Automata for relation algebra and formal proofs

Damien Pous

- ÉCOLE NORMALE SUPÉRIEURE DE LYON -
Laboratoire de l'Informatique du Parallélisme - UMR5668 - LIP

HABILITATION À DIRIGER DES RECHERCHES

présentée et soutenue publiquement le 27 septembre 2016 par

Damien POUS

Automata
for relation algebra and formal proofs

devant la commission d'examen formée de

Luca	ACETO	rapporteur
Arnaud	DURAND	examinateur
Georges	GONTHIER	examinateur
Peter	JIPSEN	rapporteur et examinateur
Olivier	LAURENT	examinateur
François	POTTIER	rapporteur et examinateur
Igor	WALUKIEWICZ	examinateur

Acknowledgements

Je tiens tout d'abord à remercier Lucas Aceto, Peter Jipsen et François Pottier, qui ont accepté de relire ce manuscrit puis fait de nombreuses remarques afin de l'améliorer. Je remercie également Arnaud Durand, Georges Gonthier, Olivier Laurent et Igor Walukiewicz qui ont accepté de prendre part au jury.

Viennent ensuite les collègues, étudiants, anciens professeurs ou amis, hurluberlus sans qui ce travail n'aurait jamais vu le jour: de Jacques Sauloy à Filippo Bonchi, en passant par Daniel Hirschkoff, Tom (de Savoie), Monstrencage et Iouri.

Je suis également redevable aux institutions: le CNRS pour la liberté qu'il m'accorde, Plume pour sa convivialité inégalée, et le LIP pour l'air frais et le nouveau point de vue qu'apportent chaque déménagement sur les problèmes qui nous résistent trop.

Merci enfin à Pamitos et nos joyeux trublions, Lucas et Hugo.

Contents

Introduction

We review in this manuscript several results we obtained since our PhD. We organised those results into four chapters corresponding to four distinct but related fields in computer science.

The first chapter is a biased introduction to the *calculus of relations*. This field was initiated by DeMorgan, Peirce and then Schröder in the late XIXth century, and studied quite extensively by Tarski in the 1940's. It consists in understanding the algebraic and algorithmic properties of operations on binary relations. Our introduction is biased in the sense that we remove the most problematic operation from the beginning, set-theoretic complement, and that we consider reflexive transitive closure. This brings us to the concepts of allegories (Freyd and Scedrov, Andréka and Bredikhin) and Kleene algebra (Kleene, Conway, Kozen); and this allows us to state our recent results with Paul Brunet on their least common generalisation.

The second chapter is about algorithms for checking equivalence of finite automata. Such algorithms provide decision procedures for Kleene algebra; they are also a basic block for verification software: programs that make it possible to test the validity of other programs. There we present in detail a new algorithm we have discovered with Filippo Bonchi, based on a proof technique from concurrency theory: *bisimulations up to congruence*. We also present an extension of a standard algorithm by Hopcroft and Karp to deal with *symbolic automata*. Such automata are useful when working on large alphabets.

The third chapter pertains to the domain of formal mechanised proofs, where one uses the computer to write and proofcheck mathematical proofs (be they proofs of mathematical theorems, or correctness proofs for programs or systems). There, we discuss some applications of a library we developed for the Coq proof assistant, where we provide tools for automated reasoning in the calculus of relations, notably for Kleene algebra. These automation tools were obtained by implementing and certifying automata algorithms as well as important results about Kleene algebra.

In the last chapter we present an abstract theory of *coinduction*, a mathematical device coined by Milner in concurrency theory. This tool provides

powerful proof methods, especially for the study of state-based systems; it was a surprise to discover that it could also be used to obtain efficient algorithms, as in the second chapter. Our abstract theory of coinduction is a refinement of the work of Sangiorgi in the 1990's; it makes it simpler to provide enhancements of the coinductive proof method, and to mechanise them in proof assistants. We believe that it could help us to find new decision procedures for the calculus of relations, and then to certify them.

We point to the publications we assembled to obtain this manuscript in the notes on page 89.

Notation

We denote sets by capital letters $X, Y, S, T \ldots$ and functions by lower case letters $f, g, \ldots$ Given sets X and Y, $X \times Y$ is their Cartesian product, $X \uplus Y$ is their disjoint union and X^Y is the set of functions $f \colon Y \to X$. The collection of subsets of X is denoted by $\mathcal{P}(X)$. For a set of letters Σ, $\Sigma^\star$ denotes the set of all finite words over Σ; ϵ the empty word; and uv the concatenation of words $u, v \in \Sigma^\star$. We use 2 for the set $\{0, 1\}$.

Chapter 1

Relation algebra

We consider algebraic and algorithmic questions related to binary relations. On the algebraic side, we want to understand and characterise the laws governing the behaviour of standard operations on relations: union, intersection, composition, converse, etc.... On the algorithmic side, we look for decision procedures for equality or inclusion of relations.

We start by defining formally the calculus of relations; then we focus on two well-studied fragments of particular importance: Kleene algebras and allegories. Trying to unify those fragments lead us to a new result with Brunet, and several open questions.

We also define Kleene algebra with tests, a framework introduced by Kozen making it possible to deal with both relations and predicates.

1.1 The (positive) calculus of relations

Given a set P, a *relation* on P is a set of pairs of elements from P. For instance, the usual order on natural numbers is a relation. In the sequel, relations are ranged over using letters R, S, their set is written $\mathcal{P}(P\times P)$, and we write $p\ R\ q$ for $\langle p, q\rangle \in R$.

The set of relations is equipped with a partial order, set-theoretic inclusion ($\subseteq$), and three binary operations: set-theoretic union, written $R+S$, set-theoretic intersection, written $R\cap S$, and relational composition:

$$R\cdot S \triangleq \{\langle p, q\rangle \mid \exists r \in P,\ p\ R\ r\ \wedge\ r\ S\ q\}\ .$$

It also contains three specific relations: the empty relation, written 0, the universal relation, written $\top$, and the identity relation:

$$1 \triangleq \{\langle p, p\rangle \mid p \in P\}\ .$$

Lastly, one can consider three unary operations: set-theoretic complement, written R^c, converse (or transpose), R°, and reflexive-transitive closure, $R^\star$,

defined as follows:

$$R^c \triangleq \{\langle p, q\rangle \mid \neg p\, R\, q\} \ ,$$
$$R^\circ \triangleq \{\langle p, q\rangle \mid q\, R\, p\} \ ,$$
$$R^\star \triangleq \{\langle p, q\rangle \mid \exists p_0, \ldots, p_n,\ p_0 = p \ \wedge\ p_n = q \ \wedge\ \forall i < n,\ p_i\, R\, p_{i+1}\} \ .$$

We restrict ourselves to this list of operations here, even though it is not exhaustive. These operations make it possible to state many properties in a concise way, without mentioning the points related by the relations. Here are a few examples:

$1 \subseteq R$	R is reflexive: $\forall p \in P, p\, R\, p$
$R{\cdot}R \subseteq R$	R is transitive: $\forall pqr,\ p\, R\, r \wedge r\, R\, q \Rightarrow p\, R\, q$
$R{\cdot}R^\star \cap 1 = 0$	R is acyclic: $\forall p_0 \ldots p_n,\ n{>}0,\ (\forall i,\ p_i\, R\, p_{i+1}) \Rightarrow p_0 \neq p_n$
$R^\circ{\cdot}S \subseteq S{\cdot}R^\circ$	R and S commute: $\forall pqr,\ r\, R\, p \wedge r\, S\, q \Rightarrow \exists t,\ q\, R\, t \wedge p\, S\, t$

Moreover, these operations satisfy many laws. Some of these laws are extremely simple (composition is associative, $(R{\cdot}R'){\cdot}R'' = R{\cdot}(R'{\cdot}R'')$; the empty relation absorbs composition, $R \cdot 0 = 0 = 0 \cdot R$; reflexive-transitive closures are transitive, $R^\star \cdot R^\star \subseteq R^\star$). Others are much more complicated and counter-intuitive. To see this the interested reader can try to determine which of the following equations and inequations are universally true. (Over the eleven corresponding inequations, eight are true.)

$$1 \cap R \subseteq R{\cdot}R \cap R{\cdot}R{\cdot}R \tag{1.1}$$
$$(R + S)^\star = R^\star{\cdot}(S{\cdot}R^\star)^\star \tag{1.2}$$
$$(R + S)^\star = ((1 + R){\cdot}S)^\star \tag{1.3}$$
$$R{\cdot}(S \cap T) = R{\cdot}S \cap R{\cdot}T \tag{1.4}$$
$$R{\cdot}S \cap T \subseteq R{\cdot}(S \cap R^\circ{\cdot}T) \tag{1.5}$$
$$R{\cdot}S \cap T \subseteq (R \cap T{\cdot}S^\circ){\cdot}(S \cap R^\circ{\cdot}T) \tag{1.6}$$
$$(R \cap S{\cdot}\top){\cdot}T = R{\cdot}T \cap S{\cdot}\top \tag{1.7}$$

Two questions arise naturally:

1. is it possible to axiomatise the set of laws that are universally true, that is, to give a small number of elementary laws from which all valid laws follow?

2. is it possible to decide whether a law is valid or not?

When considering all the operations listed above, the answer is negative in both cases. Indeed, Monk proved that there cannot be a finite equational axiomatisation [94], and Tarski proved that the theory is actually undecidable [135, 137]. In both cases, reflexive-transitive closure is not necessary

but the complement plays a crucial role. Thus we focus in the sequel on the *positive* fragments, where complement is excluded.

Now we set up the concepts and notation needed in the sequel.

Let Σ be a set, whose elements are denoted by letters a, b. *Expressions* are defined by the following grammar:

$$e, f, g ::= e + f \mid e \cap f \mid e \cdot f \mid e^\circ \mid e^\star \mid 0 \mid 1 \mid \top \mid a \qquad (a \in \Sigma) \ .$$

We often omit the operator "$\cdot$" from expressions, writing ef for $e \cdot f$. Given a set E and a function $\sigma : \Sigma \to \mathcal{P}(E \times E)$ mapping a letter from Σ to a relation on E, we define inductively the extension $\hat{\sigma}$ of σ to expressions:

$$\begin{array}{lll} \hat{\sigma}(e+f) \triangleq \hat{\sigma}(e) + \hat{\sigma}(f) & \hat{\sigma}(e^\circ) \triangleq \hat{\sigma}(e)^\circ & \hat{\sigma}(1) \triangleq 1 \\ \hat{\sigma}(e \cap f) \triangleq \hat{\sigma}(e) \cap \hat{\sigma}(f) & \hat{\sigma}(e^\star) \triangleq \hat{\sigma}(e)^\star & \hat{\sigma}(\top) \triangleq \top \\ \hat{\sigma}(e \cdot f) \triangleq \hat{\sigma}(e) \cdot \hat{\sigma}(f) & \hat{\sigma}(0) \triangleq 0 & \hat{\sigma}(a) \triangleq \sigma(a) \end{array}$$

Given two expressions e and f, an equation is *valid*, written $\vDash e = f$, if for every set E and for every function $\sigma : \Sigma \to \mathcal{P}(E \times E)$, we have $\hat{\sigma}(e) = \hat{\sigma}(f)$. Intuitively, an equation is valid if it is universally true in relations, if it holds whatever the relations we use to interpret its variables.

Similarly, an inequation is valid, written $\vDash e \subseteq f$, if $\hat{\sigma}(e) \subseteq \hat{\sigma}(f)$ for every set E and every function $\sigma : \Sigma \to \mathcal{P}(E \times E)$. As soon as we have intersection or union, characterising valid equations is equivalent to characterising valid inequations: for all expressions e, f, we have $\vDash e = f$ iff $\vDash e \subseteq f$ and $\vDash f \subseteq e$; and $\vDash e \subseteq f$ iff $\vDash e + f = f$ iff $\vDash e \cap f = e$.

1.2 The ideal fragment: Kleene algebra

In this section we remove from the syntax the operations of intersection and converse, as well as the constant $\top$. In other words, we restrict to regular expressions:

$$e, f, g ::= e + f \mid e \cdot f \mid e^\star \mid 0 \mid 1 \mid a \qquad (a \in \Sigma) \ .$$

we shall see that with such a restriction, the validity of an equation is decidable, and more precisely, PSPACE-complete.

1.2.1 Decidability

Let letters u, v range over finite words over the alphabet Σ. A *language* is a set of words. We define inductively a function $[\cdot]$ associating a language to each expression:

$$\begin{array}{ll} [e+f] \triangleq [e] \cup [f] & [0] \triangleq \emptyset \\ [e \cdot f] \triangleq \{uv \mid u \in [e], v \in [f]\} & [1] \triangleq \{\epsilon\} \\ [e^\star] \triangleq \{u_1 \dots u_n \mid \forall i, u_i \in [e]\} & [a] \triangleq \{a\} \end{array}$$

The key result about this fragment of the calculus of relations is the following characterisation: an equation is valid for relations if and only if it corresponds to an equality of languages.

Theorem 1.2.1. *For all regular expressions e, f, we have*

$$\vDash e = f \qquad \textit{iff} \qquad [e] = [f] \ .$$

Inequations can be characterised in a similar way:

$$\vDash e \subseteq f \qquad \textit{iff} \qquad [e] \subseteq [f] \ .$$

This theorem is relatively easy. Its main consequence in practice is the decidability of the validity of equations: $[e]$ and $[f]$ are regular languages which we can easily represent using finite automata in order to compare them. This characterisation also gives us the precise complexity of the problem, as language equivalence of regular expressions is PSPACE-complete [91].

1.2.2 Axiomatisability

In 1956, Kleene asks for axiomatisations of the previous theory [41]: is it possible to find a small set of axioms (i.e., equations), from which all valid equations between regular expressions follow?

In the sixties, Salomaa gives two axiomatisations [125] which are not purely algebraic, and Redko proves that no finite equational axiomatisation can be complete [118]. Conway studies extensively this kind of questions in his monograph on regular algebra and finite automata [41], but we have to wait for the nineties for new results: Krob and Kozen independently show that one can axiomatise this theory in a finite way, but using axioms that are not just equations, but implications between equations. (We move from varieties to quasi-varieties.)

Krob's proof is long and difficult [85], but it provides a complete picture: first he gives a purely equational axiomatisation, infinite but with more structure than Salomaa's axioms. Then he shows that those infinitely many axioms can be derived from various finite axiomatisations involving implications between equations.

On the contrary, Kozen goes straight to the point and focuses on a specific finite axiomatisation (with implications). His proof is not simple either, but much shorter [77, 78].

Theorem 1.2.2 (Kozen'91, Krob'91)**.** *For all regular expressions e, f, we have $[e] = [f]$ if and only if the equality $e = f$ is derivable from the axioms listed in Figure 1.1, where notation $e \leq f$ is a shorthand for $e + f = f$.*

$$\left.\begin{aligned} e + (f + g) &= (e + f) + g \\ e + f &= f + e \\ e + 0 &= e \\ e + e &= e \end{aligned}\right\} \langle +, 0\rangle \text{ is a commutative and idempotent monoid}$$

$$\left.\begin{aligned} e(fg) &= (ef)g \\ e1 &= e \\ 1e &= e \end{aligned}\right\} \langle \cdot, 1\rangle \text{ is a monoid}$$

$$\left.\begin{aligned} e(f + g) &= ef + eg \\ (e + f)g &= eg + fg \\ e0 &= 0 \\ 0e &= 0 \end{aligned}\right\} \text{distributivity between the two monoids}$$

$$\left.\begin{aligned} 1 + ee^\star &= e^\star \\ ef \leq f \quad &\Rightarrow \quad e^\star f \leq f \\ fe \leq f \quad &\Rightarrow \quad fe^\star \leq f \end{aligned}\right\} \text{laws about Kleene star}$$

Figure 1.1: The axioms of Kleene algebra.

These axioms can be decomposed into four groups: the first three correspond to the fact that we have an idempotent non-commutative semiring; the last group of axioms characterises the operation of reflexive-transitive closure, often called "Kleene star" in this context. This group is not entirely symmetric: the law $1+e^\star e = e^\star$ is omitted as it can be derived from the other axioms. The last two axioms are implications; intuitively, they tell that if an expression f is invariant under composition with another expression e, then it is also invariant with $e^\star$. The expressive power of the axiomatisation mainly comes from those two implications: they make it possible to reason inductively on Kleene star, in a purely algebraic way. (An equipollent and perhaps more intuitive set of axioms states that that $e^\star f$ is the smallest fixpoint of the function $x \mapsto f + xe$, and symmetrically for $fe^\star$.)

One easily checks that each of these axioms is valid in the model of binary relations, but also when interpreting the expressions e, f, g as arbitrary languages. The converse implication from Theorem 1.2.2 follows from this remark: we prove only valid equations using those axioms.

The difficulty lies in the other implication: the completeness of these axioms, the fact that every valid equation can be deduced from these axioms. We do not detail the proof here; a key step consists in showing that the set of matrices with coefficients in a Kleene algebra forms a new Kleene

algebra (a Kleene algebra being a structure satisfying the axioms from Figure 1.1).

1.3 The strange fragment: allegories

Now consider a different fragment, where we only have composition, intersection, converse, and constants 1 and $\top$. For reasons to become clear in Section 1.4, we reuse letters u, v, w to denote the corresponding regular expressions, which we call *terms*:

$$u, v, w ::= u \cdot v \mid u \cap v \mid u^\circ \mid 1 \mid \top \mid a \qquad (a \in \Sigma) \ .$$

Modulo the presence of the constant $\top$, this fragment was studied by Andréka and Bredikhin [9], and by Freyd and Scedrov [52] under the name of (representable) *allegories*. We will see that one can decide the validity of inequations in this fragment, but that the corresponding theory is not finitely axiomatisable, even using implications between equations.

1.3.1 Decidability

The key idea consists in characterising valid inequations by the existence of graph homomorphisms. More precisely, homomorphisms of directed and edge-labelled graphs with two distinguished vertices.

Definition 1.3.1 (Graph)**.** *A* graph *is a tuple $\langle V, E, \iota, o\rangle$, where V is a set of* vertices*, $E \subseteq V \times \Sigma \times V$ is a set of labelled edges, and $\iota, o \in V$ are two distinguished vertices, respectively called* input *and* output*.*

We let letters G, H range over graphs and we define the following operations:

- $G \cdot H$ is the graph obtained by composing the two graphs in series, that is, by putting them one after the other and by merging the output of G with the input of H;
- $G \cap H$ is the graph obtained by composing the two graphs in parallel, that is, by putting them side by side and by merging their inputs and their outputs;
- G° is the graph obtained from G by exchanging input and output (without reversing edges);
- $\underline{1}$ is the graph without edges and with a single vertex ($\langle \{*\}, \emptyset, *, *\rangle$);
- $\underline{\top}$ is the graph without edges and with two vertices, where input and output are distinct ($\langle \{\star, \bullet\}, \emptyset, \star, \bullet\rangle$);

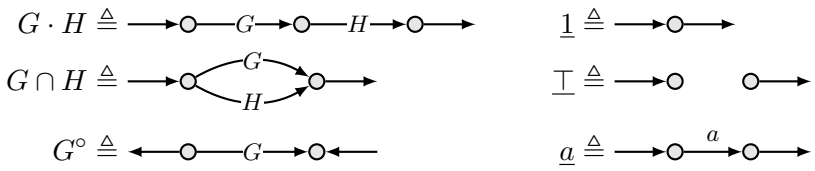


Figure 1.2: Operations on graphs.

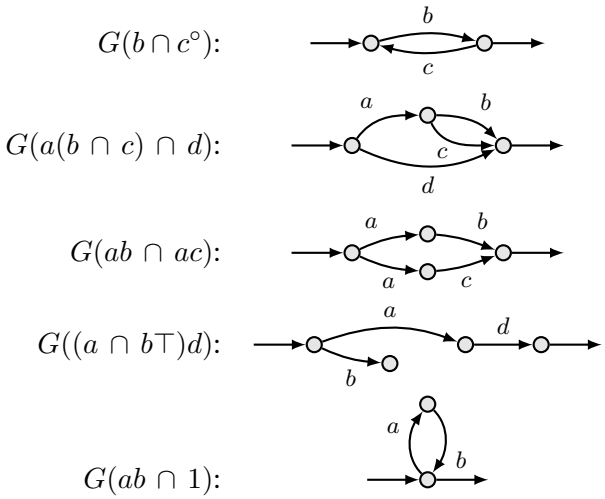


Figure 1.3: Graphs associated to some terms.

- for $a \in \Sigma$, $\underline{a}$ is the graph with two vertices and an edge labelled a from the input to the output $(\langle\{\star, \bullet\}, \{\langle \star, a, \bullet\rangle\}, \star, \bullet\rangle)$.

These operations are depicted in Figure 1.2; the input and the output of each graph is denoted using unlabelled arrows. These operations make it possible to associate a graph $G(u)$ to every term u, by structural induction:

$$\begin{aligned} G(u \cdot v) &\triangleq G(u) \cdot G(v) & G(1) &\triangleq \underline{1} \\ G(u \cap v) &\triangleq G(u) \cap G(v) & G(\top) &\triangleq \underline{\top} \\ G(u^\circ) &\triangleq G(u)^\circ & G(a) &\triangleq \underline{a} \end{aligned}$$

The graphs of a few terms are drawn in Figure 1.3. These are series-parallel graphs as long as we do not use converse and identity, that introduce loops in presence of intersection, nor the constant $\top$, that can disconnect some parts of the graphs.

Some graphs are not associated to any term. The canonical counter-example is the following one. (The labelling and the orientation of the five

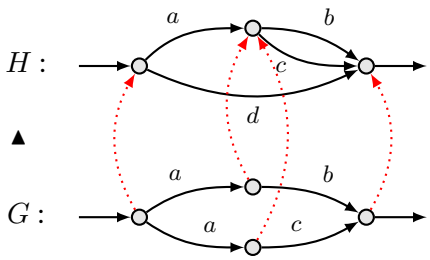
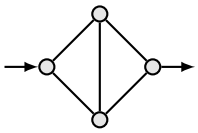


Figure 1.4: A graph homomorphism.

edges is irrelevant so that we omit this information.)



(1.8)

One can compare graphs using homomorphisms:

Definition 1.3.2. *A* homomorphism *from the graph G to the graph H is a function from vertices of G to vertices of H that preserves labelled edges, input, and output. We write $H \blacktriangleleft G$ when there exists a homomorphism from G to H.*

The relation $\blacktriangleleft$ is a preorder on graphs. As an example, the graph of $a(b \cap c) \cap d$ is smaller than that of $ab \cap ac$, thanks to the homomorphism depicted in Figure 1.4 using dotted arrows. Note that homomorphisms need not be injective or surjective, so that the preorder is completely unrelated to the sizes of the graphs: a graph may perfectly be smaller than another one, in the sense of the preorder, while having more vertices or edges (and vice-versa).

The nice property of the fragment considered here is the following characterisation: an inequation is valid for relations if and only if there exists a homomorphism between the underlying graphs:

Theorem 1.3.3 ([9, Theorem 1], [52, page 208])**.** *For all terms u, v, we have*

$$\vDash u \subseteq v \qquad \textit{iff} \qquad G(u) \blacktriangleleft G(v) \ .$$

Graphs of terms being finite, one can look for a homomorphism between two such graphs in an exhaustive way, whence the decidability of the problem.

Chandra and Merlin actually proved a similar result earlier, in the context of databases [37]: they showed that the containment of conjunctive queries can be reduced the problem of finding a graph homomorphism. There, a conjunctive query is a first order formula without function symbols, using only conjunctions, truth, and existential quantifiers. The considered graphs are actually hyper-graphs: edges correspond to predicates

appearing in the formulas, which might have arbitrary arity; and they can have arbitrarily many designated vertices: these correspond to the free variables of the formulas.

Except for the identity constant, the semantics of an allegorical expression is precisely such a conjunctive formula, with at most two free variables and where all predicates have arity two. Theorem 1.3.3 is thus almost a corollary of the result by Chandra and Merlin.

Concerning complexity, the graph homomorphism problem is well-known to be NP-complete, and so is the containment of conjunctive queries. However we are in a very restricted case with allegories: the graphs of terms have treewidth at most two. General results thus ensure that the problem is polynomial [57]. More pragmatically, given an (arbitrary) graph G with n vertices and an expression u of size m, a simple dynamic programming algorithm decides whether there exists a homomorphism from $G(u)$ into G in $O(n^2m)$ operations. (We plan to develop those observations, which are possibly new, in a future paper.)

1.3.2 Axiomatisability

Freyd and Scedrov define *allegories* [52] as structures satisfying the axioms from Figure 1.5[1], where notation $e \subseteq f$ is a shorthand for $e \cap f = e$. First note that composition does not distribute over intersections: composition is monotone in its two arguments, which entails the following inequations but not their converses:

$$e(f \cap g) \subseteq ef \cap eg$$
$$(f \cap g)e \subseteq fe \cap ge$$

One can also deduce from the axioms that converse reverses composition, distributes over intersections, and preserves constants 1 and $\top$:

$$(e \cap f)^\circ = e^\circ \cap f^\circ \qquad \top^\circ = \top$$
$$(ef)^\circ = f^\circ e^\circ \qquad 1^\circ = 1$$

The last axiom in Figure 1.5 is uncommon. It is called *modularity law* and is equivalent in presence of the other axioms to its symmetrical counterpart:

$$ef \cap g \subseteq e(f \cap e^\circ g)$$

It also entails the following inequation, known as *Dedekind's inequality*:

$$ef \cap g \subseteq (e \cap gf^\circ)(f \cap e^\circ g)$$

[1] Up to some details: they do not consider the constant $\top$, and they work in a categorical setting, where the various operations are typed.

$$
\left.\begin{aligned} e \cap (f \cap g) &= (e \cap f) \cap g \\ e \cap f &= f \cap e \\ e \cap \top &= e \\ e \cap e &= e \end{aligned}\right\} \langle \cap, \top \rangle \text{ is a commutative and idempotent monoid}
$$

$$
\left.\begin{aligned} e(fg) &= (ef)g \\ e1 &= e \\ 1e &= e \end{aligned}\right\} \langle \cdot, 1 \rangle \text{ is a monoid}
$$

$$
\left.\begin{aligned} e(f \cap g) &\subseteq ef \\ (f \cap g)e &\subseteq fe \end{aligned}\right\} \text{composition is monotone}
$$

$$
\left.\begin{aligned} e^{\circ\circ} &= e \\ (e \cap f)^\circ &\subseteq e^\circ \\ (ef)^\circ &\subseteq f^\circ e^\circ \end{aligned}\right\} \text{converse is a monotone involution reversing composition}
$$

$$
\left.\begin{aligned} ef \cap g \subseteq (e \cap gf^\circ)f \end{aligned}\right\} \text{modularity law}
$$

Figure 1.5: Axioms of allegories.

Unfortunately, this finite and purely equational axiomatisation is not complete for relations: some valid equations are not consequences of the axioms[2]. Freyd and Scedrov actually proved that there exists no finite equational axiomatisation [52, page 210]. Hodkinson and Mikulás moreover showed that there cannot be a finite first-order axiomatisation [63], and in particular a quasi-equational one like, e.g., for Kleene algebra.

1.4 Putting it all together: Kleene allegories

Let us come back to the initial problem, that of the positive calculus of relations. We have seen that two fragments are decidable: the fragment corresponding to regular expressions $(+, \cdot, \cdot^\star, 0, 1)$, and that corresponding to allegories $(\cap, \cdot, \cdot^\circ, \top, 1)$. What happens when we take all operations?

First note that the function $[\cdot]$ associating a (regular) language to every

[2]The counter-examples the author is aware of are not really informative. Freyd and Scedrov claim that homomorphisms which equate at most two vertices at a time give rise to laws provable from these axioms ; in practice, laws requiring more are quite convoluted.

regular expression can be extended to the operations of allegories:

$$\begin{aligned}[e \cap f] &\triangleq [e] \cap [f] \\ [e^\circ] &\triangleq \{a_n \dots a_1 \mid a_1 \dots a_n \in [e]\} \\ [\top] &\triangleq \Sigma^\star\end{aligned}$$

However, the characterisation obtained in Theorem 1.2.1 no longer works with these operations. Indeed, we have for instance

$$\begin{aligned}[a \cap b] = \{a\} \cap \{b\} = \emptyset = [0] \text{ but } &\nvDash a \cap b = 0 \\ [a^\circ] = \{a\} = [a] \text{ but } &\nvDash a^\circ = a \\ [a] = \{a\} \not\subseteq \{aaa\} = [aa^\circ a] \text{ but } &\vDash a \subseteq aa^\circ a \\ [\top a \top b \top] \neq [\top b \top a \top] \text{ but } &\vDash \top a \top b \top = \top b \top a \top\end{aligned}$$

To obtain a characterisation, we actually have to replace words (elements of $\Sigma^\star$) by graphs, and thus consider languages of graphs.

Definition 1.4.1. *The* language of graphs *of an expression e, written $\mathcal{G}(e)$, is defined as follows, by induction on e:*

$$\begin{aligned}\mathcal{G}(e+f) &\triangleq \mathcal{G}(e) \cup \mathcal{G}(f) & \mathcal{G}(0) &\triangleq \emptyset \\ \mathcal{G}(e \cap f) &\triangleq \{G \cap H \mid G \in \mathcal{G}(e),\ H \in \mathcal{G}(f)\} & \mathcal{G}(\top) &\triangleq \{\underline{\top}\} \\ \mathcal{G}(e \cdot f) &\triangleq \{G \cdot H \mid G \in \mathcal{G}(e),\ H \in \mathcal{G}(f)\} & \mathcal{G}(1) &\triangleq \{\underline{1}\} \\ \mathcal{G}(e^\star) &\triangleq \{G_1 \cdot \dots \cdot G_n \mid n \in \mathbb{N},\ \forall i \leq n, , G_i \in \mathcal{G}(e)\} & \mathcal{G}(a) &\triangleq \{\underline{a}\} \\ \mathcal{G}(e^\circ) &\triangleq \{G^\circ \mid G \in \mathcal{G}(e)\}\end{aligned}$$

This definition generalises the usual notion of language: when the considered expression contains no intersection, no converse, and no constant $\top$, then the associated graphs are isomorphic to words: these are simple threads labelled by letters in Σ.

To generalise also allegories, we have to make use of graph homomorphisms. Given a set L of graphs, we write ${}^\blacktriangleleft L$ for is *downward closure*:

$${}^\blacktriangleleft L \triangleq \{G \mid \exists H, G \blacktriangleleft H,\ H \in L\} \ .$$

We finally obtain the following characterisation:

Theorem 1.4.2. *For all expressions e and f, we have*

$$\vDash e \subseteq f \qquad \textit{iff} \qquad \mathcal{G}(e) \subseteq {}^\blacktriangleleft \mathcal{G}(f) \ .$$

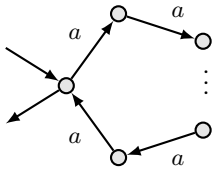
We stated this characterisation with Brunet [32, Theorem 6]; its proof merely consists in assembling results from the literature. This characterisation generalises both Theorem 1.2.1 and Theorem 1.3.3. If e and f are

regular expressions, then all graphs in $\mathcal{G}(e)$ and $\mathcal{G}(f)$ are threads, and the unique possible homomorphism between two such graphs is the identity; whence $\mathcal{G}(e) \subseteq {}^{\blacktriangleleft}\mathcal{G}(f)$ iff $\mathcal{G}(e) \subseteq \mathcal{G}(f)$. If instead e and f are terms u and v, then $\mathcal{G}(e) = \{G(u)\}$ and $\mathcal{G}(f) = \{G(v)\}$, so that $\mathcal{G}(e) \subseteq {}^{\blacktriangleleft}\mathcal{G}(f)$ is equivalent to $G(u) \blacktriangleleft G(v)$.

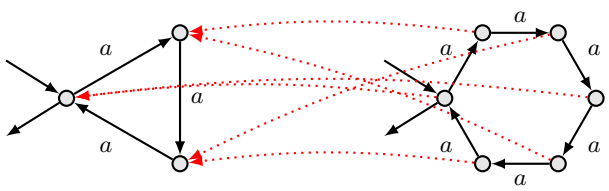
Note also that for all graph languages L, K, we have $L \subseteq {}^{\blacktriangleleft}K$ iff ${}^{\blacktriangleleft}L \subseteq {}^{\blacktriangleleft}K$. Valid equations are thus characterised as follows:

$$\vDash e = f \qquad \text{iff} \qquad {}^{\blacktriangleleft}\mathcal{G}(e) = {}^{\blacktriangleleft}\mathcal{G}(f) \ .$$

To illustrate this theorem, consider expressions $e \triangleq a^+ \cap 1$ and $f \triangleq (aa)^+ \cap 1$, where g^+ is a shorthand for $gg^\star$. The set of graphs $\mathcal{G}(e)$ is the set of non-trivial cycles labelled with a:



On the other side, $\mathcal{G}(f)$ is the set of non-trivial cycles of even length. Thus we immediately get $\mathcal{G}(f) \subseteq \mathcal{G}(e) \subseteq {}^{\blacktriangleleft}\mathcal{G}(e)$, whence $\vDash f \subseteq e$. The converse inequation is also valid: to each cycle from $\mathcal{G}(e)$, possibly of odd length, one can associate the cycle of double length, in $\mathcal{G}(f)$; indeed, there is a homomorphism from this cycle of double length into the shorter one:



The following three laws can be proved valid in the same way.

$$(a \cap bb)^\star \subseteq a^\star \cap b^\star$$
$$((a \cap b)(1 \cap b)(a \cap b))^\star \subseteq (a \cap bb)^\star$$
$$(a \cap b\top)^\star(1 \cap b\top) = (1 \cap \top b^\circ)(a \cap \top b^\circ)^\star$$

1.4.1 Decidability

Together with Brunet [32], we proposed an automata model allowing us to recognise languages of graphs associated to expressions. This automata model takes inspiration from Petri nets [101, 97], which make it possible to explore richer structures than plain words. To each expression e, we associate what we call a *Petri automaton*, whose language is precisely ${}^{\blacktriangleleft}\mathcal{G}(e)$.

Thanks to Theorem 1.4.2, the problem of validity of equations or inequations thus reduces to the problem of comparing Petri automata.

As of today, we solved this algorithmic problem only for a fragment of the calculus: we have to forbid converse and constants 1 and $\top$, and replace reflexive-transitive closure $\cdot^\star$ by transitive closure $\cdot^+$ (because reflexive-transitive closure implicitly contains the identity: we have $1 = 0^\star$). The corresponding equational theory was recently studied by Andréka, Mikulás, and Németi [8]; over this signature, it coincides with the equational theory of languages. Under this restriction, the considered graphs are always acyclic, so that the automata become simpler to compare: we have shown that the problem of comparing these automata is EXPSPACE-complete [32]. This fragment is thus EXPSPACE-easy. (Our reduction does not allow us to deduce a lower-bound.)

Note that intersection is the problematic operation: without intersection (and associated constant $\top$), we obtain *Kleene algebras with converse*, for which Bloom, Ésik and Stefanescu have obtained decidability [17], and for which we obtained PSPACE-completeness with Brunet [30, 31].

Coming back to the whole calculus, Brunet just proved a Kleene theorem for our automata model [29]: given a Petri automaton, one can extract an expression accepting the same set of graphs. As a consequence, decidability of the whole calculus is equivalent to decidability of this automata model.

1.4.2 Axiomatisability

Hodkinson and Mikulás proved that there cannot be a finite axiomatisation whenever the considered fragment contains composition, intersection, and converse [63]. Even a quasi-equational one.

Without intersection and associated constant $\top$, Bernátsky and Ésik have shown that the following five axioms suffice when added to a complete axiomatisation of Kleene algebras (e.g., those from Figure 1.1) [45].

$$(ef)^\circ = f^\circ e^\circ \qquad e^{\circ\star} = e^{\star\circ} \qquad e \subseteq ee^\circ e$$
$$(e+f)^\circ = e^\circ + f^\circ \qquad e^{\circ\circ} = e$$

Andréka and Mikulás cleaned up the situation for the remaining fragments excluding Kleene star [7], but their finite axiomatisability result is wrong when the identity is included, and they use an alternative definition of $\top$. (With their definition, $\top$ can be axiomatised just as a top element: laws such as (1.7) or $\top a \top b \top = \top b \top a \top$ are not valid for them.) Therefore, several questions remain open, even without considering Kleene star.

With Kleene star, we would like to understand whether Kleene algebra axioms (Figure 1.1) are sufficient when added to a complete axiomatisation of (representable) allegories. We could not find any counter-example

to completeness so far, and a proof of such a result seems challenging: the existing proofs of completeness for Kleene algebra are far from trivial.

Another question is whether partial axiomatisations give rise to decidable equational theories. For instance, by using a finer notion of graph homomorphism, Gutiérrez has shown that allegories, as axiomatised in Figure 1.5, are decidable [58]. Could such a result be extended to deal with Kleene star?

1.5 Kleene algebra with tests

We finally consider Kleene algebra with tests (KAT), an equational system for program verification, which can also be seen as an extension of the calculus of relations. The theory of KAT has been developed by Kozen et al. [79, 40, 81], it has received much attention for its applications in various verification tasks ranging from compiler optimisation [82] to program schematology [10], and very recently for network programming analysis [6, 49].

As in the case of Kleene algebra, KAT admits a finite based quasi-equational axiomatisation, and it enjoys a decidable equational theory (actually, PSPACE-complete) thanks to a reduction to an automata problem. We will use an algorithm implemented in Coq in Chapter 3. Unlike in the previous sections, we introduce KAT as an algebraic structure rather than through the calculus of relations: such a presentation is possible but less relevant here.

A Kleene algebra with tests is a tuple $\mathcal{K} = \langle X, B, [\cdot] \rangle$ where X is a Kleene algebra (Figure 1.1), B is a Boolean algebra of *tests*, and $[\cdot] : B \to X$ is a homomorphism from $(B, \wedge, \vee, \top, \bot)$ to $(X, \cdot, +, 1, 0)$.

The prototypical example is that of binary relations and predicates: given a support set P, binary relations on P form a Kleene algebra, (decidable) predicates on P form a Boolean algebra, and every predicate $\phi \in \mathcal{P}(P)$ can be faithfully represented as a relation $[\phi] \in \mathcal{P}(P \times P)$:

$$[\phi] = \{(p, p) \mid p \in \phi\} \ .$$

One checks easily that this embedding is a homomorphism of appropriate type:

$$[\phi \cap \psi] = [\phi] \cdot [\psi] \qquad [\phi \cup \psi] = [\phi] + [\psi] \qquad [P] = 1 \qquad [\emptyset] = 0$$

This model is typically used to interpret imperative programs (see Section 3.2.1): such programs are state transformers, i.e., binary relations between states, and the conditions appearing in these programs are just predicates on states. The Kleene algebra component deals with the control-flow

graph of the programs—sequential composition, iteration, and branching—while the Boolean algebra component deals with the conditions appearing in if-then-else statements, while loops, or pre- and post-assertions.

The equational theory of Kleene algebra with tests is complete over this relational model [83]: any equation $e = f$ that holds universally in this model can be proved from the axioms of KAT (i.e., Kleene algebra axioms, Boolean algebra axioms, and the axioms corresponding to the homomorphism from tests to programs). In practice, this means that if an equation cannot be proved from KAT axioms, then it cannot be universally true on binary relations: proving its validity for a particular instantiation of the variables necessarily requires one to exploit additional properties of this particular instance.

We describe two other models in the sequel: the syntactic model and the model of guarded string languages. There are however other important models of KAT. First of all, any Kleene algebra can be extended into a Kleene algebra with tests by embedding the two-element Boolean lattice. We also have traces models (where one keeps track of the whole execution traces of the programs rather than just their starting and ending points), matrices over a Kleene algebra with tests, but also models inherited from semirings like min-plus and max-plus algebra, or convex-polygon semirings [70]. The latter models have a degenerate Kleene star operation; they become useful when one constructs matrices over them, for instance to study shortest path algorithms.

1.5.1 KAT expressions

Let p, q range over a set Σ of *letters* (or *actions*), and let $a_1, \ldots, a_n$ be the elements of a finite set Θ of *primitive tests*. *Boolean expressions* and *KAT expressions* are defined by the following syntax:

$$a, b ::= a_i \in \Theta \mid a \wedge a \mid a \vee a \mid \neg a \mid \top \mid \bot \qquad \text{(Boolean expressions)}$$
$$e, f ::= p \in \Sigma \mid [a] \mid e \cdot f \mid e + f \mid e^\star \mid 1 \mid 0 \ . \qquad \text{(KAT expressions)}$$

(KAT expressions are regular expressions over an alphabet consisting either of plain letters p, q, or Boolean formulas a, b over primitive tests $a_1, \ldots, a_n$.) Given a Kleene algebra with tests $\mathcal{K} = \langle X, B, [\cdot] \rangle$, any pair of maps $\theta : \Theta \to B$ and $\sigma : \Sigma \to X$ gives rise to a KAT homomorphism allowing one to interpret expressions in $\mathcal{K}$. Given two such expressions e and f, the equation $e = f$ is a *KAT theorem*, written $\mathrm{KAT} \vdash e = f$, when the equation holds in any Kleene algebra with tests, under any interpretation. One checks easily that KAT expressions quotiented by the latter relation form a Kleene algebra with tests; this is the free Kleene algebra with tests over Σ and Θ.

The following (in)equations illustrate the kind of laws that hold in all

Kleene algebra with tests:

$$[a \vee \neg a] = 1 \qquad [a \wedge (\neg a \vee b)] = [a][b] = [\neg(\neg a \vee \neg b)]$$

$$e^\star e^\star = e^\star \qquad (e+f)^\star = e^\star(fe^\star)^\star \qquad (e + eef)^\star \leq (e + ef)^\star$$

$$[a]([\neg a]e)^\star = [a] \qquad [a]([a]e[\neg a] + [\neg a]f[a])^\star[a] \leq (ef)^\star$$

(As before in this chapter, the preorder $\leq$ is defined by $e \leq f \triangleq e + f = f$.) The laws from the first line come from the Boolean algebra structure, while the ones from the second line come from the Kleene algebra structure. The two laws from the last line are more interesting: their proof must mix both Boolean algebra and Kleene algebra reasoning.

1.5.2 Guarded string languages

Guarded string languages are the natural generalisation of string languages for Kleene algebra with tests. We briefly define them.

An *atom* is a function from primitive tests (Θ) to Booleans; it indicates which of these tests are satisfied. We let α, β range over atoms, the set of which is denoted by At. We let u, v range over *guarded strings* [73]: alternating sequences of atoms and letters, which both start and end with an atom:

$$\alpha_1, p_1, \dots, \alpha_n, p_n, \alpha_{n+1} \ .$$

The concatenation $u * v$ of two guarded strings u, v is a partial operation: it is defined only if the last atom of u is equal to the first atom of v; it consists in concatenating the two sequences and removing one copy of the shared atom in the middle.

$$(\dots, p_n, \alpha_{n+1}) * (\beta_1, q_1, \dots) = \begin{cases} \dots, p_n, \beta_1, q_1, \dots & \text{if } \alpha_{n+1} = \beta_1 \\ \text{undefined otherwise} \end{cases}$$

This partial operation is easily shown to be associative. The Kleene algebra with tests of guarded string languages is obtained by considering sets of guarded strings for X and sets of atoms for B:

$$\begin{aligned} X &= \mathcal{P}((At \times \Sigma)^\star \times At) & B &= \mathcal{P}(At) \\ L \cdot K &= \{u * v \mid u \in L \wedge v \in K\} & a \wedge b &= a \cap b \\ L + K &= L \cup K & a \vee b &= a \cup b \\ L^\star &= \{u_1 * \dots * u_n \mid \exists u_1 \dots u_n, \forall i \leq n, u_i \in L\} & \neg a &= At \setminus a \\ 1 &= \{\alpha \mid \alpha \in At\} & \top &= At \\ 0 &= \emptyset \qquad [a] = \{\alpha \mid \alpha \in a\} & \bot &= \emptyset \end{aligned}$$

(Note that we slightly abuse notation by letting α denote either an atom, or a guarded string reduced to an atom.)

1.5.3 Completeness

Let G be the unique homomorphism from KAT expressions to guarded string languages such that a primitive test is mapped to the set of atoms that declare it to hold, and a letter p is mapped to the set of all guarded strings containing a single letter, p:

$$G(a_i) = \{\alpha \mid \alpha(a_i) \text{ is true}\} \qquad G(p) = \{\alpha p \beta \mid \alpha, \beta \in At\}$$

Completeness of KAT over guarded string languages can be stated as follows.

Theorem 1.5.1. *For all KAT expressions e, f, $\mathrm{KAT} \vdash e = f$ iff $G(e) = G(f)$.*

The left-to-right implication is easy since guarded strings form a model of KAT. The converse implication is the difficult one; it was proved by Kozen and Smith [83]. Their proof consists in bringing KAT expressions into a format that makes it possible to apply completeness of Kleene algebra over languages (Theorem 1.2.1).

This theorem allows one to prove (in)equations valid in every model of KAT, by resorting to an algorithm deciding guarded string language equivalence. We exploit this idea to automate formal proofs in Chapter 3.

1.5.4 Eliminating hypotheses

Theorem 1.5.1 gives a decision procedure for the equational theory of KAT, that is, in absence of any hypothesis. For instance, an equation like $(p + q)^\star = p^\star q^\star$, which is provably true under the hypothesis that $pq = qp$, cannot be deduced by means of this theorem.

Some hypotheses can however be exploited [39, 59]: those having one of the following shapes.

(i) $e = 0$;

(ii) $[a]e = e[b]$, $[a]e \leq e[b]$, or $e[b] \leq [a]e$;

(iii) $e \leq [a]e$ or $e \leq e[a]$

(iv) $a = b$ or $a \leq b$;

(v) $[a]p = [a]$ or $p[a] = [a]$, for atomic $p \in \Sigma$;

Equations of the first kind (i) are called "Hoare" equations, for reasons to become apparent in Section 3.2.2. They can be eliminated using the following implication:

$$\begin{cases} e + ugu = f + ugu \\ g = 0 \end{cases} \quad \text{entails} \quad e = f \ . \tag{†}$$

This implication is valid for any term u, and the method is complete [59] when u is taken to be the universal KAT expression, $\Sigma^\star$. Intuitively, for this choice of u, ugu recognises all guarded strings that contain a guarded string of g as a substring. Therefore, when checking that $e + ugu = f + ugu$ are language equivalent rather than $e = f$, we rule out all counter-examples to $e = f$ that contain a substring belonging to g: such counter-examples are irrelevant since g is known to be empty.

Equations of the shape (iii) and (iv) are actually special cases of those of the shape (ii), which are in turn equivalent to Hoare equations. For instance, we have $[a]e \leq e[b]$ iff $[a]e[\neg b] = 0$. Moreover, two hypotheses of shape (i) can be merged into a single one using the fact that $e = 0 \wedge f = 0$ iff $e + f = 0$. Therefore, we can aggregate all hypotheses of shape (i-iv) into a single one (of shape (i)), and use the above technique just once.

Hypotheses of shape (v) are handled differently, using the following equivalence:

$$[a]p = [a] \quad \text{iff} \quad p = [\neg a]p + [a] \ , \tag{‡}$$

This equivalence allows us to substitute $[\neg a]p + [a]$ for p in the considered goal—whence the need for p to be atomic. Again, the method is complete [59], i.e.,

$$\mathrm{KAT} \vdash ([a]p = [a] \Rightarrow e = f) \quad \text{iff} \quad \mathrm{KAT} \vdash e\theta = f\theta \ ,$$

where θ is the substitution $\{p \mapsto [\neg a]p + [a]\}$.

1.5.5 KAT and the calculus of relations

We conclude this chapter with three additional questions.

First, in our presentation of the calculus of relations, we did exclude set-theoretic complement from the beginning, as it immediately brings undecidability and incompleteness (Section 1.1). On the other hand, by restricting complementation to tests, Kleene algebra with tests remains decidable and finitely based. This raises the following question: in which other fragments of the calculus of relations can we add Boolean tests while retaining decidability and finite axiomatisability? It would seem reasonable that any fragment that is decidable or finitely axiomatisable without tests would remain so with such an addition. Still, this deserves a proper study.

Second, one can easily define a weaker notion of Kleene algebra with tests, where tests are not a Boolean algebra but a Heyting algebra, or just a bounded distributive lattice. Do these alternative remain decidable and finitely axiomatisable? We believe that this is the case for bounded distributive lattices, but the case of Heyting algebras seems harder. (The Boolean

case is no longer conservative, and the free Heyting algebra over a single generator is already infinite.)

The case of bounded distributive lattices is interesting because it is a fragment of the calculus of relations: it can be encoded using operations $\langle \cdot, +, \cap, \cdot^\star, 1, 0 \rangle$. Indeed, starting from a KAT expression without complementation, one can push all occurrences of $[.]$ towards the leaves, and encode a primitive test $[a_i]$ using the expression $a_i \cap 1$. (If the KAT expression is over the alphabets Σ and Θ, the resulting expression is over the alphabet $\Sigma \uplus \Theta$.) Of course this fragment of the calculus is much more expressive: expressions such as $p \cap q$, $pq \cap 1$, or $(p \cap 1)p$ do not belong to the image of this translation. Still, this means that decidability of Kleene algebra with tests in a bounded distributive lattice is a necessary step for decidability of the $\langle \cdot, +, \cap, \cdot^\star, 1, 0 \rangle$ fragment of the calculus of relations.

Lastly, we have seen that certain kinds of hypotheses can be eliminated in Kleene algebra with tests (Section 1.5.4). What kind of hypotheses can be eliminated in other fragments of the calculus of relations?

Chapter 2

Automata algorithms

Checking language equivalence of finite automata is a classic problem in computer science, with many applications in areas ranging from compilers to model checking. We have seen in the previous chapter that the validity of (in)equations in Kleene algebra reduces to this problem. In this chapter we present new algorithms for it.

A wide range of algorithms in computer science build on the ability to check language equivalence or inclusion of finite automata. In model-checking for instance, one can build an automaton for a formula and an automaton for a model, and then check that the latter is included in the former. More advanced constructions need to build a sequence of automata by applying a transducer, and to stop whenever two subsequent automata recognise the same language [24]. Another field of application is that of various extensions of Kleene algebra (Chapter 1), whose equational theories are reducible to language equivalence of various kinds of automata: regular expressions and finite automata for plain Kleene algebra [78], "closed" automata for Kleene algebra with converse [17, 45], or guarded string automata for Kleene algebra with tests (KAT) [81].

Equivalence of deterministic finite automata (DFA) can be checked either via minimisation [64] or, more directly, through Hopcroft and Karp's algorithm [66]. This latter algorithm for checking language equivalence of finite automata can be seen as an instance of Huet's first-order unification algorithm without occur-check [67, Section 5.8]: one tries to unify the two automata recursively, keeping track of the generated equivalence classes of states using an efficient union-find data-structure.

We describe this algorithm formally in Section 2.1; its complexity has been studied by Tarjan [134]: checking language equivalence of two states in a DFA with n states over an alphabet of size k requires $O(nk\alpha(k,n))$ operations, where $\alpha(k,n)$ is a *very* slow-growing inverse of Ackermann's function. This might look rather satisfactory, except that: 1) in most applications one starts with non-deterministic automata (NFA), and 2) sometimes

the alphabet is too large to be iterated naively.

For the first point, it is well-known that NFA can be determinised using the powerset construction, and that there can be exponentially many reachable sets. In fact, language equivalence becomes PSPACE-complete for NFA over an alphabet with at least two letters [91]—and coNP-complete with one letter. In Section 2.2 we give an algorithm we developed with Filippo Bonchi [21], which can improve exponentially over both Hopcroft and Karp's algorithm and more recent algorithms based on *antichains* [140, 2, 44]. In those cases, our algorithm does not build the whole deterministic automaton, but just a small part of it.

The second point is raised for instance with the automata required for deciding Kleene algebra with tests (Section 1.5). We propose in Section 2.3 to use symbolic automata, where the transition function is represented in a compact way using binary decision diagrams (BDD) [34, 35]. The key idea consists in exploring reachable pairs symbolically, so as to avoid redundancies. This idea can be combined with existing optimisations, and we show in particular a nice integration with the disjoint sets forest data-structure from Hopcroft and Karp's algorithm.

2.1 Deterministic automata

We start by recalling Hopcroft and Karp's algorithm [66], showing that it exploits an instance of what is nowadays called a *coinduction proof principle* [92, 127, 123].

In Section 2.3 we will need to work with Moore machines [95] rather than automata: the accepting status of a state is not necessarily a Boolean, but a value in a fixed yet arbitrary set. Since this generalisation is harmless, we stick to the standard automata terminology.

A deterministic finite automaton (DFA) over the alphabet Σ and with outputs in B is a triple (S, o, t), where S is a finite set of states, $o\colon S \to B$ is the output function, and $t\colon S \to S^\Sigma$ is the (total) transition function which returns, for each state x and for each letter $a \in \Sigma$, the next state $t_a(x)$. For $w \in \Sigma^\star$, we denote by $x \xrightarrow{w} x'$ the least relation such that (1) $x \xrightarrow{\epsilon} x$ and (2) $x \xrightarrow{au} x'$ if $x \xrightarrow{a} x''$ and $x'' \xrightarrow{u} x'$ for some x''.

The *language* of a state $x \in S$ of a DFA is the function $[x]\colon \Sigma^\star \to B$ defined as follows:

$$[x](\epsilon) = o(x) \ , \qquad [x](aw) = [t_a(x)](w) \ .$$

(When the output set is 2, these functions are indeed characteristic functions of formal languages). Two states $x, y \in S$ are said to be *language equivalent* (written $x \sim y$) when they accept the same language.

Throughout the chapter, we consider a fixed automaton (S, o, t) and we study the following problem: given two states in S, is it the case that

they are language equivalent? This problem generalises the familiar problem of checking whether two automata accept the same language: just take the union of the two automata as the automaton (S, o, t), and determine whether their respective starting states are language equivalent.

2.1.1 Language equivalence via coinduction

We first define the notion of bisimulation. We make explicit the underlying notion of progression, which we need in the sequel.

Definition 2.1.1 (Progression, Bisimulation)**.** *Given two relations $R, R' \subseteq S^2$ on states, R* progresses to *R', denoted $R \rightarrowtail R'$, if whenever $x\ R\ y$ then*

1. *$o(x) = o(y)$ and*
2. *for all $a \in \Sigma$, $t_a(x)\ R'\ t_a(y)$.*

A bisimulation *is a relation R such that $R \rightarrowtail R$.*

Proposition 2.1.2 (Coinduction)**.** *Two states are language equivalent iff there exists a bisimulation that relates them.*

Bisimulation is thus a sound and complete proof technique for checking language equivalence of DFA. Accordingly, we obtain the simple algorithm described in Figure 2.1.

This algorithm works as follows: the variable `R` contains a relation which is a bisimulation candidate and the variable `todo` contains a queue of pairs that remain to be processed. To process a pair (x, y), one first checks whether it already belongs to the bisimulation candidate: in that case, the pair can be skipped since it was already processed. Otherwise, one checks that the outputs of the two states are the same (line 11), and one pushes all derivatives of the pair to the `todo` queue (line 12—this requires the type Σ of letters to be iterable, and thus finite, an assumption which is no longer required with the symbolic algorithm to be presented in Section 2.3.2). The pair (x, y) is finally added to the bisimulation candidate, and we proceed with the remainder of the queue.

Proposition 2.1.3. *For all $x, y \in S$, $x \sim y$ iff* `Naive`(x, y).

Proof. The main invariant of the loop (line 8: `R` $\rightarrowtail$ `R` $\cup$ `todo`) ensures that when `todo` becomes empty, then `R` contains a bisimulation. Another invariant is that R contains the starting states. Therefore, thanks to Proposition 2.1.2, if the algorithm answers true, then the starting states were indeed language equivalent.

A third invariant of the loop is that for any pair (x', y') in `todo`, there exists a word w such that $x \xrightarrow{w} x'$ and $y \xrightarrow{w} y'$. Therefore, if we reach a pair of states whose outputs are distinct—line 11, then the word w associated to that pair witnesses the fact that the two initial states are not equivalent. □

```
1 val o: α → β
2 val t: α → Σ → α
3
4 let Naive(x,y) =
5   let R = Set.empty () in
6   let todo = Queue.singleton (x,y) in
7   while Queue.not_empty todo do
8     (* invariant: R ⟼ R ∪ todo *)
9     let (x,y) = Queue.pop todo in
10    if Set.mem (x,y) R then continue
11    if o x ≠ o y then return false
12    forall a ∈ Σ do Queue.push todo (t x a, t y a)
13    Set.add (x,y) R
14  done
15  return true
```

Figure 2.1: Naive algorithm for checking language equivalence. An abstract view of Hopcroft and Karp's algorithm HK(x, y) is obtained by replacing the test in step 10 with Set.mem (x,y) e(R).

For a concrete example, consider the DFA with input alphabet $\Sigma = \{a\}$ and outputs in $B = 2$ in the left-hand side of Figure 2.2. Accepting states are overlined; suppose we want to check that x and u are language equivalent. During the initialisation, (x, u) is inserted in todo. At the first iteration, since $o(x) = 0 = o(u)$, (x, u) is inserted in R and (y, v) in todo. At the second iteration, since $o(y) = 1 = o(v)$, (y, v) is inserted in R and (z, w) in todo. At the third iteration, since $o(z) = 0 = o(w)$, (z, w) is inserted in R and (y, v) in todo. At the fourth iteration, since (y, v) is already in R, the algorithm does nothing. Since there are no more pairs to check in todo, the relation R is a bisimulation and the algorithm terminates returning true.

These iterations are concisely described by the numbered dashed lines in Figure 2.2. The line i means that the connected pair is inserted in R at iteration i. (In the sequel, when enumerating iterations, we ignore those where a pair from todo is already in R so that there is nothing to do.)

2.1.2 Hopcroft and Karp's algorithm

The naive algorithm is quadratic: a new pair is added to R at each non-trivial iteration, and there are only n^2 such pairs, where $n = |S|$ is the number of states of the DFA. To make this algorithm (almost) linear, Hopcroft and Karp actually record a set of *equivalence classes* rather than a set of visited pairs. As a consequence, their algorithm may stop earlier, if it encounters a pair of states that is not already in R but belongs to its reflexive, sym-

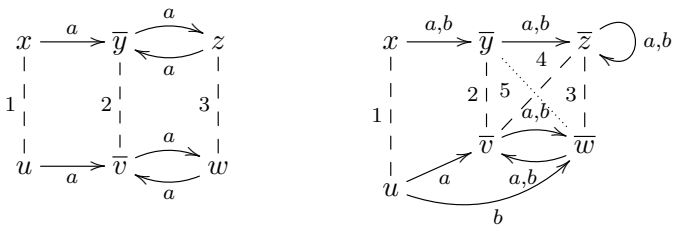


Figure 2.2: Checking for language equivalence in a DFA.

metric, and transitive closure. For instance, in the right-hand side example from Figure 2.2, we can stop when we encounter the dotted pair (y, w) since these two states already belong to the same equivalence class according to the four previous pairs.

With this optimisation, the produced relation `R` contains at most n pairs. Formally, ignoring the concrete data structure used to store equivalence classes, Hopcroft and Karp's algorithm consists in replacing line 10 in Figure 2.1 with

```
if Set.mem (x,y) e(R) then continue
```

where $e\colon \mathcal{P}(S^2) \to \mathcal{P}(S^2)$ is the function mapping a relation $R \subseteq S^2$ into its symmetric, reflexive, and transitive closure. We refer to this algorithm as `HK`.

2.1.3 Bisimulations up to equivalence

We now show that the optimisation used by Hopcroft and Karp corresponds to exploiting an "up-to technique". Let us consider the right-hand side example from Figure 2.2. $\mathsf{HK}(x, u)$ constructs the following relation, represented with dashed lines.

$$R_{\mathsf{HK}} = \{(x, u), (y, v), (z, w), (z, v)\}$$

This relation is not a bisimulation: it contains the pair (x, u), whose b-transitions lead to (y, w), which is not in R_{HK}. Instead, this pair belongs to $e(R_{\mathsf{HK}})$; the candidate R_{HK} is only a bisimulation up to e:

Definition 2.1.4 (Bisimulation up-to)**.** *Let $f\colon \mathcal{P}(S^2) \to \mathcal{P}(S^2)$ be a function on relations. A relation R is a* bisimulation up to f *if $R \rightarrowtail f(R)$, i.e., if $x\ R\ y$, then*

1. *$o(x) = o(y)$ and*
2. *for all $a \in \Sigma$, $t_a(x)\ f(R)\ t_a(y)$.*

With this definition, Hopcroft and Karp's algorithm consists in trying to build a bisimulation up to equivalence closure (e). To prove the correctness of the algorithm, it suffices to show

Theorem 2.1.5. *Every bisimulation up to e is contained in a bisimulation.*

This theorem is not especially difficult, we prove it in Chapter 4.

Corollary 2.1.6. *For all $x, y \in S$, $x \sim y$ iff* $\mathsf{HK}(x, y)$.

Proof. As for Proposition 2.1.3, by using the invariant $R \rightarrowtail e(R) \cup todo$. We deduce that R is a bisimulation up to e after the loop. We conclude with Theorem 2.1.5 and Proposition 2.1.2. □

An informal complexity analysis of HK goes as follows. One can enter the main loop at most $n = |S|$ times: at the beginning $e(R)$ is a discrete partition with n equivalence classes, and we merge two equivalence classes at each iteration. The inner iteration (line 12) costs $k = |\Sigma|$ operations.

To obtain the complexity in $O(nk\alpha(k, n)))$, one has to represent the equivalence relation $e(R)$ using a *disjoint set forest* data-structure [66]: this makes it possible to implement the test on line 10 and the update on line 13 in almost constant amortised time [134].

2.2 Non-deterministic automata

We now move from DFA to non-deterministic automata (NFA). A NFA over the alphabet Σ is a triple (S, o, t), where S is a finite set of states, $o\colon S \to B$ is the output function, and $t\colon S \to \mathcal{P}(S)^\Sigma$ is the transition relation: it assigns to each state $x \in S$ and letter $a \in \Sigma$ a set of possible successors.

The *powerset construction* transforms any NFA (S, o, t) into the DFA $(\mathcal{P}(S), o^\sharp, t^\sharp)$ where $o^\sharp\colon \mathcal{P}(S) \to B$ and $t^\sharp\colon \mathcal{P}(S) \to \mathcal{P}(S)^\Sigma$ are defined for all $X \in \mathcal{P}(S)$ and $a \in \Sigma$ as follows:

$$o^\sharp(X) = \sum_{x \in X} o(x) \qquad\qquad t^\sharp_a(X) = \sum_{x \in X} t_a(x)$$

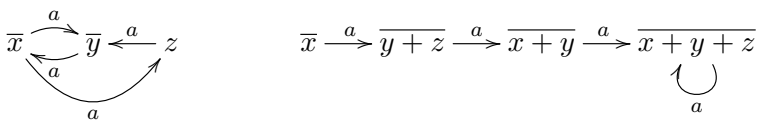
(For $o^\sharp$ to be properly defined, we need B to be a semilattice; we use the sum symbol to denote both the corresponding operation on B and set-theoretic union on $\mathcal{P}(S)$.) By definition, $o^\sharp$ and $t^\sharp$ are semi-lattice homomorphisms. These properties are fundamental for the up-to technique we are going to introduce. In order to stress the difference with generic DFA, which usually do not carry this structure, we use the following definition.

Definition 2.2.1. *A* determinised NFA *is a DFA $(\mathcal{P}(S), o^\sharp, t^\sharp)$ obtained via the powerset construction of some NFA (S, o, t).*

```
1  val o: α → β
2  val t: α → Σ → α set
3
4  let Naive(X,Y) =
5    let R = Set.empty () in
6    let todo = Queue.singleton (X,Y) in
7    while Queue.not_empty todo do
8      (* invariant: R ⟼ R ∪ todo *)
9      let (X,Y) = Queue.pop todo in
10     if Set.mem (X,Y) R then continue
11     if o♯ X ≠ o♯ Y then return false
12     forall a ∈ Σ do Queue.push todo (t♯ X a, t♯ Y a)
13     Set.add (X,Y) R
14   done
15   return true
```

Figure 2.3: On-the-fly naive algorithm, for checking the equivalence of sets of states X and Y of an NFA (S, o, t). `HK`(X, Y) is obtained by replacing the test in line 10 with `Set.mem (X',Y')` e(`R`), and `HKC`(X, Y) is obtained by replacing it with `Set.mem (X',Y')` c(`R`$\cup$`todo`).

Hereafter, we use a lighter notation for representing states of determinised NFA: in place of the singleton $\{x\}$, we write x and, in place of $\{x_1, \ldots, x_n\}$, we write $x_1 + \cdots + x_n$ (thus 0 for $\emptyset$). Consider for instance the NFA (S, o, t) depicted below (left) and part of the determinised NFA $(\mathcal{P}(S), o^\sharp, t^\sharp)$ (right).



In the determinised NFA, x makes one single a-transition into $y + z$. This state is final: $o^\sharp(y + z) = o^\sharp(y) + o^\sharp(z) = o(y) + o(z) = 1 + 0 = 1$; it makes an a-transition into $t^\sharp_a(y + z) = t^\sharp_a(y) + t^\sharp_a(z) = t_a(y) + t_a(z) = x + y$.

Algorithms for NFA can be obtained by computing the determinised NFA on-the-fly [46]: starting from the algorithms for DFA (Figure 2.1), it suffices to work with sets of states, and to inline the powerset construction. The corresponding code is just the composition of the previous algorithm with the powerset construction, we give it explicitly in Figure 2.3 for the sake of clarity. The naive algorithm (`Naive`) does not use any up to technique, Hopcroft and Karp's algorithm (`HK`) reasons up to equivalence at line 10.

2.2.1 Bisimulations up to context

The semi-lattice structure $(\mathcal{P}(S), +, 0)$ carried by determinised NFA makes it possible to introduce a new up-to technique, which is not available with plain DFA: *up to context*. This technique is grounded on a simple observation on determinised NFA: for all sets X and Y of states of the original NFA, the union of the language recognised by X and the language recognised by Y is equal to the language recognised by the union of X and Y. In symbols:

$$[X + Y] = [X] + [Y] \tag{2.1}$$

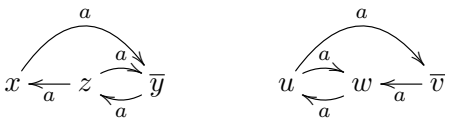
Therefore, if a bisimulation R relates the set of states X_1 with Y_1, and X_2 with Y_2, then $[X_1] = [Y_1]$ and $[X_2] = [Y_2]$ and we can immediately conclude by (2.1) that $X_1 + X_2$ and $Y_1 + Y_2$ are language equivalent as well.

Definition 2.2.2 (Context closure). *Let $u \colon \mathcal{P}(\mathcal{P}(S)^2) \to \mathcal{P}(\mathcal{P}(S)^2)$ be the function mapping a relation R on sets of states to the smallest relation which contains R and which is compatible with union:*

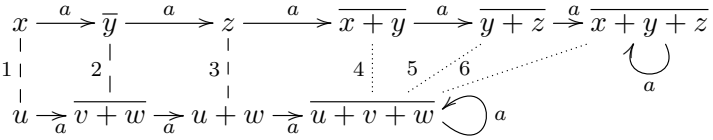
$$\frac{X_1 \; u(R) \; Y_1 \qquad X_2 \; u(R) \; Y_2}{X_1 + X_2 \; u(R) \; Y_1 + Y_2} \quad .$$

Proposition 2.2.3. *Every bisimulation up to u is contained in a bisimulation.*

To illustrate this idea, let us check the equivalence of states x and u in the following NFA.



The determinised automaton is depicted below.



The numbered lines shows a bisimulation containing x and u. Actually, this is the relation that is built by Hopcroft and Karp's algorithm. The dashed lines (numbered by 1, 2, 3) form a smaller relation which is not a bisimulation, but a bisimulation up to context: the equivalence of $x+y$ and $u+v+w$ is deduced from the fact that x is related with u and y with $v + w$, without the need to further explore the automaton.

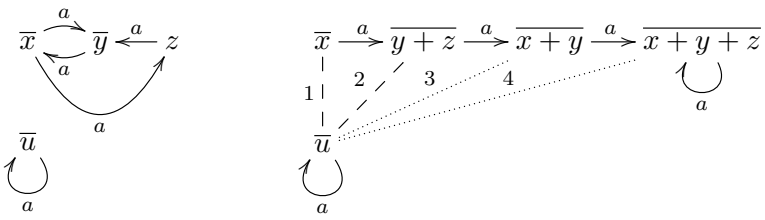


Figure 2.4: A bisimulation up to congruence.

2.2.2 Bisimulations up to congruence

The techniques of up-to equivalence and up-to context can actually be combined, resulting in a powerful proof technique which we call *bisimulation up to congruence*.

Definition 2.2.4 (Congruence closure)**.** *Let* $c\colon \mathcal{P}(\mathcal{P}(S)^2) \to \mathcal{P}(\mathcal{P}(S)^2)$ *be the function mapping a relation* R *on sets of states to the least equivalence relation which contains* R *and which is compatible with union.*

Theorem 2.2.5. *Every bisimulation up to* c *is contained in a bisimulation.*

As before for bisimulations up to equivalence, we prove this theorem in Chapter 4. An example illustrating a bisimulation up to congruence is given in Figure 2.4. The relation R expressed by the dashed numbered lines (formally $R = \{(x,\ u), (y+z,\ u)\}$) is neither a bisimulation nor a bisimulation up to equivalence or up to context since $y + z \xrightarrow{a} x + y$ and $u \xrightarrow{a} u$, but $(x + y,\ u) \notin e(R) + u(R)$. However, R is a bisimulation up to congruence. Indeed, we have $(x + y,\ u) \in c(R)$:

$$\begin{aligned} x + y \ &c(R)\ u + y && ((x,u) \in R) \\ &c(R)\ y + z + y && ((y+z,u) \in R) \\ &=\ \ y + z \\ &c(R)\ u && ((y+z,u) \in R) \end{aligned}$$

In contrast, we need four pairs to get a bisimulation up to equivalence containing (x, u): this is the relation depicted with both dashed and dotted lines in Figure 2.4.

Note that we can deduce many other equations from R; in fact, $c(R)$ defines the following partition of sets of states:

$$\{0\}, \{y\}, \{z\}, \{x, u, x+y, x+z, \text{ and the 9 remaining subsets}\}.$$

2.2.3 Optimised algorithm for NFA

The algorithm we developed with Filippo Bonchi [21], called HKC in the sequel, relies on up to congruence: line 10 from Figure 2.3 becomes

```
if Set.mem (X,Y) c(R∪todo) then continue
```

Observe that we use $c(R \cup todo)$ rather than $c(R)$: this allows us to skip more pairs (potentially exponentially many, see the discussion after Remark 2.2.7), and this is safe since all pairs in `todo` will eventually be processed.

Corollary 2.2.6. *For all $X, Y \in \mathcal{P}(S)$, $X \sim Y$ iff* HKC(X, Y).

Proof. As for Proposition 2.1.3, by using the invariant $R \rightarrowtail c(R \cup todo)$ for the loop. We deduce that R is a bisimulation up to congruence after the loop. We conclude with Theorem 2.2.5 and Proposition 2.1.2. □

The most important point about these three algorithms is that they compute the states of the determinised NFA lazily. This means that only *accessible* states need to be computed. This is of practical importance since the determinised NFA can be exponentially large. In case of a negative answer, the three algorithms stop even before all accessible states have been explored; otherwise, if a bisimulation (possibly up-to) is found, it depends on the algorithm:

- With Naive, all accessible states need to be visited, by definition of bisimulation.
- With HK, the only case where some accessible states can be avoided is when a pair (X, X) is encountered: the algorithm skips this pair so that the successors of X are not necessarily computed (this situation never happens when starting with disjoint automata). In the other cases where a pair (X, Y) is skipped, X and Y are necessarily already related with some other states in R, so that their successors will eventually be explored.
- With HKC, accessible states are often skipped. For a simple example, let us recall the execution of HKC on the NFA from Figure 2.4. After two iterations, $R = \{(x, u), (y + z, u)\}$. Since $x + y \ c(R) \ u$, the algorithm stops without building the states $x + y$ and $x + y + z$. Similarly, in the example from the Introduction, HKC does not construct the four states corresponding to pairs 4, 5, and 6.

This ability of HKC to ignore parts of the determinised NFA can bring an exponential speed-up. For an example, consider the family of NFA in Figure 2.5, where n is an arbitrary natural number. Taken together, the states

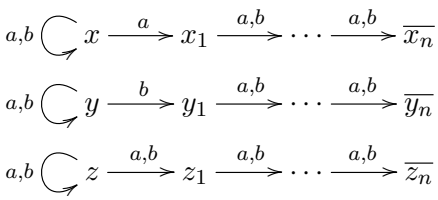


Figure 2.5: Family of examples where HKC exponentially improves over AC and HK; we have $x + y \sim z$.

x and y are equivalent to z: they recognise the language $(a{+}b)^\star(a{+}b)^{n+1}$. Alone, x recognises the language $(a{+}b)^\star a(a{+}b)^n$, which is known for having a minimal DFA with 2^n states. Therefore, checking $x + y \sim z$ via minimisation (as in [64]) requires exponential time, and the same holds for Naive and HK since all accessible states must be visited. This is not the case with HKC, which requires only polynomial time on this example. Indeed, HKC$(x{+}y, z)$ builds the relation

$$\begin{aligned} R' = &\{(x + y,\ z)\} \\ &\cup \{(x + Y_i + y_{i+1},\ Z_{i+1}) \mid i < n\} \\ &\cup \{(x + Y_i + x_{i+1},\ Z_{i+1}) \mid i < n\}\ , \end{aligned}$$

where $Y_i = y{+}y_1{+}\ldots{+}y_i$, and $Z_i = z{+}z_1{+}\ldots{+}z_i$. R' only contains $2n + 1$ pairs and is a bisimulation up to congruence. To see this, consider the pair $(x{+}y{+}x_1{+}y_2,\ Z_2)$ obtained from $(x{+}y,\ z)$ after reading the word ba. Although this pair does not belong to R', it belongs to its congruence closure:

$$\begin{aligned} x{+}y{+}x_1{+}y_2 \ c(R')\ Z_1{+}y_2 \qquad & (x{+}y{+}x_1\ R'\ Z_1) \\ c(R')\ x{+}y{+}y_1{+}y_2 \qquad & (x{+}y{+}y_1\ R'\ Z_1) \\ c(R')\ Z_2\ . \qquad & (x{+}y{+}y_1{+}y_2\ R'\ Z_2) \end{aligned}$$

Remark 2.2.7. *In the above derivation, the use of transitivity is crucial: R' is a bisimulation up to congruence, but not a bisimulation up to context. In fact, there exists no bisimulation up to context of linear size proving $x + y \sim z$.*

We now discuss the exploration strategy, i.e., how to choose the pair to extract from the set todo in step 3.1. When looking for a counter-example, such a strategy has a large influence: a good heuristic can help in reaching it directly, while a bad one might lead to explore exponentially many pairs first. In contrast, the strategy is not so relevant when looking for an equivalence proof (when the algorithm eventually returns true). Actually, one can prove that the number of steps performed by Naive and HK in such a case does not depend on the strategy. This is not the case with HKC: the strategy

can induce some differences. However, we experimentally observed that breadth-first and depth-first strategies usually behave similarly on random automata. This behaviour is due to the fact that we check congruence w.r.t. $R \cup \mathit{todo}$ rather than just R (line 10): with this optimisation, the example above is handled in polynomial time whatever the chosen strategy. In contrast, without this small optimisation, it requires exponential time with a depth-first strategy.

2.2.4 Computing the congruence closure

For the optimised algorithm to be effective, we need a way to check whether some pairs belong to the congruence closure of a given relation. We present a simple solution based on set rewriting; the key idea is to look at each pair (X, Y) in a relation R as a pair of rewriting rules:

$$X \to X + Y \qquad\qquad Y \to X + Y \ ,$$

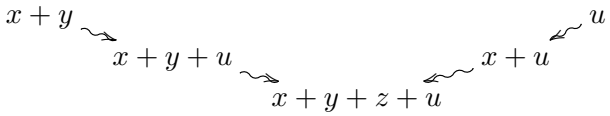
which can be used to compute normal forms for sets of states. Indeed, by idempotence, $X\ R\ Y$ entails $X\ c(R)\ X + Y$.

Definition 2.2.8. *Let $R \subseteq \mathcal{P}(S)^2$ be a relation on sets of states. We define $\rightsquigarrow_R \subseteq \mathcal{P}(S)^2$ as the smallest irreflexive relation that satisfies the following rules:*

$$\frac{X\ R\ Y}{X \rightsquigarrow_R X + Y} \qquad \frac{X\ R\ Y}{Y \rightsquigarrow_R X + Y} \qquad \frac{Z \rightsquigarrow_R Z'}{U + Z \rightsquigarrow_R U + Z'}$$

Lemma 2.2.9. *For all relations R, $\rightsquigarrow_R$ is confluent and normalising.*

In the sequel, we denote by $X{\downarrow_R}$ the normal form of a set X w.r.t. $\rightsquigarrow_R$. Intuitively, the normal form of a set is the largest set of its equivalence class. Recalling the example from Figure 2.4, the common normal form of $x + y$ and u can be computed as follows (R is the relation $\{(x, u), (y + z, u)\}$):



Theorem 2.2.10. *For all relations R, and for all $X, Y \in \mathcal{P}(S)$, we have $X{\downarrow_R} = Y{\downarrow_R}$ iff $(X, Y) \in c(R)$.*

We actually have $X{\downarrow_R} = Y{\downarrow_R}$ iff $X \subseteq Y{\downarrow_R}$ and $Y \subseteq X{\downarrow_R}$, so that the normal forms of X and Y do not necessarily need to be fully computed in practice. Still, the worst-case complexity of this sub-algorithm is quadratic in the size of the relation R (assuming we count the number of operations on sets: unions and inclusion tests—the difficulty which makes the algorithm quadratic being that we do not know in advance in which order to apply the rewriting rules).

Note that many algorithms were proposed in the literature to compute the congruence closure of a relation (see, e.g., [98, 130, 14]). However, they usually consider uninterpreted symbols or associative and commutative symbols, but not associative, commutative, and idempotent symbols, which is what we need here.

2.2.5 Using HKC for checking language inclusion

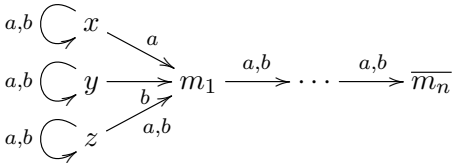
For NFA, language inclusion can be reduced to language equivalence: the semantics function $[-]$ is a semi-lattice homomorphism, so that for all sets of states X, Y, $[X{+}Y] = [Y]$ iff $[X] + [Y] = [Y]$ iff $[X] \subseteq [Y]$. Therefore, it suffices to run $\mathtt{HKC}(X{+}Y, Y)$ to check the inclusion $[X] \subseteq [Y]$.

One might wonder whether checking the two inclusions separately is more convenient than checking the equivalence directly. This is not the case: checking the equivalence directly actually allows one to skip some pairs that cannot be skipped when reasoning by double inclusion. As an example, consider the DFA on the right of Figure 2.2. The relation computed by $\mathtt{HKC}(x, u)$ contains only four pairs (because the fifth one follows from transitivity). Instead, the relations built by $\mathtt{HKC}(x, x{+}u)$ and $\mathtt{HKC}(u{+}x, u)$ would both contain five pairs: transitivity cannot be used since our relations are now oriented (from $y \leq v$, $z \leq v$ and $z \leq w$, we cannot deduce $y \leq w$). Figure 2.5 shows another example, where we get an exponential factor by checking the equivalence directly rather than through the two inclusions: transitivity, which is crucial to keep the relation computed by $\mathtt{HKC}(x{+}y, z)$ small—see Remark 2.2.7, cannot be used when checking the two inclusions separately.

In a sense, the behaviour of the coinductive proof here is similar to that of standard proofs by induction, where one often has to strengthen the induction predicate to get a (nicer) proof.

2.2.6 Exploiting similarity

Looking at the example in Figure 2.5, a natural idea would be to first quotient the automaton by graph isomorphism. By doing so, one would merge the states x_i, y_i, z_i and obtain the following automaton, for which checking $x{+}y \sim z$ is much easier.



As shown in [2, 44] for antichain algorithms, one can do more than graph isomorphism, by exploiting any preorder contained in language in-

clusion. Hereafter, we show how this idea can be embedded in HKC, resulting in an even stronger algorithm.

For the sake of clarity, we fix the preorder to be *similarity* [92], which can be computed in quadratic time [61]. (Graph isomorphism can be computed slightly more efficiently, but it is less powerful as an up-to technique.)

Definition 2.2.11 (Similarity)**.** *Let* similarity *be the largest relation $\preceq$ on states of an NFA such that $x \preceq y$ entails:*

1. *$o(x) \leq o(y)$ and*
2. *for all $a \in \Sigma, x' \in S$ such that $x \xrightarrow{a} x'$, there exists some y' such that $y \xrightarrow{a} y'$ and $x' \preceq y'$.*

To exploit similarity pairs in HKC, it suffices to notice that for any similarity pair $x \preceq y$, we have $x+y \sim y$. Let $\overline{\preceq}$ denote the relation $\{(x+y,\ y) \mid x \preceq y\}$, and let c' be the function mapping a relation R to $c(R \cup \overline{\preceq})$. The theory from Chapter 4 will allow us to deduce

Theorem 2.2.12. *Every bisimulation up to c' is contained in a bisimulation.*

Accordingly, we call HKC' the algorithm obtained from HKC (Figure 2.3) by replacing the test $(X, Y) \in c(R \cup todo)$ with $(X, Y) \in c'(R \cup todo)$. (This test can be implemented efficiently by preprocessing the NFA.)

Corollary 2.2.13. *For all sets X, Y, $X \sim Y$ iff HKC'(X, Y).*

2.2.7 Antichain algorithms

The problem of deciding NFA equivalence is PSPACE-complete [91]. However, neither HKC nor HKC' are in PSPACE: both of them keep track of the states they explored in the determinised NFA, and there can be exponentially many such states. This also holds for HK and for the more recent antichain algorithm [140] (called AC in the following) and its optimisation (AC') exploiting similarity [2, 44].

The latter algorithms can be explained in terms of coinductive proof techniques: we establish in [21] that they actually construct bisimulations up to context, i.e., bisimulations up to congruence for which one does not exploit *symmetry* and *transitivity*.

Theoretical comparison We compared the various algorithms in detail in [21]. Their relationship is summarised in Figure 2.6, where an arrow X→Y means that (a) Y can explore exponentially fewer states than X, and (b) Y can mimic X, i.e., the coinductive proof technique underlying Y is at least as powerful as the one of X.

In the general case, AC needs to explore many more states than HKC: the use of transitivity, which is missing in AC, allows HKC to drastically prune

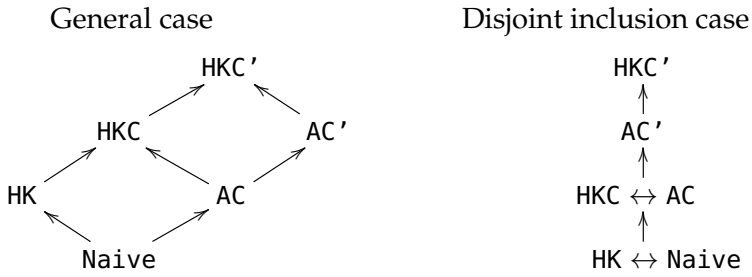


Figure 2.6: Relationships among the algorithms.

the exploration. For instance, to check $x+y \sim z$ in Figure 2.5, HKC only needs a linear number of states (see Remark 2.2.7), while AC needs exponentially many states. In contrast, in the special case where one checks for the inclusion of disjoint automata, HKC and AC exhibit the same behaviour. Indeed, HKC cannot make use of transitivity in such a situation, as explained in Section 2.2.5. Things change when comparing HKC' and AC': even for checking inclusion of disjoint automata, AC' cannot always mimic HKC': the use of similarity tends to virtually merge states, so that HKC' can use the up to transitivity technique which AC' lacks.

Experimental comparison The theoretical relationships we have drawn in Figure 2.6 are substantially confirmed by an empirical evaluation of the performance of the algorithms. Here, we only give a brief overview; see [21] for a complete description of those experiments.

We compared our OCaml implementation [108] for HK, HKC and HKC', and the `libvata` C++ library [87] for AC and AC'. We use a breadth-first exploration strategy: we represent the set `todo` from Figure 2.3 as a FIFO queue. As mentioned at the end of Section 2.2.3, considering a depth-first strategy here does not alter the behaviour of HKC in a noticeable way.

We performed experiments using both random automata and a set of automata arising from model-checking problems.

• *Random automata.* We used Tabakov and Vardi's model [133] to generate 1000 random NFA with two letters and a given number of states. We executed all algorithms on these NFA, and we measured the number of processed pairs, i.e., the number of required iterations (like HKC, AC is a loop inside which pairs are processed). We observe that HKC improves over AC by one order of magnitude, and AC improves over HK by two orders of magnitude. Using up-to similarity (HKC' and AC') does not improve performance much; in fact, similarity is almost the identity relation on such random automata. The corresponding distributions for HK, HKC, and AC are plotted on Figure 2.7, for automata with 100 states. Note that while HKC only improves

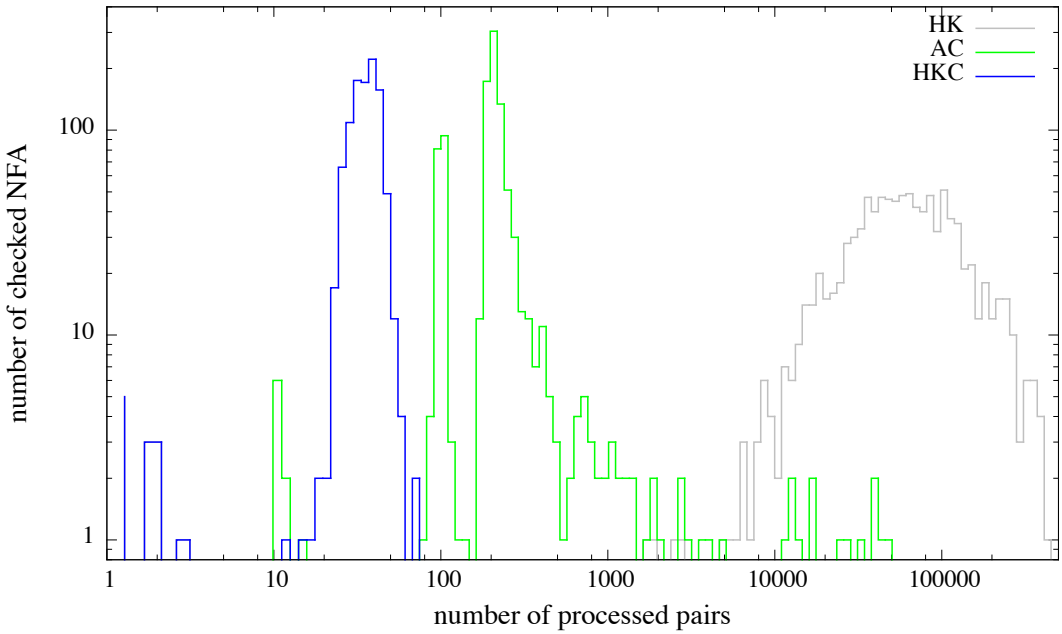


Figure 2.7: Distributions of the number of processed pairs, for a thousand experiments with random NFA.

by one order of magnitude over `AC` when considering the average case, it improves by several orders of magnitude when considering the worst cases.

• *Model checking automata*. Abdulla et al. [2, 44] used automata sequences arising from regular model-checking experiments [24] to compare their algorithm (`AC'`) against `AC`. We reused these sequences to test `HKC'` against `AC'` in a concrete scenario. For all those sequences, we checked the inclusions of all consecutive pairs, in both directions. The timings are given in Table 2.1, where we report the median values (50%), the last deciles (90%), the last percentiles (99%), and the maximum values (100%). We distinguish between the experiments for which a counter-example was found, and those for which the inclusion did hold. For `HKC'` and `AC'`, we display the time required to compute similarity on a separate line: this preliminary step is shared by the two algorithms. As expected, `HKC` and `AC` roughly behave the same: we test inclusions of disjoint automata. `HKC'` is however quite faster than `AC'`: up to transitivity can be exploited thanks to similarity pairs. Also note that over the 546 positive answers, 368 are obtained immediately by similarity.

algorithm	inclusions (546 pairs)				counter-examples (518 pairs)			
	50%	90%	99%	100%	50%	90%	99%	100%
AC	0.036	0.860	4.981	5.084	0.009	0.094	1.412	2.887
HKC	0.049	0.798	6.494	6.762	0.000	0.014	0.916	2.685
sim_time	0.039	0.185	0.574	0.618	0.038	0.193	0.577	0.593
AC'-sim_time	0.013	0.167	1.326	1.480	0.012	0.107	1.047	1.134
HKC'-sim_time	0.000	0.034	0.224	0.345	0.001	0.005	0.025	0.383

Table 2.1: Timings, in seconds, for language inclusion of disjoint NFA generated from model-checking.

2.2.8 Discussion

Our implementation of HKC is available online [108], together with proofs mechanised in the Coq proof assistant and an interactive applet making it possible to test the presented algorithms online, on user-provided examples.

Complexity. The previous algorithms and those based on antichains have exponential complexity in the worst case while they behave rather well in practice. For instance, in Figure 2.7, one can notice that over a thousand random automata, very few require to explore a large amount of pairs. This suggests that an accurate analysis of the average complexity might be promising. An inherent problem comes from the difficulty to characterise the average shape of determinised NFA [133]. To avoid this problem, with HKC, we could try to focus on the properties of congruence relations. For instance, given a number of states, how long can be a sequence of (incrementally independent) pairs of sets of states whose congruence closure collapses into the full relation? (This number is an upper-bound for the size of the relations produced by HKC.) One can find ad-hoc examples where this number is exponential, but we suspect it to be rather small in average.

Model checking. The experiments summarised in Table 2.1 show the efficiency of our approach for regular model-checking using automata on finite words.

In order to face other model-checking problems, it would be useful to extend up-to techniques to automata on infinite words, or trees. Unfortunately, the determinisation of these automata (the so called Safra's construction) does not seem suitable for exploiting either antichains or up to congruence. However, for some problems like LTL realisability [47] that can be solved without prior determinisation (the so-called Safraless approaches), antichains have been crucial in obtaining efficient procedures. We leave as future work to explore whether up-to techniques could further improve such procedures.

2.3 Automata with a large alphabet

In Section 1.5 we have recalled a completeness result by Kozen and Cohen (Theorem 1.5.1), making it possible to reduce the equational theory of Kleene algebra with tests to a problem of language equivalence. Unfortunately, the corresponding alphabet is exponentially large in the number of primitive tests. As such, it renders standard algorithms for language equivalence intractable, even for reasonably small inputs. This difficulty is shared with other fields where various people proposed to work with *symbolic automata* to cope with large, potentially infinite, alphabets [35, 139]. By symbolic automata, we mean finite automata whose transition function is represented using a compact data-structure, typically binary decision diagrams (BDDs) [34, 35], allowing one to explore the automata in a symbolic way.

D'Antoni and Veanes proposed a minimisation algorithm for symbolic automata [43], which is much more efficient than the adaptations of the traditional algorithms [95, 65, 99]. Here we focus on the simpler problem of language equivalence: while language equivalence can be solved by minimisation, minimisation has complexity $n \ln n$ where Hopcroft and Karp's algorithm is almost linear (for DFA—see the discussion at the end of Section 2.1).

In this section we describe a simple coinductive algorithm for checking language equivalence of symbolic automata (Section 2.3.2). This algorithm is generic enough to support the various improvements discussed in the previous section. In the case of up-to equivalence, we show how to combine binary decisions diagrams and *disjoint set forests*, the efficient data-structure used in Hopcroft and Karp's algorithm. This results in a new version of their algorithm, for symbolic automata (Section 2.3.4).

2.3.1 Binary decision diagrams

Assume an ordered set $(A, <)$ and an arbitrary set B. Binary decision diagrams are directed acyclic graphs that can be used to represent functions of type $2^A \to B$. When $B = 2$ is the two-elements set, BDDs thus intuitively represent Boolean formulas with variables in A.

Formally, a *(multi-terminal, ordered) binary decision diagram* (BDD) is a pair (N, c) where N is a finite set of nodes and c is a function of type $N \to B \uplus (A \times N \times N)$ such that if $c(n) = (a, l, r)$ and either $c(l) = (a', \_, \_)$ or $c(r) = (a', \_, \_)$, then $a < a'$.

The condition on c ensures that the underlying graph is acyclic, which makes it possible to associate a function $\lceil n \rceil \colon 2^A \to B$ to each node n of a

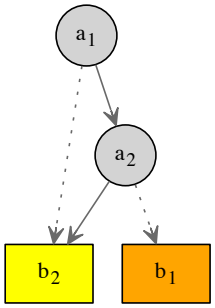
BDD:

$$\lceil n \rceil(\alpha) = \begin{cases} b & \text{if } c(n) = b \in B \\ \lceil l \rceil(\alpha) & \text{if } c(n) = (a, l, r) \text{ and } \alpha(a) = 0 \\ \lceil r \rceil(\alpha) & \text{if } c(n) = (a, l, r) \text{ and } \alpha(a) = 1 \end{cases}$$

Let us now recall the standard graphical representation of BDDs:

- A node n such that $c(n) = b \in B$ is represented by a square box labelled by b.
- A node n such that $c(n) = (a, l, r) \in A \times N \times N$ is a decision node, which we picture by a circle labelled by a, with a dotted arrow towards the *left child* (l) and a plain arrow towards the *right child* (r).

For instance, the following drawing represents a BDD with four nodes; its top-most node denotes the function given on the right-hand side.



$$\alpha \mapsto \begin{cases} b_1 & \text{if } \alpha(a_1) = 1 \text{ and } \alpha(a_2) = 0 \\ b_2 & \text{otherwise} \end{cases}$$

A BDD is *reduced* if c is injective, and $c(n) = (a, l, r)$ entails $l \neq r$. (The above example BDD is reduced.) Any BDD can be transformed into a reduced one. When A is finite, reduced (ordered) BDD nodes are in one-to-one correspondence with functions from 2^A to B [34, 35]. The main interest in this data-structure is that it is often extremely compact.

In the sequel, we only work with reduced ordered BDDs, which we simply call BDDs. We denote by $\mathrm{BDD}_A[B]$ the set of nodes of a BDD with values in B, which is large enough to represent all considered functions. We moreover let $\lfloor f \rfloor$ denote the unique BDD node representing a given function $f : 2^A \to B$. This notation is useful to give abstract specifications to BDD operations: in the sequel, all usages of this notation actually underpin efficient BDD operations.

```
type β node = β descr hash_consed
and  β descr = V of β | N of A × β node × β node

val hashcons: β descr → β node
val c: β node → β descr
val memo_rec: ((α'→β'→γ)→α'→β'→γ)→α'→β'→γ
 (* with α' = α hash_consed, β' = β hash_consed *)

let constant v = hashcons (V v)
let node a l r = if l==r then l else hashcons (N(a,l,r))

let apply (f: α → β → γ): α node → β node → γ node =
 memo_rec (fun app x y →
 match c(x), c(y) with
   | V v, V w → constant (f v w)
   | N(a,l,r), V _ → node a (app l y) (app r y)
   | V _, N(a,l,r) → node a (app x l) (app x r)
   | N(a,l,r), N(a',l',r') →
     if a=a' then node a  (app l l') (app r r')
     if a<a' then node a  (app l y ) (app r y )
     if a>a' then node a' (app x l') (app x r'))
```

Figure 2.8: An implementation of BDDs.

Implementation. To better explain parts of the proposed algorithms, we give a simple implementation of BDDs in Figure 2.8.

The type for BDD nodes is given first: we use Filliâtre's hash-consing library [48] to enforce unique representation of each node, whence the two type declarations and the two conversion functions `hashcons` and `c` between those types. The third utility function `memo_rec` is just a convenient operator for defining recursive memoised functions on pairs of hash-consed values.

The function `constant` creates a constant node, making sure it was not already created. The function `node` creates a new decision node, unless that node is useless and can be replaced by one of its two children. The generic function `apply` is central to BDDs [34, 35]: many operations are just instances of this function. Its specification is the following:

$$\texttt{apply}\ f\ x\ y = \lfloor \alpha \mapsto f(\lceil x \rceil(\alpha))(\lceil y \rceil(\alpha)) \rfloor$$

This function is obtained by "zipping" the two BDDs together until a constant is reached. Memoisation is used to exploit sharing and to avoid performing the same computations again and again.

Suppose now that we want to define logical disjunction on Boolean BDD nodes. Its specification is the following:

$$x \vee y = \lfloor \alpha \mapsto \lceil x \rceil(\alpha) \vee \lceil y \rceil(\alpha) \rfloor.$$

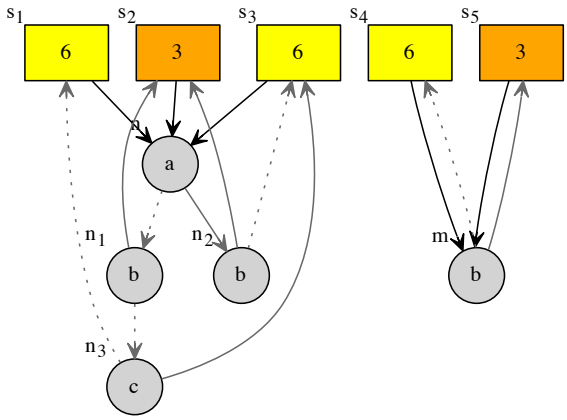
We can thus simply use the `apply` function, applied to the Boolean disjunction function:

```
let dsj: bool node → bool node → bool node = apply (∨)
```

Note that this definition could actually be slightly optimised by inlining `apply`'s code, and noticing that the result is already known whenever one of the two arguments is a constant:

```
let dsj: bool node → bool node → bool node =
 memo_rec (fun dsj x y →
 match c(x), c(y) with
  | V true, _ | _, V false → x
  | _, V true | V false, _ → y
  | N(a,l,r), N(a',l',r') →
    if a=a' then node a (dsj l l') (dsj r r')
    if a<a' then node a (dsj l y ) (dsj r y )
    if a>a' then node a' (dsj x l') (dsj x r'))
```

We ignore such optimisations in the sequel, for the sake of clarity.



	s_1, s_2, s_3								s_4, s_5							
a	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
b	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
c	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
t	s_1	s_3	s_2	s_2	s_3	s_3	s_2	s_2	s_4	s_4	s_5	s_5	s_4	s_4	s_5	s_5

Figure 2.9: A symbolic DFA with five states.

2.3.2 Symbolic automata

A standard technique [35, 60, 139, 43] for working with automata over a large input alphabet consists in using BDDs to represent the transition function: a *symbolic DFA* with output set B and input alphabet $\Sigma = 2^A$ for some set A is a triple $\langle S, t, o\rangle$ where S is the set of states, $t\colon S \to \mathrm{BDD}_A[S]$ maps states into nodes of a BDD over A with values in S, and $o\colon S \to B$ is the output function.

Such a symbolic DFA is depicted in Figure 2.9. It has five states, input alphabet $2^{\{a,b,c\}}$, and natural numbers as output set. We represent the BDD graphically; rather than giving the functions t and o separately, we label the square box corresponding to a state x with its output value $o(x)$ and we link this box to the node $t(x)$ defining the transitions of x using a solid arrow. The explicit transition table is given below the drawing.

The simple algorithm described in Figure 2.1 is not optimal when working with such symbolic DFA: at each non-trivial iteration of the main loop, one goes through all letters of $\Sigma = 2^A$ to push all the derivatives of the current pair of states to the queue `todo` (line 12), resulting in a lot of redundancies.

Suppose for instance that we run the algorithm on the DFA of Figure 2.9, starting from states s_1 and s_4. After the first iteration, `R` contains the pair (s_1, s_4), and the queue `todo` contains eight pairs:

$$(s_1, s_4), (s_3, s_4), (s_2, s_5), (s_2, s_5), (s_3, s_4), (s_3, s_4), (s_2, s_5), (s_2, s_5)$$

Assume that elements of this queue are popped from left to right. The first element is removed during the following iteration, since (s_1, s_4) already is in `R`. Then (s_3, s_4) is processed: it is added to `R`, and the above eight pairs are appended again to the queue, which now has fourteen elements. The following pair is processed similarly, resulting in a queue with twenty one $(14 - 1 + 8)$ pairs. Since all pairs of this queue are already in `R`, it is finally emptied through twenty one iterations, and the algorithm returns true.

Note that it would be even worse if the input alphabet was actually declared to be $2^{\{a,b,c,d\}}$: even though the bit d of all letters is irrelevant for the considered DFA, each non-trivial iteration of the algorithm would push even more copies of each pair to the `todo` queue.

What we propose here is to exploit the symbolic representation, so that a given pair is pushed only once. Intuitively, we want to recognise that starting from the pair of nodes (n, m), the letters 010, 011, 110 and 111 are equivalent[1], since they lead to the same pair, (s_2, s_5). Similarly, the letters 001, 100, and 101 are equivalent: they lead to the pair (s_3, s_4).

This idea is easy to implement using BDDs: like for the `apply` function (Figure 2.8), it suffices to zip the two BDDs together, and to push pairs when we reach two leaves. We use for that the procedure `iter2` from Figure 2.10, which successively applies a given function to all pairs reachable from two nodes. Its code is almost identical to `apply`, except that nothing is constructed (and memoisation is just used to remember those pairs that have already been visited).

We finally modify the simple algorithm from Section 2.1 by using this procedure on line 12: we obtain the code given in Figure 2.11. We apply `iter2` to its first argument once and for all (line 7), so that we maximise memoisation: a pair of nodes that has been visited in the past will never be visited again, since all pairs of states reachable from that pair of nodes are already guaranteed to be processed. (As an invariant, we have that all pairs reachable from a pair of nodes memoised in `push_pairs` appear in `r` $\cup$ `todo`.)

Let us illustrate this algorithm by running it on the DFA from Figure 2.9, starting from states s_1 and s_4 as previously. During the first iteration, the pair (s_1, s_4) is added to `R`, and `push_pairs` is called on the pair of nodes (n, m). This call virtually results in building the following BDD, where leaves consist of calls to `Queue.push todo`.

[1] Letters being elements of $2^{\{a,b,c\}}$, we represent them with bit-vectors of length three.

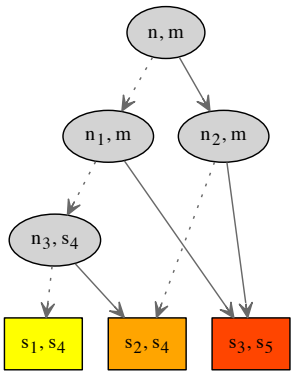
```
let iter2 (f: α × β → unit): α node → β node → unit =
 memo_rec (fun iter2 x y →
 match c(x), c(y) with
   | V v, V w → f (v,w)
   | V _, N(_,l,r) → iter2 x l; iter2 x r
   | N(_,l,r), V _ → iter2 l y; iter2 r y
   | N(a,l,r), N(a',l',r') →
     if a=a' then iter2 l l'; iter2 r r'
     if a<a' then iter2 l y ; iter2 r y
     if a>a' then iter2 x l'; iter2 x r')
```

Figure 2.10: Iterating over the set of pairs reachable from two nodes.

```
val o: α → β
val t: α → α bdd → α

let symb_Naive(x,y) =
  let R = Set.empty() in
  let todo = Queue.singleton (x,y) in
  let push_pairs = iter2 (Queue.push todo) in
  while Queue.not_empty todo do
    let (x,y) = Queue.pop todo in
    if Set.mem (x,y) R then continue
    if o x ≠ o y then return false
    push_pairs (t x) (t y)
    Set.add (x,y) R
  done;
  return true
```

Figure 2.11: Symbolic algorithm for checking language equivalence.



The following three pairs are thus pushed to `todo`.

$$(s_1, s_4), (s_3, s_4), (s_2, s_5)$$

The first pair is removed by a trivial iteration: (s_1, s_4) already belongs to `R`. The two other pairs are processed by adding them to `R`, but without pushing any new pair to `todo`: thanks to memoisation, the two expected calls to `push_pairs n m` are skipped.

All in all, each reachable pair is pushed only once to the `todo` queue. More importantly, the derivatives of a given pair are explored symbolically. In particular, the algorithm would execute exactly in the same way, even if the alphabet was actually declared to be much larger (for instance because the considered states were part of a bigger automaton with more letters). In fact, the main loop is executed at most n^2 times, where n is the total number of BDD nodes (both leaves and decision nodes) reachable from the starting states.

Finally note that in the code from Figure 2.11, the candidate relation `R` is redundant, as the pairs it contains are also stored implicitly in the memoisation table of `iter2` (except for the initial pair). The corresponding lines (5, 10, and 13) can thus be removed.

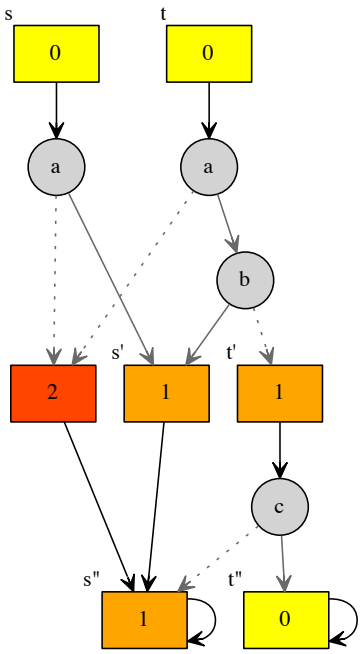
2.3.3 Displaying symbolic counter-examples.

The bisimulation-based algorithms for language equivalence can be instrumented to produce counter-examples in case of failure, i.e., a word which is accepted by one state and not by the other.

An advantage of the previous algorithm is that those counter-examples can be displayed symbolically; thus enhancing readability. This is particularly important in the context of formal assisted proofs (e.g., in Chapter 3, when working with KAT in Coq [110]), where a plain guarded string is

often too big to be useful to the user, while a 'symbolic' guarded string—where only the relevant bits are displayed—can be really helpful to understand which hypotheses have to be used to solve the current goal.

Consider for instance the following automaton.



Intuitively, the topmost states s and t are not equivalent because t can take two transitions to reach t'', with output 0, while with two transitions, s can only reach s'', with output 1. More precisely, the word $100\ 001$ over $2^{\{a,b,c\}}$ is a counter-example: we have

$$[s](100\ 001) = [s'](001) = o(s'') = 1 \ ,$$
$$[t](100\ 001) = [t'](001) = o(t'') = 0 \ .$$

But there are plenty of other counter-examples of length two: it suffices that: a be assigned true and b be assigned false in the first letter, and that c be assigned true in the second letter. The values of the bit c in the first letter, and of the bits a and b in the second letter do not change the above computation. As a consequence, this counter-example is best described as the pseudo-word $10\text{-}\ \text{-}\text{-}1$, or alternatively the word $(a \wedge \neg b)\, c$ whose letters are conjunctions of literals indicating the least requirements to get a counter example.

The algorithm from Figure 2.11 can be modified so as to give this information back to the user. The required modifications are as follows:

- modify the queue `todo` to store triples (w, x, y) where (x, y) is a pair of states to process, and w is the associated potential counter-example;
- modify the function `iter2` (Figure 2.10), so that it uses an additional argument to record the encountered node labels, with negative polarity when going through the recursive call for the left child, and positive polarity for the right child;
- modify line 11 of the main algorithm to return the symbolic word associated with the current pair when the output test fails.

2.3.4 Disjoint sets forests on BDDs

The previous algorithm can be freely enhanced by using up-to techniques, as described in Section 2: it suffices to modify line 10 to skip pairs more or less aggressively, according to the chosen up-to technique. For an up-to technique f, line 10 thus becomes

```
if Set.mem (x,y) (f r) then continue .
```

The up-to-equivalence technique used in Hopcroft and Karp's algorithm can however be integrated in a deeper way, by exploiting the fact that we work with BDDs. This leads to a second algorithm, which we describe in this section.

Let us first recall *disjoint sets forests*, the data structure used by Hopcroft and Karp to represent equivalence classes. As explained in Section 2.1, this data-structure makes it possible to check whether two elements belong to the same class and to merge two equivalence classes, both in almost constant amortised time [134].

The idea consists in storing a partial map from elements to elements and whose underlying graph is acyclic. An element for which the map is not defined is the *representative* of its equivalence class, and the representative of an element pointing in the map to some y is the representative of y. Two elements are equivalent if and only if they lead to the same representative; to merge two equivalence classes, it suffices to add a link from the representative of one class to the representative of the other class. Two optimisations are required to obtain the announced theoretical complexity:

- when following the path leading from an element to its representative, one should compress it in some way, by modifying the map so that the elements in this path become closer to their representative. There are various ways of compressing paths, in the sequel, we use the method called *halving* [134];
- when merging two classes, one should make the smallest one point to the biggest one, to avoid generating too many long paths. Again,

```
let unify (f: β × β → unit): β node → β node → unit =
 (* the disjoint sets forest *)
 let m = Hmap.empty() in
 let link x y = Hmap.add m x y in
 (* representative of a node *)
 let rec repr x =
  match Hmap.get m x with
    | None → x
    | Some y → match Hmap.get m y with
      | None → y
      | Some z → link x z; repr z
 in
 let rec unify x y =
  let x = repr x in
  let y = repr y in
  if x ≠ y then
  match c(x), c(y) with
    | V v, V w → link x y; f (v,w)
    | V _, N(_,l,r) → link y x; unify x l; unify x r
    | N(_,l,r), V _ → link x y; unify l y; unify r y
    | N(a,l,r), N(a',l',r') →
      if a=a' then link x y; unify l l'; unify r r'
      if a<a' then link x y; unify l y ; unify r y
      if a>a' then link y x; unify x l'; unify x r')
 in unify
```

Figure 2.12: Unifying two nodes of a BDD, using disjoint set forests.

there are several possible heuristics, but we elude this point in the sequel.

As explained above, the simplest thing to do would be to replace the bisimulation candidate `R` from Figure 2.11 by a disjoint sets forest over the states of the considered automaton, as we did in Section 2.1.3. Instead, a new idea consists in relating the BDD nodes of the symbolic automaton rather than just its states (i.e., just the BDD leaves). By doing so, one avoids visiting pairs of nodes that have already been visited up to equivalence.

Concerning the implementation, we first introduce a BDD unification algorithm (Figure 2.12), i.e., a variant of the function `iter2` which uses disjoint sets forest rather than plain memoisation. This function first creates an empty forest (we use Filliâtre's module `Hmap` of maps over hash-consed values to represent the corresponding partial maps). The function `link` adds a link between two representatives; the recursive terminal function `repr`

```
1 let symb_HK(x,y) =
2   let todo = Queue.singleton (x,y) in
3   let push_pairs = unify (Queue.push todo) in
4   while Queue.not_empty todo do
5     let (x,y) = Queue.pop todo in
6     if o x ≠ o y then return false
7     push_pairs (t x) (t y)
8   done;
9   return true
```

Figure 2.13: Symbolic algorithm optimised with disjoint set forests.

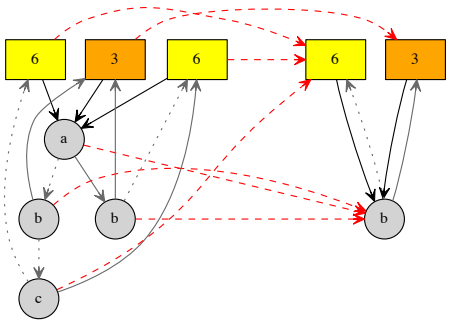
looks for the representative of a node and implements halving. The inner function `unify` is defined similarly as `iter2`, except that it first takes the representative of the two given nodes, and that it adds a link from one to the other before recursing.

Those links can be put in any direction on lines 18 and 22, and we should actually use an appropriate heuristic to take this decision, as explained above. In the four other cases, we put a link either from the node to the leaf, or from the node with the smallest label to the node with the biggest label. By proceeding this way, we somehow optimise the BDD, by leaving as few decision nodes as possible.

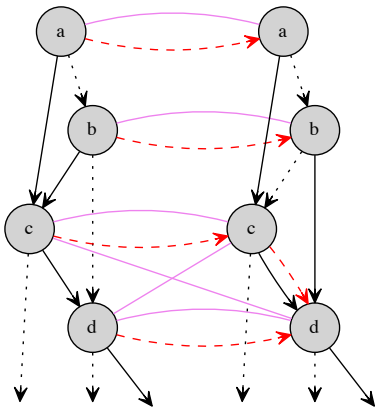
It is important to notice that there is actually no choice left in those four cases: we work implicitly with the optimised BDD obtained by mapping all nodes to their representatives, so that we have to maintain the invariant that this optimised BDD is ordered and thus acyclic. (Notice that this optimised BDD need not be reduced anymore: the children of a given node might be silently equated, and a node might have several representations since its children might be silently equated with the children of another node with the same label.)

We finally obtain the algorithm given in Figure 2.13. It is similar to the previous one (Figure 2.11), except that we use the new function `unify` to push pairs into the `todo` queue, and that we no longer store the bisimulation candidate `R`: this relation is subsumed by the restriction of the disjoint set forests to BDD leaves.

If we execute this algorithm on the symbolic DFA from Figure 2.9, between states s_1 and s_4, we obtain the disjoint set forest depicted below using dashed red arrows. This actually corresponds to the pairs which would be visited by the first symbolic algorithm (Figure 2.11).



If instead we start from the top-most nodes in the following partly described automaton, we would get the disjoint set forest depicted similarly in red, while the first algorithm would go through all violet lines, one of which is superfluous.



The corresponding optimised BDD consists of the three nodes labelled with a, b, and d on the right-hand side. This BDD is not reduced, as explained above: the node labelled with b should be removed since it points twice to the node labelled with d, and removing this node makes the node labelled with a useless, in turn.

2.3.5 Discussion

An implementation of the previous algorithms is available online as an OCaml library [109].

Complexity. Concerning complexity, while the algorithm from Figure 2.11 is quadratic in the number n of BDD nodes (and leaves) that are reachable from the starting symbolic DFA, the optimised algorithm from Figure 2.13 performs at most n iterations: two equivalence classes of nodes are merged each time a link is added, and we start with the discrete partition of nodes.

Unfortunately, we cannot immediately deduce that the algorithm is almost linear, as did Tarjan for Hopcroft and Karp's algorithm [134]. The problem is that we cannot always freely choose how to link two representatives (i.e., on lines 19, 20, 23, and 24 in Figure 2.12), so that we cannot guarantee that the amortised complexity of maintaining those equivalence classes is almost constant. We conjecture that such a result holds, however, as the choice we enforce in those cases virtually suppresses binary decision nodes, and thus reduces the complexity of subsequent BDD unifications. Also note that together with Goel, Khanna and Larkin, Tarjan recently showed that the almost constant amortised complexity is still reached (asymptotically) with randomised linking [54].

KAT. Our initial motivation for this work was to obtain an efficient decision procedure for Kleene algebra with tests. Indeed, we proposed various constructions for building efficiently a symbolic automaton out of a KAT expression: symbolic versions of the extensions of Brzozowski's derivatives [36] and Antimirov' partial derivatives [11] to KAT, as well as a generalisation of Ilie and Yu's inductive construction [69]. We skip those constructions in the present manuscript, referring the interested reader to [111].

Unification with row types. As mentioned at the beginning of this chapter, Hopcroft and Karp's algorithm can be seen as an instance of Huet's first-order unification algorithm for recursive terms (i.e., without occur-check). The algorithm presented in Figure 2.13, and more specifically the BDD unification sub-algorithm (Figure 2.12) is reminiscent of Rémy's extension of Huet's algorithm for dealing with *row types*—to obtain an ML-like type inference algorithm in presence of extensible records [119, 120, 103].

More precisely, row types are almost-constant functions from a given set of labels to types, typically represented as association lists with a default value. Unification of such row types is performed pointwise, and is implemented by zipping the two association lists together, as we do here with BDDs (which generalise from almost constant functions to functions with finitely many output values).

It would thus be interesting to understand whether our generalisation of this unification sub-algorithm, from association lists to BDDs, could be useful in the context of unification: either by exploiting the richer structure of functions represented by BDDs, or just for the sake of efficiency, when the

set of labels is large (e.g., for type inference on object-oriented programs, where labels correspond to method names).

Chapter 3

Automation in the Coq proof assistant

Proof assistants such as Coq or Isabelle/HOL are software tools that make it possible to certify mathematical proofs or programs.

We present a Coq library for relation algebra, including tools for automated reasoning about various fragments up to Kleene algebra with tests. We show how to exploit these tools in the context of program verification by proving equivalences of while programs, correctness of some standard compiler optimisations, Hoare rules for partial correctness, and a particularly challenging equivalence of flowchart schemes.

Proof assistants have gained in popularity since the last decade, with remarkable achievements such as a certified realistic C-compiler [88], a complete formal proof of Feit-Thompson's odd order theorem—a milestone result in group theory [55]—and Kepler's conjecture—a long standing conjecture about optimal sphere packing (cf. the Flyspeck project). The first two projects used the Coq proof assistant. The third one used a mixture of Isabelle and HOL Light.

Formal proofs provide the highest confidence degree: corner cases cannot be forgotten by inadvertence, all details must be written down, and even the boring details are properly reviewed, since the reviewer actually is the computer. However, writing a formal proof requires a lot of work and expertise, as one regularly has to struggle against the system to transform mathematical intuitions into an acceptable proof. A crucial aspect with this respect is automation. Automation is mandatory in any non-trivial development, to take care of the boring details and let the author focus on the challenging parts of the proof. In Coq, part of the automation is provided by high-level *tactics* like `ring` for solving polynomial equations, `tauto` for propositional tautologies, `lia` for linear integer arithmetic, `omega` for Presburger arithmetic, or `congruence` for deciding whether an equality is a con-

sequence of a set of other equalities. Depending on the proof under development, the use of such tactics reduces the final proof script by an order of magnitude. All this without sacrificing the global trust level: these tactics provide formal proofs.

One of our mid-term objectives is to develop a comprehensive library for relation algebra, including proofs of standard results and powerful automation tools. We already made an important step towards this objective: the RelationAlgebra library [110] is a medium-sized library which currently contains Kozen's completeness proofs for Kleene algebra and KAT, as well as associated decision procedures and tactics. We plan to develop tactics for the other decidable fragments of the calculus of relations: this library is designed from the beginning to support the whole calculus of relations, in an axiomatic way.

3.1 Relation algebra and KAT in Coq

We want to deal with all fragments of the calculus of relations, axiomatically: monoids $\langle \cdot, 1 \rangle$, allegories $\langle \cdot, \cap, \cdot^\circ, 1 \rangle$, Kleene algebras $\langle \cdot, +, \cdot^\star, 1, 0 \rangle$ and all other combinations. The model of binary relations is the guideline: if possible, we use a complete axiomatisation (e.g., Kleene algebra), otherwise we use an incomplete yet useful one (e.g., allegories).

In order to factor these axiomatisations, we use a system of *bitmask*: all operations from Chapter 1 are always defined (plus a few ones actually, like left and right residuals [117, 53]), but the axioms they satisfy vary according to a parameter, the bitmask, describing which operations are currently considered. Using Coq's dependent types, this makes it really easy to define various concepts once and for all: the free model, normalisation functions, and general results from universal algebra that do not depend on the considered fragment.

Another specificity for which Coq's dependent types help a lot is that we actually define categories rather than algebras: all the operations we consider can easily be typed in such a way that composition $(\cdot)$ and identity (1) become the basic ingredients of a category[1]. Doing so makes it possible to work with heterogeneous relations, between distincts sets, rather than just homogeneous ones, on a single and fixed set. Lattice-theoretic operations act on each homset, converse is a contravariant identity-on-object functor, and Kleene star only operates on square homsets: those with the same source and target.

Doing so makes it possible to deal with models such as heterogeneous relations or rectangular matrices, the latter model being extremely useful for the proof of completeness of Kleene algebra (Theorem 1.2.2). On the other hand, handling this slight generalisation is non-trivial and deeply

[1] This is actually the way Freyd and Scedrov define allegories [52].

impacts the whole infrastructure (e.g., for reification). This actually leads us to prove "untyping theorems", which make it possible to prove typed equations by resorting to untyped computations [106, 107].

3.1.1 Decision procedure for KAT

The decision procedure for KAT cannot be formulated, a priori, as a simple rewriting system: it involves automata algorithms, it cannot be defined in Ltac, at the meta-level, and it does not produce a certificate which could easily be checked in Coq, a posteriori. This leaves us with only one possibility: defining a reflexive tactic [25, 4, 56].

Doing so is challenging: we have to prove completeness of KAT axioms w.r.t. guarded string languages (Theorem 1.5.1), and to provide a provably correct algorithm for language equivalence of KAT expressions.

The completeness theorem for KAT is far from trivial; we actually have to formalise a lot of preliminary material: finite sums, finite sets, unique decomposition of Boolean expressions into sums of atoms, regular expression derivatives, expansion theorem for regular expressions, matrices, automata… This theorem relies on the completeness of Kleene algebra, which requires at some point to establish decidability of language equivalence for DFA. We do so by providing a "mathematical algorithm", which is reasonably easy to prove correct and complete, but which is absurdly inefficient.

In contrast, we need a reasonably efficient algorithm for language equivalence of KAT expressions, which does not need to be proved complete: correctness is sufficient for the tactic to work. To this end, we use the naive coinductive algorithm from the previous chapter, using an extension of Antimirov' partial derivatives [11] to obtain automata from KAT expressions.

We do not give more details here, the interested reader can consult [110] or the library, which is documented [115]. Putting the previous ingredients together, we obtain a reflexive tactic called `kat`, which allows one to discharge automatically any goal belonging to the equational theory of KAT. This tactic works on any model of KAT: those already declared in the library (relations, languages, matrices, traces), but also the ones declared by the user.

For the sake of simplicity, the Coq algorithm we implemented for KAT does not produce a counter-example in case of failure. To be able to give such a counter-example to the user, we actually run an OCaml version of the algorithm first. This has two advantages: the tactic is faster in case of failure, and the counter-example—a guarded string—can be pretty-printed in a nicer way, using symbolism as explained in Section 2.3.3.

3.1.2 Eliminating hypotheses

The above `kat` tactic works for the equational theory of KAT, that is, for solving (in)equations that hold in any model of KAT, under any interpretation. In particular, this tactic does not make use of any hypothesis which is specific to the model or to the interpretation.

The techniques we discussed in Section 1.5.4 to eliminate some hypotheses in KAT can be easily automated in Coq. We first prove once and for all the appropriate equivalences and implications (the tactic `kat` is useful for that). Then we define some tactics in Ltac that collect hypotheses of shape (i-iv), put them into shape (i), and aggregate them into a single one which is finally used to update the goal according to (†). Separately, we define a tactic that rewrites in the goal using all hypotheses of shape (v), through (‡). Finally, we obtain a tactic called `hkat`, that just preprocesses the conclusion of the goal using all hypotheses of shape (i-v) and then calls the `kat` tactic. Note that the completeness of this method [59] is a meta-theorem; we do not need to formalise it.

3.2 Case studies

We now present some examples of Coq formalisations where one can take advantage of our library.

3.2.1 Bigstep semantics of 'while' programs

The bigstep semantics of 'while' programs is taught in almost every course on semantics and programming languages. Such programs can be embedded into KAT in a straightforward way [80], thus providing us with proper tools to reason about them. Let us formalise such a language in Coq.

Assume a type `state` of states, a type `loc` of memory locations, and an `update` function to update the value of a memory location. Call *arithmetic expression* any function from states to natural numbers, and *Boolean expression* any function from states to Booleans (we use a partially shallow embedding). The 'while' language is defined by the inductive type below:

```
Variable loc, state: Set.
Variable update: loc → nat → state → state.

Definition expr := state → nat.
Definition test := state → bool.
```

```
Inductive prog :=
| skp
| aff (l: loc) (e: expr)
| seq (p q: prog)
| ite (b: test) (p q: prog)
| whl (b: test) (p: prog).
```

The bigstep semantics of such programs is given as a "state transformer", i.e., a binary relation between states. Following standard textbooks, one can define this semantics in Coq using an inductive predicate:

```
Notation "l←e" := (aff l e).  Notation "p;q" := (seq p q).

Inductive bstep: prog → rel state state :=
| s_skp: ∀ s, bstep skp s s
| s_aff: ∀ l e s, bstep (l←e) s (update l (e s) s)
| s_seq: ∀ p q s s' s'', bstep p s s' → bstep q s' s'' → bstep (p;q) s s''
| s_ite_ff: ∀ b p q s s', ¬ b s → bstep q s s' → bstep (ite b p q) s s'
| s_ite_tt: ∀ b p q s s', b s → bstep p s s' → bstep (ite b p q) s s'
| s_whl_ff: ∀ b p s, ¬ b s → bstep (whl b p) s s
| s_whl_tt: ∀ b p s s', b s → bstep (seq p (whl b p)) s s' → bstep (whl b p) s s'.
```

Alternatively, one can define this semantic through the relational model of KAT, by induction over the program structure:

```
Fixpoint bstep (p: prog): rel state state :=
  match p with
    | skp ⇒ 1
    | seq p q ⇒ bstep p·bstep q
    | aff l e ⇒ upd l e
    | ite b p q ⇒ [b]·bstep p + [¬b]·bstep q
    | whl b p ⇒ ([b]·bstep p)^*·[¬b]
  end.
```

(Notations come for free since binary relations are already declared as a model of KAT.) The 'skip' instruction is interpreted as the identity relation; sequential composition is interpreted by relational composition. Assignments are interpreted using the following auxiliary function:

```
Definition upd l e: rel state state := fun s s' ⇒ s' = update l (e s) s.
```

For the 'if-then-else' statement, the Boolean expression `b` is a predicate on states, i.e., a test in our relational model of KAT; this test is used to guard both branches of the possible execution paths. Accordingly for the 'while' loop, we iterate the body of the loop guarded by the test, using Kleene star. We make sure one cannot exit the loop before the condition gets false by post-guarding the iteration with the negation of this test.

This alternative definition is easily proved equivalent to the previous one. Its relative conciseness makes it easier to read (once one knows KAT notation); more importantly, this definition allows us to exploit all theorems and tactics about KAT, for free. For instance, suppose that one wants to prove some program equivalences. First define program equivalence, through the bigstep semantics:

```
Notation "p ∼ q" := (bstep p == bstep q).
```

(The "==" symbol denotes equality in the considered KAT model; in this case, relational equality.) The following lemmas about unfolding loops and dead code elimination, can be proved automatically.

```
Lemma two_loops b p: whl b (whl b p) ∼ whl b p.
Proof. simpl. kat. Qed.
```

```
(* ([b]·(([b]·bstep p)*·[¬b]))*·[¬b] == ([b]·bstep p)*·[¬b] *)

Lemma fold_loop b p: whl b (p ; ite b p skp) ∼ whl b p.
Proof. simpl. kat. Qed.
(* ([b]·(bstep p·([b]·bstep p + [¬b]·1)))*·[¬b] == ([b]·bstep p)*·[¬b] *)

Lemma dead_code a b p q r:  whl (a ∨ b) p ; ite b q r ∼ whl (a ∨ b) p ; r.
Proof. simpl. kat. Qed.
(* ([a ∨ b]·bstep p)*·[¬(a ∨ b)]·([b]·bstep q + [¬b]·bstep r)
                              == ([a ∨ b]·bstep p)*·[¬(a ∨ b)]·bstep r *)
```

(The semicolon in program expressions is a notation for sequential composition; the comments below each proof show the intermediate goal where the `bstep` fixpoint has been simplified, thus revealing the underlying KAT equality.)

Of course, the `kat` tactic cannot prove arbitrary program equivalences: the theory of KAT only deals with the control-flow graph of the programs and with the Boolean expressions, not with the concrete meaning of assignments or arithmetic expressions. We can however mix automatic steps with manual ones. Consider for instance the following example, where we prove that an assignment can be delayed. Our tactics cannot solve it automatically since some reasoning about assignments is required; however, by asserting manually a simple fact (in this case, an equation of shape (ii)), the goal becomes provable by the `hkat` tactic.

```
Definition subst l e (b: test): test := fun s ⇒ b (update l (e s) s).
Lemma aff_ite l e b p q: l←e; ite b p q ∼ ite (subst l e b) (l←e; p) (l←e; q).
Proof.
  simpl. (* upd l e·([b]·bstep p + [¬b]·bstep q) ==
     [subst l e b]·(upd l e·bstep p)·[¬subst l e b]·(upd l e·bstep q) *)
  assert (upd l e·[b] == [subst l e b]·upd l e)
     by (cbv; firstorder; subst; eauto).
  hkat.
Qed.
```

3.2.2 Hoare logic for partial correctness

Propositional Hoare logic for partial correctness [62] of while programs is subsumed by KAT [80]. The key ingredient in Hoare logic is the notion of a "Hoare triple" $\{A\}p\{B\}$, where p is a program, and A, B are two formulas about the memory manipulated by the program, respectively called pre- and post-conditions. A Hoare triple $\{A\}p\{B\}$ is *valid* if whenever the program p starts in some state s satisfying A and terminates in a state s', then s' satisfies B. Such a statement can be translated into KAT as a simple equation:

$$[A]p[\neg B] = 0$$

Indeed, $[A]p[\neg B] = 0$ precisely means that there is no execution path along p that starts in A and ends in $\neg B$. Such equations are Hoare equations (they have the shape (i) from Section 3.1.2), so that they can be eliminated automatically. As a consequence, inference rules of Hoare logic can be proved automatically using the `hkat` tactic. For instance, for the 'while' rule, we get the following script:

```
Lemma rule_whl A b p: {A ∧ b} p {A} → {A} whl b p {A ∧ ¬b}.
Proof. simpl. hkat. Qed.
(* [A ∧ b]·bstep p·[¬A]== 0 → [A]·(([b]·bstep p)^*·[¬b])·[¬(A ∧ ¬b)]== 0 *)
```

3.2.3 Compiler optimisations

Kozen and Patron [82] use KAT to verify a rather large range of standard compiler optimisations, by equational reasoning. Citing their abstract, they cover "*dead code elimination, common subexpression elimination, copy propagation, loop hoisting, induction variable elimination, instruction scheduling, algebraic simplification, loop unrolling, elimination of redundant instructions, array bounds check elimination, and introduction of sentinels*". They cannot use automation, so that the size of their proofs ranges from a few lines to half a page of KAT computations.

We formalised all those equational proofs using our library. Most of them can actually be solved instantaneously, by a simple call to the `hkat` tactic. For the few remaining ones, we gave three to four line proofs, consisting of first rewriting using hypotheses that cannot be eliminated, and then a call to `hkat`.

The reason why `hkat` performs so well is that most assumptions allowing to optimise the code in these examples are of the shape (i-v). For instance, to state that an instruction p has no effect when $[a]$ is satisfied, we use an assumption $[a]p = [a]$. Similarly, to state that the execution of a program x systematically enforces $[a]$, we use an assumption $x = x[a]$. The assumptions that cannot be eliminated are typically those of the shape $pq = qp$: "the instructions p and q commute"; such assumptions have to be used manually.

3.2.4 Flowchart schemes

The last example we discuss here is due to Paterson, it consists in proving the equivalence of two flowchart schemes (i.e., goto programs—see Manna's book [90] for a complete description of this model). These two schemes are given in Figure 3.1; Manna proves their equivalence using several successive graph transformations. His proof is really high-level and informal; it is one page long, plus three additional pages to draw intermediate flowcharts schemes. Angus and Kozen [10] give a rather detailed

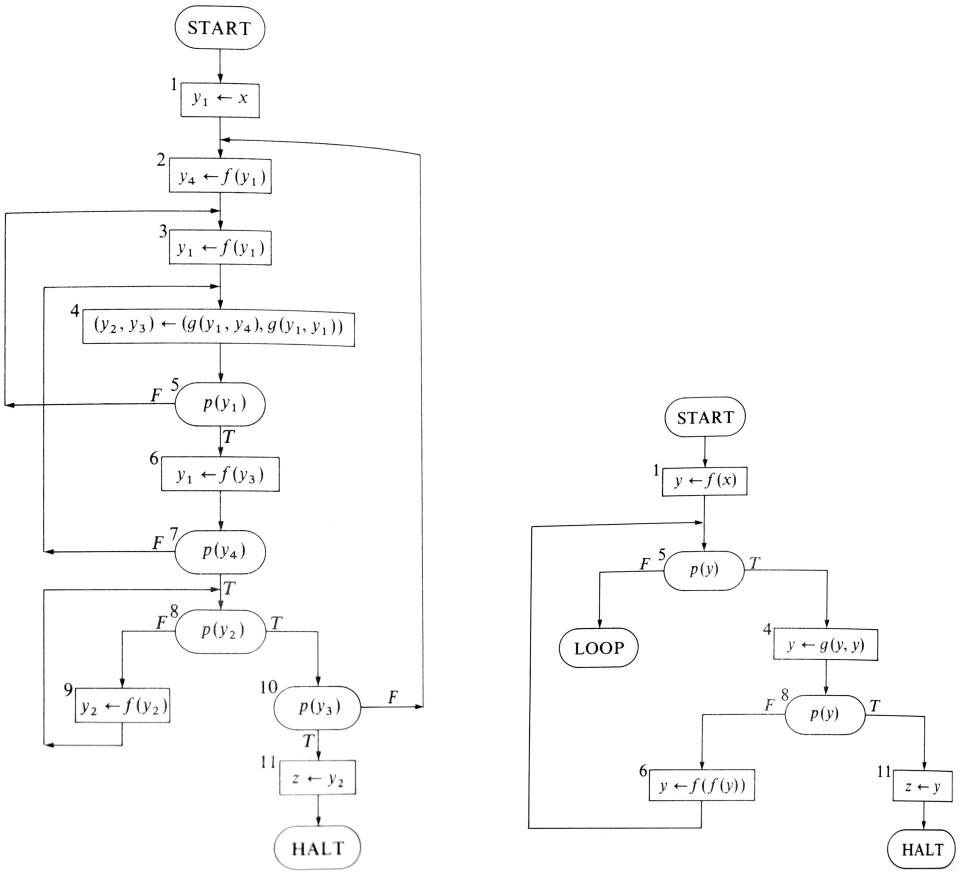


Figure 3.1: Paterson's equivalent flowchart schemes [90, pages 254 and 258].

equational proof in KAT, which is about six pages long. Using the `hkat` tactic together with some ad-hoc rewriting tools, we managed to formalise Angus and Kozen's proof in three rather sparse screens.

Like in Angus and Kozen's proof, we progressively modify the KAT expression corresponding to the first scheme, to make it evolve towards the expression corresponding to the second scheme. Our mechanised proof thus roughly consists in a sequence of transitivity steps closed by `hkat`, allowing us to perform some rewriting steps manually and to move to the next step. This is illustrated schematically by the code presented in Fig. 3.2.

Most of our transitivity steps (the y_i's) already appear in Angus and Kozen's proof; we can actually skip a lot of their steps, thanks to `hkat`. Some of these simplifications can be spectacular: for instance, they need one page to justify the passage between their expressions (24) and (27), while a simple call to `hkat` does the job; similarly for the page they need between their steps (38) and (43).

```
Lemma Paterson: x_1 == z.
Proof.
  transitivity y_1. hkat.        (* x_1 == y_1 *)
  a few rewriting steps transforming y_1 into x_2.
  transitivity y_2. hkat.        (* x_2 == y_2 *)
  a few rewriting steps transforming y_2 into x_3.
  (* ... *)
  transitivity y_12. hkat.       (* x_12 == y_12 *)
  a few rewriting steps transforming y_12 into x_13.
  hkat.                          (* x_13 == z *)
Qed.
```

Figure 3.2: Skeleton for the proof of equivalence of Paterson's flowchart schemes.

3.3 Discussion

Several formalisations of algorithms and results related to regular expressions and languages have been proposed since we released our Coq reflexive decision procedure for Kleene algebra [26]: partial derivatives for regular expressions [5], regular expression equivalence [42, 84, 13, 96], regular expression matching [75]. None of these works contains a formalised proof of completeness for Kleene algebra, so that they cannot be used to obtain a general tactic for KA (note however that Krauss and Nipkow [84] obtain an Isabelle/HOL tactic for binary relations using Theorem 1.2.1 to sidestep the completeness proof—but they cannot deal with other models of KA).

On the algebraic side, Struth et al. [50, 12] showed how to formalise and use relation algebra and Kleene algebra in Isabelle/HOL; they exploit the automation tools provided by this assistant, but they do not try to define decision procedures specific to Kleene algebra, and they do not prove completeness.

The presented tools allowed us to shorten significantly a number of paper proofs—those about Hoare logic, compiler optimisations, and flowchart schemes. Getting a way to guarantee that such proofs are correct is important: although mathematically simple, they tend to be hard to proofread (we invite the sceptical reader to check Angus and Kozen's paper proof of Paterson example [10]). Moreover, automation greatly helps when searching for such proofs: being able to get either a proof or a counter-example for any proposed equation is a big plus: it makes it much easier to progress in the overall proof.

Our library is rather exhaustive as far as Kleene algebra with tests is concerned: axiomatisation, models, completeness proof, decision procedure, elimination of hypotheses. Many things remain to be done for other fragments of the calculus of relations: decision procedure for Kleene al-

gebra with converse, axiomatisation and decision of identity-free Kleene lattices, elimination of other kinds of hypotheses.

3.4 Appendix: overall structure of the library

The Coq library can be browsed online [115]; it is documented and axiom-free. Many aspects of this implementation work cannot be explained here: how to encode the algebraic hierarchy, how to work efficiently with finite sets and finite sums, how to exploit symmetry arguments, reflexive normalisation tactics, tactics about lattices, finite ordinals and encodings of set-theoretic constructs in ordinals…

The library is medium-sized: according to `coqwc`, the library consists of 4377 lines of specifications and 3020 lines of proofs, that distribute as follows. We hope to maintain such a reasonable codebase when developing further extensions.

	specif.	proofs	comments
ordinals, comparisons, finite sets…	674	323	225
algebraic hierarchy	490	374	216
models (languages, relations…)	1279	461	404
linear algebra, matrices	534	418	163
completeness, decision procedures, tactics	1400	1444	740

We close this chapter with a succinct description of each module from the library; their dependencies are depicted in Figure 3.3.

Utilities

`common`: basic tactics and definitions used throughout the library
`comparisons`: types with decidable equality and comparison function
`positives`: simple facts about binary positive numbers
`ordinal`: finite ordinals, finite sets of finite ordinals
`pair`: encoding pairs of ordinals as ordinals
`powerfix`: simple pseudo-fixpoint iterator
`lset`: sup-semilattice of finite sets represented as lists

Algebraic hierarchy

`level`: bitmasks allowing us to refer to arbitrary points in the hierarchy
`lattice`: “flat” structures, from preorders to Boolean lattices
`monoid`: typed structures, from po-monoids to residuated Kleene lattices
`kat`: Kleene algebra with tests

`kleene`: Basic facts about Kleene algebra
`normalisation`: normalisation and semi-decision tactics for relation algebra

Models

`prop`: distributive lattice of propositions
`boolean`: Boolean trivial lattice, extended to a monoid.
`rel`: heterogeneous binary relations
`lang`: word languages
`traces`: trace languages
`atoms`: atoms of the free Boolean lattice over a finite set
`glang`: guarded string languages
`lsyntax`: free lattice (Boolean expressions)
`syntax`: free relation algebra
`regex`: regular expressions
`gregex`: KAT expressions (typed—for completeness)
`ugregex`: untyped KAT expressions (for the decision procedure)

Untyping theorems

`untyping`: untyping theorem for structures below KA with converse
`kat_untyping`: untyping theorem for guarded string languages

Linear algebra

`sups`: finite suprema/infima (a la bigop, from ssreflect)
`sums`: finite sums
`matrix`: matrices over all structures supporting this construction
`matrix_ext`: additional operations and properties about matrices
`rmx`: matrices of regular expressions
`bmx`: matrices of Booleans

Automata, completeness

`dfa`: deterministic automata, decidability of language inclusion
`nfa`: matricial non-deterministic finite state automata
`ugregex_dec`: decision of language equivalence for KAT expressions
`ka_completeness`: (untyped) completeness of Kleene algebra
`kat_completeness`: (typed) completeness of Kleene algebra with tests
`kat_reification`: tools and definitions for KAT reification
`kat_tac`: decision tactics for KA and KAT, elimination of hypotheses

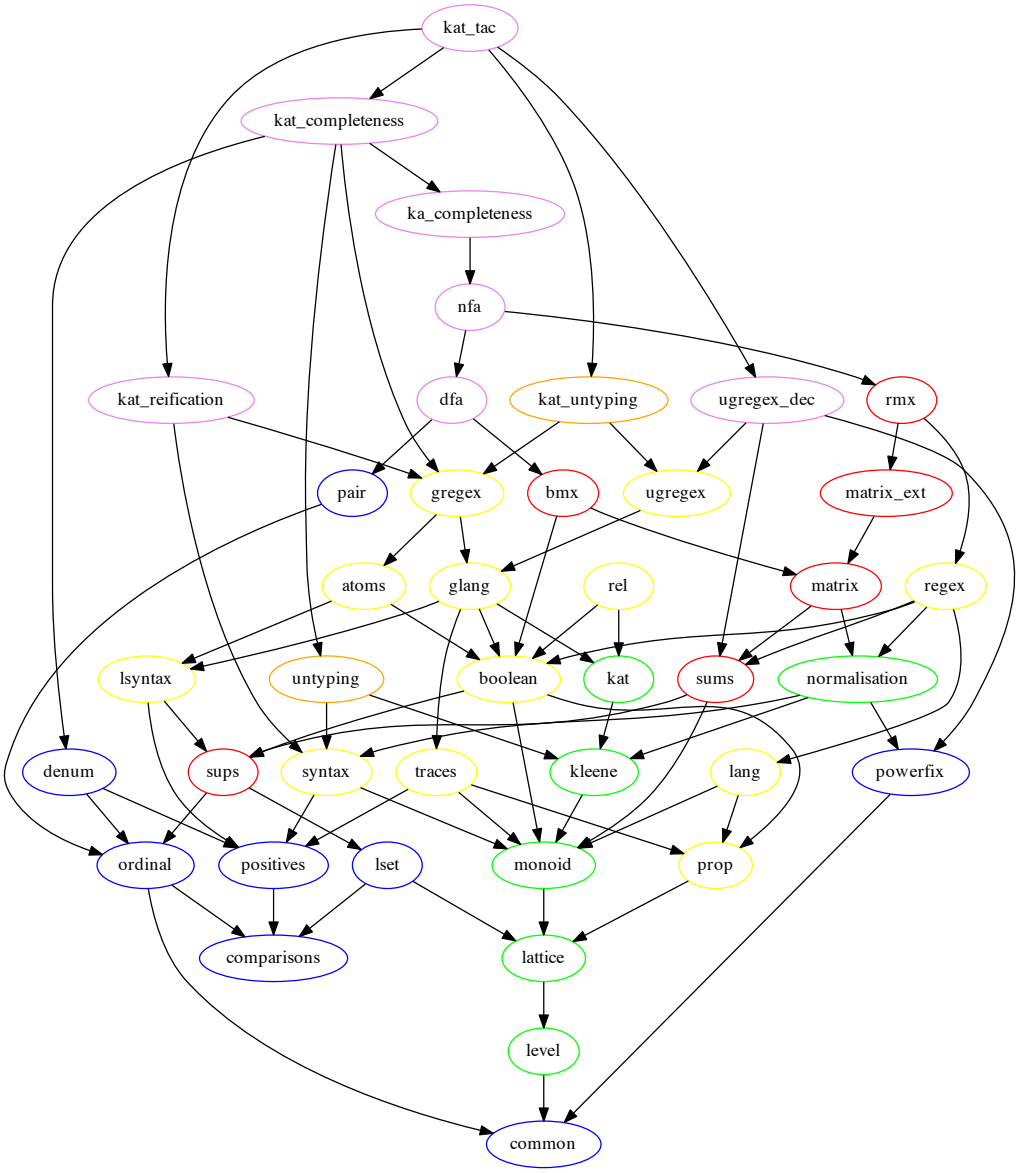


Figure 3.3: Dependencies of the modules from the RelationAlgebra library.

Chapter 4

Abstract coinduction

Coinduction is a simple mathematical tool, dual to induction. It provides powerful proof techniques for checking properties of different kinds of systems, and we have seen in Chapter 2 that it can lead to efficient algorithms, using so-called "up-to techniques". In this chapter we give an abstract account of these enhancements to coinduction, in complete lattices.

Coinduction follows from Knaster-Tarski's fixpoint theorem on complete lattices [74, 136]. It was first used implicitly, for instance in finite automata algorithms [66, 65], until Milner popularised it by proposing *bisimilarity* as a natural way to compare concurrent programs [92]. It was then widely used, for instance to analyse other process calculi [93], the lambda-calculus [3], cryptographic protocols [1], distributed implementations [51], concurrent ML [72], analytic differential equations [124], or C compilers [88, 129].

The reason for such a success is that like induction, coinduction provides a powerful proof technique: to prove some property by coinduction, it suffices to exhibit an *invariant*. Typically, in program semantics, one can prove the equivalence of two programs by exhibiting a *bisimulation relation* that contains those two programs. The point there is that while program equivalence is a global property (for instance, because equivalent programs should remain equivalent under arbitrary contexts), the conditions for a relation to be a bisimulation are local. By using coinduction one can thus ensure a global property by checking only local properties. Very much like induction allows one to reduce a proof about arbitrary natural numbers to a proof about zero and successor.

From the beginning [92], Milner introduced *enhancements* of the bisimulation proof method. They make it possible to work with relations that are much smaller than actual bisimulations and yet ensure program equivalence: they are always contained in a bisimulation. Those relations are usually called *bisimulations up to*. The benefits of these enhancements can be spectacular; a bisimulation up to can be finite whereas any enclosing

bisimulation is infinite. Sometimes it may be hard even *to define* an enclosing bisimulation, let alone carrying out the whole proof. There are many possible enhancements, and they proved useful, if not essential, in proofs about name-passing languages [23, 71, 128], languages with information hiding mechanisms (e.g., existential types, encryption and decryption constructs [1, 132, 131]), and higher-order languages [86, 76]. We used such enhancements in Chapter 2 to improve Hopcroft and Karp's algorithm in the case of non-deterministic finite automata [21, 22].

Sangiorgi developed a first theory of those enhancements [126, 128], which we further refined during our PhD [104, 105], resulting in a book chapter [116]. In this line of work, the emphasis was put on compositionality: given the wide variety of enhancements, it is crucial to have tools to analyse each of them separately, and then to combine them when needed in a concrete proof. Since enhancements do not compose in general, Sangiorgi proposed a notion of *respectful* enhancement. These form a subclass of the valid enhancements, and they enjoy nice compositional properties: they are closed under union and composition. One can thus establish a dictionary of respectful enhancements, and then use any combination of those in concrete proofs. In our refinement of this framework, respectfulness was modified into *compatibility*, a slightly more natural notion which essentially plays the same role but leads to a smoother theory.

Three years ago, Hur et al. proposed *parameterized coinduction* [68], an extremely neat variation on Knaster-Tarski theorem which allows one to present coinductive proofs incrementally, without having to exhibit the invariant (or bisimulation relation) from the beginning. Such a possibility is especially useful in the context of mechanised formal proofs: the process of discovering the appropriate invariant becomes interactive and amenable to automation. They also show in this paper how to exploit respectful enhancements with parameterized coinduction. When doing so, they define the greatest respectful enhancement and they remark in passing that it is so powerful that there is no point in using a different one.

This simple remark recently lead us to rework our theory of enhanced coinduction: unexpectedly, in addition important simplifications, we obtain an alternative proof of Knaster-Tarski's theorem, the distinction between respectful and compatible enhancements vanishes, and parameterized coinduction becomes a byproduct of the theory [112].

4.1 Notation and preliminary material

A *complete lattice* is a triple $\langle X, \leqslant, \bigvee \rangle$ where $\langle X, \leqslant \rangle$ is a partial order (reflexive, transitive, and antisymmetric) such that any subset Y of X has a least

upper bound $\bigvee Y$: for all $z \in X$,

$$\bigvee Y \leqslant z \qquad \text{iff} \qquad \forall y \in Y,\ y \leqslant z$$

A complete lattice always has a bottom element, written $\bot$, and a binary join operation, written with infix symbol $\vee$:

$$\bot \triangleq \bigvee \emptyset \qquad\qquad x \vee y \triangleq \bigvee \{x, y\}$$

Arbitrary greatest lower bounds can be derived from least upper bounds. We denote binary ones (meets) with the infix symbol $\wedge$.

Standard examples of complete lattices include: subsets (of a given set) ordered with inclusion; binary relations (on a given set) ordered with inclusion again, and functions into a complete lattice, ordered pointwise. A fourth example, used thoroughly in this paper, is the set of *monotone* functions on a complete lattice.

More precisely, given a complete lattice $\langle X, \leqslant, \bigvee \rangle$, a function $f : X \to X$ is monotone if it preserves the partial order:

$$\forall x, y \in X,\ x \leqslant y \ \Rightarrow f(x) \leqslant f(y)$$

We write $[X \to X]$ for the set of monotone functions on X. When ordered pointwise, this set forms a complete lattice: for all $f, g : [X \to X]$ and $F \subseteq [X \to X]$,

$$f \leqslant g \ \triangleq\ \forall x \in X, f(x) \leqslant g(x) \qquad\qquad \bigvee F \ \triangleq\ x \mapsto \bigvee_{f \in F} f(x)$$

A *post-fixpoint* of a function $f : [X \to X]$ is an element x such that $x \leqslant f(x)$; a *fixpoint* is an element x such that $x = f(x)$.

In the sequel we mostly work within a generic complete lattice X, the corresponding lattice of monotone functions $[X \to X]$, and that of monotone functions on $[X \to X]$: $[[X \to X] \to [X \to X]]$. To avoid confusion, we use the following convention: letters x, y, z range over elements of X, letters f, g, b, c, t range over functions in $[X \to X]$, and uppercase letters F, B, S, T are reserved for functions in $[[X \to X] \to [X \to X]]$. We follow the same convention in concrete examples, except that we use bold fonts.

This discipline allows us to overload most symbols in the sequel: for instance, depending on the context, $\bot$ can denote the empty set, the bottom element of an abstract complete lattice X, or the everywhere-bottom function in $[X \to X]$.

To further alleviate notation, we denote the identity function by 1, and both function composition and function application by juxtaposition:

- fx denotes the application of a function f to an element x, usually written $f(x)$;

- fg denotes the composition of two functions f and g, usually written $f \circ g$.

(Similarly for Fg and FB.) We associate juxtapositions to the right when there is no ambiguity. For instance, we write fgx for $f(gx) = (fg)x$, fgb for $f(gb) = (fg)b$, and TTf for $T(Tf) = (TT)f$. In contrast, we keep parentheses in expressions such that $(Bf)g$ and $B(fg)$ which are not equal in general.

4.2 Knaster-Tarski and Compatibility

We fix throughout the chapter a complete lattice $\langle X, \leqslant, \bigvee \rangle$ and a monotone function $b : [X \to X]$. Knaster-Tarski's theorem characterises the greatest fixpoint νb of b as the least upper bound of all its post-fixpoints:

$$\nu b = \bigvee_{x \leqslant bx} x \qquad\qquad \frac{x \leqslant y \leqslant by}{x \leqslant \nu b} \tag{4.1}$$

The corresponding coinduction principle is given on the right-hand side. In words, to prove that x is below in the greatest fixpoint, find a post-fixpoint y above x. The idea of enhancements is to use an additional function f and to look for post-fixpoints of bf rather than b: we switch to the following principle of coinduction up to f

$$\frac{x \leqslant y \leqslant bfy}{x \leqslant \nu b} \tag{4.2}$$

The function f typically enlarges its argument, and the post-fixpoints of bf can be much smaller than those of b; these are the bisimulations up to we alluded to in the Introduction. The function f corresponds to a valid enhancement when the above rule holds, or, equivalently, when $\nu(bf) \leqslant \nu b$. Our primary goal is to obtain such functions.

A monotone function $f : [X \to X]$ is *compatible (for b)* if $fb \leqslant bf$. Compatible functions yield valid enhancements (see Remark 4.2.6), and it is straightforward to check that 1 and b are compatible, that the composition of two compatible functions is compatible, and that the least upper bound of a family of compatible functions is compatible.

Definition 4.2.1. *We call* companion of b *the monotone function obtained as the least upper bound of all compatible functions:*

$$t \triangleq \bigvee_{fb \leqslant bf} f$$

Lemma 4.2.2. *The companion is compatible:*

$$tb \leqslant bt \tag{4.3}$$

Thus this is the greatest compatible function. It moreover satisfies

$$b \leqslant t \tag{4.4}$$
$$1 \leqslant t \tag{4.5}$$
$$tt \leqslant t \tag{4.6}$$

The last two inequalities entail idempotence, i.e., $tt = t$.

The companion makes it possible to provide an alternative definition of the greatest (post-)fixpoint:

Theorem 4.2.3. *The greatest fixpoint of b is the value of the companion on the bottom element.*

$$\nu b = t\bot \tag{4.7}$$

Proof. We first show that $t\bot$ is the greatest post-fixpoint:

1. $t\bot$ is a post-fixpoint: we have $t\bot \leqslant tb\bot \leqslant bt\bot$ by monotonicity and compatibility of t;
2. it is the largest: if $x \leqslant bx$, then the constant-to-x function $\hat{x}$ is compatible and thus smaller than t, so that $x = \hat{x}\bot \leqslant t\bot$.

We conclude that $t\bot$ is a fixpoint as in Knaster-Tarski's proof: from monotonicity of b and the first point, $bt\bot$ is also a post-fixpoint, and thus $bt\bot \leqslant t\bot$ by the second point. □

Using idempotence of the companion, we also get

Corollary 4.2.4. *The companion preserves the greatest fixpoint:*

$$t\nu b = \nu b \tag{4.8}$$

Typically, when b is the function defining bisimilarity on some process calculus, we recover the fact that if contextual closure is compatible (and thus below t) then bisimilarity is closed under contexts—see Section 4.6.

In the sequel we write $b^\dagger$ for the composite function bt. This function is an improved version of b, with more post-fixpoints (we have $b \leqslant b^\dagger$) but the same greatest fixpoint:

Theorem 4.2.5. *The companion is a valid enhancement, we have*

$$\nu b^\dagger = \nu b \tag{4.9}$$

Proof. From (4.5) we deduce $b \leqslant b^\dagger$, and thus $\nu b \leqslant \nu b^\dagger$. The interesting result is the other inequality. Since $\nu b^\dagger$ is the greatest post-fixpoint of $b^\dagger$, it suffices to show that any post-fixpoint x of $b^\dagger$ is smaller than νb. We have

$$\begin{aligned} tx \leqslant tb^\dagger x = tbtx & \qquad \text{(assumption on } x \text{ and monotonicity of } t\text{)} \\ \leqslant bttx & \qquad \text{(t is compatible (4.3))} \\ \leqslant btx = b^\dagger x & \qquad \text{(by (4.6) and monotonicity of } b\text{)} \end{aligned}$$

Thus tx is a post-fixpoint of b, and $tx \leqslant \nu b$. We conclude with (4.5): we have $x \leqslant tx \leqslant \nu b$. □

Remark 4.2.6. *In earlier work by Sangiorgi [126] and then by the author [104], where the emphasis was on compatible functions rather than on the greatest one (the companion), the corresponding result is "if f is compatible, and $x \leqslant bfx$ then $x \leqslant \nu b$". Such a result requires a convoluted proof. Indeed, when f is an arbitrary compatible function, one does not have $1 \leqslant f$ and $ff \leqslant f$, and the above proof breaks. Instead, one constructs the sequence $f^0 x \triangleq x$, $f^{i+1}x \triangleq ff^i x$ and one shows by recurrence that $f^i x \leqslant bf^{i+1}x$. One deduces that $f^\omega x \triangleq \bigvee_i f^i x$ is a post-fixpoint, so that $x \leqslant f^\omega x \leqslant \nu b$. Focusing on the companion makes it possible to avoid this use of natural numbers.*

Remark 4.2.7. *One might hope to enhance the function b further by using the companion of $b^\dagger$. However we stagnate when doing so: let $t^\star$ be the companion of $b^\dagger$, we have*

$$t^\star = t \tag{4.10}$$

$$b^{\dagger\dagger} = b^\dagger \tag{4.11}$$

Equation (4.10) generalises Theorem 4.2.5: we have $\nu b^\dagger = t^\star \bot = t\bot = \nu b$.

4.3 Examples

We illustrate the previous notions on three examples: Milner's calculus of communicating systems (CCS) [92], from which the theory of up-to techniques originates; finite automata, for which we used bisimulations up to equivalence and bisimulation up to congruence in Chapter 2; and Rutten's stream calculus [124].

4.3.1 Milner's CCS

Let us consider a fragment of Milner's calculus of communicating systems (CCS). We fix a set of *names* $a, b \ldots$, and a set of process constants $A, B, \ldots$. CCS *processes* and *labels* are defined by the following grammar:

$$\begin{aligned} P, Q &::= A \mid 0 \mid \alpha.P \mid P|P \\ \alpha, \beta &::= a \mid \overline{a} \mid \tau \end{aligned}$$

$$\frac{A \triangleq P \qquad P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'} \qquad\qquad \frac{}{\alpha.P \xrightarrow{\alpha} P}$$

$$\frac{P \xrightarrow{a} P' \qquad Q \xrightarrow{\overline{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q'} \qquad\qquad \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q}$$

Figure 4.1: Labelled transition system of a fragment of CCS.

We let $\mathcal{R}, \mathcal{S}$ range over binary relations on processes.

The corresponding labelled transition system (LTS) is given in Figure 4.1. The first rule accounts for recursion: it assumes that each process constant is associated to a process in some global table. The two symmetric rule for parallel composition are omitted.

Let $\mathbf{b}$ be the following monotone function on the lattice of binary relations on processes:

$$\begin{aligned}\mathbf{b} : \ \mathcal{R} \mapsto \{\langle P, Q\rangle \mid \forall \alpha,\ & \\ & \forall P',\ P \xrightarrow{\alpha} P' \text{ entails } \exists Q',\ Q \xrightarrow{\alpha} Q' \text{ and } P' \ \mathcal{R}\ Q' \\ & \forall Q',\ Q \xrightarrow{\alpha} Q' \text{ entails } \exists P',\ P \xrightarrow{\alpha} P' \text{ and } P' \ \mathcal{R}\ Q' \ \}\end{aligned}$$

The so-called *bisimulations* are the post-fixpoints of $\mathbf{b}$, and *bisimilarity* ($\sim$) is its greatest-fixpoint.

An enhanced coinductive proof. Consider the following process definitions, and let us try to prove that $A \sim B$.

$$\begin{aligned} A &\triangleq a.b.D & B &\triangleq a.b.C \\ C &\triangleq \overline{a}.(A|C) & D &\triangleq \overline{a}.(B|D) \end{aligned}$$

Any bisimulation containing the pair $\langle A, B\rangle$ must be infinite. Instead, the companion of $\mathbf{b}$ allows us work with the finite relation $\mathcal{S} \triangleq \{\langle A, B\rangle, \langle C, D\rangle\}$: we have $\mathcal{S} \leqslant \mathbf{b}^\dagger\mathcal{S}$.

Indeed, we have $A \xrightarrow{a} b.D$ and $B \xrightarrow{a} b.C$. While the pair $\langle b.D, b.C\rangle$ does not belong to $\mathcal{S}$, we can use the following functions to cancel the b prefixes and to transpose C and D:

$$\begin{aligned} \mathbf{c}^{\cdot} &: \mathcal{R} \mapsto \{\langle \alpha.P,\ \alpha.Q\rangle \mid \alpha \text{ a label}, P \ \mathcal{R}\ Q\} \\ \mathbf{i} &: \mathcal{R} \mapsto \mathcal{R}^\circ = \{\langle Q,\ P\rangle \mid P \ \mathcal{R}\ Q\} \end{aligned}$$

We thus have $\langle A, B\rangle \in \mathbf{bic}^{\cdot}\mathcal{S}$. The function $\mathbf{i}$ is trivially compatible for $\mathbf{b}$, and we shall see in Section 4.6 that $\mathbf{c}^{\cdot}$ is below the companion of $\mathbf{b}$, written $\mathbf{t}$ in the sequel. Whence $\mathbf{ic}^{\cdot} \leqslant \mathbf{tt} \leqslant \mathbf{t}$, and thus $\langle A, B\rangle \in \mathbf{b}^\dagger\mathcal{S}$.

Similarly, we have $C \xrightarrow{\overline{a}} A|C$ and $D \xrightarrow{\overline{a}} B|D$, and we use the following function to cancel parallel composition and recover the two pairs from $\mathcal{S}$:

$$\mathbf{c}^{|} : \mathcal{R} \mapsto \{\langle P|P',\ Q|Q'\rangle \mid P\ \mathcal{R}\ Q, P'\ \mathcal{R}\ Q'\}$$

This function is below $\mathbf{t}$ (see Section 4.6 again), so that we get $\langle C, D\rangle \in \mathbf{bc}^{|}\mathcal{S} \leqslant \mathbf{b}^{\dagger}\mathcal{S}$.

Note that thanks to the companion, we only had to study transitions along labels a and $\overline{a}$, even though the processes at hand also perform transitions labelled b and τ. (For instance, we have $A \xrightarrow{ab\overline{a}\tau} b.C|B|D$). The fact that the starting processes cannot diverge one from the other using those actions is somehow factored once and for all, in the proofs that $\mathbf{c}^{\cdot}$ and $\mathbf{c}^{|}$ are valid enhancements.

Modularity. As pointed out by Hur et al. [68], working with the companion rather than with specific compatible functions is quite convenient: it does not require us to announce a global up-to technique up-front. (Here, something like $\mathbf{ic}^{\cdot} \vee \mathbf{c}^{|}$.) In each sub-case of the proof, we can just extract from the companion whatever is needed for that case. This approach is much more robust, especially in the context of computer-assisted proofs. Suppose for instance that one slightly changes the definition of D into $\overline{a}.(D|B)$. One can still conclude by reasoning up to commutativity of parallel composition, and this additional technique is already available in the companion: there is no need to update the declared up-to technique, one just needs to adjust the proof locally. (Of course one needs to prove that this new kind of enhancement is available in the companion, but this can be done separately, and once and for all.)

Code reuse. Although this was not needed in the previous example, one can also show that the following function is compatible:

$$\mathbf{j} :\ \mathcal{R} \mapsto \mathcal{R}\mathcal{R} = \{\langle P,\ R\rangle \mid \exists Q,\ P\ \mathcal{R}\ Q,\ Q\ \mathcal{R}\ R\}$$

Thus $\mathbf{j} \leqslant \mathbf{t}$, and together with (4.6), $\mathbf{jt} \leqslant \mathbf{t}$. In other words, for any relation R the relation $\mathbf{t}\mathcal{R}$ is transitive. Similarly, the constant-to-identity function $\hat{1}$ is compatible, so that $\mathbf{t}\mathcal{R}$ is always a reflexive relation. More generally, from $\mathbf{c}^{\cdot}, \mathbf{c}^{|}, \hat{1}, \mathbf{i}, \mathbf{j} \leqslant \mathbf{t}$ and $\mathbf{tt} \leqslant \mathbf{t}$, we deduce that for any relation $\mathcal{R}$, $\mathbf{t}\mathcal{R}$ is a congruence containing both $\mathcal{R}$ and $\sim$.

In the context of proof assistants, this simple realisation makes it possible to reuse standard technology for automating equational reasoning (e.g., in the Coq proof assistant, setoid rewriting).

Also note that since $\sim = \mathbf{t}\bot$, we obtain as a special case that bisimilarity is a congruence. In particular, once the aforementioned technology has been settled for $\mathbf{t}\mathcal{R}$ for an arbitrary $\mathcal{R}$, tools for equational reasoning about bisimilarity come for free.

4.3.2 Finite automata

In Chapter 2 we defined several coinductive algorithms for language equivalence of finite automata. It remained to show that some up-to techniques are sound. Given a DFA $\langle S, o, t\rangle$, define the following monotone function on relations on S:

$$\mathbf{b} : \ \mathcal{R} \mapsto \{\langle x, y\rangle \mid o(x) = o(y) \text{ and } \forall a \in \Sigma,\ t_a(x)\ \mathcal{R}\ t_a(y)\} \ .$$

The post-fixpoints of b precisely correspond to the bisimulations (Definition 2.1.1). Moreover, given a monotone function f, the post-fixpoints of bf correspond to the bisimulations up to f (Definition 2.1.4).

We have explained Hopcroft and Karp's algorithm in terms of bisimulations up to equivalence, and for that we needed that every bisimulation up to equivalence be contained in a bisimulation (Theorem 2.1.5). As in the case of CCS, we only need to show that the functions $\hat{1}$, $\mathbf{i}$, and $\mathbf{j}$ are compatible, which is straightforward. From the closure properties of the companion (Proposition 4.2.2), it follows that the equivalence closure function is contained in the companion, and thus a valid enhancement.

Similarly, to obtain up-to-context and up-to-congruence for a determinised NFA $\langle S, o^\sharp, t^\sharp\rangle$, we only have to show that the following function is compatible.

$$\mathbf{c}^+ : \mathcal{R} \mapsto \{\langle X{+}X',\ Y{+}Y'\rangle \mid X\ \mathcal{R}\ Y, X'\ \mathcal{R}\ Y'\}$$

This comes from the fact that the functions $t^\sharp$ and $o^\sharp$ are semilattice homomorphisms. As above, we deduce that the companion contains the contextual closure and congruence closure functions, whence Proposition 2.2.3 and Theorem 2.2.5.

4.3.3 The stream calculus

As a third example, we consider the *stream calculus*, as developed by Rutten [124]. Let us denote by $\mathbb{R}^\omega$ the set of *streams*, i.e., infinite sequences $\sigma, \tau \ldots$ of real numbers.

Together with the following function associating to each stream its first element and its tail, $\mathbb{R}^\omega$ is a final coalgebra for the functor $FX = \mathbb{R} \times X$

$$\begin{aligned} \mathbb{R}^\omega &\to \mathbb{R} \times \mathbb{R}^\omega \\ \sigma &\mapsto \langle \sigma_0, \sigma'\rangle \end{aligned}$$

One can thus define streams by *behavioural differential equations* (i.e., using F-coalgebras). For instance, the everywhere-0 stream $\hat{0}$ can be defined by the following equations:

$$\hat{0}_0 = 0 \qquad\qquad \hat{0}' = \hat{0}$$

Similarly, pointwise addition of streams can be defined by

$$(\sigma + \tau)_0 = \sigma_0 + \tau_0 \qquad\qquad (\sigma + \tau)' = \sigma' + \tau'$$

Shuffle product. Things become more interesting for more complex operations. Take for instance the *shuffle product* of streams, usually defined by the following formula:

$$(\sigma \otimes \tau)_n = \sum_{k=0}^{n} \binom{n}{k} \times \sigma_k \times \tau_{n-k}$$

This operation can alternatively be defined using the following differential equations, which no longer involve binomial coefficients:

$$(\sigma \otimes \tau)_0 = \sigma_0 \times \tau_0 \qquad\qquad (\sigma \otimes \tau)' = \sigma' \otimes \tau \;+\; \sigma \otimes \tau'$$

As noticed by Rutten, proving a simple property like associativity can be difficult with the former definition, as it would involve double summations of terms with several binomial coefficients. In contrast, one can give a straightforward coinductive proof.

Let $\mathbf{b}$ be the following (monotone) function on binary relations on streams:

$$\mathbf{b} : \; \mathcal{R} \mapsto \{\langle \sigma, \; \tau \rangle \mid \sigma_0 = \tau_0 \text{ and } \sigma' \; \mathcal{R} \; \tau'\}$$

One can check that its greatest fixpoint is the identity relation: $\langle \sigma, \tau \rangle \in \nu\mathbf{b}$ iff $\sigma = \tau$. One can thus prove stream equalities by coinduction.

As a trivial example consider commutativity of stream addition: it is immediate to see that the relation $\{\langle \sigma + \tau, \; \tau + \sigma \rangle \mid \sigma, \tau \in \mathbb{R}^\omega\}$ is a post-fixpoint of $\mathbf{b}$; this relation is thus contained in the identity, and stream addition is commutative.

Coming back to associativity of the shuffle product, we might accordingly try to use the following relation:

$$\mathcal{S} \triangleq \{\langle (\sigma \otimes \tau) \otimes \rho, \; \sigma \otimes (\tau \otimes \rho) \rangle \mid \sigma, \tau, \rho \in \mathbb{R}^\omega\}$$

Unfortunately, this relation is not a post-fixpoint of $\mathbf{b}$: assuming distributivity has already been proved, we have

$$((\sigma \otimes \tau) \otimes \rho)' = (\sigma' \otimes \tau) \otimes \rho \;+\; (\sigma \otimes \tau') \otimes \rho \;+\; (\sigma \otimes \tau) \otimes \rho'$$
$$(\sigma \otimes (\tau \otimes \rho))' = \sigma' \otimes (\tau \otimes \rho) \;+\; \sigma \otimes (\tau' \otimes \rho) \;+\; \sigma \otimes (\tau \otimes \rho')$$

and those two streams are not related by $\mathcal{S}$. Like in the previous example in CCS, we would like to cancel the two sums on both sides, in order to recover three pairs in $\mathcal{S}$. This is possible using the companion of $\mathbf{b}$: the following function is easily shown to be compatible for $\mathbf{b}$, so that $\mathcal{S} \leqslant \mathbf{b}^\dagger\mathcal{S}$.

$$\mathbf{c}^+ : \mathcal{R} \mapsto \{\langle \sigma + \rho, \; \tau + \omega \rangle \mid \sigma \; \mathcal{R} \; \tau, \; \rho \; \mathcal{R} \; \omega\}$$

We have thus obtained a straightforward proof of associativity of the shuffle product.

Exponentiation. Let us consider a third natural operation on streams: *exponentiation*, defined by the following differential equation:

$$e^{\sigma}_0 = e^{\sigma_0} \qquad\qquad e^{\sigma\prime} = \sigma' \otimes e^{\sigma}$$

As expected, we have $e^{\sigma+\tau} = e^{\sigma} \otimes e^{\tau}$. To prove it by following the same path as above, one needs to cancel a shuffle product, thus calling for the following function:

$$\mathbf{c}^{\otimes} : \mathcal{R} \mapsto \{\langle \sigma \otimes \rho,\ \tau \otimes \omega\rangle \mid \sigma\ \mathcal{R}\ \tau,\ \rho\ \mathcal{R}\ \omega\}$$

While this function is indeed below the companion of $\mathbf{b}$, it is not compatible for $\mathbf{b}$. To understand why, let us try to prove compatibility of this function, i.e., $\mathbf{c}^{\otimes}\mathbf{b} \leqslant \mathbf{b}\mathbf{c}^{\otimes}$. Let $\mathcal{R}$ be a relation. We have

$$\mathbf{c}^{\otimes}\mathbf{b}\mathcal{R} = \{\langle \sigma \otimes \rho,\ \tau \otimes \omega\rangle \mid \sigma\ \mathbf{b}\mathcal{R}\ \tau,\ \rho\ \mathbf{b}\mathcal{R}\ \omega\}$$

So assuming $\sigma\ \mathbf{b}\mathcal{R}\ \tau$ and $\rho\ \mathbf{b}\mathcal{R}\ \omega$, we have to show that $\langle \sigma \otimes \rho,\ \tau \otimes \omega\rangle \in \mathbf{b}\mathbf{c}^{\otimes}\mathcal{R}$. That $(\sigma \otimes \rho)_0 = (\tau \otimes \omega)_0$ is easy; the problem comes from the tails of those streams:

$$(\sigma \otimes \rho)' = \sigma' \otimes \rho\ +\ \sigma \otimes \rho'$$
$$(\tau \otimes \omega)' = \tau' \otimes \omega\ +\ \tau \otimes \omega'$$

First we need to cancel the sum operation (using $\mathbf{c}^{+}$). Second, while we have $\sigma'\ \mathcal{R}\ \tau'$ and $\rho'\ \mathcal{R}\ \omega'$ by assumption, we only have $\rho\ \mathbf{b}\mathcal{R}\ \omega$ and $\sigma\ \mathbf{b}\mathcal{R}\ \tau$. In the end, instead of $\mathbf{c}^{\otimes}\mathbf{b} \leqslant \mathbf{b}\mathbf{c}^{\otimes}$, we have

$$\mathbf{c}^{\otimes}\mathbf{b} \leqslant \mathbf{b}\mathbf{c}^{+}\mathbf{c}^{\otimes}(\mathbf{b} \vee 1) \tag{4.12}$$

We shall see in the following section that such a result nevertheless ensures that the function $\mathbf{c}^{\otimes}$ is below the companion of $\mathbf{b}$, and can thus safely be used in enhanced coinductive proofs about streams.

4.4 Compatibility up-to

In this section we show that the companion is a coinductive object. This gives us powerful proof techniques to obtain enhancements.

Definition 4.4.1. *Let B be the following function from monotone functions on X to monotone functions on X:*

$$B : [X \to X] \to [X \to X]$$
$$g \mapsto \bigvee_{fb \leqslant bg} f$$

Lemma 4.4.2. *B is monotone, and for all functions $f, g : [X \to X]$,*

$$f \leqslant Bg \quad \textit{iff} \quad fb \leqslant bg \tag{4.13}$$

In particular, f is compatible if and only if it is a post-fixpoint of B, so that the companion is the greatest fixpoint of B:

$$t = \nu B \tag{4.14}$$

We can thus reuse the machinery from Section 4.2 with B, in the second-order lattice $[X \to X]$. Let T be the companion of B, and let $B^\dagger \triangleq BT$.

In the previous section, in the example about streams, we had a first function $\mathbf{c}^+$, which was compatible and thus trivially below the companion. In contrast, we claimed that the function $\mathbf{c}^\otimes$ was below the companion, although it is not compatible: we do not have $\mathbf{c}^\otimes\mathbf{b} \leqslant \mathbf{b}\mathbf{c}^\otimes$, i.e., $\mathbf{c}^\otimes \leqslant \mathbf{B}\mathbf{c}^\otimes$ (where $\mathbf{B}$ is the higher-order function associated to the function $\mathbf{b}$ by Definition 4.4.1). Instead, we had $\mathbf{c}^\otimes \leqslant \mathbf{B}(\mathbf{c}^+\mathbf{c}^\otimes(\mathbf{b} \vee 1))$. Using the results below, we will deduce from this inequality that $\mathbf{c}^\otimes \leqslant \mathbf{B}^\dagger\mathbf{c}^\otimes$, so that $\mathbf{c}^\otimes \leqslant \nu\mathbf{B}$: the function $\mathbf{c}^\otimes$ indeed lives below the companion.

In a sense, we face the standard scenario of bisimulation proofs, but in the lattice of monotone functions: $\mathbf{c}^\otimes$ is an obvious coinductive candidate, but it is too weak, we should strengthen it to get a post-fixpoint. Luckily, instead of doing so, we can use an enhancement to build on the knowledge accumulated so far about the companion (in this case, that it contains $\mathbf{c}^+$, amongst other things.)

Getting back to the abstract framework, the second-order companion T enjoys many good properties, listed in Proposition 4.4.4 below. We first need to establish a compatibility result for B.

Lemma 4.4.3. *The function $S : f \mapsto ff$ is compatible for B.*

Proposition 4.4.4. *For any function $f : [X \to X]$, we have*

$$t \leqslant Tf \tag{4.15}$$
$$b \leqslant Tf \tag{4.16}$$
$$1 \leqslant Tf \tag{4.17}$$
$$f \leqslant Tf \tag{4.18}$$
$$TTf \leqslant Tf \tag{4.19}$$
$$(Tf)(Tf) \leqslant Tf \tag{4.20}$$

In particular, Tf is always an idempotent function.

Coming back to the example about the shuffle product on streams, write $\mathbf{t}$ and $\mathbf{T}$ for the companions of $\mathbf{b}$ and $\mathbf{B}$. It is now straightforward to

check that $\mathbf{c}^+\mathbf{c}^\otimes(\mathbf{b} \vee 1) \leqslant \mathbf{T}\mathbf{c}^\otimes$:

$$\begin{aligned}
\mathbf{c}^+\mathbf{c}^\otimes(\mathbf{b} \vee 1) &\leqslant \mathbf{t}\mathbf{c}^\otimes(\mathbf{b} \vee 1) && (\mathbf{c}^+ \text{ is compatible}) \\
&\leqslant (\mathbf{T}\mathbf{c}^\otimes)\mathbf{c}^\otimes(\mathbf{T}\mathbf{c}^\otimes \vee \mathbf{T}\mathbf{c}^\otimes) && \text{(by (4.15), (4.16), and (4.17))} \\
&\leqslant (\mathbf{T}\mathbf{c}^\otimes)(\mathbf{T}\mathbf{c}^\otimes)(\mathbf{T}\mathbf{c}^\otimes) && \text{(by (4.18))} \\
&\leqslant \mathbf{T}\mathbf{c}^\otimes && \text{(by (4.20) twice)}
\end{aligned}$$

So from (4.12) we deduce $\mathbf{c}^\otimes \leqslant \mathbf{B}^\dagger\mathbf{c}^\otimes$ and thus $\mathbf{c}^\otimes \leqslant \mathbf{t}$, as announced earlier.

(Note that we chose to make the function $\mathbf{c}^+\mathbf{c}^\otimes(\mathbf{b} \vee 1)$ explicit here for the sake of explanation. In a direct proof, one would prove $\mathbf{c}^\otimes \leqslant \mathbf{B}^\dagger\mathbf{c}^\otimes$ by extracting the required components out of $\mathbf{T}\mathbf{c}^\otimes$ on the fly, exactly as we did in Section 4.3.1 but at the second-order level.)

4.5 Symmetry arguments

We now give a rather general result allowing to exploit symmetry arguments in various proofs. This result formally justifies standard practice in bisimulation proofs with paper and pencil. When it comes to formal, mechanised, proofs, it is crucial to have such results, to factor the code and avoid cut-and-paste.

Let $i : [X \to X]$ be a monotone involution on X:

$$ii = 1 \tag{4.21}$$

Call an element $x \in X$ *symmetric* if $ix = x$ (which is equivalent to $ix \leqslant x$ thanks to (4.21)). Call a function $f : [X \to X]$ *symmetric* if $fi = if$ (which is equivalent to f being compatible for i, again using (4.21)).

As a concrete example in the lattice of binary relations, the natural candidate for the function i is the transposition function from Section 4.3.1:

$$\mathbf{i} : \mathcal{R} \mapsto \mathcal{R}^{-1} = \{\langle Q,\ P\rangle \mid P\ \mathcal{R}\ Q\}$$

With such a choice, a relation $\mathcal{R}$ is symmetric if $\mathcal{R}^{-1} = \mathcal{R}$, and a function f is symmetric if $f(\mathcal{R}^{-1}) = f(\mathcal{R})^{-1}$ for all relations $\mathcal{R}$.

When the function b is symmetric, the following proposition can be used to factor proofs about symmetric candidates, both at the level of elements (X) and at the level of enhancements ($[X \to X]$). We instantiate this result in the following section, when reasoning about bisimilarity in CCS and the π-calculus.

Proposition 4.5.1. *Let $s : [X \to X]$ be a monotone function such that the function b decomposes as follows:*

$$b = s \wedge isi$$

Then νb is symmetric, $it = t$, and

$$\frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \frac{P \xrightarrow{\alpha} P'}{(\nu a)P \xrightarrow{\alpha} (\nu a)P'} \alpha \neq a, \overline{a} \qquad \frac{!P|P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'}$$

Figure 4.2: Remaining rules for the LTS of CCS.

1. *for all $x, y \in X$ with x symmetric, $x \leqslant bty$ iff $x \leqslant sty$;*
2. *for all $f, g : [X \to X]$ with f symmetric, $fb \leqslant bTg$ iff $fb \leqslant sTg$.*

4.6 Example: up-to congruence for CCS

In this section, we illustrate the above framework by applying it to recover up-to-context and up-to congruence for CCS, in a compositional way. Indeed, thanks to the closure properties of the companion, it suffices to show that the functions associated to each syntactic construction are below $\mathbf{t}$ to obtain the full context closure function. Combined with the fact that the transposition function $\mathbf{i}$ and the squaring function $\mathbf{j}$ are also below $\mathbf{t}$, we will immediately obtain the full congruence closure.

For the sake of completeness, let us consider here the entire calculus of communicating systems. In addition to the operations used in Section 4.3.1, there is choice, name restriction, and replication.

$$P, Q ::= A \mid 0 \mid \alpha.P \mid P|P \mid P + P \mid (\nu a)P \mid !P$$
$$\alpha, \beta ::= a \mid \overline{a} \mid \tau$$

The additional rules for the labelled transition system are given in Figure 4.2. (The symmetrical rule for choice is omitted.)

Recall the function $\mathbf{b}$ from Section 4.3.1, which we used to define bisimilarity. Let $\mathbf{s}$ be the "first half" of this function:

$$\mathbf{s} : \mathcal{R} \mapsto \{\langle P, Q\rangle \mid \forall \alpha, P', P \xrightarrow{\alpha} P' \text{ entails } \exists Q',\ Q \xrightarrow{\alpha} Q' \text{ and } P' \mathcal{R} Q'\}$$

With such a function, we only play from left to right. The post-fixpoints of $\mathbf{s}$ are the *simulations*, and its greatest fixpoint is *similarity*. The composite function $\mathbf{isi}$ corresponds to simulations again, but played from right to left. As expected the function $\mathbf{b}$ for bisimilarity thus decomposes as required in Proposition 4.5.1:

$$\mathbf{b} = \mathbf{s} \wedge \mathbf{isi} \tag{4.22}$$

Applied in this setting, equivalence (1) from Proposition 4.5.1 is not so surprising: when analysing the transitions of a symmetric bisimulation

candidate, we can restrict ourselves to the left-to-right part of the bisimulation game. Note that thanks to the companion, we do not need y to be symmetric (because ty is). The second equivalence (2) from Proposition 4.5.1 is quite important in the sequel: one can also restrict ourselves to the left-to-right part of the bisimulation game when analysing the behaviour of a potential enhancement, provided it is symmetric.

Following closely the syntax of the calculus, we define the following functions on binary relations:

$$\begin{aligned}
\mathbf{c}^{\cdot} &: \mathcal{R} \mapsto \{\langle \alpha.P, \alpha.Q\rangle \mid \alpha \text{ a label}, P\ \mathcal{R}\ Q\}\\
\mathbf{c}^{|} &: \mathcal{R} \mapsto \{\langle P|P', Q|Q'\rangle \mid P\ \mathcal{R}\ Q,\ P'\ \mathcal{R}\ Q'\}\\
\mathbf{c}^{+} &: \mathcal{R} \mapsto \{\langle P+P', Q+Q'\rangle \mid P\ \mathcal{R}\ Q,\ P'\ \mathcal{R}\ Q'\}\\
\mathbf{c}^{!} &: \mathcal{R} \mapsto \{\langle !P, !Q\rangle \mid P\ \mathcal{R}\ Q\}\\
\mathbf{c}^{\nu} &: \mathcal{R} \mapsto \{\langle (\nu a)P, (\nu a)Q\rangle \mid a \text{ a name}, P\ \mathcal{R}\ Q\}
\end{aligned}$$

The functions $\mathbf{c}^{|}$ and $\mathbf{c}^{\cdot}$ have already been defined in Section 4.3.1; we include them here to emphasise the uniformity of those definitions.

Let c be one of the above functions; we want to prove $c \leqslant \mathbf{t}$, where $\mathbf{t}$ is the companion of $\mathbf{b}$. From the results of Section 4.4, it thus suffices to prove $c\mathbf{b} \leqslant \mathbf{b}\mathbf{T}c$, where $\mathbf{T}$ is the companion of the second-order function $\mathbf{B}$ associated to $\mathbf{b}$. And since all the above functions are symmetric, it suffices by Proposition 4.5.1(2) to prove

$$c\mathbf{b} \leqslant \mathbf{s}\mathbf{T}c$$

For the "dynamic" operations that disappear after a single transition (functions $\mathbf{c}^{\cdot}$ and $\mathbf{c}^{+}$), we actually do not need coinduction at all. Routine computations lead to

$$\mathbf{c}^{\cdot}\mathbf{b} \leqslant \mathbf{s}\mathbf{b} \qquad\qquad \mathbf{c}^{+}\mathbf{b} \leqslant \mathbf{s}$$

(For $\mathbf{c}^{\cdot}$, we have $\mathbf{c}^{\cdot} \leqslant \mathbf{s}$.) This is fine because we have both $\mathbf{b} \leqslant \mathbf{T}\bot$ (4.16) and $1 \leqslant \mathbf{T}\bot$ (4.17).

Instead, we do need coinduction for the "static" operations, which persist through transitions. The simplest is name restriction: we have $\mathbf{c}^{\nu}\mathbf{b} \leqslant \mathbf{s}\mathbf{c}^{\nu}$, and $\mathbf{c}^{\nu} \leqslant \mathbf{T}\mathbf{c}^{\nu}$ by (4.18). Parallel composition requires more care, we give a detailed proof to better illustrate our method.

Lemma 4.6.1. *We have* $\mathbf{c}^{|}\mathbf{b} \leqslant \mathbf{s}\mathbf{T}\mathbf{c}^{|}$, *whence* $\mathbf{c}^{|} \leqslant \mathbf{t}$.

Proof. Let $\mathcal{R}$ be a relation, an let P, R, Q, S be processes such that $\langle P, Q\rangle$ and $\langle R, S\rangle$ belong to $\mathbf{b}\mathcal{R}$. We have to show that $\langle P|R, Q|S\rangle$ belongs to $\mathbf{s}\mathbf{T}\mathbf{c}^{|}$. Thus suppose that $P|R \xrightarrow{\alpha} P_0$ and let us find some Q_0 such that $Q|S \xrightarrow{\alpha} Q_0$ and $\langle P_0, Q_0\rangle \in \mathbf{T}\mathbf{c}^{|}\mathcal{R}$. There are four cases according to the rules of parallel composition (Figure 4.1):

1. $P_0 = P'|R'$ with $\alpha = \tau$, and for some name a, $P \xrightarrow{a} P'$, and $R \xrightarrow{\overline{a}} R'$. From our assumptions about $\langle P, Q\rangle$ and $\langle R, S\rangle$, we obtain processes Q' and S' such that $Q \xrightarrow{a} Q'$, $S \xrightarrow{\overline{a}} S'$, $P' \ \mathcal{R} \ Q'$ and $R' \ \mathcal{R} \ S'$. We deduce that $Q|S \xrightarrow{\tau} Q'|S'$, and the pair $\langle P'|R', Q'|S'\rangle$ belongs to $\mathbf{c}^{|}\mathcal{R}$ and hence $\mathbf{Tc}^{|}\mathcal{R}$ by (4.18).

2. same as above but with a and $\overline{a}$ exchanged.

3. $P_0 = P'|R$ with $P \xrightarrow{\alpha} P'$. From the hypothesis about the pair $\langle P, Q\rangle$ we obtain a process Q' such that $Q \xrightarrow{a} Q'$ and $P' \ \mathcal{R} \ Q'$. We deduce that $Q|S \xrightarrow{\alpha} Q'|S$, and it remains to show

$$\langle P'|R, Q'|S\rangle \in \mathbf{Tc}^{|}\mathcal{R}$$

   This is not as direct as before: rather than $R \ \mathcal{R} \ S$, we have $R \ \mathbf{b}\mathcal{R} \ S$. From (4.18) and (4.20), we have $\mathbf{c}^{|}\mathbf{Tc}^{|} \leqslant \mathbf{Tc}^{|}$. Therefore, it suffices to show that $P' \ \mathbf{Tc}^{|}\mathcal{R} \ Q'$ and $R \ \mathbf{Tc}^{|}\mathcal{R} \ S$. The former holds thanks to (4.17); for the latter we use (4.16) instead.

4. $P_0 = P|R'$ with $R \xrightarrow{\alpha} R'$. This case is handled as above. □

(Note that the above proof amounts to proving $\mathbf{c}^{|}\mathbf{b} \leqslant \mathbf{sc}^{|}(\mathbf{b} \vee 1)$ and then showing that $\mathbf{c}^{|}(\mathbf{b} \vee 1) \leqslant \mathbf{Tc}^{|}$ using the generic properties of $\mathbf{T}$—Proposition 4.4.4.)

We skip replication in this manuscript. This operation is quite challenging as far as up-to techniques are concerned: there was a slight mistake in [128], it requires specific rule formats [121], and people formalising up-to techniques in proof assistants have eluded this operation so far [38, 102]. Nevertheless, the techniques developed above allow us to give a much cleaner proof, thus amenable to formalisation [112]. The pi-calculus can also be handled in a streamlined way, despite the various subtleties arising due to the input prefix and to scope-extrusion [112].

4.7 Respectful vs. compatible

Before turning to parameterized coinduction, we discuss a historical peculiarity which has been causing some troubles since the introduction of up-to techniques, and which we can nicely resolve by using the companion function.

When Sangiorgi studied the bisimulation proof technique [126], he introduced the notion of *respectful* function to obtain compositionality results. With the present notation, a monotone function $f : [X \to X]$ is respectful (for b) if for all $x, y \in X$,

$$x \leqslant y \text{ and } x \leqslant by \text{ entail } fx \leqslant bfy \ .$$

Let $b' \triangleq b \wedge 1$ (i.e., $b'y = by \wedge y$). Without the assumption $x \leqslant y$, respectfulness would be equivalent to compatibility (for b). With this assumption, it is equivalent to compatibility for b'.

One can easily show that any compatible function (for b) is respectful, but some interesting respectful functions are not compatible. This is the reason why Sangiorgi needed this refinement. For instance the context closure function in CCS is respectful but not compatible. Hur et al. used respectfulness for the same reason [68].

In our own previous work [104, 116, 89], some of which with Sangiorgi, we found that the theory of plain compatibility was somewhat nicer to develop than that of respectfulness, so that we proposed to use the function b' rather than b when necessary (doing so is always possible). Although this was not their only reason, Parrow and Pohjola also chose a function b such that $b = b'$ in their theory of up-to techniques for the psi-calculus [102].

In this paper, we used the most natural function $\mathbf{b}$ to define strong bisimilarity in CCS, and this function does not satisfy $\mathbf{b} = \mathbf{b}'$. So how is it possible that we could obtain up-to context?

The point is that with the companion function, we do not need the up-to context function to be compatible stricto-senso. It just has to be below the companion function t. For instance, in our proof for the parallel composition operation in CCS, we obtained $\mathbf{c}^{|}\mathbf{b} \leqslant \mathbf{b}\mathbf{T}\mathbf{c}^{|}$, which does not entail compatibility of $\mathbf{c}^{|}$.This contrasts with the literature, where we would prove $\mathbf{c}^{|}\mathbf{b}' \leqslant \mathbf{b}'\mathbf{c}^{|}$, i.e., that $\mathbf{c}^{|}$ is respectful (compatible for $\mathbf{b}'$).

That we can recover up-to context in CCS and π without switching to $\mathbf{b}'$ is not a coincidence: the greatest respectful function always coincides with the greatest compatible function:

Proposition 4.7.1. *Let t' be the companion of $b' = b \wedge 1$. We have $t' = t$ and $b'^{\dagger} = b^{\dagger}$.*

In other words, the historical trade-off between b and b', or compatibility and respectfulness, is irrelevant. The functions b and b' lead to the same coinductive proof principle once enhanced with their companion. One can actually go even further and show that obtaining specific up-to techniques is equally hard with b' and b: their (enhanced) second-order proof principles also collapse.

Proposition 4.7.2. *Let B' be the second-order function associated to b', and let T' be the companion of B' (so that $\nu B' = t' = T'\bot$). We have $T' = T$ and $B'^{\dagger} = B^{\dagger}$.*

4.8 Parameterized coinduction

Recall the coinductive proof principle, as provided by Knaster-Tarski's theorem:

$$\frac{x \leqslant y \leqslant by}{x \leqslant \nu b}$$

This approach has an important drawback: the need to define the coinduction invariant (y) up-front. This does not match the standard practice, where the coinductive predicate is obtained incrementally from x, by progressively extending it until it becomes a post-fixpoint. In the context of a paper proof, one can always gather the final coinductive predicate a posteriori, to display it at the beginning of the proof. In the context of interactive formal proofs, this becomes really inconvenient.

To solve this problem, Hur et al. proposed to use *parameterized coinduction* [68]. The trick consists in defining an auxiliary function $G_b : [X \to X]$ with the following properties:

$$G_b \bot = \nu b \tag{4.23}$$

$$G_b x = b(x \vee G_b x) \tag{4.24}$$

$$y \leqslant G_b(y \vee x) \;\Rightarrow\; y \leqslant G_b x \tag{4.25}$$

Concretely, they define $G_b x$ as the greatest fixpoint of the function mapping an element z to $b(z \vee x)$. They also show how to use up-to techniques with parameterized coinduction, using the greatest respectful function (which coincides with t, according to Section 4.7). As we do here the idea is to switch to $b^\dagger = bt$, and thus they use $G_{b^\dagger}$ rather than G_b. Doing so leads to a fourth equation [68, Theorem 13]:

$$G_{b^\dagger} = tG_{b^\dagger} \tag{4.26}$$

Surprisingly, we actually have

Theorem 4.8.1. $G_{b^\dagger} = b^\dagger$.

(This result follows from Theorem 4.8.2 below; note that $G_b \neq b$ in general: the companion plays a crucial role in this result.)

This means we do not need the machinery of the $G_{b^\dagger}$ function to implement parameterized coinduction; it is already provided by the companion. Following this idea, we give an alternative presentation of parameterized coinduction.

Let us first prove the following counterpart to (4.25):

Theorem 4.8.2. *For all $x, y \in X$, if $y \leqslant bt(y \vee x)$ then $y \leqslant tx$.*

$$\frac{y \leqslant t\bot}{y \leqslant \nu b}\ \textsc{Init} \qquad \frac{y \leqslant x}{y \leqslant tx}\ \textsc{Done}$$

$$\frac{y \leqslant ftx \qquad f \leqslant t}{y \leqslant tx}\ \textsc{Up to } f \qquad \frac{y \leqslant bt(y \vee x)}{y \leqslant tx}\ \textsc{CoInd}$$

Figure 4.3: A proof system for parameterized enhanced coinduction.

Proof. Assume $y \leqslant bt(y \vee x)$ (H), and let $f : z \mapsto \bigvee_{x \leqslant z} y$. This function maps the points above x to y and all other points to the bottom element. In particular, $fx = y$, so that we have to show $fx \leqslant tx$.

Abstracting over x, we actually show that $f \leqslant t$. To this end, we use second-order coinduction up-to, and we prove $f \leqslant B^\dagger f$, i.e, $fb \leqslant bTf$. Let $z \in X$. If $x \not\leqslant bz$, then $fbz = \bot \leqslant b(Tf)z$. Otherwise, assume $x \leqslant bz$ (H'); we have

$$\begin{aligned}
fbz &= y && \text{(by definition of } f \text{ and } (H'))\\
&\leqslant bt(y \vee x) && \text{(by } (H))\\
&= bt(fx \vee x) && \text{(by definition of } f)\\
&\leqslant bt(fbz \vee bz) && \text{(by } (H'))\\
&= bt(fb \vee b)z
\end{aligned}$$

We easily check that $t(fb \vee b) \leqslant Tf$ using Proposition 4.4.4, so that we have $fb \leqslant bTf$, as required. □

Intuitively, tx contains everything that can safely be deduced from x, not necessarily in a guarded way. In particular, x can be deduced from tx. Instead, $btx = G_{b^\dagger}x$ is more restrictive and corresponds to guarded deductions only: we do not have $x \leqslant btx$ in general. With this intuition, the above theorem reads as follows: to deduce y from x, one can assume y provided one switches to guarded deductions.

This leads us to the "proof system" given in Figure 4.3. The four rules are valid: if their premises hold, so do their conclusion. The first one is for initialisation: to prove that y is below the greatest fixpoint, deduce it from the empty context. The second rule is an axiom rule: if y belongs to the context x, then we can deduce y. The third one makes it possible to use any enhancement known to be below the companion. Typically, when some congruence closure is below the companion, this rule makes it possible to use equational (or inductive) reasoning. The fourth rule is just Theorem 4.8.2: it corresponds to an actual coinductive step. Also note that since $b \leqslant t$, the third rule can be used to play one step of the bisimulation game, without

storing the current value of y in the context. Doing so corresponds to using Equation (4.24) from Hur et al.' formalism.

To give an example, let us revisit the example from Section 4.3.1. We wanted to prove that $A \sim B$, and we guessed that the relation $\{\langle A, B\rangle, \langle C, D\rangle\}$ could be used as a bisimulation candidate, thanks to several enhancements. With the proof system from Figure 4.3, we can give a parameterized-style proof, where we do not guess the bisimulation candidate in advance.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\langle A, B\rangle \in \{\langle C, D\rangle, \langle A, B\rangle\}}{\langle A, B\rangle \in \mathbf{t}\{\langle C, D\rangle, \langle A, B\rangle\}}\,(\textsc{Done}) \quad \dfrac{\langle C, D\rangle \in \{\langle C, D\rangle, \langle A, B\rangle\}}{\langle C, D\rangle \in \mathbf{t}\{\langle C, D\rangle, \langle A, B\rangle\}}\,(\textsc{Done})}{\langle A|C, B|D\rangle \in \mathbf{t}\{\langle C, D\rangle, \langle A, B\rangle\}}\,(\textsc{Up to } \mathbf{c}^{|} \leqslant \mathbf{t})}{\langle C, D\rangle \in \mathbf{bt}\{\langle C, D\rangle, \langle A, B\rangle\}}\,(\textsc{Def. of } \mathbf{b})}{\langle C, D\rangle \in \mathbf{t}\langle A, B\rangle}\,(\textsc{CoInd})}{\langle D, C\rangle \in \mathbf{t}\langle A, B\rangle}\,(\textsc{Up to } \mathbf{i} \leqslant \mathbf{t})}{\langle b.D, b.C\rangle \in \mathbf{t}\langle A, B\rangle}\,(\textsc{Up to } \mathbf{c}^{\cdot} \leqslant \mathbf{t})}{\langle A, B\rangle \in \mathbf{bt}\langle A, B\rangle}\,(\textsc{Def. of } \mathbf{b})}{\langle A, B\rangle \in \mathbf{t}\bot}\,(\textsc{CoInd})}{A \sim B}\,(\textsc{Init})$$

As previously, the required enhancements do not need to be declared upfront, they are extracted from the companion using the rule (Up to), when needed. We left aside the proofs of $\mathbf{c}^{\cdot}, \mathbf{c}^{|}, \mathbf{i} \leqslant \mathbf{t}$ in the above example, because we obtained them once and for all in Sections 4.3.1 and 4.6. Note however that the third rule (Up to) allows us to jump to the next level in the middle of a proof: since $t = \nu B = T\bot$, one can fulfil its second premise by using the same proof system. In fact, by the results of Section 4.4, these four rules cover not only enhanced coinduction, but also the enhancements themselves. Nothing prevents us from continuing with the next level again, although we did not find any concrete application so far.

4.9 Extensional characterisation of the companion

At the same time we submitted this work about the largest compatible function [112], Parrow and Weber submitted a note [100] where they give a nice and intriguing characterisation of the largest respectful function, i.e., thanks to Proposition 4.7.1, of the companion. They use Sangiorgi's notion of progression [126, 128]; we recast their result here in terms of monotone functions.

Let us first recall the abstract counterpart to Milner's stratification of bisimilarity. Given the monotone function $b : [X \to X]$, define a sequence

$(b_\alpha)_\alpha$ of elements of X indexed by ordinals, by transfinite induction:

$$b_{\alpha+1} \triangleq bb_\alpha \qquad\qquad b_\lambda \triangleq \bigwedge_{\alpha<\lambda} b_\alpha$$

The initialisation is given by the special case $\lambda = 0$ ($b_0 = \top$). This sequence is decreasing and it always stationates at some ordinal, typically ω when b is co-continuous. It is well-known that the greatest fixpoint of b is the limit of this sequence (see, e.g., Rubin and Rubin [122]):

Theorem 4.9.1. *We have $\nu b = \bigwedge_\alpha b_\alpha$.*

The counterpart to Parrow and Weber's characterisation is the following:

Theorem 4.9.2. *For all $x \in X$, we have $tx = \bigwedge\{b_\alpha \mid x \leqslant b_\alpha\}$.*

In words, the companion maps a point to the smallest element of the stratification that contains it. When $x = \bot$, we recover Theorem 4.9.1; when $x = \nu b$, we recover Corollary 4.2.4.

4.10 Discussion

The presented theory as well as the examples have been formalised as a Coq library [114].

GSOS is a rule format that was introduced to ensure congruence properties for bisimilarity [16]. We have recently shown that it also gives rise to a respectful contextual closure function: up-to context can always be used for GSOS specifications [19]. In light of the present results, we can deduce that such a closure is always contained in the companion in two ways: first by reusing the existing proofs of respectfulness and switching to the companion by Proposition 4.7.1; second, by an easy generalisation of our treatment of CCS (Section 4.6).

The latter approach is rather intriguing from a categorical point of view. Indeed, GSOS specifications can be seen abstractly as distributive laws [138, 15]. More precisely, when Σ is the functor corresponding to a term signature, and when $FX = (P_\omega X)^A$ is the functor whose coalgebras are the finitary branching LTS with labels in A, we have that a GSOS specification is exactly a distributive law

$$\Sigma(F \times Id) \Rightarrow FT$$

The fact that we have $F \times Id$ on the left makes it quite natural to consider respectfulness rather than compatibility when studying up-to-context techniques in this setting. The present results however suggest that there might be a more direct path, by using a categorical version of the companion.

Similarly, we would like to understand the categorical counterpart to Theorem 4.9.1 (the extensional characterisation of the companion through the stratification of the greatest fixpoint).

Notes

Chapter 1 about relation algebra is a translation of my course notes for the EJCIM'16 research school [113] (in French). The new results announced there were obtained with my current PhD student, Paul Brunet. (Precise complexity of Kleene algebra with converse [30, 31], and new automata model for Kleene allegories [32].)

The first part of Chapter 2 (Sections 2.1 and 2.2 about coinductive algorithms for DFA and NFA) is largely based on our work with Filippo Bonchi [21, 22]. An implementation, Coq proofs, and a web applet are available [108]. We subsequently extended this work with Alexandra Silva and Georgianna Caltais to handle various decorated trace semantics, including must-testing [18].

The second part of Chapter 2, about symbolic algorithms (Section 2.3) roughly corresponds to the first part of [111]. An implementation and a web applet for KAT are available [109].

Chapter 3 about automation in the Coq proof assistant is based on [110]. We initiated this work with Thomas Braibant, my first PhD student, with whom we implemented a first Coq tactic for Kleene algebra [26, 28], as well as tools for rewriting modulo associativity and commutativity [27]. We recently used the new library [115] with Paul Brunet and Insa Stucke to prove the correctness of several algorithms from graph theory, in a relation algebraic way [33].

The last chapter, about abstract coinduction, is based on [112]; Coq proofs are also available [114]. In another line of work with Daniela Petrisan, Jurriaan Rot, and Filippo Bonchi, we gave a categorical account to enhancements of the coinductive proof method [19, 20]. There we start from the observation that bialgebras for a distributive law give rise to well-behaved systems [138], and we use fibrations to state simple conditions under which generalisations of the various up-to techniques discussed in Chapters 2 and 4 are compatible.

Bibliography

[1] M. Abadi and A. D. Gordon. A bisimulation method for cryptographic protocols. *Nord. J. Comput.*, 5(4):267–, 1998.

[2] P. A. Abdulla, Y.-F. Chen, L. Holík, R. Mayr, and T. Vojnar. When simulation meets antichains. In *Proc. TACAS*, volume 6015 of *LNCS*, pages 158–174. Springer, 2010.

[3] S. Abramsky. The Lazy Lambda Calculus. In D. A. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison Wesley, 1990.

[4] S. F. Allen, R. L. Constable, D. J. Howe, and W. E. Aitken. The semantics of reflected proof. In *Proc. LICS*, pages 95–105. IEEE, 1990.

[5] J. B. Almeida, N. Moreira, D. Pereira, and S. M. de Sousa. Partial derivative automata formalized in Coq. In *Proc. CIAA*, volume 6482 of *LNCS*, pages 59–68. Springer, 2010.

[6] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. Netkat: semantic foundations for networks. In *Proc. POPL*, pages 113–126. ACM, 2014.

[7] H. Andréka and S. Mikulás. Axiomatizability of positive algebras of binary relations. *Algebra Universalis*, 66(1):7–34, 2011.

[8] H. Andréka, S. Mikulás, and I. Németi. The equational theory of Kleene lattices. *Theoretical Computer Science*, 412(52):7099–7108, 2011.

[9] H. Andréka and D. Bredikhin. The equational theory of union-free algebras of relations. *Algebra Universalis*, 33(4):516–532, 1995.

[10] A. Angus and D. Kozen. Kleene algebra with tests and program schematology. Technical Report TR2001-1844, CS Dpt., Cornell University, July 2001.

[11] V. M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319, 1996.

[12] A. Armstrong and G. Struth. Automated reasoning in higher-order regular algebra. In *Proc. RAMiCS*, volume 7560 of *LNCS*, pages 66–81. Springer, 2012.

[13] A. Asperti. A compact proof of decidability for regular expression equivalence. In *Proc. ITP*, volume 7406 of *LNCS*, pages 283–298. Springer, 2012.

[14] L. Bachmair, I. V. Ramakrishnan, A. Tiwari, and L. Vigneron. Congruence closure modulo associativity and commutativity. In *Proc. FroCoS*, volume 1794 of *LNCS*, pages 245–259. Springer, 2000.

[15] F. Bartels. Generalised coinduction. *Mathematical Structures in Computer Science*, 13(2):321–348, 2003.

[16] B. Bloom, S. Istrail, and A. R. Meyer. Bisimulation can't be traced. In *Proc. POPL*, pages 229–239. ACM, 1988.

[17] S. L. Bloom, Z. Ésik, and G. Stefanescu. Notes on equational theories of relations. *Algebra Universalis*, 33(1):98–126, 1995.

[18] F. Bonchi, G. Caltais, D. Pous, and A. Silva. Brzozowski's and up-to algorithms for must testing. In *Proc. APLAS*, volume 8301 of *LNCS*, pages 1–16. Springer, 2013.

[19] F. Bonchi, D. Petrisan, D. Pous, and J. Rot. Coinduction up-to in a fibrational setting. In *Proc. CSL-LICS*, pages 20:1–20:9. ACM, 2014.

[20] F. Bonchi, D. Petrisan, D. Pous, and J. Rot. Lax bialgebras and up-to techniques for weak bisimulations. In *Proc. CONCUR*, volume 42 of *LIPIcs*, pages 240–253. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[21] F. Bonchi and D. Pous. Checking NFA equivalence with bisimulations up to congruence. In *Proc. POPL*, pages 457–468. ACM, 2013.

[22] F. Bonchi and D. Pous. Hacking nondeterminism with induction and coinduction. *Commun. ACM*, 58(2):87–95, Jan. 2015.

[23] M. Boreale and D. Sangiorgi. Bisimulation in name-passing calculi without matching. In *Proc. LICS*. IEEE, 1998.

[24] A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In *Proc. CAV*, volume 3114 of *LNCS*, pages 372–386. Springer, 2004.

[25] R. Boyer and J. Moore. Metafunctions: proving them correct and using them efficiently as new proof procedures. In *The Correctness Problem in Computer Science*, pages 103–184. NY: Academic Press, 1981.

[26] T. Braibant and D. Pous. An efficient Coq tactic for deciding Kleene algebras. In *Proc. 1st ITP*, volume 6172 of *LNCS*, pages 163–178. Springer, 2010.

[27] T. Braibant and D. Pous. Tactics for reasoning modulo AC in Coq. In *Proc. 1st CPP*, volume 7086 of *LNCS*, pages 167–182. Springer, 2011.

[28] T. Braibant and D. Pous. Deciding Kleene algebras in Coq. *Logical Methods in Computer Science*, 8(1):1–16, 2012.

[29] P. Brunet. A Kleene theorem for Petri automata. Submitted, 2016.

[30] P. Brunet and D. Pous. Kleene algebra with converse. In *Proc. RAMiCS*, volume 8428 of *LNCS*, pages 101–118. Springer, 2014.

[31] P. Brunet and D. Pous. Algorithms for Kleene algebra with converse. *Journal of Logical and Algebraic Methods in Programming*, 2015.

[32] P. Brunet and D. Pous. Petri automata for Kleene allegories. In *Proc. LICS*, pages 68–79. ACM, 2015.

[33] P. Brunet, D. Pous, and I. Stucke. Cardinalities of relations in Coq. Submitted, 2016.

[34] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[35] R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[36] J. A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, 1964.

[37] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proc. STOC*, pages 77–90. ACM, 1977.

[38] K. Chaudhuri, M. Cimini, and D. Miller. A lightweight formalization of the metatheory of bisimulation-up-to. In *Proc. CPP*, pages 157–166. ACM, 2015.

[39] E. Cohen. Hypotheses in Kleene algebra. Technical report, Bellcore, Morristown, N.J., 1994.

[40] E. Cohen, D. Kozen, and F. Smith. The complexity of Kleene algebra with tests. Technical Report TR96-1598, CS Dpt., Cornell University, 1996.

[41] J. H. Conway. *Regular algebra and finite machines*. Chapman and Hall, 1971.

[42] T. Coquand and V. Siles. A decision procedure for regular expression equivalence in type theory. In *Proc. CPP*, volume 7086 of *LNCS*, pages 119–134. Springer, 2011.

[43] L. D'Antoni and M. Veanes. Minimization of symbolic automata. In *Proc. POPL*, pages 541–553. ACM, 2014.

[44] L. Doyen and J.-F. Raskin. Antichain Algorithms for Finite Automata. In *Proc. TACAS*, volume 6015 of *LNCS*. Springer, 2010.

[45] Z. Ésik and L. Bernátsky. Equational properties of Kleene algebras of relations with conversion. *Theoretical Computer Science*, 137(2):237–251, 1995.

[46] J.-C. Fernandez, L. Mounier, C. Jard, and T. Jéron. On-the-fly verification of finite transition systems. *Formal Methods in System Design*, 1(2/3):251–273, 1992.

[47] E. Filiot, N. Jin, and J.-F. Raskin. An antichain algorithm for LTL realizability. In *Proc. CAV*, volume 5643 of *LNCS*, pages 263–277. Springer, 2009.

[48] J.-C. Filliâtre and S. Conchon. Type-safe modular hash-consing. In *Proc. ML*, pages 12–19. ACM, 2006.

[49] N. Foster, D. Kozen, M. Milano, A. Silva, and L. Thompson. A coalgebraic decision procedure for NetKAT. In *Proc. POPL*. ACM, 2015.

[50] S. Foster and G. Struth. Automated analysis of regular algebra. In *Proc. IJCAR*, volume 7364 of *LNCS*, pages 271–285. Springer, 2012.

[51] C. Fournet, J. Lévy, and A. Schmitt. An asynchronous, distributed implementation of mobile ambients. In *Proc. IFIP TCS*, volume 1872 of *LNCS*, pages 348–364. Springer, 2000.

[52] P. Freyd and A. Scedrov. *Categories, Allegories*. North Holland, 1990.

[53] N. Galatos, P. Jipsen, T. Kowalski, and H. Ono. *Residuated Lattices: An Algebraic Glimpse at Substructural Logics*. Elsevier, 2007.

[54] A. Goel, S. Khanna, D. Larkin, and R. E. Tarjan. Disjoint set union with randomized linking. In *Proc. SODA*, pages 1005–1017. SIAM, 2014.

[55] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. L. Roux, A. Mahboubi, R. O'Connor, S. O. Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. A machine-checked proof of the odd order theorem. In *Proc. ITP*, volume 7998 of *LNCS*, pages 163–179. Springer, 2013.

[56] B. Grégoire and A. Mahboubi. Proving equalities in a commutative ring done right in Coq. In *Proc. TPHOL*, volume 3603 of *LNCS*, pages 98–113. Springer, 2005.

[57] M. Grohe. The complexity of homomorphism and constraint satisfaction problems seen from the other side. *J. ACM*, 54(1):1:1–1:24, Mar. 2007.

[58] C. Gutiérrez. Decidability of the equational theory of allegories. In *Proc. 4th RelMiCS*, pages 91–96, 1998.

[59] C. Hardin and D. Kozen. On the elimination of hypotheses in Kleene algebra with tests. Technical Report TR2002-1879, CS Dpt., Cornell University, October 2002.

[60] J. G. Henriksen, J. L. Jensen, M. E. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Proc. TACAS*, volume 1019 of *LNCS*, pages 89–110. Springer, 1995.

[61] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *Proc. FOCS*, pages 453–462. IEEE, 1995.

[62] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[63] I. Hodkinson and S. Mikulás. Axiomatizability of reducts of algebras of relations. *Algebra Universalis*, 43(2):127–156, 2000.

[64] J. E. Hopcroft. An n log n algorithm for minimizing in a finite automaton. In *Proc. International Symposium of Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

[65] J. E. Hopcroft. An $n\log n$ algorithm for minimizing states in a finite automaton. Technical report, Stanford University, 1971.

[66] J. E. Hopcroft and R. M. Karp. A linear algorithm for testing equivalence of finite automata. Technical Report 114, Cornell University, December 1971.

[67] G. Huet. *Résolution d'équations dans les langages d'ordre 1,2, ... ,ω*. PhD thesis, Université Paris VII, 1976. Thèse d'État.

[68] C. Hur, G. Neis, D. Dreyer, and V. Vafeiadis. The power of parameterization in coinductive proof. In *Proc. POPL*, pages 193–206. ACM, 2013.

[69] L. Ilie and S. Yu. Follow automata. *Information and Computation*, 186(1):140–162, 2003.

[70] K. Iwano and K. Steiglitz. A semiring on convex polygons and zero-sum cycle problems. *SIAM J. Comput.*, 19(5):883–901, 1990.

[71] A. Jeffrey and J. Rathke. Towards a theory of bisimulation for local names. In *Proc. LICS*, pages 56–66, 1999.

[72] A. Jeffrey and J. Rathke. A theory of bisimulation for a fragment of concurrent ML with local names. *Theoretical Computer Science*, 323(1-3):1–48, 2004.

[73] D. M. Kaplan. Regular expressions and the equivalence of programs. *J. Comput. Syst. Sci.*, 3(4):361–386, 1969.

[74] B. Knaster. Un théorème sur les fonctions d'ensembles. *Annales de la Société Polonaise de Mathématiques*, 6:133–134, 1928.

[75] V. Komendantsky. Reflexive toolbox for regular expression matching: verification of functional programs in Coq+ssreflect. In *Proc. PLPV*, pages 61–70. ACM, 2012.

[76] V. Koutavas and M. Wand. Small bisimulations for reasoning about higher-order imperative programs. In *Proc. POPL*, pages 141–152. ACM, 2006.

[77] D. Kozen. A completeness theorem for Kleene Algebras and the algebra of regular events. In *Proc. LICS*, pages 214–225. IEEE Computer Society, 1991.

[78] D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, 1994.

[79] D. Kozen. Kleene algebra with tests. *Transactions on Programming Languages and Systems*, 19(3):427–443, May 1997.

[80] D. Kozen. On Hoare logic and Kleene algebra with tests. *ACM Trans. Comput. Log.*, 1(1):60–76, 2000.

[81] D. Kozen. On the coalgebraic theory of Kleene algebra with tests. Technical report, CIS, Cornell University, March 2008.

[82] D. Kozen and M.-C. Patron. Certification of compiler optimizations using Kleene algebra with tests. In *Proc. CL2000*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 568–582. Springer, 2000.

[83] D. Kozen and F. Smith. Kleene algebra with tests: Completeness and decidability. In *Proc. CSL*, volume 1258 of *LNCS*, pages 244–259. Springer, September 1996.

[84] A. Krauss and T. Nipkow. Proof pearl: Regular expression equivalence and relation algebra. *Journal of Algebraic Reasoning*, 49(1):95–106, 2012.

[85] D. Krob. Complete systems of B-rational identities. *Theoretical Computer Science*, 89(2):207–343, 1991.

[86] S. B. Lassen. *Relational Reasoning about Functions and Nondeterminism*. PhD thesis, Department of CS, University of Aarhus, 1998.

[87] O. Lengál, J. Simácek, and T. Vojnar. Vata: A library for efficient manipulation of non-deterministic tree automata. In *Proc. TACAS*, volume 7214 of *LNCS*, pages 79–94. Springer, 2012.

[88] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[89] J. Madiot, D. Pous, and D. Sangiorgi. Bisimulations up-to: Beyond first-order transition systems. In *Proc. CONCUR*, volume 8704 of *LNCS*, pages 93–108. Springer, 2014.

[90] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974.

[91] A. Meyer and L. J. Stockmeyer. Word problems requiring exponential time. In *Proc. STOC*, pages 1–9. ACM, 1973.

[92] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[93] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I/II. *Information and Computation*, 100(1):1–77, 1992.

[94] D. Monk. On representable relation algebras. *Michigan Math. J.*, 11(3):207–210, 09 1964.

[95] E. F. Moore. Gedanken-experiments on sequential machines. *Automata Studies, Annals of Mathematical Studies*, 34:129–153, 1956.

[96] N. Moreira, D. Pereira, and S. M. de Sousa. Deciding regular expressions (in-)equivalence in Coq. In *Proc. RAMiCS*, volume 7560 of *LNCS*, pages 98–113. Springer, 2012.

[97] T. Murata. Petri nets: Properties, analysis and applications. *Proc. of the IEEE*, 77(4):541–580, Apr 1989.

[98] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, 1980.

[99] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[100] J. Parrow and T. Weber. The largest respectful function. To appear in Logical Methods in Computer Science.

[101] C. A. Petri. Fundamentals of a theory of asynchronous information flow. In *Proc. IFIP Congress*, pages 386–390, 1962.

[102] J. Å. Pohjola and J. Parrow. Bisimulation up-to techniques for psi-calculi. In *Proc. CPP*, pages 142–153. ACM, 2016.

[103] F. Pottier and D. Rémy. *Advanced Topics in Types and Programming Languages*, chapter The Essence of ML Type Inference. MIT Press, 2004.

[104] D. Pous. Complete lattices and up-to techniques. In *Proc. APLAS*, volume 4807 of *LNCS*, pages 351–366. Springer, 2007.

[105] D. Pous. *Techniques modulo pour les bisimulations*. PhD thesis, École Normale Supérieure de Lyon, February 2008.

[106] D. Pous. Untyping typed algebraic structures and colouring proof nets of cyclic linear logic. In *Proc. CSL*, volume 6247 of *LNCS*, pages 484–498. Springer, August 2010.

[107] D. Pous. Untyping typed algebras and colouring cyclic Linear Logic. *Logical Methods in Computer Science*, 8(2), 2012.

[108] D. Pous. Web appendix to [21, 22], with omitted proofs, Coq development, and interactive applet. `http://perso.ens-lyon.fr/damien.pous/hkc`, 2012.

[109] D. Pous. Web appendix to [111], with implementation and interactive applet. `http://perso.ens-lyon.fr/damien.pous/symkat`, 2012.

[110] D. Pous. Kleene Algebra with Tests and Coq tools for while programs. In *Proc. ITP*, volume 7998 of *LNCS*, pages 180–196. Springer, 2013.

[111] D. Pous. Symbolic algorithms for language equivalence and Kleene Algebra with Tests. In *Proc. POPL*, pages 357–368. ACM, 2015.

[112] D. Pous. Coinduction all the way up. In *Proc. LICS*, pages 307–316. ACM, 2016.

[113] D. Pous. *Informatique Mathématique, une photographie en 2016*, chapter Algèbres de relations. CNRS Editions, 2016.

[114] D. Pous. Web appendix to [112], with omitted proofs and Coq formalisation. `http://perso.ens-lyon.fr/damien.pous/cawu`, 2016.

[115] D. Pous. RelationAlgebra: Coq library covering relation algebra and KAT. `http://perso.ens-lyon.fr/damien.pous/ra`, developed since 2012.

[116] D. Pous and D. Sangiorgi. *Advanced Topics in Bisimulation and Coinduction*, chapter about "Enhancements of the coinductive proof method". Cambridge University Press, 2011.

[117] V. Pratt. Action logic and pure induction. In *Proc. JELIA*, volume 478 of *LNCS*, pages 97–120. Springer, 1990.

[118] V. Redko. On defining relations for the algebra of regular events. *Ukr. Mat. Z.*, 16:120–, 1964.

[119] D. Rémy. *Algèbres Touffues. Application au Typage Polymorphe des Objets Enregistrements dans les Langages Fonctionnels*. PhD thesis, Université Paris VII, 1990. Thèse de doctorat.

[120] D. Rémy. Extension of ML type system with a sorted equational theory on types, 1992. Research Report 1766.

[121] J. Rot and M. M. Bonsangue. Combining bialgebraic semantics and equations. In *Proc. FoSSaCS*, volume 8412 of *LNCS*, pages 381–395. Springer, 2014.

[122] H. Rubin and J. E. Rubin. *Equivalents of the Axiom of Choice*. North Holland, 1963.

[123] J. Rutten. Automata and coinduction (an exercise in coalgebra). In *Proc. CONCUR*, volume 1466 of *LNCS*, pages 194–218. Springer, 1998.

[124] J. J. M. M. Rutten. A coinductive calculus of streams. *Mathematical Structures in Computer Science*, 15(1):93–147, 2005.

[125] A. Salomaa. Two complete axiom systems for the algebra of regular events. *Journal of the ACM*, 13(1):158–169, 1966.

[126] D. Sangiorgi. On the bisimulation proof method. *Mathematical Structures in Computer Science*, 8:447–479, 1998.

[127] D. Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2011.

[128] D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[129] J. Sevcík, V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, and P. Sewell. Compcerttso: A verified compiler for relaxed-memory concurrency. *Journal of the ACM*, 60(3):22, 2013.

[130] R. E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, 1984.

[131] E. Sumii and B. C. Pierce. A bisimulation for dynamic sealing. *Theoretical Computer Science*, 375(1-3):169–192, 2007.

[132] E. Sumii and B. C. Pierce. A bisimulation for type abstraction and recursion. *Journal of the ACM*, 54(5), 2007.

[133] D. Tabakov and M. Vardi. Experimental evaluation of classical automata constructions. In *Proc. LPAR*, volume 3835 of *LNCS*, pages 396–411. Springer, 2005.

[134] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[135] A. Tarski. On the calculus of relations. *J. Symbolic logic*, 6:73–89, 1941.

[136] A. Tarski. A Lattice-Theoretical Fixpoint Theorem and its Applications. *Pacific Journal of Mathematics*, 5(2):285–309, June 1955.

[137] A. Tarski and S. Givant. *A Formalization of Set Theory without Variables*, volume 41 of *Colloquium Publications*. American Mathematical Society, Providence, Rhode Island, 1987.

[138] D. Turi and G. D. Plotkin. Towards a mathematical operational semantics. In *Proc. LICS*, pages 280–291. IEEE, 1997.

[139] M. Veanes. Applications of symbolic finite automata. In *Proc. CIAA*, volume 7982 of *LNCS*, pages 16–23. Springer, 2013.

[140] M. D. Wulf, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Antichains: A new algorithm for checking universality of finite automata. In *Proc. CAV*, volume 4144 of *LNCS*, pages 17–30. Springer, 2006.