{y})$, the computation of $[\pi_{\mathcal{M}}](u)$ evaluates $[v](u_1, \dots, u_{k'})$ where s is the current state of $\mathcal{M}$, f maps each u_i to a head $f(i)$, $\vec{y}$ describes the letters currently read by each head, and $u_1, \dots, u_{k'}$ is the list of suffixes remaining to be read by heads that did not reach the end of the input.

The initial gadget and $\mathit{read}_1, \dots, \mathit{read}_k$ gadgets ensure that the first fully labeled sequent describes the initial configuration according to the above correspondence. The transition gadget and read_i gadgets preserve the above invariant, and accurately simulate a transition of $\mathcal{M}$. The final gadget allows one to stop the computation of $[\pi_{\mathcal{M}}](u)$ whenever $\mathcal{M}$ stops on input u, and returns the same value. □

3.4.2 From cyclic proofs to JMAs

Let π be a regular proof with conclusion $A^* \vdash \mathbb{B}$. Let k be the maximal number of star formulas in the sequents of π. We build a JMA $\mathcal{M}$ with k heads such that $L(\mathcal{M}) = [\pi]$.

The idea of the construction is to store all necessary information on the current state of the computation in π into the state space of $\mathcal{M}$, besides the content of star formulas. This includes the current address in π, and the actual letters corresponding to the alphabet formulas, together with some information linking star formulas to heads of the automaton.

This allows $\mathcal{M}$ to mimic the computation of $[\pi]$ on an input u, in a similar way as the converse translation from Section 3.4.1. In particular, we keep the invariant that the value associated to each star formula is the suffix of u to the right of the corresponding head of $\mathcal{M}$.

State space of $\mathcal{M}$. Let m be the maximal number of alphabet formulas in the sequents of π. We use a register with m slots, each one possibly storing a letter from A. Let $R = \bigcup_{i=0}^{m} A^i$ be the set of possible register values. An element $b_1 \dots b_i$ of R describes the content of the i alphabet formulas of the current sequent. We denote the empty register by $\Diamond$. Intuitively, the register needs to store the values that have been processed by the automaton, but are still unknown in the proof π as they are represented by alphabet formulas.

Let $\mathcal{F}$ be the set $\bigcup_{i=0}^{k} [\![1, k]\!]^i$. An element $f \in \mathcal{F}$ associates to each A^* formula of a sequent the index of a head of $\mathcal{M}$. This allows us to keep track of the correspondence between heads of $\mathcal{M}$ and suffixes of the input word being processed by π.

We define the state space of $\mathcal{M}$ as $S = (\mathit{Addr}(\pi) \times R \times \mathcal{F}) \uplus \{s_{acc}, s_{rej}\}$.

Notice that $\mathit{Addr}(\pi)$ is infinite, so $\mathcal{M}$ is an infinite-state JMA. However, if π has finitely many subtrees, we will be able to quotient $\mathit{Addr}(\pi)$ by $v \sim w$ if v and w correspond to the same subtree, and obtain a finite-state JMA.

If (v, r, f) is a state of $\mathcal{M}$, we will always have $|r| = m'$ and $|f| = k'$, where m' (resp. k') is the number of alphabet (resp. star) formulas in $\pi(v)$. Moreover, for all $i \in [\![1, m']\!]$, the i^{th} alphabet formula contains the letter $r(i)$ stored in the i^{th} slot of the register r.

The initial state is $s_0 = (\epsilon, \Diamond, 1)$. It points to the root of π, with empty register, and maps the only star formula to head 1.

Transition function of $\mathcal{M}$. If $s = (v, r, f)$ is a state of $\mathcal{M}$, and $\vec{a} = (a_1, \dots, a_k)$ is the tuple of letters read by each head with $a_i \in A_{\triangleleft}$, we want to define $\delta(s, \vec{a}) = (s', \alpha) \in S \times \mathit{Act}^k$.

We write α_{id} for the action tuple (◌, …, ◌) leaving each head at the same position. We write move_i (resp. $\mathit{jump}_{i,j}$) for the element of Act^k which associates to heads $i' \neq i$ the action ◌ and to head i the action ⏵| (resp. jump to head j).

First of all, if the rule applied to v in π is an axiom $(\mathbf{tt})$ (resp. $(\mathbf{ff})$), we set $s' = s_{acc}$ (resp. s_{rej}) and $\alpha = \alpha_{id}$. This allows $\mathcal{M}$ to stop the computation and return the same value as $[\pi]$. Otherwise, we define $s' = (v', r', f')$ and α depending on the rule applied to v in π. By Proposition 3.2, we can assume that the proof π does not use the weakening rule. Let m' (resp., k') be the number of alphabet (resp. star) formulas in $\pi(v)$.

Contraction rule:
We set $v' = v0$, and do a case analysis on the principal formula:

- i^{th} alphabet formula: we set $f' = f$, $r' = r(1) \cdots r(i-1) \cdot r(i) \cdot r(i) \cdot r(i+1) \cdots r(m')$ and $\alpha = \alpha_{id}$.

- i^{th} star formula: let $j \in [\![1, k]\!]$ be the smallest integer not appearing in f, corresponding to the index of the first available head. We want to allocate it to this new copy, by making it jump to the position of the head $f(i)$. We take $r' = r$, $f' = f(1) \cdots f(i) \cdot j \cdot f(i+1) \cdots f(k')$, and $\alpha = \mathit{jump}_{j,f(i)}$.

Star rule:
Let i be the index of the principal star formula. We now want the head $j \triangleq f(i)$ pointing on this formula to move right. The letter processed by this head will be added to the register.

- if $\vec{a}(j) = \triangleleft$, the head reached the end of the input. This corresponds to the left premiss of the $*$ rule. We set $v = v0$, $f' = f(1) \cdots f(i-1)f(i+1) \cdots f(k')$, $r' = r$ and $\alpha = \alpha_{id}$.

- if $\vec{a}(j) \in A$, we set $v' = v1$, $f' = f$, $\alpha = move_i$, and $r' = r(1) \cdots r(i')\vec{a}(i)r(i'+1) \cdots r(m')$, where i' is the number of A formulas before the principal star formula.

Alphabet rule:

Let i be the index of the principal A formula, and $a = r(i)$ be the letter associated to it. We define $v' = va$, $f' = f$, $\alpha = \alpha_{id}$, and $r' = r(1) \ldots r(i-1)r(i+1) \ldots r(m')$, i.e., we erase the i^{th} slot.

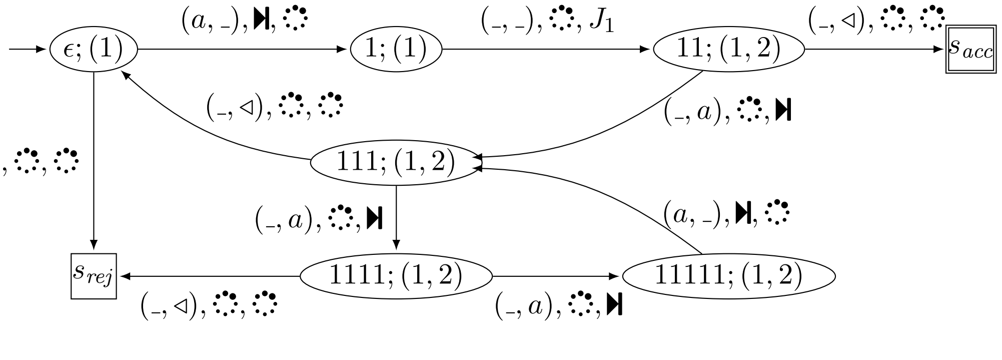
This completes the description of the JMA $\mathcal{M} = \langle S, k, s_0, s_{acc}, s_{rej}, \delta \rangle$.

Lemma 13. *The JMA $\mathcal{M}$ is progressing, and $L(\mathcal{M}) = [\pi]$.*

Proof. A loop $(v, r, f) \to (v, r, f)$ in the transition graph of $\mathcal{M}$ corresponds to a path in π from v to some vu, two addresses corresponding to the same subtree. By validity of π, the branch vu^ω contains a validating thread. So there is a thread t from $\langle v, i \rangle$ to $\langle vu, j \rangle$ for some i, j, containing a $*$ position. Let $h = f(i)$ be the index of the head pointing to the A^* formula in position $\langle v, i \rangle$. By construction of δ, this head did not jump during the loop (or the thread t would be cut), and performs at least one action ⧐ (where the thread t visits a $*$ position). We proved that the JMA $\mathcal{M}$ is progressing, since any loop verifies the progressing condition.

The fact that $L(\mathcal{M}) = [\pi]$ is proved by induction on the computation of $[\pi]$ on any input u. At any point, the construction of $\mathcal{M}$ preserved the announced invariants: registers store the contents of alphabet formulas, for each i the content of the i^{th} star formula is the suffix to the right of the head number $f(i)$. The transition function δ is built to follow the computation of $[\pi](u)$ on any u, and return the same result when a (**tt**) or (**ff**) axiom is reached. □

Example 3.3. We can obtain a progressing JMA for the language $L = \{a^{2^n} \mid n \in \mathbb{N}\}$ by translating the proof from Figure 3.4 using the above procedure. As there are at most two star formulas in the sequents of the proof, the produced JMA has two heads. As there is only one letter in the alphabet, we can just forget the register. Similarly we consider that any **ff** part (resp. **tt** part) of the proof corresponds to the state s_{rej} (resp. s_{acc}). Using _ for reading any symbol (a letter a or $\triangleleft$), we can represent the obtained automaton as follows:



Remark 3.3. Our encoding from regular proofs to JMAs would still work if we had included an exchange rule in the system, and the encoding from JMAs to regular proofs does not require the exchange rule. Therefore, such a rule would not increase the expressive power.

3.4.3 The affine case: regular languages

Looking at the encodings in the two previous sections, we can observe that:

- the encoding of an affine regular proof is a JMA with a single head: in absence of contraction, all sequents in proof ending with $A^* \vdash \mathbb{B}$ have at most one star formula;
- the encoding of a JMA with a single head does not require contraction: this rule is used only for the initial gadget and when the action of a head is to jump on another one.

As a consequence, we have a correspondence between affine regular proofs and JMAs with a single head, whence, by Lemma 9:

Theorem 3.3. *The regular languages are those recognizable by affine regular proofs.*

3.5 Conclusion and future work

We have defined a cyclic proof system where proofs denote formal languages, as well as a new automata model: jumping multihead automata. We have shown that regular proofs correspond precisely to the languages recognizable by jumping multihead automata, which turn out to be the DLogSpace languages. Moreover we have shown that the restriction to affine regular proofs corresponds to the regular languages. We see two directions for future work.

First, we restricted to sequents of the shape $E \vdash \mathbb{B}$ in order to focus on languages. As we shall see in Chapter 4, the proof system we started from (LKA [28]) however makes it possible to deal with sequents of the shape $E \vdash e$: it suffices to include right introduction rules for the alphabet (A) and star formulas (A^*). By doing so, we obtain a system where proofs of $A^* \vdash A^*$ denote *transductions*: functions from words to words. We conjecture that in the affine case, we obtain exactly the *subsequential transductions* [75], i.e., transductions definable by deterministic 1-way transducers. In the general case (with contraction), we would need a notion of jumping multihead transducers.

Second, we used a *cut-free* proof system. While adding the cut rule for the presented system (restricted to sequents $E \vdash \mathbb{B}$) seems peculiar since the input and output are not of the same shape, it becomes reasonable when moving to general sequents for transductions. We have observed that we can go beyond MSO-definable transductions when doing so, even in the affine case. We would like to investigate and hopefully characterize the corresponding class of transductions.

Chapter 4

Cyclic proofs and functional programs

Abstract

We extend the cyclic proof system from the previous chapter to deal with regular expression types and arrow types, so that proofs in C can be seen as strongly typed functional programs. We show that they denote computable total functions and we analyze the relative strength of C and Gödel's system T. In the general case, we prove that the two systems capture the same functions on natural numbers. In the affine case, i.e., when contraction is removed, we prove that they capture precisely the primitive recursive functions—providing an alternative and more general proof of a result by Dal Lago, about an affine version of system T.

Without contraction, we manage to give a direct and uniform encoding of C into T, by analyzing cycles and translating them into explicit recursions schemes. Whether such a direct and uniform translation from C to T can be given in the presence of contraction remains open.

We obtain the two upper bounds on the expressivity of C using a different technique: we formalize weak normalization of a small step reduction semantics in subsystems of second-order arithmetic: ACA_0 and RCA_0.

4.1 Introduction

As explained in the previous chapter, a natural question in non-wellfounded proof theory is whether specific cyclic and inductive proof systems have the same logical strength. Indeed, while inductive proofs can usually be translated easily into cyclic ones (see, e.g., [17]), the converse problem is often harder [77, 12], or impossible [11, 25].

Here we propose a cyclic proof system which we study from the other side of the Curry-Howard correspondence: the proof system is seen as a type system, and proofs (i.e., typing derivations) as programs. Intuitively, those programs are unstructured yet strongly typed functional programs; we call them cyclic programs in the sequel. Despite the strongly typed discipline, the corresponding language is low-level, and closer in spirit to assembly or goto programs than to higher-level languages with while loops, like C or Pascal.

As in the previous chapter, we import from cyclic proof theory a validity criterion which makes it possible to ensure termination of cyclic programs. This criterion is non-local, but syntactic and decidable via Büchi automata algorithms. Although we work here with structured values and total operations, this criterion can also be seen as an instance of the celebrated size change termination principle [64].

We characterize the computational strength of cyclic programs in terms of more traditional devices: primitive recursive functions and Gödel's system T (i.e., simply typed lambda-calculus with natural numbers and recursion).

We take as types the formulas of intuitionistic multiplicative additive linear logic (IMALL) with a least fixpoint operator for lists. We can thus manipulate datatypes consisting of natural numbers and functions, but also pairs, lists, or sums, without the need for encodings. Our cyclic proof system, which we call *system C*, is basically the sequent system LAL for action lattices from [28], to which we add the three usual structural rules: exchange, weakening and contraction. Proceeding this way makes it possible to consider the *affine* fragment $\mathrm{C_{aff}}$ of C, where the contraction rule is forbidden.

Accordingly, we use a variant of Gödel's system T with the same formulas/types as C in order to ease comparisons. We define this type system in a slightly non-standard way: like for C, we use explicit structural rules in order to be able to talk about the affine fragment $\mathrm{T_{aff}}$ of T.

Contraction indeed plays an important role in those systems: we show that

1. $\mathrm{C_{aff}}$ and $\mathrm{T_{aff}}$ are equally expressive (at all types), and their functions on natural numbers are the primitive recursive functions;

2. C and T capture the same functions on natural numbers, those that are provably total in Peano arithmetic.

We obtain those results via the translations summarized below, where dotted arrows denote encodings restricted to functions on natural numbers.

$$\begin{tikzcd}
\mathrm{T} \arrow[r, bend left, "\text{Thm. 4.2}"] & \mathrm{C} \arrow[l, bend left, dotted, "\text{Cor. 1, via } \mathsf{ACA}_0"]
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathrm{T_{aff}} \arrow[r, bend left, "\text{Thm. 4.2}"] & \mathrm{C_{aff}} \arrow[l, bend left, "\text{Thm. 4.3}"] \arrow[r, bend right, dotted, "\text{Cor. 3, via } \mathsf{RCA}_0"'] & \text{prim. rec.} \arrow[l, bend right, dotted, "\text{Thm. 4.1}"']
\end{tikzcd}$$

As expected, we can translate terms of T into cyclic programs of C (Theorem 4.2); this is a compilation process, the translation is uniform and maps affine terms to affine programs. We also observe that we do not need contraction to encode primitive recursive functions into C (Theorem 4.1).

Conversely, encoding cyclic programs into T is much harder: this is a decompilation process, we have to delineate possibly complex cycle structures in order to use the very strict recursion capabilities of T. Seen from the logic side, this is in fact an instance of the difficult problem of translating cyclic proofs into inductive ones [77, 12, 11, 25]. We manage to provide a direct and uniform encoding in the affine case (Theorem 4.3), which we do not know how to extend in the presence of contraction.

In order to get our upper bounds on the expressivity of C and affine C for functions on natural numbers (Corollary 1 and Corollary 3), we define a small steps reduction semantics for C. This semantics matches the higher-level and higher-order semantics we use elsewhere in the chapter, and we prove that it is weakly normalizing. We obtain Corollary 1 by observing that this weak normalization proof can be performed inside the subsystem ACA_0 of second order arithmetic [78], whose provably recursive functions are precisely those from system T.

For the affine case (Corollary 3), Dal Lago's system $\mathcal{H}(\emptyset)$ [23] is a variant of Gödel's system T which characterizes primitive recursive functions and which is really close to our affine version of T. Unfortunately, we need additive pairs in order to translate affine C into affine T. Those are not available in $\mathcal{H}(\emptyset)$, and it is not clear how to extend Dal Lago's proof to deal with such operations: his proof is complex and relies on a semantics based on geometry of interaction, whose extension to additives is notoriously difficult [42, 10, 4, 54]. We instead prove Corollary 3 by using another proof of weak normalization for C, which works only on the image of our translation from affine T to C. This argument can be formalized into another subsystem of second order arithmetic, RCA_0, which is known to define only the primitive recursive functions [7]. This yields an alternative and more general proof of Dal Lago's result (Corollary 2).

On system C as a programming language. As explained above, the cyclic proof system C can be seen as a type system for a low-level programming language manipulating structured values. Even though this language is pure—no side effects—and strongly typed, we insist that it is low-level because loops are expressed using goto instructions rather than high-level constructs such as while loops or iterators. Accordingly, the (cyclic) type system ensures termination via a global yet decidable criterion (an ω-regular condition). This is in sharp contrast with other terminating programming languages such as system T (or Agda, Coq), where termination is ensured using a local notion of guardedness: there, although recursive definitions can be nested in complicated ways, termination is ensured by imposing that each recursive call must be guarded.

Related work System T was originally introduced by Gödel in [45] as an equational theory built up over a fragment of the term calculus that we identify as T here. That work introduced the 'Dialectica' *functional interpretation*, that allows T to interpret Peano Arithmetic.[1] Our work can be seen as a counterpart in T to recent work on cyclic arithmetic [77, 25].

Other infinitary versions of system T are well-known, in particular [81]. These also induce a 'term model' of T where recursors are replaced by infinitely long yet well-founded terms. This difference resembles the difference between logical systems with ω-branching versus their non-wellfounded counterparts, e.g., as in arithmetic [77, 25].

Although there are works on using cyclic proof systems to perform proof search and reason automatically about inductive types or program equivalence [66, 67], those ideas do not really apply in system C because it is a programming language with a fancy (i.e., cyclic) yet simple type system. Proof search would correspond to type inhabitation—a

[1] Gödel only treated Heyting Arithmetic, the intuitionistic counterpart of Peano Arithmetic. An interpretation of the latter is duly obtained by composition with an appropriate double-negation translation.

trivial problem for closed types in our setting: the types we use are not expressive enough to serve as specifications.

4.2 System C and its semantics

4.2.1 Regular expressions as types

We let the letters a, b range over the elements of a fixed set A of *type variables*. We define *types* with the following syntax.

$$e, f := a \mid e \cdot f \mid e + f \mid e^* \mid 1 \mid e \to f \mid e \cap f$$

The five first entries correspond to regular expressions; the arrow adds function spaces. As a first approximation, the intersection operator ($\cap$) can be identified with the product operator ($\cdot$): both operators are interpreted as Cartesian product in the high-level semantics we define below. Our set of rules will turn the former into an *additive* conjunction and the latter into a *multiplicative* conjunction. We use the above notations for the connectives rather than those from linear logic because:

- we want to emphasize that expressions e, f are types rather than formulas (although we shall also call them *formulas* when this is more natural);
- in the presence of contraction and weakening, the linear behavior the various connectives disappears.

We assume a family $(D_a)_{a \in A}$ of sets indexed by A, representing elements of atomic types. To every type e, we associate a set $[e]$ of *values*, by induction on e:

$$\begin{aligned} [e \cdot f] \triangleq [e \cap f] &\triangleq [e] \times [f] & [1] &\triangleq 1 \\ [e + f] &\triangleq [e] + [f] & [e^*] &\triangleq [e]^* \\ [e \to f] &\triangleq [f]^{[e]} & [a] &\triangleq D_a \end{aligned}$$

We have that $[1^*]$ is in bijection with $\mathbb{N}$, so that we can use 1^* as a type for natural numbers.

We let E, F range over finite sequences of types. Given such a sequence $E = e_0, \ldots, e_{n-1}$, we write $[E]$ for $[e_0] \times \cdots \times [e_{n-1}]$.

We will define a sequent proof system where sequents have the shape $E \vdash e$, and proofs of such sequents, cyclic programs, will denote functions from $[E]$ to $[e]$.

4.2.2 Non-wellfounded proofs

The rules of C are given in Figure 4.1; in addition to the *structural rules* (exchange, weakening, contraction, axiom, and cut), we have introduction rules on the left and on the right for each type connective (*logical rules*). Those rules are standard, they are those of intuitionistic multiplicative additive linear logic, when interpreting $\cdot$ as multiplicative conjunction ($\otimes$), $+$ as additive disjunction ($\oplus$), $\cap$ as additive conjunction ($\&$), and $\to$ as linear arrow ($\multimap$). The rules for type e^* correspond to unfolding rules, looking at e^* as the least fixpoint expression $\mu x.1 + e \cdot x$ (e.g., from the μ-calculus).

$$
{\scriptstyle id}\,\frac{}{e \vdash e}
\qquad
{\scriptstyle cut}\,\frac{E \vdash e \quad e, F \vdash g}{E, F \vdash g}
\qquad
{\scriptstyle x}\,\frac{E, f, e, F \vdash g}{E, e, f, F \vdash g}
\qquad
{\scriptstyle w}\,\frac{E \vdash g}{e, E \vdash g}
\qquad
{\scriptstyle c}\,\frac{e, e, E \vdash g}{e, E \vdash g}
$$

$$
{\scriptstyle \cdot\text{-}l}\,\frac{e, f, E \vdash g}{e \cdot f, E \vdash g}
\qquad
{\scriptstyle \cdot\text{-}r}\,\frac{E \vdash e \quad F \vdash f}{E, F \vdash e \cdot f}
$$

$$
{\scriptstyle +\text{-}l}\,\frac{e, E \vdash g \quad f, E \vdash g}{e + f, E \vdash g}
\qquad
{\scriptstyle +\text{-}r_i}\,\frac{E \vdash e_i}{E \vdash e_0 + e_1}\; i \in \{0,1\}
$$

$$
{\scriptstyle *\text{-}l}\,\frac{E \vdash g \quad e, e^*, E \vdash g}{e^*, E \vdash g}
\qquad
{\scriptstyle *\text{-}r_\epsilon}\,\frac{}{\vdash e^*}
\qquad
{\scriptstyle *\text{-}r_{::}}\,\frac{E \vdash e \quad F \vdash e^*}{E, F \vdash e^*}
$$

$$
{\scriptstyle 1\text{-}l}\,\frac{E \vdash g}{1, E \vdash g}
\qquad
{\scriptstyle 1\text{-}r}\,\frac{}{\vdash 1}
$$

$$
{\scriptstyle \to\text{-}l}\,\frac{E \vdash e \quad f, F \vdash g}{e \to f, E, F \vdash g}
\qquad
{\scriptstyle \to\text{-}r}\,\frac{e, E \vdash f}{E \vdash e \to f}
$$

$$
{\scriptstyle \cap\text{-}l_i}\,\frac{e_i, E \vdash g}{e_0 \cap e_1, E \vdash g}\; i \in \{0,1\}
\qquad
{\scriptstyle \cap\text{-}r}\,\frac{E \vdash e \quad E \vdash f}{E \vdash e \cap f}
$$

Figure 4.1: The rules of C.

Those rules are also essentially the same as those used for action lattices in [28]. The only differences are that they can be slightly simplified here since we have the exchange rule, and that there is only one arrow, being in a commutative setting—again, due to the exchange rule.

Given those rules, we define preproofs, ancestry, threads and validity exactly as in Chapter 3. We nonetheless repeat those definitions in order to ease the reading.

Recall that a (binary, possibly infinite) tree is a non-empty and prefix-closed subset of the set $\{0,1\}^*$ of addresses. In this chapter, we write $\sqsubseteq$ (resp. $\sqsubset$) for the prefix relation (resp. strict prefix) on addresses.

Definition 4.1. A *preproof* is a labeling π of a tree by sequents such that, for every node v with children $v_1, \dots v_n$ ($n = 0, 1, 2$), the expression $\dfrac{\pi(v_1) \ \cdots \ \pi(v_n)}{\pi(v)}$ is an instance of a rule from Figure 4.1. Given an address v in a preproof π, we write π_v for the sub-preproof rooted at v, defined by $\pi_v(w) = \pi(vw)$. A preproof is *regular* if it has finitely many distinct subtrees. A preproof is *cut-free* (resp. *affine*, *linear*) if it does not use the *cut* rule (resp. *c* rule, *c* and *w* rules).

The formula e in an instance of the *cut* rule is called the *cut formula*; the formulas appearing in lists E, F of any rule instance are called *auxiliary formulas*, and the non auxiliary formula appearing in the antecedent of the conclusion of the logical rules is called the *principal formula*.

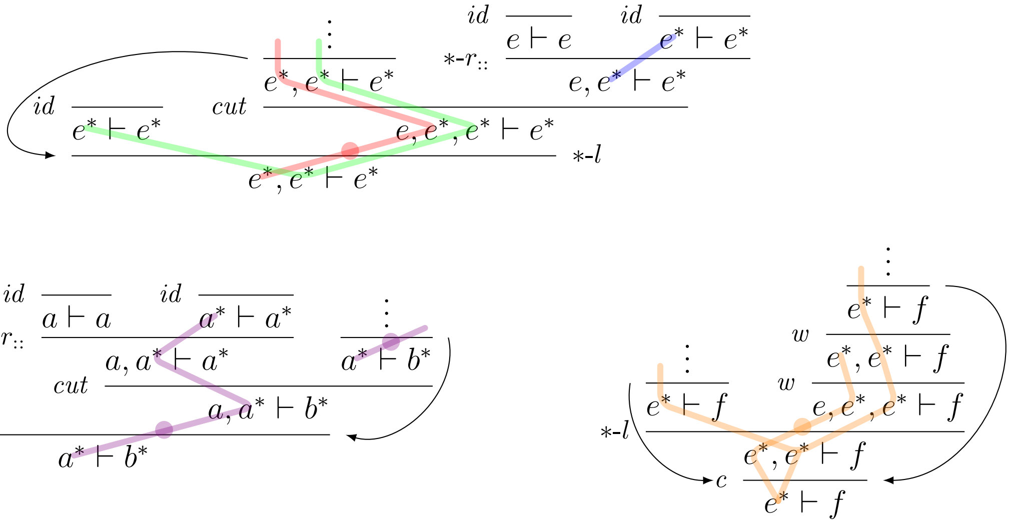


Figure 4.2: Three regular preproofs.

Three examples of regular preproofs are depicted in Figure 4.2. The bottom-right one has exactly the same structure as the one on the right in Figure 3.2. We define below a validity criterion, which is satisfied only by the topmost preproof. Before doing so, we need to define threads. As in Chapter 3, those are the branches of the shaded trees depicted on the preproofs.

All rules but the cut rule have the subformula property: every formula appearing in the premisses is a subformula of one of the formulas appearing in the conclusion, usually called its immediate descendant in the literature. We use a slightly stricter notion of ancestry here.

Definition 4.2. A *position* in a preproof π is a pair $\langle v, i\rangle$ consisting of an address v and an index i such that $\pi(v) = E \vdash f$ and E_i is a star formula. A $*$-l *address* is an address pointing at the conclusion of a $*$-l step. A position $\langle v, i\rangle$ is a *principal* when v is a $*$-l address and $i = 0$.

A position $\langle v, i\rangle$ is a *parent* of a position $\langle w, j\rangle$ if $|v| = |w| + 1$ and, looking at the rule applied at address w the two positions point at the same place in the lists E, F of auxiliary formulas, or at the formula e (resp. e or f) when this is the contraction rule (resp. exchange rule), or at the principal formula e^* when this is the $*$-l rule and $v = w1$. We write $\langle v, i\rangle \lhd \langle w, j\rangle$ in the former cases, and $\langle v, i\rangle \mathrel{\triangleleft\!\!\!\cdot} \langle w, j\rangle$ in the latter case (in which case $i = 1$ and $j = 0$). We say that $\langle v, i\rangle$ is an *ancestor* of $\langle w, j\rangle$ when those positions are related by the transitive closure of the parentship relation.

The graph of the parentship relation is depicted in Figure 4.2 using shaded thick lines and an additional bullet to indicate when we pass principal steps ($\mathrel{\triangleleft\!\!\!\cdot}$). Note that in rule $*$-l, the occurrence of e in the second premiss is not a parent of e^* in the conclusion. Due to this restriction, positions linked by the ancestry relation all point to the same star formula.

Remark 4.1. Suppose that $u \sqsubseteq v$ are addresses in a preproof π. Then a position at v is the ancestor of at most one position at u, and it is only in the presence of contraction that a position at u may have two or more ancestors at v.

Definition 4.3. A *thread* is a branch of the ancestry graph, i.e., a set of positions forming a linear order with respect to the ancestry relation; it is *principal* when it visits a principal position, *spectator* if it is never principal, *valid* if it is principal infinitely many often.

In the topmost preproof of Figure 4.2, the infinite red thread

$$\langle \epsilon, 0\rangle \mathrel{\dot{\rhd}} \langle 1, 1\rangle \rhd \langle 10, 0\rangle \mathrel{\dot{\rhd}} \langle 101, 1\rangle \rhd \langle 1010, 0\rangle \dots$$

is valid while the infinite green thread

$$\langle \epsilon, 1\rangle \rhd \langle 1, 2\rangle \rhd \langle 10, 1\rangle \rhd \langle 101, 2\rangle \rhd \langle 1010, 1\rangle \dots$$

is spectator. In the bottom left preproof, all threads are finite: the instances of the *cut* rule disconnect the copies of the thread $\langle \epsilon, 0\rangle \mathrel{\dot{\rhd}} \langle 1, 1\rangle$ occurring in the only infinite branch of the preproof. In the remaining preproof, all infinite threads are spectator: principal steps force the thread to terminate.

Definition 4.4. A preproof is *valid* if every infinite branch contains a valid thread. A *proof* is a valid preproof. We write $\pi : E \vdash e$ when π is a proof whose root is labeled by $E \vdash e$.

In Figure 4.2, only the first preproof is valid, thanks to the infinite red thread. The second preproof is invalid: every thread is finite. The third preproof is invalid: infinite threads along the (infinitely many) infinite branches are all spectator.

Like in Chapter 3, this validity criterion is decidable for regular preproofs: it can be formulated as a Büchi condition, and checked via standard automata algorithms. It is essentially the same as in LKA [28], which in turn is an instance of the one for μMALL [31]: we just had to extend the notion of ancestry to cover the weakening and contraction rules. This induces an important subtlety:

Remark 4.2. In a fixed branch of an affine preproof, every maximal thread is determined by its first element (a position). This is not true with contraction since we can choose which parent position to follow at each contraction step.

That a sequent $E \vdash e$ is provable in system C is not something interesting per se: most sequents are provable, essentially because every closed type is inhabited (see Lemma 15 below). Instead of provability, we do focus on proofs themselves, and on their computational content.

4.2.3 Computational interpretation of system C

We now show how to interpret a proof $\pi : E \vdash e$ as a function $[\pi] : [E] \to [e]$. Like in Chapter 3.3, we cannot reason directly by induction on proofs and we use instead the following relation which we prove to be well-founded.

Definition 4.5. A *computation* in a fixed proof π is a pair $\langle v, s\rangle$ consisting of an address v of π with $\pi(v) = E \vdash e$, and a value $s \in [E]$. Given two computations, we write $\langle v, s\rangle \prec \langle w, t\rangle$ when $|v| = |w| + 1$ and

1. for all i, j s.t. $\langle v, i\rangle \lhd \langle w, j\rangle$, we have $s_i = t_j$, and
2. for all i, j s.t. $\langle v, i\rangle \mathrel{\lhd\!\!\!\cdot\,} \langle w, j\rangle$, we have $|s_i| < |t_j|$.

(Recall that positions such as $\langle v, i\rangle$ and $\langle w, j\rangle$ in the above definition always refer to star formulas.) The two conditions state that the values assigned to star formulas should remain the same along auxiliary steps and decrease in length along principal steps.

Lemma 14. *The relation $\prec$ on computations is well-founded.*

Proof. Exactly like for Lemma 6. □

Definition 4.6. The *return value* $[v](s)$ of a computation $\langle v, s\rangle$ with $\pi(v) = E \vdash e$ is a value in $[e]$ defined by well-founded induction on $\prec$ and case analysis on the rule used at address v[2].

$$\begin{aligned} id &: [v](s) \triangleq s \\ cut &: [v](s, t) \triangleq [v1]([v0](s), t) \end{aligned}$$

$$\begin{aligned} x &: [v](s, x, y, t) \triangleq [v0](s, y, x, t) \\ w &: [v](x, s) \triangleq [v0](s) \\ c &: [v](x, s) \triangleq [v0](x, x, s) \end{aligned}$$

$\cdot$-l : $[v](\langle x, y\rangle, s) \triangleq [v0](x, y, s)$

$\cdot$-r : $[v](s, t) \triangleq \langle [v0](s), [v1](t)\rangle$

$\rightarrow$-l : $[v](h, s, t) \triangleq [v1](h([v0](s)), t)$

$\rightarrow$-r : $[v](h) \triangleq (x \mapsto [v0](x, h))$

$*$-l : $[v](l, s)$ is defined by case analysis on l:

- $[v](\epsilon, s) \triangleq [v0](s)$
- $[v](x :: q, s) \triangleq [v1](x, q, s)$

$*$-r_ϵ : $[v]() \triangleq \epsilon$

$*$-$r_{::}$: $[v](s, t) \triangleq [v0](s) :: [v1](t)$

$+$-l : $[v](x, s)$ is defined by case analysis on x:

- if $x \in [e_0]$, $[v](x, s) \triangleq [v0](x, s)$
- if $x \in [e_1]$, $[v](x, s) \triangleq [v1](x, s)$

$+$-r_i : $[v](s) \triangleq [v0](s)$

1-l : $[v](\langle\rangle, s) \triangleq [v0](s)$

1-r : $[v]() \triangleq \langle\rangle$

$\cap$-l_i : $[v](\langle x_0, x_1\rangle, s) \triangleq [v0](x_i, s)$

$\cap$-r : $[v](s) \triangleq \langle [v0](s), [v1](s)\rangle$

(In the *cut*, x, $\rightarrow$-l, $\cdot$-r and $*$-$r_{::}$ cases, the size of the tuples s and t is chosen consistently with the corresponding rule instances.)

In each case, the recursive calls are made on strictly smaller computations: they occur on direct subproofs, the values associated to auxiliary formulas are left unchanged, and in the second subcase of the $*$-l case, the length of the list associated to the principal formula decreases by one.

Note that in the *cut* and $\rightarrow$-l cases, the values $[v0](s)$ and $h([v0](s))$ might be arbitrarily large. This is not problematic: the corresponding positions have no children, so that those

[2] Here and elsewhere in the chapter, we use commas and we omit brackets to display tuples of values such as s in a return value $[v](s)$. We also restrict our usage of brackets, ϵ and :: to display values which happen to be tuples or lists (i.e., elements of $[1]$, $[e \cdot f]$, $[e \cap f]$ or $[e^*]$ for some types e, f).

values are left unconstrained by the relation $\prec$. Similarly, in order to define the graph of the returned function in the $\rightarrow$-r-case, we call the inductive hypothesis an arbitrary number of times, with arbitrarily large values for x.

Definition 4.7. The semantics of a proof $\pi : E \vdash e$ is the function $[\pi] : [E] \rightarrow [e]$ defined by $[\pi](s) \triangleq [\epsilon](s)$.

The above semantics presents proofs of C as goto programs (the address v in a computation $[v](s)$ being the program counter) operating on a structured memory and using a strongly typed discipline to avoid runtime errors. Accordingly, we sometimes call proofs of C *cyclic programs.*

Remark 4.3. We could have given a syntax for untyped cyclic programs (as sequences of gotos and basic instructions operating on a finite set of registers), and then presented the proof system C as a two-layers type system for those untyped cyclic programs. The first layer (preproofs) would have ensured that the values stored in the registers are manipulated along a simply typed discipline, ensuring properties such as 'progress' and 'subject-reduction'. The second layer (the global validity criterion) would have ensured termination. The syntax of those untyped programs would be quite redundant with the definition of C itself (essentially, one instruction per rule from Figure 4.1), whence our choice to omit it.

Let us compute the semantics of the first and only proof (cyclic program) in Figure 4.2. We have

$$\begin{aligned}
[\epsilon](\epsilon, l) &= [0](l) = l \\
[\epsilon](x :: q, l) &= [1](x, q, l) = [11](x, [10](q, l)) = [110](x) :: [111]([10](q, l)) = x :: [10](q, l) \\
&= x :: [\epsilon](q, l)
\end{aligned}$$

In the last equality we used the fact that $\pi_{10} = \pi_\epsilon$, so that $[10] = [\epsilon]$. We recognize for $[\epsilon]$ the standard definition of list concatenation, which is recursive on its first argument. Trying to perform such computations on the two invalid preproofs from Figure 4.2 would give rise to non-terminating behaviors, e.g., $[\epsilon](x :: q) \rightsquigarrow [11](x :: q) = [\epsilon](x :: q)$ in the second preproof.

4.2.4 Weakening and contraction

A type is *closed* when it does not contain variables; it is *positive* when it does not contain negative connectives $(\rightarrow, \cap)$.

Lemma 15. *For every closed type e, there are linear regular proofs* $\mathrm{rem}_e : e \vdash 1$ *and* $\mathrm{inh}_e : \vdash e$.

Proof. We proceed by induction on e. The first interesting case is the weakening of a star formula e^* which is depicted on the left of Figure 4.3. The rule marked IH_e is the weakening rule derived for e by induction hypothesis and the widget on the left in Figure 4.4. The second interesting case is the weakening of an arrow formula $e \rightarrow f$ depicted on the right of Figure 4.3: we use the fact that e is inhabited, by induction. The third interesting case is for inhabitation of arrow types, where use the fact that e can be erased, by induction. □

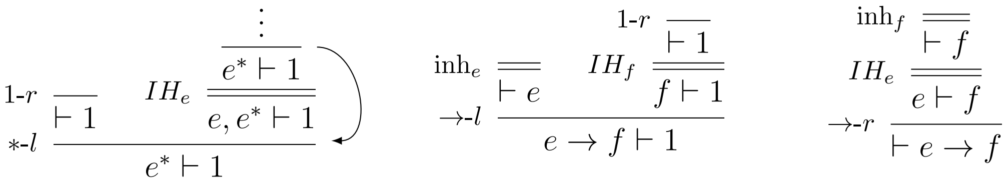


Figure 4.3: Erasing star and arrow formulas, inhabiting arrow formulas.

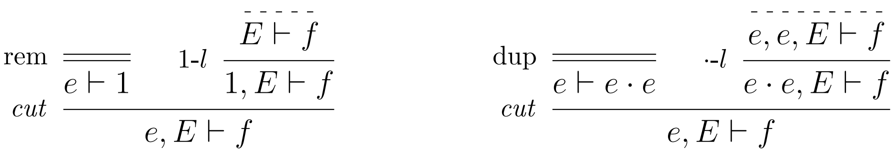


Figure 4.4: Deriving weakening and contraction.

As a consequence, weakening is admissible for closed types, by replacing it with the gadget on the left in Figure 4.4; moreover, every closed sequent is derivable, already in the linear fragment of C.

The linear system also allows for some form of duplication: while arrow types cannot be duplicated, basic types such as natural numbers (1^*) or lists of natural numbers (1^{**}) can.

Lemma 16. *For every positive closed type e, there is a linear regular proof* $\mathrm{dup}_e : e \vdash e \cdot e$ *such that for all $x \in [e]$, $[\mathrm{dup}_e](x) = \langle x, x \rangle$.*

Proof. We proceed by induction on e; the interesting case is the duplication of a star formula e^*, which is depicted in Figure 4.5. The subproofs labeled with 'cons' consist of an application of the $*$-$r_{::}$ rule followed by two identity axioms. The rule marked IH_e at address 110 is the contraction rule derived for e by induction hypothesis and the widget on the right in Figure 4.4. □

Like above, it follows that positive closed instances of the contraction rule are derivable in the linear system, using the gadget on the right in Figure 4.4. However, they are not admissible in general: the gadget does cut the potential threads on the contracted formula, so that it cannot be freely used in arbitrary proofs. For instance, anticipating Section 4.2.5 below, if we use it to replace the contraction on a star formula in the proof from Figure 4.7, the affine preproof we obtain is not valid: the green/blue thread is cut at each iteration. Actually, if contraction on closed types was derivable in a thread-preserving way, and thus admissible, we would obtain a counter-example to Corollary 3 below.

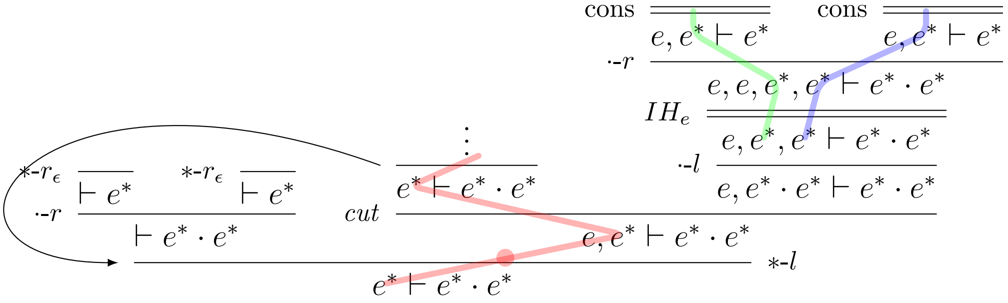

cons
$e, e^* \vdash e^*$
cons
$e, e^* \vdash e^*$
$\cdot$-r
$e, e, e^*, e^* \vdash e^* \cdot e^*$
IH_e
$e, e^*, e^* \vdash e^* \cdot e^*$
$\cdot$-l
$e, e^* \cdot e^* \vdash e^* \cdot e^*$
$*$-r_ϵ
$\vdash e^*$
$*$-r_ϵ
$\vdash e^*$
$\cdot$-r
$\vdash e^* \cdot e^*$
$e^* \vdash e^* \cdot e^*$
cut
$e, e^* \vdash e^* \cdot e^*$
$*$-l
$e^* \vdash e^* \cdot e^*$

Figure 4.5: Duplicating a star formula.

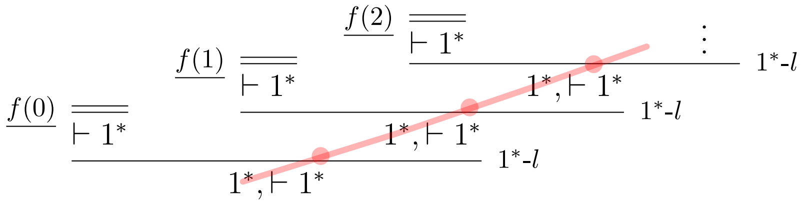
4.2.5 Functions on natural numbers

Natural numbers can be represented through the type 1^* of lists over the singleton set. The logical rules for this specific instance of the star type can be optimized as follows:

$$1^*\text{-}l\ \frac{E \vdash g \quad 1^*, E \vdash g}{1^*, E \vdash g} \qquad 1^*\text{-}r_0\ \frac{}{\vdash 1^*} \qquad 1^*\text{-}r_S\ \frac{E \vdash 1^*}{E \vdash 1^*}$$

Those rules are immediate consequences of the logical rules for 1 and star. Using these rules, we deduce that for all $n \in \mathbb{N}$, we can build a finite proof $\underline{n} : \ \vdash 1^*$ such that $[\underline{n}]() = n$.

Similarly, for every function (even an uncomputable one) $f : \mathbb{N} \to \mathbb{N}$, we can obtain a proof $\underline{f} : 1^* \vdash 1^*$ such that $[\underline{f}] = f$: repeatedly apply the 1^*-l rule to obtain a comb-shape infinite tree, and fill the remaining leaves with finite proofs for the successive values of the function. This proof, which is essentially the graph of the function f, is linear and cut-free, but not regular in general.



Our first expressivity result for regular proofs is:

Theorem 4.1. *For every primitive recursive function $f : \mathbb{N} \times \cdots \times \mathbb{N} \to \mathbb{N}$, there exists a linear and regular proof $\pi : 1^*, \ldots, 1^* \vdash 1^*$ such that $[\pi] = f$.*

Proof. By induction on the definition scheme for primitive recursive functions. The constant 0-ary function and the successor 1-ary functions give rise to simple finite proofs. The projection functions just require weakening for 1^* (Lemma 15). Function composition is implemented using the *cut* rule, as expected, but it also requires duplicating the arguments to provide them to the composed functions. For instance, to compose a 2-ary

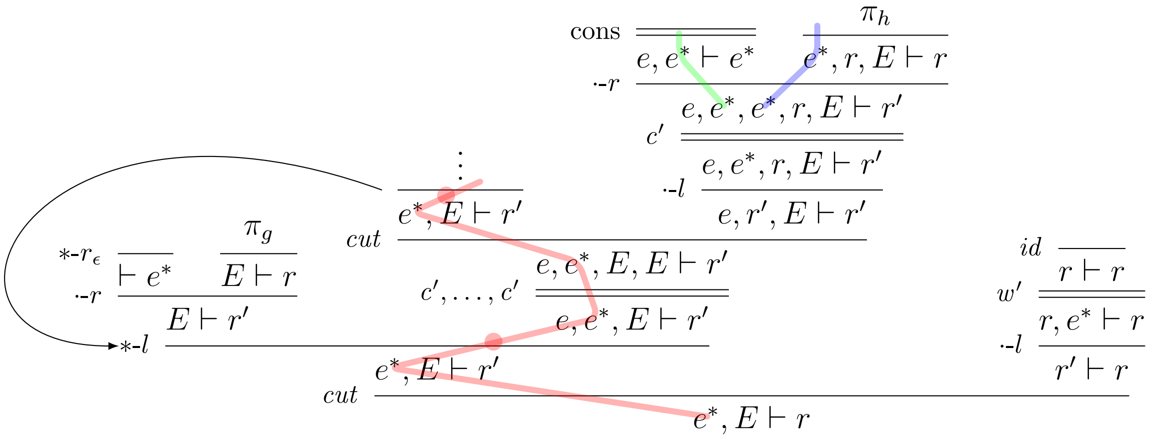
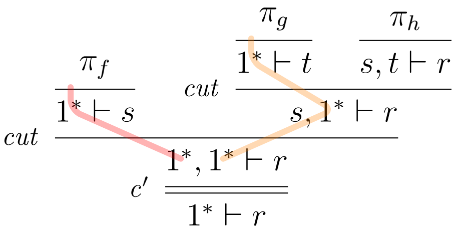


Figure 4.6: Regular linear proof for primitive recursion; $e \triangleq 1$, $r \triangleq 1^*$; $r' \triangleq e^* \cdot r$.

function h with two 1-ary functions f, g, we use the following scheme:



We used the abbreviations $r = s = t = 1^*$ to distinguish between the respective return types of h, f and g, and we marked with c' our usage of the derivable contraction rule (Lemma 16). That this step cuts the threads is not problematic here: cycles cannot visit this contraction step.

It remains to deal with primitive recursion. Suppose f is defined as follows:

$$\begin{cases} f\ 0\ \vec{y} & = g\ \vec{y} \\ f\ (Sx)\ \vec{y} & = h\ x\ (f\ x\ \vec{y})\ \vec{y} \end{cases}$$

where g and h are primitive recursive functions of respective arity n and $n+2$. By induction hypothesis there exist proofs π_g and π_h that encode g and h. In the recursive definition above, one can observe that both x and $\vec{y}$ are used twice. The latter can easily be handled using the derivable contraction rule since they are not involved in the termination argument. On the contrary, the duplication of x is problematic since the corresponding thread should validate the recursion. To circumvent this difficulty, we perform a recursion that returns a copy of the recursive argument together with the expected return value. We write E for the sequence of 1^*s of length n (i.e., the types for $\vec{y}$). We use $r = 1^*$ to denote the return type of the primitive recursion scheme, and $e^* = 1^*$ to denote the type of the recursive argument. We set $r' = e^* \cdot r$ and we construct the proof in Figure 4.6, where the subproof labeled with "cons" consists of a $*$-$r_{::}$ step followed by two identity axioms. □

Note that when displaying proofs, we omit usages of the exchange rule, which typically make it possible to apply left introduction rules on arbitrary formulas rather than just on

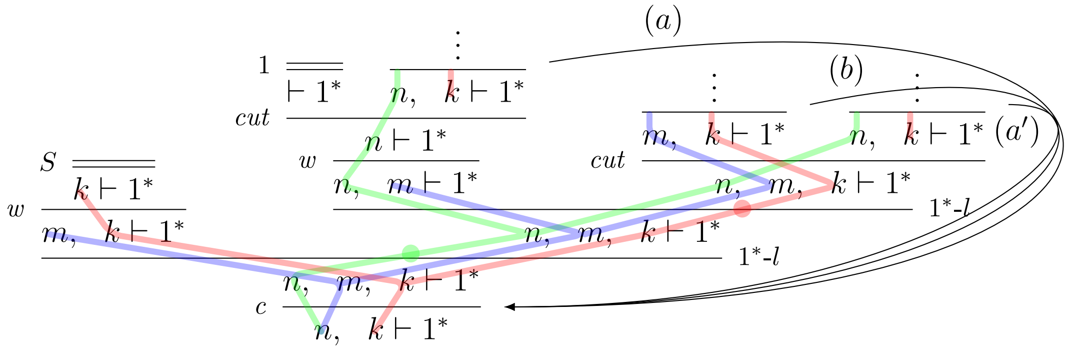


Figure 4.7: A regular proof for Ackermann-Péter's function; $n \triangleq m \triangleq k \triangleq 1^*$.

the first one. Moreover, we sometimes abbreviate sequences of steps or standalone proofs using double bars.

The above argument works in the fragment of C without arrows, sums, and intersections, and where star and unit are replaced with a base type for natural numbers together with the dedicated rules for 1^*. Pairs are exploited only to avoid using the contraction rule and remain in the affine fragment: with contraction, we could build a proof whose sequents mention only the formula 1^*.

As announced in the introduction, contraction makes it possible to go beyond primitive recursion:

Example 4.1. We give a regular proof whose semantics is Ackermann-Péter's function in Figure 4.7. The subproof labeled with S is a proof for the successor function. The subproof labeled with 1 is a proof for the constant value 1.

The preproof is valid: every infinite branch either goes infinitely often through loops (a) or (a'), in which case it is validated by the green and blue thread, where we go right on contraction steps (switching from green to blue) when the next visited backpointer is (b); or it eventually goes only through loop (b), in which case it is validated by the red thread.

Its semantics satisfies the same recursive equations as those defining Ackermann-Péter's function: we have

$$\begin{aligned}
[\epsilon](0,k) &= [0](0,0,k) = [00](0,k) = [000](k) = Sk\\
[\epsilon](Sn,0) &= [0](Sn,Sn,0) = [01](n,Sn,0) = [010](n,Sn) = [0100](n) = [01001](n,1)\\
&= [\epsilon](n,1)\\
[\epsilon](Sn,Sk) &= [0](Sn,Sn,Sk) = [01](n,Sn,Sk) = [011](n,Sn,k) = [0111](n,[0110](Sn,k))\\
&= [\epsilon](n,[\epsilon](Sn,k))
\end{aligned}$$

We prove in the Section 4.3 that we can actually represent all system T functions with regular proofs.

4.2.6 Turing completeness

We can also go beyond total functions by forgetting the validity criterion: we can encode the minimization operator μ using a regular but invalid preproof, so that every computable partial function can be represented by a regular preproof

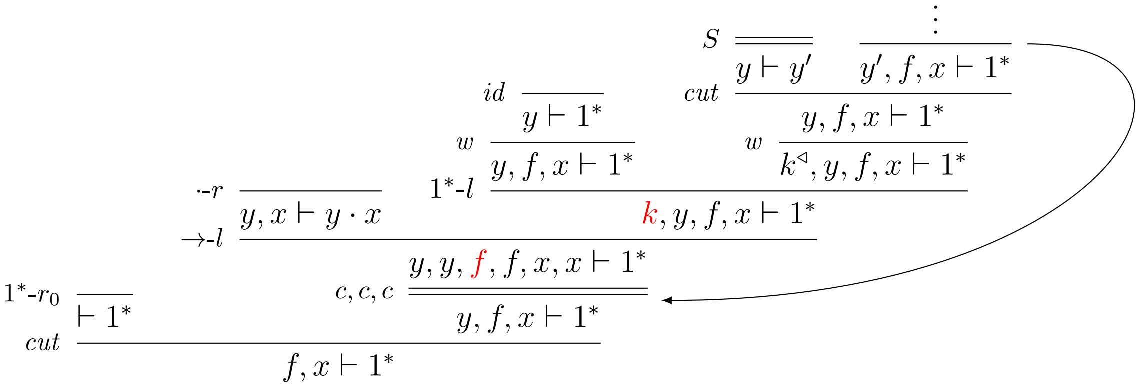


Figure 4.8: The preproof π_μ for minimization.

We define μ with one integer parameter x, as any tuple of parameters can be encoded in one. Thus μ is defined as follows: if $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, then $\mu(f)(x)$ is the smallest $y \in \mathbb{N}$ such that $f(y, x) = 0$, and is undefined if no such y exists.

Therefore, the μ operator has type $(\mathbb{N} \times \mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N}$. The preproof π_μ is represented in Figure 4.8. In this figure, x, y stand for 1^*, f stands for $1^* \cdot 1^* \to 1^*$, and k stands for 1^*: it stores the result $f(y, x)$. We note $k^\triangleleft$ the predecessor of k and y' the successor of y. Principal formulas may be emphasized by a red font.

The principle behind this preproof is simply to compute $k = f(y, x)$ for $y = 0, 1, 2, \ldots,$ and returns y as soon as $k = 0$. The preproof is not valid, as the infinite branch contains no validating thread. The only infinite thread in this branch is the one following x, which is never principal.

In order to give a semantic to such an invalid preproof (as a partial function on natural numbers), one can use the small-step semantic from Section 4.6: feed the proof with natural numbers and try to compute a result value with leftmost innermost reduction strategy. If this terminates, we can read back a natural number by Lemma 21, otherwise the function is undefined at the considered point.

4.3 Extended, resource-tracking system T

We define in this section the variant of system T we will work with. We use the following syntax for terms, where x ranges over a set of *variables* and i ranges over $0, 1$.

$$\begin{array}{rllll} M, N, O ::= & x & \mid \lambda x.M & \mid MN \\ & & \mid \langle M, N \rangle & \mid \text{let } \langle x, y \rangle := M \text{ in } N \\ & & \mid \langle\rangle & \mid \text{let } \langle\rangle := M \text{ in } N \\ & & \mid \mathbf{i}_i M & \mid \mathbf{D}(M; x.N; x.O) \\ & & \mid [] \mid M :: N & \mid \mathbf{R}(M; N; x.y.O) \\ & & \mid \langle\langle M, N \rangle\rangle & \mid \mathbf{p}_i M \end{array}$$

It consists of a lambda-calculus extended with pairs, singletons, sums, lists, and additive pairs. We let Γ, Δ range over *typing environments*, i.e., lists of pairs of a variable

$$
{\scriptstyle id}\,\frac{}{x : e \vdash x : e}
\qquad
{\scriptstyle x}\,\frac{\Gamma, y : f, x : e, \Delta \vdash M : g}{\Gamma, x : e, y : f, \Delta \vdash M : g}
\qquad
{\scriptstyle w}\,\frac{\Gamma \vdash M : f}{x : e, \Gamma \vdash M : f}
\qquad
{\scriptstyle c}\,\frac{x : e, x : e, \Gamma \vdash M : f}{x : e, \Gamma \vdash M : f}
$$

$$
{\scriptstyle \cdot\text{-}e}\,\frac{\Gamma \vdash M : e \cdot f \quad x : e, y : f, \Delta \vdash N : g}{\Gamma, \Delta \vdash \text{let } \langle x, y\rangle := M \text{ in } N : g}
\qquad
{\scriptstyle \cdot\text{-}i}\,\frac{\Gamma \vdash M : e \quad \Delta \vdash N : f}{\Gamma, \Delta \vdash \langle M, N\rangle : e \cdot f}
$$

$$
{\scriptstyle +\text{-}e}\,\frac{\Gamma \vdash S : e + f \quad x : e, \Delta \vdash M : g \quad y : f, \Delta \vdash N : g}{\Gamma, \Delta \vdash \mathbf{D}(S; x.M; y.N) : g}
\qquad
{\scriptstyle +\text{-}i_j}\,\frac{\Gamma \vdash M : e_j}{\Gamma \vdash \mathbf{i}_j M : e_0 + e_1}\; j \in \{0, 1\}
$$

$$
{\scriptstyle *\text{-}e}\,\frac{\Gamma \vdash L : e^* \quad \Delta \vdash M : g \quad x : e, y : g \vdash N : g}{\Gamma, \Delta \vdash \mathbf{R}(L; M; x.y.N) : g}
\qquad
{\scriptstyle *\text{-}i_\epsilon}\,\frac{}{\vdash [] : e^*}
\qquad
{\scriptstyle *\text{-}i_{::}}\,\frac{\Gamma \vdash M : e \quad \Delta \vdash N : e^*}{\Gamma, \Delta \vdash M :: N : e^*}
$$

$$
{\scriptstyle 1\text{-}e}\,\frac{\Gamma \vdash M : 1 \quad \Delta \vdash N : g}{\Gamma, \Delta \vdash \text{let } \langle\rangle := M \text{ in } N : g}
\qquad
{\scriptstyle 1\text{-}i}\,\frac{}{\vdash \langle\rangle : 1}
$$

$$
{\scriptstyle \to\text{-}e}\,\frac{\Gamma \vdash M : e \to f \quad \Delta \vdash N : e}{\Gamma, \Delta \vdash MN : f}
\qquad
{\scriptstyle \to\text{-}i}\,\frac{x : e, \Gamma \vdash M : f}{\Gamma \vdash \lambda x.M : e \to f}
$$

$$
{\scriptstyle \cap\text{-}e_i}\,\frac{\Gamma \vdash M : e_0 \cap e_1}{\Gamma \vdash \mathbf{p}_i M : e_i}\; i \in \{0, 1\}
\qquad
{\scriptstyle \cap\text{-}i}\,\frac{\Gamma \vdash M : e \quad \Gamma \vdash N : f}{\Gamma \vdash \langle\!\langle M, N\rangle\!\rangle : e \cap f}
$$

Figure 4.9: Typing rules for system T.

and a type. The type system is given in Figure 4.9. Unlike for C, typing derivations are just finite trees built from the rules, as usual. This type system however departs from the standard presentations in that it keeps track of resources: the rules for the various connectives are those of a linearly typed lambda-calculus. We include contraction and weakening rules (c, w), so that the standard typing rules for system T are all admissible (see [62, Appendix B.1] for more details on the equivalence between this version of system T and the standard one).

The structural and introduction rules are term-decorated versions of the corresponding rules of C (Figure 4.1). In contrast, the elimination rules differ: they follow the 'natural deduction' scheme and each of them intuitively contains a *cut* on the corresponding formula.

Let us focus on the recursion operator on lists ($\mathbf{R}$). This operator expects a list as first argument, and then two arguments for the cases of the empty and non-empty lists. Intuitively, we have

$$
\begin{aligned}
\mathbf{R}([]; M; x.y.N) &= M \\
\mathbf{R}(X{::}Q; M; x.y.N) &= N \{x \leftarrow X; y \leftarrow \mathbf{R}(Q; M; x.y.N)\}
\end{aligned}
$$

Note that this is an *iterator* rather than a *recursor*: the tail of the list (Q) is not given to N. This is not a restriction since recursors can be encoded from iterators and pairs. Its (elimination) typing rule is the following one:

$$
{\scriptstyle *\text{-}e}\,\frac{\Gamma \vdash L : e^* \quad \Delta \vdash M : g \quad x : e, y : g \vdash N : g}{\Gamma, \Delta \vdash \mathbf{R}(L; M; x.y.N) : g}
$$

Like in Dal Lago's system $\mathcal{H}(\emptyset)$ [23], the important point is that the third argument (the one being iterated) is typed in the empty environment—except for its two variables x for the head of the list and y for the value of the recursive call on the tail of the list. This is crucial in the affine system to get a linear recursion operator; this is not a restriction in the full system, thanks to arrows and contraction (see [62, Appendix B.1]).

Terms should always be considered as equipped with their typing derivation. A typed term is *affine* (resp. *linear*) when its typing derivation does not use c (resp. c and w).

Given a typing environment $\Gamma = x_1 : e_1, \ldots, x_n : e_n$, we write $\underline{\Gamma}$ for the list of types $e_1, \ldots, e_n$.

Definition 4.8. The semantics of a typed term $\Gamma \vdash M : e$ is the function $[M] \colon [\underline{\Gamma}] \to [e]$ defined as follows by induction on the typing derivation:

id : $[x](s) \triangleq s$

x : $[M](s,u,v,t) \triangleq [M](s,v,u,t)$

w : $[M](v,s) \triangleq [M](s)$

c : $[M](v,s) \triangleq [M](v,v,s)$

$\cdot$-i : $[\langle M, N\rangle](s,t) \triangleq \langle [M](s), [N](t)\rangle$

$+$-i_j : $[\mathbf{i}_j M](s) \triangleq [M](s)$

$*$-i_ϵ : $[[]]() \triangleq \epsilon.$

$*$-$i_{::}$: $[M :: N](s,t) \triangleq [M](s) :: [N](t)$

1-i : $[\langle\rangle]() \triangleq 1$

$\to$-i : $[\lambda x.M](s) \triangleq (u \mapsto [M](u,s))$

$\cap$-i : $[\langle\!\langle M, N\rangle\!\rangle](s) \triangleq \langle [M](s), [N](s)\rangle$

$\cdot$-e : $[\text{let } \langle x, y\rangle := M \text{ in } N](s,t) \triangleq [N](u,v)$ where the induction provided $[M](s) = \langle u, v\rangle$.

$+$-e : $[\mathbf{D}(S; x.M; y.N)](s,t) \triangleq [M]([S](s),t)$ if $[S](s) \in [e]$ and $[N]([S](s),t)$ otherwise.

$*$-e : $[\mathbf{R}(L; M; x.y.N)](s,t) \triangleq h(x_1, h(x_2, \ldots h(x_n, a) \ldots))$, where the induction provided a list $[L](s) = x_1, \ldots, x_n$, an element $a \triangleq [M](t)$, and a function $h \triangleq [N]$.

1-e : $[\text{let } \langle\rangle := M \text{ in } N](s,t) \triangleq [N](t)$

$\to$-e : $[MN](s,t) \triangleq [M](s)([N](t))$

$\cap$-e_i : $[\mathbf{p}_i M](s) \triangleq [M](s)$

Note that in the contraction case (c), the two occurrences of M are shorthands for two distinct stages of the typing derivation: the recursive call is made on a smaller typing derivation, even though the displayed term remains unchanged.

Example 4.2. We can define list concatenation as follows:

$$\lambda h.\lambda k.\mathbf{R}(h;k;x.qk.x{::}qk)$$

This term has type $e^* \to e^* \to e^*$ for every type e. Note that this term is strictly linear: it is typed without exchange, contraction and weakening.

Example 4.3. Remember that we code natural numbers as lists over the singleton set. Writing 1 for the constant $\langle\rangle{::}[]$ and S for the successor function $\lambda n.\langle\rangle{::}n$, we can define Ackermann-Péter's function as follows:

$$\lambda n.\mathbf{R}(n;S;\_.f.\lambda k.\mathbf{R}(k;f1;\_.r.fr))$$

This term can be typed with type $1^* \to 1^* \to 1^*$ in the empty environment. The outer recursion produces a function of type $1^* \to 1^*$. This term is not affine: we need the contraction rule since f is used twice in the outer recursion.

As announced before, system C contains system T:

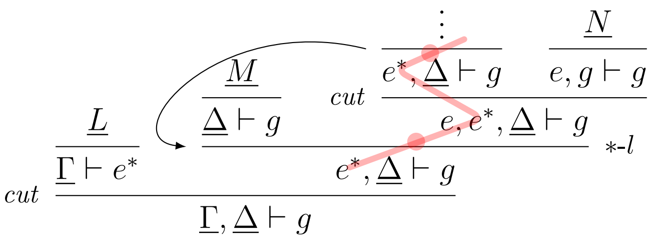
Theorem 4.2. *For every typing derivation $\Gamma \vdash M : e$, there exists a regular proof $\underline{M} : \underline{\Gamma} \vdash e$ such that $[\underline{M}] = [M]$. If M is affine/linear, so is $\underline{M}$.*

Proof. We proceed by induction on the typing derivation. The structural rules (exchange, weakening, contraction and identity) as well as the introduction rules of system T translate immediately to their counterparts in system C. It remains to deal with the elimination rules of system T. Leaving the $*$-e rule aside, they all translate into a cut on the eliminated formula, followed by an application of the corresponding left introduction rule (and an identity rule for the negative connectives $\cap$ and $\to$). For instance, for the $\cdot$-e case (i.e., term let $\langle x,y\rangle := M$ in N), we obtain two regular proofs $\underline{M} : \underline{\Gamma} \vdash e\cdot f$ and $\underline{N} : e, f, \underline{\Delta} \vdash g$ by induction, and we construct the following preproof:

$$\mathit{cut}\;\dfrac{\dfrac{\underline{M}}{\underline{\Gamma}\vdash e\cdot f}\qquad \cdot\text{-}l\;\dfrac{\dfrac{\underline{N}}{e,f,\underline{\Delta}\vdash g}}{e\cdot f,\underline{\Delta}\vdash g}}{\underline{\Gamma},\underline{\Delta}\vdash g}$$

This preproof is regular and valid: every infinite branch eventually belongs either to $\underline{M}$ or $\underline{N}$.

The $*$-e case (i.e., term $\mathbf{R}(L;M;x.y.N)$) is the only one where we introduce circularities: we obtain by induction three regular proofs $\underline{L} : \underline{\Gamma} \vdash e^*$, $\underline{M} : \underline{\Delta} \vdash g$ and $\underline{N} : e, g \vdash g$, and we construct the following preproof:


⋮
$\underline{N}$
$\underline{M}$
$e^*, \underline{\Delta} \vdash g$
$e, g \vdash g$
cut
$\underline{L}$
$\underline{\Delta} \vdash g$
$e, e^*, \underline{\Delta} \vdash g$
$*$-l
$\underline{\Gamma} \vdash e^*$
$e^*, \underline{\Delta} \vdash g$
cut
$\underline{\Gamma}, \underline{\Delta} \vdash g$

This preproof is regular by construction, and valid: the only infinite branch that does not eventually belong either to $\underline{L}$, $\underline{M}$ or $\underline{N}$ is the one along the constructed cycle, which it is validated by the red thread on e^*.

We use the contraction (resp. weakening) typing rule from system T only to translate contraction (resp. weakening) nodes in the starting proof, whence the second part of the statement. Moreover, we do not need to forge any new formula: all types appearing in $\underline{M}$ already appear in M. □

Encoding the term given in Example 4.2 for list concatenation yields the first proof in Figure 4.2. In contrast, encoding the term we provided for Ackermann-Péter's function (Example 4.3) does not yield the proof given in Figure 4.7: the outer recursion in this term constructs functional values, which give rise through the encoding to cycles over sequents with arrow types on the right. More importantly, the proof in Figure 4.7 has a non-trivial cycle structure, while in the proofs in the image of the encoding every infinite branch eventually loops on a single cycle of the finite presentation of the proof.

4.4 From affine C to affine T (using $\cap$ and $\rightarrow$)

The converse direction, encoding cyclic proofs into system T terms, is much harder since we have to delineate the possibly complex cycle structure of the starting proof in order to recover simple structural recursion schemes.

We provide a direct translation for the affine case in this section, where we proceed in two steps: first we show that affine regular proofs can be presented in such a way that cycles are associated to star formulas and occur in a hierarchic way (this is the notion of *ranked proof* in Section 4.4.3), this makes it possible to proceed bottom up in a second step, translating cycles associated to a given star formula into blocks of functions defined by mutual structural recursion (Section 4.4.4).

The second step is inspired by the one sketched in [28, Theorem 33] to translate regular proofs in LAL into equational proofs in action lattices. However, the authors of [28] did not realize that the first step we describe here is necessary, so that their argument is incorrect. The technique we present here makes it possible to repair it.

4.4.1 Proofs with backpointers

We first formalize precisely how regular proofs are represented by finite graphs with backpointers, as pictured earlier in the chapter.

Definition 4.9. A *proof with backpointers* (*bp-proof* for short) is a pair $\pi_{bp} = \langle \pi, Pts \rangle$ where π is a proof, and Pts is a set of *backpointers*, where each backpointer pt has a *source* address $src(pt)$ and a *target* address $tgt(pt)$, such that

- For all $pt \in Pts$, $tgt(pt) \sqsubset src(pt)$ and the subtrees of π rooted in $src(pt)$ and $tgt(pt)$ are isomorphic.
- For every infinite branch B of π, there exists a unique $pt \in Pts$ with $src(pt) \in B$.

An address of a bp-proof is a *source* if it is the source of a backpointer, it is *canonical* if it is a prefix of a source address.

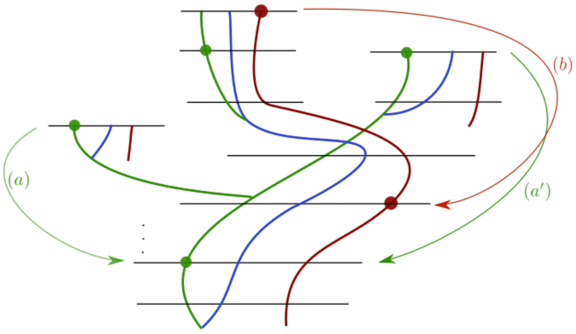


Figure 4.10: Threads and backpointers of the ibp-proof for Ackermann-Péter's function.

This definition is similar to that of 'cycle normal form' from [16]. The backpointers define a 'bar' across the proof, and by weak König's lemma the definition implies that in every bp-proof $\langle \pi, Pts \rangle$, the set Pts must be finite. To define a bp-proof, it suffices to describe the (finite) restriction of π to canonical addresses, as it was done earlier in the figures of this chapter. Moreover, every regular proof can be represented as a bp-proof. We show below that backpointers can be assumed to satisfy additional properties related to threads.

4.4.2 Idempotent normal form

Let π be a regular proof and let s be the maximal length of sequent antecedents in π. Let $\mathcal{F}$ be the set of partial functions $[\![0, s-1]\!] \to [\![0, s-1]\!]$. This set equipped with composition $\circ$ is a finite monoid. An element $f \in \mathcal{F}$ is *idempotent* if $f \circ f = f$.

If $u \sqsubset v$ are addresses in π, we define $f_{u,v} \in \mathcal{F}$ by

$$f_{u,v}(j) \triangleq \begin{cases} i & \text{if } \langle v, j \rangle \text{ is an ancestor of } \langle u, i \rangle \\ \textit{undefined} & \text{if no such } i \text{ exists} \end{cases}$$

Given a backpointer pt, we write f_{pt} for $f_{tgt(pt),src(pt)}$.

We say that a bp-proof is in *idempotent normal form*, or an *ibp-proof*, if for all backpointers pt, $tgt(pt)$ is a $*$-l address and f_{pt} is an idempotent with $f_{pt}(0) = 0$. This means that the branches that eventually loop only through this backpointer can be validated by the thread which is principal at $tgt(pt)$. Since there are other infinite branches in general, the validity criterion is still required.

Example 4.4. Let us go back to the proof for Ackermann-Péter's function given in Figure 4.7. The depicted backpointers do not point to $*$-l addresses; in order to have this property, we must shift the three backpointers one level up. We get idempotent backpointers by doing so: (a) and (a') both give rise to the idempotent partial function $0, 1 \mapsto 0$, and (b) to the idempotent $0, 1 \mapsto 1;\ 2 \mapsto 2$. However, while (a) and (a') preserve the principal position ($f_a(0) = f_{a'}(0) = 0$), this is not the case for (b): $f_b(0) = 1$. To fix this, observe that the branches that eventually visit only (b) are validated by the red thread on k. Accordingly, the backpointer (b) should thus point to the red $*$-l step on k rather than the green one on n. In order to obtain this, it suffices to shift (b) one level further up.

Doing so, we obtain an ibp-proof whose shape is depicted in Figure 4.10: the three backpointers are idempotent and preserve their principal position[3].

Proposition 4.1. Every regular proof π can be extended into an ibp-proof $\langle \pi, \mathit{Pts} \rangle$.

The rest of this subsection is devoted to the proof of this proposition. The key idea is that since $\mathcal{F}$ is a finite monoid, any sequence containing sufficiently many elements has an idempotent infix. This makes it possible to cut every infinite branch of the starting proof by inserting an idempotent backpointer between two of the infinitely many principal positions of a thread validating the branch.

Let π be a regular proof. We have to define a set of backpointers turning π into an ibp-proof.

We first establish a generic lemma. A *backpointer condition* P is a property of bp-proofs of the form: "for each backpointer pt, a property $P(pt)$ depending only on $\mathit{src}(pt)$, $\mathit{tgt}(pt)$, and the branch from the root of the proof to $\mathit{src}(pt)$ is verified".

We say that a backpointer pt is *correct* when it verifies the first item from Definition 4.9, i.e., the subtrees rooted in $\mathit{src}(pt)$ and $\mathit{tgt}(pt)$ are isomorphic.

Lemma 17. *Let π be a preproof and P be a backpointer condition such that for every infinite branch of π, there exists a correct backpointer pt such that $P(pt)$ is satisfied. Then π can be turned into a bp-preproof where all backpointers satisfy P.*

Proof. For each infinite branch ρ of π, we define the backpointer pt_ρ given by the hypothesis of the Lemma.

Let $\mathit{Pts}_0 = \{pt_\rho \mid \rho \text{ branch of } \pi\}$, and $\mathit{Pts}_1 = \{pt \in \mathit{Pts}_0 \mid \forall pt' \in \mathit{Pts}_0, \mathit{src}(pt') \not\sqsubset \mathit{src}(pt)\}$, i.e., we only keep pointers from Pts_0 with a minimal source. We show that Pts_1 is finite. Indeed, assume Pts_1 is infinite, and let $T = \{u \mid \exists pt \in \mathit{Pts}_1, u \sqsubseteq \mathit{src}(pt)\}$. Since T contains all sources from Pts_1, and that this sources are incomparable with each other, T is infinite. By König's lemma, since T is finitely branching, T contains an infinite branch ρ. By definition of Pts_1, there exists $pt \in \mathit{Pts}_1$ with $\mathit{src}(pt) \sqsubseteq \mathit{src}(pt_\rho)$. Let v be an address of ρ with $\mathit{src}(pt) \sqsubseteq v$. Since ρ is contained in T, there must be $pt' \in \mathit{Pts}_1$ with $v \sqsubseteq \mathit{src}(pt')$. We obtain $\mathit{src}(pt) \sqsubset \mathit{src}(pt')$, contradicting the fact that $pt' \in \mathit{Pts}_1$. We can thus conclude that Pts_1 is finite. Let $\mathit{Pts}_2 = \{pt \in \mathit{Pts}_1 \mid \forall pt' \in \mathit{Pts}_1, \mathit{src}(pt) = \mathit{src}(pt') \Rightarrow \mathit{tgt}(pt) \sqsubseteq \mathit{tgt}(pt')\}$, i.e., for each possible source we keep only the pointer with the smallest target. Since each pointer pt in Pts_2 is correct and satisfies $P(pt)$, and since each branch of π contains the source of exactly one pointer from Pts_2, we obtain that $\langle \pi, \mathit{Pts}_2 \rangle$ is a bp-proof satisfying the backpointer condition P. □

Thanks to Lemma 17, in order to show Proposition 4.1 it suffices to show the following lemma:

Lemma 18. *If π is a regular proof, every infinite branch ρ of π can be equipped with an idempotent correct backpointer.*

[3] The pictures can be slightly confusing here, because we do not include exchange steps. While formally, principal positions always have index 0, whence the constraint $f_{pt}(0) = 0$ in our definition of ibp-proof, the index of the principal position for the generalized $*$-l step used at address 01 (on k in Figure 4.7) is graphically 2, so that the constraint for the shifted backpointer (b) becomes $f_b(2) = 2$ with this graphical intuition.

Proof. Let s be the maximal length of sequent antecedents in π and $\mathcal{F}$ be the set of partial functions on $[\![0, s-1[\![$.

Let $eval : \mathcal{F}^* \to \mathcal{F}$ be the evaluation morphism, defined inductively by $eval(\epsilon) = id$ and $eval(\vec{u}f) = eval(\vec{u}) \circ f$. Since $\mathcal{F}$ is a finite monoid, there exists $m \in \mathbb{N}$ such that any word $\vec{u} \in \mathcal{F}^m$ contains an infix $\vec{v} \in \mathcal{F}^+$ such that $eval(\vec{v})$ is idempotent. This is a well-known basic consequence of Ramsey's theorem.

We say that two $*$-l addresses u, v have same *type* if the subtrees rooted in u, v in π are isomorphic. By extension, the type of a position is the type of its address.

Since π is valid and the number of distinct types is finite, every branch of π contains a thread going through infinitely many principal positions of the same type, and in particular it is the case for the branch ρ where we want to find an idempotent correct backpointer. Let $n \in \mathbb{N}$ such that the prefix of ρ of length n contains a thread which goes through $m+1$ such positions $\langle v_0, 0\rangle, \langle v_1, 0\rangle, \ldots, \langle v_m, 0\rangle$ of the same type.

For all $i \in [\![1, m]\!]$, we define $f_i = f_{v_{i-1}, v_i} \in \mathcal{F}$ as above. By choice of m, there exists $i < j \in [\![1, m]\!]$ such that $f = f_i \circ f_{i+1} \circ \cdots \circ f_j$ is idempotent. Moreover, as witnessed by the thread t, we have $f(0) = 0$. We define a backpointer pt with $src(pt) = v_j$ and $tgt(pt) = v_{i-1}$. □

Together with Lemma 17, we can conclude that every regular proof can be extended into an ibp-proof: we have proved Proposition 4.1.

4.4.3 Ranked proofs

We still need one more step before translating proofs into system T terms: we use the following notion of ranks in order to organize cycles in such a way that they can be translated into structural recursions. Intuitively, we mark positions with natural numbers (their rank) in such a way that positions marked with the same rank give rise to a single recursive definition, and positions with highest rank give rise to the outermost recursors in the produced term.

A *ranked proof* is a tuple $\langle \pi, Pts, rk\rangle$ such that $\pi_{bp} = \langle \pi, Pts\rangle$ is an ibp-proof and rk is a function from positions of π to $\mathbb{N}$ satisfying the following properties, where we write $rk(v)$ for $rk\langle v, 0\rangle$ when v is a $*$-l address.

(BP) backpointers preserve ranks: for all $pt \in Pts$, for all i, $rk\langle src(pt), i\rangle = rk\langle tgt(pt), i\rangle$.

(Con) Positions with the same rank are strongly connected via threads and backpointers with that rank.

(Dec) Ranks decrease along threads, except when passing through $*$-l steps of higher ranks: if $\langle v, i\rangle$ is the parent of $\langle w, j\rangle$, then either we have $rk\langle v, i\rangle \leq rk\langle w, j\rangle$, or v is a $*$-l address and $rk\langle w, j\rangle, rk\langle v, i\rangle < rk(v)$.

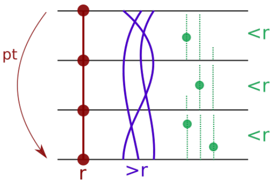
(Thd) Backpointers preserve threads of higher ranks: for all $pt \in Pts$, for all i such that $rk\langle tgt(pt), i\rangle > rk\langle tgt(pt), 0\rangle$, there is a thread from $\langle tgt(pt), i\rangle$ to $\langle src(pt), i\rangle$.

(Blk) If $u \sqsubset v \sqsubset w$ are $*$-l addresses with $rk(u) = rk(w)$, then $rk(v) \leq rk(u)$.

(Org) A $*$-l address v is an *origin* of rank r if v is a minimal $*$-l address with $rk(v) = r$. We require that if $u \sqsubset v$ are origin addresses then $rk(u) > rk(v)$.

These conditions are meant to enforce some inductive structure on the proof. They are such that in a ranked proof, cycles in computations can be considered as nested "`for`" loops, where higher ranks correspond to outer loops. We briefly give some explanations on how to interpret these rules in light of this intuition. Rules (BP) and (Con) ensure the local coherence of ranks with respect to the structure of the proof. Rule (Dec) expresses that lower ranks correspond to innermost loops, by restricting how the computation can transition from a rank to another. Rule (Thd) and (Blk) express that computations in inner loops do not interfer with outer loops, it simply put them on pause. Finally, rule (Org) stipulates that outer loops start before inner ones in the computation.

Let us now investigate the formal consequences of these rules. By (BP) a ranked proof uses only finitely many ranks. Rule (Blk) implies that the threads enforced by condition (Thd) are actually spectator from $\langle tgt(pt), i\rangle$ to $\langle src(pt), i\rangle$. Together with (Dec), this means that threads along a backpointer with rank r behave like in the picture below:



Note that the conditions on ranks imply validity:

Lemma 19. *Every (affine) ranked preproof is valid.*

Proof. We must exhibit a valid thread for each infinite branch of the preproof.

Let $\langle \pi, Pts, rk\rangle$ be an (affine) ranked preproof. Let ρ be an infinite branch of π, corresponding to an infinite path b in the canonical graph of π, staying in canonical address and following backpointers. Let Pts^∞ be the restriction of Pts to the backpointers that are seen infinitely often when going along b. This set is not empty because b is infinite and Pts is finite. Let r be the maximal rank in Pts^∞ and bp be the associated backpointer:

$$r = \max\{rk(src(pt)) \mid pt \in Pts^\infty\} = rk(src(bp))$$

There exists some node v in the infinite path b such that from this node the only backpointers that are seen form exactly the set Pts^∞. Note that from this point every node is between $tgt(pt)$ and $src(pt)$ for some $pt \in Pts^\infty$ (depending on the current node). Let's follow (in b) the thread of the principal formula of the first occurrence of the node $src(bp)$ after v. Then the thread goes only through positions of the proof that are located between the target and the source of a backpointer of rank $r' \leq r$. If $r' < r$, the thread exists and stays spectator between those points by (Thd). If $r' = r$, the thread also exists between the target and the source of the backpointer because π being a ranked preproof implies in particular that it is an ibp-preproof. Moreover this thread is principal infinitely often because the node $src(bp)$ is visited infinitely often. Thus any branch ρ is valid, and the ranked preproof π is valid. □

Before proving that affine regular proof can be assigned ranks, we prove the following strengthening of Lemma 18.

Lemma 20. *Let π be a regular proof, and $\langle u, 0\rangle$ be a principal position of π. Every infinite branch of π can be equipped by a correct idempotent backpointer pt such that*

- *either $\langle src(pt), 0\rangle$ and $\langle tgt(pt), 0\rangle$ are ancestors of $\langle u, 0\rangle$,*

- *or the segment $[tgt(pt), src(pt)]$ contains no principal position that is an ancestor of $\langle u, 0\rangle$.*

Proof. This is an adaptation of the proof of Lemma 18. When the branch ρ is fixed, two cases can occur:

- if infinitely many ancestors of $\langle u, 0\rangle$ are principal on ρ, then infinitely many of them have the same type, and we can use the proof of Lemma 18 to define a correct idempotent backpointer between two of them.

- if only finitely many ancestors of $\langle u, 0\rangle$ are principal on ρ, it suffices to consider a suffix ρ' of ρ containing none of these positions, and use the proof of Lemma 18 to define a correct idempotent backpointer in this suffix. □

We can now establish the main proposition of this subsection:

Proposition 4.2. Every affine and regular proof π can be extended into a ranked proof $\langle \pi, Pts, rk\rangle$.

Proof. We describe a recursive algorithm that builds a set of backpointers and assigns ranks to all canonical positions.

1. Use Proposition 4.1 to obtain Pts_0 such that $\pi^0_{bp} = \langle \pi, Pts_0\rangle$ is an ibp-proof, and consider the *canonical graph* G_0 consisting of the restriction of π to canonical addresses, and where sources and targets of backpointers from Pts_0 are identified.

2. We consider each strongly connected component (SCC) of G_0 separately, to assign ranks in the corresponding parts of π. When ranks have been assigned in each SCC, a shift is applied (i.e., all ranks of the same SCC are shifted by the same amount) so that different SCCs do not share ranks, and rules (Dec) and (Org) are respected.

3. We proceed as follows to assign ranks within a SCC of the canonical graph. By strong connectedness, we build an infinite path visiting all nodes of this SCC infinitely many times. This corresponds to an infinite branch in π, which must be validated by a *master thread*: a thread going through all backpointers infinitely many times. All positions of this master thread—exactly one per sequent of the SCC since we are in the affine case—are assigned with a maximal rank M. (This rank M is actually a placeholder standing for “maximal rank in the current SCC”: it has to be shifted to an appropriate value after the subsequent recursive calls are completed.)

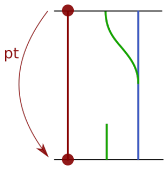
4. We now need to reorganize backpointers of the ibp-proof in order to respect rule (Thd) and (Blk), by forbidding a $*$-l rule of maximal rank M to occur in the scope of a backpointer linking rules of lower rank (to be assigned later). We do so using Lemma 17 and Lemma 20, taking the origin of rank M as distinguished position $\langle u, 0\rangle$. The latter lemma gives us idempotent backpointers that are either linking addresses of rank M, or that do not contain addresses of rank M in their scope. In this last case the thread of rank M is spectator between the source and the target of the backpointer.

5. At this point, we are done with the positions of rank M. Since backpointers have been updated, we need to recompute the canonical graph. We remove from it all $*$-l steps of rank M, thus creating new SCCs, and we proceed recursively from step 2.

This process terminates, because the maximal number of positions with unassigned rank in a sequent of the considered SCC decreases at each step. Indeed, the master thread visits every sequent of the considered SCC, so that we assign the rank M to a star position in each of these sequents.

The algorithm generates a set of backpointers Pts and a rank function rk such that $\langle \pi, Pts, rk \rangle$ is a ranked proof. Rule (BP) is ensured by the identification of sources and target of pointers in canonical proof graphs. Rules (Dec) and (Org) are ensured when shifting the ranks of SCC after internal computations. Rule (Thd) is ensured by the choice of a maximal rank for the master thread, that must be preserved in all paths of the canonical graph. Rule (Blk) is ensured by step 4 and by avoiding overlapping of ranks between different SCCs. Rule (Con) is ensured by step 3, where all positions assigned with the same rank are connected by a thread, and by avoiding overlapping of ranks between SCCs. □

To see why this construction fails in the presence of contraction, consider the ibp-proof for Ackermann-Péter's function given in Figure 4.10. It contains the pattern depicted on the right, where the green thread is not preserved by the red backpointer, but is somehow "saved" via an auxiliary blue thread. When considering an infinite branch visiting infinitely many times the three backpointers (a, a', b) from Figure 4.10, we obtain a validating thread that alternates between blue and green positions (see Example 4.1). We should assign a maximal rank to all these positions, but then condition (Thd) is violated for the red backpointer, no matter how we try to shift it away.



4.4.4 Affine translation

We can finally translate ranked proofs into system T terms.

Given a list of expressions $E = e_1, \ldots, e_n$ and a list of variables $X = x_1, \ldots, x_n$, we write $X : E$ for the typing environment $x_1 : e_1, \ldots, x_n : e_n$. We moreover write $E \to f$ for the type $e_1 \to \ldots \to e_n \to f$.

Theorem 4.3. *For every regular and affine proof $\pi : E \vdash e$ and every list X of variables of size $|E|$ there exists an affine term M such that $X : E \vdash M : e$ and $[\pi] = [M]$.*

Proof. By Proposition 4.2, it suffices to prove the property for ranked proofs. We do so by lexicographic induction on the *rank* of the proof followed by its *size*, where the rank of a ranked proof is its highest assigned rank and the size of a bp-proof is its number of canonical addresses.

If the rule applied at the root of the proof is not a $*$-l rule, there are no backpointers pointing to the root, so that the subproofs rooted at its premisses are standalone and ranked proofs of strictly smaller size and at most same rank. We translate those by induction, and we combine the results to obtain the desired term. For instance, in the case of a *cut*, we obtain two terms M and N and we construct the term $(\lambda x.M)N$. The

$$\text{1-}i\ \frac{}{\vdash \langle\rangle : 1} \qquad \text{*-}i_\epsilon\ \frac{}{\vdash [] : e^*} \qquad \text{*-}i_{::}\ \frac{X : E \vdash M : e \quad Y : F \vdash N : e^*}{X : E, Y : F \vdash M :: N : e^*}$$

$$\cdot\text{-}i\ \frac{X : E \vdash M : e \quad Y : F \vdash N : f}{X : E, Y : F \vdash \langle M, N\rangle : e \cdot f} \qquad +\text{-}i_i\ \frac{X : E \vdash M : e_i}{X : E \vdash \mathbf{i}_i M : e_0 + e_1}\ i \in \{0,1\}$$

$$\to\text{-}i\ \frac{x : e, X : E \vdash M : f}{X : E \vdash \lambda x.M : e \to f} \qquad \cap\text{-}i\ \frac{X : E \vdash M : e \quad X : E \vdash N : f}{X : E \vdash \langle\!\langle M, N\rangle\!\rangle : e \cap f}$$

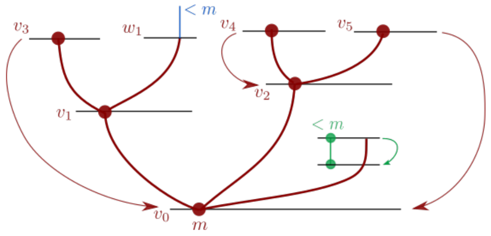
Figure 4.11: Typing derivations for translating right rules of C into T. Green sequents represent the results obtained through the induction.

cases for right introduction rules are given in Figure 4.11; the ones for left introduction rules and cut are given in Figure 4.12.

Otherwise, the root must be of the form $e^*, E_0 \vdash e_0$, and its rank m must be maximal by condition (Org). This is where we have to output a recursor. We explore the ancestry tree of e^* as long as its rank is m and we find:

- canonical $*$-l addresses $v_0, \ldots, v_n, \ldots, v_{n'}$ of rank m, labeled with sequents $(e^*, E_i \vdash e_i)_{i \in [\![0,n]\!]}$ (with $v_0 = \epsilon$), such that $v_0, \ldots, v_n$ are not sources and $v_{n+1}, \ldots, v_{n'}$ are sources (pointing to the former ones);
- canonical addresses $w_1, \ldots, w_p$ labeled with sequents $(F_j, e^*, F'_j \vdash f_j)_{j \in [\![1,p]\!]}$ such that $\langle w_j, |F_j|\rangle$ has rank $< m$.

The situation is illustrated in the following picture:



We construct a term that defines simultaneously all functions $([v_i])_{i \in [\![0,n]\!]}$, by an encoding of mutual recursion. The addresses w_j correspond to points where we escape from this recursion, e.g., to enter a recursion on another argument.

Let $g \triangleq e^* \cap \bigcap_{i \in [\![0,n]\!]}(E_i \to e_i)$. This type g is the 'invariant' of our recursion: it contains room for all the mutually defined functions and for a copy of the starting recursive argument.

Given a list x, X of variables for the sequence e^*, E_0, we construct a term M of the form

$$M \triangleq (\mathbf{p}_0\mathbf{p}_1\mathbf{R}(x; M^\epsilon; y.k.M^{::}))\ X_1\ \ldots\ X_l$$

with $\vdash M^\epsilon : g$ and $y : e, k : g \vdash M^{::} : g$, so that we have $x : e^*, X : E_0 \vdash M : e_0$ as expected.

$$1\text{-}e\;\dfrac{id\;\dfrac{}{x : 1 \vdash x : 1} \qquad X : E \vdash M : e}{x : 1, X : E \vdash \text{let } \langle\rangle := x \text{ in } M : e}$$

$$\cdot\text{-}e\;\dfrac{id\;\dfrac{}{z : e \cdot f \vdash z : e \cdot f} \qquad x : e, y : f, X : E \vdash N : g}{z : e \cdot f, X : E \vdash \text{let } \langle x, y\rangle := z \text{ in } N : g}$$

$$+\text{-}e\;\dfrac{id\;\dfrac{}{z : e + f \vdash z : e + f} \qquad x : e, X : E \vdash M : g \qquad y : f, X : E \vdash N : g}{z : e + f, X : E \vdash \mathbf{D}(z; x.M; y.N) : g}$$

$$\to\text{-}e\;\dfrac{\to\text{-}i\;\dfrac{x : f, Y : F \vdash M : g}{Y : F \vdash \lambda x.M : f \to g} \qquad \to\text{-}e\;\dfrac{id\;\dfrac{}{y : e \to f \vdash y : e \to f} \qquad X : E \vdash N : e}{y : e \to f, X : E \vdash yN : f}}{y : e \to f, X : E, Y : F \vdash (\lambda x.M)(yN) : g}$$

$$\to\text{-}e\;\dfrac{\cap\text{-}e_i\;\dfrac{id\;\dfrac{}{z : e_0 \cap e_1 \vdash z : e_0 \cap e_1}}{z : e_0 \cap e_1 \vdash \mathbf{p}_i z : e_i} \qquad \to\text{-}i\;\dfrac{x : e_i, X : E \vdash M : f}{X : E \vdash \lambda x.M : e_i \to f}}{z : e_0 \cap e_1, X : E \vdash (\lambda x.M)(\mathbf{p}_i z) : f}$$

$$\to\text{-}e\;\dfrac{X : E \vdash N : e \qquad \to\text{-}i\;\dfrac{x : e, Y : F \vdash M : g}{Y : F \vdash \lambda x.M : e \to g}}{X : E, Y : F \vdash (\lambda x.M)N : g}$$

Figure 4.12: Typing derivations for translating left rules of C into T. Green sequents represent the results obtained through the induction. The last derivation is the one for the cut rule, as described in Section 4.4.4. The derivations used for weakening and identity are trivial and omitted here.

This term iterates the function $\lambda yk.M^{::}$ over the list x, starting from M^{ϵ}, to obtain a value of type g; then it calls the first mutually defined function in that value.

Defining M^{ϵ} is easy. For all $i \in [\![0, n]\!]$, the subproof rooted at v_i0, i.e., the left premiss of the $*$-l node at v_i, is a standalone ranked proof of $E_i \vdash e_i$, with strictly smaller rank and size. Indeed, by (Blk), backpointers whose source belongs to this subproof may not point below it. We can thus translate these subproofs by induction and obtain terms $M_i^{\epsilon} \vdash E_i \to e_i$ for all $i \leq n$. We combine them as follows:

$$M^{\epsilon} \triangleq \langle\!\langle [], \langle\!\langle M_0^{\epsilon}, \dots, M_n^{\epsilon} \rangle\!\rangle \rangle\!\rangle$$

Defining $M^{::}$ is more involved. Our goal here is to obtain for all $i \leq n$ a term $M_i^{::}$ of type $E_i \to e_i$ in environment $y : e, k : g$. Then we will combine those terms as follows:

$$M^{::} \triangleq \langle\!\langle y :: \mathbf{p}_0 k, \langle\!\langle M_0^{::}, \dots, M_n^{::} \rangle\!\rangle \rangle\!\rangle$$

As expected, we use the subproof rooted at v_i1 to define $M_i^{::}$. However, this subproof ends with $e, e^*, E_i \vdash e_i$, and is not standalone: backpointers along e^* may escape this subproof. To obtain a ranked proof of $e, g, E_i \vdash e_i$, we copy this subproof bottom up, substituting ancestors of e^* by g as long as their rank is m. Several situations appear when doing so:

- we reach a $*$-l node for which e^* is principal: an address v_{k_0} with $k_0 \leq n'$. If $k_0 \leq n$ we set $k \triangleq k_0$, otherwise v_{k_0} is the source of a backpointer to v_{k_1} for some $k_1 \leq n$ and we set $k \triangleq k_1$. We stop copying and we insert the following finite proof:

$$\begin{array}{rc} \to\text{-}l, id & \overline{\overline{}} \\ & E_k \to e_k, E_i \vdash e_k \\ \cap\text{-}l_0, \cap\text{-}l_1 & \overline{\overline{}} \\ & g, E_k \vdash e_k \end{array}$$

- we reach a node for which e^* is spectator and its rank decreases. This means we reached an address w_j for some $j \in [\![1, p]\!]$. We insert a $\cap$-l_0 rule to transform the type g in the produced proof back into an e^*, and we copy the remainder of the ranked proof as is, without performing the substitution anymore.

- we reach a backpointer following another star formula. Since m is maximal, the target of this backpointer must be above v_i1 by (Blk). Moreover if e^* still occurs at the source of this backpointer, its thread must have been preserved by (Thd) and remained spectator, so that e^* was uniformly substituted into g. The backpointer can thus be inserted in the copied proof.

The produced object is a ranked proof (with smaller rank); in particular, the ranks of principal positions it contains must have their origins inside it by (Blk), so that condition (Org) is preserved. We can thus obtain $M_i^{::}$ by induction. □

The type g used as invariant for recursions in the above translation is reminiscent of the type r we used to encode primitive recursion (Figure 4.6). Its first component gives access to a copy of the current value of type e^* in those cases where we exit the recursion before exhausting this value.

It is crucial that g is defined using additive pairs in order to obtain an affine term. Indeed, while M^{ϵ} is typed in the empty context, the variables y and k must be provided to all components of $M^{::}$. Contraction would thus be required if we had been using multiplicative pairs. Symmetrically, having additive pairs makes it possible to avoid weakenings at the various places where values of type g are used (to perform recursive calls, to get the current value of type e^*, and to eventually call the first mutually defined function).

Remark 4.4. Let C' be the fragment of C where contraction is allowed, except on star formulas. The above argument still works and gives us a direct and uniform encoding of C' into T: threads in C' behave exactly like in affine C. Moreover, contraction on star formulas is derivable in C' (by an easy adaptation of Lemma 16), so that Theorem 4.2 can be refined into an encoding of T into C'. C' and T are thus equally expressive, at all types.

This correspondence makes C' quite appealing and we could have chosen to take it as the main system. However, C' unnecessarily rules out programs such as the implementation of Ackermann-Péter's function in Figure 4.7 (which does not require the arrow type, unlike the implementation we obtain in C' via Example 4.3 and Theorem 4.2—it is actually not clear that we can implement this function in C' without using arrow types).

The structure of threads is more subtle in C than in C'—cf. Remark 4.2; we find it intriguing and we would like to advocate its study.

4.5 Subsystems of second-order arithmetic

We define in this section the second-order logics ACA_0 and RCA_0, as well as the properties we need about them. A comprehensive introduction to these theories and the 'reverse mathematics' program can be found in [78, 47]. Also, an excellent introduction to the functional interpretations of proofs, including for the theories covered here, is [8].

4.5.1 Some 'second-order' theories of arithmetic

We consider a two-sorted first-order language, henceforth called 'second-order logic' as is traditional, consisting of individual variables x, y, z etc., terms s, t, u etc., and set variables X, Y, Z etc. We have quantifiers for both the individual sort and the set sort. There is a single binary relation symbol $\in$ connecting the two sorts, allowing us to write formulas of the form $t \in X$. (We may sometimes write $X(t)$ instead.) We have an equality relation for the individual sort; set equality is expressed by extensionality: $X = Y \triangleq \forall x, (X(x) \Leftrightarrow Y(x))$.

The *language of arithmetic* consists of the non-logical symbols $0, S, +, \times, <$, with their usual intended interpretations. A *theory* is just a set of closed formulas, and we say that a theory T *proves* a formula φ if φ is a logical consequence of T. The base theory $\mathsf{Q2}$ extends second-order logic by basic axioms governing the behavior of the non-logical symbols, namely stating that $(0, S0, +, \times, <)$ is a commutative semiring discretely ordered by $<$, with S representing the successor. *Bounded* quantifiers are of the shape $\exists x, (x < t \wedge \varphi)$ and $\forall x(x < t \Rightarrow \varphi)$.

Definition 4.10 (Arithmetical hierarchy)**.** A possibly open formula is in $\Sigma^0_0 = \Pi^0_0 = \Delta^0_0$ if it has only bounded quantifiers. From here we define the *arithmetical hierarchy* as follows:

- Σ^0_{k+1} formulas are those of the form $\exists\vec{x}, \varphi$ with $\varphi \in \Pi^0_k$.
- Π^0_{k+1} formulas are those of the form $\forall\vec{x}, \varphi$ with $\varphi \in \Sigma^0_k$.

The formulas of the arithmetical hierarchy are the *arithmetical formulas*, those that do not contain second-order quantifiers. A formula is Δ^0_k (provably in a theory T) if it is equivalent to both a Σ^0_k formula and a Π^0_k formula (resp. provably in T).

We define the following axiom schemata for *induction* and *comprehension*, where free variables may occur in φ:

$$(\varphi(0) \wedge \forall x, (\varphi(x) \Rightarrow \varphi(Sx))) \Rightarrow \forall x, \varphi(x) \tag{induction}$$
$$\exists X \forall x, (X(x) \Leftrightarrow \varphi) \tag{comprehension}$$

Definition 4.11 (ACA_0, RCA_0)**.**

- ACA_0 extends $\mathsf{Q2}$ by instances of induction and comprehension where φ is arithmetical.
- RCA_0 extends $\mathsf{Q2}$ by instances of induction where $\varphi \in \Sigma^0_1$ and instances of comprehension where φ is provably Δ^0_1.

Note that ACA_0 can equivalently be defined as $\mathsf{Q2}$ extended with arithmetical instances of comprehension and the following single induction axiom:

$$(\forall X, X(0) \wedge \forall x, (X(x) \Rightarrow X(Sx))) \Rightarrow \forall x, X(x) \tag{induction'}$$

Also note that in the above definition of RCA_0, the available instances of comprehension and the notion of RCA_0 itself are mutually defined. It is equivalent to extending $\mathsf{Q2}$ by Σ^0_1 instances of induction and the following axiom scheme, where φ and ψ vary over Σ^0_1 formulas.

$$\forall x(\varphi \Leftrightarrow \neg\psi) \Rightarrow \exists X \forall x(X(x) \Leftrightarrow \varphi)$$

We often write formulas in natural language to stand for their obvious formalization in arithmetic. We do not concern ourselves with such low-level encodings in the sequel. Statements written in natural language are typically robust under the choice of encoding.

4.5.2 Provably total computable functions

The utility of the second-order theories we have introduced, for this work, lies in the fact that they may reason about programs and potentially infinite computations, by way of quantification over set variables. What is more, the functions they may well-define, or programs that they may prove terminating, are well-understood, in terms of their computational strength: we may freely use such functions in logical formulas without affecting logical complexity.

Proposition 4.3 (Witnessing for ACA_0)**.** Suppose ACA_0 proves $\forall\vec{x}\exists y, \varphi(\vec{x}, y)$, where φ is Σ^0_1 and contains no set symbols. Then there is a term M of T with a typing derivation $x_1 : 1^*, \ldots, x_n : 1^* \vdash M : 1^*$ such that $\mathbb{N} \models \forall\vec{x}, \varphi(\vec{x}, [M])$.

This result follows immediately from the conservativity of ACA_0 over Peano Arithmetic and thence, under the Gödel-Gentzen double-negation translation, Gödel's Dialectica functional interpretation of Heyting Arithmetic into T (see, e.g., [8] for more details).

A similar characterization of RCA_0 is known: this theory is conservative over $I\Sigma_1$, the restriction of Peano Arithmetic to Σ_1-induction, which is known to well-define only primitive recursive functions. This result was originally established by Parsons in his *predicative* functional interpretation [70], though there are also direct proofs, e.g., by cut-elimination (see [19]).

Proposition 4.4 (Witnessing for RCA_0)**.** Suppose RCA_0 proves $\forall \vec{x} \exists y, \varphi(\vec{x}, y)$, where φ is Σ^0_1 and contains no set symbols. Then there is a primitive recursive function f such that $\mathbb{N} \vDash \forall \vec{x}, \varphi(\vec{x}, f(\vec{x}))$.

4.5.3 Reverse mathematics of cyclic proof checking

While the notion of preproof can easily be formalized already in RCA_0, dealing with the validity criterion is non-trivial: we must be able to verify it within our theories too. In fact, the correctness of a generic cyclic proof checker is not available in RCA_0 [25]. However, it is known that for any fixed preproof, RCA_0 can check whether it is valid or not:

Proposition 4.5 ([25], also implicit in [57])**.** Let π be a regular proof. Then RCA_0 proves that π (written as a finite graph) is a proof, i.e., that each infinite branch contains a valid thread.

This is a nontrivial result that is obtained by formalizing the reduction of proof validity to the universality problem for nondeterministic Büchi automata and proving the correctness of a universality algorithm.

4.6 Small steps reduction semantics for C

We fix a regular proof π in this section. We define a simplified version of the rewriting system used in [28] to prove cut-elimination in the system LAL. *Programs* are defined via the following syntax, where v ranges over addresses.

$$P, Q ::= \langle\rangle \mid [] \mid P :: P \mid v(P_1, \dots, P_n)$$

The first three entries correspond to constructors for singletons and lists. The fourth one corresponds to calling the node v of π with the given list of arguments. This syntax is much simpler than that used in [28]: we put constructors only for singletons and lists, which are the only types we want to observe in the present work. In particular, we do not need lambda abstractions to represent functional values. Also note that here programs are always 'closed'.

We use a simple type system to rule out ill-formed programs. Typing judgments have the form $\vdash P : e$; intuitively meaning that the program P produces values of type e.

$$\frac{}{\vdash \langle\rangle : 1} \qquad\qquad \frac{}{\vdash [] : e^*} \qquad\qquad \frac{\vdash P : e \qquad \vdash Q : e^*}{\vdash P :: Q \ : e^*}$$

$$\frac{\vdash P_1 : e_1 \quad \dots \quad \vdash P_n : e_n}{\vdash v(P_1, \dots, P_n) : f}\ \pi(v) = e_1, \dots, e_n \vdash f$$

Every program has at most one typing derivation (relatively to the fixed proof π), which can be computed in linear time. This argument is easily formalisable in RCA_0.

We associate to every program P of type e a semantic value $[P] \in [e]$, by induction:

$$[\langle\rangle] \triangleq \langle\rangle \qquad [[]] \triangleq \epsilon \qquad [P{::}Q] \triangleq [P] :: [Q] \qquad [v(P_1, \dots, P_n)] \triangleq [v]([P_1], \dots, [P_n])$$

Note that in the last case, $[v]$ is the semantics of the node v in the proof π (Definition 4.6). This semantics cannot be defined ACA_0 or RCA_0: values may be objects of arbitrary type.

Definition 4.12 (Reduction)**.** *Reduction*, written $\leadsto$, is the smallest relation on programs which is closed under all contexts and satisfies the following rules, defined by case analysis on the rules used at addresses mentioned in the program. We use v (resp. w) to range over addresses of left (resp. right) introduction rules, and u to range over other addresses. We moreover assume that the sizes of the vectors match those that arise from the implicit typing derivations.

$$\begin{array}{lrcl}
id: & u(P) & \leadsto & P \\
cut: & u(\vec{P}, \vec{Q}) & \leadsto & u1(u0(\vec{P}), \vec{Q})
\end{array}
\qquad
\begin{array}{lrcl}
x: & u(\vec{P}, Q, R, \vec{S}) & \leadsto & u0(\vec{P}, R, Q, \vec{S}) \\
w: & u(P, \vec{R}) & \leadsto & u0(\vec{R}) \\
c: & u(P, \vec{Q}) & \leadsto & u0(P, P, \vec{Q})
\end{array}$$

$$\begin{array}{lrcl}
1\text{-}l: & v(\langle\rangle, \vec{R}) & \leadsto & v0(\vec{R}) \\
*\text{-}l: & v([], \vec{R}) & \leadsto & v0(\vec{R}) \\
*\text{-}l: & v(P{::}Q, \vec{R}) & \leadsto & v1(P, Q, \vec{R})
\end{array}
\qquad
\begin{array}{lrcl}
1\text{-}r: & w() & \leadsto & \langle\rangle \\
*\text{-}r_\epsilon: & w() & \leadsto & [] \\
*\text{-}r_{::}: & w(\vec{P}, \vec{Q}) & \leadsto & w0(\vec{P}){::}w1(\vec{Q})
\end{array}$$

$$\begin{array}{lrcl}
\cdot\text{-}l/\cdot\text{-}r: & v(w(\vec{P}, \vec{Q}), \vec{R}) & \leadsto & v0(w0(\vec{P}), w1(\vec{Q}), \vec{R}) \\
+\text{-}l/+\text{-}r_i: & v(w(\vec{P}), \vec{R}) & \leadsto & vi(w0(\vec{P}), \vec{R}) \\
\to\text{-}l/\to\text{-}r: & v(w(\vec{P}), \vec{Q}, \vec{R}) & \leadsto & v1(w0(v0(\vec{Q}), \vec{P}), \vec{R}) \\
\cap\text{-}l_i/\cap\text{-}r: & v(w(\vec{P}), \vec{R}) & \leadsto & v0(wi(\vec{P}), \vec{R})
\end{array}$$

We used a compact presentation of these rules; for instance, the $\cdot$-$l/\cdot$-r rule should be understood as follows:

If π_w ends ${}_{\cdot\text{-}r}\dfrac{E \vdash e \quad F \vdash f}{E, F \vdash e \cdot f}$, π_v ends ${}_{\cdot\text{-}l}\dfrac{e, f, G \vdash g}{e \cdot f, G \vdash g}$, and $|E| = |\vec{P}|$,
then $v(w(\vec{P}, \vec{Q}), \vec{R}) \leadsto v0(w0(\vec{P}), v1(\vec{Q}), \vec{R})$.

As expected, subject reduction holds, so that we only work with well-typed programs in the sequel. Also note that $\leadsto$ is computable in RCA_0, and so is provably Δ^0_1. We also have the following characterization of irreducible programs, still in RCA_0

Lemma 21. *If P is irreducible, then P is of the form*

- $\langle\rangle$, $[]$, *or* $P_1 :: P_2$ *for some programs* P_1, P_2*; or,*
- $v(\vec{P})$ *for some address v such that π_v ends with $+$-r_i, $\cdot$-r, $\cap$-r or $\to$-r.*

Proof. The characterization given in the statement is computable, and so we may prove the lemma by Σ^0_1-induction on the structure of programs. If P starts with a constructor, we are done; otherwise, if $P = v[\vec{P}]$ then v cannot be a structural rule, the identity rule, or the cut rule, otherwise P would reduce. If v is a left introduction rule then by induction P_1 (which is irreducible) must be a constructor or of the form $w[\vec{Q}]$ with w a right introduction rule, thus enabling a reduction step for P, a contradiction. □

It follows that every irreducible program of type e^* is a list of irreducible programs of type e.

We also have that reductions preserve the semantics. We use this property only at the meta-level, it cannot even be stated in ACA_0 since it involves higher-order objects:

Proposition 4.6 (Semantic preservation)**.** For all programs P, P', if $P \rightsquigarrow P'$ then $[P] = [P']$.

Given a natural number n, let us write $\underline{n}$ for its encoding as a program of type 1^*, such that $[\underline{n}] = n$. By Lemma 21, the irreducible programs of type 1^* are all of this shape. This encoding makes it possible to reason about proofs from natural numbers to natural numbers: if $\pi : 1^* \vdash 1^*$, then for all n, $[\pi](n)$ can be obtained by reducing the program $\pi(\underline{n})$. (Writing $\pi(\vec{P})$ for $\epsilon(\vec{P})$.)

4.6.1 Weak normalization in ACA_0, in the general case

We write $P \downarrow_\pi P'$ when P reduces to an irreducible P' via the left-most innermost strategy. We want to show:

Theorem 4.4 (Weak normalisation)**.** *For every fixed regular proof π, ACA_0 proves that for all P, there exists P' with $P \downarrow_\pi P'$.*

Note that π is fixed, and that the universal quantification on P only ranges over those computations that can be performed within π. Since π is regular, those programs involve only finitely many types, and the statement we prove inside ACA_0 does not imply consistency of Peano arithmetic.

To prove this theorem, we use the following sets R_e of *reducible programs*, defined by induction on e. Those are inspired by reducibility candidates [82, 43].

$$\begin{aligned}
\mathsf{R}_1 &\triangleq \{P \mid P \downarrow_\pi \langle\rangle\}\\
\mathsf{R}_{e^*} &\triangleq \{P \mid P \downarrow_\pi Q_1 :: \cdots :: Q_n,\ \text{with } Q_1, \ldots, Q_n \in \mathsf{R}_e\}\\
\mathsf{R}_{e\cdot f} &\triangleq \left\{P \mid P \downarrow_\pi v(\vec{Q}, \vec{R}),\ \text{with } v \text{ a } \cdot\text{-}r \text{ step},\ v0(\vec{Q}) \in \mathsf{R}_e,\ \text{and } v1(\vec{R}) \in \mathsf{R}_f\right\}\\
\mathsf{R}_{e\cap f} &\triangleq \left\{P \mid P \downarrow_\pi v(\vec{Q}),\ \text{with } v \text{ a } \cap\text{-}r \text{ step},\ v0(\vec{Q}) \in \mathsf{R}_e,\ \text{and}\ \ v1(\vec{Q}) \in \mathsf{R}_f\right\}\\
\mathsf{R}_{e_0+e_1} &\triangleq \left\{P \mid P \downarrow_\pi v(\vec{Q}),\ \text{with } v \text{ a } +\text{-}r_i \text{ step and}\ \ vi(\vec{Q}) \in \mathsf{R}_{e_i}\right\}\\
\mathsf{R}_{e\to f} &\triangleq \left\{P \mid P \downarrow_\pi v(\vec{Q}),\ \text{with } v \text{ a } \to\text{-}r \text{ step and}\ \ \forall Q \in \mathsf{R}_e, v0(Q, \vec{Q}) \in \mathsf{R}_f\right\}
\end{aligned}$$

(Like earlier in the chapter, in the third case, we assume that the lengths of the vectors are consistent with the rule instances used at v.)

Note that these sets are defined non-uniformly in ACA_0: we use separate instances of comprehension at each stage. This is not a problem: we will need only finitely many of them since the starting proof π is regular.

Every program in R_e is weakly normalisable by definition, so that it suffices to show that all programs of type e belong to R_e. We proceed by induction on the syntax of programs. The constructor cases are straightforward; for the remaining case we use the following proposition. If $\vec{P} = P_1, \ldots, P_n$ and $E = E_1, \ldots, E_n$, we write $\vec{P} \in \mathsf{R}_E$ when $P_i \in \mathsf{R}_{E_i}$ for all i.

Proposition 4.7. For every address $w : E \vdash e$, and for all programs $\vec{P} \in \mathsf{R}_E$, we have $w(\vec{P}) \in \mathsf{R}_e$.

This property on addresses is locally preserved by the rules of C. This observation is not sufficient to conclude since we work with non-wellfounded proofs. We actually prove a strengthening of local preservation, by contraposite:

Lemma 22. *For every address $w : E \vdash e$, for all programs $\vec{P} \in \mathsf{R}_E$ such that $w(\vec{P}) \notin \mathsf{R}_e$, there are $v, F, f, \vec{Q}$ such that $|v| = |w| + 1$, $v : F \vdash f$, $v(\vec{Q}) \notin \mathsf{R}_f$, and:*

1. *for all i, j s.t. $\langle v, i \rangle \lhd \langle w, j \rangle$, we have $|Q_i| = |P_j|$, and*
2. *for all i, j s.t. $\langle v, i \rangle \mathrel{\dot{\lhd}} \langle w, j \rangle$, we have $|Q_i| < |P_j|$.*

(Where given $P \in \mathsf{R}_{e^*}$, we write $|P|$ for the length of the list given by the definition of R_{e^*}.)

Proof. We abbreviate $P \downarrow_\pi P'$ as $P \downarrow P'$ in this proof, and we often use the fact that if $P \in \mathsf{R}_e$, then $P \downarrow P'$ for some $P' \in \mathsf{R}_e$, which we abbreviate as $P \downarrow P' \in \mathsf{R}_e$. We also write $P \in \mathsf{R}^{\downarrow}_e$ when $P \in \mathsf{R}_e$ and P is irreducible. We use the notation $\rightsquigarrow$ only for left-most innermost reduction steps.

We can assume w.l.o.g. that the elements of $\vec{P}$ are irreducible. We reason by case analysis on the rule used at w; we only list the most significant cases. We call the vector $\vec{Q}$ we have to provide the *witness.*

cut : π_w ends ${\scriptstyle cut}\,\dfrac{E \vdash e \quad e, F \vdash f}{E, F \vdash f}$. Assume $\vec{P} \in \mathsf{R}^{\downarrow}_E$, $\vec{Q} \in \mathsf{R}^{\downarrow}_F$ and $w(\vec{P}, \vec{Q}) \notin \mathsf{R}_f$. There are two cases:

- if $w0(\vec{P}) \notin \mathsf{R}_e$ then we choose $v = w0$, taking $\vec{P}$ as witness.
- if $w0(\vec{P}) \in \mathsf{R}_e$ then we choose $v = w1$, taking $w0(\vec{P}), \vec{Q}$ as witness since

$$w(\vec{P}, \vec{Q}) \rightsquigarrow w1(w0(\vec{P}), \vec{Q})$$

c : π_w ends ${\scriptstyle c}\,\dfrac{e, E \vdash g}{e, e, E \vdash g}$. Assuming $P \in \mathsf{R}^{\downarrow}_e$, $\vec{P} \in \mathsf{R}^{\downarrow}_E$, we take $v = w0$ with witness $P, P, \vec{P}$, since

$$w(P, \vec{P}) \rightsquigarrow w0(P, P, \vec{P})$$

$\to$-r : π_w ends $\to\text{-}r\,\dfrac{e, E \vdash f}{E \vdash e \to f}$. Assume $\vec{P} \in \mathsf{R}^{\downarrow}_{E}$ and $w(\vec{P}) \notin \mathsf{R}_{e\to f}$. $w(\vec{P})$ is irreducible, so that there must be a $R \in \mathsf{R}_e$ such that $w0(R, \vec{P}) \notin \mathsf{R}_f$. We choose $v = w0$ with $R, \vec{P}$ as witness.

$\to$-l : π_w ends $\to\text{-}l\,\dfrac{E \vdash e \quad f, F \vdash g}{e \to f, E, F \vdash g}$. Assume $P = u[\vec{R}] \in \mathsf{R}^{\downarrow}_{e\to f}$, $\vec{P} \in \mathsf{R}^{\downarrow}_{E}$, $\vec{Q} \in \mathsf{R}^{\downarrow}_{F}$ and $w(P, \vec{P}, \vec{Q}) \notin \mathsf{R}_g$. There are two cases:

- if $w0(\vec{P}) \notin \mathsf{R}_e$, we take $v = w0$ with witness $\vec{P}$.
- if $w0(\vec{P}) \in \mathsf{R}_e$, then $w0(\vec{P}) \downarrow P_0 \in \mathsf{R}_e$. By definition of $\mathsf{R}_{e\to f}$ we obtain $u0(P_0, \vec{R}) \in \mathsf{R}_f$. We choose $v = w1$, taking $u0(P_0, \vec{R}), \vec{Q}$ as witness, since

$$\begin{aligned} w(P, \vec{P}, \vec{Q}) &\rightsquigarrow w1(u0(w0[\vec{P}], \vec{R}), \vec{Q}) \\ &\rightsquigarrow^* w1(u0(P_0, \vec{R}), \vec{Q}) \end{aligned}$$

$*$-$r_{::}$: π_w ends $*\text{-}r_{::}\,\dfrac{E \vdash e \quad F \vdash e^*}{E, F \vdash e^*}$. Assume $\vec{P} \in \mathsf{R}^{\downarrow}_{E}$, $\vec{Q} \in \mathsf{R}^{\downarrow}_{F}$, and $w(\vec{P}, \vec{Q}) \notin \mathsf{R}_{e^*}$. If $w0(\vec{P}) \notin \mathsf{R}_e$ we take $v = w0$ with $\vec{P}$ as witness. Otherwise $w0(\vec{P}) \downarrow R_0 \in \mathsf{R}_e$ and we take $v = w1$ with $\vec{Q}$ as witness. Indeed, if we had $w1(\vec{Q}) \in \mathsf{R}_{e^*}$ then we would get $w1(\vec{Q}) \downarrow R_1 :: \cdots :: R_n$ with the R_i in R_e; this would contradict the assumption about w since

$$\begin{aligned} w(\vec{P}, \vec{Q}) &\rightsquigarrow w0(\vec{P}) :: w1(\vec{Q}) \\ &\rightsquigarrow^* R_0 :: w1(\vec{Q}) \\ &\rightsquigarrow^* R_0 :: R_1 :: \cdots :: R_n \end{aligned}$$

$*$-l : π_w ends $*\text{-}l\,\dfrac{E \vdash g \quad e, e^*, E \vdash g}{e^*, E \vdash g}$. Assume $P \in \mathsf{R}^{\downarrow}_{e^*}$, $\vec{P} \in \mathsf{R}^{\downarrow}_{E}$ and $w(P, \vec{P}) \notin \mathsf{R}_g$. According to the definition of R_{e^*} we can distinguish two cases:

- if $P = []$, we take $v = w0$ with $\vec{P}$ as witness:

$$w(P, \vec{P}) \rightsquigarrow w0(\vec{P})$$

- or $P = X :: Q$, and we take $v = w1$ with $X, Q, \vec{P}$ as witness:

$$w(P, \vec{P}) \rightsquigarrow w1(X, Q, \vec{P})$$

  We have $|P| = |Q| + 1$ in this case, so that we satisfy the condition 2/ for $i = 1$ and $j = 0$. (this is the only place where this condition is not void)

The condition 1/ is straightforward to check in all cases. □

Proof of Proposition 4.7. Suppose by contradiction that for some address $w : E \vdash e$ we have $\vec{P} \in \mathsf{R}_E$ such that $w(\vec{P}) \notin \mathsf{R}_e$. By using Lemma 22 repeatedly, we can construct an infinite branch of π starting at w. We conclude like in Lemma 14. □

This concludes the ACA_0 proof of Theorem 4.4 and we deduce:

Corollary 1. *If $\pi : 1^* \dots 1^* \vdash 1^*$ is a regular proof, then there exists a term M from system T such that $[\pi] = [M]$.*

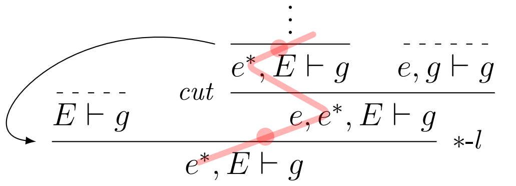
Proof (for unary functions). By Proposition 4.5 and Theorem 4.4 we obtain a proof in ACA_0 of "$\forall n, \exists m, \pi(\underline{n}) \downarrow_\pi \underline{m}$". By Proposition 4.3, we extract a system T term M such that for all n, $\pi(\underline{n}) \leadsto^* \underline{[M](n)}$. By Proposition 4.6, we deduce for all n, $[\pi](n) = [\pi(\underline{n})] = [\underline{[M](n)}] = [M](n)$. □

4.6.2 Weak normalization in RCA_0, in the affine case

Given Proposition 4.4, it could be tempting to revisit the proof from the previous section, trying to see if we could use RCA_0 instead of ACA_0 in the absence of contraction. This fails, however, because the R_e sets already require set comprehensions outside Δ^0_1 (due to the quantifier alternation in the definition of $\mathsf{R}_{e\to f}$). We need only finitely many such sets for a given regular proof, so that we could hope to use only their defining formulas, but then our main induction on the syntax of programs, to prove that all programs of type e belong to R_e, is not a Σ^0_1-induction.

Instead, we use another termination argument, relying on the translation from Section 4.4.

Definition 4.13. A *simple proof* is an ibp-proof such that for every backpointer pt, $src(pt) = tgt(pt)10$ and the rule used at $tgt(pt)1$ is a *cut*, as illustrated on the left below.



$$\frac{E \vdash g \quad e, g \vdash g}{e^*, E \vdash g}\ *\text{-}l'$$

In other words, a simple proof is a well-founded proof using the derivable rule on the right.

Our translation from T to C (Theorem 4.2) actually produces simple proofs, so that by Theorem 4.3, every affine proof can be translated into a simple affine proof with the same semantics.

Accordingly, we assume in the rest of this section that the fixed proof π is affine and simple. This assumption makes it possible to optimize the notion of reduction: we write $\dot{\leadsto}$ for the relation defined like in Definition 4.12, except that when v is the target of a backpointer, we use the following rule instead of the two $*$-l reduction rules:

$$v(P_1{::}\dots{::}P_n{::}[], \vec{R}) \mathrel{\dot{\leadsto}} v11(P_1, \dots, v11(P_n, v0[\vec{R}]))$$

This rule has to be compared with the $2n+1$ reductions we can obtain with $\leadsto$:

$$\begin{aligned}
v(P_1{::}\dots{::}P_n{::}[], \vec{R}) &\leadsto v1(P_1, P_2{::}\dots{::}P_n{::}[], \vec{R})\\
&\leadsto v11(P_1, v10(P_2{::}\dots{::}P_n{::}[], \vec{R}))\\
&\dots\\
&\leadsto v11(P_1, \dots, v(10)^n 11(P_n, v(10)^n([], \vec{R})))\\
&\leadsto v11(P_1, \dots, v(10)^n 11(P_n, v(10)^n 0(\vec{R})))
\end{aligned}$$

Due to the backpointer from $v01$ to v, we have $\pi_{v(01)^n} = \pi_v$, so that the semantics is preserved. The main advantage of $\dot{\rightsquigarrow}$ is that when $P \dot{\rightsquigarrow} P'$, if P contains only canonical addresses, then so does P'.

Lemma 23. *If there is an infinite leftmost innermost reduction sequence along $\rightsquigarrow$, then there is an infinite reduction sequence along $\dot{\rightsquigarrow}$ where programs only contain canonical addresses.*

Proof. By mapping addresses into their canonical addresses and compressing finite sequences of reductions as above. □

We assume all programs only mention canonical addresses in the sequel. Let $m(P)$ be the finite multiset of (canonical) addresses mentioned in a program P. These multisets can be represented and computed in RCA_0 via appropriate encodings; we write $m(u)$ for the number of occurrences of an address u in a multiset m. We write $\succeq$ for the multiset ordering, where addresses are ordered by reverse prefix ordering (i.e., longer addresses are considered as smaller):

$$m \succeq m' \triangleq \forall v, m(v) \geq m'(v) \vee \exists u, u \sqsubseteq v, m(u) > m'(u)$$

Lemma 24. *If $P \dot{\rightsquigarrow} P'$ then $m(P) \succ m(P')$.*

Proof. By straightforward analysis of the reduction rules. (Note that the reduction rule for contraction fails this property because it duplicates arbitrary addresses.) □

At the meta-level, these two lemmas suffice to conclude that every leftmost innermost reduction sequence along $\rightsquigarrow$ terminates: since we have finitely many canonical addresses in π, the reverse prefix ordering on canonical addresses is well-founded, as well as the above multiset ordering. This latter result cannot be proved uniformly in RCA_0, however [78, Theorem IX.5.4]. Instead, we use the folklore fact that the multiset order on a fixed and finite order is provably well-founded in RCA_0:

Proposition 4.8. For all $n \in \mathbb{N}$, RCA_0 proves that the multiset order on $[\![0, n]\!]$ is well-founded.

Proof. This is part of [78, Theorem IX.5.4], where the corresponding proof is mentioned as straightforward. We give an explicit proof in [62, Appendix D.3]. □

That we restrict to the multiset order on a finite and total order in the above statement is not a restriction since every finite partial order—like our reverse prefix ordering on canonical addresses—embeds in a finite total order.

Theorem 4.5 (Weak normalisation)**.** *For every fixed affine simple proof π, RCA_0 proves that for all P, there exists P' with $P \downarrow_\pi P'$.*

Proof. Write P_n for the n-th reduct of P via the leftmost innermost strategy (if any). It suffices to show that there exists n such that P_n is irreducible. Suppose by contradiction that for all n, P_n can be reduced, i.e., $P_n \rightsquigarrow P_{n+1}$ since we fixed a strategy. By Lemma 23 and Lemma 24, we find an infinite decreasing sequence of multisets over $[\![0, k]\!]$ where k is the maximal length of canonical addresses in π, contradicting Proposition 4.8. □

Corollary 2. *If $x_1{:}1^* \dots x_n{:}1^* \vdash M : 1^*$ is an affine term of T, then $[M]$ is primitive recursive.*

Proof. Translate M into a simple affine proof using Theorem 4.2. Then proceed like for Corollary 1, using Theorem 4.5 and Proposition 4.4 instead of Theorem 4.4 and Proposition 4.3. Note that Proposition 4.5 is not required here since simple proofs do not need any validity criterion. □

This corollary generalizes Dal Lago's upper bound for $\mathcal{H}(\emptyset)$ [23]: our proof handles additive pairs, which we do not know how to handle using Dal Lago's method. Also note that the cyclic proof machinery is not required to obtain this corollary: we use the easy translation from T into C (Theorem 4.3) in order to obtain a small steps semantics which is convenient to work with, but this translation only produces simple proofs, which can be presented inductively, as finite trees.

Instead, the following corollary about affine C requires the machinery from Section 4.4 to delineate the cycle structure of affine proofs. We do not know of a more direct approach so far.

Corollary 3. *If $\pi : 1^* \dots 1^* \vdash 1^*$ is an affine regular proof, then $[\pi]$ is primitive recursive.*

Proof. Translate π into an affine term using Theorem 4.3 and conclude with Corollary 2. □

We also recover as a corollary the following known fact [78, Theorem IX.5.4]:

Corollary 4. RCA_0 *cannot prove that the multiset order on $\mathbb{N}$ is well-founded.*

Proof. If this was a theorem of RCA_0, then we would get a uniform proof of Theorem 4.5, from which we could extract a 'universal primitive recursive function' whose complexity would bound the complexity of all primitive recursive functions (via Theorem 4.1). □

4.7 Conclusion and future work

We proposed the cyclic sequent proof system C, which we equipped with both a denotational semantics (Definition 4.7), and a small steps operational semantics (Definition 4.12). Under this interpretation, regular proofs of system C can be seen as functional programs with unstructured recursion (gotos), whose termination is guaranteed by a decidable validity criterion: an instance of the Size Change Principle [64].

We studied the expressive power of system C as a programming language, by comparing it with an appropriate version of Gödel's system T—a structured programming language. Encoding cyclic programs into structurally recursive ones is nontrivial, but we managed to give a direct encoding from C to T in the affine case. To obtain upper bounds on the complexity of functions of C and its affine variant we then appealed to proofs of totality in systems of second-order arithmetic, thus obtaining simulations in T and primitive recursive arithmetic, respectively.

We used the connectives of IMALL plus a least fixpoint operator for lists to illustrate the genericity of our approach. Small fragments of C are already complete w.r.t. the considered classes of functions (e.g., 1^* and $\cdot$ do suffice to capture primitive recursive functions). Conversely, other least fixpoint operators could be handled (e.g., $\mu x.e + x \cdot x$ for binary trees with leaves in e). Cyclic systems with both least and greatest fixpoints have been studied [39, 31]; whether they correspond to appropriate extensions of T is left for future work.

Our current translation of C with contraction into T works for natural number functions, but it does not scale directly to higher types. Indeed, the technique we use (usual reducibility and hereditary recursivity arguments to obtain a proof of totality in ACA_0) is restricted to computations returning finite values. It would thus be interesting to attain a 'direct' translation in the style of the one we obtained for the affine case in Section 4.4. As explained in Remark 4.4, higher types do not seem to be problematic *per se*, but we need a better understanding of the structure of threads with contractions on star formulas.

The type levels of recursors in T programs are closely related to the logical complexity of induction in Peano Arithmetic (in the sense of Definition 4.10). At this level of granularity, it was observed recently in [25] that there is indeed a difference between cyclic and inductive proofs: cyclic proofs using Σ_n formulas is equivalent to inductive proofs using Σ_{n+1} formulas (over Π_{n+1} theorems). It would be natural to expect, therefore, that C restricted to level n types is equivalent to T restricted to level $n+1$ recursors (over level $n+1$ functions). This would be consistent with the fact that we do have an implementation of Ackermann-Péter's function in C at level 0 (Figure 4.7), but that remains a topic for future work.

Bibliography

[1] P. A. Abdulla, Y. Chen, L. Clemente, L. Holík, C. Hong, R. Mayr, and T. Vojnar. Simulation subsumption in ramsey-based Büchi automata universality and inclusion testing. In *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 132–147. Springer, 2010. doi:10.1007/978-3-642-14295-6_14.

[2] P. A. Abdulla, Y. Chen, L. Clemente, L. Holík, C. Hong, R. Mayr, and T. Vojnar. Advanced ramsey-based Büchi automata inclusion testing. In *CONCUR*, volume 6901 of *Lecture Notes in Computer Science*, pages 187–202. Springer, 2011. doi:10.1007/978-3-642-23217-6_13.

[3] P. A. Abdulla, Y.-F. Chen, L. Holík, R. Mayr, and T. Vojnar. When simulation meets antichains. In *TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 158–174. Springer, 2010. doi:10.1007/978-3-642-12002-2_14.

[4] S. Abramsky, E. Haghverdi, and P. Scott. Geometry of interaction and linear combinatory algebras. *Mathematical Structures in Computer Science*, 12(5):625–665, 2002. doi:10.1017/S0960129502003730.

[5] B. Afshari and G. E. Leigh. Cut-free completeness for modal mu-calculus. In *LiCS*, pages 1–12, 2017. doi:10.1109/LICS.2017.8005088.

[6] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987. doi:10.1016/0890-5401(87)90052-6.

[7] J. Avigad. Formalizing forcing arguments in subsystems of second-order arithmetic. *Annals of Pure and Applied Logic*, 82(2):165 – 191, 1996. URL: http://www.sciencedirect.com/science/article/pii/0168007296000036, doi:https://doi.org/10.1016/0168-0072(96)00003-6.

[8] J. Avigad and S. Feferman. Gödel's functional interpretation. In S. R. Buss, editor, *Handbook of Proof Theory*. Elsevier, 1998.

[9] C. Baier and J. Katoen. *Principles of model checking*. MIT Press, 2008.

[10] P. Baillot and M. Pedicini. Elementary complexity and geometry of interaction. *Fundamenta Informaticae*, 45(1-2):1–31, 2001.

[11] S. Berardi and M. Tatsuta. Classical system of martin-löf's inductive definitions is not equivalent to cyclic proof system. In *FoSSaCS*, pages 301–317, 2017. doi:10.1007/978-3-662-54458-7_18.

[12] S. Berardi and M. Tatsuta. Equivalence of inductive definitions and cyclic proofs under arithmetic. In *LiCS*, pages 1–12, 2017. doi:10.1109/LICS.2017.8005114.

[13] B. Bollig, P. Habermehl, C. Kern, and M. Leucker. Angluin-style learning of NFA. In C. Boutilier, editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 1004–1009, 2009. URL: http://ijcai.org/Proceedings/09/Papers/170.pdf.

[14] F. Bonchi and D. Pous. Checking NFA equivalence with bisimulations up to congruence. In *POPL*, pages 457–468. ACM, 2013. doi:10.1145/2429069.2429124.

[15] G. Boolos and R. C. Jeffrey. *Computability and logic (2. ed.).* Cambridge University Press, 1987.

[16] J. Brotherston. Cyclic proofs for first-order logic with inductive definitions. In *TABLEAUX*, volume 3702 of *Lecture Notes in Artificial Intelligence*, pages 78–92. Springer, 2005. doi:10.1007/11554554_8.

[17] J. Brotherston and A. Simpson. Sequent calculi for induction and infinite descent. *Journal of Logic and Computation*, 21(6):1177–1216, 2011. doi:10.1093/logcom/exq052.

[18] J. R. Büchi. On a decision method in restricted second order arithmetic. In S. Mac Lane and D. Siefkes, editors, *The Collected Works of J. Richard Büchi*, pages 425–435. Springer New York, New York, NY, 1990.

[19] S. R. Buss. The witness function method and provably recursive functions of Peano arithmetic. In *Studies in Logic and the Foundations of Mathematics*, volume 134, pages 29–68. Elsevier, 1995. doi:10.1016/S0049-237X(06)80038-8.

[20] S. R. Buss. *Handbook of proof theory*, volume 137. Elsevier, 1998.

[21] H. Calbrix, M. Nivat, and A. Podelski. Ultimately periodic words of rational w-languages. In *MFPS*, volume 802 of *Lecture Notes in Computer Science*, pages 554–566. Springer, 1993. doi:10.1007/3-540-58027-1_27.

[22] J. H. Conway. *Regular algebra and finite machines.* Chapman and Hall, 1971.

[23] U. Dal Lago. The geometry of linear higher-order recursion. *ACM Trans. Comput. Log.*, 10(2):8:1–8:38, 2009. doi:10.1145/1462179.1462180.

[24] L. D'Antoni, Z. Kincaid, and F. Wang. A symbolic decision procedure for symbolic alternating finite automata. In *Proceedings of the Thirty-Fourth Conference on the Mathematical Foundations of Programming Semantics, MFPS 2018, Dalhousie University, Halifax, Canada, June 6-9, 2018*, pages 79–99, 2018. doi:10.1016/j.entcs.2018.03.017.

[25] A. Das. On the logical complexity of cyclic arithmetic. *Logical Methods in Computer Science*, 16(1), Jan. 2020. doi:10.23638/LMCS-16(1:1)2020.

[26] A. Das, A. Doumane, and D. Pous. Left-handed completeness for Kleene algebra, via cyclic proofs. In *LPAR*, volume 57 of *EPiC Series in Computing*, pages 271–289. Easychair, 2018. doi:10.29007/hzq3.

[27] A. Das and D. Pous. A cut-free cyclic proof system for Kleene algebra. In *TABLEAUX*, volume 10501 of *Lecture Notes in Computer Science*, pages 261–277. Springer, 2017. doi:10.1007/978-3-319-66902-1_16.

[28] A. Das and D. Pous. Non-wellfounded proof theory for (Kleene+action)(algebras+lattices). In *CSL*, volume 119 of *LIPIcs*, pages 18:1–18:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.CSL.2018.19.

[29] C. Dax, M. Hofmann, and M. Lange. A proof system for the linear time μ-calculus. In *FSTTCS*, volume 4337 of *Lecture Notes in Computer Science*, pages 273–284. Springer, 2006. doi:10.1007/11944836_26.

[30] A. Doumane. Constructive completeness for the linear-time μ-calculus. In *LiCS*, pages 1–12, 2017. doi:10.1109/LICS.2017.8005075.

[31] A. Doumane, D. Baelde, and A. Saurin. Infinitary proof theory: the multiplicative additive case. In *CSL*, volume 62 of *LIPIcs*, pages 42:1–42:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Sept. 2016. doi:10.4230/LIPIcs.CSL.2016.42.

[32] W. F. Dowling and J. H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *J. Log. Program.*, 1(3):267–284, 1984. doi:10.1016/0743-1066(84)90014-1.

[33] L. Doyen and J. Raskin. Improved algorithms for the automata-based approach to model-checking. In *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 451–465. Springer, 2007. doi:10.1007/978-3-540-71209-1_34.

[34] L. Doyen and J. Raskin. Antichains for the automata-based approach to model-checking. *Logical Methods in Computer Science*, 5(1), 2009. doi:10.2168/LMCS-5(1:5)2009.

[35] L. Doyen and J.-F. Raskin. Antichain Algorithms for Finite Automata. In *TACAS*, volume 6015 of *Lecture Notes in Computer Science*. Springer, 2010. doi:10.1007/978-3-642-12002-2_2.

[36] N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. *Electron. Notes Theor. Comput. Sci.*, 89(4):543–560, 2003. doi:10.1016/S1571-0661(05)82542-3.

[37] S. Fogarty and M. Y. Vardi. Büchi complementation and size-change termination. In *TACAS*, volume 5505 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 2009. doi:10.1007/978-3-642-00768-2_2.

[38] S. Fogarty and M. Y. Vardi. Efficient Büchi universality checking. In *TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 205–220. Springer, 2010. doi:10.1007/978-3-642-12002-2_17.

[39] J. Fortier and L. Santocanale. Cuts for circular proofs: semantics and cut-elimination. In *CSL*, volume 23 of *LIPIcs*, pages 248–262, 2013. doi:10.4230/LIPIcs.CSL.2013.248.

[40] A. Frisch and L. Cardelli. Greedy regular expression matching. In *ICALP*, volume 3142 of *Lecture Notes in Computer Science*, pages 618–629. Springer, 2004. doi:10.1007/978-3-540-27836-8_53.

[41] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *CAV*, pages 53–65. Springer, 2001.

[42] J.-Y. Girard. Geometry of interaction iii: accommodating the additives. In *workshop on Advances in linear logic*, pages 329–389. Cambridge University Press, 1995.

[43] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*. Cambridge University Press, USA, 1989.

[44] S. Gurumurthy, O. Kupferman, F. Somenzi, and M. Y. Vardi. On complementing nondeterministic Büchi automata. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 96–110. Springer, 2003.

[45] V. K. Gödel. Über eine bisher noch nicht benütze erweiterung des finiten standpunktes. *Dialectica*, 12(3-4):280–287, 1958. doi:10.1111/j.1746-8361.1958.tb01464.x.

[46] F. Henglein and L. Nielsen. Regular expression containment: coinductive axiomatization and computational interpretation. In *POPL, 2011*, pages 385–398. ACM, 2011. doi:10.1145/1926385.1926429.

[47] D. R. Hirschfeldt. *Slicing the truth: On the computable and reverse mathematics of combinatorial principles*. World Scientific, 2014.

[48] M. Holzer, M. Kutrib, and A. Malcher. Multi-head finite automata: Characterizations, concepts and open problems. In *International Workshop on The Complexity of Simple Programs (CSP)*, pages 93–107, 2008. doi:10.4204/EPTCS.1.9.

[49] G. J. Holzmann. The model checker SPIN. *IEEE Trans. on soft. eng.*, 23(5):279–295, 1997.

[50] J. N. Hooker. Solving the incremental satisfiability problem. *J. Log. Program.*, 15(1&2):177–186, 1993. doi:10.1016/0743-1066(93)90018-C.

[51] J. E. Hopcroft. An n log n algorithm for minimizing in a finite automaton. In *International Symposium of Theory of Machines and Computations*, pages 189–196. Academic Press, NY, USA, 1971.

[52] J. E. Hopcroft and R. M. Karp. A linear algorithm for testing equivalence of finite automata. Technical Report 114, Cornell Univ., December 1971. URL: http://techreports.library.cornell.edu:8081/Dienst/UI/1.0/Display/cul.cs/TR71-114.

[53] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition.* Pearson international edition. Addison-Wesley, 2007.

[54] N. Hoshino, K. Muroya, and I. Hasuo. Memoryful geometry of interaction: from coalgebraic components to algebraic effects. In *CSL-LICS*, page 52. ACM, 2014. doi:10.1145/2603088.2603124.

[55] M. Hutagalung, M. Lange, and E. Lozes. Revealing vs. concealing: More simulation games for Büchi inclusion. In *LATA*, pages 347–358. Springer, 2013.

[56] B. Knaster. Un théorème sur les fonctions d'ensembles. *Annales de la Société Polonaise de Mathématiques*, 6:133–134, 1928.

[57] L. Kołodziejczyk, H. Michalewski, P. Pradic, and M. Skrzypczak. The logical strength of Büchi's decidability theorem. *Logical Methods in Computer Science*, 15(2), May 2019. doi:10.23638/LMCS-15(2:16)2019.

[58] D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, 1994. doi:10.1006/inco.1994.1037.

[59] D. Kuperberg, L. Pinault, and D. Pous. Coinductive algorithms for büchi automata. In P. Hofman and M. Skrzypczak, editors, *DLT*, volume 11647 of *Lecture Notes in Computer Science*, pages 206–220. Springer, 2019. doi:10.1007/978-3-030-24886-4_15.

[60] D. Kuperberg, L. Pinault, and D. Pous. Cyclic proofs and jumping automata, 2019. doi:10.4230/LIPIcs.FSTTCS.2019.44.

[61] D. Kuperberg, L. Pinault, and D. Pous. Cyclic proofs, system t, and the power of contraction. *Proc. ACM Program. Lang.*, 5(POPL), 2021. doi:10.1145/3434282.

[62] D. Kuperberg, L. Pinault, and D. Pous. Cyclic proofs, system T, and the power of contraction – extended version, with appendices, 2021. URL: https://hal.archives-ouvertes.fr/hal-02487175/document.

[63] O. Kupferman and M. Y. Vardi. Weak alternating automata are not that weak. *ACM Trans. Comput. Log.*, 2(3):408–429, 2001. doi:10.1145/377978.377993.

[64] C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In C. Hankin and D. Schmidt, editors, *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, pages 81–92. ACM, 2001. doi:10.1145/360204.360210.

[65] H. R. Lewis and C. H. Papadimitriou. *Elements of the theory of computation, 2nd Edition.* Prentice Hall, 1998.

[66] D. Lucanu, E. Goriac, G. Caltais, and G. Rosu. CIRC: A behavioral verification tool based on circular coinduction. In *CALCO*, volume 5728 of *Lecture Notes in Computer Science*, pages 433–442. Springer, 2009. doi:10.1007/978-3-642-03741-2_30.

[67] D. Lucanu and V. Rusu. Program equivalence by circular reasoning. *Formal Aspects of Computing*, 27(4):701–726, 2015. doi:10.1007/s00165-014-0319-6.

[68] R. Mayr and L. Clemente. Advanced automata minimization. In *POPL, 2013*, pages 63–74. ACM, 2013. doi:10.1145/2429069.2429079.

[69] A. Meduna and P. Zemek. Jumping finite automata. *Int. J. Found. Comput. Sci.*, 23(7):1555–1578, 2012. doi:10.1142/S0129054112500244.

[70] C. Parsons. On n-quantifier induction. *The Journal of Symbolic Logic*, 37(3):466–482, 1972.

[71] D. Perrin and J. Pin. *Infinite words - automata, semigroups, logic and games*, volume 141 of *Pure and applied mathematics series.* Elsevier Morgan Kaufmann, 2004.

[72] D. Perrin and J.-É. Pin. Semigroups and automata on infinite words. *NATO ASI Series C Mathematical and Physical Sciences-Advanced Study Institute*, 466:49–72, 1995.

[73] D. Pous. Coinduction all the way up. In *LICS*, pages 307–316. ACM, 2016. doi:10.1145/2933575.2934564.

[74] M. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, 1959. doi:10.1147/rd.32.0114.

[75] M. P. Schützenberger. Sur une variante des fonctions sequentielles. *Theor. Comput. Sci.*, 4(1):47–57, 1977.

[76] M. G. Scutellà. A note on dowling and gallier's top-down algorithm for propositional horn satisfiability. *J. Log. Program.*, 8(3):265–273, 1990. doi:10.1016/0743-1066(90)90026-2.

[77] A. Simpson. Cyclic arithmetic is equivalent to peano arithmetic. In *FoSSaCS*, pages 283–300, 2017. doi:10.1007/978-3-662-54458-7_17.

[78] S. G. Simpson. *Subsystems of second order arithmetic*, volume 1. Cambridge University Press, 2009.

[79] M. H. B. Sωrensen and P. Urzyczyn. Lectures on the curry-howard isomorphism. 1998. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.17.7385.

[80] D. Tabakov and M. Y. Vardi. Experimental evaluation of classical automata constructions. In G. Sutcliffe and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference, LPAR 2005, Montego Bay, Jamaica, December 2-6, 2005, Proceedings*, volume 3835 of *Lecture Notes in Computer Science*, pages 396–411. Springer, 2005. doi:10.1007/11591191_28.

[81] W. W. Tait. Infinitely long terms of transfinite type. In J. Crossley and M. Dummett, editors, *Formal Systems and Recursive Functions*, volume 40 of *Studies in Logic and the Foundations of Mathematics*, pages 176 – 185. Elsevier, 1965. doi:10.1016/S0049-237X(08)71689-6.

[82] W. W. Tait. Intensional interpretations of functionals of finite type I. *The Journal of Symbolic Logic*, 32(2):198–212, 1967. doi:10.2307/2271658.

[83] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975. doi:10.1145/321879.321884.

[84] A. Tarski. A Lattice-Theoretical Fixpoint Theorem and its Applications. *Pacific Journal of Mathematics*, 5(2):285–309, June 1955.

[85] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 133–191. Elsevier and MIT Press, 1990. doi:10.1016/b978-0-444-88074-1.50009-3.

[86] S. Thompson. *Type theory and functional programming.* International computer science series. Addison-Wesley, 1991.

[87] M.-H. Tsai, Y.-K. Tsay, and Y.-S. Hwang. Goal for games, omega-automata, and logics. In N. Sharygina and H. Veith, editors, *CAV*, pages 883–889, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[88] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for concurrency*, pages 238–266. Springer, 1996.

[89] M. Y. Vardi and T. Wilke. Automata: from logics to algorithms. In J. Flum, E. Grädel, and T. Wilke, editors, *Logic and Automata: History and Perspectives [in Honor of Wolfgang Thomas]*, volume 2 of *Texts in Logic and Games*, pages 629–736. Amsterdam University Press, 2008.

[90] J. Whittemore, J. Kim, and K. A. Sakallah. SATIRE: A new incremental satisfiability engine. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 542–545. ACM, 2001. doi:10.1145/378239.379019.

[91] M. D. Wulf, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Antichains: A new algorithm for checking universality of finite automata. In *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 17–30. Springer, 2006. doi:10.1007/11817963_5.

[92] A. C. Yao and R. L. Rivest. k+1 heads are better than k. *Journal of the ACM*, 25(2):337–340, 1978. doi:10.1145/322063.322076.