Algèbres de Kleene, réécriture modulo AC et circuits en coq

Thomas Braibant

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial : 7 août 2006

Présentée par

Thomas BRAIBANT

Thèse dirigée par **Jean-Bernard STEFANI**

préparée au sein du **Laboratoire d'Informatique de Grenoble**
et de **L'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Kleene Algebras, Rewriting Modulo AC, and Circuits in Coq

Thèse soutenue publiquement le **17 février 2012**,
devant le jury composé de :

Gérard BERRY
Directeur de recherche à Inria, Rapporteur
Dan R. GHICA
Senior Lecturer, Birmingham University, Examinateur
Christine PAULIN-MOHRING
Professeur à l'Université Paris Sud, Rapporteur
Lawrence C. PAULSON
Professor, Cambridge University, Président
Damien POUS
Chargé de recherche au CNRS, Encadrant de thèse
Jean-Bernard STEFANI
Directeur de recherche à Inria, Directeur de thèse
Laurent THÉRY
Chargé de recherche à Inria, Examinateur

Remerciements

Merci à Christine Paulin-Mohring et Gérard Berry de m'avoir fait l'honneur de relire cette thèse. Leurs remarques m'ont permis d'améliorer sensiblement ce manuscrit, et leurs rapports m'ont apporté un regain de confiance en moi, à une période où j'en avais besoin. En particulier, ils m'ont fait comprendre qu'une thèse, ce n'était pas la fin du monde, et qu'il serait toujours temps de continuer après.

Merci à Larry Paulson, qui a bien voulu présider, à Dan Ghica et à Laurent Théry. J'ai grandement apprécié les discussions que nous avons eues, et j'espère les poursuivre lorsque nous serons moins contraints par le temps.

Merci à Damien d'avoir encadré cette thèse. Nos échanges m'ont permis de me poser les bonnes questions et de corriger, je l'espère, mes erreurs. Merci à Jean-Bernard pour son soutien moral, scientifique et administratif. Chacun à sa manière m'a fait mûrir sur le plan humain et scientifique. Ils m'ont aussi laissé une grande liberté quand j'ai voulu orienter mon travail de recherche hors du cadre qui avait été défini. Qu'ils en soient remerciés.

Merci à ces sept personnes qui ont formé mon jury de thèse: j'espère être à la hauteur de l'effort qui m'a été consacré.

Merci à l'équipe Sardes, dans son ensemble. Merci à Alan et Sergueï pour les séminaires après le déjeuner, qui ont donnés lieu à de nombreux échanges constructifs et amicaux entre Zélotes de la Théorie et de la Pratique. Merci à Jean, Quentin et aux autres "non-permanents" pour les réunions techniques du jeudi soir. Merci à Diane.

Merci à Pascal, Michaël et Sylvain, collègues grenoblois de l'autre coté de la rivière, qui m'ont encadré durant mon monitorat. Ils ont fait de moi un meilleur enseignant.

Merci à l'équipe POPART pour ses séminaires, et plus particulièrement aux discussions avec Alain, Gwenaël et Pascal, collègues grenoblois du bon coté de la rivière, mais dont je n'ai pas su assez profiter.

Merci aux collègues parisiens, en vrac, Xavier, Luc, Jade, Hugo, Pierre Bout', Matthieu et Stéphane. J'espère vous retrouver très bientôt pour mon plus grand plaisir. Merci à Daniel, une inspiration.

Merci à tous ceux, trop nombreux pour être nommés ici, avec qui j'ai passé de bons moments lors de conférences, et plus particulièrement à Munich, en Ecosse, en Oregon ou à Taiwan.

Cette thèse doit beaucoup au soutien que j'ai reçu de mes proches durant ces trois ans. Merci à Sylvain, Cécile et Aline, grenoblois ou presque. Merci à Bertrand, Martin, Jpla, Clément, Thibaud, Anne-Sandrine, Claire et les autres pour les parties de pêche sur les bords du lac de Bitche, les points ours en montagne et les visites d'abbayes en Belgique. Merci à tous les membres de ma famille qui m'ont soutenu et accompagnés dans ma vocation à devenir la troisième génération de docteurs, même si je ne pourrais toujours réparer ni ordinateurs, ni rotules. Merci à Pauline, même si c'est peu au regard de ce que cette thèse te doit.

Il me reste à remercier tous les gens que je n'ai pas remercié plus haut, mais que j'ai cotoyé ces dernières années, avant de mettre un point final à ce manuscrit.

Contents

List of Figures

Introduction

Formal verification in general has taken a huge leap in the last decade, and machine checked proofs of formal properties can be handled on sizable artefacts. A formal proof of the four-colour theorem (which eluded proof attempts by mathematicians for more than one century) has been given by Gonthier and Werner in [71]. CompCert [109], a lightly-optimising compiler for a large subset of the C programming language has been programmed and proved semantically preserving in the formal system Coq. The implementation in C of the seL4 micro-kernel [99] has been proved to implement correctly its specification in the formal system Isabelle/HOL. Formal verification has become so much accessible that a whole graduate level course on Software Foundations [125] has been formalised and machine-checked and that the material for the course is literally its *proof script*.

Context of the thesis

Formal verification has its roots in the work of Frege, during the late 19th century, who gave the first formal presentation of first order logic, effectively laying the ground for *formal proofs*. A formal proof is a proof whose validity can be verified by checking that each step of the derivation is well-formed with respect to a given formal system.

The first tool to automatically check the correctness of a proof, called "Automath" was developed by de Bruijn in the late 1960s, quickly followed by the Nqthm theorem prover by Boyer and Moore. There is a difference in spirit between these two systems: the former aimed solely at checking derivations (i.e., *proof terms*) that were written by hand, while the latter aimed at being a fully automatic theorem prover (though relying on user-guidance to find proofs). There is a profound dichotomy underlining these two approaches: while checking formal proofs is usually simple, the problem of finding proofs may be computationally intractable, when even possible.

Indeed, Church and Turing independently proved during the 1930s that the validity problem of first-order logic formulae (a particular case of Hilbert's Entscheidungsproblem) is undecidable, and thus finding a proof of a general theorem using pure automation is challenging. However, some particular expressive, yet decidable, fragments are amenable to efficient *automated theorem proving*. For instance, the *satisfiability problem* (SAT) of a Boolean formula is NP-complete in general, but many practical instances can be solved quickly using modern algorithms. The *satisfiability modulo theory problem* (SMT) is an extension of SAT in which predicates from a variety of underlying (decidable) first-order theories are allowed. Arguably, *abstract interpretation* and *model checking* belong to the realm of automated theorem proving, and focus on properties about *programs* and *finite state systems*.

On the contrary, the act of checking formal proofs is amenable to mechanical verification, but producing a formal proof requires to make explicit every step of the proof. To cope with the amount of details imposed by the formal setting, the introduction of automation is mandatory to help the user to build proofs. Modern *interactive theorem provers* feature *tactics* that ease

the burden of formal proofs while producing verifiable proof terms. A *proof script* is a succession of definitions, statement of theorems and their proofs. The proof script is usually built as the result of an interaction between the theorem prover and the user. During a proof, the theorem prover accounts for the remaining *goals* before the proof may be deemed complete, and the user provides a succession of tactics to achieve each one. Tactics may transform or solve goals, or may decompose goals into several other goals. Powerful tactics make it possible to focus on the key arguments that require insight, while leaving the remaining (simple) parts to be discharged by automation.

Interactive theorem proving in Coq. A recent survey [157] compared formalisations of the proof of the irrationality of $\sqrt{2}$ in 17 different provers. A more restrictive list appears in the introduction of [46], namely ACL2, Coq, Isabelle/HOL[1], PVS and Twelf. Among those, Coq [153] is our interactive prover of choice in this thesis. Coq is based originally on the Calculus of Constructions [54], extended with inductive types (the Calculus of Inductive Constructions [55]). Coq is intimately built on the Curry-de Bruijn-Howard isomorphism, that is the correspondence between the types of the λ-calculus, seen as statements, and the terms of the λ-calculus that are the proof objects. In this setting, proving a theorem (seen as a type) requires to provide a λ-term that inhabits this type. This isomorphism is traditionally presented through the similarity between the two following rules: function application in the λ-calculus on the left, and modus-ponens on the right.

$$\frac{f : A \to B \qquad x : A}{f\ x : B} \qquad\qquad \frac{A \implies B \qquad A}{B}$$

The rules are identical except for the parts on the left of the columns: indeed, $x : A$ must be interpreted as "x proves A" rather than as "x is of type A". Then, $f\ x$ proves B since f proves $A \implies B$, and x proves A. As an example, a proof of

$$(A \to B \to C) \to (A \to B) \to A \to C$$

is the function

$$\lambda(f : A \to B \to C)(g : A \to B)(x : A).(f\ x\ (g\ x)).$$

Coq terms are equipped with a notion of *reduction*. For instance, the addition of Peano numbers can be defined in Coq so that 2+2 reduces to 4. The reduction rules of Coq form a confluent and strongly normalising system: every term has a *normal form*. For instance, given a predicate P over natural numbers, the normal form of the proposition $P(2 + 2)$ is $P\ 4$. The following *conversion rule* allows one to benefits from the above observation: a term t with type A as also type B whenever A and B have the same normal forms.

$$\frac{t : A \qquad A \equiv B}{t : B}$$

Therefore, given a term t of type $P\ 4$ then t is also a proof for $P(3+1)$ or $P(12/3)$ or $P((\lambda x.4)12)$, by conversion. That is, arbitrary computations may occur as part of the proof, but do not appear in the proof term.

Coq features *dependent product types* which unify the notions of logical implication, universal quantification and polymorphism. That is, λ-abstractions are typed with the dependent product using the following rule: if for all x with type A, the term t has type B, the term $\lambda x : A.t$ has

[1] Representative of the HOL family

type $\forall x : A, B$, where B can mention the variable x. If x does not appear in B, the type $\forall x : A.B$ shall be written $A \to B$ and represents either a function from A to B or a logical implication between the propositions A and B. Using dependent types, it is possible for instance to define the type `vector` A n of vectors of size n and elements of type A, as well as a an higher-order map function that operates on vectors, of type:

$$\forall A, \forall B, (A \to B) \to \forall (n : \mathbb{N}), \texttt{vector}\ A\ n \to \texttt{vector}\ B\ n.$$

Note that in this case, the type of map expresses the fact that the length of the vector does not change. Another example of use of dependent types is the term t of type $\forall x : \mathbb{N}, x = x$ that states that equality is reflexive. By conversion, $t\ 4$ is a proof term for $2 + 2 = 4$, but also for $1664/(2^5 * 13) = 5 - 1$.

Constructive type theory. Using Coq, a proof of A is a term t of type A that provides a computable inhabitant of A: the *constructive type theory* underlying Coq forbids the use of non-constructive terms such as the axiom of the excluded middle

$$\overline{A \vee \neg A}$$

The axiom says that either A or $\neg A$ is provable but it does not specify which. (Other non-constructive equivalent axioms are Peirce's law and the elimination of the double-negation.) Therefore, in Coq, a proof of $A \vee B$ provides either a proof of A or a proof of B; and a proof of $\exists x, x + 1 = 2 * x$ packages a witness, 1, and a proof that $2 = 2$. In fact, this restriction to the constructive logic is a prerequisite to make all the functions that can be built in the system *computable*. Another such prerequisite is that each function must *terminate*, either provably or structurally.

The de Bruijn criterion. Coq is composed of two main parts: a small kernel which is just a type-checker for the proof terms; and a proof engine that allows the user to build proof terms using tactics. This separation makes it possible to check the proof terms independently from their elaboration, thus meeting the *de Bruijn criterion*: the amount of code that must be trusted is kept small, and it is possible to provide independent proof checkers. (Note that Coq features such a minimalist stand-alone proof checker.) Moreover, the recent addition of a *plugin* mechanism to Coq allows to work with user-defined extensions that enrich it with new features (for instance, new tactics), without compromising the whole system: each new tactic must in the end build a proof-term whose correctness is checked by the kernel.

Programming in Coq. Coq's programming language (the definition of terms) may be seen as an extension of ML with explicit polymorphism, inductives and dependent types, restricted to terminating functions. The absence of distinction between programs and proof terms makes it possible to use programs in proofs. Consider for instance the following inductive predicate:

```
Inductive le : nat → nat → Prop :=
| le_n : ∀ n, le 0 n
| le_S : ∀ n m, le n m → le (S n) (S m).
```

A proof of `le` 3 5 is the term `le_S` 2 4 (`le_S` 1 3 (`le_S` 0 2 (`le_n` 2))). Remark that the size of this proof term grows with the (unary) size of the first argument.

Alternatively to this inductive presentation of this predicate, one can define by recursion a function `leb`: `nat` → `nat` → `bool` that checks that its first argument is smaller than its second argument. (Note that the result of this function is a Boolean, not a proposition.)

```
Fixpoint leb (n m : nat) :=
  match n,m with
  | 0,_ ⇒ true
  | S n, S m ⇒ leb n m
  | _, 0 ⇒ false
  end.
```

It is then possible to prove a lemma that links the computational function `leb`, with the inductive predicate `le`:

```
Lemma leb_le: ∀ a b, leb a b = true → le a b.
```

Hence, a possible proof of the proposition `le` 16 64 is the term `leb_le` 16 64 (`eq_refl true`) where `eq_refl` has type ∀ `x`, `x` = `x`. Here, we use computations as part of a proof: the normal form of `leb` 16 64 is indeed `true`. The type-checker can indeed check that the type of `eq_refl true` is convertible with `leb` 16 64 = `true`, which yields a valid proof of the statement `le` 16 64. This approach is called *computational reflection*, and in this particular case, `leb` is a *reflexive decision procedure* for the predicate `le`. While this example is trivial, this approach is central in some of our contributions. We shall give more details about this feature of Coq, at the end of this introduction.

Contributions

A decision procedure for Kleene algebras

We started this introduction with the enumeration of some outstanding achievements in formal proofs that were done during the last decade. Then, we pointed out the fact that building formal proofs in proof assistants is eased by powerful tactics.

Binary relations are pervasive in formal proofs about programming languages meta-theory [125], compiler verification [109], or modelisation of memory models [3]. From a mathematical point of view, proofs dealing with binary relations are arguably best presented when the relations are considered as abstract objects that are equipped with operations like composition, union, reflexive and transitive closure, converse, etc. Consider for instance the following example where R and S are arbitrary homogeneous binary relations, $\circ$ is relation composition, $\cup$ is union, $\_^\star$ is the reflexive and transitive closure and $\subseteq$ is relation inclusion.

$$S \circ (S \circ S^\star \circ R^\star \cup R^\star) \subseteq S \circ S^\star \circ R^\star.$$

Recall that by definition

$$R \subseteq S \triangleq \forall x \forall y, xRy \implies xSy$$
$$x(R \circ S)z \triangleq \exists y, xRy \wedge ySz$$

and ponder the fact that the above expression is more synthetic than the usual point-wise formulation, which requires to handle explicitly the existential quantification induced by the composition. In this point-free presentation of binary relations, called *relation algebra*, the reasoning is done via the axiomatisation of operations on binary relations. Moving to this algebraic setting makes it possible to implement various decision procedures for decidable fragments of relation algebra. In particular, binary relations with union, composition, and reflexive and transitive closure form a model of *Kleene algebra*, whose equational theory is decidable.

Our first contribution is a formalisation of Kleene algebras in Coq, equipped with an efficient decision procedure for the equational theory. This decision procedure required the formalisation in Coq of some usual finite automata constructions: construction of finite automata from regular

expressions, removal of ϵ-transitions, determinisation, and equivalence checking. Moreover, Kozen's theorem [100] must be formalised to apply this decision procedure to any model of Kleene algebras. This theorem states that the model of regular languages is *initial* among models of Kleene algebra: that is, any equation valid in the model of regular languages is valid in any other model of Kleene algebra, and a fortiori, in the model of binary relations.

This contribution do not aim at being minimalistic, and we make ours the following quote from Georges Gonthier:

> *"Perhaps this is the most promising aspect of formal proof: it is not merely a method to make absolutely sure we have not made a mistake in a proof, but also a tool that shows us and compels us to understand why a proof works."*

Indeed, verifying a simple decision procedure for regular expression equivalence and showing how to reduce equations between binary relations to equations between regular languages can be done without relying on Kozen's theorem and do not require a big formalisation. Neither does using out-of-the-shelf automated theorem provers with the rules of Kleene algebras. However, formalising Kozen's theorem compelled us to formalise automata in Coq but also to formalise a small algebraic hierarchy and to work with matrices which are essential to Kozen's proof. Moreover, taking care of the efficiency of the various steps of the decision procedure prompted us to change some early design-choices of algorithms and data-structures. Finally, this decision procedure is packaged as a general purpose tactic that may be used in arbitrary Coq developments.

Tools for rewriting modulo associativity and commutativity

Our second contribution aims at solving a practical and pervasive problem: a lot of small and trivial proof steps have to be done by the user to handle simple associativity and commutativity proofs. These steps are tedious to write and pollute the proof script. For instance, in relation algebra, the composition of relation is associative, and the union is both associative and commutative, and one often needs to use these properties in proofs just to be able to perform a step that requires insight. We investigated how to define a new high-level *rewrite* tactic that helps the user to focus on the proof steps that require sapience, by leveraging automatically the associativity and commutativity (AC) properties of some operators.

At the heart of this new rewriting tactic, there is a decision procedure for equality modulo associativity and commutativity. No need to say, this decision procedure is built using computational reflection. One of our main contributions is that we also use a novel reification technique that makes it possible to rewrite with an arbitrary numbers of associative commutative (or associative only) user-defined operations, with or without neutral elements, in the context of an user-defined equivalence relation (a *setoid relation*). This novel reification technique makes the tactic easy to apply for the user, without additional overhead.

A formalisation of hardware circuits in Coq

Verification of hardware components is a well-trodden ground. Some of the earliest practical works involving automated or interactive theorem proving dealt with proofs of correctness for hardware components. As a consequence, model checking as seen significant industry adoption, and interactive theorem proving methods are used by major hardware companies. Arguably, interactive verification of circuits is usually done on high-level models of circuits that are specified as predicates of the logic. For instance, an adder circuit may be modelled as the following predicate, where `nat_to_bv` converts a Peano integer to a bit-vector (modelled as a list of Booleans) while `bv_to_nat` is the dual function.

```
Definition ADD (a b out: list bool) : Prop := out = nat_to_bv (bv_to_nat a + bv_to_nat b).
```

This is a rather high-level representation of an adder circuit, suitable to be used as a base block in the formalisation of bigger circuits. More detailed adder implementations, defined as the composition of smaller blocks or basic "gates" (that is, predicates), can be proved correct w.r.t. the high-level one above. This kind of embedding of circuits is called a shallow-embedding: circuits are defined as predicates of the logic of the underlying theorem prover.

Our third contribution is a Coq library for the specification and verification of hardware components that defines a deep-embedding of circuits in Coq: circuits are modelled as a Coq data-type, equipped with a meaning function. This library builds upon the rich dependent types of Coq to define a data-structure that represents circuits that are "inherently well-formed". We shall demonstrate that using a small number of constructions makes it possible to build gate-level descriptions of circuits, which are yet amenable to high-level specifications. For instance, we shall prove that a gate-level description of an adder implements a high-level function, the addition of machine words. Using dependent-types makes it easier to define and reason about *circuit generators*: Coq functions that build circuits of, e.g., parametric size.

This work stems from the will to formalise other categories of finite state systems than the finite state automaton that underlies the decision procedure for regular language equivalence. Therefore, in our setting, the modelled circuits may be purely combinatorial (without state-holding elements), as well as synchronous (with state holding elements).

Coq features at work

These three contributions rely heavily on a small number of Coq properties. First, we define decision procedures using computational reflection: this requires that some computation may happen during proof-checking. Moreover, even the statements of the correctness of our decision procedures could hardly be stated without dependent types. (Note that with respect to meta-level tactics, decision procedures built using computational reflection have the advantage that their behaviour is specified in the type system of the prover itself.) Finally, our first two contributions rely on the presence of *type-classes* in Coq. Type-classes are useful to define abstract and modular algebraic structures in a way that provides support for notation overloading but also for sharing theorems between various instances of a given structure. But type-classes are also crucial for the automatic inference of properties of function symbols that underlies our rewriting modulo AC tactic.

This set of features (computation inside propositions, dependent types, and type-classes based modularity) is not present in any other current proof assistant. We reckon that our formalisations are quite Coq specific, in the sense that they rely on these particular features. We shall review through this manuscript several related contributions formalised in other theorem provers, and give some hints about the practical usefulness of these Coq features. This justifies *après coup* our choice of Coq as working framework.

Some Coq features

This section is devoted to a presentation of some advanced Coq features that will be used thoroughly in this dissertation. We assume that the reader has some basic familiarity with the Coq system. Some introductory textbooks are: Software Foundations by Pierce et al [125]; the tutorial Coq in a Hurry by Bertot [21]; the Coq'Art by Bertot and Casteran [22]; or Certified Programming With Dependent Types by Chlipala [46].

Figure 1 A map function on length-indexed lists (vectors)

```
Inductive vect (A: Type) : nat → Type :=
  vnil : vect A 0
| vcons : ∀ {n} (x:A), vect A n → vect A (S n).

Notation "x :: y" := (vcons _ x y).
Notation "[]" := (vnil _).
```

```
Context {A B : Type} (f : A → B).
Fixpoint map {n} (v: vect A n) : vect B n :=
match v with
  | [] ⇒ []
  | x :: v ⇒ (f x) :: (map v)
end.
```

Dependent types.

From the (functional) programmer point of view, a type gives an approximation of the behaviour of a function. For instance in OCaml, the type of the `map` function on lists,

$$(\alpha \to \beta) \to \alpha \texttt{ list} \to \beta \texttt{ list}$$

puts strong restrictions on what the underlying `map` function may do. Dependent types allows for more expressive specifications through types, but also, arguably, for sheerer expressiveness in what functions may do.

A good introduction to dependent types is the data type of length-indexed lists, called `vect` (see Fig. 1). This is a refinement of the usual `list` type that makes manifest the length of the underlying list. Note that this refinement does not incur much overhead, thanks to notations: for instance, we define a `map` function on vectors in Fig. 1 as we would have done on lists. Yet, the type of `map` proves for free the fact that the length of the resulting vector is equal to the length of the argument.

The vernacular command `Context` binds variables in a Coq section: the `map` function in Fig. 1 takes 5 arguments. However, the arguments `A`, `B` and `n` may be inferred by Coq from the type of the function `f` and the vector `v`: therefore, we declare these three arguments as *implicit arguments* using curly-braces instead of parentheses in the binders. In the following, we shall freely use the `Variable` keyword as a synonym of `Context` when we do not want to declare implicit arguments.

A vector is the data-structure of choice to package the arguments of an n-ary homogeneous function: a function that takes n arguments of a given type A, and returns a result of type A. The type `type_of A n` of such an n-ary function may be defined as follows and, given an inhabitant of `type_of A n` and a vector of arguments of length `n`, it is possible to apply the former to the latter using the `apply` function. (Note that we choose to reverse the intuitive order of the arguments of `apply` in the actual Coq definition in order to keep it simple, without adding much dependent-type boilerplate.)

```
Fixpoint type_of A n : Type :=
match n with
  | 0 ⇒ A
  | S n ⇒ A → type_of A n
end.
```

```
Fixpoint apply {A} {n} (v : vect A n) : type_of A n → A:=
match v with
  | [] ⇒ fun f ⇒ f
  | x:: v ⇒ fun f ⇒ apply v (f x)
end.
```

Remark that the type of `apply` forbids the application of, e.g, an unary function to a vector of length 3. In the following examples, we use the `Eval` command to compute the result of the application of an n-ary homogeneous function to a vector of length n.

```
Eval compute in apply (1 :: 2 :: 3 :: []) (fun x y z ⇒ x+y+z). (* 6 : nat *)
Eval compute in apply (1 :: 6 :: []) (fun x y ⇒ x+y).          (* 7 : nat *)
```

Basic introduction to type-classes in Coq.

The overall behaviour of Coq type-classes [147] is quite intuitive. The following excerpt shows a simple Haskell program on the left-hand side, that exploits a type-class `Hash` to get a number out of certain kind of values; and its translation into its Coq equivalent on the right-hand side.

```
class Hash a where
 hash :: a → Int
instance Hash Int where
 hash = id
instance (Hash a) ⇒ (Hash [a]) where
 hash = sum . map hash
main = print
 (hash 4, hash [4,5,6], hash [[4,5],[]])
```

```
Class Hash (A : Type) :=
 { hash: A → nat }.
Instance hash_n: Hash nat :=
 { hash x := x }.
Instance hash_l A: Hash A → Hash (list A) :=
 { hash l := fold_left (fun a x ⇒ a + hash x) l 0 }.
Eval simpl in
 (hash 4, hash [4;5;6], hash [[4;5];[]])    .
```

Coq type-classes are firstorder; everything is done with plain Coq terms. In particular, the `Class` keyword produces a dependent record and the `Instance` keyword acts like a standard definition. With the above code we get values of the following types:

```
Hash: Type → Type
hash: ∀ A, Hash A → A → nat
```

```
hash_n: Hash nat
hash_l: ∀ A, Hash A → Hash (list A)
```

The function `hash` is a *class projection*: it gives access to a field of the class. The subtlety is that the first two arguments of this function are implicit: they are automatically inserted by unification and type-class resolution. More precisely, when we write "`hash` [4;5;6] ", Coq actually reads "`@hash _ _` [4;5;6] ". By unification, the first placeholder has to be `list nat`, and Coq needs to guess a term of type `Hash (list nat)` to fill the second placeholder. This term is obtained by a simple proof search, using the two available instances for the class `Hash`, which yields "`@hash_l nat hash_n`". This proof search step is called *type-class resolution*. Accordingly, we get the following explicit terms for the three calls to `hash` in the above example.

input term	explicit, instantiated, term
`hash` 4	`@hash nat hash_n` 4
`hash` [4;5;6]	`@hash (list nat) (@hash_l nat hash_n)` [4;5;6]
`hash` [[4;5];[]]	`@hash (list (list nat)) (@hash_l (list nat) (hash_l nat hash_n))` [[4;5];[]]

Defining an algebraic structure. Then, consider a possible Coq definition of a group structure and some related lemmas in Fig. 2. Again, type-classes provide overloading: the notations provided for the class-projections (`op`, `inv` and `unit`) leave the group structure implicit. This makes it possible to write much shorter and readable terms, by letting Coq infer the obvious boilerplate: for instance, the notation `x*y` expands to `@op G x y` in the three lemmas from the right-hand side. What is more important is that these lemmas are parameterised by an arbitrary group structure: upon use, the type-classes resolution mechanism shall instantiate the group parameter in a suitable manner. (Note that we do not advocate that this definition of a group structure is the best way to proceed in Coq. We shall come back on this point in the next chapter.)

Some useful type-classes. Some examples of type-classes from Coq's standard library that are of common use in this manuscript are given in Fig. 3: they are useful to express properties about binary relations. The syntax :> in the definition of `Equivalence` indicates that an instance of the `Equivalence` class may be seen as an instance of, e.g., `Reflexive`. Finally, we will often deal with *proper morphisms* (See Fig. 4): functions that are compatible with an

Figure 2 Group theory using type-classes

```
Class group := {
  X : Type;
  op : X → X → X;
  inv : X → X;
  unit : X;
  op_ass: ∀ x y z, op (op x y) z = op x (op y z);
  op_inv: ∀ x, op x (inv x) = unit;
  unit_right : ∀ x, op x unit = x
}.

Notation "x * y" := (op x y).
Notation "1" := unit.
Notation "~ x" := (inv x).
```

```
Section t.
  Context {G :group}.

  Lemma unit_unique: ∀ x, x*x = x → x = 1.
  Proof. ... Qed.

  Lemma inv_unique: ∀ x y, x*y = 1 → y = ~x.
  Proof. ... Qed.

  Lemma inv_distr: ∀ x y, ~(x*y) = (~y)*(~x).
  Proof. ... Qed.
End t.
```

Figure 3 Definitions from the standard library

```
Context {A} (R: relation A).
Class Reflexive := reflexivity : ∀ x, R x x.
Class Symmetric := symmetry : ∀ x y, R x y → R y x.
Class Transitive := transitivity :
  ∀ x y z, R x y → R y z → R x z.
```

```
Class Equivalence := {
  Equivalence_Reflexive :> Reflexive R ;
  Equivalence_Symmetric :> Symmetric R ;
  Equivalence_Transitive :> Transitive R }.
```

user-defined relation and map equals to equals[2]. As an example, `Qeq` is the usual equality on rational numbers: the following instance states that addition on rational numbers maps equal numbers w.r.t. `Qeq` to equal numbers w.r.t. `Qeq`.

```
Instance q_add_compat : Proper (Qeq ⇒ Qeq ⇒ Qeq) (Qplus). Proof. ... Qed.
```

Coq's rewrite tactic is able to leverage this kind of information to perform generalised rewriting steps that involve user defined equalities: in a nutshell, type-class resolution may be used to compose instances of the `Proper` type-class to build a proof that replacing a term with a related one is valid in a given context.

Some pointers. This concludes our very short introduction to type-classes in Coq; we invite the reader to consult [147, 153] for more details.

[2] In the following, a set equipped with an user-defined equivalence relation is called a *setoid*

Figure 4 Proper morphisms

```
Class Proper {A} (R : relation A) (m : A) : Prop := proper_prf : R m m.

(* The relation R in Proper is usually instantiated with the following definition. *)
Definition respectful (A B : Type) (R : relation A) (R' : relation B) : relation (A → B) :=
  (fun (f g : A → B) ⇒ ∀ x y : A, R x y → R' (f x) (g y)).

Notation " R ⇒ R' " := (@respectful _ _ R R')
```

Proof by computational reflection

Computational reflection (or reflection in short) is a general purpose technique that aims at replacing proof steps with computations. Poincaré once pointed out that one does not "prove" "2+2=4", but that one "checks" it [128] (by computing the normal form of $2+2$). This simple example demonstrates that computation may help to simplify proofs: the reduction of $2+2$ in 4 is trivial, but the same methodology may be applied to the embedding of complex decision procedures as functions in the logic of the proof assistant.

A typical example of use of reflection occurs when proving that some numbers are composite. Suppose defined a predicate `Composite : positive → Prop` over positive numbers (represented as binary numbers, for efficiency reasons). A typical proof that a number is composite would require human acumen (or the use of external tools) to find a suitable decomposition. However, since compositeness is a decidable property, one can define in Coq a decision procedure `is_composite: positive → bool`, e.g., by implementing a factorisation algorithm, and then prove that this decision procedure is *correct*:

```
Lemma dp_correct (n : positive) : is_composite n = true → Composite n.
```

Therefore, a proof that 576077 is composite is the following:

```
Goal Composite 576077. Proof. exact (dp_correct 576077 (eq_refl true)). Qed.
```

Then, it is possible to gain extra confidence in this decision procedure by proving its *completeness* which amounts to strengthening the above implication in an equivalence. However, proving the completeness of a decision procedure is not mandatory: it may be easier in practice to define semi-decision procedure; or proving completeness may not be worth the burden. (Note that usually, completeness of a decision procedure requires its termination [106] which must be ensured nevertheless for any Coq function.)

Finally, remark that the size of the above proof is *constant* except for the fact that it must mention the number whose compositeness we assess. This is of little importance in this particular case but the same applies to more elaborate reflexive decision procedures. For instance, this means that a proof of a Boolean tautology using a reflexive SAT-solver would not mention the actual derivations built by the solver: the fact that such derivations exist would be checked when running the decision procedure. However, a reflexive decision procedure will be executed by Coq's reduction machinery, which is comparatively slow with respect to the execution of, e.g., OCaml programs. Therefore, a reflexive tactic that may be used intensively gains from being implemented using efficient algorithms and data structures.

Digression: validation of traces. Efficient external tools, for instance SMT-solvers or computer algebra systems implemented in arbitrary languages, may be used as tactics provided that there is some way to interpret their results as proof-terms. While the interfacing of external tools with Coq to produce such proof-terms may be challenging, this is how some Coq tactics are implemented: for instance, `firstorder` (a semi-decision procedure for first order logic) and `omega` (a decision procedure for quantifier-free Presburger arithmetic goals).

A popular middle ground between proof-term producing oracles and reflexive decision procedures is called *trace validation*: that is, to use an external tool focused on proof-search (implemented with any suitable optimisation) that produces a *witness* (a more or less detailed solution to the given problem); and to check this witness using a verified decision procedure. Depending on the level of details of the witness, and the particular problem at hand, more or less computations are required from the checker. For instance, for the above example about checking that a number n is composite, a suitable witness could be:

- a factor f of the number (the checker must verify that f divides n);

Figure 5 Reification primer

```
Inductive term: Type :=
| Op : term → term → term
| Unit: term
| Inv: term → term
| Const : X → term.
```

```
Fixpoint eval t : X :=
match t with
  | Op s t ⇒ (eval s) ∗ (eval t)
  | Inv s ⇒ inv (eval s)
  | Unit ⇒ 1
  | Const x ⇒ x end.
```

- or a factor f and the quotient q (the checker must verify that $f * q = n$);
- or the list of the prime factors of n with their multiplicities (the checker must verify that their product is n).

Note that this approach works best when finding a witness is computationally difficult while checking it remains "easy" (and that the size of the witness remains tractable).

Reification. In Coq, computation is possible on open terms: it is possible to compute the result of `plus 0 n` even if `n` is free. Yet, it is not possible to compute the result of `plus n 0`, since `plus` is defined by recursion on its first argument. Therefore, reflexive decision procedure need to be applied to closed terms to avoid getting stuck in computations. This requires to "reflect" goals into a concrete syntax in which free variables are reified into constants and terms are reified as abstract syntax trees.

We shall give the intuition about general purpose reification mechanisms through an example. We settle in the context of an abstract group with carrier `X`, an inverse `inv : X → X`, an associative operation `∗: X → X → X` and its right unit `1: X`. We define a data type for reified expressions in Fig. 5, together with an interpretation function (or "evaluation function"). From a general point of view, a data-type for reified expressions is meant to describe a syntactic representation of the algebraic structure at hand: here, we have one constructor for each element of the algebraic structure (`Op`, `Unit` and `Inv`). The `Const` constructor is a catch-all case for subexpressions that cannot be modelled in this data type. Such subexpressions may be free variables but also arbitrary terms of type `X`. For instance, consider the following evaluation that maps a reified term to an "user-level" term.

```
Variable a b : X. Variable f : X → X.
Eval compute in eval (Inv (Op (Op Unit (Const a)) (Inv (Const (f b))))).
(* inv (1 * a * inv (f b)) *)
```

From a general point of view, the reification process abstracts from a given model and put forward the underlying algebraic structure as an abstract syntax tree. This makes it possible to define algorithms that operate at the level of the algebraic structure, and prove them using algebraic laws.

As an example, we define and prove correct a function that simplifies the occurrences of the neutral element. In Fig. 6, we define "smart constructors" `Op`' and `Inv`' that handle the actual simplifications, and the fixpoint `simpl`. It is then possible to prove for instance that `simpl` preserves the evaluation (relying implicitly on axioms similar to the ones in Fig. 2).

```
Lemma simpl_correct: ∀ t, eval (simpl t) = eval t.
```

However, the definition of a (correct) decision procedure for equations in such groups requires the ability to test (in the logic) the equality of the terms that occur under the `Const` constructor, which is usually not possible (consider for instance the case where `X` is `Prop`, on which it is not possible to perform case analysis). The usual practice is to add a level of indirection: that is,

Figure 6 Simplification of reified terms

```
Definition Op' s t :=
match s,t with
  | Unit, _ ⇒ t
  | _, Unit ⇒ s
  | s,t ⇒ Op s t
end.
```

```
Definition Inv' t :=
match t with
  | Unit ⇒ Unit
  | t ⇒ t
end.
```

```
Fixpoint simpl t :=
match t with
  | Op s t ⇒ Op' (simpl s) (simpl t)
  | Inv s ⇒ Inv' (simpl s)
  | t ⇒ t
end.
```

Figure 7 Reification with environments

```
Inductive term: Type :=
| Op : term → term → term
| Unit: term
| Inv : term → term
| Var : nat → term.

Variable env : nat → X.
```

```
Fixpoint eval t : X :=
match t with
  | Op s t ⇒ (eval s) ∗ (eval t)
  | Inv s ⇒ inv (eval s)
  | Unit ⇒ 1
  | Var x ⇒ env x
end.
```

to define an *environment* to store the constants and index them with, e.g., natural number. This requires to replace the `Const` constructor with a `Var` constructor (that contains the index of the underlying constant in the environment) and to parametrise the `eval` function with an environment (see Fig. 7). As an example, consider the following evaluation that maps a reified term to an "user-level" term, using the provided environment.

```
Let env := fun x ⇒ match x with | 0 ⇒ a | 1 ⇒ f a | _ ⇒ b end.
Eval compute in eval env (Op (Var 0) (Inv (Op (Var 1) (Var 2)))). (* a * inv (f a * b) *)
```

This allows one to build simplification functions that rely on the identity of constants: e.g., this makes it possible to implement a function that simplifies `Op (Inv (Var x)) (Var x)` into `Unit`. Then, let `dp` be a reflexive decision procedure such that the following `dp_correct` lemma holds.

```
Lemma dp_correct: ∀ (s t: term) env, dp s t = true → eval env s = eval env t.
```

In order to apply the `dp_correct` reflexive decision procedure to a goal `a = b`, the user must provide the terms `s`, `t`, `env`, such that `eval env s` is convertible with `a` and `eval env t` is convertible with `b`. While it is possible to do it "by hand", this step is usually done by a *reification tactic*, implemented at the meta-level. We describe such a reification tactic in chapter 2.

Notes about this document

We discuss our efficient decision procedure for Kleene algebras in chapter 1. We describe our tactics for working modulo associativity and commutativity in chapter 2. Finally, we consider our formalisation of circuits in chapter 3.

Chapter 1

Tools for Kleene algebras

Introduction

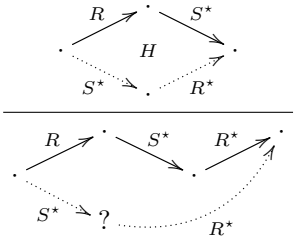
A starting point for this work is the following remark: proofs about abstract rewriting (e.g., Newman's Lemma, equivalence between weak confluence and the Church-Rosser property, termination theorems based on commutation properties) are best presented using informal "diagram chasing arguments". This is illustrated by Fig. 1.1, where the same state of a typical proof of the theorem "weak-confluence implies the Church-Rosser property" is represented three times. Informal diagrams are drawn on the left. The upper-right part corresponds to a naive formalisation where the points related by relations are mentioned explicitly. This is not satisfactory: a lot of points have to be defined, and the goal is stated in a rather verbose way. On the contrary, if we move to an algebraic setting (the lower right-hand side part), where binary relations are seen as abstract objects that can be composed using various operators (e.g., union, intersection, relational composition, iteration), statements become rather compact, making the goal easier to read and reason about.

More importantly, moving to the abstract "point-free" setting makes it possible to implement decision procedures. For instance, once we rewrite the hypothesis in the left-hand side of the last goal of Fig. 1.1, we obtain the inclusion $\mathtt{S}^\star \circ \mathtt{R}^\star \circ \mathtt{R}^\star \subseteq \mathtt{S}^\star \circ \mathtt{R}^\star$. This is a straightforward theorem of Kleene algebras: the tactic we describe in this chapter proves it automatically.

1.1 Definitions

In this section, we review some basic definitions of finite automata theory that will be used in the later sections.

Figure 1.1 Diagrammatic, concrete, and abstract presentations of the same state in a proof.



$$\frac{\forall p, r, q, pRr \to rS^\star q \to \exists s, pS^\star s \wedge sR^\star q \qquad pRq \qquad qS^\star s \qquad sR^\star q'}{\exists s, pS^\star s \wedge sR^\star q'}$$

$$\frac{R \circ S^\star \subseteq S^\star \circ R^\star}{R \circ S^\star \circ R^\star \subseteq S^\star \circ R^\star}$$

1.1.1 Regular expressions and automata

In the context of automata theory, we call an *alphabet* a non-empty set Σ, which is a set of base elements (or *letters* or *variables*). A *word* is a finite sequence of letters, with the special case of the empty word, written ϵ. The set of words on Σ is written $\Sigma^\star$. A *language* on the alphabet Σ is a set of words on Σ. We define several operations on languages:

$$K \cup L \triangleq \{u \mid u \in K \text{ or } u \in L\}$$
$$K \cdot L \triangleq \{u \cdot v \mid u \in K, v \in L\}$$
$$L^\star \triangleq \bigcup_{0 \leq n} L^n$$

where $L^0 \triangleq \{\epsilon\}$ and $L^{n+1} \triangleq L^n \cdot L$ for $0 \leq n$

Definition 1 (Regular expression)**.** *The family of regular expressions over an alphabet Σ, written $\mathbf{RE_\Sigma}$, is the smallest set such that:*

- *0,1 and $a \in \Sigma$ are regular expressions;*
- *if E and F are regular expressions, then $E + F$, $E \cdot F$ and $E^\star$ are regular expressions.*

The standard interpretation of regular expressions maps each regular expression E to the language $L(E)$:

- *$L(0) = \emptyset$, $L(1) = \{\epsilon\}$, $L(a) = \{a\}$ for $a \in \Sigma$*
- *$L(E + F) = L(E) \cup L(F)$, $L(E \cdot F) = L(E) \cdot L(F)$, $L(E^\star) = L(E)^\star$*

The family of regular sets over a finite alphabet, written $\mathbf{RL_\Sigma}$, is defined as the image of $\mathbf{RE_\Sigma}$ by L.

Different regular expressions may denote the same language. This leads to the notion of *equivalent* regular expressions: two regular expressions E and F are equivalent (written $E \sim F$) if they denote the same language, i.e., if $L(E) = L(F)$. The easiest way to prove such equivalences is to go through finite automata theory: indeed, we shall see that languages that can be described by finite automata are exactly the same as the languages denoted by regular expressions [98].

Definition 2 (Finite automata)**.** *Let Σ be an alphabet. We formally denote a non-deterministic finite automata (or NFA) $\mathcal{A}$ on the alphabet Σ by a 4-uple $\langle Q, T, I, F\rangle$ where Q is a finite set of states, T is a finite subset of $Q \times \Sigma \times Q$ called the set of transitions, I is a subset of Q called the set of initial states, and F is a subset of Q called the set of final (or accepting) states.*

The transition function $\delta_\mathcal{A}$ of the automata $\mathcal{A}$ is the mapping from $Q \times \Sigma$ to $\mathcal{P}(Q)$, where $\mathcal{P}(Q)$ denote the power set of Q, such that for each state p and each letter a,

$$\delta_\mathcal{A}(p, a) = \{q \mid (p, a, q) \in T\}$$

This transition function is extended to words by $\overline{\delta_\mathcal{A}} : Q \times \Sigma^\star \to \mathcal{P}(Q)$ such that:

$$\overline{\delta}_\mathcal{A}(p, \epsilon) = \{p\} \qquad \overline{\delta}_\mathcal{A}(p, a \cdot w) = \bigcup_{q \in \delta(p,a)} \overline{\delta}(q, w)$$

Finally, the language recognised by $\mathcal{A}$ is the set of words that may lead from an initial state to an accepting state. Formally,

$$L(\mathcal{A}) = \bigcup_{p \in I} \left\{w \in \Sigma^\star \mid \overline{\delta}_\mathcal{A}(p, w) \cap F \neq \emptyset\right\}$$

Note that when the automaton is obvious, we may drop the subscripts for δ and $\overline{\delta}$.

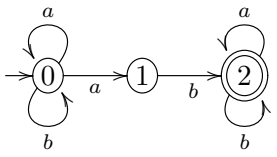
Automata are often depicted as labeled graphs called *transition diagrams*: there is a vertex for each state in Q, and an arc from q to q' labeled by a in the graph for each transition (q, a, q') in T. In this dissertation, accepting states are marked with a double circle, and unlabeled arrows point to the starting states. (In the following, transitions diagrams will be the preferred notation for describing finite automata.)

Alternatively, automata can be depicted by *transitions matrix* where the rows and the columns of the matrix correspond to states (since the labels of the states are irrelevant, we can assume that the states are labelled by integers, e.g., ranging from 0 to $|Q| - 1$). Generally speaking, we define a matrix over $\mathcal{R}$ of size $(n \times m)$ (that is, n lines and m columns) to be a family of elements of $\mathcal{R}$ indexed by the cartesian product of the integers sets $\{0, \ldots, n-1\}$ and $\{0, \ldots, m-1\}$. The family of rectangular $(n \times m)$ matrices over elements $\mathcal{R}$ will be written $\mathcal{M}_{(n,m)}(\mathcal{R})$. In the particular case of a transition matrix, the element at row i and column j is the subset of the alphabet that labels transitions going from i to j. To be more precise, given an automata $\langle Q, T, I, F\rangle$ on an alphabet Σ, its transition matrix M is defined by:

$$M_{ij} = \{a \in \Sigma \mid (i, a, j) \in T\}.$$

Example 1. *In this example, the alphabet is $\Sigma = \{a, b\}$. We present on the left an automata $\mathcal{A}$, and its transition diagram on the right. The language recognised by $\mathcal{A}$ is $L(\mathcal{A}) = L(\Sigma^\star \cdot a \cdot b \cdot \Sigma^\star)$, i.e., the set of words that contain the factor $a \cdot b$.*

$$\begin{array}{rcl} Q & = & \{0, 1, 2\} \\ T & = & \{(0, a, 0), (0, b, 0), (0, a, 1), \\ & & (1, b, 2), (2, a, 2), (2, b, 2)\} \\ I & = & \{0\} \\ F & = & \{2\} \end{array}$$



Its transition matrix is:

$$M = \begin{pmatrix} \{a, b\} & \{a\} & \emptyset \\ \emptyset & \emptyset & \{b\} \\ \emptyset & \emptyset & \{a, b\} \end{pmatrix}$$

We presented the definition of non-deterministic finite automata: the transition function δ returns a set of states. In the previous example, being in state 1 and reading the letter a allows the automata to go to state 1 or to state 2: an NFA has the power to be in several states at once while reading a word. We now turn to the definition of other kinds of automata.

Definition 3. *A non-deterministic finite automaton with ϵ-transitions (or ϵ-NFA) on an alphabet Σ is a 4-uple $A = \langle Q, T, I, F\rangle$ where Q is a finite set of states, I and F are subsets of Q, and T is a subset of $Q \times (\Sigma \cup \{\epsilon\}) \times Q$.*

The only difference between NFAs and ϵ-NFAs is that the latter allow the automata to take transitions labelled by the empty word. Despite this relaxation, ϵ-NFAs can recognise the same class of language as NFAs. Indeed, an NFA is an ϵ-NFA, and it is possible to convert an ϵ-NFA to an NFA through the *removal of ϵ-transitions*. This construction goes as follows: given an ϵ-NFA $\mathcal{A} = \langle Q, T, I, F\rangle$, we build the NFA $\mathcal{A}' = \langle Q, T', I', F\rangle$ such that:

$$\begin{aligned} R &= \{(x, y) \mid (x, \epsilon, y) \in T\} \\ T' &= \left\{(p, a, q') \mid (p, a, q) \in T \text{ and } (q, q') \in R^\star\right\} \\ I' &= \{j \mid i \in I \text{ and } (i, j) \in R^\star\}. \end{aligned}$$

The relation R is called the *ϵ-transitions relation*; $R^\star$ corresponds to the usual reflexive and transitive closure of a binary relation. The new set of initial states I' corresponds to former

initial states, as well as the states that can be accessed from an initial state using only ϵ-transitions, i.e., the reflexive and transitive closure of the ϵ-transitions. Similarly, the new set of transitions T' corresponds to all the former transitions, postfixed by the reflexive and transitive closure of the ϵ-transitions. While $\mathcal{A}$ and $\mathcal{A}'$ have the same number of states, $\mathcal{A}'$ may have far more transitions than $\mathcal{A}$. However, the language recognised by $\mathcal{A}'$ is equal to the language recognised by $\mathcal{A}$.

Definition 4. *A deterministic finite automaton on an alphabet Σ is an NFA $\mathcal{A} = \langle Q, T, I, F\rangle$ where $|I| = 1$, and for all $q \in Q$ and $a \in \Sigma$, $|\delta(q, a)| \leq 1$. Moreover, $\mathcal{A}$ is* complete *if $|\delta(q, a)| = 1$ for all states and input letters.*

In the following, we will only consider complete deterministic finite automata (written DFAs). We assume, without loss of generality, that the transition function of a DFA has type $Q \times \Sigma \to Q$.

The equivalence of DFAs and NFAs was first studied by Rabin and Scott [130]. The simplest construction of a DFA is called the *subset construction*: it builds a DFA with one state for each subset of the set of states of the original NFA. Often, only a few such sets of states can be accessed from the initial state of the DFA: inaccessible states can be thrown away, effectively reducing the number of states of the DFA. The construction we present here is actually called the *accessible subset construction*.

Given an NFA $\mathcal{N} = \langle Q, T, I, F\rangle$, we build a DFA $\mathcal{D} = \langle Q', T', \{I\}, F'\rangle$ such that

$$\begin{aligned}
\Delta(P, a) &= \{q \in Q \mid \exists p \in P, (p, a, q) \in T\} \\
Q' &= \bigcup_{n\in\mathbb{N}} Q_n \qquad \text{where} \qquad \begin{cases} Q_0 = \{I\} \\ Q_{n+1} = \bigcup_{a\in\Sigma} \{\Delta(P, a) \mid P \in Q_n\} \end{cases} \\
T' &= \left\{(P, a, \Delta(P, a)) \mid P \in Q', a \in \Sigma\right\} \\
F' &= \left\{P \mid P \in Q', P \cap F \neq \emptyset\right\}
\end{aligned}$$

Note that the cardinal of Q' is bounded by $2^{|Q|}$ (hence, this construction is well-defined), but is often much smaller in practice. The language recognised by $\mathcal{D}$ is equal to the language recognised by $\mathcal{N}$. Moreover, we can define an injection ρ from $[1, \ldots, |Q'|]$ to subsets of Q such that:

$$\forall s \in Q', \forall a \in \Sigma, \rho(\delta_{\mathcal{D}}(s, a)) = \bigcup_{p\in\rho(s)} (\delta_{\mathcal{N}}(p, a))$$

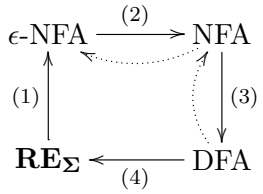
To sum up, the three classes of automata we defined (DFAs, NFAs and ϵ-NFAs) recognise the same class of languages. We shall now find that the languages denoted by regular expressions are exactly the languages accepted by finite automata, namely the *regular languages*.

Theorem 1 (Kleene)**.** *Let $\mathcal{L}$ be a language.*

$$\exists \mathcal{A}, L(\mathcal{A}) = \mathcal{L} \iff \exists E : \mathbf{RE}_{\mathbf{\Sigma}}, L(E) = \mathcal{L}$$

Figure 1.2 shows the constructions required to prove this equivalence. An arrow from A to B means that there is a construction from an object of type A to an equivalent object of type B. A dotted arrow indicates that there is an injection from A to B. We have already studied the removal of ϵ-transitions, and the accessible subset construction. We proceed to prove that for every regular expression, there is an equivalent ϵ-NFA.

Figure 1.2 Showing the equivalence of four different notations for regular languages



(1) Automata construction
(2) ϵ-transitions removal
(3) Accessible subset construction
(4) State elimination

1.1.2 From regular expressions to automata

There are several ways of constructing an ϵ-NFA from a regular expression. We choose here Thompson's construction [155] because of its simplicity, and to lay the ground for the remaining of this chapter (we will discuss this choice of presentation in §1.9). All of the automata we construct here are ϵ-NFAs which may have several initial and accepting states. We proceed by structural induction on the given regular expression and the corresponding steps are depicted in Fig. 1.3. (Note that in some constructions of Fig. 1.3, we use larger arrows to indicate that we consider sets of initial states, or all-pairs transitions between two sets of states in these drawings. Conversely, we will have to consider sets of final states.)

The base cases for the induction are the variable case, the ϵ case and the $\emptyset$ case (which corresponds to an automaton with two states and no transition). The ϵ-NFA in Fig. 1.3(a) clearly recognises the corresponding language. There are three inductive cases to consider, depending on the shape of the regular expression:

Case $A + B$**.** By induction, there are two ϵ-NFAs $\mathcal{M}_A = \langle Q_A, T_A, I_A, F_A\rangle$ and $\mathcal{M}_B = \langle Q_B, T_B, I_B, F_B\rangle$ such that $L(A) = L(\mathcal{M}_A)$ and $L(B) = L(\mathcal{M}_B)$. Since we may rename the states of a ϵ-NFA, we may assume that Q_A and Q_B are disjoint. We build the following automaton, as depicted in Fig. 1.3(b):

$$\mathcal{M}_{A+B} = \langle Q_A \cup Q_B, T_A \cup T_B, I_A \cup I_B, F_A \cup F_B\rangle$$

Case $A \cdot B$**.** Similarly, for the product, we build the following automaton (as shown in Fig. 1.3(c)) where we add ϵ-transitions between the final states of $\mathcal{M}_A$, and the initial states of $\mathcal{M}_B$:

$$\mathcal{M}_{A\cdot B} = \langle Q_A \cup Q_B, T_A \cup T_B \cup \{(f, \epsilon, i) \mid f \in F_A, i \in I_B\}, I_A, F_B\rangle$$

Case $A^\star$**.** By induction, there is an ϵ-NFAs $\mathcal{M}_A = \langle Q, T, I, F\rangle$ such that $L(A) = L(\mathcal{M}_A)$. We assume that the state labelled with 0 is not in Q. We build the following automaton, as depicted in Fig. 1.3(d), where we add ϵ-transitions between the accepting states and initial states of $\mathcal{M}_A$, as well as an automaton in parallel that recognises $\{\epsilon\}$:

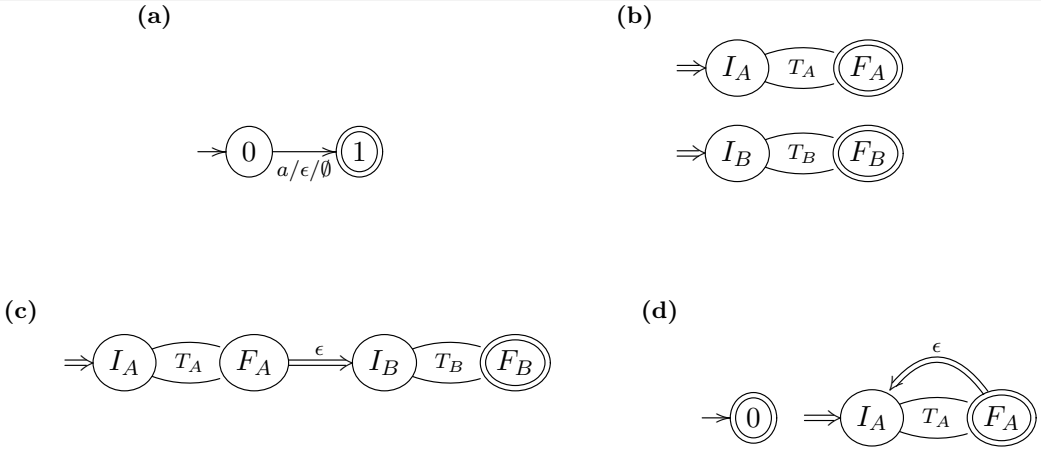
$$\mathcal{M}_{A^\star} = \langle Q \cup \{0\}, T \cup \{(f, \epsilon, i) \mid f \in F, i \in I\}, I \cup \{0\}, F \cup \{0\}\rangle$$

Theorem 2. *Let $\mathcal{M}$ be the automaton built from a regular expression E using Thompson's construction. We have:*

$$L(\mathcal{M}) = L(E).$$

Proof. The proof is routine induction over the structure of the regular expression E. □

Figure 1.3 Thompson's construction



1.1.3 Building regular expressions from automata

There are two standard methods to compute a regular expression that denote the language recognised by an automaton. The first one is combinatorial [88], and goes through the construction of regular expressions that describe progressively broader sets of paths in the transition diagram of a DFA $\mathcal{D} = \langle (q_i)_{0 \leq i \leq n}, T, \{q_0\}, F \rangle$. One can define by induction regular expressions that denote sets of words that label paths from a state q_i to a state q_j without using states numbered higher than a given k:

$$E_{ij}^k = E_{ik}^{k-1} \cdot \left(E_{kk}^{k-1}\right)^\star \cdot E_{kj}^{k-1} + E_{ij}^{k-1},$$
$$E_{ij}^0 = \begin{cases} \{a \mid \delta(q_i, a) = q_j\} & \text{if } i \neq j, \\ \{a \mid \delta(q_i, a) = q_j\} \cup \{\epsilon\} & \text{if } i = j. \end{cases}$$

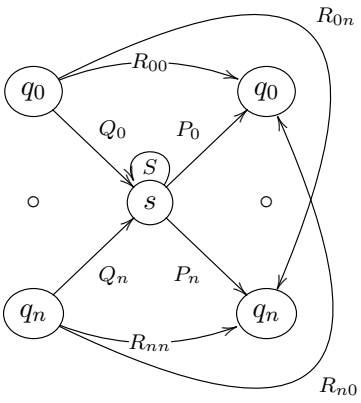
Then, E_{ij}^n denotes the language recognised by the whole automaton starting at state q_i and going to state q_j. Let E be the sum of the regular expressions E_{0j}^n for q_j in F. Then, E denotes the language recognised by the automaton:

$$L(\mathcal{D}) = L(E).$$

The second method, namely the *state elimination method* [87], is less combinatorial and more intuitive. The idea is to eliminate one by one states in an automaton to reduce its size. In order to do so, we have to consider automata that have regular expressions as labels[1]. Figure 1.4 shows a generic state s about to be eliminated in an automaton with $n+1$ states. The arc from q_i to s is labelled by Q_i, and the arc from s to q_j is labelled by P_j. (Note that the P_i's and Q_i's may be the regular expression 0 if some of these arcs do not exist.) Without loss of generality, we can assume that s is the state with number $n+1$, which gives a particular block shape to the transition matrix of the automaton: the $(n \times n)$ matrix R corresponds to transitions not going through s, the (1×1) matrix S corresponds to transitions going from s to s, the $(n \times 1)$ rectangular matrix Q corresponds to the transitions going from one of the q_i's to s, and the $(1 \times n)$ rectangular matrix P corresponds to the transitions going from s to one the q_i's.

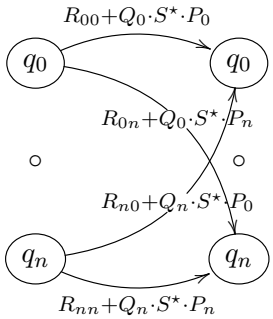
[1] In this case, the language recognised by an automaton is the language denoted by the union, over all paths going from the initial state to a final state, of the regular expressions that label that path (i.e., the concatenation of the regular expressions that label the transitions in the path).

Figure 1.4 A state s about to be eliminated (transition diagram and transition matrix)



$$\left(\begin{array}{ccc|c} & \vdots & & \vdots \\ \cdots & R_{ij} & \cdots & Q_i \\ & \vdots & & \vdots \\ \hline \cdots & P_j & \cdots & S \end{array}\right)$$

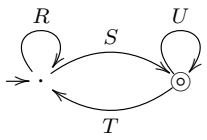
Figure 1.5 Result of eliminating s from Fig. 1.4 (transition diagram and transition matrix)



$$\left(\begin{array}{ccc} & \vdots & \\ \cdots & R_{ij} + Q_i \cdot S^\star \cdot P_j & \cdots \\ & \vdots & \end{array}\right)$$

Figure 1.5 shows what happens when we remove the state s. In order to compensate for the elimination of s, we introduce, for each pair (q_i, q_j), a regular expression that represents all the paths that start at q_i, go through s, perhaps doing some loops, and finally go to p_j. A suitable regular expression is $Q_i \cdot S^\star \cdot P_j$.

Then, the strategy to compute a regular expression from a DFA $\mathcal{D} = \langle Q, T, \{q_0\}, F\rangle$ is to apply the state elimination to remove all states except the initial state and a given final state q_j, producing a reduced finite automaton with one[2] or two states. It is then easy to compute the regular expression E_j associated to this given final state. Indeed, the following generic two-state automaton recognises the language denoted by $(R + S \cdot U^\star \cdot T)^\star \cdot S \cdot U^\star$.



A regular expression that denotes the language recognised by $\mathcal{D}$ is the sum of the E_j's over q_j in F. Note that the state-elimination method may be seen as a general transformation of the transition matrix of an automaton. In fact, it can also be applied to NFAs or ϵ-NFAs, if the final summation is extended to several initial states. This is actually why we presented this construction: we will see analogues of it in later sections.

[2] If the initial state is also a final state.

We have now completed the proof of Thm. 1. We proceed to use the aforementioned automata construction to actually check the equivalence of regular expressions.

1.1.4 Checking equivalence of regular expressions.

To prove that two regular expressions denote the same language, we can check that the DFAs corresponding to these expressions are equivalent.

The Myhill-Nerode theorem [117] provides a first method to check equivalence of DFAs.

Theorem 3 (Myhill-Nerode)**.** *Let $L \subset \Sigma^*$. The following statement are equivalent:*

1. *There is a DFA that recognises L.*
2. *The relation $\equiv_L$ defined by $x \equiv_L y \triangleq \forall w \in \Sigma^\star, x \cdot w \in L \iff y \cdot w \in L$ is of finite index.*

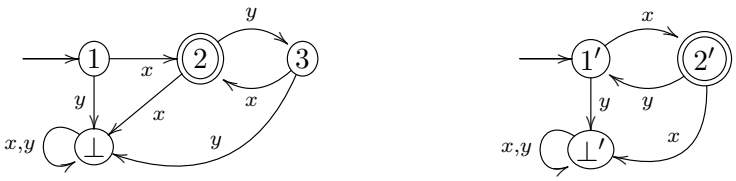
Thus, for a given DFA $\mathcal{A}$, there exists a DFA $\mathcal{B}$ with a minimum number of state that recognise the same language; and such a minimised DFA is unique (except that states can be given different names). The number of states of $\mathcal{B}$ corresponds to the number of equivalence classes of $\equiv_{L(\mathcal{A})}$. Thus, two DFAs are equivalent, if their respective minimised DFAs are equal up-to isomorphisms, which can be checked by exploring all state permutations.

There is a more direct and efficient approach that does not require minimisation: one can perform an on the fly *2-simulation* check, using an almost linear algorithm by Hopcroft and Karp [1]. This algorithm proceeds as follows: it computes the disjoint union of the two DFAs, and checks that the former initial states are equivalent by constructing incrementally an equivalence relation on states. Intuitively, two states are equivalent if they can match each other's transitions to reach equivalent states, with the constraint that no accepting state can be equivalent to a non-accepting state. Formally, a suitable equivalence relation $\approx$ is called a *2-simulation* and satisfies:

$$\forall q, q' \in Q, \forall a \in \Sigma, q \approx q' \implies (\delta(q,a) \approx \delta(q',a) \text{ and } q \in F \iff q' \in F)$$

This implies that starting from two equivalent states, the automaton recognises the same language.

Example 2. *Here, we prove that $L(x \cdot (y \cdot x)^\star) = L((x \cdot y)^\star \cdot x)$. We first build a DFA that recognise each side of the equation:*



Then, we check that the following relation is a 2-simulation relation:

$$\bot \approx \bot' \qquad 2 \approx 2' \qquad 3 \approx 1' \approx 1$$

Since the initial states of the two automata are in the 2-simulation relation $\approx$, the automata recognise the same language. Hence, $x \cdot (y \cdot x)^\star \sim (x \cdot y)^\star \cdot x$ holds.

From the implementation point-of-view, it suffices to fold through pairs of states along the transition relation, starting from the pair of initial states, and to maintain a data structures that keeps track of the equivalence classes of states. Using an efficient disjoint-sets data structure makes it possible to merge equivalence classes in quasi-constant time [57], and makes the whole equivalence testing algorithm run in quasi-linear time w.r.t. to the size of the disjoint sum automaton.

Figure 1.6 The rules of Kleene algebra

$$x + (y + z) \equiv (x + y) + z \qquad x + y \equiv y + x \qquad x + x \equiv x \qquad x + 0 \equiv x$$

$$x \cdot (y \cdot z) \equiv (x \cdot y) \cdot z \qquad x \cdot 1 \equiv x \qquad 1 \cdot x \equiv x$$

$$x \cdot (y + z) \equiv x \cdot y + x \cdot z \qquad (x + y) \cdot z \equiv x \cdot z + y \cdot z \qquad x \cdot 0 \equiv 0 \qquad 0 \cdot x \equiv 0$$

$$1 + x \cdot x^\star \equiv x^\star \quad \text{(Ax1)} \qquad \frac{x \cdot y \leq y}{x^\star \cdot y \leq y} \quad \text{(Ax2)} \qquad \frac{y \cdot x \leq y}{y \cdot x^\star \leq y} \quad \text{(Ax3)}$$

$$\frac{x \equiv y \qquad y \equiv z}{x \equiv z} \qquad \frac{x \equiv y}{y \equiv x} \qquad x \equiv x$$

$$\frac{x \equiv x' \qquad y \equiv y'}{x \cdot y \equiv x' \cdot y'} \qquad \frac{x \equiv x' \qquad y \equiv y'}{x + y \equiv x' + y'} \qquad \frac{x \equiv x'}{x^\star \equiv x'^\star}$$

1.1.5 Toward the axiomatisation of the equality of regular languages

We have recalled that the languages recognised by finite automata are the same as the languages denoted by regular expressions. We have also recalled that it is possible to check the equivalence of regular expressions through finite automata constructions. We now devote a section to the axiomatisation of the equality of regular languages, in the more general context of Kleene algebras, as proposed by Kozen [100].

1.2 Kleene algebras

The equational theory of $\mathbf{RE}_\Sigma$ has been called the *algebra of regular events*, and was first studied by Kleene [98], who posed axiomatisation as an open problem. Redko [132] proved in 1964 that no finite set of equational axioms can characterise the algebra of regular events. Salomaa [137] gave two complete axiomatisations in 1966 that depend on inference rules that are not valid under substitution, making the axiomatisations unsound under non-standard interpretations[3] Other axiomatisations involve infinitary treatment, which is not suited for algebraic reasoning. In order to solve the problem in an algebraic fashion, we need to move to the context of Kleene algebras.

Definition 5 (Kleene algebra)**.** *A* Kleene algebra *[100] is a structure* $\mathcal{K} = \langle K, +, \cdot, \_^\star, 0, 1 \rangle$ *with binary operations* $+$ *and* $\cdot$*, unary operation* $\_^\star$*, and constants 0 and 1 that satisfies the axioms and inference rules in Fig. 1.6 (where* $\leq$ *is the preorder defined by* $x \leq y \triangleq x + y \equiv y$*).*

Terms of Kleene algebras, ranged over using lower-case letters, are called *regular expressions*, irrespective of the considered model.

1.2.1 Examples of Kleene algebras

Kleene algebras abound in computer science, and we are going to give some examples. An important one is $\mathbf{RL}_\Sigma$.

[3] Interpretations of the regular expressions that differ from L.

Example 3 (Regular languages)**.** *The structure* $\langle \mathbf{RL}_{\mathbf{\Sigma}}, \cup, \cdot, \_^\star, \emptyset, \{\epsilon\}\rangle$ *is a Kleene algebra.*

If we consider two regular expressions α and β that are provably equal using the rules in Fig. 1.6, we have $\alpha \sim \beta$. That is, the axioms of Kleene algebra together with the axioms of equational reasoning constitute a correct deductive system for the equivalence of regular expressions.

The next example comes from the starting point for this work: proofs involving binary relations, as we mentioned in §1.

Example 4 (Binary relations)**.** *If we consider the family of binary relations on a set* X *with operations*

$$\begin{aligned} Id &\triangleq \{(x,x) \mid x \in X\} \\ R \circ S &\triangleq \{(x,z) \mid \exists y, (x,y) \in R \wedge (y,z) \in S\} \\ R \cup S &\triangleq \{(x,y) \mid (x,y) \in R \vee (x,y) \in S\} \\ R^\star &\triangleq \bigcup_{0 \leq n} R^n \quad \textit{(where } R^0 \triangleq Id \textit{ and } R^{n+1} \triangleq R^n \circ R \textit{ for } 0 \leq n \textit{)} \end{aligned}$$

then the structure $\langle \mathcal{P}(X \times X), \cup, \circ, \_^\star, \emptyset, Id\rangle$ *is a Kleene algebra.*

This means that the equational theory of Kleene algebras can be used as an alternative to "long-winded low-level proofs" in order to prove lemmas like the following:

$$\begin{aligned} S \circ (S \circ S^\star \circ R^\star \cup R^*) &\subseteq S \circ S^\star \circ R^\star \qquad [154] \\ (S \cup R)^\star &\subseteq S^\star \cup R^\star \cup R^\star \circ S^\star \cup R^\star \circ (S^\star \circ R^\star)^+ \circ S^\star \end{aligned}$$

In fact, these are just two inequalities of regular expressions, (interpreted over relations instead of regular languages). We shall see that decidability of the equivalence of regular expressions can be lifted to binary relations. Since the above inequations hold in the model of regular languages, they also hold in the model of homogeneous relations.

From binary relations, we can move to finite graphs to find other examples of Kleene algebras: The family of boolean matrices of size $(n \times n)$ can represent *adjacency matrices*; if we consider *weighted graphs*, we can interpret the family of $(n \times n)$ matrices over $\mathbb{R}^+ \cup \{\infty\}$ as *weighted adjacency matrices*. Both form Kleene algebras. Indeed, given a Kleene algebras of elements, it is possible to construct a general star operation on matrices, which happens to correspond to the computation of all-pairs shortest paths for tropical algebras, or to the computation of all-pairs reachability for the Kleene algebras of booleans. Generally speaking, if $\mathcal{K}$ is a Kleene algebra, then the family of square matrices over $\mathcal{K}$ form a Kleene algebra.

Example 5 (Tropical algebra)**.** *We consider a* $(min, +)$ *algebra (or tropical algebra*[4]*), with operators:*

$$x \oplus y = \min\{x, y\} \qquad x \otimes y = x + y \qquad x^\star = 0$$

The structure $\mathbb{T} = \langle \mathbb{R}^+ \cup \{\infty\}, \oplus, \otimes, \_^\star, \infty, 0\rangle$ *is a Kleene algebra. Moreover, the family of square matrices over* $\mathbb{T}$ *form a Kleene algebra* $\mathcal{M}_{(n,n)}(\mathbb{T})$*. The matrix multiplication correspond to the distance product:*

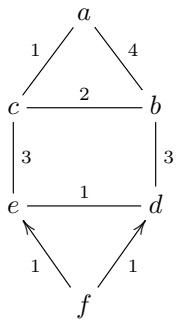
$$(A \times B)_{(i,j)} = \min_{k=1}^{n} \left\{ A_{(i,k)} \oplus B_{(k,j)} \right\}.$$

The star operation on matrices correspond to the computation of all-pairs shortest paths.

[4]Note that there are many variations on this particular Kleene algebra. For instance, it is possible to consider the whole extended real number line, taking $a \oplus b$ as minimum, $a \otimes b$ as addition with $-\infty \otimes \infty = \infty$, and $a^\star$ being 0 for non-negative a and $-\infty$ for negative a. This forms also a Kleene algebra.

We shall present the construction of the Kleene algebra of matrices over a given Kleene algebra in the next section. Yet note that this is a generic construction: in the previous example, it happens that the (generic) matrix multiplication corresponds to the distance product and that the (generic) star operation on matrices corresponds to the computation of all-pairs shortest paths. The next example shows how this construction may be used in practice.

Example 6 (Computing shortest paths in a graph)**.** *Consider the following oriented graph, where each edge is weighted (we use lines for bi-directional edges). The corresponding weighted adjacency matrix M is given in the middle (lines and columns are not labelled but correspond to the vertices of the graph in the usual alphabetic order). Finally, we represent the matrix $M^\star$ on the right.*



Graph

$$\begin{pmatrix} \infty & 4 & 1 & \infty & \infty & \infty \\ 4 & \infty & 2 & 3 & \infty & \infty \\ 1 & 2 & \infty & \infty & 3 & \infty \\ \infty & 3 & \infty & \infty & 1 & \infty \\ \infty & \infty & 3 & 1 & \infty & \infty \\ \infty & \infty & \infty & 1 & 1 & \infty \end{pmatrix}$$

Weighted adjacency matrix

$$\begin{pmatrix} 0 & 3 & 1 & 5 & 4 & \infty \\ 3 & 0 & 2 & 3 & 4 & \infty \\ 1 & 2 & 0 & 4 & 3 & \infty \\ 5 & 3 & 4 & 0 & 1 & \infty \\ 4 & 4 & 3 & 1 & 0 & \infty \\ 5 & 4 & 4 & 1 & 1 & 0 \end{pmatrix}$$

All-pairs shortest path

1.2.2 Elementary consequences

We have given several examples of models of Kleene algebras. We can now derive some elementary standard properties from the axioms of Kleene algebra [53, 100]: each of these theorems will apply to any model of Kleene algebra.

Theorem 4. *For any regular expression a, a^* is the unique element satisfying (Ax1), (Ax2) and (Ax3). Moreover, the following identities hold:*

$$x^\star \equiv x^\star \cdot x^\star \equiv (x^\star)^\star \equiv (x+1)^\star \equiv 1 + x^\star \cdot x.$$

Theorem 5 (Bisimulation rule)**.** *The following implication holds:*

$$a \cdot x \equiv x \cdot b \quad \Longrightarrow \quad a^\star \cdot x \equiv x \cdot b^\star$$

Theorem 6 (Denesting rule)**.** *The following equation holds:*

$$(x+y)^\star \equiv x^\star \cdot (y \cdot x^\star)^\star$$

Theorem 7 (Sliding rule)**.** *The following equation holds:*

$$x \cdot (y \cdot x)^\star \equiv (x \cdot y)^\star \cdot x$$

1.2.3 Deciding equations of Kleene algebras

We have studied several models of Kleene algebras, and some general standard theorems. The remainder of this section is devoted to the *mechanisation* of proofs of equations in Kleene algebra. Thanks to finite automata theory [98, 130], we know that equality of regular languages is decidable:

"two regular expressions are equivalent if and only if the corresponding deterministic automata are equivalent".

However, this result does not directly apply to arbitrary models of Kleene algebras. We actually need a more recent theorem by Kozen [100] (independently proved by Krob [105]):

"two regular expressions α and β denote the same regular language if and only if $\alpha \equiv \beta$ is derivable using the axioms of Kleene algebra".

In other words, regular languages are the initial model of Kleene algebras: we can resort to finite automata algorithms to solve equations in an arbitrary Kleene algebra $\mathcal{K}$.

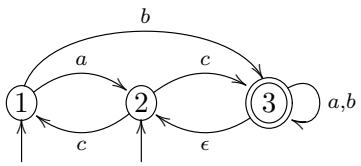
A note on the vocabulary. We make precise the signification we attach to some words. In the following, we will study Kozen's *initiality* theorem, and the corresponding implementation of a decision procedure for Kleene algebras in Coq, the latter being the main focus of this chapter.

This decision procedure will be *sound*, i.e., we prove that equations holding in the model of regular languages can be lifted to arbitrary Kleene algebras. This decision procedure will be *complete*: every equation that is a consequence of the axioms of Kleene algebra is provable using this methodology. Therefore, even if Kozen's theorem was named "A completeness theorem for the algebra of regular events" because the axioms of Kleene algebras he proposed are complete for the equational theory of regular languages, we will prefer to use the term "initiality theorem" to avoid confusion.

1.2.4 The main idea

The main idea of Kozen's proof is to encode finite automata using matrices over a Kleene algebra $\mathcal{K}$, and to replay the algorithms at this algebraic level. Indeed, a finite automaton with transitions labelled by the elements of $\mathcal{K}$ can be represented with three matrices $\langle u, M, v\rangle \in \mathcal{M}_{(1,n)}(\mathcal{K}) \times \mathcal{M}_{(n,n)}(\mathcal{K}) \times \mathcal{M}_{(n,1)}(\mathcal{K})$ such that n is the number of states of the automaton; u and v are 0-1 vectors respectively coding for the sets of initial and accepting states; and M is an analogue of the transition matrix: $M_{i,j}$ is a regular expression that labels transitions from state i to state j. In the following section, we will make precise the translation in this setting of the various kinds of finite automata from §1.1, but we first give an example to give the intuition behind these *matricial automata*, and Kozen's proof.

Example 7. *Consider the following non-deterministic finite automaton with ϵ-transitions, with three states:*



This automaton can be represented with the following matrices:

$$u = \begin{bmatrix} 1 & 1 & 0 \end{bmatrix} \qquad M = \begin{bmatrix} 0 & a & b \\ c & 0 & c \\ 0 & 1 & a+b \end{bmatrix} \qquad v = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

We remark that the product $u \cdot M \cdot v$ is a scalar (i.e., a regular expression), which can be thought of as the set of one-letter words accepted by the automaton—in the example, $b + c$. Similarly, $u \cdot M \cdot M \cdot v$ corresponds to the set of two-letters words accepted by the automaton—in the example $a \cdot c + (b + c) \cdot (a + b)$, or fully expanded:

$$a \cdot c + b \cdot a + b \cdot b + c \cdot a + c \cdot b.$$

Therefore, to mimic the behaviour of a finite automaton, we just need to iterate over the matrix M. This is possible thanks to another theorem, which actually is the crux of the initiality theorem: *"square matrices over a Kleene algebra form a Kleene algebra"*. We hence have a star operation on matrices, and we can interpret an automaton algebraically, by considering the product $u \cdot M^\star \cdot v$. Again, in the example, this computation yields

$$(a \cdot c + b + c) \cdot (a + b + c)^\star \cdot (a + b)^\star$$

which denotes exactly the language recognised by the automaton.

To show his initiality theorem, Kozen relies and improves on earlier work to show how to encode the classical combinatorial constructions of §1.1 into such matricial automata. This is the topic of the next section.

1.3 Initiality theorem

In this section, we give an account of the proof of Kozen's initiality theorem from [100], for the sake of self-containedness. Even if there are some differences with regards to this seminal paper, we will omit most of the proofs in this section: the corresponding lemmas will be formalised in Coq in the next sections.

1.3.1 Typed Kleene algebras

From matrices to typed Kleene algebras. As was hinted above, matrices over a Kleene algebra are the key ingredient of Kozen's initiality theorem. However, we will need to work with *rectangular* matrices at several places, to prove the correctness of some steps of the algorithm. (Moreover, dealing with rectangular matrices allows one to treat vectors as a special case of matrices, and thus to factorise proofs.)

While *square* matrices over a semiring form a semiring, this is not immediately the case for *rectangular* matrices: the various operations are only partial. For instance, it is not possible to take the product of two matrices whose dimensions do not agree. In order to alleviate this problem, Kozen relied on several ad-hoc arguments in his proof [100]. He later defined and studied the notion of *typed Kleene algebra* [103], in which objects have types. While the main motivation was non-square matrices, there are many other interpretations [101, 103]: heterogeneous binary relations (rather than binary relations on a single set), traces, Kleene algebra with tests.

Typed Kleene algebras lay the ground to argue formally why some theorems of Kleene algebra hold even for non-square matrices. For instance, Thm. 5:

$$a \cdot x \equiv x \cdot b \to a^\star \cdot x \equiv x \cdot b^\star$$

holds even if a is interpreted as a $n \times n$ matrix, b as a $m \times m$ matrix, and x as a $n \times m$ matrix.

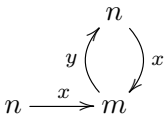
Figure 1.7 From Kleene algebras to typed Kleene algebras.

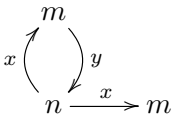
```
X: Type.

dot:  X → X → X.
one:  X.
plus: X → X → X.
zero: X.
star: X → X.

dot_neutral_left: ∀ x, dot one x ≡ x.
...
```

```
T: Type.
X: T → T → Type.

dot:  ∀ n m p, X n m → X m p → X n p.
one:  ∀ n,     X n n.
plus: ∀ n m,   X n m → X n m → X n m.
zero: ∀ n m,   X n m.
star: ∀ n,     X n n → X n n.

dot_neutral_left: ∀ n m (x: X n m), dot one x ≡ x.
...
```

Adding types. The idea is to introduce a type discipline in which regular expressions α have types of the form $n \to m$, where n and m are elements of a set of indices (or types). That is, we generalise the algebraic structures using *types*. An example is given in Fig. 1.7: a typical Coq signature[5] for Kleene algebras is presented on the left-hand side; we moved to the signature on the right-hand side, where a set `T` of indices is used to constrain the various operations. These abstract indices can be thought of as matrix dimensions. We actually moved to a categorical setting: `T` is a set of objects, `X n m` is the set of morphisms from `n` to `m` (written $n \to m$), `one` is the set of identities, and `dot` is composition. The semi-lattice operation (`plus`) operates on fixed (but arbitrary) homsets, and accordingly `zero` has all types. Kleene star operates only on *square* morphisms—those whose source and target coincide.

Removing types. As we mentioned, typed structures not only make it easier to work with rectangular matrices, they also give rise to a wider range of models. In particular, we can consider heterogeneous binary relations rather than binary relations on a single fixed set. This leads to the following question: how can the initiality theorem be extended to this more general setting? Consider for example the equation $x \cdot (y \cdot x)^\star \equiv (x \cdot y)^\star \cdot x$ we studied in §1.1.4 in the model of regular languages and showed to be a general theorem (Thm. 7). This is also a theorem of typed Kleene algebras as soon as x and y are respectively given types $n \to m$ and $m \to n$, for some n, m.





However, how to ensure that the automata algorithms respect types and actually give valid, well-typed, proofs? For instance, what is the typed equivalent of the subset-construction and equivalence checking that apply to the above automata? Extending the decision procedure to work with typed elements would require to devise typed analogues to the usual finite automata algorithms. For instance, it would most certainly require to keep track of the types of nodes. Instead of going in this direction, we rely on the following theorem, which allows one to erase

[5] We use Coq here since we will actually use the signature on the right-hand side in the later sections.

types, i.e., to transform a typed equality goal into an untyped one:

$$\frac{\vdash u \equiv v \qquad \Gamma \vdash u \rhd \alpha : n \to m \qquad \Gamma \vdash v \rhd \beta : n \to m}{\vdash \alpha \equiv \beta : n \to m} \tag{$*$}$$

Here, $\Gamma \vdash u \rhd \alpha : n \to m$ reads "under the evaluation and typing context Γ, the untyped term u can be evaluated to α, of type $n \to m$"; this predicate can be defined inductively in a straightforward way, for various algebraic structures. The theorem can then be rephrased as follows: "if the untyped terms u and v are equal modulo the rules of Kleene algebra, then for all typed interpretations α and β of u and v, α and β are equal modulo the rules of typed Kleene algebra".

We rely on corresponding theorems for semi-lattices, monoids, semirings, and Kleene algebras, so that all decision procedures for untyped settings can be used in typed settings—and in particular with rectangular matrices. Therefore, in the following, we will also use the symbol $\equiv$ to denote equations in the setting of typed Kleene algebras. See Pous' work [129] for a theoretical study of these untyping theorems; also note that Kozen [103] investigated a similar question and came up with a slightly different solution: he solves the case of the Horn theory rather than the equational theory, at the cost of working in a restrained form of Kleene algebras.

1.3.2 Matrices over a Kleene algebra

It is a well-known result that square matrices over a semiring form a semiring [53], and this result can be extended to rectangular matrices in the light of the previous remarks. We now proceed to the construction of the typed Kleene algebra of matrices over a given Kleene algebra. In the following, we fix $\mathcal{K}$ to be a given Kleene algebra.

We have to define a star operation on matrices and prove that it satisfies the laws (Ax1), (Ax2) and (Ax3). We will proceed in two steps. First, we will give an intuitive definition of the star operation on matrices (denoted $\natural$), and we will then relate this definition to the usual definition of Kozen [100] (denoted $\star$, as usual). The star operation on matrices has been called *closure* [1] in the context of closed semirings[6], and we may use this term to reflect the analogy with computations of, e.g., transitive closure of transitions relations.

An intuition on the closure of a matrix. This definition is made by induction on the size of the square[7] matrix X. The problem is trivial if the matrix is empty or of size 1×1: we simply use the star operation on the elements. Otherwise, partition X into four blocks:

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

Note that the size of the blocks do not matter, as long as one take square matrices for A and D. Then, the star operation $\natural$ on the matrix X may be defined as:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}^{\natural} \triangleq \begin{bmatrix} E & F \\ G & H \end{bmatrix} \quad \text{where} \quad \begin{cases} E = (A + B \cdot D^{\natural} \cdot C)^{\natural} \\ F = E \cdot B \cdot D^{\natural} \\ G = D^{\natural} \cdot C \cdot E \\ H = (D + C \cdot A^{\natural} \cdot B)^{\natural} \end{cases}$$

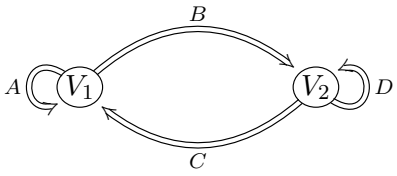
[6] Closed semirings are idempotent semiring equipped with an infinite summation operator, that satisfies infinitary associativity, commutativity, idempotence, and distributivity laws. The sole purpose of countable sums seem to define the $\star$ operation and semi-rings always form Kleene algebras [102].

[7] Remember that in a typed Kleene algebra, the Kleene star operates only on square morphisms, i.e., elements of type `X n n`.

The special case where C is zero might give more insight:

$$\left[\begin{array}{c|c} A & B \\ \hline 0 & D \end{array}\right]^\natural = \left[\begin{array}{c|c} A^\natural & A^\natural \cdot B \cdot D^\natural \\ \hline 0 & D^\natural \end{array}\right] \tag{‡}$$

This definition can be explained if we interpret the matrix X of size $n+m$ as a graph $G = (V, X)$, where the vertices are partitioned in two sets V_1 (resp. V_2) of size n (resp. m). The matrix A (of size $n \times n$) represents the edges between vertices of V_1; the matrix D (of size $m \times m$) represents the edges between vertices of V_2; the matrix B represents the edges that go from V_1 to V_2; the matrix C represents edges from V_2 to vertices in V_1. This arrangement can be depicted as:



In this interpretation, $X^\natural$ is the reflexive and transitive closure of the transition relation X, as it is hinted by the definition of the four quadrants of $X^\natural$. Indeed, a path between two vertices of V_1 can be decomposed in two elementary forms: either it stays in V_1 (taking an edge in A); or it jumps to V_2 (B), loops in V_2 ($D^\natural$), and jumps back to V_1 (D). Each path between two vertices of V_1 can be represented as a succession of edges in $A + B \cdot D^\natural \cdot C$. Hence, $E = (A + B \cdot D^\natural \cdot C)^\natural$ represents all the paths between vertices of V_1. A path from V_1 to V_2 may stay in come and go between V_1 and V_2 (taking an edge in E), and then finally jumps to V_2 (taking an edge in B) to remain there (taking an edge in $D^\natural$). Hence, the upper-left quadrant of $X^\natural$ is $E \cdot B \cdot D^\natural$. The other quadrants are defined through similar reasoning.

An equivalent definition. The actual definition of $X^\star$ by Kozen [100] is different: there, the lower-right quadrant is expressed as $H' = D^\star + D^\star \cdot C \cdot E \cdot B \cdot D^\star$. We may trace back this expression to the textbook by Aho et al. [1, p. 205] where the authors study the complexity of the computation of the reflexive and transitive closure of a transition matrix. They compute the quadrants of the matrix through a sequence of steps:

$$X^\star = \begin{bmatrix} E' & F' \\ G' & H' \end{bmatrix} \quad \text{where} \quad \left\{\begin{array}{lcl} T_1 &=& D^\star \\ T_2 &=& B \cdot T_1 \\ E' &=& (A + T_2 \cdot C)^\star \\ F' &=& E' \cdot T_2 \\ T_3 &=& T_1 \cdot C \\ G' &=& T_3 \cdot E' \\ H' &=& T_1 + G' \cdot T_2 \end{array}\right. \quad \text{i.e.} \quad \left\{\begin{array}{lcl} E' &=& (A + B \cdot D^\star \cdot C)^\star \\ F' &=& E \cdot B \cdot D^\star \\ G' &=& D^\star \cdot C \cdot E \\ H' &=& D^\star + G' \cdot B \cdot D^\star \end{array}\right.$$

First, we can prove that these two constructions compute the same result.

Theorem 8. *For all square matrix X, we have $X^\natural \equiv X^\star$*

Proof. The proof goes by induction on the size of the matrix. For the inductive case, the following equations hold trivially.

$$E \equiv E' \qquad F \equiv F' \qquad G \equiv G'$$

Then, to show $H \equiv H'$ we can show that the following equation holds even if c and d are interpreted as rectangular matrices:

$$d^\star + d^\star \cdot c \cdot \big(a + b \cdot d^\star \cdot c\big)^\star \cdot b \cdot d^\star \equiv (d + c \cdot a^\star \cdot b)^\star$$

The proof of this equation is routine:

$$\begin{aligned}
d^\star + d^\star \cdot c \cdot (a + b \cdot d^\star \cdot c)^\star \cdot b \cdot d^\star &\equiv d^\star + d^\star \cdot c \cdot a^\star \cdot (b \cdot d^\star \cdot c \cdot a^\star)^\star \cdot b \cdot d^\star && \text{by Thm. 6} \\
&\equiv d^\star \cdot (1 + (c \cdot a^\star) \cdot ((b \cdot d^\star) \cdot (c \cdot a^\star))^\star \cdot (b \cdot d^\star) \\
&\equiv d^\star \cdot (1 + (c \cdot a^\star \cdot (b \cdot d^\star))^\star \cdot (c \cdot a^\star) \cdot (b \cdot d^\star)) && \text{by Thm. 7} \\
&\equiv d^\star \cdot (c \cdot a^\star \cdot b \cdot d^\star)^\star && \text{by Thm. 4} \\
&\equiv (d + c \cdot a^\star \cdot b)^\star && \text{by Thm. 6}
\end{aligned}$$

(Note that, an elegant alternative to the trial and error method to find this proof is to use the decision procedure from the later sections: it verifies instantaneously the equation.) □

While the second construction may look more mysterious, it is of high practical interest: using these steps to compute $X^\star$, one needs only to compute the star of 2 matrices of lesser size, and 6 matrix multiplications. However the definition of $X^\natural$ we gave above requires to compute the star of 4 matrices and 7 matrix multiplications. (Note that this is not innocuous: it is demonstrated in [1] that the time $T(n)$ to compute the closure of a square matrix of size n using the efficient method is of the same order than the time $M(n)$ required to multiply two square matrices of size n: if $T(3n) \leq 27T(n)$ and $4M(n) \leq M(2n)$ then $T(n) \in \Theta(M(n))$. Using the alternate inefficient construction yields an exponential blowup.)

In the following, we will therefore use the usual (and efficient) definition of the star operation.

Theorem 9. *For any $X \in \mathcal{M}_{(n,n)}(\mathcal{K})$, the matrix $X^\star$ satisfies the axioms of Kleene algebra. That is,*

$$1 + X \cdot X^\star \equiv X^\star$$

and for any $U \in \mathcal{M}_{(n,m)}(\mathcal{K})$,

$$\begin{aligned}
X \cdot U \leq U &\to X^\star \cdot U \leq U \\
U \cdot X \leq U &\to U \cdot X^\star \leq U
\end{aligned}$$

Proof. The proof that this operation satisfies the laws of Kleene algebra is complicated. It is done in [100] for the case where U is a square matrix, but can be extended to rectangular matrices. Note that the initial remark that Kleene algebras are closed under the formation of matrices essentially goes back to Conway's book [53]. □

To conclude, the standard sum and product operations on matrices together with the above star operation define a typed Kleene algebra (we will make this claim formal in the Coq section). We will now use these matrices to build matricial representation of automata, suitable for algebraic reasoning.

1.3.3 Finite automata

In this section, we refine the intuition we gave in §1.2.3 about the notion of automata over an arbitrary Kleene algebra.

Definition 6 (Matricial finite automaton)**.** *A matricial finite automaton over a given Kleene algebra $\mathcal{K}$ is a triple $\langle u, M, v\rangle \in \mathcal{M}_{(1,n)}(\mathcal{K}) \times \mathcal{M}_{(n,n)}(\mathcal{K}) \times \mathcal{M}_{(n,1)}(\mathcal{K})$: n is the number of states of the automaton; the* states *of the automata are the row and columns indices; u and v are 0-1 vectors respectively coding for the sets of initial and accepting states; and M is the transition matrix: $M_{i,j}$ labels transitions from state i to state j.*

Definition 7 (Evaluation)**.** *The evaluation*[8] *of the automaton $\langle u, M, v\rangle$ is the product $u \cdot M^\star \cdot v$.*

We proceed to revisit the various kinds of automata from §1.1 in the matricial setting. By imposing some constraints on the matrices u and M, we get various kinds of automata:

- An automaton $\langle u, M, v\rangle$ is *simple* if there exist 0-1 matrices J and M_a such that:

$$M = J + \sum_{a\in\Sigma} a \cdot M_a$$

  (this corresponds to the usual definition of ϵ-NFAs);
- in addition, if J is equal to the zero matrix, then the automaton is *ϵ-free* (this corresponds to the usual definition of NFAs);
- finally, if u and all rows of M_a have exactly one 1, then the automaton is *deterministic* (this corresponds to the usual definition of DFAs)

In the following, we shall convert some matricial automata to the combinatorial definitions. Starting from a simple matricial automaton $\langle u, M, v\rangle$ with n states such that

$$M = J + \sum_{a} a \cdot M_a,$$

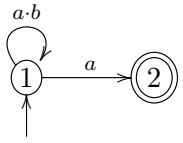
we define:

$$\begin{aligned} Q &= [1, \ldots, n] \\ I &= \{i \mid u_{0i} = 1\} \\ F &= \{i \mid v_{i0} = 1\} \\ T &= \{(p, a, q) \mid (M_a)_{pq} = 1\} \cup \{(p, \epsilon, q) \mid J_{pq} = 1\} \end{aligned}$$

The automaton $\langle Q, T, I, F\rangle$ recognise the language denoted by the regular expression $u \cdot M^\star \cdot v$. We will not use the converse transformation in the following, i.e, from the combinatorial automata to matricial automata. However, converting matricial automata to combinatorial automata allows us to employ, e.g, the Hopcroft-Karp algorithm as an oracle, to build a 2-simulation relation that can be faithfully used in the matricial setting. We will come back to this in §1.3.4.

The next example show that the above definition of matricial automata encompasses automata whose transitions are labelled by arbitrary regular expressions.

Example 8. *The following automaton is not simple:*



$$u = \begin{bmatrix} 1 & 0 \end{bmatrix} \qquad M = \begin{bmatrix} a \cdot b & a \\ 0 & 0 \end{bmatrix} \qquad v = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

[8] Note that the evaluation of an automaton is actually an element of the Kleene algebra $\mathcal{K}$. The language accepted by this automaton is the image under L of the regular expression $u \cdot M^\star \cdot v$. Since we are mainly interested in syntactic objects in the following, we could nevertheless use the term "language accepted by the automata" to denote this regular expression.

It evaluates to the regular expression

$$\begin{bmatrix}1 & 0\end{bmatrix} \cdot \begin{bmatrix}a\cdot b & a\\ 0 & 0\end{bmatrix}^\star \cdot \begin{bmatrix}0\\1\end{bmatrix} \equiv \begin{bmatrix}1 & 0\end{bmatrix} \cdot \begin{bmatrix}(a\cdot b)^\star & (a\cdot b)^\star\cdot a\\ 0 & 1\end{bmatrix} \cdot \begin{bmatrix}0\\1\end{bmatrix} \qquad \text{by } (\ddagger)$$
$$\equiv (a\cdot b)^\star \cdot a.$$

1.3.4 Proving the correctness and completeness

Kozen's initiality theorem states that to prove that an equality $\alpha \equiv \beta$ holds, it suffices to check that the underlying regular languages are equal. Thus, the algorithmic part is not different from the verification of $\alpha \sim \beta$, and goes through the conversion of the regular expressions to deterministic automata. The novelty with respect to this algorithmic construction is that Kozen show how to encode algebraically the combinatorial constructions of §1.1.2 in the matricial automata setting, and that each matricial automata construction preserves the evaluation (in the formal sense that it is possible to derive the equality of the evaluation of the two automata from the axioms of Kleene algebra). The equality of regular languages is then lifted to a theorem that holds in any Kleene algebra.

The first step is to construct a matricial automaton equivalent to a given regular expression. This construction is implicit in the work of Kleene [98] and appears in the work of Conway [53]. The second step is to eliminate the ϵ-transitions that appears in this automaton. The algebraic analogue to the ϵ-transition closure of §1.1 appears in the work of Sakarovitch [136]. The third step is to determinise the automaton to obtain a DFA. The algebraic analogue to the subset construction was introduced by Kozen [100], and independently by Krob [105]. The fourth step is to check that the two deterministic automata corresponding to the initial regular expressions are equivalent. Kozen [100] showed how to encode algebraically the minimisation of finite automata, and conclude his proof by showing that if the underlying minimised automaton are isomorphic, then, the evaluations of the matricial automata are equal (which yields a proof that the original regular expressions are indeed equal in any Kleene algebra). We will present an alternative to this construction, an algebraic analogue of the equivalence test of DFAs, which is more efficient in practice.

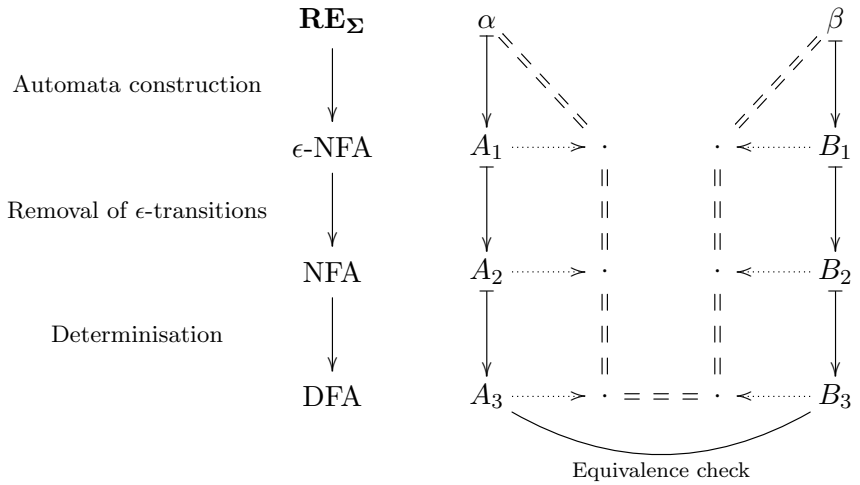
The overall structure of the proof is depicted in Fig. 1.8. The kind of matricial automaton involved is recalled on the left-hand side; the outer part of the righ-hand side corresponds to computations: starting from two regular expressions α and β, two DFA A_3 and B_3 are constructed and tested for equivalence. The proof corresponds to the inner equalities ($==$): we show that each automata construction preserves the semantics of the initial regular expression, through computation of the evaluation (the dotted arrows).

The automata construction depicted in this section are not meant to be efficient, and are done through matrix constructions: we give here an account of Kozen's proof, and efficiency is not required for decidability. However, we lay the ground to argue why the efficient constructions of §1.6 (where we give a more computational account of the initiality theorem as it is currently implemented in our Coq development) are also valid.

Building automata. There are several ways of constructing an ϵ-NFA from a regular expression. Following Kozen [100], we choose here Thompson's construction [155] because of its simplicity. We presented this construction in §1.1.2 . Its algebraic analogue is only a matter of block matrix constructions, and we easily show that the ϵ-NFA built from α evaluates to α, using algebraic laws. Note that this algorithm constructs an automaton which may have several initial and accepting states.

It proceeds by structural induction on the given regular expression. The corresponding matrix construction steps are depicted in Fig. 1.9, and are the exact counterparts of the con-

Figure 1.8 The big picture of the soundness proof



structions in Fig. 1.3. For each case, we provide three matrices that correspond to the vector of initial states, the transition matrix, and the vector of final states.

- Figure 1.9(a) corresponds to the base cases (variable, one, zero).
- Figure 1.9(b) corresponds to union (plus): we recursively build two matricial sub-automata $\mathcal{A} = \langle i_A, A, f_a \rangle$ and $\mathcal{B} = \langle i_B, B, f_b \rangle$, and add them in parallel. This corresponds to the formation of the disjoint union of the set of states, taking the initial states to be the union of the start states of $\mathcal{A}$ and $\mathcal{B}$, and the final states to be the union of the final states $\mathcal{A}$ and $\mathcal{B}$. The transition matrix features exactly the transitions that occurred either in $\mathcal{A}$ or $\mathcal{B}$ with no transition going from $\mathcal{A}$ to $\mathcal{B}$ nor from $\mathcal{B}$ to $\mathcal{A}$.
- Figure 1.9(c) corresponds to concatenation (product): we add ϵ-transitions between the accepting states of the first sub-automaton and the initial states of the second sub-automaton. This is the purpose of the upper right corner of the transition matrix: $f_A \cdot i_B$ is a rectangular matrix of 0 and 1 that represents exactly these ϵ-transitions.
- Figure 1.9(c) corresponds to iteration (star): we add ϵ-transitions between the accepting states and initial states of the sub-automaton (the factor $f_A \cdot i_A$): we shall see that if $\langle i_A, A, f_A \rangle$ evaluates to the regular expression α, then the automaton $\langle i_A, A + f_A \cdot i_A, f_A \rangle$ evaluates to $\alpha \cdot \alpha^\star$. Therefore, we add in parallel an automaton that evaluates to 1.

It remains to prove that these automaton evaluate to the correct regular expressions.

Theorem 10. *Let $\mathcal{A} = \langle u, M, v \rangle$ be the automaton obtained with the construction in Fig. 1.9 with a regular expression α. The automaton $\mathcal{A}$ is simple and the equation $u \cdot M^\star \cdot v \equiv \alpha$ can be derived from the rules of Kleene algebras.*

Proof. The proof goes by induction on the structure of the regular expression. In each case, we consider the product $u \cdot M \cdot v$, and use the law (‡) to compute the star of the matrix M which is triangular by blocks. For the base case, let $\alpha \in \{a/\epsilon/\emptyset\}$, we have:

$$\begin{bmatrix} 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 & \alpha \\ 0 & 0 \end{bmatrix}^\star \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} \equiv \begin{bmatrix} 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & \alpha \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$
$$\equiv \alpha.$$

Figure 1.9 Thompson's construction

(a) Base cases

$$\begin{bmatrix} 1 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & a/\epsilon/\emptyset \\ 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

(b) Plus

$$\left[\begin{array}{c|c} i_A & i_B \end{array}\right] \qquad \left[\begin{array}{c|c} A & 0 \\ \hline 0 & B \end{array}\right] \qquad \left[\begin{array}{c} f_A \\ \hline f_B \end{array}\right]$$

(c) Product

$$\left[\begin{array}{c|c} i_A & 0 \end{array}\right] \qquad \left[\begin{array}{c|c} A & f_A \cdot i_B \\ \hline 0 & B \end{array}\right] \qquad \left[\begin{array}{c} 0 \\ \hline f_B \end{array}\right]$$

(d) Star

$$\left[\begin{array}{c|c} 1 & i_A \end{array}\right] \qquad \left[\begin{array}{c|c} 1 & 0 \\ \hline 0 & A + f_A \cdot i_A \end{array}\right] \qquad \left[\begin{array}{c} 1 \\ \hline f_A \end{array}\right]$$

The other constructions are obtained in a very similar way. By induction hypothesis, we have $\alpha \equiv i_A \cdot A^\star \cdot f_A$ and $\beta \equiv i_B \cdot B^\star \cdot f_B$. We show that the automaton depicted in Fig. 1.9(b) evaluates to $\alpha + \beta$:

$$\begin{aligned} \left[\begin{array}{c|c} i_A & i_B \end{array}\right] \cdot \left[\begin{array}{c|c} A & 0 \\ \hline 0 & B \end{array}\right]^\star \cdot \left[\begin{array}{c} f_A \\ \hline f_B \end{array}\right] &\equiv \left[\begin{array}{c|c} i_A & i_B \end{array}\right] \cdot \left[\begin{array}{c|c} A^\star & 0 \\ \hline 0 & B^\star \end{array}\right] \cdot \left[\begin{array}{c} f_A \\ \hline f_B \end{array}\right] \\ &\equiv i_A \cdot A^\star \cdot f_A + i_B \cdot B^\star \cdot f_B \\ &\equiv \alpha + \beta. \end{aligned}$$

Similarly, the automaton shown in Fig. 1.9(c) evaluates to $\alpha \cdot \beta$:

$$\begin{aligned} \left[\begin{array}{c|c} i_A & 0 \end{array}\right] \cdot \left[\begin{array}{c|c} A & f_A \cdot i_B \\ \hline 0 & B \end{array}\right]^\star \cdot \left[\begin{array}{c} 0 \\ \hline f_B \end{array}\right] &\equiv \left[\begin{array}{c|c} i_A & 0 \end{array}\right] \cdot \left[\begin{array}{c|c} A^\star & A^\star \cdot f_A \cdot i_B \cdot B^\star \\ \hline 0 & B^\star \end{array}\right] \cdot \left[\begin{array}{c} 0 \\ \hline f_B \end{array}\right] \\ &\equiv i_A \cdot A^\star \cdot f_A \cdot i_B \cdot B^\star \cdot f_B \\ &\equiv \alpha \cdot \beta. \end{aligned}$$

Finally, the automaton shown in Fig. 1.9(d) evaluates to $\alpha^\star$:

$$\begin{aligned} \left[\begin{array}{c|c} 1 & i_A \end{array}\right] \cdot \left[\begin{array}{c|c} 1 & 0 \\ \hline 0 & A + f_A \cdot i_A \end{array}\right]^\star \cdot \left[\begin{array}{c} 1 \\ \hline f_A \end{array}\right] &\equiv \left[\begin{array}{c|c} 1 & i_A \end{array}\right] \cdot \left[\begin{array}{c|c} 1 & 0 \\ \hline 0 & (A + f_A \cdot i_A)^\star \end{array}\right] \cdot \left[\begin{array}{c} 1 \\ \hline f_A \end{array}\right] \\ &\equiv 1 + i_A \cdot (A + f_A \cdot i_A)^\star \cdot f_A \\ &\equiv 1 + i_A \cdot A^\star \cdot (f_A \cdot i_A \cdot A^\star)^\star \cdot f_A \quad \text{(using Thm. 6)} \\ &\equiv 1 + i_A \cdot A^\star \cdot f_A \cdot (i_A \cdot A^\star \cdot f_A)^\star \quad \text{(using Thm. 7)} \\ &\equiv 1 + \alpha \cdot \alpha^\star. \end{aligned}$$

Remark that all the above matrices are simple by construction. □

Removing ϵ-transitions. The automata $\mathcal{A} = \langle u, M, v \rangle$ obtained with Thompson's construction may contain ϵ-transitions: we have to get rid of these. Since the above construction builds simple automata, the transition matrices can be written as

$$M = J + M' \qquad \text{with} \quad M' = \sum_{a \in \Sigma} a \cdot M_a$$

where J and the M_a are 0-1 matrices, and J corresponds to the graph of ϵ-transitions. As we explained in §1.1, the first step of the removal of ϵ-transitions requires to compute their reflexive and transitive closure. Then, the set of initial states and the transition relation are updated. The algebraic counterpart of the computation of the reflexive and transitive closure of the ϵ-transitions is the computation of the matrix $J^\star$, as we hinted at in §1.2.1. Therefore, we build the following ϵ-free matricial automaton $\mathcal{A} = \langle u \cdot J^\star, M' \cdot J^\star, v \rangle$, and we prove that it evaluates to the same regular expression as $\mathcal{A}$. Algebraically, we have:

$$\begin{aligned} u \cdot \left(J + M'\right)^\star \cdot v &\equiv u \cdot J^\star \cdot \left(M' \cdot J^\star\right)^\star \cdot v && \text{(cf Thm. 6)} \\ &\equiv (u \cdot J^\star) \cdot \left(M' \cdot J^\star\right)^\star \cdot v. \end{aligned}$$

Determinisation. We now turn to the algebraic analogue to the subset construction. The goal here is to build a DFA $\langle \widehat{u}, \widehat{M}, \widehat{v} \rangle$ whose states correspond to set of states from an initial NFA $\langle u, M, v \rangle$ with states Q.

Let $\mathcal{P}(Q)$ denote the power set of Q. An element s of $\mathcal{P}(Q)$ can be identified with its characteristic vector e_s in $\{0,1\}^Q$. We let X denote the 0-1 matrix of size $|\mathcal{P}(Q)| \times |Q|$ defined by:

$$X_{s,j} \triangleq \begin{cases} 1 & \text{if } j \in s \\ 0 & \text{otherwise} \end{cases}$$

The intuition behind X is that this is a "decoding matrix": it sends characteristic vectors of states of the DFA to the corresponding subset of states of the NFA. By definition, we know that:

$$M = \sum_{a \in \Sigma} a \cdot M_a$$

For each $a \in \Sigma$, let $\widehat{M}_a$ be the $|\mathcal{P}(Q)| \times |\mathcal{P}(Q)|$ matrix that sends a set of states s to the set of the images of the elements of s. We remark that the vector $e_{s \cdot M_a}$ corresponds to the set of the images of s by transitions labelled with a. Therefore, the row s of $\widehat{M}_a$ is defined as:

$$e_s \cdot \widehat{M}_a \triangleq e_{s \cdot M_a}$$

Using the above definition, for each a in Σ, the matrix $\widehat{M}_a$ has exactly one 1 on each line. Then, let

$$\widehat{M} \triangleq \sum_{a \in \Sigma} a \cdot \widehat{M}_a \qquad \widehat{u} \triangleq e_u \qquad \widehat{v} \triangleq X \cdot v$$

Theorem 11. *The automaton $\langle \widehat{u}, \widehat{M}, \widehat{v} \rangle$ is simple, deterministic, and evaluates to the same regular expression as $\langle u, M, v \rangle$.*

Proof. By construction, the automaton is deterministic: $\widehat{u}$ and each line of $\widehat{M}_a$ contains exactly one 1. Moreover, the following commutation properties hold:

$$\widehat{M} \cdot X \equiv X \cdot M \quad (1) \qquad \widehat{u} \cdot X \equiv u \quad (2) \qquad \widehat{v} \equiv X \cdot v \quad (3)$$

If we interpret X as a decoding marix, (1) can be read as follows: executing a transition in the DFA and decoding the result amounts to decoding the given state, and executing parallel transitions in the NFA. Similarly, (2) proves that the initial state of the DFA corresponds to the set of initial states of the NFA. Finally, (3) states that the final (set of) states of the DFA

correspond to (set of) states that contain at least one final state of the NFA. By Theorem 5, we deduce $\widehat{M^\star} \cdot X \equiv X \cdot M^\star$ from (1). Then, we conclude with (2,3):

$$\begin{aligned}\widehat{u} \cdot \widehat{M^\star} \cdot \widehat{v} &\equiv \widehat{u} \cdot \widehat{M^\star} \cdot X \cdot v \\ &\equiv \widehat{u} \cdot X \cdot M^\star \cdot v \\ &\equiv u \cdot M^\star \cdot v\end{aligned}$$

□

Accessible subset construction. Here, we consider a refinement of the above construction to yield a DFA such that all states are accessible, using the standard depth-first enumeration of the accessible subsets we presented in §1.1. Remember, starting with a NFA with n states $\mathcal{N} = \langle [1...n], T, I, F \rangle$ that corresponds to the matricial automaton $\langle u, M, v \rangle$, the accessible subset construction returns a DFA $\mathcal{D}$ with $\widehat{n}$ states, together with an injective function ρ from $[1...\widehat{n}]$ to subsets of $[1...n]$. We let X be the rectangular $(\widehat{n}, n)$ 0-1 matrix defined by:

$$X_{s,j} \triangleq \begin{cases} 1 & \text{if } j \in \rho(s) \\ 0 & \text{otherwise} \end{cases}$$

For each $a \in \Sigma$, let $\widehat{M_a}$ be the $\widehat{n} \times \widehat{n}$ matrix that sends an accessible set of state s to the set of the images of the elements of s. By analogy with the previous construction, the row s of $\widehat{M_a}$ is defined as:

$$e_s \cdot \widehat{M_a} \triangleq e_{(\rho(s) \cdot M_a)}$$

By analysis of the algorithm, it is possible to prove commutation properties that correspond to the above (1,2,3), and that the matricial DFA $\langle \widehat{u}, \widehat{A}, \widehat{v} \rangle$ built using the accessible subset construction evaluates like the starting matricial NFA.

Note that there is a slight difference in spirit between the algebraic version of accessible subset construction and the algebraic version of the "normal" subset construction: the former construction actually relies on an external algorithm to compute the DFA and the bijection ρ. By contrast, the latter construction is only a matter of matrix construction. In the following sections, when we implement the decision procedure for Kleene algebras in Coq, we will put a strong emphasis on this: we shall see how we reflect efficient automata constructions to proofs that the evaluation of the automata are preserved.

Checking equivalence. We now proceed to the equivalence check. In his seminal paper, Kozen used an algebraic analogue of automata minimisation to check equivalence of DFAs. Here, we use a slightly different method, relying on the computation of a 2-simulation relation between DFAs: we give an algebraic analogue to this equivalence test, and shows that it suffices to conclude Kozen's proof. This is a minor deviation from Kozen's proof, but does not appear in his work, thus we will give more details than in previous steps.

Given two DFAs $\mathcal{A} = \langle i_A, A, f_A \rangle$ and $\mathcal{B} = \langle i_B, B, f_B \rangle$, we start by computing the automaton $\langle u, M, v \rangle$ that corresponds to their disjoint union (using the "plus" construction of Fig. 1.9). Then, we use an external algorithm to compute a 2-simulation relation $\approx$ that relates the former initial states $\overline{i_A}$ and $\overline{i_B}$. (Note that here, we use a 2-simulation relation that relates states of the disjoint union automaton, rather than states in two different automata. However, it is clear that there exists a 2-simulation that relates the former initial states in the disjoint union if and only if there exists a 2-simulation that relates the initial states of the original DFAs.) This

relation is then used to define a square 0-1 matrix that encodes the equivalence relation on states obtained with a successful run of the algorithm:

$$Y_{i,j} \triangleq \begin{cases} 1 & \text{if } i \approx j \\ 0 & \text{otherwise} \end{cases}$$

This matrix satisfies the following properties:

$$1 \leq Y \quad (1) \qquad Y \cdot Y \leq Y \quad (2) \qquad Y \cdot M \leq M \cdot Y \quad (3)$$

$$\overline{i_A} \cdot Y \equiv \overline{i_B} \cdot Y \quad (4) \qquad Y \cdot v \equiv v \quad (5)$$

Equations $(1, 2)$ correspond to the fact that Y encodes a reflexive and transitive relation. Equation (3) comes from the fact that Y is a simulation: transitions starting from related states yield related states. The last two equations assess that the starting states are related (4), and that related states are either accepting or non-accepting (5).

Proof of equation (3). This is the most interesting case. If we unfold the matricial product, we have to check that the following inequation holds for any states i and k :

$$\sum_j Y_{ij} \cdot M_{jk} \leq \sum_r M_{ir} \cdot Y_{rk} \ .$$

It suffices to check[9] that, for any state j, we have:

$$Y_{ij} \cdot M_{jk} \leq \sum_r M_{ir} \cdot Y_{rk} \ .$$

If $i \approx j$ does not hold, the left-hand side is 0, and the inequation holds. Otherwise, we have to check that:

$$i \approx j \vdash M_{jk} \leq \sum_r M_{ir} \cdot Y_{rk} \ .$$

By unfolding the definition of M, this inequation must hold for each letter $a \in \Sigma$:

$$i \approx j \vdash (M_a)_{jk} \leq \sum_r (M_a)_{ir} \cdot Y_{rk}$$

If $\delta(j, a) \neq k$, the left-hand side is 0, and the inequation holds trivially. Otherwise, we have to check that:

$$i \approx j, \delta(j, a) = k \vdash 1 \leq \sum_r (M_a)_{ir} \cdot Y_{rk}$$

It suffices to prove that $1 \leq (M_a)_{ir} \cdot Y_{rk}$ when $r = \delta(i, a)$. That is, to check the left-hand side statement (or, the corresponding right-hand side diagram):

$$i \approx j, \delta(j, a) = k, r = \delta(i, a) \vdash 1 \leq Y_{rk} \qquad \begin{array}{rcl} i & \longrightarrow & \delta(i, a) \\ \approx \big| & & \big| \approx ? \\ j & \longrightarrow & \delta(j, a) \end{array}$$

Since the relation $\approx$ is a 2-simulation, the above property holds. □

[9] Recall that $a \leq b \triangleq a + b \equiv b$, and hence, if $a \leq c$ and $b \leq c$ hold, then $a + b \leq c$ hold.

These properties allow us to conclude using algebraic reasoning: from $(1, 2, 3)$ and Kleene algebra laws, we deduce

$$M^\star \cdot Y \equiv Y \cdot (M \cdot Y)^\star \ . \tag{6}$$

Correctness follows:

$$\begin{aligned}
\overline{i_A} \cdot M^\star \cdot v &\equiv \overline{i_A} \cdot M^\star \cdot Y \cdot v && \text{(by 5)} \\
&\equiv \overline{i_A} \cdot Y \cdot (M \cdot Y)^\star \cdot v && \text{(by 6)} \\
&\equiv \overline{i_B} \cdot Y \cdot (M \cdot Y)^\star \cdot v && \text{(by 4)} \\
&\equiv \overline{i_B} \cdot M^\star \cdot Y \cdot v && \text{(by 6)} \\
&\equiv \overline{i_B} \cdot M^\star \cdot v \ . && \text{(by 5)}
\end{aligned}$$

Then, we remark that:

$$\begin{aligned}
i_A \cdot A^\star \cdot f_A &\equiv \overline{i_A} \cdot M^\star \cdot v \\
&\equiv (\overline{i_A} + \overline{i_B}) \cdot M^\star \cdot v && \text{(by idempotence)} \\
&\equiv i_A \cdot A^\star \cdot f_A + i_B \cdot B \cdot f_B.
\end{aligned}$$

Using a symmetric argument, we have:

$$i_B \cdot B^\star \cdot f_B \equiv i_A \cdot A^\star \cdot f_A + i_B \cdot B^\star \cdot f_B.$$

Therefore, we conclude:

$$i_A \cdot A^\star \cdot f_A \equiv i_B \cdot B^\star \cdot f_B.$$

In other words, if two (matricial) DFAs are equivalent, then we obtain a proof that the evaluation of these automata are equal modulo the rules of Kleene algebras. That is, we obtained the bottom line equality of Fig. 1.8.

1.3.5 Conclusion

We can now conclude Kozen's proof. By combining the proofs from the above paragraphs according to Fig. 1.8, we prove that if two regular expressions are equivalent, then the corresponding equality can be derived from the rules of Kleene algebras. That is, if α and β denote the same language, then the equation $\alpha \equiv \beta$ is derivable: the rules of Kleene algebras from 1.6 are complete for the algebra of regular events.

Theorem 12 (Completeness)**.** *Let α and β be two regular expressions over Σ that denote the same language, i.e., $\alpha \sim \beta$. Then $\alpha \equiv \beta$ holds.*

Proof. Let $\mathcal{A} = \langle i_A, A, f_A \rangle$ and $\mathcal{B} = \langle i_B, B, f_B \rangle$ be deterministic finite automata over Σ such that

$$\alpha \equiv i_A \cdot A^\star \cdot f_A \qquad\qquad \beta \equiv i_B \cdot B^\star \cdot f_B$$

The DFAs $\mathcal{A}$ and $\mathcal{B}$ must be equivalent, otherwise, the languages denoted by α and β would not be equal. Therefore, we can apply the construction from the previous paragraph, and obtain:

$$i_A \cdot A^\star \cdot f_A \equiv i_B \cdot B^\star \cdot f_B.$$

Hence, we have derived:

$$\alpha \equiv \beta.$$

□

Intermezzo

From Kozen's theorem, we define a reflexive tactic. This means that we implement a decision procedure for equivalence of regular expressions as a Coq program, and prove its correctness and completeness within the proof assistant:

```
Definition decide_kleene: regex → regex → bool := ...
Theorem Kozen94: ∀ a b, decide_kleene a b = true ↔ a ≡ b.
```

The above statement corresponds to correctness and completeness with respect to the syntactic "free" Kleene algebra: `regex` is the inductive Coq type for regular expressions over a given set of variables, and $\equiv$ is the inductive equality predicate generated by the axioms of Kleene algebras and the rules of equational reasoning (see Fig. 1.10). This decision procedure on the syntactic free Kleene algebra can be lifted to other models of Kleene algebras using reification mechanisms: in particular, this decision procedure can be used to decide equations that involve heterogeneous binary relations, homogeneous binary relations, or, more generally, every typed or untyped model of Kleene algebra we mentioned in §1.2.1. (See §1.4.6 for more details about our reification mechanism.)

Moving to Coq, some design choices must be made in order to implement the aforementioned decision procedure and prove it correct and complete. Indeed, there is a fair amount of preliminary work and definitions that must be done to be able to proceed to the end. Here we summarise the salient points of the previous section as the requirements we had to take into account for the design of the underlying library.

Efficiency. The equational theory of Kleene algebras is PSPACE-complete [113]: this means that the decision procedure must be written with care in Coq, using efficient out-of-the-shelf algorithms. Notably, the matricial representation of automata is not efficient enough[10], so that formalising Kozen's mathematical proof (from the previous §1.3) would not scale computationally. Instead, we need to choose appropriate data-structures for automata and algorithms, and to reflect these data-structures in matricial automata representation only in proofs, using adequate translation functions.

Matrices. As explained in §1.3, Kozen's proof relies on the theory of matrices over regular expressions, which we thus need to formalise. As the later proof requires a lot of matricial reasoning, the formalisation of matrices must be easy to use from the proof point of view. Yet, we need to formalise the (slightly more complicated) theory of non-square matrices over a Kleene algebra to handle Kozen's proof. In fact, these two requirements can be handled together thanks to the generalisation to typed Kleene algebras: while only square matrices form a model of Kleene algebra, rectangular matrices form a model of typed Kleene algebras. This makes it possible to internalise vectors as a special case of rectangular matrices: this saves us from re-defining their theory separately, and from offsprings like special functions to handle various products between a vector and a matrix.

Modular development. Following mathematical and programming practices, we aim at a modular development. That is, we need to share notations, theorems and tactics as much as possible between structures to improve readibility, usability, and maintainability.

[10] Note that the first release of our decision procedure was made using algorithms working on such matricial automata.

Figure 1.10 Regular expressions, axiomatic equality

```
Inductive regex: Set :=
| dot:  regex → regex → regex
| plus: regex → regex → regex
| star: regex → regex
| one:  regex
| zero: regex
| var:  positive → regex.

Notation "x · y" := (dot x y).
Notation "x + y" := (plus x y).
Notation "x*" := (star x).
Notation "1" := (one).
Notation "0" := (zero).

Inductive eq: regex → regex → Prop :=
| eq_trans: Transitive eq
| eq_sym: Symmetric eq
| eq_refl: Reflexive eq

| dot_compat: Proper (eq ⇒ eq  ⇒ eq) dot
| plus_compat: Proper (eq ⇒ eq ⇒ eq) plus
| star_compat: Proper (eq ⇒ eq ⇒ er) star.

| dot_assoc: ∀ x y z, eq (x·(y·z)) ((x·y)·z)
| dot_neutral_left: ∀ x, eq (1·x) x
| dot_neutral_right: ∀ x, eq (x·1) x

| plus_neutral_left: ∀ x, eq (0+x) x
| plus_idem: ∀ x, eq (x+x) x
| plus_assoc: ∀ x y z, eq (x+(y+z)) ((x+y)+z)
| plus_com: ∀ x y, eq (x+y) (y+x)

| dot_ann_left: ∀ x, eq (0·x) 0
| dot_ann_right: ∀ x, eq (x·0) 0
| dot_distr_left: ∀ x y z,
    eq ((x+y)·z) ((x·z)+(y·z))
| dot_distr_right: ∀ x y z,
    eq (x·(y+z)) ((x·y)+(x·z))

| star_make_left: ∀ x, eq  (1+x*·x) (x*)
| star_make_right: ∀ x, eq (1+x·x*) (x*)
| star_destruct_left: ∀ x y,
    eq (x·y+y) y → eq (x*·y+y) y
| star_destruct_right: ∀ x y,
    eq (y·x+y) y → eq (y·x*+y) y
```

1.4 Underlying design choices

According to the above constraints and objectives, a central decision was to build on the introduction of type-classes in Coq [147]. This section is devoted to the explanation of our methodology: how to use type-classes to define the algebraic hierarchy in a modular way, how to formalise typed algebras, how to reify the corresponding expressions.

1.4.1 Using type-classes to structure the development

We use type-classes to achieve two tasks: 1) sharing and overloading notation, basic laws, and theorems; 2) getting a modular definition of Kleene algebra, by mimicking the standard mathematical hierarchy: a Kleene algebra contains an idempotent semiring, which is itself composed of a monoid and a semi-lattice. This very small hierarchy is summarised below.

```
SemiLattice <:
  Monoid    <:  IdemSemiRing <: KleeneAlgebra
```

A peculiarity of our implementation of this hierarchy is that we want to work with the typed versions of the above algebraic structures, to handle heterogeneous binary relations in the end, as well as rectangular matrices in the proof of Kozen's initiality theorem. Thus, we will use type-classes reminiscent of the typed signature that was hinted at in Fig. 1.7.

1.4.2 Separation between operations and laws

The problem of formalising mathematical structures or algebraic hierarchies in type theory is well-known and usually considered as difficult [13, 24, 49, 67, 68]. One of the key difference in solutions to this problem is the amount of *bundling* emphasised by a given solution: to rephrase it, when defining the classes for algebraic structures, the question is whether or not to package operations together with their laws. At one end of the spectrum lies the *fully unbundled approach*, where each component of a "structure" is passed as a separate argument to function and theorems. At the other end of the spectrum, existing solutions encompass nested dependent records (coined as *telescopes* [39]) and the more refined (and more recent) *packed classes* [67].

There are several issues at hand with the use of dependent records: how to share an operation between various algebraic structures (structures with identical operations can differ by axioms); how to deal with the so-called multiple inheritance problem; how to deal with the long projections paths that appears if we stack up algebraic structures (accessing deeply nested components has an increasing cost).

There are other issues at hand with the use of the fully unbundled approach: it induces a proliferation of arguments for theorem and definitions, which increases dramatically the size of the terms, making theorem proving impractical. For instance, in the proof of the Cayley-Hamilton theorem [67], one need to consider the "ring of polynomials over the ring of matrices over a general commutative ring", which would yield an exponential blowup, would the fully unbundled approach be used.

We actually split the two: we separated the operational contents from the proof contents. It makes it possible to define structures sharing the same operations (hence notations), but not necessarily the same laws. It also solves the "diamond problem": shared operations between algebraic structures (the proof contents) are not spuriously distinguished by the projection-paths that access them. However, this is a risky move, because of the efficiency issues we mentioned. Since we basically stop at Kleene algebra, this choice is not critical for the library in its current state. However, based on preliminary experiments, having this separation is problematic when considering richer structures like residuated Kleene lattices [92] or allegories [66].

Figure 1.11 Classes for the typed algebraic operations.

```
Class Graph := {
 T: Type;
 X: T → T → Type;
 equal: ∀ n m, relation (X n m);
 equal_:> ∀ n m, Equivalence (equal n m) }.

Class Monoid_Ops (G: Graph) := {
 dot: ∀ n m p, X n m → X m p → X n p;
 one: ∀ n,     X n n }.

Notation "x ≡ y"  := (equal _ _ x y).
Notation "x · y"  := (dot _ _ _ x y).
Notation "1"      := (one _).
```

```
Class SemiLattice_Ops (G: Graph) := {
 plus: ∀ n m, X n m → X n m → X n m;
 zero: ∀ n m, X n m;
 leq:  ∀ n m, relation (X n m) :=
       fun n m x y ⇒ plus n m x y ≡ y }.

Class Star_Op (G: Graph) := {
 star: ∀ n, X n n → X n n }.

Notation "x ⊆ y" := (leq _ _ x y).
Notation "x + y"  := (plus _ _ x y).
Notation "0"      := (zero _ _).
Notation "x*"     := (star _ x).
```

1.4.3 Classes for algebraic operations.

We associate an intuitive notation to each operation, by using the name provided by the corresponding class projection. To make the effect of these definitions completely clear, assume that we have a graph equipped with monoid operations (i.e., a typing context with `G: Graph` and `Mo: Monoid_Ops G`) and consider the following proposition:

```
∀ (n m: T) (x: X n m) (y: X m n), x·y ≡ 1.
```

If we unfold notations, we get:

```
∀ (n m: T) (x: X n m) (y: X m n), equal _ _ (dot _ _ _ x y) (one _).
```

Necessarily, by unification, the six place-holders have to be filled as follows:

```
∀ (n m: T) (x: X n m) (y: X m n), equal n n (dot n m n x y) (one n).
```

Now comes type-class resolution: the functions `T`, `X`, `equal`, `dot`, and `one`, which are class projections, have implicit arguments that are automatically filled by type-class resolution (the graph instance for all of them, and the monoid operations instance for `dot` and `one`). All in all, the above concise proposition actually expands into:

```
∀ (n m: @T G) (x: @X G n m) (y: @X G m n), @equal G n n (@dot G Mo n m n x y) (@one G Mo n).
```

1.4.4 Classes for algebraic laws.

We now move away from the syntax to the actual algebraic structures: the type-classes that package the laws for the corresponding four algebraic structures we are interested in. They are given in Fig. 1.12; we use the section mechanism to assume a graph together with the operations, which become parameters when we close the section.

The `Monoid` class actually corresponds to the definition of a category: we assume that composition (`dot`) is associative and has `one` as neutral element. Its first field, `dot_compat`, requires that composition also preserves the user-defined equality: it has to map equals to equals. (This field is declared with a special symbol (`:>`) and uses the standard `Proper` class, which is exploited by Coq to perform rewriting with user-defined relations; doing so adds `dot_compat` as a hint for type-class resolution, so that we can automatically rewrite in `dot` operands whenever it makes sense.) Also note that since this class does not mention semi-lattice operations nor the star operation, it does not depend on `SLo` and `Ko` when we close the section. We do not comment on the `SemiLattice` class, which is quite similar.

Figure 1.12 Classes for the typed algebraic structures.

```
Section.
 Context (G: Graph) {Mo: Monoid_Ops G} {SLo: SemiLattice_Ops G} {Ko: Star_Op G}.

 Class Monoid := {
   dot_compat:> ∀ n m p, Proper (equal n m ⇒ equal m p ⇒ equal n p) (dot n m p);
   dot_assoc: ∀ n m p q (x: X n m) (y: X m p) (z: X p q), x·(y·z) ≡ (x·y)·z;
   dot_neutral_left: ∀ n m (x: X n m), 1·x ≡ x;
   dot_neutral_right: ∀ n m (x: X m n), x·1 ≡ x }.

 Class SemiLattice := {
   plus_compat:> ∀ n m, Proper (equal n m ⇒ equal n m ⇒ equal n m) (plus n m);
   plus_neutral_left: ∀ n m (x: X n m), 0+x ≡ x;
   plus_idem: ∀ n m (x: X n m), x+x ≡ x;
   plus_assoc: ∀ n m (x y z: X n m), x+(y+z) ≡ (x+y)+z;
   plus_com: ∀ n m (x y: X n m), x+y ≡ y+x }.

 Class IdemSemiRing := {
   Monoid_:> Monoid;
   SemiLattice_:> SemiLattice;
   dot_ann_left: ∀ n m p (x: X m p), (0 : X n m) · x ≡ 0;
   dot_ann_right: ∀ n m p (x: X p m), x · (0 : X m n) ≡ 0;
   dot_distr_left: ∀ n m p (x y: X n m) (z: X m p), (x+y)·z ≡ x·z + y·z;
   dot_distr_right: ∀ n m p (x y: X m n) (z: X p m), z·(x+y) ≡ z·x + z·y }.

 Class KleeneAlgebra := {
   IdemSemiRing_:> IdemSemiRing;
   star_make_left: ∀ n (x: X n n), 1 + x^⋆·x ≡ x^⋆;
   star_destruct_left: ∀ n m (x: X n n) (y: X n m), x·y ⊆ y → x^⋆·y ⊆ y;
   star_destruct_right: ∀ n m (x: X n n) (y: X m n), y·x ⊆ y → y·x^⋆ ⊆ y }.
End.
```

The first two fields of `IdemSemiRing` implement the expected inheritance relationship: an idempotent semiring is composed of a monoid and a semi-lattice whose operations properly distribute. By declaring these two fields with a :>, the corresponding projections are added as hints to type-class resolution, so that one can automatically use any theorem about monoids or semi-lattices in the context of a semiring. Note that we have to use the textual version of 0 (`zero`) in two laws to explicitly give its domain and co-domain; indeed, in both cases, the argument `n` cannot be inferred from the context, it has to be specified.

Finally, we obtain the class for Kleene algebras by inheriting from `IdemSemiRing` and requiring the three laws about Kleene star to hold. The fact that Kleene star is a proper morphism for `equal` is a consequence of the other axioms; this is why we do not include a `star_compat` field in the signature.

The following example illustrates the ease of use of this approach. Here is how we would state and prove a lemma about idempotent semirings:

```
Goal ∀ `{IdemSemiRing} n (x y: X n n), x·(y+1)+x ≡ x·y+x.
Proof.
  intros; rewrite dot_distr_right, dot_neutral_right. (* (x·y+x)+x ≡ x·y+x *)
  rewrite ←plus_assoc, plus_idem; reflexivity.
Qed.
```

Figure 1.13 Instances for heterogeneous binary relations and languages.

```
Definition rel A B := A → B → Prop.
Instance rel_G: Graph := {
  T := Type;
  X := rel;
  equal A B R S := ∀ i j, R i j ↔ S i j }.
Proof...
Instance rel_Mo: Monoid_Ops rel_G := {
  dot A B C R S :=
    fun (i: A)(j: C) ⇒ ∃ k: B, R i k ∧ S k j;
  one A := fun (i j: A) ⇒ i=j }.
...
Instance rel_KA: KleeneAlgebra rel_G.
Proof...
```

```
Definition lang A := list A → Prop
Instance lang_G A: Graph := {
  T := unit;
  X _ _ := lang A;
  equal _ _ L K := ∀ w, L w ↔ K w }.
Proof...
Instance lang_Mo A: Monoid_Ops (lang_G A) := {
  dot _ _ _ L K :=
    fun w ⇒ ∃ u v, w=u·v ∧ L u ∧ K v;
  one _ := fun w ⇒ w=[] }.
...
Instance lang_KA: KleeneAlgebra lang_G.
Proof...
```

In the above example, the '{ `IdemSemiRing`} notation introduces and gives names to a a generic idempotent semiring as well as all its parameters (a graph, monoid operations, and semi-lattice operations); when we use lemmas like `dot_distr_right` or `plus_assoc`, type-class resolution automatically finds appropriate instances to fill their implicit arguments. Of course, since such simple and boring goals occur frequently in larger and more interesting proofs, we actually defined high-level tactics to solve them automatically. For example, we have a reflexive tactic called `semiring_reflexivity` which would solve this goal directly: this is the counterpart to `ring` [76] for the equational theory of typed, idempotent, non-commutative semirings.

1.4.5 Concrete structures

It remains to populate the above classes with concrete structures, i.e., to declare models of Kleene algebra. We sketched the case of heterogeneous binary relations and languages in Fig. 1.13. As expected, it suffices to define a graph equipped with the various operations, and to prove that they validate all the axioms. The situation is slightly peculiar for languages, which form an *untyped* model: although the instances are parametrised by a set `A` coding for the alphabet, there is no notion of domain/co-domain of a language. In fact, all operations are total, they actually lie in a one-object category where domain and co-domain are trivial. Accordingly, we use the singleton type `unit` for the index type `T` in the graph instance, and all operations just ignore the superfluous parameters.

1.4.6 Reification: handling typed models.

We also need to define a syntactic model in which to perform computations: since we define a reflexive tactic, the first step is to reify the goal (an equality between two expressions in an arbitrary model) to use a syntactical representation.

For instance, suppose that we have a goal of the form $\mathtt{S}\cdot(\mathtt{R}\cdot\mathtt{S})^\star + \mathtt{f}\ \mathtt{R} \equiv \mathtt{f}\ \mathtt{R} + (\mathtt{S}\cdot\mathtt{R})^\star\cdot\mathtt{S}$, where `R` and `S` are binary relations and `f` is an arbitrary function on relations. The usual methodology in Coq consists in defining a syntax and an evaluation function such that this goal can be converted into the following one:

$$\mathtt{eval}\ (\mathtt{var}\ 1\cdot(\mathtt{var}\ 2\cdot\mathtt{var}\ 1)^\star + \mathtt{var}\ 3) \equiv \mathtt{eval}\ (\mathtt{var}\ 3 + (\mathtt{var}\ 1\cdot\mathtt{var}\ 2)^\star\cdot\mathtt{var}\ 1),$$

where `eval` implicitly uses a reification environment, which corresponds to the assignment:

$$\{1 \mapsto \mathtt{S};\ 2 \mapsto \mathtt{R};\ 3 \mapsto \mathtt{f}\ \mathtt{R}\}.$$

Figure 1.14 Typed syntax for reification and evaluation function.

```
Context `{KA: KleeneAlgebra}.
Variables src, tgt: label → T.
Inductive reified: T → T → Type :=
| r_dot: ∀ n m p,
    reified n m → reified m p → reified n p
| r_one: ∀ n, reified n n
| ...
| r_var: ∀ i, reified (src i) (tgt i).

Variable env: ∀ i, X (src i) (tgt i).
Fixpoint eval n m (x: reified n m): X n m :=
  match x with
  | r_dot _ _ _ x y ⇒ eval x ·eval y
  | r_one _ ⇒ 1
  | ...
  | r_var i ⇒ env i
  end.
```

Typed syntax. The situation is slightly more involved here since we work with typed models: `R` might be a relation from a set `A` to another set `B`, `S` and `f R` being relations from `B` to `A`. As a consequence, we have to keep track of domain/co-domain information when we define the syntax and the reification environments. The corresponding definitions are given in Fig. 1.14. We assume an arbitrary Kleene algebra (in the previous example, it would be the algebra of heterogeneous binary relations) and two functions `src` and `tgt` associating a domain and a co-domain to each variable (`label` is an alias for `positive`, the type of positive numbers, which we use to index variables). The `reified` inductive type corresponds to the typed reification syntax: it has dependently typed constructors for all operations of Kleene algebras, and an additional constructor for variables, which is typed according to functions `src` and `tgt`. To define the evaluation function, we furthermore assume an assignation `env` from variables to elements of the Kleene algebra with domain and co-domain as specified by `src` and `tgt`.

Then it suffices to parse the goal, looking for type-class projections to detect projections of interest (recall for example that a starred sub-term is always of the form `@star _ _ _ _`, regardless of the current model—this model is given in the first two placeholders) to reify a goal to this typed syntax. At first, we implemented this step as a simple `Ltac` tactic. For efficiency and simplicity reasons, we finally moved to an OCaml implementation in a small plugin: this allows one to use efficient data structures like hash-tables to compute the reification environment, and to avoid type-checking the reified terms at each step of their construction. (Note that the latter point renders the trivial user-level reification implementable in `Ltac` unpractical for large goals, making the use of an OCaml plugin almost mandatory.)

Untyped regular expressions. To build a reflexive tactic using the above syntax, we need a theorem of the following form (keeping the reification environment implicit for the sake of readability):

```
Theorem f_correct: ∀ n m (x y: reified n m), f x y = true → eval x ≡ eval y.
```

The function `f` is the decision procedure; it works on reified terms so that its type has to be `∀ n m, reified n m → reified n m → bool`. However, as we explained in §1.3.1 it would be rather impractical to implement this decision procedure in the typed setting. Therefore, we rely on the untyping theorem we mentioned for Kleene algebras (and which is developed further in [129]), and build a decision procedure for the regular expressions `regex` from Fig. 1.10. From the computational point of view, this is the main model we shall work with. (Note however that we shall of course use typed models, especially matrices, in the proof part.) As announced, we will get:

```
Definition decide_kleene: regex → regex → bool := ...
Theorem Kozen94: ∀ x y: regex, decide_kleene x y = true ↔ x ≡ y.
```

(Here the symbol ≡ expands to the inductive equality predicate `eq` from Fig. 1.10.)

1.5 Matrices

In this section, we describe our implementation of matrices, building on the previously described algebraic framework. Matrices are indeed the crux of Kozen's initiality theorem.

1.5.1 Coq representation for matrices

We assume a Graph `G`: `Graph`, and an object `u`: `T`. We furthermore abbreviate the type `X u u` as `X`, and we shall see that this is without loss of generality for the constructions of interest in this proof. The type `X` is the type of elements of the matrices, sometimes called scalars. We shall now discuss two possible implementations of matrices over `X`.

Dependently typed presentation. A matrix can be seen as a partial map from pairs of integers to `X`. A Coq type for matrices and a sum operation could be defined as follows:

```
Definition MX (n m: nat) := ∀ i j, i<n → j<m → X.
Definition mx_equal n m (M N: MX n m) i j (Hi: i < n) (Hj: j < m) :=
    M i j Hi Hj ≡ N i j Hi Hj
Definition plus n m (M N: MX n m) i j (Hi: i<n) (Hj: j<m) :=
    M i j Hi Hj + N i j Hi Hj.
```

This corresponds to the dependent types approach: a matrix is a map to `X` from two integers and two proofs that these integers are lower than the bounds of the matrix. Except for the concrete representation, this is the approach followed in [23, 28, 67]. With such a type, every access to a matrix element is made by exhibiting two proofs, to ensure that indices lie within the bounds. This is not problematic for simple operations like the above `plus` function: it suffices to pass the proofs around. However, this requires more boilerplate for other functions, like block decomposition operations.

Infinite functions. We actually adopt another strategy: we move bounds checks to equality proofs, by working with the following definitions:

```
Definition MX n m := nat → nat → X.
Definition mx_equal n m (M N: MX n m) i j (Hi: i < n) (Hj: j < m) := M i j ≡ N i j
```

Here, a matrix is an infinite function from pairs of integers to `X`, and equality is restricted to the domain of the matrix. With these definitions, we do not need to manipulate proofs when defining matrix operations, so that subsequent definitions are easier to write. For instance, the functions for matrix multiplication is given in Fig. 1.15. For multiplication, we use a naive function to compute the appropriate sum: there is no need to provide an explicit proof that each call to the matrices is performed within the bounds.

Figure 1.15 Definition of matricial product and identity matrix.

```
Context {SLo: SemiLattice_Ops G}.
Fixpoint sum i k (f: nat → X) :=
  match k with
  | 0 ⇒ 0
  | S k ⇒ f i + sum (S i) k f
  end.

Context {Mo: Monoid_Ops G}.
Definition mx_dot n m p (M: MX n m) (N: MX m p) :=
  fun i j ⇒ sum 0 m (fun k ⇒ M i k · N k j).

Definition mx_one n: MX n n :=
  fun i j ⇒ if eq_nat_bool i j then 1 else 0.
```

Correctness. Actually, nothing prevents us from, e.g., accessing M outside its bounds, when extracting sub-blocks. However, if the definitions of the operations were wrong, we would not be able to prove some obvious properties about them. Indeed, harmony lemmas that relate different operations allows to get rid of wrong definitions: for instance, we can check that re-composing the extracted sub-blocks of a matrix is equal to the original matrix.

Bounds checks are required a posteriori only, when proving properties about these matrices operations, e.g., associativity of the product. These proofs are done within the interactive proof mode, so that they can be solved with high level tactics like `omega`. (Note that this separation between proofs and programs could also be achieved syntactically—even with a dependently typed definition of matrices—by using Coq's `Program` feature. We prefer our approach for its simplicity: `Program` tends to clutter the computational part of terms, and reasoning about them becomes un-easy.)

Although the correctness proof of our algorithm heavily relies on matricial reasoning, and in particular block matrix decompositions, we have not found major drawbacks to this approach yet. We actually believe that it would scale smoothly to even more intensive usages of matrices, e.g., linear algebra [70].

Phantom types. Unfortunately, these non-dependent definitions allow one to type the following code, where the three additional arguments of `dot` are implicit:

```
Definition ill_dot n p (M: MX n 16) (N: MX 64 p): MX n p := dot M N.
```

This definition is accepted because of the conversion rule: since the dependent type `MX n m` does not mention `n` nor `m` in its body, these type informations can be discarded by the type system using the conversion rule (we actually have `MX n 16 = MX n 64`). While such an ill-formed definition will be detected at proof-time, it is a bit sad not to benefit from the advantages of a strongly typed programming language here to get rid of this bogus definition at once. We solved this problem at the cost of some syntactic sugar, by resorting to an inductive singleton definition, reifying bounds in *phantom types*:

```
Inductive MX (n m: nat) := box: (nat → nat → X) → MX n m.
Definition get (n m: nat) (M: MX n m) := match M with box f ⇒ f end.
Definition plus (n m: nat) (M N: MX n m) := box n m (fun i j ⇒ get M i j + get N i j).
```

Coq no longer equates types `MX n 16` and `MX n 64` with this definition, so that the above `ill_dot` function is rejected, and we can trust inferred implicit arguments (e.g., the `m` argument of `dot`). From a practical point of view, this makes the code dealing with matrics a lot easier to debug.

(Note that `get` can be declared as a coercion, to reduce the syntactic overhead. Moreover, once the development finished, we could actually drop the inductive! This would actually result in a slight improvement in performance – removing the superfluous ι reductions that occur when `get` is applied to a matrix.)

Computation. We also advocate this lightweight representation from the efficiency point of view. First, using non-dependent types is more efficient: not a single boundary proof gets evaluated in matrix computations (e.g., matrix multiplications). Second, using functions to represent matrices is two-edged: on the one hand, if the matrix resulting of a computation is seldom used, then computing its elements by need is efficient; on the other hand, making numerous accesses to the same expensive computation may be a burden. To this end, we defined a *memoisation* operator that computes all elements of a given matrix, stores the results in a map, and returns the closure that looks up in the map rather than recomputing the result. This memoisation operator is proved to be an identity; it can be inserted in matrix computations in a transparent way, at judicious places.

```
Definition mx_force n m (M: MX n m): MX n m :=
  let l := mx_to_lists M in box n m (fun i j ⇒ nth i (nth j l)).
Lemma mx_force_id : ∀ n m (M : MX n m), mx_force M ≡ M.
```

The first version of our decision procedure used such matrix computations for the construction of automata using Thompson's algorithm and the removal of ϵ-transition (which amounts to the computation of the star of a 0-1 matrix). While we have later chosen to avoid such matrix computations to be more efficient, there may be some interesting way to improve efficiency while keeping this simple interface. For instance, we could consider using binary indexing and more efficient data-structure for memoisation, or to have specialised constructors and operations for sparse matrices.

Dénès and Bertot recently argued in a similar fashion that using "untyped" representations in computations yields better performance. They give computational and efficient counterparts to the algebraic operations on matrices provided in the `ssreflect` library (in which matrices are defined as dependently-typed finite functions). While their current untyped representation is based on lists of lists, they plan to rely on the forthcoming addition of native persistent-arrays and native integers [9] in the logic of Coq to improve the overall performances. (Note that our definition of matrices is actually agnostic from the point of view of the underlying representation, and that this makes it possible to tailor the underlying data-structure—be it a function, lists of lists, or a representation of sparse matrices—to the actual problem at hand.)

1.5.2 Lifting the algebraic hierarchy

As expected, we declare the previous operations as new instances, so that we can directly use notations, lemmas, and tactics with matrices. The type of these instances are given below:

```
Instance mx_G: Graph := { T := nat; X := MX; equal := mx_equal }.

Instance mx_SLo: SemiLattice_Ops G → SemiLattice_Ops mx_G.
Instance mx_Mo:  SemiLattice_Ops G → Monoid_Ops G → Monoid_Ops mx_G.

Instance mx_SL:  '{SemiLattice G} → SemiLattice mx_G.
Instance mx_ISR: '{IdemSemiRing G} → IdemSemiRing mx_G.
```

Thus, starting with an arbitrary (typed) idempotent semiring of elements `X`, we can build the (typed) idempotent semiring of rectangular matrices over these elements.

1.5.3 Computing the star of a matrix

Using the previous instances to get notations, lemmas, and tactics about matrices, we conclude this section with the Kleene star operation. The type of these instances are given below:

```
Instance mx_Ko:  SemiLattice_Ops G → Monoid_Ops G → Star_Op G → Star_Op mx_G.
Instance mx_KA:  '{KleeneAlgebra G} → KleeneAlgebra mx_G.
```

We have to define the star operation on matrices, and show that it satisfies the laws for Kleene star. We conclude this section about matrices by a brief description of this construction in Coq—see §1.3.2 for more details.

Recall that the construction proceeds by induction on the size of the matrix: the problem is trivial if the matrix is empty or of size 1×1; otherwise, we decompose the matrix into four blocks and we recurse as follows [1]:

$$\left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right]^\star = \left[\begin{array}{c|c} E & E \cdot B \cdot D' \\ \hline D' \cdot C \cdot E & D' + D' \cdot C \cdot E \cdot B \cdot D' \end{array}\right] \quad \text{where} \begin{cases} D' &= D^\star \\ E &= (A + B \cdot D^\star \cdot C)^\star \end{cases} \qquad (\dagger)$$

Figure 1.16 Definition of the star operation on matrices.

```
Definition mx_star' x n
  (sx: MX x x → MX x x)
  (sn: MX n n → MX n n)
  (M: MX (x+n) (x+n)): MX (x+n) (x+n) :=
 let A := mx_sub00 M in
 let B := mx_sub01 M in
 let C := mx_sub10 M in
 let D := mx_sub11 M in
 let D' := sn D in
 let E := sx (A+B·D'·C) in
   mx_blocks
        E          (E·B·D')
    (D'·C·E) (D'+D'·C·E·B·D').
```

```
Definition mx_star_11 (M: MX 1 1): MX 1 1 :=
  fun _ _ ⇒ (M 0 0)*.

Fixpoint mx_star n: MX n n → MX n n :=
  match n with
  | 0 ⇒ fun M ⇒ M
  | S n ⇒ mx_star' mx_star_11 (mx_star n)
  end.

Theorem mx_star_block x n (M: MX (x+n) (x+n)):
  mx_star (x+n) M ≡
    mx_star' (mx_star x) (mx_star n) M.
Proof...
```

As long as we take square matrices for A and D, the way we decompose the matrix does not matter (we actually have to prove it). In practice, since we work with Coq natural numbers (`nat`), we choose A of size 1×1: this allows recursion to go smoothly and helps to keep proofs simple. We reckon that using a divide-and-conquer scheme might be more efficient. Yet, the definition of the star of a matrix is used only in proofs: we may freely use the slower and more simple version.

The corresponding code is given in Fig. 1.16. First, we implement a function `mx_star'` that decomposes its input into four quadrants, according to the above definition by blocks (†). The function `mx_star'` is defined through open-recursion: it takes as arguments two functions to perform the recursive calls (i.e., to compute A' and D'). The final function `mx_star` is defined as a fixpoint to tie the knot of the recursion, using `mx_star_11` to compute the star of a 1×1 matrix by using the star operation on the underlying element. Note that by making explicit the general block definition with the auxiliary function `mx_star'`, we can easily state theorem `mx_star_block`: equation (†) holds for each possible decomposition of the matrix. In particular, it makes it possible to prove interesting lemmas like the following, which are necessary as soon as one needs to deals with block matrices.

```
Lemma mx_blocks_star_diagonal n m (M: MX n n) (N: MX m m) :
    (mx_blocks M 0 0 N)* ≡ mx_blocks (M*) 0 0 (N*).
Lemma mx_blocks_star_trigonal n m (M: MX n n) (N: MX m m) (P: MX n m):
    (mx_blocks M P 0 N)* ≡ mx_blocks (M*)  (M* · P · N*)  0 (N*).
```

The lemmas above conclude our study of our formalisation of matrices in Coq.

1.6 The algorithm and its proof

We now focus on the heart of our library, the implementation of the decision procedure. We first discuss the implementation of the various kinds of automata we will need in Coq, before discussing how we amended the computational part of Kozen's proof for efficiency reasons.

1.6.1 Representation of automata in Coq

The various representations of finite automata we need are depicted in Fig. 1.17. The first record type (`MAUT.t`) correspond to the matricial representation of automata from §1.3.4 (through this section, `MX n m` is the type of $n \times m$ matrices over regular expressions): it is exactly a representation of the matricial automata, that packages together the number of states (field

Figure 1.17 Coq types and evaluation functions of the four automata representations.

```
Module MAUT.
 Record t := build {
  size:    nat;
  initial: MX 1    size;
  delta:   MX size size;
  final:   MX size 1
 }.
 Definition eval(A: t): regex :=
  mx_to_scal ((initial A)·(delta A)*·(final A)).
End MAUT.
```

```
Module eNFA.
 Record t := build {
  size:    state;
  labels:  label;
  epsilon: state→stateset;
  delta:   label→state→stateset;
  initial: state;
  final:   state }.
 Definition to_MAUT(A: t): MAUT.t.
 Definition eval A := MAUT.eval (to_MAUT A).
End eNFA.
```

```
Module NFA.
 Record t := build {
  size:    state;
  labels:  label;
  delta:   label→state→stateset;
  initial: stateset;
  final:   stateset }.
 Definition to_MAUT(A: t): MAUT.t.
 Definition eval A := MAUT.eval (to_MAUT A).
End NFA.
```

```
Module DFA.
 Record t := build {
  size:    state;
  labels:  label;
  delta:   label→state→state;
  initial: state;
  final:   stateset }.
 Definition to_MAUT(A: t): MAUT.t.
 Definition eval A := MAUT.eval (to_MAUT A).
End DFA.
```

`size`), the matrix of initial states, the transition matrix, and the matrix of final states. The function `MAUT.eval` computes the term of the syntactic Kleene algebra that is the evaluation of the automaton and casts it to a scalar. (Like in the previous section, we are solely interested here in syntactic objects, and will therefore use the words "the evaluation of the automata" to denote the regular expression that results from `MAUT.eval`.) It must be noted that this matricial representation is computationally inefficient[11]; however, it is only used in proofs.

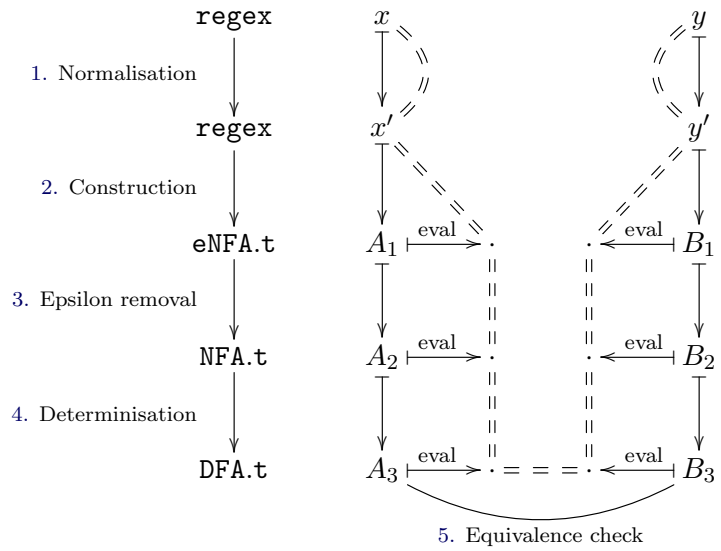
Then, we can use some properties of ϵ-NFAs, NFAs, and DFAs to provide some tailored, more computational, definitions in Coq that overcome the inefficiencies of the matricial representation. The field `labels` bounds the maximal label that occur in a given automaton (this allows one to iterate over a subset of the whole alphabet Σ in some automata constructions). The transition relations are encoded as maps indexed by labels and states (fields `delta` and `epsilon`): we rely on the efficient implementations of finite sets and finite maps from the standard Coq library as the data-structure underlying the functions.

The only peculiarity is our representation of ϵ-NFAs that features a single initial state and a single final state. This is without loss of generality[12], and is tailored for our ϵ-NFA construction algorithm: it happens that the efficient automata construction we implemented produces automata with a single initial state, and a single final state, justifying this choice of representation. The other fields should be self-explanatory. In each case, we define a translation function to matricial automata (`to_MAUT`), so that each kind of automata can eventually be evaluated into a regular expression.

[11] Indeed, the elements of the matrix are regular expressions that may need to be simplified, e.g., $a{\cdot}1{+}1^{\star}{\cdot}(b{+}0)$. A more efficient representation is a family of boolean matrices, one for each letter of the alphabet, and one to account for ϵ-transitions.

[12] All ϵ-NFAs can be represented that way.

Figure 1.18 Overall picture for the algorithm and its correctness.



1.6.2 Road-map

The algorithm we chose to implement to decide whether two regular expressions denote the same language follows the same major steps as in §1.3.4. However, we have to take a particular care about efficiency; this drives our choices about both data-structures and algorithms, which differ slightly from the previous ones. We summarise here our implementation choices:

1. normalise both expressions to turn them into "strict star form" to reduce their size, and improve on the efficiency of the later steps;
2. build ϵ-NFAs using a variant of Ilie and Yu's construction [91], which produces smaller automata than Thompson's construction;
3. remove ϵ-transitions to get NFAs;
4. use the accessible subset construction to get DFAs;
5. check that the two DFAs are equivalent using Hopcroft and Karp's algorithm.

The overall structure of the algorithm and the correctness proofs are depicted in Fig. 1.18 (which is the counterpart of Fig. 1.8). Datatypes are indicated on the left-hand side. Again, the outer part corresponds to computations, while the inner-part corresponds to the equalities: we prove that each automata construction preserves the semantics of the initial regular expression. That is, each kind of automata can be translated to a matricial automata (`MAUT.t`) which can then be evaluated to a regular expression equal to the original one.

1.6.3 Normalisation to strict star form

Recall that the equational theory of regular expression is not finitely based [132]: there is no complete rewriting system that decides equivalence of regular expressions. Yet, regular expressions may be simplified using convergent rewriting systems (that preserve equivalence) in order to reduce their size. And actually, we shall see that reducing regular expressions to a particular kind of normal forms may be leveraged to get a simpler and more efficient algorithm to remove ϵ-transitions.

Figure 1.19 Converting expressions to strict star form

```
Fixpoint remove (e: regex) :=
if contains_one e then
match e with
  | RegExp.plus a b ⇒
    plus_but_one (remove a) (remove b)
  | RegExp.dot a b ⇒
    plus_but_one (remove a) (remove b)
  | RegExp.star e ⇒ e
  | e ⇒ e
end else e.
```

```
Fixpoint ssf (e: regex) : regex :=
match e with
  | RegExp.plus a b ⇒
    RegExp.plus (ssf a) (ssf b)
  | RegExp.dot a b ⇒
    dot' (ssf a) (ssf b)
  | RegExp.star e ⇒
    star' (remove (ssf e))
  | e ⇒ e
end.
```

We use the syntactic simplification procedure proposed by Brüggemann-Klein [38] to transform any stared expression $x^\star$, where x accepts the empty word, into an equivalent expression $y^\star$ such that y does not accept the empty word. That is, we put the regular expressions into *strict star form*: all occurences of the star operation act on *strict regular expressions*, that do not accept the empty word. For instance, this procedure transforms the expression on the left-hand side below into an equivalent strict star form regular expression on the right-hand side:

$$((a+1)\cdot((b+1)^\star\cdot c+d^\star))^\star \quad \rightarrow \quad (a+b^\star\cdot c+d)^\star$$

In Coq, this procedure translates into two simple fixpoints described in Fig. 1.19, and "smart" constructors `dot'` and `star'` that respectively avoid building terms like $x\cdot 1$, and $1^\star$. The `plus_but_one` function is slightly different: it avoids adding 1 or 0 under a star. The following theorem corresponds to the first step of the overall proof, as depicted in Fig. 1.18.

```
Theorem ssf_correct: ∀ x, ssf x ≡ x.
```

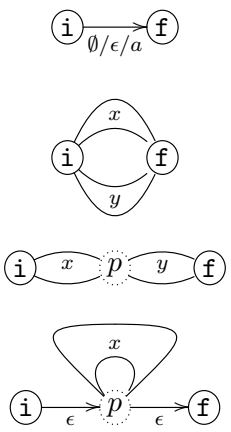
We also prove the completeness of this procedure, i.e., that it returns expressions in strict star form. In the following, we will prove that applying our automata construction to this particular kind of expression yields particular ϵ-NFAs, that are then amenable to more efficient ϵ-transitions removal.

```
Inductive strict_star_form: regex → Prop := ...
Theorem ssf_complete: ∀ x, strict_star_form (ssf x).
```

1.6.4 Automata constructions algorithms

At first, we implemented Thompson's construction, for its simplicity; we finally switched to a variant of Ilie and Yu's construction [91], which produces smaller automata. This algorithm constructs an automaton with a single initial state and a single accepting state (respectively denoted by i and f); it proceeds by structural induction on the given regular expression. The corresponding steps are depicted on the left-hand side of Fig. 1.20; the first drawing corresponds to the base cases (zero, one, variable); the second one is union (plus): we recursively build the two sub-automata between i and f; the third one is concatenation: we introduce a new state, p, build the first sub-automaton between i and p, and the second one between p and f; the last one is for iteration (star): we build the sub-automata between a new state p and p itself, and we link i, p, and f with two ϵ-transitions. The corresponding Coq code is given on the right-hand side. We use an accumulator (`A`) to which we recursively add states and transitions

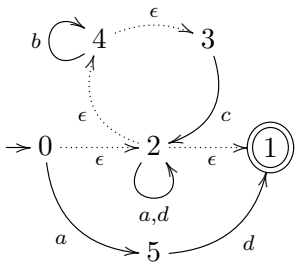
Figure 1.20 Construction algorithm—a variant of Ilie and Yu's construction.




```
Fixpoint build x i f A :=
match x with
  | zero     ⇒ A
  | one      ⇒ add_one i f A
  | var a    ⇒ add_var a i f A
  | plus x y ⇒ build x i f (build y i f A)
  | dot x y  ⇒
     let (p,A) := incr A in
       build x i p (build y p f A)
  | star x   ⇒
     let (p,A) := incr A in
       add_one i p (build x p p (add_one p f A))
end.
```

(the functions `add_one` and `add_var` respectively add epsilon and labelled transitions to the accumulator—the function `incr` adds a new state to the accumulator and returns this state together with the extended accumulator). We give an example to illustrate the way states are numbered by the algorithm.

Example 9. *The strict star form expression* $a \cdot d + (a + b^\star \cdot c + d)^\star$ *yields the automaton:*



We actually implemented this algorithm twice, by using two distinct datatypes for the accumulator: first, with a high-level matricial representation; then with efficient maps for storing ϵ-transitions and labelled transitions. Doing so allows us to separate the correctness proof into an algebraic part, which we can do with the high-level representation, and an implementation-dependent part, showing that the two versions are equivalent.

In Fig. 1.21, we amend the record types `MAUT.t` and `eNFA.t` from Fig. 1.17 to remove the fields for initial and final states: the initial and final state of the automata are always numbered respectively 0 and 1. (The other difference being that we exhibit the underlying maps rather than functions on the efficient side—`pre_eNFA`.) On the high-level side—`pre_MAUT`, we use generic matricial constructions: adding a transition to the automaton consists in performing an addition with the matrix containing only that transition (`mx_point i f x` is the matrix with `x` at position `(i,f)` and zeros everywhere else); adding a state to the automaton consists in adding an empty row and an empty column to the matrix, thanks to the `mx_blocks` function that builds a matrix out of four quadrants. We do not include the corresponding details for the low-level representation: it would require to delve on the particular implementations we used for maps and sets, they are slightly verbose and they can easily be deduced. Notice that `pre_eNFA` does not include a generic `add` function: while the matricial representation allows us to label transitions with arbitrary regular expressions, the efficient representation statically ensures that transitions are labelled either with epsilon or with a variable (a letter of the alphabet).

Figure 1.21 Two intermediate representations for automata

```
Module pre_MAUT.
 Record t := mk {
   size:  nat;
   delta: MX size size }.
 Definition to_MAUT i f A := MAUT.mk
   (mx_point 0 i 1) (delta A) (mx_point f 0 1).
 Definition eval i f := MAUT.eval ∘ (to_MAUT i f)

 Definition add (x: regex) i f A :=
   mk _ (delta A + mx_point i f x)
 Definition add_one := add 1.
 Definition add_var a := add (var a).
 Definition incr A := let mk n M := A in
   (n, mk (n+1) (mx_blocks M 0 0 0)).
 Fixpoint build x i f A := (* Fig. 1.22 *).

 Definition empty := mk 2 0.
 Definition regex_to_MAUT x :=
   to_MAUT 0 1 (build x 0 1 empty).
End pre_MAUT.
```

```
Module pre_eNFA.
 Record t := mk {
   size:     state;
   labels:   label;
   epsmap:   statemap stateset;
   deltamap: statelabelmap stateset }.

 Definition to_eNFA i f A := ...




 Definition add_one i f A := ...
 Definition add_var a i f A := ...
 Definition incr A := ...

 Fixpoint build x := (* Fig. 1.20 *).

 Definition empty := mk 2 0 [] [].
 Definition regex_to_eNFA x :=
   to_eNFA 0 1 (build x 0 1 empty).
End pre_eNFA.
```

The final construction functions, from `regex` to `MAUT.t` or `eNFA.t`, are obtained by calling `build` between the two states 0 and 1 of an empty accumulator. (Note that the occurrence of 0 in the definition of `pre_MAUT.empty` denotes the empty $(2,2)$-matrix; similarly, `[]` denotes the empty map on the right-hand side). Since the two versions of the algorithm only differ by their underlying data structures, proving that they are equivalent is routine (here, [=] denotes matricial automata equality):

```
Lemma constructions_equiv: ∀ x, regex_to_MAUT x [=] eNFA.to_MAUT (regex_to_eNFA x).
```

Let us now focus on the algebraic part of the proof. We have to show:

```
Theorem construction_correct: ∀ x, MAUT.eval (regex_to_MAUT x) ≡ x.
```

The key lemma is the following one: calling `build x i f A` to insert an automaton for the regular expression `x` between the states `i` and `f` of `A` is equivalent to inserting directly a transition with label `x` (recall that transitions can be labelled with arbitrary regular expressions in matricial automata); moreover, this holds whatever the initial and final states `s` and `t` we choose for evaluating the automaton.

```
Lemma build_correct: ∀ x i f s t A, i<size A → f<size A → s<size A → t<size A →
    eval s t (build x i f A) ≡ eval s t (add x i f A).
```

As expected, we proceed by structural induction on the regular expression `x`. As an example of the involved algebraic reasoning, the following property of star w.r.t. block matrices is used twice in the proof of the above lemma: with $(x, y, z) = (e, 0, f)$, it gives the case of a concatenation $(e \cdot f)$; with $(x, y, z) = (1, e, 1)$ it yields iteration $(e^\star)$. In both cases, the state (p) generated by the construction corresponds to the last line and the last column on the left-hand side: x labels the transition going from i to p, y labels the transition going from p to p, and z labels the transition going from p to f.

Note that this laws follows from the general characterisation of the star operation on block matrices (Equation (†) in Sect. 1.5.3), and that this is the analogue of the state elimination

method we explained in §1.1.3. However, the situation is somehow simpler here, since we are in the simple case where there is a single initial state and a single final state.

$$\left[\begin{array}{c|c} u & 0 \end{array}\right] \cdot \left[\begin{array}{ccc|c} & \vdots & & 0 \\ \cdots & A_{i,f} & \cdots & x \\ & \vdots & & 0 \\ \hline 0 & z & 0 & y \end{array}\right]^\star \cdot \left[\begin{array}{c} v \\ \hline 0 \end{array}\right] = u \cdot \left[\begin{array}{ccc} & \vdots & \\ \cdots & A_{i,f} + x \cdot y^\star \cdot z & \cdots \\ & \vdots & \end{array}\right]^\star \cdot v$$

In the special case where `A` is the empty accumulator, lemma `build_correct` gives:

$$\begin{aligned} \texttt{MAUT.eval (regex\_to\_MAUT x)} &\equiv \texttt{eval 0 1 (build x 0 1 empty)} \\ &\equiv \texttt{eval 0 1 (add x 0 1 empty)} \\ &\equiv \left[\begin{array}{cc} 1 & 0 \end{array}\right] \cdot \left[\begin{array}{cc} 0 & x \\ 0 & 0 \end{array}\right]^\star \cdot \left[\begin{array}{c} 0 \\ 1 \end{array}\right] \\ &\equiv \left[\begin{array}{cc} 1 & 0 \end{array}\right] \cdot \left[\begin{array}{cc} 1 & x \\ 0 & 1 \end{array}\right] \cdot \left[\begin{array}{c} 0 \\ 1 \end{array}\right] \\ &\equiv \texttt{x} \end{aligned}$$

i.e., theorem `construction_correct`.
Finally, by combining the equivalence of the two algorithms (lemma `constructions_equiv`) and the correctness of the high-level one (theorem `construction_correct`), we obtain the correctness of the efficient construction algorithm. In other words, we can fill the two triangles corresponding to the second step in Fig. 1.18:

`Theorem construction_correct': ∀ x, eNFA.eval (regex_to_eNFA x) ≡ x.`

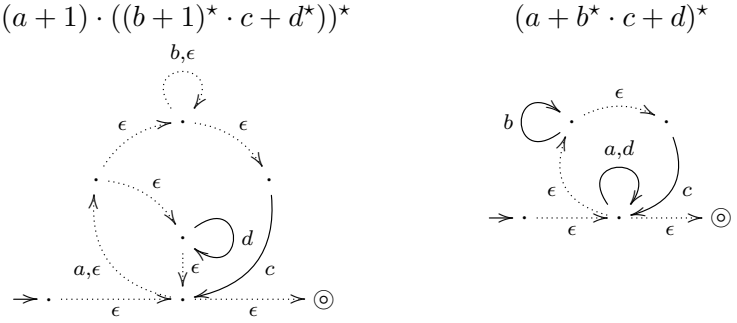
1.6.5 Digression: Comparison with Ilie and Yu's construction

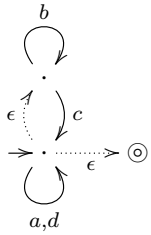
Let us make a digression here to compare our algorithm that constructs ϵ-NFAs with the one proposed by Ilie and Yu [91, Algorithm 4, p. 144]. The steps of the recursive procedure, as presented in Fig. 1.20, are exactly the same; the only difference is that they refine the automaton by merging some states and removing useless transitions on the fly:

(a) the state introduced in the `dot` case is removed when it is preceded or followed by a single ϵ-transition;
(b) all states along an ϵ-cycle introduced in the `star` case are merged into a single state;
(c) if at the end of the algorithm, the initial state only has one outgoing ϵ-transition, the initial state is shifted along this transition;
(d) duplicated transitions are merged into a single one.

Consider for example Fig. 1.22, where we have executed the construction algorithm of Fig. 1.20 on two regular expressions (these are the expressions from Sect. 1.6.3—the right-hand side expression is the strict star form of the left-hand side one). Running Ilie and Yu's construction on the right-hand side expression of Fig. 1.22 yields the automaton below. This automaton is actually smaller than the one we generate: two states and two ϵ-transitions are removed using (a) and (c). Moreover, thanks to optimisation (b), Ilie and Yu also get this automaton when starting from the left-hand side expression, although this expression is not in strict star form.

Figure 1.22 Running the construction algorithm on an expression and its strict star form.





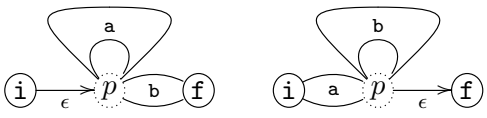
These optimisations are all more or less difficult to implement or to prove. For instance, we did not implement (a) because this optimisation is not simple to code efficiently nor to validate at the algebraic level. Similarly, while step (c) is easier to implement, proving its correctness would require substantial additional work. On the contrary, our presentation of the algorithm directly enforces (d): the data structures we use systematically merge duplicate transitions. The remaining optimisation is (b), which would be even harder to implement and to prove correct than (a). Fortunately by working with expressions in strict star form, the need for this optimisation vanishes: we shall see that ϵ-cycles cannot appear.

In the end, although we implement (b) by putting expressions in strict star form first, the only difference with Ilie and Yu's construction is that we do not perform steps (a) and (c).

The only interesting occurrences of (a) are expressions in strict star form (and thus do not contain products where one of the operands is equal to 1) with sub-terms like $a^\star\cdot b$ or $a\cdot b^\star$ (note that a,b and c may be arbitrary expressions). For these expressions, we would build the automata below:

In these two cases, the optimisation (a) would remove the superfluous intermediate state p'. Would it be needed for efficiency reasons, we could handle these by extending the construction of Fig.1.20 with the following special cases:

```
...
| dot (star a) b ⇒ let (p,A) := incr A in
add_one i f (build a p p (build b p f  A))
| dot a (star b) ⇒ let (p,A) := incr A in
build a i f (build b p p (add_one p f  A))
...
```



The proof of these two special cases would be similar to the proof of the star case, or to the proof of the dot case, using the same lemma on block matrices. Note that these two cases

also cover expressions with sub-terms like $a^\star \cdot b^\star$ or $a^\star \cdot (b^\star + c^\star)$. However, these two cases do not cover expressions with sub-terms like $a \cdot b^\star \cdot c$ where we would build the automaton on the left-hand side below, while, in the end, Ilie and Yu's construction would build the automaton on the right-hand side below:

1.6.6 Epsilon-transitions removal

The automata obtained with the above construction contain ϵ-transitions: each starred sub-expression produces two ϵ-transitions, and each occurrence of 1 gives one ϵ-transition. Indeed, their transitions matrices are of the form

$$M = J + N \quad \text{with} \quad N = \sum_a a \cdot N_a$$

where J and the N_a are 0-1 matrices. These matrices just correspond to the graphs of epsilon and labelled transitions.

We have already seen how the ϵ-transitions are removed algebraically 1.3.4. It involves the computation of the reflexive and transitive closure of the matrix J. Although this is how we prove the correctness of this step, computing $J^\star$ algebraically is inefficient: we have to implement a proper transitive closure algorithm for the low-level representation of automata. We actually rely on a property of the construction from §1.6.4: when given regular expressions in strict star form, the produced ϵ-NFAs have acyclic ϵ-transitions. Intuitively, the only possibility for introducing an ϵ-cycle in the construction from §1.6.4 comes from star expressions. Therefore, by rewriting all occurrences of the star operation so that they act on strict regular expressions, we prevent the formation of ϵ-cycles.

Consider again Fig. 1.22. There are two epsilon-loops in the left hand-side automaton, corresponding to the two occurrences of star that are applied to non-strict expressions ($(b+1)^\star$ and the whole term). On the contrary, in the automaton generated from the strict star form—the second regular expression, the states belonging to these loops are merged and the corresponding transitions are absent: the ϵ-transitions form a directed acyclic graph (here, a tree).

This acyclicity property makes it possible to use a very simple algorithm to compute the transitive closure. With respect to standard algorithms for the general cyclic case, this algorithm is easier to implement, slightly more efficient, and simpler to certify.

Concretely, this means that: 1) we proved that our construction algorithm returns ϵ-NFAs whose reversed ϵ-transitions are well-founded, when the argument is in strict star form; 2) based on this assumption we implemented a simple transitive closure algorithm, using well-founded recursion and memoisation; 3) we proved that this algorithm actually yields an automaton (of type `NFA.t`) whose translation into a matricial automaton is exactly $\langle u \cdot J^\star, N \cdot J^\star, v \rangle$, so that the above algebraic proof applies. This closes the third step in Fig. 1.18.

1.6.7 Determinisation

Determinisation is exponential in worst case: this is a power-set construction. However, examples where this bound is reached are rather contrived: the empirical complexity is tractable. The algorithm we implemented is the accessible subset construction from §1.1.

Given an `NFA.t`, the algorithm constructs a bijection from reachable set of states to states of the `DFA.t`, numbering them "on the fly", and stores the transition relation over these new

Figure 1.23 Depth-first determinisation: main step

```
Definition step (loop: Store → stateset → state → Store)
(p: stateset) (np: num) (a: label) (s: Store) : Store :=
let q := delta_set a p in
let '(table,d,next) := s in
match StateSetMap.find q table with
| None ⇒
  let t' := StateSetMap.add q next table in
  let d' := StateLabelMap.add (np,a) next d in
  loop (t', d', S next) q next
| Some nq ⇒ (table, StateLabelMap.add (np,a) nq d, next)
end.
```

states. We maintain the following data-structure through the standard depth-first enumeration of the subsets accessible from the set of initial states:

```
Notation Table := (statesetmap state).
Notation Delta := (statelabelmap state).
Notation Store := (Table * Delta * state).
```

A `Table` corresponds to the bijection built so far from reachable sets of states to states of the DFA. The map `Delta` corresponds to the transition relation over the states of the DFA. A `Store` is the combination of these two parts, together with a bound which represents the next fresh state, or equivalently, the size of the `Table`. The initial store contains a single state, which corresponds to the set of initial states of the NFA. This initial store is then extended through the iteration of the function `step` in Fig. 1.23.

Assuming that the set of states `p` is mapped to the state `np`, `step loop p np a s` updates the store `s` such that all sets of states accessible from the image of `p` by a transition `a` have been added to the store. There are two cases to consider. If the set of state `nq` has already been added to the table, it suffices to add a transition labelled by `a` from `np` to `nq` in the store. Otherwise, we have to allocate a fresh state in the store that corresponds to the set `q`, and to iterate through the sets accessible from `q`.

Note that we use a form of open recursion through the `loop` argument to deal with a Coq-specific technical difficulty in the concrete implementation of this algorithm that comes from termination. Indeed, the main loop function is intuitively the following:

```
Fixpoint loop (s: Store) p np := fold_labels (step loop p np) max_label s.
```

Of course, we cannot use this definition as it is: there is no structurally decreasing argument for the fixpoint. Since the main loop is executed at most 2^n times (there are 2^n subsets of $[1 \dots n]$), we could try to use this bound directly. However, while we can easily determinise NFAs with 500 states in practice, computing 2^{500} is obviously out of reach (the binary representation of numbers does not help since we need to do structural "unary" recursion); we thus have to iterate lazily. We tried to use well-founded recursion. However, this requires to mix some non-trivial proofs about termination with the code. (Note that we actually defined our ϵ-transitions removal function by well-founded recursion, only because it was the only way to take advantage of the acyclic nature of the ϵ-transition relation.) Here, we use the following "pseudo-fixpoint operators", defined in continuation passing style:

```
Variables A B: Type.
Fixpoint linearfix n (f: (A → B) → A → B) (k: A → B) (a: A): B :=
  match n with O ⇒ k a | S n ⇒ f (linearfix n f k) a end.
```

```
Fixpoint powerfix n (f: (A → B) → A → B) (k: A → B) (a: A): B :=
  match n with O ⇒ k a | S n ⇒ f (powerfix n f (powerfix n f k)) a end.
```

Intuitively, `linearfix n f k` lazily approximates a potential fixpoint of the functional `f`: if a fixpoint is not reached after `n` iterations, it uses `k` to escape. The `powerfix` operator behaves similarly, except that it escapes after $2^n - 1$ iterations: we prove that `powerfix n f k a` is equal to `linearfix (`$2^n - 1$`) f k a`. Thanks to these operators, we can write the code to be executed using `powerfix`, while keeping the ability to reason about the simpler code obtained with a naive structural iteration over 2^n: both versions of the code are easily proved equivalent, using the intermediate `linearfix` characterisation. The final algorithm to build the store is as follows.

```
Definition build_store := powerfix size
  (fun loop (s: Store) (p: stateset) (np: state) ⇒ fold_labels (step loop p np) max_label s)
  (fun s _ _ ⇒ s) initial_store initiaux 0.
```

It is then routine to use this store to build a `DFA.t`

```
Definition NFA_to_DFA : NFA.t → DFA.t := ...
```

The proof of correction of this construction is not trivial though. It involves the exhibition of an invariant of the construction and of a variant (to ensure progress and termination). (Note that we omit here a detailed description of this invariant and this proof to focus on more interesting constructions.) It is then possible to prove that the `DFA.t` we built translates into a matricial automaton which satisfies the equations from §1.3.4. Finally. the following theorem allows us to fill the two squares corresponding to the fourth step in Fig. 1.18.

```
Theorem correct: ∀ (A: NFA.t), DFA.eval (NFA_to_DFA A) ≡ NFA.eval A.
```

1.6.8 Equivalence checking

The next step of Fig. 1.18 is to check the equivalence of the two DFAs we built. Recall the algebraic proof we made in 1.3.4: we showed how the equivalence of two states in the disjoint union of the two DFAs yields a proof of the equivalence of the evaluation of the underlying matricial automata. Therefore, we introduce in Coq a construction of the disjoint union D of two DFAs $\mathcal{A}$ and $\mathcal{B}$, and prove that it is possible to tweak the initial state of the resulting automaton so that it evaluates either to the same language as $\mathcal{A}$ or the same language as $\mathcal{B}$. In the following snippet of code, `pi0` and `pi1` are similar to `inl` and `inr`: they build the disjoint union of the set of states of the two automata. Note that since states are encoded by `positive` numbers, we can use `xO` and `xI`.

```
Lemma eval_left: ∀ (A B: DFA.t), wf A →
  eval (change_initial (merge_DFAs A B) (pi0 (initial A))) ≡ eval A.
Lemma eval_right: ∀ (A B: DFA.t), wf B → max_label A = max_label B →
  eval (change_initial (merge_DFAs A B) (pi1 (initial B))) ≡ eval B.
```

The simulation check algorithm by Hopcroft and Karp [1] we presented in §1.1.4 requires the implementation of a union-find data structure. To our knowledge, there is only one implementation of disjoint-sets in Coq [109]. However, this implementation uses `sig` types to ensure basic invariants along computations, so that reduction of the corresponding terms inside Coq is not optimal: useless proof terms are constantly built and thrown away. Although this drawback disappears when the code is extracted (the goal in [109] was to obtain a certified compiler, by extraction), this is problematic in our case: since we build a reflexive tactic, computations are performed inside Coq. Conchon and Filliâtre also certified a persistent union-find data structure in Coq [50], but this development consists in a modelling of an OCaml library, not in a proper Coq implementation that could be used to perform computations. Therefore, we had to

Figure 1.24 Checking equivalence

```
Variable A : DFA.t.
Let f := DFA.final A.
Let d := DFA.delta A.

Let ml := DFA.max_label A.
Let size := DFA.size A.
Let final s := StateSet.mem s f.
```

```
Fixpoint loop n (T: DS.T) (x y: state) :=
match n with
  | S n ⇒
   let '(b,T) := DS.test_and_unify T x y in
   if b then Some T else
   if eq_bool_bool (final x) (final y)
   then fold_num_option
        (fun a T ⇒ loop n  T (d a x) (d a y))
        ml T
   else None
  | O ⇒ Some T
end.
```

re-implement and prove this data structure from scratch. Namely, we implemented disjoint-sets forests [57] with path compression and the usual "union by rank" heuristic, along the lines of [109], but without using `sig`-types. We provide a functor that defines an efficient disjoint set implementation, given a module that packages a type of elements and efficient maps indexed by these elements. However, we omit the details of the implementation and the proofs, since they quite closely follow [109].

The Coq code for checking equivalence of DFAs is given in Fig. 1.24. We first compute the disjoint union automaton that corresponds to the given DFAs. Then, the algorithm relies on a single primitive from the disjoint set data structure, namely the `test_and_unify` function, that tests whether two states have already been declared equivalent (directly or through the reflexive and transitive closure of the equivalence relation). If the two states were equivalent, the function returns the updated version of the data structure (using the path compression algorithm), and the boolean `true`. If the two states were not already equivalent, the function returns the updated version of the data structure, in which they have been declared equivalent (using the union by rank heuristic), and the boolean `false`. In the latter case, we have to check that the states were both final or both non-final, and proceed recursively to consider all the pairs of states accessible from these states by a transition `a`. (Note that since recursion is not structural, we need to explicitly bound the recursion depth. Here, the size of the disjoint union automaton $(n + m)$ does the job.)

An invariant of this algorithm is that if the disjoint set data structure relates two states, then both are final or non-final. Another (bureaucratic) one is that the disjoint set data structure is well formed. More interesting is the variant we use to prove the correctness of the algorithm (See Fig. 1.25). We denote the equivalence relation induced by a disjoint set data structure `T` as $[\![\texttt{T}]\!]$, and the number of equivalence classes in `T` as `measure T`. Given two states `x` and `y` such that $[\![\texttt{T}]\!]$ `x y` holds, it may be the case that the transitions of `x` and `y` along a label `a` have not yet been added to `T`. Therefore, we introduce the following definition: for an arbitrary relation on states R, we say that S *closes* R if, whenever $R\ x\ y$ holds, then $S\ \delta(a,x)\ \delta(a,y)$ holds. Graphically, we have:

$$\texttt{close}\ R\ S = \quad \begin{array}{ccc} x & \text{---}R\text{---} & y \\ \downarrow & & \downarrow \\ \delta(a,x) & \text{---}S\text{---} & \delta(a,y) \end{array}$$

We say that R is closed up-to S whenever `close` R $(R \cup S)$ holds for all S. Intuitively, S may be seen as a list of pair of states that have not been added to the disjoint set, but which are accessible in one transition from a pair of equivalent states. Now, suppose that $[\![\texttt{T}]\!]$ is closed

Figure 1.25 Variant of the equivalence check

```
Definition close (R S : relation state) :=
∀ a x y (Ha: a<ml)(Hx: x<size)(Hy: y<size),
  R x y → S (d a x) (d a y).
Definition progress T T' := ∀ R,
  close (⟦T⟧) (⟦T⟧ ∪ R) → close (⟦T'⟧) (⟦T'⟧ ∪ R).
```

```
Record variant (T T': DS.T) (x y : state) : Prop :=
{ ci_prog : progress T T';
  ci_le : ∀ u v, ⟦T⟧ u v → ⟦T'⟧ u v;
  ci_measure : measure T' ≤ measure T;
  ci_sameclass : ⟦T'⟧ x y }.
```

up-to S, and that we declare two states `x` and `y` as equivalent (denoted $\mathtt{T} + [\mathtt{x}, \mathtt{y}]$). We show that

$$\mathtt{close}\ ⟦\mathtt{T}⟧\ (⟦\mathtt{T}⟧ \cup S) \implies \mathtt{close}\ ⟦\mathtt{T} + [\mathtt{x}, \mathtt{y}]⟧\ (⟦\mathtt{T} + [\mathtt{x}, \mathtt{y}]⟧ \cup R \cup S)$$

where

$$R = \{(\delta(a, x), \delta(a, y)) \mid a \in \Sigma\}\,.$$

Intuitively, the disjoint set structure grows, and we have to add pairs of elements to the up-to part, in order to match this growth. However, in the end, the up-to part will be subsumed by the relation induced by the disjoint set structure. Indeed, we prove that the disjoint set `T'` returned by the `loop` function starting from `T` verifies the following lemma:

```
Variables x y : state.
Hypothesis Hx : below x size.
Hypothesis Hy : below y size.
Variable T T' : DS.T.
Hypothesis H: (loop A (Datatypes.S size) nil T x y) = Some T'.

Lemma loop_variant : variant T T' x y.
```

Then, starting with the empty disjoint set structure, and using the empty relation as R, we obtain the following lemma:

```
Lemma loop_correct : close (⟦T⟧) (⟦T⟧).
```

It is then possible to prove that the algebraic properties we mentionned in §1.3.4 hold. Finally, the following theorem allows us to fill the bottom part of Fig. 1.18.

```
Theorem valid i j: i ∈ A → j ∈ A → equiv A i j = None →
  eval (change_initial A i) ≡ eval (change_initial A j).
```

1.6.9 Completeness: counter-examples

By combining the proofs from the above sections according to Fig. 1.18, we obtain the correctness of the decision procedure: if `decide_kleene a b` returns `true`, then `a≡b`, and thanks to the untyping theorem ($*$) from §1.3.1, we deduce that `a` and `b` are equal in any typed Kleene algebra.

We also proved the converse implication, i.e., *completeness*. This basically amounts to exhibiting a counter-example in the case where the DFAs are not equivalent. From the algorithmic point of view, this is almost straightforward: it suffices to record the word that is being read in the algorithm from §1.6.8; when two states that should be equivalent differ by their accepting status, we know that the current word is accepted by one DFA and not by the other one. Accordingly, the `decide_kleene` function actually returns an `option (list label)` rather than a boolean, so that the counter-example can be given to the user.

From the proof point of view, to obtain the reverse implication of the equivalence we mentioned in §1.2.3 we just have to show that languages (i.e., predicates over list of labels) form a

Kleene algebra in which the language accepted by a DFA is exactly the language obtained with `DFA.eval`:

```
Theorem interp_DFA_eval: ∀ A: DFA.t, DFA_language A [=] interp (DFA.eval A).
```

(`DFA.eval` actually returns a regular expression which we need to interpret as a language; `DFA_language A` is the predicate that corresponds to word acceptance in the DFA `A`, from Def. 2; [=] is language equality, i.e., pointwise equivalence of the predicates.)

1.6.10 Efficiency

We shall now consider the actual performances of the decision procedure we implemented. On typical use cases, the tactic returns instantaneously. We had to perform additional tests to check that the decision procedure scales on larger expressions. This would be important in a scenario where equations to be solved would be generated automatically by an external tool. Alternatively, and more speculatively, this decision procedure might be used as a black-box inside other decision procedures yielding inputs with size that we cannot foresee.

A key factor for efficiency turns out to be the concrete representation of numbers (i.e., states in our automata), which we detail first.

Numbers, finite sets, and finite maps. To code the decision procedure, we mainly needed a good representation of numbers, of finite sets, and of finite maps. Coq provides several representations for natural numbers: Peano integers (`nat`), binary positive numbers (`positive`), and big natural numbers in base 2^{31} (`BigN.t`), the latter being shipped with an underlying mechanism to use machine integers and perform efficient computations. (On the contrary, unary and binary numbers are allocated on the heap, as any other datatype.) Similarly, there are various implementations of finite maps and finite sets, based on ordered lists (`FMapList`), AVL trees (`FMapAVL`), or uncompressed Patricia trees (`FMapPositive`).

While Coq standard library features well-defined interfaces for finite sets and finite maps, the different definitions of numbers lack this standardisation. In particular, the provided tools vary greatly depending on the implementation. For example, the tactic `omega`, which decides Presburger's arithmetic on `nat`, is not directly available for `positive`[13]. To abstract from this choice of basic data structures, and to obtain a modular code, we designed a small interface to package natural numbers together with the various operations we need, including sets and maps. We specified these operations with respect to `nat`, and we defined several automation tactics. In particular, by automatically translating goals to the `nat` representation, we can use the `omega` tactic in a transparent way, notwithstanding the actual definition of numbers we used.

However, note that we do not parameterise the implementations of finite sets and finite maps by the definition of numbers: it would miss some opportunities for synergies. For instance, using Coq's `positive` numbers makes it possible to use efficient radix-2 search trees as maps (and sets): accessing an element is done by reading the key without having to make comparison as would be the case using AVL based trees. Hence, our decision procedure remains modular w.r.t. to the implementation of numbers, but this modularity may be more coarse grained than usual.

Tests. We experimented with several implementations of this interface, to compare their relative performances: we measure the time required by the decision procedure to prove $x = x$, where x is a given regular expression. We denote by "number of internal nodes" the number of

[13] Note that there is some ongoing standardisation work on this

`plus`, `dot`, or `star` nodes in the abstract syntax tree of the regular expression, and we denote by the "number of variables" the size of the underlying alphabet.

Of course, unary natural numbers behave badly since they bring an additional exponential factor. However, thanks to the efficient implementation of radix-2 search trees for finite maps and finite sets (`FMapPositive` and `FSetPositive`), we actually get higher performances by using `positive` binary numbers rather than machine integers (`BigN.t`), even if the latter benefits from machine arithmetic. This is no longer true with the extracted code: using machine integers is faster on large expressions with a thousand internal nodes . We could of course benefit from the addition of imperative features to Coq, as proposed in [9]. The use of persistent arrays together with better support for machine-integers would certainly improve over the situation we describe.

We performed intensive tests on randomly generated regular expressions. The decision procedure tends to run in less than one second for expressions with 200 internal nodes and 30 variables, and less than one minute for even larger expressions (1000 internal nodes, 100 variables), which are very unlikely to appear in "human-written" proofs. The above timings correspond to the "tactic scenario", where execution takes place inside Coq; when extracting to OCaml, the resulting code executes approximately 20 times faster.

We do not include a more precise timing table here for several reasons: timings are machine-dependent, generating random regular expressions in a uniform way is difficult, and the nature of the algorithm makes it highly non-deterministic from the complexity point of view. Indeed, the running time is roughly proportional to the sizes of the generated DFAs, and these sizes greatly vary from one case to another (exponential in the worst case, pretty small in the average case). In particular, with large expressions, the mean time recorded on a thousand random tests is not significant: the standard deviation is too high. Nonetheless, our benchmarks are available for the interested reader [33].

1.7 Additional constructions

We now turn to the presentation of two other automata constructions we used at some point in our development, but were later dropped.

1.7.1 Using Thompson's construction, and matrix computations

In the algebraic presentation of Kozen's proof (§1.3), we used Thompson's construction, for the sake of simplicity. We also used it for a while as our automata construction algorithm in the Coq part, until we understood that we could prove algebraically the correction of a more efficient construction.

More precisely, our automata construction was building a 3-uple of Coq matrices, following exactly Thompson's construction from Fig. 1.9. At this point, we relied heavily on matrix computations, and the use of cleverly placed memoisation operators to take advantage of the call-by-value evaluation of Coq's `vm_compute` machinery.

At first, we tried to work with matrices over arbitrary regular expressions, prove that these matrices were simple, and use these hypotheses in proofs. However, from the computational point of view, it was obviously wrong. A slightly better idea was to define an automata in terms of a 0-1 matrix of ϵ-transitions J, and a family of 0-1 matrices of labelled transitions M_a. Using the (degenerated) Kleene algebra of booleans[14] as the elements of these matrices allowed for slightly more efficient computations: for instance, the summation occurring in the matrix product could be implemented lazily.

[14] where $+$ is 'or', $\cdot$ is 'and', and $x^\star$ is true

Figure 1.26 Coq code for minimisation.

```
(* a              : label                        *)
(* i              : state                        *)
(* p,q,pt,pf,inv  : set state                    *)
(* P              : set (set state)              *)
(* L              : set (label * set state)      *)

Variables states, finaux: stateset.
Variable  labels: set label.
Variable  delta: state → label → state.

Definition delta_inv:
  label * stateset → stateset := ....

Definition splittable p inv :=
 let (pt,pf) := partition (fun i ⇒ i ∈ inv) p
 in if is_empty pt || is_empty pf
    then None
    else Some (pt,pf).

Definition update_splitters p pf pt L :=
  fold (fun a L ⇒ if (a,p) ∈ L
    then {(a,pf),(a,pt)} ∪ L \ {(a,p)}
    else
      if cardinal pf < cardinal pt
      then {(a,pf)} ∪ L
      else {(a,pt)} ∪ L
  ) labels L.
```

```
Definition split inv P L :=
  fold (fun p acc ⇒
    match splittable p inv with
      | None ⇒ acc
      | Some (pf,pt) ⇒
        let (P,L) := acc in
          ({pf, pt} ∪ P \ p,
           update_splitters p pf pt L))
    end
  ) P (P,L).

Function loop (P,L) {wf RPL (P,L)} :=
  match choose L with
    | None ⇒ P
    | Some x ⇒ loop
       (split (delta_inv x) P (L\x))4
  end.

Definition partition :=
  loop
    {finals, states\finals}
    (labels × {finals}).
```

In this setting, it was possible to actually compute the matrix $J^\star$ and the matrices $M_a \cdot J^\star$, yielding a matricial NFA. However, this construction step was most surely the bottleneck of our decision procedure: the construction algorithm using block matrices was building big automata, and we had to compute the star of the whole ϵ-transition matrix whatever the number of ϵ-transitions. Indeed, the matrices we built were rather sparse. This led us to restart almost from scratch, and to seek more efficient constructions.

The decisive move was to add the conceptual and notational overhead of the "low-level" representations of automata `eNFA.t`, `NFA.t` and `DFA.t`, emphasising a clear separation between the computational world, and the world in which proofs were to be made. In a nutshell, this experience led us to believe that making a distingo between the low-level implementation that will compute, and the high-level representation used in proof is worth the effort.

1.7.2 Minimisation algorithms

In the first version of this decision procedure, we followed scrupulously the computational content of Kozen's proof. Therefore, we implemented and proved correct a minimisation algorithm. The Coq implementation is sketched in Fig. 1.26: it consists in a 'while' loop containing two nested 'for' loops, translated using the `fold` operation of finite sets. The termination of the external loop is ensured using a well-founded relation (the algorithm could be rewritten so as to use structural recursion only, we found the resulting code less clear and harder to prove, however).

The idea of the algorithm is to start from an initial partition of states (final and non final

states), and to refine this partition whenever one of its elements is `splittable`: i.e., when a move from a set of state can lead to two different sets by a transition with a given label `a`. The implementation of this predicate is made efficient by pre-computing the inverse transition graph (`delta_inv`).

We implemented a variant of Hopcroft's minimisation algorithm [77, 86]. This algorithm uses a set `L` of *splitters*, i.e., pairs (label, state set) w.r.t. which one must attempt to split classes of the partition. The crux of the algorithm is to keep from adding too much redundancy in `L`: if a pair `(a,q)` is not in this set, then either every class of the partition is already split w.r.t. `(a,q)`, or `L` contains enough pairs to subsume `(a,q)`.

We met two problems with this algorithm. First, the "process the smaller half" optimisation (the inner `if` of the `update_splitters` function) happened to be rather difficult to prove correct. Second, this partition refinement algorithm runs in $O(n \log n)$ only when one combines the "process the smaller half" idea (which is presented here) with a doubly-linked list representation of partition blocks. The latter point is required to be able to compute the function `update_splitters(q)` in time $O(|\delta^{-1}(q)|)$.

We devoted some time to the definition of a refined version of this algorithm, using the correct underlying data structures. However, we later moved to the equivalence check we presented in §1.6.8, which made the minimisation step useless. With hindsight, we realise that such a minimisation step may have been the perfect candidate for *verification*. Indeed, we could have used an oracle (read: an OCaml or a Coq program) to build a minimised automaton, and a certified verifier (the equivalence test from §1.6.8) could have been used to check this result. (Note however that verifying that a minimised automaton is *the* minimal automaton corresponding to a given DFA requires more computations, but this is not necessary for many practical applications which only require smaller automata.)

1.8 Some complete examples

To conclude this chapter, we present some interesting examples of use of our tactics to solve practical problems. For the sake of readability, we shall remain high-level: the interested reader may consult the corresponding proof scripts [33].

1.8.1 Mechanised Church-Rosser theorems

We sketch the definition of the model of Kleene algebras that corresponds to the binary relations from Coq standard library: this allows to give short mechanised proofs of so-called "Church-Rosser" theorems. First, recall that the composition of binary is defined as follows:

```
Definition comp A (R S : relation A) : relation A := (fun x z ⇒ ∃ y, R x y ∧ S y z).
```

Then, note that homogeneous binary relations form an untyped model: the instances in the left-hand side of Fig. 1.27 are parametrised by a set `A` coding for the domain and the co-domain of the relations, and all operations are total. As it was the case with languages, we use the singleton type `unit` for the index type `T` in the graph instance, and all operations just ignore the superfluous parameters.

We state some such "Church-Rosser" theorems from [151] on the right-hand side of Fig. 1.27. While these theorems cannot be solved automatically using our tactic, they can nevertheless be given short proofs (i.e., less than 7 lines) that make intensive use of the `kleene_reflexivity` decision procedure. (Note that while we state them in the particular model of homogeneous relations, these theorems are actually valid in any Kleene algebra.)

Figure 1.27 Some Church-Rosser theorems

```
Context {A: Type}.
Instance RG: Graph := {
  T := unit;
  X n m := relation A;
  equal n m := same_relation A}.
Instance RSL_Ops: SemiLattice_Ops RG :=
{ plus n m := union A;
  zero n m := empty  (* fun x y ⇒ False *)}.
Instance RM_Ops: Monoid_Ops RG :=
{ dot n m p := comp A;
  one n := eq}.
Instance RS_Op: Star_Op RG :=
{ star n := clos_refl_trans A}.
```

```
Variable (a b : relation A).
Theorem SemiConfluence_is_ChurchRosser:
b · a* ⊆ a* · b*  ↔  (a+b)* ⊆ a* · b*.

Theorem WeakConfluence_is_ChurchRosser:
b* · a* ⊆ a* · b*  ↔  (a+b)* ⊆ a* · b*.

Theorem BubbleSort:
b · a ⊆ a · b  →  (a+b)* ⊆ a* · b*.

Theorem Hindley_Rosen:
b·a ⊆ a*·(b+1) → b*·a* ⊆ a*·b*.
```

1.8.2 McNugget Numbers and the Coin Problem

A number n is a McNugget number when it can be expressed as $6x+9y+20z$, with x, y, z being positive numbers. A well-known mathematical riddle ask what is the highest number N that is not a McNugget number. That is, find the least N such that:

$$\forall n, N \leq n \implies \exists x, \exists y, \exists z, n = x * 6 + y * 9 + z * 20.$$

There exists many similar problems. For instance, one can prove that

$$\forall n, 8 \leq n \implies \exists x, \exists y, n = x * 3 + y * 5.$$

More generally, let $(a_i)_{1 \leq i \leq p}$ be a family of relatively prime numbers. The *Coin Problem* asks what is the biggest n that cannot be expressed as a sum of multiples of the a_i. While being an interesting mathematical problem on its own, this is also a good source of examples for our tactic. For instance, the inequation

$$\forall n, 8 \leq n \implies \exists x, \exists y, n = x * 3 + y * 5$$

can be translated into the following inequality in Kleene algebras

$$a^8 \cdot a^\star \leq (a^3 + a^5)^\star \qquad \text{where } \begin{cases} a^0 & = 1 \\ a^{(n+1)} & = a \cdot a^n. \end{cases}$$

The following excerpt shows that these inequations can be stated, and proved, in the context of an abstract Kleene algebra. Our decision procedure solves these two inequations almost instantaneously.

```
Context {KA: KleeneAlgebra}.
Variable t : T.
Variable a : X t t.

Fixpoint pow {A} n (x : X A A) : X A A :=
match n with
| 0 ⇒ 1
| S n ⇒ x · pow n x
end.
```

```
Goal pow 8 a · a* ⊆ (pow 3 a + pow 5 a)*.
simpl pow. Time kleene_reflexivity. (* 0.01 s *)
Qed.

Goal pow 43 a · a* ⊆ (pow 6 a + pow 9 a + pow 20 a)*.
simpl pow. Time kleene_reflexivity. (* 0.01 s *)
Qed.
```

1.9 Related works and discussion

Finite automata theory

The notion of strict star form §1.6.3 was inspired by the standard notion of *star normal form* [38] and the idea of *star unavoidability* [91]. To the best of our knowledge, using this notion to get ϵ-NFAs with acyclic epsilon-transitions is a new idea.

Our presentation of finite automata (§1.1) laid the ground for Kozen's proof and our Coq implementation (which follows roughly the same steps). In particular, we went through the construction of ϵ-NFAs and the subsequent removal of epsilon-transitions. However, there exists several direct translations from regular expressions to NFAs, e.g., the Berry-Sethi construction of the *position automaton* and its ulterior refinements, the *continuation automaton* [43, 91] and the *equation automaton* [45]. It has been demonstrated in [44] that the equation automaton is a quotient of the continuation automaton, which is in turn a quotient of the position automaton. Moreover, each of these construction may be implemented to run in quadratic time with respect to the size of the input regular expression (see for instance the survey of these constructions given in [131]).

A preliminary investigation shows that these direct constructions of NFAs from regular expressions may be more efficient than our implementation of a variant of Ilie and Yu's construction (without all the optimisations that are required to build a quotient of the position automaton) followed by the removal of the epsilon-transitions. That is, we implemented Berry-Sethi's construction of the position automaton, and assessed the variation of the performances of our decision procedure. While this new version of decision procedure seems to be slightly less efficient on small examples, we measured an improvement of roughly 40% on bigger expressions.

This suggests that our tactic would benefit from switching to a (certified) implementation of one of the aforementioned direct construction of NFAs. Note that we would not need to give a proof of these constructions in the matricial setting: we could use the fact that our existing decision procedure is complete with respect to the model of regular languages (see §1.6.9). Hence, it would be possible to use a more efficient algorithm as a decision procedure for the equational theory of Kleene algebra, as soon as we are able to prove that it decides the equivalence of regular expressions.

Formalising finite automata theory

In a recent paper, Krauss and Nipkow [104] described an elegant equivalence checker for regular expressions, based on derivatives of regular expressions. It works by constructing a bisimulation relation between derivatives of regular expressions. The regular expressions are then mapped to binary relations, which yields an automatic and complete proof method for (in)equalities of binary relations over union, composition, (reflexive) and transitive closure. They state:

> *"[F]ormalising Kozen's theorem is not easy—the proof amounts to replaying the well-known automaton constructions in an algebraic setting, using matrices. Moreover, the automata theory needed does not come for free either."*

Indeed, they manage to lift their decision procedure based on regular expression derivatives to the binary relations, which yields a low-cost and elegant alternative to a whole development of automata theory and the formalisation of Kozen's proof. Compared with our development, they cut corners in three places: going this way requires less notational and conceptual overhead (matrices, automata); they only formalise the soundness proof (not completeness), which is sufficient to build a tactic; they concentrate on the claimed main use case, namely homogeneous binary relations. To conclude, they rightfully state:

"In a nutshell, the succinctness of our development is the result of the chosen formalisation and of concentrating on the essentials."

Indeed, their approach is much simpler for several reasons. First, they implemented an algorithm based on Brzozowski's derivatives [41, 135], which is much simpler than ours, but also less efficient: the DFAs are produced directly from the regular expressions, but they can be much larger [91]. Second, they do not prove Kozen's initiality theorem: they prove correctness in the model of regular languages and they use a nice mathematical trick to reach the model of binary relations. As a consequence, their tactic cannot be used with other models like matrices, $(min, +)$ algebras, or weighted relations. Third, they do not formalise the proof of completeness, or equivalently, the fact that the algorithm always terminates (Isabelle/HOL computations do not need to terminate so that they can use a "while-option" combinator). Indeed, proving termination and completeness belongs in the realm of meta-theory and is not required to use the tactic. Finally, we do not include detailed comparisons of the relative performances of our implementations of certified procedure for regular expression equivalence. Yet, in the tactic scenario, we claim that our decision procedure is roughly one order of magnitude faster than theirs on simple expressions (with 100 internal nodes and 10 variables), and roughly two orders of magnitude faster than theirs on bigger expressions (500 internal nodes and 50 variables).

More recently, Wu, Zhang and Urban [158], developed a formalisation of the Myhill-Nerode theorem based on regular expressions. In this paper, they argue that formalising automata in HOL-based theorem provers is difficult because

"they need to be represented as graphs, matrices or functions, none of which are inductive data-types. Also, convenient operations for disjoint unions of graphs, matrices and functions are not easily formalisable in HOL."

Then, they take the view that a regular language is one language where there exists a regular expression that matches all of its strings. The reason is that regular expressions can be defined as an inductive datatype. The paper proceed to prove that a central result of regular languages—the Myhill-Nerode theorem—can be recreated by only using regular expressions. Recall that this theorem states that a language has finitely many equivalence classes of words if and only if the language is regular, where the Myhill-Nerode relation on a language A is

$$R_A\ x\ y \triangleq \forall z, (x \cdot z \in A) \iff (y \cdot z \in A).$$

We made no attempt at formalising this theorem, however, there seems to be little obstacle to the formalisation of its text-book proofs using our library.

Wu, Zhang and Urban argue that their development is more concise than similar developments in Coq[15], one by Filliâtre [64] and one by Almeida et al [5]: the former aimed at formalising Kleene's theorem, while the latter aimed at proving the correctness of Mirkin's construction of partial derivative automata in Coq. While Wu, Zhang and Urban do not make a comparison with ours, we reckon that their development is more concise. However, ours subsumes the proof of Kleene's theorem (the language described by regular expressions are exactly the one defined by finite automata), comprises efficient automata constructions, and was aimed at the proof of Kozen's theorem and its later use as a decision procedure.

Note that while they justify their work by the fact that formalising automata *"can be a real hassle in HOL-based provers"*, and that neither Isabelle/HOL nor HOL4 nor HOL-light support matrices and graphs with libraries, we did not encounter such difficulties in Coq (omitting the intrinsic difficulty of working with rectangular matrices at some points).

[15] but not really comparable in scope

In addition to the formalisations of Filliâtre and Almeida et al we mentioned, note that Briais also formalised decidability of regular languages equality [35] (but not Kozen's initiality theorem). However, his approach is not computational, so that even straightforward identities cannot be checked by letting Coq compute.

Even more recently, Coquand and Siles [56] formalised Bzrozowski's algorithm in Coq. They complete the formal investigations made by Krauss and Nipkow [104], who did not prove formally the termination of their algorithm; and by Almeida et al [5] who did not finish the proof of correctness of their procedure. To this end, Coquand and Siles introduce a new definition of finiteness in type theory, which can be put to use to define functions by well-founded recursion. Thus, they can use the proof that the set of derivatives of a regular expression is finite as the recursive argument for the decision procedure for equivalence of regular expressions. (Note that they use a trick by B. Barras and G. Gonthier that "guards" the proof of well-foundedness by adding constructors in a lazy fashion. Thus, the actual proof of well-foundedness of an example is never computed.) Yet, note that their algorithm performs poorly on medium to big expressions. Again, we do not include detailed benchmarks, but in the tactic scenario, their decision procedure is several order of magnitude slower than ours on expressions with, e.g., 50 internal nodes and 4 internal variables, and do not scale to bigger expressions. For instance, we could not check that the McNugget inequation holds using their implementation in a reasonable amount of time.

Algebraic tools for binary relations

The idea of reasoning about binary relations algebraically is old [60, 152]. Among others [97, 121], Struth applied this idea within an interactive theorem prover [151]. He later turned to automated first-order theorem provers (ATP): Höfner and him verified facts about various relation algebras [84, 85] using Prover9, a resolution/paramodulation based ATP. Our approaches are quite different: we implemented a decision procedure for a decidable theory, whereas their proposal consists in feeding a generic automated prover with the axioms of some algebras, and to see how far the prover can go by itself. As a consequence, their methodology applies directly to a very wide class of goals and algebras, while we are restricted to the equational theory of Kleene algebras. On the other hand, our tactic always terminates, while Prover9 is unpredictable: even for very simple goals, it can diverge, find a proof immediately, or find a proof in a few minutes [85].

Narboux defined a set of Coq tactics for diagrammatic proofs [115]. He works in the concrete setting of binary relations, which makes it possible to represent more diagrams, but does not scale to other models. The level of automation is rather low: it basically reduces to a set of hints for the `auto` tactic.

Formalisation of algebraic hierarchies.

The problem of formalising mathematical structures or algebraic hierarchies in type theory is well-known and usually considered as difficult [13, 24, 49, 67, 68]. Thanks to the recent addition of first-class type-classes [147], we can use a very simple and naive solution here, which gives us overloading for notations, lemmas, and tactics, as well as modularity, sharing, and a basis for reification (Sect. 1.4).

Spitters and van der Weegen [148] recently described how to implement algebraic hierarchy using type-classes. They take the radical view that bundling of operations and laws should be kept to a minimum. We discussed the drawbacks of this approach in §1.4.2, and emphasise the efficiency issues that appear with this solution: in practice, type-class resolution becomes

too slow to be used in sizable developments that feature many instances. While technical improvements may make the situation slightly better, there is strong evidence that more involved solutions, like packed classes[67], are necessary to define large algebraic hierarchies.

More recently, Foster, Struth and Weber [65] demonstrated the use of the Isabelle/HOL theorem prover to build the basis of a repository of algebraic structures. They implement a theory hierarchy for relation and Kleene algebras using Isabelle's local specification facilities (called *locales*), which allow for modular development. Their hierarchy includes several refinements of Kleene algebra, and they prove more than (simple) 800 facts in this setting, using the automated theorem proving features included in Isabelle/HOL. Note that they cannot reuse the decision procedure from [104] to decide (in)-equations in Kleene algebra, since it only applies to the specific model of binary relations.

1.10 Conclusion

We have proved the correctness and completeness of an efficient reflexive decision procedure for Kleene algebras. That is, we have carried a variant of Kozen's proof of the initiality of the model of regular languages for Kleene algebras in Coq via executable finite automata constructions. Moreover, in order to get an efficient decision procedure, we had to use efficient data structures and efficient out-of-the-shelf automata algorithms. To our knowledge, this is the first certified efficient implementation of these algorithms and their integration as a generic tactic.

Our development is roughly 11 000 lines long, which decompose as 5300 lines of specifications (functions, definition of instances of type-classes, etc), 4500 lines of proof, and 1000 lines of comments. More precisely, the algebraic hierarchy (including matrices) accounts for around 2000 lines of specification, and 1200 lines of proofs. The decision procedure by itself (and its correctness proof) accounts for around 1800 lines of specification and definitions, and 2400 lines of proofs. The definition of several models of Kleene algebra or general purpose libraries of definitions and lemmas (especially about finite sets and finite maps) accounts for the remainder.

In the end, we did not use much automation besides tactics to decide equations in (non-commutative) semirings, proof search hints for the usual `auto` tactic of Coq, and decision procedure for arithmetic goals (like `omega`). Note that using type-classes to share lemmas between structures was obviously crucial for keeping the "proof script vs definitions" ratio down.

However, the amount of algebraic reasoning we had to handle prompted our decision to investigate tactics to rewrite equations modulo associativity and commutativity of some operators.

Chapter 2

Tactics for rewriting modulo AC

The amount of algebraic reasoning we had to handle to complete the decision procedure of the previous chapter prompted our decision to investigate tactics to rewrite equations modulo associativity and commutativity of some operators.

In this chapter, we propose a solution to this short-comings for the Coq proof-assistant: we extend the usual rewriting tactic to automatically exploit commutativity and associativity (AC), or just associativity (A) of some operations.

2.1 Introduction

Motivations. Typical hand-written mathematical proofs deal with commutativity and associativity of operations in a liberal way. Unfortunately, a proof assistant requires a formal justification of all reasoning steps, so that the user often needs to make boring term re-orderings before applying a theorem or using an hypothesis. Suppose for example that one wants to rewrite using a simple hypothesis like `H`: $\forall$ `x`, `x+-x` $= 0$ in a term like `a+b+c+-(c+a)`. Since Coq standard `rewrite` tactic matches terms syntactically, this is not possible directly. Instead, one has to reshape the goal using the appropriate commutativity and associativity lemmas:

```
Lemma add_comm: ∀ x y, x+y=y+x.      Lemma add_assoc: ∀ x y z, x+(y+z)=(x+y)+z.

  (* ((a+b)+c)+-(c+a) = ... *)              rewrite (add_comm a b), ← (add_assoc b a c).
  (* (b+(a+c))+-(c+a) = ... *)              rewrite (add_comm c a), ← add_assoc.
  (* b+((a+c)+-(a+c)) = ... *)              rewrite H.       (* b+0 = ... ** )
```

This is not satisfactory for several reasons. First, the proof script is too verbose for such a simple reasoning step. Second, while reading such a proof script is easy, writing it can be painful: there are several sequences of rewrites yielding to the desired term, and finding a reasonably short one is difficult. Third, we need to copy-paste parts of the goal to pick the right occurrences when rewriting the associativity or commutativity lemmas: this is not a good practice since the resulting script breaks when the goal is subject to small modifications. (Note that one could also select occurrences by their positions, but this is at least as difficult for the user which then has to count the number of occurrences to skip, and even more fragile since these numbers cannot be used to understand the proof when the script breaks after some modification of the goal.)

Trusted unification vs untrusted matching. There are two main approaches to implementing rewriting modulo AC in a proof-assistant. First, one can extend the unification mechanism and the conversion test of the system to work modulo AC [126, 150]. This option is quite powerful, since most existing tactics would then work modulo AC. In the case of Coq, it however

requires non-trivial modifications of the core (the unification mechanism) and the kernel (the conversion test) of the proof assistant. As a consequence, this obfuscates the meta-theory: we need a new proof of strong normalisation and we increase the trusted code base. Moreover, this would make the unification mechanism less efficient: unification modulo AC yields *finite sets* of unifiers, while there is at most one unifier [124] in the Calculus of Constructions (in the decidable fragment where terms are restricted to higher-order patterns).

On the contrary, we can restrict ourselves to pattern matching modulo AC and use the core-system itself to validate all rewriting steps [32]. While being less powerful in theory, this does not require modification of the kernel of the proof-assistant, and does not affect the efficiency of the general purpose unification algorithm. We chose this option.

Contributions, scope of the library. Besides the facts that such tools did not exist in Coq before and that they apparently no longer exist in Isabelle/HOL (see §2.9 for a more thorough discussion), the main contributions of this work lie in the way standard algorithms and ideas are combined together to get tactics that are efficient, easy to use, and covering a large range of situations:

- We can have any number of associative and possibly commutative operations, each possibly having a neutral element. For instance, we can have the operations `min`, `max`, $+$, and $*$ on natural numbers, where `max` and $+$ share the neutral element 0, $*$ has neutral element 1, and `min` has no neutral element.
- We deal with arbitrary user-defined equivalence relations. This is important for rational numbers or propositions, for example, where addition and subtraction (resp. conjunction and disjunction) are not AC w.r.t Leibniz equality, but for the relation `Qeq` (resp. `iff`).
- We handle "uninterpreted" function symbols: n-ary functions for which the only assumption is that they preserve the appropriate equivalence relation. For example, subtraction on rational numbers is a proper morphism for `Qeq`, while pointwise addition of numerators and denominators is not. (Note that any function is a proper morphism for Leibniz equality.)
- The interface we provide is straightforward to use: it suffices to declare instances of the appropriate type-classes for the operations of interest, and our tactics will exploit this information automatically. Moreover, since the type-class implementation is first-class, this gives the ability to work with polymorphic operations in a transparent way. For instance, concatenation of lists is declared as associative once and for all.

Methodology. Recalling the example from the beginning, an alternative to explicit sequences of rewrites consists in making a transitivity step through a term that matches the hypothesis' left-hand side syntactically.

```
a, b, c: Z
H: ∀ x, x + −x = 0
======================
((a+b)+c)+−(c+a) = ...
```

```
transitivity (b+((a+c)+−(a+c))).
  (* ((a+b)+c)+-(c+a) = b+((a+c)+-(a+c)) *)
  ring. (* aac_reflexivity *)
  (* b+((a+c)+-(a+c)) = ... *)
  rewrite H.
  (* b+0 = ... *)
```

Although the `ring` tactic [76] solves the first sub-goal here, this is not always the case (e.g., there are AC operations that are not part of a ring structure). Therefore, we have to build a new tactic for equality modulo A/AC: `aac_reflexivity`. Another drawback is that we have to copy-paste and modify the term manually, so that the script can break if the goal evolves. This can be a good practice in some cases: the transitivity step can be considered as a robust

Figure 2.1 Classes for declaring properties of operations.

```
Variables (X: Type) (R: relation X) (op: X → X → X).
Class Associative := law_assoc: ∀ x y z, R (op x (op y z)) (op (op x y) z).
Class Commutative := law_comm: ∀ x y, R (op x y) (op y x).
Class Unit (e: X) := { law_id_left: ∀ x, R (op e x) x; law_id_right: ∀ x, R (op x e) x }.
```

and readable documentation point; in other situations we want this step to be inferred by the system, by pattern matching modulo A/AC [90].

All in all, we proceed as follows to achieve a whole automation of the process. Let $\equiv_{AC}$ denote equality modulo A/AC; to rewrite using a universally quantified hypothesis of the form $H : \forall \overline{x}, p\overline{x} = q\overline{x}$ in a goal G, we take the following steps, which correspond to building the proof-tree on the right-hand side:

1. choose a context C and a substitution σ such that $G \equiv_{AC} C[p\sigma]$ (pattern matching modulo AC);
2. make a transitivity step through $C[p\sigma]$;
3. close this step using a dedicated decision procedure (`aac_reflexivity`);
4. use the standard `rewrite`;
5. let the user continue the proof.

$$\dfrac{\dfrac{}{G \equiv_{AC} C[p\sigma]}\,3 \qquad \dfrac{H \qquad \dfrac{\ldots}{C[q\sigma]}\,5}{C[p\sigma]}\,4}{G}\,2$$

For the sake of efficiency, we implement the first step as an OCaml oracle, and we check the results of this (untrusted) matching function in the third step, using the certified decision procedure `aac_reflexivity`. We define a reflexive tactic [4, 32, 76]: like in the previous chapter, this means that we implement the decision procedure as a Coq function over "reified" terms, which we prove correct inside the proof assistant. This step was actually quite challenging: to our knowledge, `aac_reflexivity` is the first Coq reflexive decision procedure that handles uninterpreted function symbols. (By contrast, existing reflexive decision procedures in Coq, like `ring`, work for a fixed structure and cannot deal with uninterpreted function symbols.) In addition to the non-trivial reification process, a particular difficulty comes from the (arbitrary) arity of these symbols. To overcome this problem in an elegant way, our solution relies on a dependently typed syntax for reified terms.

2.2 User interface

In this section, we shall see the user interface of our tools and in particular how we declare new operations

Declaring A/AC operations. We rely on type-classes [147] to declare the properties of functions and A/AC binary operations. This allows the user to extend both the decision procedure and the matching algorithm with new A/AC operations or units in a very natural way. Moreover, this is the basis of our reification mechanism (see §2.5.3).

The classes corresponding to the various properties that can be declared are given in Fig. 2.1: being associative, commutative, and having a neutral element. Basically, a user only needs to provide instances of these classes in order to use our tactics in a setting with new A or AC

Figure 2.2 Example instances.

```
Instance plus_A: Associative eq plus.
Instance plus_C: Commutative eq plus.
Instance plus_U: Unit eq plus 0.

Instance app_A X: Associative eq (app X).
Instance app_U X: Unit eq (app X) (nil X).

Instance and_A: Associative iff and.
Instance and_C: Commutative iff and.
Instance and_U: Unit iff and True.
Instance and_P: Proper (iff ⇒ iff ⇒ iff) and.
Instance not_P: Proper (iff ⇒ iff) not.
```

operations. These classes are parameterised by a relation, `R`, so that one can work with an arbitrary equivalence relation.

Fig. 2.2 also contains examples of instances. Note that polymorphic values (`app`, `nil`) are declared in a straightforward way; for propositional connectives (`and`, `not`), we also need to show that they preserve equivalence of propositions (`iff`), since this is not Leibniz equality. We use for that the standard `Proper` type-class (when the relation `R` is Leibniz equality, these instances are inferred automatically). Of course, while we provide these instances, more can be defined by the user.

Example usage. The main tactics we provide are `aac_rewrite`, to rewrite modulo A/AC, and `aac_reflexivity` to decide an equality modulo A/AC. Here is an simple example where we use both of them in the context of operations on sets:

```
H1: ∀ x y z, x∩y ∪ x∩z = x∩(y∪z)
H2: ∀ x y, x∩x = x
a, b, c, d: set
======================
(a∩c ∪ b∩c∩d) ∩ c = (a ∪ d∩b) ∩ c
```

```
Proof.
 aac_rewrite H1; (* c ∩ (a ∪ b∩d) ∩ c = ... *)
 aac_rewrite H2; (* c ∩ (a ∪ b∩d)     = ... *)
 aac_reflexivity.
Qed.
```

As expected, we provide variations to rewrite using the hypothesis from right to left, or in the right-hand side of the goal.

Listing instances. There might be several ways of rewriting a given equation: several subterms may match, so that the user might need to select which occurrences to rewrite. The situation can be even worse when rewriting modulo AC: unlike with syntactical matching, there might be several ways to instantiate the pattern so that it matches a given occurrence modulo AC. (E.g., matching the pattern $x + y + y$ at the root of the term $a + a + b + b$ yields two substitutions: $\{x \mapsto a + a; y \mapsto b\}$ and the symmetrical one – assuming there is no neutral element.) To help the user, we provide an additional tactic, `aac_instances`, to display the possible occurrences together with the corresponding instantiations. The user can then use the tactic `aac_rewrite` with the appropriate options.

2.3 Some complete examples

In this section, we do not aim at showing impressive examples of use of AC in proofs, like the examples that were studied in the context of Maude [48]. We shall rather see how these tactics apply in concrete examples that arise in day-to-day Coq proofs that occur in various contexts.

2.3.1 Homogeneous binary relations

Our first example takes place in the context of abstract rewriting that was used in the introduction of the previous chapter. We define in Fig. 2.3 some notations for the usual operations on

Figure 2.3 Homogeneous binary relations

```
Require Import Relations.
Import Instances.Relations.
Variable A : Type.
Notation "R*" := (clos_refl_trans A R).
Notation "R+S" := (union A R S).
Notation "R·S" := (comp A R S).
Notation "R ≤ S" := (inclusion A R S).
Notation "R ≡ S" := (same_relation A R S).
Notation "1" := (@eq A).

Lemma plus_destruct R S T : R ≤ T → S ≤ T → R + S ≤ T.
Lemma star_induction R S : 1 + R·S ≤ S → R* ≤ S.
Lemma star_trans R: (R* · R* ) ≡ (R*).
Lemma one_leq_star R : 1 ≤ R *.
Lemma star_absorb R : R · R* ≤ R*.
Lemma distr_l R S T : ((R + S) · T) ≡ (R·T + S·T).
```

homogeneous binary relations from Coq's standard library, as well as some lemmas. We import the standard instances that declare these operations as associative and commutative (depending of the operation) from a module of our library, `Instances.Relations`.

Then, the following proof that weak-confluence entails the Church-Rosser property follows the pen-and-paper mathematical proof.

```
Variables R S: relation A.
Theorem WeakConfluence_is_ChurchRosser (H: R · S* ≤ S* · R*): (R+S)* ≤ S* · R*.
Proof.
apply star_induction; apply plus_destruct.
(*  1 ≤ S* · R* *)
do 2 rewrite ← one_leq_star.
(*  1 ≤ 1 · 1 *)
• aac_reflexivity.
(* (R + S) · (S* · R*) ≤ S* · R* *)
rewrite distr_l; apply plus_destruct.
  (* R · (S* · R*) ≤ S* · R* *)
  aac_rewrite H.
  (*  S* · R* · R* ≤ S* · R* *)
  • aac_rewrite (star_trans); reflexivity.
  (* S · (S* · R*) ≤ S* · R* *)
  • aac_rewrite (star_absorb); reflexivity.
Qed.
```

In this proof, the use of the tactic `aac_rewrite` makes the reordering of parentheses implicit. We only use the tactic `aac_reflexivity` in a trivial way to solve a sub-goal that only deals with the fact that the identity relation `eq` is the neutral element for the composition of relations. (Note that we could have used the tactic `kleene_reflexivity` from the previous chapter in three places, instead of the lines starting with •.)

2.3.2 Arithmetic in Z

While Coq features some decision procedures like `nsatz` that decides systems of equations over commutative rings or `lia` that solves linear arithmetic goals over $\mathbb{Z}$, these tactics do not provide much help when they cannot solve the goal. Therefore, our rewriting modulo AC tactic complement these decision procedures, to ease the proof-engineering.

Our second example takes place in the context of proofs about hardware circuits that implement arithmetic functions. This involves a lot of reasoning about machine-integers of fixed precision. For instance, the correctness proof of an adder circuit from the next chapter requires to prove the goal given in Fig. 2.4.

Note that in this setting, our rewriting tactic uses the AC properties of the multiplication and the addition inside the arguments of `mod`, an uninterpreted symbol. (As a matter of fact, the

Figure 2.4 Arithmetic in $\mathbb{Z}$

```
Theorem Zmult_plus_distr: ∀ n m p : Z, (n + m) * p = n * p + m * p.
Theorem Zmod_plus_weak: ∀ a b : Z, (a + b) mod a = b mod a
Theorem Zmod_small: ∀ a n : Z, 0 ≤ a < n → a mod n = a.

Variables low k: Z. Variables n m: nat.
Hypothesis Hlow: 0 ≤ low < 2^n.
Hypothesis Hk:  0 ≤ k < 2^m.
Goal ((2^m + k) * 2^n + low) mod (2^n * 2^m) = ((2^m + k) mod 2^m) * 2^n + low.
Proof.
  aac_rewrite Zmult_plus_distr.
  aac_rewrite Zmod_plus_weak in_right.
  (* (2^m * 2^n + k * 2^n + low)  mod (2^n * 2^m) = k  mod 2^m * 2^n + low  *)
  rewrite (Zmod_small k 2^m Hk).
  (*(2^m * 2^n + k * 2^n + low)  mod (2^n * 2^m) = k * 2^n + low*)
  aac_rewrite Zmod_plus_weak.
  (* (2^n * k + low)  mod (2^n * 2^m) = k * 2^n + low  *)
  rewrite Zmod_small.
  (* 2^n * k + low = k * 2^n + low  *)
  aac_reflexivity.
  ...  (* proof of 0 ≤ 2^n * k + low < 2^n * 2^m  *)
Qed.
```

`mod` operation is not handled by other current Coq tactics such as `lia`, `omega`, ...) Generally, `aac_rewrite` allows for more economical design of libraries: for instance, the `Zmod_plus_weak` lemma from Fig. 2.4 usually comes in two flavours.

```
Theorem Zmod_plus_weak_r: ∀ a b : Z, (a + b) mod a = b mod a
Theorem Zmod_plus_weak_l: ∀ a b : Z, (b + a) mod a = b mod a
```

Since these two lemmas collapse modulo AC, we argue that a systematic use of our tactics could simplify the design of libraries (removing useless lemmas). Note that, in our example in Fig. 2.4, this means that we may use `rewrite` instead of the first two occurrences of `aac_rewrite`, provided that we pick the right flavour of each lemma. Yet, we consider that a consistent use of `aac_rewrite` is an elegant alternative to the profusion of lemma to remember.

Note that we could also enrich libraries with more powerful theorems. For instance, even if the following one is stated in Coq's standard library: it is seldom applicable as it is, using Coq's `rewrite` tactic.

```
Theorem Zmod_full: ∀ a b c, (c*a + b) mod a = b mod a.
```

2.3.3 Operations on bit-vectors

Our third example takes place in the context of proofs about bit-vectors (fixed-length list of booleans). We assume the definition of bit-vectors and the existence of some basic operations on them in Fig. 2.5. In particular, notice the `map2` function that takes as argument a binary boolean function f and generates a function on pairs of vectors of size n by applying f pointwise. The instances on the right-hand side of Fig. 2.5 illustrate the fact that if the underlying function is AC or A, then so is the function operating on vectors. Moreover, if the underlying function has a neutral element, then the function operating on vectors has also a neutral element (which is a constant vector).

Figure 2.5 Operations on bit-vectors

```
Import Instances.Bool.
Variable bvector : nat → Type.
Variable map2: (bool → bool → bool) →
  ∀ n, bvector n → bvector n → bvector n.
Variable map: (bool → bool) →
  ∀ n, bvector n → bvector n.
Variable const : bool → ∀ n, bvector n.

Notation "u ∧ v" := (map2 andb _ u v).
Notation "u ∨ v" := (map2 orb _ u v).
Notation "u ⊕ v" := (map2 xorb _ u v).
Notation "¬ u" := (map negb _ u).

Instance map2_A {f} {A: Associative eq f} {n}:
@Associative (bvector n) (eq) (map2 f n).

Instance map2_C {f} {C: Commutative eq f} {n}:
@Commutative (bvector n) (eq) (map2 f n).

Instance map2_U {f} {u:bool} {C: Unit eq f u} {n}:
@Unit (bvector n) (eq) (map2 f n) (const u n ).


Notation "0" := (const true _).
Notation "1" := (const false _).
```

Figure 2.6 Some lemmas about bit-vectors

```
Lemma bv_and_or_distr : ∀ n (v1 v2 v3 : bvector n), v1 ∧ (v2 ∨ v3) = (v1 ∧ v2) ∨ (v1 ∧ v3).
Lemma bv_and_neg_false : ∀ n (v : bvector n), v ∧ ¬ v = 0.
Lemma bv_and_false : ∀ n (v : bvector n), 0 ∧ v = 0.
Lemma bv_and_empty : ∀ n (u v : bvector n), u ∧ v = 0 → u ∧ ¬ v = u.
```

We present some lemmas about these operations on bit-vectors in Fig. 2.6 where we leave the operators priority implicit. (Note that these lemmas look exactly like their counterparts on booleans, thanks to the property we mentioned above.) We proceed to prove a typical goal that arises in the context of randomised algorithms about boolean bit-vectors [11].

```
Variable n : nat. Variable K x y : bvector n.
Hypothesis H : y ∧ K = 0.
Goal ((K ∧ ¬x) ∨ y) ∧ ¬K = y.
  aac_rewrite (bv_and_or_distr n). (* (¬K ∧ y) ∨ (¬K ∧ (¬x ∧ K)) = y *)
  aac_rewrite (bv_and_neg_false n). (* (0 ∧ ¬x) ∨ (¬K ∧ y) = y *)
  rewrite bv_and_false. (* 0 ∨ (¬K ∧ y) = y *)
  aac_rewrite (bv_and_empty n y K); [ reflexivity | assumption ].
Qed.
```

While this goal amounts to a Boolean problem (which would be solved likewise), we want to draw the attention to the fact that the use of our tactics seems to ease the proof-engineering in this context.

2.4 Definitions

We have presented the user-interface of our tactics, and some complete examples. In this section, we review some basic definitions of term rewriting that will be used in the later sections. (Note that the amount of mathematical background required here is much lighter than in the previous chapter.)

Definition 8. *A signature* Σ *is a set of* function symbols *where each* $f \in \Sigma$ *is associated with an arity* $n \in \mathbb{N}$*. The elements of* Σ *with arity* 0 *are called* constants*.*

In the following, we let $f, g, h, \ldots$ range over function symbols, reserving letters $a, b, c, \ldots$ for constants.

Definition 9. *Let Σ be a signature and X be a set of* variables *such that $\Sigma \cap X = \emptyset$. The set $T(\Sigma, X)$ of Σ-terms over X is inductively defined as*

- $X \subseteq T(\Sigma, X)$
- *for all $n \geq 0$, for all $f \in \Sigma$ with arity n, and for all $t_1, \ldots, t_n \in T(\Sigma, X)$, we have $f(t_1, \ldots, t_n) \in T(\Sigma, X)$.*

That is, variables are terms, and application of function symbols to terms yields terms. We denote terms that do not contains variables by $T(\Sigma)$. We shall emphasise the presence of variables in terms as follows: unless stated otherwise, a term *is an element of $T(\Sigma)$, while a* pattern *is an element of $T(\Sigma, X)$ (that may contain variables).*

Definition 10. *Let Σ be a signature, and X a set of variables. A* substitution *is a partial function of type $X \to T(\Sigma)$ that maps variables to terms.*

In the following, we let σ range over substitutions. Substitutions are extended to partial functions from patterns to terms in $T(\Sigma)$ as expected. To simplify notations, we shall use σ to denote both the substitution and its extension. Moreover, we shall use the usual postfix notation for substitutions, writing $p\sigma$ instead of $\sigma(p)$.

Definition 11 (Identities)**.** *Let Σ be a signature, and X a set of variables. An identity is a pair $(s, t) \in T(\Sigma, X) \times T(\Sigma, X)$. Identities will be written as $\langle s, t \rangle$.*

Definition 12 (Equational theory)**.** *The equational theory generated by an arbitrary set E of identities corresponds to the equations that may be derived using the following inference rules:*

$$\frac{\langle s, t \rangle \in E}{E \vdash s \equiv t}$$

$$\frac{E \vdash s \equiv t \qquad E \vdash t \equiv u}{E \vdash s \equiv u} \qquad \frac{E \vdash s \equiv t}{E \vdash t \equiv s} \qquad \frac{}{E \vdash s \equiv s}$$

$$\frac{E \vdash s \equiv t}{E \vdash s\sigma \equiv t\sigma}$$

$$\frac{E \vdash s_1 \equiv t_1 \qquad \ldots \qquad E \vdash s_n \equiv t_n}{E \vdash f(s_1, \ldots, s_n) \equiv f(t_1, \ldots, t_n)}$$

In the following, binary function symbols (written with an infix symbol, $\diamond$) can be associative (axiom $A_\diamond$) and optionally commutative (axiom $C_\diamond$); these symbols may be equipped with a left and right unit u (axiom $U_{u,\diamond}$):

$$A_\diamond : x \diamond (y \diamond z) \equiv (x \diamond y) \diamond z \qquad C_\diamond : x \diamond y \equiv y \diamond x \qquad U_{u,\diamond} : x \diamond u \equiv x \wedge u \diamond x \equiv x$$

We use $\oplus_i$ (or $\oplus$) for associative-commutative symbols (AC), and $\otimes_i$ (or $\otimes$) for associative only symbols (A). We denote by $\equiv_{AC}$ the equational theory generated by these axioms on $T(\Sigma)$. For instance, in a non-commutative semi-ring $(+, *, 0, 1)$, $\equiv_{AC}$ is generated by

$$E = \{A_+, C_+, A_*, U_{1,*}, U_{0,+}\}.$$

As was hinted at in the introduction, we shall prove that certain equalities are derivable from the rules of equational reasoning and the axioms of AC for some operations. This is a particular instance of the *word problem*.

Definition 13 (Word problem)**.** *Let Σ be a signature, X a set of variables, and E a set of identites. The word problem for E is the problem of deciding $E \vdash s \equiv t$ for arbitrary $s, t \in T(\Sigma, X)$. In the following, we focus on the* ground word problem*, the word problem restricted to terms s and t that do not contain variables.*

We now turn to *equational unification*: the unification of two terms, w.r.t a set of identities.

Definition 14. *Let Σ be a signature, X a set of variables, and E a set of identites. A E-unification problem is a finite set of equations $S = \{\langle s_1, t_1\rangle, \ldots, \langle s_n, t_n\rangle\}$ between terms in $T(\Sigma, X)$. A solution to this problem is a substitution σ such that*

$$\forall i, 1 \leq i \leq n, E \vdash \sigma(s_i) \equiv \sigma(t_i)$$

In the following, we are only interested in a restricted case of equational unification: the matching problem.

Definition 15. *Let Σ be a signature, X a set of variables, and E a set of identites. Let p be a pattern (i.e., $p \in T(\Sigma, X)$) and t a term of $T(\Sigma)$. A solution to the matching problem (denoted $p \leq_E t$) is a substitution σ such that $E \vdash \sigma(p) \equiv t$.*

Finally, the following folk theorem is an obvious prerequisite for the decision procedure for equality modulo AC, and the existence of an algorithm for matching modulo AC.

Theorem 13. *The ground word problem for $\equiv_{AC}$ and the matching modulo AC problem are decidable.*

2.5 Deciding equality modulo AC

In this section, we describe the stand-alone `aac_reflexivity` tactic, which decides equality modulo AC in Coq. Recall that this tactic is extensible through the definition of new type-class instances, and deals with uninterpreted homogeneous function symbols of arbitrary arity. For the sake of clarity, we initially omit the case where binary operations have units. We will describe this case in a second time in §2.7.1.

A two-level approach. We use the so called 2-level approach [14]: we define an inductive type for terms, `T`, and a function `eval`: `T` → `X` that maps reified terms to user-level terms, in some type `X` equipped with an equivalence relation `R`, which we sometimes denote by ≡. This allows us to reason and compute on the syntactic representation of terms, whatever the user-level model.

We follow the usual practice which consists in reducing equational reasoning to the computation and comparison of canonical forms. We define a function `norm`: `T` → `T` that computes the canonical form of a given reified term and we prove it correct: that is, it is possible to derive a proof of equality between the images of a term and its canonical form, via the evaluation function `eval`.

```
Definition norm: T → T := ...
Lemma eval_norm: ∀ u, eval (norm u) ≡ eval u.
Theorem decide: ∀ u v, compare (norm u) (norm v) = Eq → eval u ≡ eval v.
```

In the terminology of [14], this is what is called the *autarkic way*: the verification of the equality is performed inside the proof-assistant, using the conversion rule. To prove `eval u ≡ eval v`, it suffices to apply the theorem `decide` and to let the proof-assistant check by computation that the premise `compare (norm u) (norm v) = Eq` holds.

Note that the formalisation of the completeness of `compare` is not required in the above `decide` theorem: this is usual when implementing such decision procedures [76].

As usual, the implementation of decision procedure needs to meet two objectives. First, the normalisation function (`norm`) must be efficient, and this dictates some choices for the representation of terms. Second, the decision procedure can only be applied to terms in the image of the evaluation function (`eval`): therefore, there is an incentive to have an expressive representation of terms in the reified setting. However, we shall syntactically rule out ill-formed terms (that is, terms whose image by `eval` is ill-typed), to keep `eval` simple (that is, not requiring to handle "run-time errors" in any way).

2.5.1 Representation of reified terms

Parametrisation. We emphasized that existing reflexive decision procedures in Coq, like `ring`, work with a fixed signature. For instance, a ring structure is parametrised by two associative and commutative binary operations $\oplus$ and $\otimes$, such that $\oplus$ is distributive over $\otimes$; and by two constants 0 and 1 that are neutral elements for $\oplus$ and $\otimes$. Declaring a new ring requires to prove that a given ring signature satisfies the ring axioms. Then, each time the `ring` tactic is invoked in a given ring structure, it builds an environment that maps a set of indices to sub-terms that are neither ring constants, nor ring operations applied to other terms. As an example, the following Coq expression

$$\texttt{Zmod (a+b) a} + \texttt{b} = \texttt{b} + \texttt{Zmod (b+a) a}$$

would be reified as

$$\texttt{var 1} + \texttt{var 3} = \texttt{var 3} + \texttt{var 2}$$

in the environment

$$\{1 \mapsto \texttt{Zmod (a+b) a}; 2 \mapsto \texttt{Zmod (b+a) a}; 3 \mapsto \texttt{b}\}.$$

In this case, `ring` would fail to prove the equation: the reified terms have different variables (that is, indexes) on each side.

By contrast, each time the tactic is invoked, we compute the signature, which is then used as a parameter of the reified terms (the "reification environment"). To be more precise, the reification environement is the disjoint sum of a signature for the binary associative (and optionaly commutative) symbols and a signature for the function symbols (including the constants). Note that the only constraint we put on the second signature is that function symbols must be homogeneous: the type of the arguments and the type of the result of the function must be the same. Informally, the complete environment would be

$$\{1 \mapsto \texttt{Zmod}; 2 \mapsto \texttt{a}; 3 \mapsto \texttt{b}\} \cup \{4 \mapsto \texttt{+}\}.$$

Packaging the reification environment. We need Coq types to package information about binary operations and uninterpreted function symbols. They are given in Fig. 2.7, where `respectful` is the definition from Coq standard library for declaring proper morphisms. We first define functions to express the fact that n-ary functions are proper morphisms. An "uninterpreted symbol package" contains the arity of the symbol, the corresponding function, and the proof that this is a proper morphism. A "binary package" contains a binary operation together with the proofs that it is a proper morphism, associative, and possibly commutative (we use the type-classes from Fig. 2.1).

The fact that symbols arity is stored in the package is crucial: by doing so, we can use standard finite maps to store all function symbols, irrespective of their arity. More precisely,

Figure 2.7 Types for symbols.

```
(* type of n-ary homogeneous functions *)
Fixpoint type_of (X: Type) (n: nat): Type :=
  match n with O ⇒ X | S n ⇒ X → type_of X n end.

(* relation to be preserved by n-ary functions *)
Fixpoint rel_of (X: Type) (R: relation X) (n: nat): relation (type_of X n) :=
  match n with O ⇒ R | S n ⇒ respectful R (rel_of n) end.

Module Bin.
 Record pack X R := {
  value: X → X → X;
  compat: Proper (R ⇒ R ⇒ R) value;
  assoc: Associative R value;
  comm: option (Commutative R value) }.
End Bin.

Module Sym.
 Record pack X R := {
  ar: nat;
  value: type_of X ar;
  compat: Proper (rel_of X R ar) value }.
End Sym.
```

we use two environments, one for uninterpreted symbols and one for binary operations; both of them are represented as non-dependent functions from a set of indices to the corresponding package types:

```
Variables (X: Type) (R: relation X).
Variable e_sym: idx → Sym.pack X R.
Variable e_bin: idx → Bin.pack X R.
```

(The type `idx` of indices is an alias for `positive`, the set of binary positive numbers; like in the previous chapter, this allows us to define the above functions efficiently, using positive maps).

Syntax of reified terms. We now turn to the concrete representation of terms. The first difficulty is to choose an appropriate representation for AC and A nodes, to avoid manipulating binary trees. As it is usually done, we flatten these binary nodes using variadic nodes. Since binary operations do not necessarily come with a neutral element, we use non-empty lists (resp. non-empty multi-sets) to reflect the fact that A operations (resp. AC operations) must have at least one argument.

Note that we could require A operations to have at least two arguments but this would slightly obfuscate the code. Moreover, this does not fit nicely with multi-sets, and thus, would prevent some code sharing. In the same fashion, we could ensure in the data-type that A/AC symbols cannot have a term headed by the same symbol as argument. For the sake of simplicity, we enforce this constraint only on the canonical forms of terms.

The second difficulty is to prevent ill-formed terms, like "`abs` 16 64", where a unary function is applied to more than one argument. One could define a predicate stating that terms are well-formed [51], and use this extra hypothesis in later reasonings. We found it nicer to use dependent types to enforce the constraint that symbols are applied to the right number of arguments, according to their declared arity: it suffices to use vectors of arguments rather than lists.

The resulting data-type for reified terms is given in Fig. 2.8. This definition allows for a simple and total implementation of `eval`, given on the right-hand side. For uninterpreted symbols, recall the trick from our introductory example which consists in using an accumulator to store the successive partial applications, until the function gets all its arguments.

As expected, this syntax allows us to reify arbitrary user-level terms. For instance, take `(a*S(b+b))-b`. We first construct the following environments where we store information about

Figure 2.8 Data-type for terms, and related evaluation function.

```
(* non-empty lists/multisets *)
Inductive nelist A :=
| nil: A → nelist A
| cons: A → nelist A → nelist A.

Definition nemset A := nelist (A*positive).

(* reified terms *)
Inductive T: Type :=
| bin_ac: idx → nemset T → T
| bin_a : idx → nelist T → T
| sym: ∀ i, vT (Sym.ar (e_sym i)) → T
with vT: nat → Type :=
| vnil: vT 0
| vcons: ∀ n, T → vT n → vT (S n).

Fixpoint eval (u: T): X :=
match u with
| bin_ac i l ⇒ let o:=Bin.value (e_bin i) in
   nefold_map o (fun(u,n)⇒copy o n (eval u)) l
| bin_a i l ⇒ let o:=Bin.value (e_bin i) in
   nefold_map o eval l
| sym i v ⇒ xeval v (Sym.value (e_sym i))
end
with xeval {i} (v: vT i): Sym.type_of i→X :=
match v with
| vnil ⇒ (fun f ⇒ f)
| vcons _ u v ⇒ (fun f ⇒ xeval v (f (eval u)))
end.
```

all atoms:

e_sym	e_bin
1 ⇒ ⟪ ar := 1; value := S; compat := ... ⟫	1 ⇒ ⟪ value := plus; compat := ... ; assoc := _ ; comm := Some ... ⟫
2 ⇒ ⟪ ar := 0; value := a; compat := ... ⟫	_ ⇒ ⟪ value := mult; compat := ... ; assoc := _ ; comm := None ⟫
3 ⇒ ⟪ ar := 0; value := b; compat := ... ⟫	
_ ⇒ ⟪ ar := 2; value := minus; compat := ... ⟫	

These environment functions are total: they associate a semantic value to indices that might be considered as out-of-the-bounds. This makes using it convenient and easy to reason about: there is no need to return an option or a default value in case undefined symbols are encountered in the term[1]. However, this requires environments to contain at least one value. We can then build a reified term whose evaluation in the above environments reduces to the starting user-level terms:

```
Let t := sym 4 [[bin_a 2 [(sym 2 [[]]); (sym 1 [[bin_ac 1 [(sym 3 [[]],1);( sym 3 [[]],1) ]])]; sym 3 [[]]].
Goal eval e_sym e_bin t = (a*S(b+b))−b.
  reflexivity.
Qed.
```

Note that we cannot use two environments `e_bin_a` and `e_bin_ac`: since each environment requires a default value, it would not be possible to reify terms (and to use our tactics) in a setting with only A or only AC operations. Moreover, having a single environment for all binary operations makes it easier to handle neutral elements (see §2.7.1).

2.5.2 The algorithm and its proof

Canonical forms are computed as follows: terms are recursively flattened under A/AC nodes and arguments of AC nodes are sorted with respect to a total order. We give excerpts of this algorithm below, focusing on AC nodes (the treatment of A nodes is somewhat simpler). First, we define smart constructors that prevent building unary AC or A nodes.

```
Definition bin_ac' i (u: nemset T): T := match u with nil (u,1) ⇒ u  | _ ⇒ bin_ac i u end.
Definition bin_a'  i (u: nelist T): T := match u with nil (u)   ⇒ u  | _ ⇒ bin_a  i u end.
```

[1]Note that "undefined" symbols shall not occur in the terms built using our tactics, but that we nevertheless have to handle them somehow in the evaluation.

Figure 2.9 Normalisation of multi sets

```
Definition extract_ac i (s: T): nemset T :=
  match s with bin_ac j m when i = j ⇒ m | _ ⇒ [s,1] end.

(* taking n copies of a given non-empty multiset *)
Definition copy_mset n (l: nemset T): nemset T :=
  match n with xH ⇒ l | _ ⇒ nelist_map (fun (v,m) ⇒ (v,n*m)) l end.

Fixpoint nefold_map {A B : Type} (comb: B → B → B) (f : A → B) : nelist A → B := ...

Definition norm_mset norm i (u: nemset T): nemset T :=
  nefold_map merge_sort (fun (x,n) ⇒ copy_mset n (extract_ac i (norm x))) u
```

Then, we define a function `norm_msets norm i` that normalises (w.r.t. `norm`) and sorts a multi-set, ensuring that none of its children starts with an AC node with index `i`. Similarly, we define a function `norm_list` that normalises and flattens a non-empty list. Finally, the canonisation function is given below:

```
Fixpoint norm (u: T): T := match u with
| bin_ac i l ⇒  if is_commutative e_bin i then bin_ac' i (norm_msets norm i l)  else u
| bin_a i l ⇒  bin_a' i (norm_lists norm i l)
| sym i l ⇒  sym i (vnorm l)
end
with vnorm {i} (l: vT i): vT i :=  match l with
| vnil ⇒ vnil
| vcons _ u l ⇒ vcons (norm u) (vnorm l) end.
```

Remark that `norm` depends on the information contained in the environments: indeed, the look-up `is_commutative e_bin i` in the definition of `norm` is required to make sure that the binary operation `i` is actually commutative (remember that we need to store A and AC symbols in the same environment, so that we might[2] have AC nodes whose corresponding operation is not commutative). Similarly, to handle neutral elements (§2.7.1), we will rely on the environment to detect whether some value is a unit for a given binary operation.

Correctness and completeness. We prove that the normalisation function is sound w.r.t to the evaluation function. That is, the images through `eval` of a reified term and its canonical form are equal w.r.t. the user-level equivalence relation ≡ (note that `eval` and `norm` implicitly depend on `e_bin` and `e_sym`).

```
Theorem eval_norm: ∀ u, eval (norm u) ≡ eval u.
```

The proof of this theorem relies on the above defensive test against ill-formed terms, the look-up made by `norm` before going through an AC node. Since invalid AC nodes are left intact, we do not need the missing commutativity hypothesis when proving the correctness of `norm`. It is then routine to prove the correctness of the decision procedure.

```
Theorem decide (u v: T) : compare (norm u) (norm v) = Eq → eval u ≡ eval v.
```

We did not prove completeness in Coq. First, this is not needed to get a sound tactic. Second, this proof would be quite verbose: we should axiomatize what is equality modulo AC on reified terms and prove that reified terms that are provably equal modulo AC are equal via

[2] Again, this shall not occur in the terms built using our tactics, but we nevertheless have to handle this situation

the evaluation function. Third, we would not be able to completely prove the completeness of the resulting tactic anyway, since one cannot reason about the reification function in the proof-assistant [30, 76].

Efficiency. The dependently typed representation of terms we chose in order to simplify proofs does not preclude efficient computation. Technically, we rely on `vm_compute` [75] which performs strict evaluation. Thus, there is a slight overhead due to the passing of the extra arguments w.r.t. the untyped syntax with a well-formedness predicate. Algorithmically, the complexity of the procedure is dominated by the merging of sorted multi-sets, which relies on a linear comparison function. We did not put this decision procedure through an extensive testing; however, we claim that it returns instantaneously on practical examples.

Moreover, the size of the generated proof is linear with respect to the size of the starting terms. The proof term we generate is the application of the `decide` theorem to the type of the carrier `X`, the setoid relation we consider `R`, the reification environments `e_sym` and `e_bin`, the reified terms `u` and `v`, and a proof term for `Eq` = `Eq`. (Recall that the Coq has to check that the premise `compare u v` = `Eq` of the `decide` theorem is convertible to `Eq` = `Eq`.)

```
exact (decide X R e_sym e_bin u v (eq_refl comparison Eq))
```

Notice that the size of the reification environments is linear with respect to the size of the original terms, and that the reified terms are smaller than the original terms.

By contrast, using the tactic language to build a proof out of associativity and commutativity lemmas would usually yield a quadratic proof. That is, the proof term built by each invocation of the standard rewrite tactic with the associativity and commutativity lemmas contains explicitly the sub-terms that the lemmas are applied to, and the context in which this rewrite step takes place.

2.5.3 Reification

Following the reflexive approach to solve an equality modulo AC, it suffices to apply the above theorem `decide` (§2.5.2) and to let Coq compute. To do so, we still need to provide two environments `e_bin` and `e_sym` and two terms `u` and `v` whose evaluation is convertible with the user-level terms.

Type-class based reification. In the previous chapter, we relied on projections of type-classes fields to guess how to reify the terms. However, this would force the users to use our definitions and notations from the beginning, or to use ad-hoc tactics to make projections appear.

Instead, we let the users work with their own definitions, and we exploit type-classes to perform reification. The idea is to query the type-class resolution mechanism to decide whether a given subterm should be reified as an AC operation, an A operation, or an uninterpreted function symbol. Recall that the definition of the `Associative` class is as follows.

```
Class Associative (X: Type) (R: relation X) (op: X → X → X) := law_assoc: ...
```

We reify the goal with respect to a given carrier type `Y`, and a given setoid relation `S`: `relation Y`. Upon encoutering a term `f a b` whose head-symbol is of type `Y → Y → Y`, we query the type-class resolution mechanism for an instance of the class `Associative Y S f`. If the resolution succeeds, we register the binary symbol `f` as associative and we proceed to query for an instance of the class `Commutative Y S f`; otherwise, we stop the analysis of this symbol.

To be more precise, we use a two-phases analysis. The inference of binary A or AC operations takes place first, by querying for instances of the classes `Associative` and `Commutative` on all

binary applications that occur in the term, including terms like `f a b c`, which is a binary application of `f a` to two arguments `b` and `c`. (Note that we do not require `a` to be of the same type as `b` and `c` here, which makes it possible to handle parametrised associative operations like `List.app`.) The remaining difficulty is to discriminate whether other applications should be considered as a function symbol applied to several arguments, or as a constant. On an example, considering the application `f (a+b) (b+c) c`, it suffices to query for `Proper` instances in the following order:

1.	`Proper (R ⇒ R ⇒ R ⇒ R)`	`(f)`	?
2.	`Proper (R ⇒ R ⇒ R)`	`(f (a+b))`	?
3.	`Proper (R ⇒ R)`	`(f (a+b) (b+c))`	?
4.	`Proper (R)`	`(f (a+b) (b+c) c)`	?

The first query that succeeds tells which partial application is a proper morphism, and with which arity. The last query always succeeds since the relation `R` is reflexive: the inference of constants – symbols of arity 0 – is the catch-all case of reification. We then proceed recursively on the remaining arguments; in the example, if the second call is the first to succeed, we do not try to reify the first argument `(a+b)`: the partial application `f (a+b)` is considered as an atom.

Reification language. We use OCaml to perform this reification step. Using the meta-language OCaml rather than the meta-language of tactics Ltac is a matter of convenience: it allows us to use more efficient data-structures. For instance, we use hash-tables to memoise queries to type-class resolution, which would have been difficult to mimic in Ltac or using canonical structures [73].

The resulting code is non-trivial, but too technical to be presented here. Most of the difficulties come from the fact that we reify uninterpreted functions symbols using a dependently typed syntax, and that our reification environments contain dependent records: producing such Coq values from OCaml is tricky to get right, and is quite verbose.

Finally, we wrap up the previous elements in a tactic, `aac_reflexivity`, which automates this process, and solves equations modulo AC. To be more precise, given a goal whose head symbol is `R` with type `relation X`, the tactic:

1. checks that `R` is an equivalence relation
2. does a traversal of the terms to compute the set of AC/A symbols and the set of uninterpreted symbols that are morphisms for `R` and memoize this information in a map indexed by Coq terms;
3. does a second traversal of the terms, to reify them according to the signature that was inferred in the previous step;
4. builds a proof term that applies the `decide` theorem;
5. makes a call to `vm_compute` to make sure that the premise of `decide` holds.

Note that knowing the relation `R` is crucial to be able to infer the signature. However, it is possible to lift the requirement that `R` is an equivalence relation. We define a new type-class `AAC_lift` (see Fig. 2.10) that is used to find the equivalence relation w.r.t. which operations are A, AC or proper morphisms, starting from the relation appearing in the goal.

For instance, in the context of `nat`, one can define an instance `AAC_lift le eq`. This makes it possible to use the tactic `aac_reflexivity` or to make rewriting modulo AC steps on goals with head-symbol `le`: if the tactics find a proof of equality modulo AC/A in the setting of `eq`, this proof will be lifted to the relation `le`. More precisely, suppose we want to rewrite modulo

Figure 2.10 Lifting

```
Class AAC_lift X (R: relation X) (E : relation X) := {
  aac_lift_equivalence : Equivalence E;
  aac_lift_proper : Proper (E ⇒ E ⇒ iff) R
}.

(* simple instances, when we have a subrelation, or an equivalence *)
Context {X: Type} {R: relation X} {E: relation X} {HE: Equivalence E}.
Instance aac_lift_subrelation {HR: @Transitive X R} {HER: subrelation E R}: AAC_lift  R E .
Instance aac_lift_proper {HR: Proper (E ⇒ E ⇒ iff) R}: AAC_lift  R E.
```

AC the equation $H : \forall \overline{x}, p\overline{x} = q\overline{x}$ in the left-hand side of the goal $foo < bar$, we would build the following proof-tree:

$$\dfrac{\text{AAC\_REFLEXIVITY}\ \dfrac{}{p\sigma =_{AC} foo} \qquad \dfrac{q\sigma < bar}{p\sigma < bar}\ \text{REWRITE H}}{foo < bar}$$

2.6 Matching modulo AC

We now turn to the second part of our work: the implementation of the oracle in OCaml that gives the possible instantiations of the rewriting identity (i.e., a universally quantified equality). This mainly boils down to matching. Recall that solving a matching problem modulo AC consists in, given a pattern p and a term t, finding a substitution σ such that $p\sigma \equiv_{AC} t$. While this is a well-trodden area, with many known algorithms [51, 63, 90, 120], we present here a simple and concise one, which can be made efficient enough for our needs.

Naive algorithm. Matching modulo AC can easily be implemented non-deterministically. For example, to match a sum $p_1 \oplus p_2$ against a term t, it suffices to consider all possible decompositions of t into a sum $t_1 \oplus t_2$. If matching p_1 against t_1 yields a solution (a substitution), it can be used as an initial state to match p_2 against t_2, yielding a more precise solution, if any. To match a variable x against a term t, there are two cases depending on whether or not the variable has already been assigned in the current substitution. If the variable has already been assigned to a value v, we check that $v \equiv_{AC} t$. If this is not the case, the substitution must be discarded since x must take incompatible values. Otherwise, i.e., if the variable is fresh, we add a mapping from x to v to the substitution. To match an uninterpreted node $f(\overline{q})$ against a term t, it must be the case that t is headed by the same symbol f, with arguments $\overline{u}$; we just match $\overline{q}$ and $\overline{u}$ pointwise.

We write $\sigma : p \lhd t : \rho$ to state that, starting with a substitution σ, it is possible to match p against t, yielding the substitution ρ. The non-deterministic matching algorithm is described in Fig. 2.11, using inference rules. Note that this algorithm is fairly general: it just tries matching with all equivalent terms w.r.t. the given equational theory. However, we shall see that it can be implemented to be efficient enough for our purposes.

Monadic implementation. We implemented a monad for non-deterministic and backtracking computations, along the lines of [149]. In this setting, it is possible to express the fact that a

Figure 2.11 Matching algorithm.

$$\frac{t \equiv_{AC} t_1 \diamond_i t_2 \qquad \sigma_1 : p_1 \lhd t_1 : \sigma_2 \qquad \sigma_2 : p_2 \lhd t_2 : \sigma_3}{\sigma_1 : p_1 \diamond_i p_2 \lhd t : \sigma_3} \qquad \frac{t = f(\overline{t_i}) \qquad \sigma_i : p_i \lhd t_i : \sigma_{i+1}}{\sigma_0 : f(\overline{p_i}) \lhd t : \sigma_n}$$

$$\frac{\sigma(x) = v \qquad v \equiv_{AC} t}{\sigma : x \lhd t : \sigma} \qquad \frac{\sigma \# x}{\sigma : x \lhd t : \sigma \cup \{x \mapsto t\}}$$

Figure 2.12 Search monad primitives.

```
val (≫=): α m → (α → β m) → β m
val (≫|): α m → α m → α m
val return: α → α m
val fail: unit → α m
```

computation may return one, several or no values. Therefore, it allows for expressing programs in a concise and elegant fashion: the monad threads computations between functions that take one argument and return collections of results. Fig. 2.12 presents the primitive functions offered by this monad: $\gg=$ is a backtracking bind operation, while $\gg|$ is the non-deterministic choice.

We have an OCaml type for terms similar to the inductive type we defined for Coq reified terms: applications of A/AC symbols are represented using their flattened normal forms. From the primitives of the monad, we derive functions operating on terms (Fig. 2.13): the function `split_ac i` implements the non-deterministic split of a term t into pairs (t_1, t_2) such that $t \equiv_{AC} t_1 \oplus_i t_2$. If the head-symbol of t is $\oplus_i$, then it suffices to split syntactically the multi-set of arguments; otherwise, we return an empty collection. The function `split_a i` implements the corresponding operation on associative only symbols.

The matching algorithm proceeds by structural recursion on the pattern, which yields to the implementation presented in Fig. 2.14 (using an informal ML-like syntax). A nice property of this algorithm is that it does not produce redundant solutions, so that we do not need to reduce the set of solutions before proposing them to the user. While keeping this simple implementation, there is some room for minor improvements. Indeed, representing terms and patterns using normal forms enable some low-level optimisations: we can consider not only the head-symbol of the pattern, but also a few more symbols before engaging in a costly operation. For instance, if the pattern has the form $x \oplus p$, and if x is not fresh in the current substitution σ, it is obviously a poor solution to try all decompositions of the term: a better solution is to remove the factor $\sigma(x)$ from the term (modulo AC or A).

Correctness. Following [51], we could have attempted to prove the correctness of this matching algorithm. While this could be an interesting formalisation work *per se*, it is not necessary for our purpose, and could even be considered an impediment. Indeed, we implement the matching algorithm as an oracle, in an arbitrary language – that happens in our case to be the meta-language in which the proof assistant is written. Thus, we are given the choice to

Figure 2.13 Search monad derived functions.

```
val split_ac: idx → term → (term * term) m
val split_a : idx → term → (term * term) m
```

Figure 2.14 Backtracking pattern matching, using monads.

```
mtch (p1 ⊕i p2) t σ = split_ac i t ≫= (λ (t1,t2) → mtch p1 t1σ ≫= mtch p2 t2)
mtch (p1 ⊗i p2) t σ = split_a  i t ≫= (λ (t1,t2) → mtch p1 t1 σ ≫= mtch p2 t2)
mtch (f(p̄)) (f(ū)) σ = fold_2 (λ acc p t → acc ≫= mtch p t) (return σ) p̄ ū
mtch (var x) t σ when Subst.find σ x = None = return (Subst.add σ x t)
mtch (var x) t σ when Subst.find σ x = Some v = if v ≡AC t then return σ else fail()
```

use a free range of optimisations, and the ability to exploit all features of the implementation language. In any case, the prophecies of this oracle, a set of solutions to the matching problem, are verified by the reflexive decision procedure we implemented in §2.5.

Efficiency Let us comment on the above algorithm. First, note that the complexity of matching modulo AC with constants is NP-complete [16], even in the elementary [63] case where there is a single AC operator, variables and constants. Second, the usual efficient implementations of matching modulo AC involve algorithms for solving diophantine equation[52, 63]. Therefore, the simple algorithm we describe could probably not compete with the state-of-the-art algorithms on big terms [62]. However, it may be the case that the size of the terms and patterns which are likely to appear in a standard Coq proof do not justify such an heavy machinery. Indeed, we implemented a slightly refined version of the above algorithm, which performs quite well in practice. We may argue that terms that appear in day-to-day Coq proofs feature a lot of uninterpreted symbols, and comparatively few associative and commutative symbols.

2.7 Bridging the gaps

Combining both the decision procedure for equality modulo AC and the algorithm for matching modulo AC, we get the `aac_rewrite` tactic for rewriting modulo AC in Coq. To be more precise, given an universally quantified equality $H : \forall \overline{x}, p\overline{x} = q\overline{x}$ to be rewritten in the left-hand side of a goal $R\ t\ s$, the matching algorithm is used as an oracle to find a substitution σ such that $p\sigma \equiv_{AC} t$. Then, we make a transitivity step to the term $p\sigma$ and we use the standard `rewrite` tactic to rewrite H. (Note that it is crucial to make the transitivity step in such a way that p syntactically appear in the goal.) It is then possible to close the first goal generated by the transitivity step using `aac_reflexivity`. We now turn to lifting some simplifying assumptions that we made in the previous sections.

2.7.1 Neutral elements

The decision procedure from §2.5 and the matching algorithm from §2.6 were defined in a context where AC or A operations did not have neutral elements. However, adding support for neutral elements (or "units") is of practical importance:

- to let the `aac_reflexivity` tactic decide more equations (like $a + \max(0, b * 1) = a + b$);
- to avoid requiring the user to normalise terms manually before performing rewriting steps (e.g., to rewrite using $\forall x, x + x = x$ in the term $a * b + b * a * 1$);
- to propose more solutions to pattern matching problems (consider rewriting the identity $\forall xy, x \cdot y \cdot x^{\perp} = y$ in $a \cdot (b \cdot (a \cdot b)^{\perp})$, where $\cdot$ is an associative only operation with a neutral element u: the variable y should be instantiated with the neutral element u).

Therefore, we shall refine the signature we work with to let some AC or A symbols have units. We proceed to examine the necessary refinements for the various parts of our setting.

Figure 2.15 Additional environment for terms with units.

```
Variable e_bin: idx → Bin.pack X R

Record binary_for (u: X) := {
  bf_idx: idx;
  bf_desc: Unit R (Bin.value (e_bin bf_idx)) u .

Record unit_pack := {
 u_value: X;
 u_desc: list (binary_for u_value) }.

Variable e_unit: idx → unit_pack.
```

Extending the pattern matching algorithm. Matching modulo AC with units does not boil down to pattern matching modulo AC against a normalised term: $a \cdot b \cdot (a \cdot b)^{\perp}$ is a normal form and the algorithm of Fig. 2.14 would not give solutions with the pattern $x \cdot y \cdot x^{\perp}$. The patch is straightforward: it suffices to let the non-deterministic splitting functions (Fig. 2.13) use the neutral element possibly associated with the given binary symbol. For instance, calling `split_a` on the previous term would return the four pairs $\langle u, a \cdot b \cdot (a \cdot b)^{\perp} \rangle$, $\langle a, b \cdot (a \cdot b)^{\perp} \rangle$, $\langle a \cdot b, (a \cdot b)^{\perp} \rangle$, and $\langle a \cdot b \cdot (a \cdot b)^{\perp}, u \rangle$, where u is the neutral element of $\cdot$.

Extending the syntax of reified terms. The obvious idea is to replace non-empty lists (resp. multi-sets) by lists (resp. multi-sets) in the definition of terms in Fig. 2.8. This has two drawbacks. First, unless the evaluation function (Fig. 2.8) becomes a partial function, every A/AC symbol must then be associated with a unit (which precludes, e.g., `min` and `max` to be defined as AC operations on relative numbers). Second, two symbols cannot share a common unit, like 0 being the unit of both `max` and `plus` on natural numbers: we would have to know at reification time how to reify 0 – is it an empty AC node for `max` or for `plus`?

Instead, we add an extra constructor for units to the data-type of terms, and a third environment to store all units together with their relationship with binary operations. The actual definition of this third environment requires a more clever crafting than the other ones. The starting point is that a unit is nothing by itself, it is a unit for some binary operations. This connection must be tracked, otherwise, the reified data-type become meaningless: if in the reified world, one forgets that 0 is a unit for `plus`, we cannot prove that the normalisation functions preserve equality. Thus, the type of the environment for units has to depend on the `e_bin` environment. This type is given in Fig. 2.15. The record `binary_for u` stores the connection between a binary operation (pointed to by its index `bf_idx`) and the constant `u`. Then, each unit is bundled with the list of operations it is a unit for (`unit_pack`): like for the environment `e_sym`, these dependent records allow us to use plain, non-dependent maps.

In the end, the syntax of reified terms depends[3] only on the environment for uninterpreted symbols (`e_sym`), to ensure that arities are respected and the environment for units (`e_unit`) depends on the environment for binary operations (`e_bin`).

Extending the decision tactic. Updating the Coq normalisation function to deal with units is fairly simple but slightly verbose. Like we used the `e_bin` environment to check that `bin_ac` nodes actually correspond to commutative operations, we exploit the information contained in `e_unit` to detect whether a unit is a neutral element for a given binary operation. Normalisation of non-empty multi-sets and non-empty lists is straightforward: it suffices to flatten (resp. flatten and sorts) the arguments of associative (resp. associative and commutative) symbols, taking care of the potential collapses that occur when a given operation is applied only to copies of its unit.

[3] as in "dependent types"

Extending the reification mechanism. The above modifications are comparatively simpler w.r.t the modifications that must be done in the OCaml reification code. While it was already quite technical, it becomes even more complicated. A practical consideration is that calling type-class resolution on all constants of the goal, to get the list of binary operations they are a unit for, would be too costly.

Indeed, the presence of units is the actual reason for the two passes of inference (See §2.5.3). That is, we perform a first pass on the goal, where we infer all AC/A operations, and for each of these, we query the type-class inference mechanism for instances of the class `Unit`. Then, we construct the reified terms in a second pass, using the previous information to distinguish units from regular constants. However, if the two phases were mixed, we may end up reifying would-be units as constants. (Consider for instance the expression 1+`b*b` in which one wants to rewrite `H`: ∀ `x y`, `x + x*y*y`: the expressions 1 may have to be reified before encountering the AC operation ∗ it is a unit for.)

In the end, the OCaml code for the reification (including the inference of the reification environments) and the posterior reconstruction of Coq terms to apply the `decide` theorem is roughly 1100 lines long[4].

2.7.2 Subterms

Another point of high practical importance is the ability to rewrite in subterms rather than at the root. Indeed, the algorithm of Fig. 2.14 does not allow to match the pattern $x \oplus x$ against the terms $f(a \oplus a)$ or $a \oplus b \oplus a$, where the occurrence appears under some context. Technically, it suffices to extend the (OCaml) pattern matching function and to write some boilerplate to accommodate contexts; the (Coq) decision procedure is not affected by this modification. Formally, subterm-matching a pattern p in a term t results in a set of solutions which are pairs $\langle C, \sigma \rangle$, where C is a context and σ is a substitution such that $C[p\sigma] \equiv_{AC} t$.

Variable extensions. It is not sufficient to call the (root) matching function on all syntactic subterms: the instance $a \oplus a$ of the pattern $x \oplus x$ is not a syntactic subterm of $a \oplus b \oplus a$. The standard trick consists in enriching the pattern using a *variable extension* [123, 144], a variable used to collect the trailing terms. In the previous case, we can extend the pattern into $y \oplus x \oplus x$, where y will be instantiated with b. It then suffices to explore syntactic subterms: when we try to subterm-match $x \oplus x$ against $(a \oplus c) \otimes (a \oplus b \oplus a)$, we extend the pattern into $y \oplus x \oplus x$ and we call the matching algorithm (Fig. 2.14) on the whole term and the subterms a, b, c, $a \oplus c$ and $a \oplus b \oplus a$. In this example, only the last call succeeds.

More generally, when rewriting with an identity $x \oplus \cdots \oplus y \equiv t$, one generates a fresh variable v, and uses the identity $v \oplus x \oplus \cdots \oplus y \equiv v \oplus t$ as a rewrite rule. When dealing with associative only symbols at the top of the rewrite rule, the extension of an identity $x \otimes \cdots \otimes y \equiv t$ requires two fresh variables v_1 and v_2: the rewrite rule becomes $v_1 \otimes x \otimes \cdots \otimes y \otimes v_2 \equiv v_1 \otimes t \otimes v_2$.

The problem with subterms and units. However, this approach is not complete in the presence of units. Let 0 be a unit for $\oplus$ and 1 be a unit for $\otimes$. Suppose for instance that we try to match the pattern $x \oplus x$ against $a \otimes b$ (recall that $\otimes$ is associative only). If the variable x can be instantiated with the neutral element 0 for $\oplus$, then the variable extension trick gives four solutions:

$$a \otimes b \oplus [] \qquad (a \oplus []) \otimes b \qquad a \otimes (b \oplus [])$$

(These are the returned contexts, in which [] denotes the hole; the substitution is always

[4]We also defined a general purpose library of bindings for Coq constants like `nat`, `list`, etc which amounts to an additionnal 600 lines.

$\{x \mapsto 0\}$.) Unfortunately, if $\otimes$ also has a neutral element 1, there are infinitely many other solutions:

$$a \otimes b \otimes (1 \oplus []) \qquad a \otimes b \oplus 0 \otimes (1 \oplus []) \qquad a \otimes b \oplus 0 \otimes (1 \oplus 0 \otimes (1 \oplus [])) \qquad \ldots$$

(Note that these solutions are distinct modulo AC, they collapse to the same term only when we replace the hole with 0.) The latter solutions seem really peculiar and they only appear when the pattern can be instantiated to be equal to a neutral element (modulo A/AC). From a mathematical viewpoint, the problem is that in this case, matching becomes *infinitary*: for any matching problem, there is a minimal (read: without redundancy) complete (read: without missing substitutions) set of solutions, but there exists matching problems for which this set is infinite. Therefore, we opted for a pragmatic solution in this case: we simply ignore such solutions, displaying a warning message. The user can still instantiate the rewriting lemma explicitly, or make the appropriate transitivity step using `aac_reflexivity`.

2.7.3 Ruling out dummy cases.

Consider the following rewriting problem:

```
Hypothesis H: ∀ x y, x * y + x * z ≡ x*(y+z).
Goal a*b*c + a*c + a*b ≡ a*(c+b*(1+c)).
```

Even if we side-step from the aforementioned problem, and consider only the matching subterms that are not making nested units appear, there is still too many solutions. E.g, one would definitely like to rule out solutions in which `x` is instantiated with 1. Thus, we introduce an option to our matching modulo AC algorithm that prevents a universally quantified variable to be instantiated with a unit. The tactic `aac_rewrite` implements this policy and omits the most peculiar solutions, while `aacu_rewrite` does not impose such a restriction. It must be noted that the above restriction does not prevent `aac_rewrite` to deal with units: it solely prevents some affectations to variables. For instance, rewriting the identity $x + x = x$ in $a * (1 + 0) + a$ is still possible.

2.8 Digression: Alternative problems and solutions

We have presented the elements at the core of our tactic to rewrite universally quantified equations modulo AC. This tactic addresses the problem of performing small rewriting steps in complex proofs. We shall now put our solution in perspective with some slightly different problems.

2.8.1 Completion modulo AC

Completion aims at transforming a set of identities into a confluent term rewriting system. While completion modulo AC may of course not terminate, it has been shown that it is the case when the input set of identities is ground [94, 110]. Integrating completion modulo AC in Coq would make it possible to obtain powerful normalisation tactics either with respect to a set of ground identities, or with respect to an arbitrary set of rewriting rules that is confluent and terminating modulo AC.

From our experience, it is not often the case that the need arises for such normalisation tactics. Most of the time, one need either to show that an equality holds modulo AC or to perform small, carefully selected rewrite steps. Moreover, normalising a goal w.r.t. the completion of an arbitrary set of rewriting rules may yield results that are unpredictable for the users.

2.8.2 Congruence closure modulo AC

Congruence closure is a particular instance of the word problem where the set of identities does not contain variables. In this case, the word problem is decidable in quasi-linear time [61, 116]. Congruence closure may be extended to handle the associativity and commutativity of given operators [10]. Integrating a decision procedure for congruence closure modulo AC in Coq would be a drop-in replacement for `aac_reflexivity`, and it would subsume some of the use cases for our rewriting tactic: when the user only rewrites ground equations and concludes his goal using a congruence closure decision procedure. Yet, rewriting modulo AC is still useful when rewriting universally quantified equations, or when the resulting goals cannot be solved by congruence closure.

2.8.3 CoqMT

CoqMT [12, 150] is an extension of Coq that investigates the incorporation of trusted user-defined decision procedure in the conversion test. We could envision to define an external and trusted decision procedure for equality modulo AC (by extraction of our decision procedure). However, there are two issues at hand. First, recall that the theory decided by `aac_reflexivity` is extensible through the declaration of instances: it is unclear that this could be embedded inside the framework of CoqMT. Second, for efficiency issues, one may want to rely on the extended conversion mechanism solely in selected steps of a proof which requires a more fine-grain control than what is possible in CoqMT.

2.8.4 Extension to the multi-sorted case

A drawback from the encoding of terms from §2.5 is that it does not support heterogeneous uninterpreted symbols. Indeed it might be the case that uninterpreted symbols have different types for their arguments, and their result. A typical use case would be proofs about arithmetic: for instance `Zpower_nat` has type `Z → nat → Z`. One may want to use a rewriting rule that uses associativity and commutativity in both arguments. We propose in Fig. 2.16 a data type for reified multi-sorted terms, focusing on the function symbols (the handling of AC/A symbols and units is almost orthogonal). This is a direct extension of the mono-sorted terms from §2.5, transforming the number of arguments (of type `nat`) into a list of indexes of types (of type `list eqType`).

While this encoding of reified terms (extended with AC/A symbols and units) is appealing, there are several issues that must be considered. First, reified terms in this setting require more parametrisation, which increases the size of the generated proof terms. Second, the actual reification becomes more technical to handle, and requires more interactions with the type class resolution mechanism. It is yet unclear that such interaction could be done in a predictable way for the user, yielding the "correct" reification. We leave a more thorough investigation of this question as a future work.

2.9 Related Works

Boyer and Moore [32] are precursors to our work in two ways. First, their paper is the earliest reference to reflection we are aware of, under the name "Metafunctions". Second, they use this methodology to prove correct a simplification function for cancellation modulo A. By contrast, we proved correct a decision procedure for equality modulo A/AC with units which can deal with arbitrary function symbols, and we used it to devise a tactic for rewriting modulo A/AC.

Figure 2.16 Reified heterogeneous terms

```
Notation eqType := pos.

(* Type equipped with a setoid-relation *)
Class EqType := {
  X: Type;
  equal: relation X;
  equal_equivalence:> Equivalence equal}.

(* Environment for types *)
Variable Types : eqType → EqType.

(* Type of heterogeneous functions of arity l *)
Fixpoint type_of (l : list eqType) (x : eqType ) :=
  match l with
  | nil ⇒ @X (Types x)
  | cons t  q ⇒ @X (Types t) → type_of q x end.

(* Relation to be preserved by heterogeneous functions of arity l  *)
Fixpoint rel_of  (l : list eqType) (x : eqType ) : relation (type_of l x) :=
  match l with
  | nil ⇒ @equal (Types x)
  | cons t  q ⇒ respectful (@equal (Types t)) (rel_of q x) end.

Record sym := mk_sym {
  args : list eqType;
  res : eqType;
  val : type_of args res;
  compat : Proper (rel_of args res) val}.

Variable env : idx → sym.
Inductive T : eqType → Type :=
  Sym : ∀ (i : idx), vT (args (env i)) →  T (res (env i))
with vT : list eqType → Type :=
  | vT_nil : vT nil
  | vT_cons : ∀ E l, T E →  vT l → vT (E::l).
```

Ring.

While there is some similarity in their goals, our decision procedure is incomparable with the Coq `ring` tactic [76]. On the one hand, `ring` can make use of distributivity and opposite laws to decide equations in a ring structure, e.g, it can prove $x + -x = 0$ or $x^2 - y^2 = (x - y) * (x + y)$. On the other hand, `aac_reflexivity` can deal with an arbitrary number of AC or A operations with their units, and more importantly, with uninterpreted function symbols. For instance, it proves equations like $\mathtt{f}(\mathtt{x} \cap \mathtt{y}) \cup \mathtt{g}(\emptyset \cup \mathtt{z}) = (\mathtt{g}\ \mathtt{z}) \cup \mathtt{f}(\mathtt{y} \cap \mathtt{x})$, where `f`, `g` are arbitrary functions on sets. However, it must be noted that on the common subset, `ring` ought to be more efficient: first, the reification process is more involved in our case; second, the decision procedure for equalities in rings can take advantage of special purpose optimisations.

Rewriting modulo AC in HOL and Isabelle.

Nipkow [119] used the Isabelle system to implement matching, unification and rewriting for various theories including AC. He presents algorithms as proof rules, relying on the Isabelle machinery and tactic language to build actual tools for equational reasoning. While this approach leads to elegant and short implementations, what is gained in conciseness and genericity is lost in efficiency, and the algorithms need not terminate. Moreover, the rewriting modulo AC tools he defines are geared toward automatic term normalisation. By contrast, our approach focuses

on providing the user with tools to select and make one rewriting step efficiently.

Slind [144] implemented an AC-unification algorithm and incorporated it in the `hol90` system, as an external and efficient oracle. It is then used to build tactics for AC rewriting, cancellation, and modus-ponens. While these tools exploit pattern matching only, an application of unification is in solving existential goals. Apart from some refinements like dealing with neutral elements and A symbols, the most salient differences between our works are that we use a reflexive decision procedure to check equality modulo A/AC rather than a tactic implemented in the meta-language, and that we use type-classes to infer and reify automatically the A/AC symbols and their units.

Support for the former tool [119] has been discontinued, and it seems to be also the case for the latter [144]. To the best of our knowledge, even though HOL-light and HOL provide some tactics to prove that two terms are equal using associativity and commutativity of a single given operation, tactics comparable to the ones we describe here no longer exist in the Isabelle/HOL family of proof assistants.

However, Isabelle implements ordered rewriting[5] [111] to handle permutative rewrite rules, i.e., identities in which the left-hand side and the right-hand side are equal up-to the renaming of the free variables. Examples of permutative rewrite rules include commutativity $x+y = y+x$, but also other equations like $x - y - z = x - z - y$ in arithmetic. The idea of ordered rewriting is to rewrite using such rules only when terms become smaller w.r.t. a given lexicographic ordering on ground terms. For instance, using this strategy, commutativity would rewrite $b + a$ into $a + b$, but would not rewrite $a + b$ into $b + a$ if $a + b$ is strictly smaller than $b + a$. Using a meta-level rewriting tactic that implements ordered rewriting, makes it possible to normalise ground terms with AC symbols lexicographically, and covers AC rewriting in some cases.

Rewriting modulo AC in Coq.

Contejean [51] implemented in Coq an algorithm for matching modulo AC, which she proved sound and complete. The emphasis is put on the proof of the matching algorithm, which corresponds to a concrete implementation in the CiME system. Although decidability of equality modulo AC is also derived, this development was not designed to obtain the kind of tactics we propose here (in particular, we could not reuse it to this end). Finally, symbols can be uninterpreted, commutative, or associative and commutative, but neither associative only symbols nor units are handled.

Gonthier et al. [73] have recently shown how to exploit a feature of Coq's unification algorithm to provide "less ad hoc automation". In particular, they automate reasoning modulo AC in a particular scenario, by diverting the unification algorithm in a complex but really neat way. Using their trick to provide the generic tactics we discuss here might be possible, but it would be difficult. Our reification process is much more complex: we have uninterpreted function symbols, we do not know in advance which operations are AC, and the handling of units requires a dependent environment. Moreover, we would have to implement matching modulo AC (which is not required in their example) using the same methodology; doing it in a sufficiently efficient way seems really challenging.

Nguyen et al. [118] used the external rewriting tool ELAN to add support for rewriting modulo AC in Coq. They perform term rewriting in the efficient ELAN environment, and check the resulting traces in Coq. This allows one to obtain a powerful normalisation tactic out of any set of rewriting rules which is confluent and terminating modulo AC. Our objectives are slightly different: we want to easily perform small rewriting steps in an arbitrarily complex proof, rather than to decide a proposition by computing and comparing normal forms.

[5] This idea previously appeared in the Boyer-Moore theorem prover [31].

The ELAN trace is replayed using elementary Coq tactics, and equalities modulo AC are proved by applying the associativity and commutativity lemmas in a clever way. On the contrary, we use the high-level (but slightly inefficient) `rewrite` tactic to perform the rewriting step, and we rely on an efficient reflexive decision procedure for proving equalities modulo AC. (Alvarado and Nguyen first proposed a version where the rewriting trace was replayed using reflection, but without support for modulo AC [7].)

From the user interface point of view, leaving out the fact that the support for this tool has been discontinued, our work improves on several points: thanks to the recent plug-in and type-class mechanisms of Coq, it suffices for a user to declare instances of the appropriate classes to get the ability to rewrite modulo AC. Even more importantly, there is no need to declare explicitly all uninterpreted function symbols, and we transparently support polymorphic operations (like `List.app`) and arbitrary equivalence relations (like `Qeq` on rational numbers, or `iff` on propositions). It would therefore be interesting to revive this tool using the new mechanisms available in Coq to provide (again) tactics for proofs by normalisation.

Alvarado defined in his thesis [6] a dependently typed representation of terms of multi-sorted algebras in order to implement rewriting in a reflexive manner. His representation captures exactly the well-formed and well-sorted terms, and is roughly similar to the one we presented in §2.16. However, he argued that his dependently-typed representation of terms could hinder computations, and had to resolve issues with the use of dependent equality in previous versions of Coq. We look forward to pursue our investigation of a reflexive decision procedure for equality modulo AC in a multi-sorted setting to assess to what extent these remarks still apply.

Maude.

Although this is not a general purpose interactive proof assistant, the Maude system [48], which is based on equational and rewriting logic, also provides an efficient algorithm for rewriting modulo AC [63]. Like ELAN, Maude could be used as an oracle to replace our OCaml matching algorithm. This would require some non-trivial interfacing work, however. Moreover, it is unclear to us how to use these tools to get all matching occurrences of a pattern in a given term: they are designed to perform normalisation with a set of rules which are a priori confluent and terminating.

2.10 Conclusion

The Coq library corresponding to the tools we presented is available from [34]. We do not use any axiom and the code consists in total of about 1400 lines of Coq and 3600 lines of OCaml. We conclude this chapter with potential directions for future works.

Heterogeneous terms. Our decision procedure cannot deal with functions whose range and domain are distinct sets, each equipped with AC or A symbols. We could extend the tactic to deal with such objects, and allow one to rewrite equations like $\forall uv, \|u + v\| \leq \|u\| + \|v\|$, where $\|\cdot\|$ would be a norm in a vector space. This would require a more involved definition of reified terms and environments to keep track of type informations (see Fig. 2.16), and the corresponding reification process seems the most challenging difficulty.

Heterogeneous operations. We could also attempt to handle heterogeneous associative operations, like multiplication of non-square matrices, or composition of morphisms in a category. For example, recall the `dot` law from the previous chapter which has type

$$\forall n \forall m \forall p, X\ n\ m \to X\ m\ p \to X\ n\ p$$

Apart from making it possible to use the tools for reasoning modulo AC we developed in the context of typed Kleene algebra, this would also be helpful for proofs in category theory. Again, the first difficulty is to adapt the definition of reified terms; for instance, this would certainly require dependently typed non-empty lists to handle associative operators.

Other decidable theories. While we focused on rewriting modulo AC, we could consider other theories whose matching problem is decidable. Such theories include, for example, the Abelian groups and the boolean rings [29] (the latter naturally appears in proofs of hardware circuits). Another possibility would be to add support for commutative only symbols, that are useful to deal with arithmetic over floating-point numbers: while this would make our development more verbose, we foresee no difficulty in doing this.

Other tactics. While we focused on rewriting modulo AC and on the decision of equations modulo AC, we could provide other tactics. For instance, some early user-feedback has shown that cancellation and modus ponens modulo AC could be useful to simplify the proof engineering.

Library support for reification. The technicity of our reification mechanism and, e.g., the changes that were required to add support for units, make us doubt that there could be a "universal reification framework" that fits all uses. In particular, we made different implementation choices in this chapter and in the previous one: in the latter, we chose to be efficient, while we privileged the flexibility of use here. However, we could try to provide some library of commonly used bindings to Coq data-structures, and to implement some common reification "recipes" that could be adapted to the various problem at hand. This would lower the amount of knowledge of Coq internals required to develop reification tactics.

Chapter 3

Verifying hardware circuits in Coq

In this chapter, we describe a new library to model and verify hardware circuits in Coq. This library allows one to easily build circuits by following the usual pen-and-paper diagrams and its main novelty is that it defines a deep-embedding circuits: we use a (dependently typed) data-type that models the architecture of circuits, and a meaning function. We propose tactics that ease the reasoning about the behaviour of the circuits, and we demonstrate that our approach is practicable by proving the correctness of various examples: a text-book divide and conquer adder of parametric size, some higher-order combinators of circuits, and some sequential circuits: a buffer, and a register.

Introduction

Verification of hardware components has been thoroughly investigated. However, obtaining provably correct hardware of significant complexity is usually considered challenging and time-consuming.

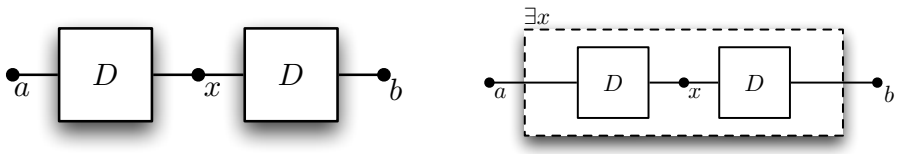
A common practice of hardware verification is to take a given design in an hardware description language and to argue about this design in a formal way, using a model checker or a theorem prover. There are two pitfalls with this. First, usual hardware description languages have limited support for the verification of parametrised designs (like designs parametrised by a size): each occurrence of such a parametric design ought to be verified separately. The second hurdle is the hierarchical verification of sizable artefacts. Model checking methods like the ones based on BDDs [40] and SAT engines have the advantage of being fully automated and thus can be used by non-specialists people[1]. Yet, they can only deal with circuits of fixed size and suffers from combinatorial explosion.

Thus, a completely different approach is to design hardware *using* theorem provers [26, 59, 74, 79, 95]. There are two different flavours to the modelisation of circuits (and more generally, programming languages) in a theorem prover. In a *shallow-embedding*, circuits are written directly in the logic of the theorem prover, while in a *deep-embedding*, circuits are represented as a particular data type of the theorem prover.

Shallow-embeddings For instance, the overall method introduced by Gordon and Melham [74, 112] to model circuits in higher-order logic is to use predicates of the logic to express the possible behaviour of devices. Depending on the level of detail of the formalisation, designs may be built using gate level primitives such as `AND`, `OR` or inverter gates, but also using higher-level primitive circuits. As an example, the `Xor` circuit and the `Not` circuit may be modelled as

[1] While formal verification using proof-assistants like Coq remains an art

Figure 3.1 Building circuits using a shallow-embedding



$D(a, x) \wedge D(x, b)$ $\quad\quad$ $\exists x, D(a, x) \wedge D(x, b)$

follows:

$$Xor(i_1, i_2, o) \triangleq (o = \neg(i_1 = i_2)) \qquad Not(i, o) \triangleq (o = \neg i)$$

These circuits may be combined using the usual conjunction of the underlying logic, and existential quantification may be used to hide internal "ports" of the devices (see Fig. 3.1).

While this particular example of shallow-embedding models circuits with *predicates* of the logic, it is also possible to represent circuits using *functions* of the theorem prover. In this setting, serial composition of circuits is modelled through function composition, and there is no such things as "ports". Such encodings have been used in large scale projects such as the VAMP project [26] in PVS, the formal verification of the floating -point adder of a commercial microprocessor [134] in ACL2, or the verification of the FM9801 microprocessor [138] in ACL2.

Generally, using a shallow-embedding to represent circuits in a theorem prover makes it easy to model circuits: the *wiring* of components is done through function applications using the (meta-level) substitution of the language of the theorem prover and one may freely use recursion to define recursive structures. Moreover proving that a circuit meets some specification amounts to the proof of an entailment between logical formulas.

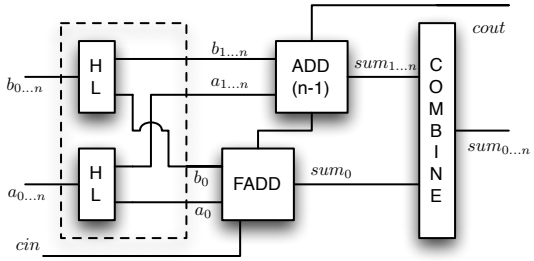
However, the lack of "syntax" of shallow-embeddings has its disadvantages. First, there are functions and predicates that do not represent circuits, which makes quantifying over circuits slippery. Moreover, it is not possible to build functions that operate over circuits: there is no way to do, e.g., pattern-matching on the definition of a predicate. The fact that these functions must be built at the meta-level [146] precludes one from proving their correctness. Incidentally, meaningful properties of circuits like their latencies, or their gates number must also be computed at the meta-level, making it impossible to *state* inside the theorem prover that a given circuit meets some complexity bounds, for instance.

Deep-embeddings. By contrast, we define a deep-embedding of circuits in Coq. That is, we define a data-type for circuits, and a meaning function. Therefore, we can write (and reason about) Coq functions that operate on the structure of circuits.

One of the usual of deep-embeddings is that defining a data type that represents logical formulas or programs requires to deal explicitly with variables bindings and substitutions. We actually side-step this difficulty by relying on *circuit combinators*. These combinators make precise the interconnection of circuits, yet making implicit low-level diagram constructs such as wires and ports. Note that while the circuit diagrams we describe have nice algebraic properties [37, 107], we do not prove algebraic laws here. A more detailed introduction to our deep-embedding of circuits is presented in the next sections.

Dependent-types. There has been a lot of work describing and verifying circuits in the HOL and ACL2 family of theorem provers. However, Coq features dependent types that are more expressive. The seminal Veritas language experiment [79] and the more recent VAMP project

Figure 3.2 A recursive n-bit ripple-carry adder



in PVS [26] hinted that these allow for specifications that are both clearer and more concise. We also argue that dependent types are invaluable for developing circuits reliably: some errors can be caught early, when type-checking the circuits or their specifications.

Recursive circuits. Circuit diagrams are also used to present recursive or parametric designs. We use Coq recursive definitions to *generate* circuits of parametric size, e.g., to generate an n-bit adder for a given n. Then, we reason about these functions rather than on the tangible (fixed-size) instantiations of such generators. Remark that circuits modelled by recursion have already been verified in other settings [95, 112]. By contrast, the novelty of our approach is that we derive circuit designs in a systematic manner: we structure circuits generators by mimicking the usual circuit diagrams, using our combinators. Then, the properties of these combinators allow us to prove the circuits correct.

Dependable circuits. We are interested in two kinds of formal dependability claims. First, we want to capture some properties of well-formedness of the diagrams. Second, we want to be able to express the *functional correctness* of circuits – the fact that a circuit complies to some specification, or that it implements a given function. Obviously, the well-formedness of a circuit is a prerequisite to its functional correctness. We will show that using dependent types, we can get this kind of verification for free. As an example, the type-system of Coq will preclude the user to make invalid compositions of circuits: e.g., serial composition of circuits will ensure that the dimensions of the circuits agree. Hence, we can focus on what is the intrinsic *meaning* of a circuit, and prove that the meaning of some circuits entails a high-level specification, e.g., some functional program.

3.1 Overview of our system

In this section, we give a global overview of the basic concepts of our methodology first, before giving a formal Coq definition to these notions in the next section. We take this opportunity to illustrate the use of our system to represent parametrised circuits through the example of a simple n-bit ripple-carry adder: it computes an n-bit sum and a 1-bit carry-out from two n-bit inputs and a 1-bit carry-in. The recursive construction scheme of this adder is presented in Fig. 3.2 (data flows from left to right), using a *full-adder*, i.e., a 1-bit adder, as basic building block. The `HL` (resp. `COMBINE`) sub-circuit is a mere combinator that decomposes a bit-vector (resp. compose two bit-vectors).

Circuit interfaces. Informally, we want to build circuits that relate n input ports to m output ports, where n, m are integers. For instance, the gate `AND` has two inputs and one

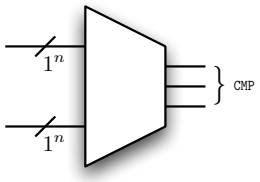
output, while the recursive n-bit adder has $2n+1$ inputs and $n+1$ outputs. (Remark that the interfaces of circuits should not be confounded with their semantics: we shall later reason about the *valuation* of ports, i.e., mappings from the ports to a given semantic domain.) However, using integers to number the ports does not give much structure: the fact that the n-bit adder has $2n+1$ input ports does not specify how they are grouped. Yet, this information must be made precise.

Informally, circuit interfaces are defined as an abstract *magma* with an operation $\diamond$ (i.e., combining in parallel a circuit with a input ports and a circuit with b input ports yields a circuit with $a \diamond b$ input ports): in this abstract setting, the interfaces $n \diamond n \diamond 1$, $n \diamond 1 \diamond n$, and $1 \diamond n \diamond n$ are distinct. The magma of natural numbers, equipped with addition, is a particular model of this setting that yields a quotient of the interfaces by the relation "having the same number of ports". Since we require a more fine grain description of the interfaces, we move away from this particular model: we use arbitrary *finite-types* as indexes for the ports rather than integers [80].

That is, a circuit with three inputs may have the input interface $\mathbf{1} \oplus \mathbf{1} \oplus \mathbf{1}$ where $\oplus$ is the disjoint sum of types (associative to the left) and $\mathbf{1}$ is a singleton type. Then, a circuit that relates inputs indexed by n to outputs indexed by m has type $\mathbb{C}\ n\ m$, where n and m are types. As an example, the full-adder, a circuit with three inputs and two outputs, has type $\mathbb{C}\ (\mathbf{1} \oplus \mathbf{1} \oplus \mathbf{1})\ (\mathbf{1} \oplus \mathbf{1})$ and the n-bit adder has type $\mathbb{C}\ (\mathbf{1} \oplus n \cdot \mathbf{1} \oplus n \cdot \mathbf{1})\ (n \cdot \mathbf{1} \oplus \mathbf{1})$, where $n \cdot A$ is an n-ary disjoint sum.

Note that this makes it possible to use other types than units to give more precise descriptions. Consider below the description of an n-bit comparator circuit that may be given the type $\mathbb{C}\ (n \cdot \mathbf{1} \oplus n \cdot \mathbf{1})\ (\texttt{CMP})$.

```
Inductive CMP : Type := | Eq | Lt | Gt.
```



On this particular example, we argue that using the three-elements type `CMP` yields a more precise description of the output interface of this circuit, that leaves less room for confusion than using $\mathbf{1} \oplus \mathbf{1} \oplus \mathbf{1}$, an other three-elements type.

Circuits combinators. The n-bit adder is made of several sub-components that are composed together. We use *circuit combinators* (or combining forms [142]) to specify the connection layout of circuits. For instance, in Fig. 3.2, the dashed-box is built by composing in parallel two `HL` circuits that are then composed serially with a combinator that reorders the ports. These combinators leave the connection points implicit in the circuits and focus on how information flow through the circuit. In particular, the port names given in Fig. 3.2 do not correspond to variables, and are provided for the sake of readability.

In our "nameless" setting, ports have to be duplicated and reordered using *plugs*: a plug is a circuit of type $\mathbb{C}\ n\ m$, defined using a map from m to n that defines how to connect an output port (indexed by m) to an input port (indexed by n). Since we use functions rather than relations, this definition naturally forbids short-circuits, e.g., two input ports connected to the same output port.

Meaning of a circuit. We now depart from the syntactic definitions of circuits to give an overview of their semantics. The meaning of a circuit x of type $\mathbb{C}\ n\ m$ is defined as a relation

between a valuation of its input ports and a valuation of its output ports. We assume a semantic domain $\mathbb{T}$, and define a *valuation* of a type n as a function of type $n \to \mathbb{T}$. In the following, and depending on the context, the type $\mathbb{T}$ may be instantiated either by Booleans (written $\mathbb{B}$) or by streams of Booleans (written $\texttt{nat} \to \mathbb{B}$). We denote by $x \vdash_m^n ins \bowtie outs$ the *meaning* of x, a relation between $ins : n \to \mathbb{T}$ and $outs : m \to \mathbb{T}$. Note that this relation, which is defined by induction on the structure of x is an abstract mathematical characterisation, which may be either axiomatic or computational (we will come back to this point later).

Abstracting from the implementation. The semantics of a circuit defines precisely its behaviour, but may be too precise, e.g., it may leak some internal implementation details (recall that it is defined by induction over the structure of the circuit). The standard technique [112] to prove that circuits meet some high-level specifications is to prove the following kind of entailment, where the high-level specification R abstracts the behaviour of the circuit `RIPPLE n`:

$$\forall ins, \forall outs, (\texttt{RIPPLE n} \vdash ins \bowtie outs) \to R\ outs\ ins$$

Then, recall that *ins* (resp. *outs*) are valuations of the input (resp. output) ports, and that, e.g., the semantics of an `AND` gate is a relation between $\mathbf{1} \oplus \mathbf{1} \to \mathbb{B}$ and $\mathbf{1} \to \mathbb{B}$. While it is possible to write specifications for these kind of objects, this is not much practical considered that, for instance, a value of type $\mathbf{1} \oplus \mathbf{1} \to \mathbb{B}$ is isomorphic to a value of type $\mathbb{B} \times \mathbb{B}$. Therefore, a decisive move in this work was to build on *type-isomorphisms* as a refinement of *data-abstraction* [112]. This makes it possible to give elegant and high-level specifications to circuits: given that a value of type $\mathbf{1} \oplus n \cdot \mathbf{1} \oplus n \cdot \mathbf{1} \to \mathbb{B}$ is isomorphic to a value of type $\mathbb{B} \times \mathbb{W}_n \times \mathbb{W}_n$ (where $\mathbb{W}_n$ is the type of integers from 0 to $2^n - 1$), we prove that the parametric n-bit adder depicted in Fig. 3.2 implements the addition with carry on $\mathbb{W}_n$.

3.2 Formal development

We now turn to the formal definition of the concepts that were overviewed in the previous section.

3.2.1 Circuit interfaces

We hinted that we use arbitrary Coq types as interfaces for the circuits, endowed with the disjoint-sum operation (denoted $\oplus$). We use the one-element type $\mathbf{1}$ as a basic interface, and we define n-ary disjoint sums of a given type A (written $n \cdot A$) as a Coq fixpoint. As an example, the type $n \cdot (\mathbf{1} \oplus \mathbf{1})$ is an interface for a circuit with n pairs of input ports. However, using a single singleton type for all ports can be confusing: there is no way to distinguish one $\mathbf{1}$ from another, except by its position in the interface (which is frustrating). Hence, we use an infinite family of one-element types $\mathbf{1}_x$ where x is a *tag*.

```
Inductive tag (t : string) : Type := _tag : tag t. (* we write 1_t for tag t*)
```

In the following, we parametrise circuits definitions by the tags that appear in the interfaces. Despite the small notational overhead of this tagging discipline, we argue that it makes it easy to follow circuit diagrams to define circuits in Coq without much room for mistakes: ill-formed combinations of circuits are ruled out by the Coq-type system. (Note that this use of tags make it possible to make names appear in the interfaces, even if our setting remain nameless. Yet, this is only a lightweight improvement on the use of positions to identify ports.)

A physical circuit may only have a finite number of ports. Therefore, we restrict ourselves to a setting in which types that appear as interfaces are *finite*. We use the usual interpretation of

Figure 3.3 Isomorphisms between types

$$\cdot \bullet \cdot \ \frac{A \to \mathbb{T} \cong \sigma \qquad B \to \mathbb{T} \cong \tau}{A \oplus B \to \mathbb{T} \cong (\sigma \times \tau)} \qquad \iota_{\mathtt{x}} \ \frac{}{\mathbf{1}_x \to \mathbb{T} \cong \mathbb{T}}$$

$$\frac{}{\mathbf{0} \to \mathbb{T} \cong \mathbf{1}} \qquad \frac{A \to \mathbb{T} \cong \sigma}{n \cdot \mathbf{1}_A \to \mathbb{T} \cong \mathtt{vector}\ \sigma\ \mathtt{n}}$$

Kuratowski finiteness [93]: a type is finite if and only if its elements can be enumerated by a list. We define a finite type A as a Coq dependent record that packages a duplicate-free list of all elements of A along the lines of [72]. However, this finiteness property is almost a meta-property of our circuits: we may safely omit this requirement from now (and, we shall come back on this in later sections).

3.2.2 Type isomorphisms

Recall that the meaning of a circuit is a relation between valuations of the interfaces like, e.g., $n \cdot \mathbf{1} \to \mathbb{B}$, which are not much palatable. We use type-isomorphisms as "lenses" to express the specification of circuits in terms of user-friendly types: for instance, the previous type turns out to be isomorphic to $\mathbb{W}_n$.

In a nutshell, we define in Coq an isomorphism between two types A and B as a pair of functions `iso`:`A` → `B` and `uniso`: `B` → `A` that are proved to be inverse of each other.

```
Class Iso (A B : Type) :={
 iso : A → B;
 uniso : B → A}.
```

```
Class Iso_Props {A B: Type} (I : Iso A B):= {
 iso_uniso : ∀ (x : B), iso (uniso x) = x;
 uniso_iso : ∀ (x : A), uniso (iso x) = x}.
```

In the following, we denote by $A \cong B$ an isomorphism between A and B (i.e., an instance of both type-classes above). As an example, a pivotal parametrised instance states the duality between disjoint-sums in the domain of the valuations and cartesian products.

```
Context {A B σ τ: Type} (IA: A → 𝕋 ≅ σ) (IB: B → 𝕋 ≅ τ).
Instance Iso_sum : (A ⊕ B → 𝕋) ≅ σ × τ := ...
```

We define some notations and give examples of such isomorphisms in Fig. 3.3. For instance, let $\mathbb{T}$ be the set of Booleans: we denote by $\iota \bullet \iota$ an isomorphism between $\mathbf{1} \oplus \mathbf{1} \to \mathbb{B}$ and $\mathbb{B} \times \mathbb{B}$.

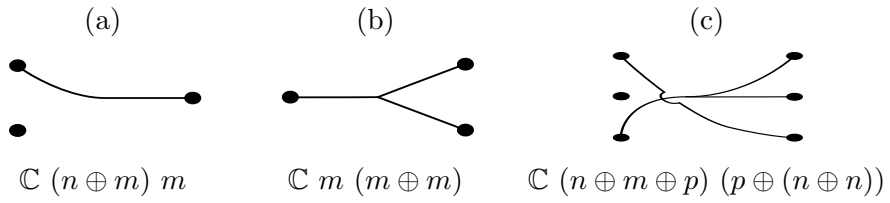
Isomorphisms naturally form a *groupoid* endowed with a unary inverse function (of type $A \cong B \to B \cong A$) and an (associative) partial composition (of type $A \cong B \to B \cong C \to A \cong C$). While we rely lightly on these properties in some proofs, we will not emphasise their use in the sequel. However, note that there may be several isomorphisms between two given types: we rely on a small set of base constructions (like the above `Iso_sum`) in order to tame this complexity when reasoning about, e.g., compositions of isomorphisms.

3.2.3 Plugs

"Plugs" are circuits of type $\mathbb{C}\ n\ m$ defined as a mapping from the output ports, indexed by m, to the input ports, indexed by n. For instance, consider the plug of type $\mathbb{C}\ (n \oplus m)\ m$ from Fig. 3.4(a) that "forgets" the first group of input ports (types must be read bottom-up on these diagrams). This plug is defined as the following function:

$$\begin{array}{ccc} m & \to & n \oplus m \\ x & \mapsto & \mathtt{inr}\ x \end{array}$$

Figure 3.4 Some examples of plugs



Observe that we define plugs as usual Coq functions to get small and computational definitions of maps. Yet, since these maps operate on the indexes of the (output) ports, there is no way to embed an arbitrary function inside our circuits to compute, e.g., the Boolean and of the valuations of two ports. Therefore, we argue that our formalisation remains a deep-embedding of circuits, despite this particular use of Coq functions.

The two other examples from Fig. 3.4 may be read as follows: (b) is a a fork, i.e., a circuit that duplicates its inputs and (c) implements some re-ordering and duplication of the ports. (Remark that, for the sake of clarity, we shall leave implicit the associativity of the $\oplus$ operation on the diagrams.) A possible definition of (b) is the following function:

$$\begin{array}{lcl} m \oplus m & \to & m \\ \texttt{inl}\ x & \mapsto & x \\ \texttt{inr}\ x & \mapsto & x \end{array}$$

We do not give a definition of (c): it would be quite verbose and not much interesting. It happens that we did not write it in Coq: if the type of the circuit gives enough information, like the examples above, it is actually possible to define such plugs using proof-search. Indeed, plugs that deal with the associativity of the ports, or even re-orderings of ports, are completely defined by the type of their interfaces, and we define a Coq tactic that computes these maps. It amounts to some case splitting and little automation. While we skip this short `Ltac` code, we emphasise that it relies on the `auto` tactic: therefore, it is of the responsibility of the user to ensure that the plug found by proof-search is the "right one" whenever several exist.

In the following formal definition of circuits, we shall therefore omit such simple plugs, not only for the sake of readability, but also because we do so in the actual Coq code: we leave holes in the code (thanks to the Coq Program feature) that will be filled automatically using the aforementioned proof-search. Yet, we still need to state what function implements a given plug, up-to isomorphism, on a case-by-case basis. We shall come back on this later in this section.

3.2.4 Abstract syntax

We now turn to the definition of the data-type that underlies our deep-embedding of circuits in Coq. This definition is parametrised by a set of *constants* or *atoms*, i.e., a collection of base gates from which all other circuits are defined.

A common approach to the representation of a typed language in a proof assistant is to define first the abstract syntax of terms, and to provide a separate inductive definition that discriminates terms that are well-formed or well-typed. This approach has the disadvantage that definitions and lemmas about circuits become cluttered with well-formedness preconditions that must be proved "by hand" explicitly. Since we are only interested in well-formed circuits, we follow the Church-style or *intrinsic* approach [17]. That is, we use a dependently typed syntax and build the typing rules as part of the definition of the terms. Therefore, all circuits are well-formed by construction. Put succinctly, we ensure that the dimensions of circuits that

Figure 3.5 Syntax

```
Variable atom: Type → Type → Type.
Inductive ℂ : Type → Type → Type :=
| Atom : ∀ (n m : Type), atom n m → ℂ n m
| Plug : ∀ (n m : Type) (f : m → n), ℂ n m
| Ser  : ∀ (n m p : Type), ℂ n m → ℂ m p → ℂ n p
| Par  : ∀ (n m p q : Type), ℂ n p → ℂ m q →  ℂ (n ⊕ m) (p ⊕ q)
| Loop : ∀ (n m p : Type), ℂ (n ⊕ p) (m ⊕ p) → ℂ n m.
```

are composed by the combinators of the language agree, and we eliminate the possibility to connect circuits "in the wrong direction".

As we hinted at in the previous section, we rely on circuits combinators in a nameless setting: the information flow is made explicit by the combinators, not by the use of variables. These combinators draw some inspiration from string diagrams and traced monoidal categories [140] but we shall not consider this relationship in the following.

We present our representation of circuits in Fig. 3.5. First, the base cases of our inductive definition are atoms (`Atom`) and plugs (`Plug`). Second, circuits may be composed in series and in parallel. We denote serial composition (`Ser`) with the infix $\triangleright$ symbol, and parallel composition (`Par`) with &. Finally, we provide a feedback operator (`Loop`) that permits retroaction. (Note that our definition of circuits is concise, and ponder the fact that it enforces statically their well-formedness. If we were to describe circuits as graphs with ports and wires, ensuring these properties would require some boilerplate.)

3.2.5 Structural semantics

We now depart from the syntax of circuits to describe their semantics. Let $\mathbb{T}$ be a given semantical domain. Recall that the meaning of a circuit of type $\mathbb{C}\ n\ m$ is a relation between a valuation of its input ports (of type $n \to \mathbb{T}$) and a valuation of its output ports (of type $m \to \mathbb{T}$).

This relation is defined as an inductive predicate over the structure of the circuit. First, the relation enforced by a circuit $x \triangleright y$ between two valuations a and c corresponds to the conjunction of the relation induced by x between a and a fresh valuation b and the relation induced by y between b and c. Then, the relation induced by a circuit $x \& y$ between the valuations a and b is slightly more complicated. Informally, it corresponds to the conjunction of the relation induced by x (resp. y) on the "left parts" (resp. "right parts") of a and b. To make this formal, recall that, syntactically, the domains of the valuations a and b are disjoint-sums. Thus, we define an operation on such valuations of type $n \oplus m \to \mathbb{T}$ that returns (resp. drops) the first part of the valuation to yield a result of type $n \to \mathbb{T}$ (resp. $m \to \mathbb{T}$). (Note that these combinators and their semantics are somehow similar to the one that were used in the first versions of the SIGNAL programming language [108], except that their approach uses names.)

We present the definition of these operations on valuations in Fig. 3.6, along with the semantics of each circuit combinators. Note that we use inference rules rather than the corresponding Coq inductive for the sake of readability. Recall that the syntax of circuits was parametrised by a given set of basic gates, `atom`: `Type → Type → Type`. Likewise, the semantics of circuits is parametrised by the semantics of these specific atoms, that is, a definition of type `spec`, as defined below.

```
Definition spec (atom: Type → Type → Type) (𝕋: Type) :=
  ∀ (a b : Type), atom a b → (a → 𝕋) → (b → 𝕋) → Prop.
```

Figure 3.6 Meaning of circuits (omitting the rule for Atom)

```
Context {𝕋 : Type}.
Definition left  {n} {m} (x : (n ⊕ m) → 𝕋) : n → 𝕋 := fun e ⇒ (x (inl _ e)).
Definition right {n} {m} (x : (n ⊕ m) → 𝕋) : m → 𝕋 := fun e ⇒ (x (inr _ e)).
Definition lift {n} {m} (f : m → n) (x : n → 𝕋) : m → 𝕋 := x ∘ f.
Definition app {n m} (x : n → 𝕋) (y : m → 𝕋) : n ⊕ m → 𝕋 :=
  fun e ⇒ match e with inl e ⇒ x e | inr e ⇒ y e end.
```

$$\textsc{KSer}\ \frac{x \vdash^{n}_{m} ins \bowtie middle \qquad y \vdash^{m}_{p} middle \bowtie outs}{x \rhd y \vdash^{n}_{p} ins \bowtie outs}$$

$$\textsc{KPar}\ \frac{x \vdash^{n}_{p} \texttt{left}\ ins \bowtie \texttt{left}\ outs \qquad y \vdash^{m}_{q} \texttt{right}\ ins \bowtie \texttt{right}\ outs}{x \& y \vdash^{n \oplus m}_{p \oplus q} ins \bowtie outs}$$

$$\textsc{KPlug}\ \frac{}{\texttt{Plug}\ f \vdash^{n}_{m} ins \bowtie \texttt{lift}\ f\ ins} \qquad \textsc{KLoop}\ \frac{x \vdash^{n \oplus p}_{m \oplus p} \texttt{app}\ ins\ r \bowtie \texttt{app}\ outs\ r}{\texttt{Loop}\ x \vdash^{n}_{m} ins \bowtie outs}$$

We typically instantiate the `atom` and `spec` parameters in the combinatorial setting (with $\mathbb{T}$ being $\mathbb{B}$) and in the sequential setting (with $\mathbb{T}$ being $\texttt{nat} \to \mathbb{B}$). This parametricity actually makes it easier to define circuits transformations, or, e.g., to lift properties of combinational circuits into the synchronous setting as we shall see in §3.4. (Remark that the definition of `spec` makes an unavoidable use of dependent types, short of dropping the aforementioned parametricity of our definition of circuits and their semantics.)

3.2.6 Modular proofs of circuits

We shall mention that we develop circuits in a modular way: to build a complex circuit, we define a Coq functor that takes as an argument a module that packages the implementations of the sub-components, and the proofs that they meet some specifications. This means that our proofs are hierarchical: we do not inspect the definitions of the sub-components when we prove a circuit. This makes it possible to enrich our library of verified circuits in an incremental manner on top of a concrete module that contains the chosen set of base gates and their specifications.

This modularity requires both to define behavioural abstractions, expressed through the logical entailment of weak specifications by the meaning relation, and high-level specifications, that do not make the valuations appear. We represent these abstractions through the definitions of particular type-classes below. The first one specifies that a given circuit meets a given specification up to two isomorphisms, while second one is a particular instance of the first, usable when the behaviour of the circuit may actually be described as a function.

```
Context {n m N M : Type} (Rn : (n→ 𝕋) ≅ N) (Rm : (m→ 𝕋) ≅ M).
Class Realise (c : ℂ n m) (R : N → M → Prop) := realise: ∀ ins outs,
    c ⊢ⁿₘ ins ⋈ outs → R (iso ins) (iso outs).
Class Implement (c : ℂ n m) (f : N → M) := implement: ∀ ins outs,
    c ⊢ⁿₘ ins ⋈ outs → iso outs = f (iso ins).
```

We typically defines instances of the second type-class to express rich specifications for, e.g., combinational circuits or stream transformers, while retaining the first one for more peculiar relational specifications.

Figure 3.7 A proof-system for plugs

```
Inductive monoid : Type :=
| Var : Type → monoid
| ◇ : monoid → monoid → monoid.
```

```
Inductive ⊨ : monoid → monoid → Type :=
| M : ∀ A B C, A ⊨ C → B ⊨ C → (A ◇ B) ⊨ C
| L : ∀ A B C, A ⊨ B → A ⊨ (B ◇ C)
| R : ∀ A B C, A ⊨ B → A ⊨ (C ◇ B)
| I : ∀ A, A ⊨ A.
```

Figure 3.8 Two evaluations of plug derivations

```
Variable (f : Type → Type) (op: Type → Type → Type).
Fixpoint meval (A : monoid) : Type := (* written A_f^op *)
  match A with | Var x ⇒ f x | A ◇ B ⇒ op (meval A) (meval B) end.
```

```
Fixpoint to_plug {A B} (x: A ⊨ B):
  A⊕id → B⊕id :=
match x with
| M a b c l r ⇒ app (to_plug l) (to_plug r)
| L a b c l  ⇒ fun x ⇒ inl (to_plug l x)
| R a b c r  ⇒ fun x ⇒ inr (to_plug r x)
| I _ ⇒ fun x ⇒ x
end.
```

```
Fixpoint to_effect {A B} (x : A ⊨ B):
  B×λx.x→𝕋 → A×λx.x→𝕋 :=
match x with
| M a b c l r ⇒ fun e ⇒  (to_effect l e,to_effect r e)
| L a b c l ⇒ fun e ⇒  (to_effect l (fst e))
| R a b c r ⇒ fun e ⇒  (to_effect r (snd e))
| I _ ⇒ (fun x ⇒ x)
end.
```

3.2.7 Digression: a Curry-Howard isomorphism at work

Having to write down the functions implemented by plugs yields quite verbose code, and having to write them at all is not satisfactory. Intuitively, the Curry-Howard isomorphism that makes it possible to find the functions that operate on wires should be extendable to find the related functions that operate on the values. This is indeed the case. We define in Fig. 3.7 a proof system for plugs, and we define two interpretations for derivations in this system: one that yields a plug, and one that yield the function that the plug implements.

For instance, recall the above example (c) that had type $\mathbb{C}\ (n \oplus m \oplus p)\ (p \oplus (n \oplus n))$. We give below a derivation of the formula $p \diamond (n \diamond n) \vDash (n \diamond m) \diamond p$:

$$\mathrm{M}\ \frac{\mathrm{R}\ \dfrac{I}{p \vDash (n \diamond m) \diamond p} \qquad \mathrm{L}\ \dfrac{\mathrm{M}\ \dfrac{I \qquad I}{n \diamond n \vDash n}}{n \diamond n \vDash n \diamond m}}{p \diamond (n \diamond n) \vDash (n \diamond m) \diamond p}\ \mathrm{L}$$

This derivation may be evaluated to the plug, that is, a function of type

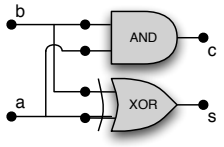
$$(p \oplus (n \oplus n)) \to (n \oplus m \oplus p);$$

and the derivation may be evaluated to the function implemented by the plug, which has type

$$((n \to \mathbb{T}) \times (m \to \mathbb{T}) \times (p \to \mathbb{T})) \to ((p \to \mathbb{T}) \times ((n \to \mathbb{T}) \times (n \to \mathbb{T}))).$$

We give the definitions of these two evaluations in Fig 3.8, and we prove that the plug actually implements its computed effect up to (trivial) isomorphisms.

Figure 3.9 Definition of a half-adder




```
Context a b s c : string.
Definition HADD : ℂ (1_a ⊕ 1_b) (1_s ⊕ 1_c) :=
Fork2 (1_a ⊕ 1_b) ▷ (XOR a b s  & AND a b c).
```

```
Lemma derivation_correct {A B} (X : A ⊨ B): Implement B̄• Ā• (Plug (to_plug X)) (to_effect X).
```

We mentioned that we use proof-search for (simple) plugs, combined with hand-written specifications, which is actually quite verbose. Using the derivation evaluations we present here make it possible to use a single phase of proof-search to compute both the plug and its specification, and could be used to reduce the verbosity of some of our definitions.

3.3 Proving some combinational circuits

In this section, we shall give some complete examples that demonstrate the use of the previous definitions, focusing on acyclic combinational circuits: more precisely, we implement some basic arithmetic circuits. We instantiate `atom` with a singleton inductive definition that corresponds to the `NOR` gate, $\mathbb{T}$ with $\mathbb{B}$ and `spec` accordingly. All other circuits in this section are defined and proved correct starting from these definitions. We first illustrate our proof methodology on a half-adder. Then, we present operations on n-bits integers that will be used to specify n-bit adders.

3.3.1 Proving a half-adder

A half-adder is a circuit that adds two 1-bit binary numbers together, producing a 1-bit number and a carry-out. We present a diagram of this circuit, along with its formal definition, in Fig. 3.9. The left-hand side of the following Coq excerpt is the statement we prove: the circuit `HADD` implements the function `hadd` on Booleans (defined as λ(a,b).(a $\oplus$ b, a $\wedge$ b), where $\oplus$ is the Boolean exclusive-or, and $\wedge$ is the Boolean and) up to isomorphisms (we use the notations from Fig. 3.3 for these). The Coq system ask us to give evidence of the goal on the right-hand side.

```
Instance HADD_Spec : Implement
(ι_a • ι_b) (* iso on inputs *)
(ι_s • ι_c) (* iso on outputs *)
HADD hadd.
```

```
I :  1_a ⊕ 1_b → 𝔹, O : 1_s ⊕ 1_c → 𝔹
H :  HADD ⊢ (1_a⊕1_b ; 1_s⊕1_c) I ⋈ O
=====================
@iso (ι_s • ι_c) O = hadd (@iso (ι_a • ι_b) I)
```

We have developed several tactics that help to prove these kind of goals. First, we automatically invert the derivation of the meaning relation in the hypothesis `H`, following the structure of the circuit, to get rid of parallel and serial combinators. This leaves the user with one meaning relation hypothesis per sub-component in the circuit (plugs included). Second, we use the type-class `Implement` as a dictionary of interesting properties. We use it to make fast-forward reasoning by applying `implements` in any hypothesis stating a meaning relation for a sub-component. The type-class resolution mechanism will look for an instance of `Implement` for this sub-component, and transform the "meaning relation" hypothesis into an equation. (Note that at this point, the user may have to interact with the proof-assistant, e.g., to choose other `Implement` instances than the ones that are picked automatically, but in many cases, this step is automatic.) At this point, the goal looks like the left-hand side of the following excerpt:

Figure 3.10 Operation on fixed-width integers

```
Record word (n:nat) := mk_word {val : Z; range: 0 ≤ val < 2^n}.   (* 𝕎_n *)

Definition repr n (x : Z) : 𝕎_n := ...
Definition high n m (x : 𝕎_(n+m)) : 𝕎_m := ...
Definition low n m (x : 𝕎_(n+m)) : 𝕎_n := ...
Definition combine n m (low : 𝕎_n) (high : 𝕎_m) : 𝕎_(n+m) := ...
Definition carry_add n (x y : 𝕎_n) (b : 𝔹) : 𝕎_n * 𝔹 :=
    let e := val x + val y + (if b then 1 else 0) in (e mod 2^n,2^n ≤ e)

Definition Φ^n_x : (n · 1_x → 𝔹) ≅ (𝕎_n) := ...
```

```
I : 1_a ⊕ 1_b → 𝔹, O : 1_s ⊕ 1_c → 𝔹
M : (1_a ⊕ 1_b) ⊕ (1_a ⊕ 1_b) → 𝔹
H0: iso M = (fun x ⇒ (x,x)) (iso I)
H1: iso (left O) = uncurry ⊕ (iso (left M))
H2: iso (right O)= uncurry ∧ (iso (right M))
============================
iso O = hadd (iso I)
```

```
I: 𝔹 * 𝔹, O: 𝔹 * 𝔹,
M : (𝔹 * 𝔹) * (𝔹 * 𝔹),
H0: M = (fun x ⇒ (x,x)) I
H1: fst O = uncurry ⊕ (fst M)
H2: snd O = uncurry ∧ (snd M)
====================
O = hadd I
```

Third, we move to the right-hand side of the excerpt: we massage the goal to make some instances of `iso` commute with the `left`, `right` and `app` operations, to push the applications of `iso` to the leaves. Then, we generalise the goal w.r.t. the sub-terms headed by `iso`. That is, all occurrences of, e.g., `iso O` of type $\mathbb{B} * \mathbb{B}$ are replaced with a fresh variable `O` of the same type. (Note that the user may be required to interact with Coq if different isos are applied to the same term in different equations.)

Finally, the proof context deals only with high-level data-types, and functions operating on these. The user may then prove the "interesting" part of the lemma, in the same way as it would have been done using a shallow-embedding.

3.3.2 n-bits integers

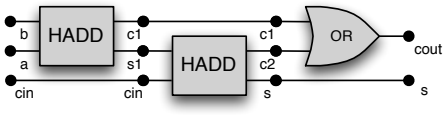
From now, we use a dependently typed definition of n-bits integers, along the lines of the fixed-size machine integers of [109]. We describe some operations on these integers in Fig. 3.10, but we omit the actual definitions of these functions when they can be inferred from the type. In the following, we prove that various (recursive) circuits implement the `carry_add` function (that adds two n-bit numbers and a carry) up-to the Φ isomorphism.

3.3.3 Two specifications of a 1-bit adder

A full-adder adds two 1-bit binary numbers with a carry-in, producing a 1-bit number and a carry-out, and is built from two half-adders. We present a diagram of this circuit, along with its formal definition in Fig. 3.11 (in which we omit the plugs, as we announced in §3.2).

From this circuit, we can derive two specifications of interest. First, the meaning of the full-adder can be expressed in terms of a Boolean function that mimics the truth-table of the circuit. Second, we can prove that this circuit actually implements the `carry_add` function up-to isomorphism. We prove these two specifications using the aforementioned tactics: the only difference is the content of the interesting parts.

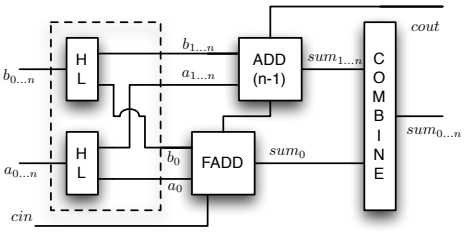
Figure 3.11 Definition of a full-adder




```
Context a b cin sum cout : string.
Program Definition FADD :
 ℂ (1_cin ⊕ (1_a ⊕ 1_b)) (1_sum ⊕ 1_cout) :=
  (ONE 1_cin & HADD a b "s1" "c1")
  ▷ ... (* associativity plug *)
  ▷ (HADD cin "s1" sum "c2" & ONE 1_"c1")
  ▷ ... (* associativity plug *)
  ▷ (ONE 1_sum & OR "c2" "c1" cout).
```

Figure 3.12 Implementation of the ripple-carry-adder from Fig. 3.2

```
Program Fixpoint RIPPLE cin a b cout sum n :
  ℂ (1_cin ⊕ n · 1_a ⊕ n · 1_b) (n · 1_sum ⊕ 1_cout) :=
match n with
  | O ⇒ ... (* Associativity *)
  | S p ⇒ ... ▷ (ONE (1_cin) & HIGHLOWS a b 1 p)
    ▷ ... ▷ (FADD a b cin sum "c" & ONE (p · 1_a ⊕ p · 1_b))
    ▷ ... ▷ (ONE (1 · 1_sum) & RIPPLE "c" a b cout s p)
    ▷ ... ▷ COMBINE sum 1 p & ONE (1_cout)
end.
```




```
Instance FADD_1 : Implement
(ι_cin • (ι_a • ι_b)) (* iso on inputs *)
(ι_sum • ι_cout) (* iso on outputs *)
FADD
(fun (c,(x,y)) ⇒ (x⊕(y⊕c),(x∧y)∨c∧(x⊕y))).
```

```
Instance FADD_2 : Implement
(ι_cin • (Φ¹_a • Φ¹_b)) (* iso on inputs *)
(Φ¹_sum • ι_cout) (* iso on outputs *)
FADD
(fun (c,(x,y)) ⇒ carry_add 1 x y c).
```

3.3.4 Ripple-carry adder

We present in Fig. 3.12 the formal definition of the ripple-carry adder from Fig. 3.2 (again, the plugs are omitted). This definition is based on two new circuits to split wires, and combine them. Indeed, to build a $1+n$-bit adder, the lowest-order wire of each parameter is connected to a full-adder, while the n high-order wires of each parameter are connected to another ripple-carry adder. Conversely, the wires corresponding to the sum must be combined together. We use two plugs to define the HL and the COMBINE circuits.

```
Definition HL x n p : ℂ ((n + p) · 1_x) (n · 1_x ⊕ p · 1_x):= Plug ...
Definition COMBINE x n p : ℂ (n · 1_x ⊕ p · 1_x) ((n + p) · 1_x):= Plug ...
```

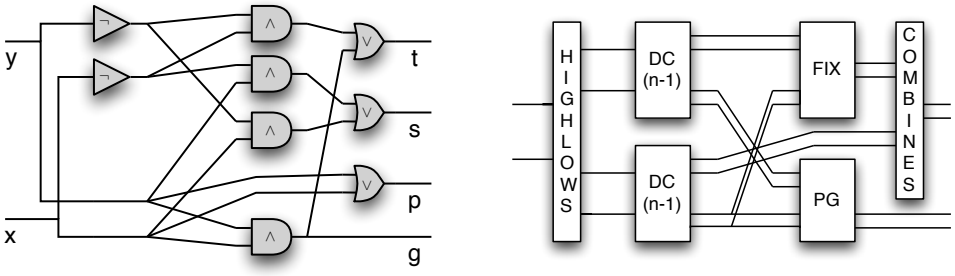
We prove that these functions on wires implement their counterparts on vectors, and then, on words. (These proofs are easy, yet not automatic.) Finally, these gates are easily combined two-by-two to build HIGHLOWS and COMBINES that work with two sets of wires at the same time to get more economical designs (i.e., designs with fewer sub-components).

```
Lemma HL_Spec x n p: Implement
(Φ^{n+p}_x) (Φ^n_x • Φ^p_x) (HL x n p)
(fun x ⇒ (low n p x, high n p x)).
```

```
Lemma COMBINE_Spec x n p: Implement
(Φ^n_x • Φ^p_x) (Φ^{n+p}_x) (COMBINE x n p)
(fun x ⇒ (combine n p (fst x) (snd x))).
```

Finally, we prove by induction on the size of the circuit that it implements the high-level carry_add addition function on words. (Note that this is a high-level specification of the circuit:

Figure 3.13 Divide and conquer adder



the `carry_add` function is not recursive and discloses nothing of the internal implementation of the device.) The interesting part of this proof boils down to the lemma `add_parts`, which we used as example in the previous chapter (§2.3.2).

```
Lemma add_parts n m (xH yH: word m) (xL yL : word n) cin:
  let (sumL,middle) := carry_add n xL yL cin in
  let (sumH,cout) := carry_add m xH yH middle in
  let sum := combine n m sumL sumH in
  carry_add (n + m) (combine n m xL xH)(combine n m yL yH) cin = (sum,cout).

Instance RIPPLE_Spec cin a b cout sum n : Implement (RIPPLE cin a b cout s n)
  (ι_cin • (Φ^n_a • Φ^n_b))  (Φ^n_sum • ι_cout)  (fun (c,(x,y)) ⇒ carry_add n x y c).
```

From the electrical engineer point of view, this design is simple (a linear chain of 1-bit adders) and slow (each full-adder must wait for the carry-in bit from the previous full-adders). In the next subsection, we address the case of a more efficient adder, which is more complicated, and a better benchmark for our methodology.

3.3.5 Divide and conquer adder

A *von Neumann adder* improves on the delay of the previous ripple-carry adder by using a divide and conquer scheme [2]. That is, the adder computes both the sum when there is a carry-in, and the sum when there is no carry-in. It is then possible to compute at the same time the sum for the high-order bits, and the sum for the low-order bits. Hence, we build a circuit that computes four pieces of data: s (resp. t), the n-bit sum of the inputs, assuming that there is no carry-in (resp. assuming that there is a carry-in); p the *carry-propagate* bit (resp. g the *carry-generate* bit), which is true when there is a carry-out of the circuit, assuming that there is a carry-in (resp. that there is no carry-in).

In a nutshell, the 2^{n+1}-adder circuit computes in parallel the 4-uple of results for the high-order and low-order part of the inputs, each of size 2^n. Then, the propagate and generate bits for both parts can be combined by the `PG` circuit to compute the propagate and generate bits for the entire circuit. In parallel, the `FIX` circuit is made of two 2^n-bit multiplexers (easily defined with a fixpoint using 1-bit multiplexers), and select the suitable high-order parts of the sum, w.r.t. the propagate and generate carry-bits of the low-order adder.

We provide two diagrams in Fig. 3.13 that depict the base case and the recursive case of the construction of `DC n`, the divide-and-conquer 2^n-adder, but we omit the actual Coq implementation for the sake of readability. We prove that this circuit implements the following Coq function:

Figure 3.14 Some useful isomorphisms

```
Definition Iso_stream A B C : ((A → B) ≅ C) → (A → stream B) ≅ (stream C) := ...
Definition Iso_stream_prod A B : (stream A ∗ stream B) ≅ (stream (A ∗ B)) := ...
Definition Iso_stream_vector A n : (vector (stream A) n) ≅ (stream (vector A n)) := ...
```

```
Definition dc n :𝕎_{2^n} ∗ 𝕎_{2^n} → 𝔹 ∗ 𝔹 ∗ 𝕎_{2^n} ∗ 𝕎_{2^n} := fun (x,y) ⇒
  let (s,g) := carry_add 2^n x y false in
  let (t,p) := carry_add 2^n x y true in (g,p,s,t).

Instance DC_Spec n : Implement (DC n) (Φ_x^{2^n} • Φ_y^{2^n}) (ι_g • ι_p • Φ_s^{2^n} • Φ_t^{2^n}) (dc n).
```

Note that this is again a high-level specification of the circuit w.r.t. its definition: the specification does not disclose the underlying computational behaviour. For instance, the `dc` function is not recursive. While the proof is conceptually as simple as the proof of the ripple-carry adder, and roughly 210 lines for the whole design, the definition of the plugs and their specification is quite verbose and amounts to a huge part of the remaining 400 lines of the file. We look forward to remove these rough edges by using thoroughly the Curry-Howard isomorphism we presented in §3.2.7.

3.4 Sequential circuits: time and loops

While we have focused the previous case studies on combinational circuits, our methodology can be applied to sequential circuits, with or without the loops that were allowed in the syntax of circuits in §3.2.4. In this section, we instantiate $\mathbb{T}$ with streams of Booleans (of type `nat` $\to \mathbb{B}$), `atom` with a two-case inductive definition that corresponds to the `NOR` and `REG` gates. We instantiate `spec` accordingly, stating that the `REG` delays a stream for one unit of time. That is, the gate `REG` implements the following `pre` function in the particular case of Booleans.

```
Definition pre {A} (d : A):
stream A → stream A := fun f t ⇒
match t with | 0 ⇒ d | S p ⇒ f p end.
```

```
Hypothesis REG_Realise_stream a out:
Implement (REG a out) (ι_a) (ι_out)
(pre false).
```

Recall that there may be several isomorphisms between two given types: to handle the complexity of reasoning with streams, we rely on a few isomorphisms, listed in Fig. 3.14. For instance, this is how we define the following bijections between valuations of n-ary disjoint sums and streams of vectors of Booleans or streams of n-bit words.

```
Remark useful_iso_1 n : (n · 1) → stream 𝔹 ≅ stream (vector 𝔹 n) := ...
Remark useful_iso_2 n : (n · 1) → stream 𝔹 ≅ stream (𝕎_n) := ...
```

Lifting combinational circuits. We first demonstrate that the parametricity of our definition of the semantics of circuits may be put to good use to prove the functional correctness of some designs in the Boolean setting, and then to mechanically lift this proof of correction to the Boolean stream setting, for the same set of gates. Recall that the meaning relation `Semantics`, denoted $\vdash$, is parametrised by the semantics of the basic gates: here, we shall drop this notation to make explicit this parameter.

We define an inductive predicate `wf` in Fig. 3.15 which is a characterisation of loop-less and delay-less circuits, built on gates that operate on streams in a point-wise manner. We first prove that the fact that this property on gates may be lifted to circuits. Then, we prove that if such a circuit implements a function `f` in the base setting, then, the same circuits implements the function `Stream.map f` in the stream setting.

Figure 3.15 Lifting combinational circuits

```
Variables (atom : Type → Type → Type) (𝕋 : Type).
Variables (specif1 : spec atom 𝕋) (specif2 : spec atom (stream 𝕋)).

(* value of the stream at time t *)
Definition time {n} (v: n → stream 𝕋) (t: nat): n → 𝕋 :=...

Definition wf_atom n m (c : atom n m) := ∀ ins outs (H : specif2 n m c ins outs),
  ∀ (t : nat), specif1 n m c (time ins t) (time outs t).

Inductive wf : ∀ n m (c : circuit atom n m), Prop :=
| wf_Atom : ∀ n m t, wf_atom n m t → wf n m (Atom atom t)
| wf_Ser : ∀ n m p x1 x2, wf n m x1 → wf m p x2 → wf n p (x1 ▷ x2)
| wf_Par : ∀ n m p q x1 x2, wf n p x1 → wf m q x2 → wf (n + m) (p + q) (x1 & x2)
| wf_Plug : ∀ n m f  , wf n m (Plug atom f).
```

```
Variables (n m : Type) (x : circuit atom n m) (Hwf: wf n m x).
Lemma lifting ins outs: Semantics atom (stream 𝕋) specif2 x ins outs →
  ∀ t, Semantics atom data specif1 x (time ins t) (time outs t).

Corollary lifting_map N M (Rn: (n → 𝕋) ≅ N) (Rm: (m → 𝕋) ≅ M) (f: N → M):
  Implements x Rn Rm f → Implements x (Iso_stream Rn) (Iso_stream Rm) (Stream.map f).
```

Note that the occurrences of `Semantics` in the `lifting_map` lemma are in a contravariant position, which explains the apparent reversal of the implication.

From a practical point of view, this result makes it possible to integrate seamlessly combinational circuits into sequential circuits, carrying the proofs in the former (simpler) setting.

A buffer. A `REG` delays one wire by one unit of time; a `FIFO` buffer generalises this behaviour in two dimensions, by chaining layers of `REG` one after another. While this circuit is simple, it is a good example for the use of high-level combinators that capture the underlying regularity in some common circuit patterns. For instance, consider the two following definitions, that replicate a sub-component in a serial or parallel manner.

```
Variable CELL : ℂ n n.
Fixpoint COMPOSEN k : ℂ n n :=
match k with
| 0 ⇒ Plug id
| S p ⇒ CELL ▷ (COMPOSEN p)
end.
```

```
Variable CELL : ℂ n m.
Fixpoint MAP k : ℂ (k · 1_n) (k · 1_m):=
match k with
| 0 ⇒ Plug id
| S p ⇒ CELL & (MAP p)
end.
```

We prove that the `COMPOSEN` combinator implements a higher-order iteration function, up-to isomorphism: if `CELL` implements a given function `f`, then `COMPOSEN k` implements the iteration of `f k` times. Respectively, we prove that the `MAP` circuit implements the higher-order `map` function on vectors. Hence, we define a `FIFO` buffer in one-line, and we prove that it implements the function below. While we do not detail the underlying proof, it is done in the same fashion as the proofs of combinational circuits.

```
Definition FIFO x n k : ℂ (k · 1_x) (k · 1_x) := COMPOSEN (MAP (REG x x) k) n.
Definition fifo n k (v : stream (vector 𝔹 k)) : stream (vector 𝔹 k) :=
  fun t ⇒ if n < t then v (t − n) else Vector.repeat k false.
Instance FIFO_Spec x n k : Implement (FIFO x n k) (useful_iso_1 n) (useful_iso_1 n) (fifo n k).
```

Figure 3.16 A memory element

```
Context a load out : string.
Program Definition MEMORY:
ℂ (1_load ⊕ 1_a) 1_out :=
Loop (1_load ⊕ 1_a) 1_out 1_out
(...   ▷ MUX2 a out load "in_reg"
▷ REG "in_reg" out ▷ Fork2 1_out).
```

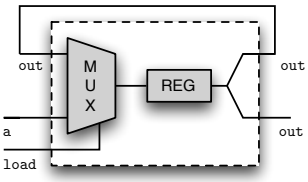


Figure 3.17 A shallow representation of Moore automata

```
Variable I O σ : Type.
Record Moore := { λ : σ → O; δ : σ → I → σ}.

Fixpoint iterate (M: Moore) (x: σ) (ins: stream I) (k: nat): σ :=
  match k with | 0 ⇒ x | S n ⇒ δ M (iterate M x ins n) (ins n) end.

Definition outputs (M: Moore) x (ins: nat → I) k : O := λ M (iterate M x ins k).
```

A memory element. Our next goal is to demonstrate how we deal with state-holding structures. Hence, we turn to the implementation of a 1-bit memory element, as implemented in Fig. 3.16. The memory is meant to hold 1-bit of information through time, which does not fit nicely in the `Implement` framework (which express the outputs of the circuit as a function of the sole inputs[2]). Hence, a first solution is to use a relational specification through the use of `Realise`:

```
Record reg_ti := {va : bool; vload : bool}.
Instance Memory_Spec : Realise (... :  1_load ⊕ 1_a → stream 𝔹 ≅ stream reg_ti) (ι_out) MEMORY
  (fun ins outs ⇒ outs = pre false (fun t ⇒ if  vload (ins t) then  va (ins t) else outs t)).
```

In this setting, the state of the memory is stored inside the history of the stream (the previous values that were taken by the output). While it is possible to reason about state-holding devices in this manner [74, 112], this is not the nicest way to reason about state-holding structures. Therefore, we move to the usual specification of synchronous circuits through *Moore machines*, a particular kind of finite state machine in which the internal state is updated w.r.t. the inputs, and the outputs are determined by the internal states (see Fig. 3.17). Hence, we specify the above memory w.r.t. its usual Moore automaton.

```
Definition reg_m : Moore reg_ti bool bool :=
  {|  λ := id; δ := fun state ins ⇒ if vload ins then va ins else state|}.
Instance Memory_Spec : Realise (... :  1_load ⊕ 1_a → stream 𝔹 ≅ stream reg_ti) (ι_out) MEMORY
(fun ins outs ⇒ ∀ t, outs t = Moore.outputs reg_m false ins t).
```

Notice that a straightforward composition of such specifications amounts to the standard composition from automata theory, taking the Cartesian product of the state sets and combining the transition functions accordingly. However, in the definition of Moore automata from Fig. 3.17, remark that the type of the states σ may be an arbitrary Coq type. For instance, this makes it possible to use a rich specification as a refinement of the state-space of the underlying automaton. We leave a more thorough investigation of state-holding devices to future work.

[2]Therefore, it is not possible to use recursively the definition of the outputs when defining the outputs.

3.5 Comparing shallow-embeddings and deep-embeddings

We have presented the core of our deep-embedding of circuits in Coq, and some simple case studies. We now turn to put this work in perspective, thoroughly discussing the pros and cons of shallow and deep embeddings of circuits. We distinguish three possibilities:

a) modelling circuits as predicates written in a subset of the logic of the theorem prover (synchronous circuits being represented as predicates over streams);
b) modelling circuits as functions written in a subset of the programming language of the theorem prover (synchronous circuits being modelled as transition functions of finite state systems, i.e., functions of type $\alpha \times \sigma \to \beta \times \sigma$);
c) modelling circuits as a data-type, endowed with a meaning function.

Concrete case studies. Using b), defining a circuit is no more difficult than writing a program. This makes it possible to design and verify sizable artifacts. For instance, a formal proof of a microprocessor designed using the PVS proof assistant was conducted in the recent VAMP project [26] and the complexity of this design is claimed to be comparable with industrial controllers with built-in floating point units. Generally speaking, using a) or c) requires to make explicit the wiring in the definitions of circuits, which induces an overhead when doing proofs. Recall that we developed some tactics to handle this overhead, but the overall ease of use is not on par with the comfort provided by b), and thus, the case studies are less impressive.

Simulating and checking designs. Using predicates to model hardware does not permit to simulate the circuits in order to get rid of logical errors, before proving them. However, the description of hardware as functions makes it easy to simulate the circuits: this is only a matter of running the function in the host theorem prover. Using a deep-embedding, this must be implemented somehow. In our particular case, we define a simulation function that yields a computational denotation of a given circuit. (Note that this requires the user to provide a suitable interpretation of each basic gate.)

```
Variable sem : ∀ a b, atom a b → (a → 𝕋) → option (b → 𝕋).
Fixpoint simulation n m (c : circuit atom n m) : (n → 𝕋) → option (m → 𝕋) := ...
```

For the sake of simplicity, we only consider loop-free circuits here, hence the `option` in the type of the result. However, if we moved from Booleans to the three-valued Scott's domain (unknown, true, false), we could envision computing fixpoints of the simulation function using Knaster-Tarski's theorem to handle loops. More generally, moving to the three-valued Scott's domain would allow us to consider the constructive semantics of circuits [143].

Notice that this simulation function defines a (partial) map from valuations to valuations which may be lifted to a (partial) map with a more meaningful type using type isomorphisms. For instance, this makes it possible to validate that the (simulated) outputs of the adders of §3.3 conform to their computational specification, before proving that they meet this specification.

```
Definition test n x y cin :=
let I := @uniso (ι • Φ^n • Φ^n) (cin ,x, y)
in match sim (RIPPLE n) I with
| None ⇒ None
| Some x ⇒ Some (@iso (Φ^n • ι) x)
end.
```

```
Eval compute in
  test 4 (Word.repr 4 8) (Word.repr 4 7) false.
  (* Some ({| val := 15; ...|} , false) *)
Eval compute in
  test 4 (Word.repr 4 8) (Word.repr 4 7) true.
  (* Some ({| val := 0; ...|} , true) *)
```

Meta-properties of circuits. Using the same ideas than for simulation, we may give other interpretations [27] of circuits defined using a deep-embedding. For instance, we can compute

the number of gates used, or the length of the critical path of a combinational circuit and thus prove that a given design meets some gate-count or delay complexity properties. As an example, we could check that the recurrence relation expressing the delay D or the number of gates G of the divide-and-conquer 2^n-adder from §3.3.5 are given by:

$$D(n) = 3(1+n)$$
$$G(n) = 3(n+5)2^n - 6$$

Dealing with this kind of intensional properties seems unfeasible using a shallow-embedding. Indeed, the above quantities must be computed at a meta-level and cannot be used in theorems.

Compilation and circuit transformations. Given an embedding of circuits in a theorem prover, we may consider two kinds of operations: synthesis of circuits (that is, compilation from an arbitrary embedding of a programming language to circuits); or transformations from circuits to circuits. Broadly speaking, these operations are translations from one language, to another.

One may take the radical view [145] that each of these languages is embodied by particular kind of terms of the programming language of the theorem prover, and not by particular abstract syntax tree types. (That is, program syntax and operational semantics need not to be formalised since all "languages" happen to be terms enjoying exactly the same semantics.) In this context, all the above transformations are implemented at the meta-level, and they may be verified as particular cases of translation validation [127]. Yet, there is no strong evidence that this is possible beyond some particular simple compilation schemes; one cannot prove general results of semantics preservation for these translations; and it is not possible to argue formally about the termination or the completeness of such meta-level programs.

Generally, shallow-embeddings of source languages do not permit to define the aforementioned kind of transformations as programs written inside the theorem prover: predicates and functions cannot be structurally analysed[3] by other functions.

However, Boolean functions may be analysed *computationally*: given a (combinational) fonction from (tuples of) Booleans to (tuples of) booleans, one could compute its underlying truth table using Shannon's formula. This truth table may be represented as a Binary Decision Diagram (BDD) and used to perform analysis and simplifications on the underlying circuit. Synchronous Decision Diagrams [156] generalise the construction of BDDs to the representation of (sequential) circuits. The construction of an SDDs from a given function f on streams of Booleans is by nature an infinite process that terminates if and only if the function f may be implemented by some finite state automaton. But constructing an SDD from a given finite circuit may yield a circuit that is less efficient than the original one. To the best of our knowledge, such transformations from shallow-embeddings of circuits to a deep-embedding have never been implemented in theorem provers. Indeed, reasoning about the (non-terminating) SDD construction algorithm may be hard.

Therefore, the use of a deep-embedding of circuits seem almost mandatory to implement and prove circuits-to-circuits transformations.

For instance, Cortadella, Kishinevsky and Grundmann [58] present a protocol to build latency-insensitive (or *elastic*) designs that is claimed to be amenable to efficient implementation, and they describe how usual synchronous circuits may be made elastic in an automatic way. We still need to investigate whether this elasticisation method may be easily implemented using our particular deep-embedding, and if possible, conduct a formal proof of semantics preservation.

[3] Their abstract syntax tree are not available

Implementable constructs. Remark that a deep-embedding of circuits makes it mandatory to settle on a given set of atoms. In our particular case, Coq's strong normalisation ensures that terms of type $\mathbb{C}\ n\ m$ reduce to constructors of the inductive definition of circuits. We actually defined a Coq function that computes a net-list representation of a given circuit, under the extra hypothesis that all its interface types are finite (which is not a strong requirement for real-life circuits). Remark that this function amounts to a mere pretty-printing of the circuits in our setting: it suffices to affect fresh names to ports of basic gates and to track the connections induced by the combinators. We do not prove properties about this pretty-printer.

Using a shallow-embedding, computing a net-list representation of a circuit must be done at the meta-level, and there are two sources of non-implementable constructs. First, the underlying language elements used in a circuit definition may be difficult to interpret in hardware. As an example, we may ponder on how to translate the application of an higher-order map function to a bit-vector. Second, several simplification may be used to model circuits. For instance, [89] studies the proof of correctness of an out-of-order execution processor with a reorder buffer, which is implemented as an unbounded buffer for the sake of simplicity. Yet, this is not directly implementable in hardware.

Summary. First, we have argued that it is currently easier to conduct the verification of sizable artifacts, like the VAMP project [26], using a shallow-embedding. However, we have also argued that going through a deep-embedding is required to verify circuits transformations, or to prove meta-properties of circuits. In the next section, we build on the above discussion to discuss several related works.

3.6 Comparisons with related work

In this presentation, our approach was to study how to create provably correct hardware inside a proof assistant. We shall not discuss the reverse approach, translating a conventional HDL description of a circuit to a proof assistant to do *a posteriori* verification of circuits: we argue that to verify circuits efficiently in a theorem prover, one has to exploit the structure of the designs, and in particular, their recursive structure. Neither shall we discuss model-checking methods [47]: we are interested in total functional correctness, not on partial verification.

Shallow-embeddings of circuits.

There has been a substantial amount of work on specification and verification of hardware in HOL. In [74, 112], HOL is used as an hardware description language and as a formalism to prove that a design meets its specification. They model circuits as predicates in the logic, using a shallow-embedding that merges the architecture of a circuit (a predicate) and its behaviour (another predicate). These seminal works demonstrated how to model iterated structures, and how to reason about temporal concepts.

Building on the former methodology, Slind et al [146] define a compiler from a synthesisable subset of HOL that creates correct-by-construction clocked synchronous hardware implementations. This approach is a particular case of translation validation [127]: the input function is refined by a proof procedure (i.e., a tactic) that uses rewriting to build a theorem that contains as a sub-term the synthesised circuit and a proof of its correctness. The generated hardware implementation is a predicate, that expresses the connection of primitive hand-shake devices. This formula may then be syntactically mapped to some Verilog constructs.

The synthesisable subset of HOL considered is at first limited to combinational (tail-recursive) functions mapping words to words. Functions that do not fall in this subset must be refined to a

tail-recursive form manually or using special purpose proof procedures (the authors provide one such tool that transforms linear-recursion to tail-recursion). Data-refinements from functions on other types than words must be also done by hand. Since the transformation from functions to circuits is done via automatic rewriting of suitable lemmas, users may tinker with the proof scripts to extend the synthesisable subset of HOL with e.g., let-expressions, or to optimise the translation. This methodology allows the designer to focus on high-level abstraction instead of reasoning and verifying at the gate level. However, this gives little control on the generated circuit, and the subset of HOL that may be synthesised this way is rather small. By contrast, our work complements their behavioural "correct by design" synthesis from a subset of the high-level language of the theorem-prover with bottom-up structural verification of circuits.

In the Boyer-Moore theorem prover (untyped, quantifier-free and first-order), Brock and Hunt proved the correctness of functions that generate correct hardware designs. Their framework allows for the definition of Mealy machines and their hierarchical composition. They first studied the correctness of an arithmetic and logic unit, parametrised by a size [95]. The same approach was then used to verify a simple, non-pipelined microprocessor design [36]. The verification of these examples fall out of what may be proved automatically using the Boyer-Moore theorem prover and requires to provide it with proofs scripts. By contrast, our case studies are less ambitious, but we may argue that our use of dependent-types to rule out ill-formed circuits simplifies their first-order, untyped presentation of circuits.

In Coq, Paulin-Mohring [122] proved the correctness of a multiplier unit, using a shallow-embedding: sequential circuits are modelled as Moore automata. More recently, Coupet-Grimal and Jakubiec [59] investigated how to take advantage of Coq's dependent types and co-inductive types in hardware verification: they use a shallow embedding of Moore and Mealy automata to describe and reason about sequential circuits. We still need to investigate some examples of sequential circuits studied in these papers.

In PVS, Beyer et al [25, 26] model circuits as functions of the PVS programming language. Then, they define an external and non-verified tool, `pvs2hdl`, that translates PVS programs to Verilog. The considered subset of PVS allows one to define (recursive, combinational) functions that have bits and bit-vectors as inputs and outputs, and allows for parameterised designs (e.g., a ripple-carry adder of arbitrary size). It is then extended to model clocked circuits, using a shallow-embedding of Mealy automata. The translation to hardware is defined as a syntactic mapping from PVS constructs to Verilog constructs. For instance, function definitions in PVS correspond to the definition of hardware modules in Verilog, while function calls correspond to the usage of such modules. (Note that this non-trivial transformation may require to unfold high-level constructions of the language like unrolling recursive definitions, or expanding functions applications.) Non-synthesisable constructs may not appear in the definitions of circuits: for instance, this prevents to use higher-order functions on bit-vectors. By contrast, arbitrary Coq constructs may appear in our definition of circuits: strong normalisation ensures that terms of type circuit are synthesisable.

Deep-embeddings of circuits.

Melham [112] gave a deep-embedding of circuits in HOL at the transistor level and defined meaning functions w.r.t. two models of the transistors behaviour: a simple switch model (where values are represented by Booleans), and a threshold model (where values are represented in the three-valued Scott's domain). Then, he studied under which conditions on the circuits the two models agree. By contrast, our deep-embedding of circuits is more high-level (we do not deal with transistors), but our circuits are still amenable to high-level proofs of functional correctness.

Hunt and Reeber [96] formalised the **DE2** language in ACL2, the successor of the Boyer-Moore theorem prover. The **DE2** language defines a deep-embedding of finite state machines, that are then amenable to formal reasoning, through the evaluation of the **DE2** definitions to ACL2 models. Since the underlying system is limited to first-order logic, it is not possible to prove semantics preservation for the kind of circuits transformations we mentioned in §3.5.

Verification of microprocessors.

Using the Boyer-Moore theorem prover, Brock and Hunt verified the simple, non-pipelined FM9001 processor [36]. Later, using ACL2, Sawada and Hunt [138] showed the correctness of an entire out-of-order processor using a Tomasulo's scheduler, with precise interrupts and a store buffer.

More recently, an outstanding work in microprocessors verification using theorem proving was conducted in the VAMP project [26]. Using PVS, the authors designed and verified:

> *"a processor with full DLX instruction set, delayed branch, Tomasulo scheduler, maskable nested precise interrupts , pipelined fully IEEE 754 compatible dual precision floating point units with variable latency, as well as separate, coherent instruction and data caches."*

This microprocessor was then synthesised and implemented on an FPGA. Arguably, this design is by far the most complex processor formally verified to date.

In Coq, Arditi [8] studied the correctness of some hardware circuits in the more general context of microprocessor verification. The complexity of the studied circuits is roughly similar to our case studies, and not on par with the two previous examples.

Algebraic definitions of circuits.

Circuit diagrams have nice algebraic properties. Lafont [107] studied the algebraic theory of Boolean circuits, and described rewriting systems and the associated canonical forms of circuits. While he mainly considered the "basic" Boolean circuits, his methods may be useful to study reversible Boolean circuits or quantum Boolean circuits. We look forward to apply similar algebraic methods to reason about circuits formalised using our deep-embedding.

Hinze [82] studied the algebraic structure of parallel prefix circuits. That is, he defined combinators that make it possible to describe succinctly all standard designs of this restricted class of circuits, and to prove them correct using algebraic reasoning. In particular, note that this class of circuits encompasses some classical adder circuits, e.g. carry-lookahead adders, or parallel sorting circuits. Defining these combinators on top of our deep-embedding of circuits should make it easy to define and verify parallel prefix circuits.

Functional languages in hardware design.

Sheeran [141] made a thorough review of the use of functional languages in hardware design, and of the challenges to address. Our work is a step towards one of them: the design and verification of parametrised designs, through the use of circuit combinators.

Lava [27] is a language embedded in Haskell to describe circuits, allowing one to define parametric circuits or higher-order combinators. While much of our goals are common, one key difference is that our encoding of circuits in Coq avoids the use of bound variables (we use only combinators). Moreover, we use dependent types, that are required to deal precisely with

parametric circuits. Finally, we prove the correctness of these parametric circuits in Coq, while verification in Lava is reduced to the verification of finite-size circuits.

Bluespec is a proprietary language that may be described as an extension of Haskell, extending the language to handle the design of circuits. Bluespec builds on the TRAC language [83] that compiles high-level description of circuits described as term-rewriting systems to circuits (expressed at the register transfer level, in Verilog).

Ghica [69] gave a denotational semantics of a functional language with imperative features, based on Reynolds' Syntactic Control of Interference [133], in terms of handshake circuits (which are incidentally shown to form a closed monoidal category). One advantage of the presented synthesis technique is the simplicity of the generated circuits: for instance, abstraction, applications, sequential composition, assignment or dereferencing of variables reduce to mere wiring.

Synchronous languages in hardware design.

Synchronous languages such as Esterel [19], Lustre [78], SIGNAL [108] and others may be used to describe the functional specification of hardware or software components of embedded systems. Such synchronous programs may then be translated to hardware implementations or synthesised to hardware circuits. Case studies in an industrial context [20] gave empirical evidence that the use of synchronous languages such as Esterel could help to build correct hardware design through compilation, that were one order of magnitude less verbose than their counterparts developed using conventional hardware description language. That is, the use of high-level primitives allows for more economical designs. Moreover, synchronous languages have been endowed with formal semantics. For instance, Schneider [139] studied an embedding of the Esterel programming language in HOL, and proved correct its translation to equations systems of guarded commands (which are a representation of hardware circuits).

3.7 Conclusion

We have presented a deep-embedding of circuits in Coq that allows one to build and reason about gate-level circuits, yet proving high-level specifications through the use of type-isomorphisms. We have hinted that dependent types are useful to prove automatically some well-formedness conditions on the circuits, and help to avoid time consuming mistakes. Then, we demonstrated how to prove by induction the correctness of some arithmetic circuits of parametric size: this could not have been possible without mimicking the structure of the usual circuit diagrams to define circuit generators in Coq. Note that the code of this library is rather small: around 4000 lines, including the examples. Finally, we conclude this chapter with potential directions for future works.

More case studies. We could continue the case studies described in §3.3. In particular, we would like to investigate how to construct parallel prefix circuits in our framework [82, 141], or to investigate the design-space of combinational and sequential multipliers, to yield real arithmetic and logic units.

Other settings. We also look forward to study how our methodology applies to other settings than Booleans or streams of Booleans. For instance, if we move from Booleans to the three-valued Scott's domain (unknown, true, false), we may interpret circuits in the so-called constructive semantics. We also hope that some of our methods could be applied to the probabilistic setting.

Figure 3.18 A dependently-typed `WHILE` language

```
Inductive aexp (E: list nat) : nat → Type :=
| AVar: ∀ n (i : Var E n), aexp E n
| AConst: ∀ n, Word.word n → aexp E n
| APlus: ∀ n, aexp E n → aexp E n → aexp E n
| ALo: ∀ n m, aexp E (n + m) → aexp E n
| AHi: ∀ n m, aexp E (n + m) → aexp E m
| ACat: ∀ n m, aexp E n → aexp E m → aexp E (n + m).
```

```
Inductive bexp (E: list nat) : Type :=
| BTrue: bexp E
| BFalse: bexp E
| BEq: ∀ n, aexp E n → aexp E n → bexp E
| BLt: ∀ n, aexp E n → aexp E n → bexp E
| BNot: bexp E → bexp E
| BAnd: bexp E → bexp E → bexp E.
```

```
Inductive com (E: list nat) : Type :=
| CSkip: com E
| CAss: ∀ n (v : Var E n) , aexp E n → com E
| CSeq: com E → com E → com E
| CIf: bexp E → com E → com E → com E
| CWhile: bexp E → com E → com E
| CNew: ∀ n, aexp E n → com (snoc E n) → com E
```

Front-ends. A preliminary investigation indicates that our definition of circuits is a good target for the compilation of a dependently typed variant of the `WHILE` language (see Fig. 3.18) defined along the lines of Ghica's Geometry of Synthesis [69]. Therefore, an interesting alternative to the gate-level description of, e.g., sequential multipliers or their use inside an iterated divider would be to generate these circuits from an high-level `WHILE` program. This would be a particular example of *behavioural synthesis*, that is, the generation of circuits using high-level front-ends. Then, we could envision to mix circuits defined in our "assembly" language of combinators with generated circuits.

Conclusion

Contributions and conclusion

We followed two different lines of research in our work. The first one deals with the addition of general purpose automation to Coq: using a proof assistant, an user may expect help when the goals belong to some decidable logical fragments. The second one deals with the development of particular formalisations in Coq: either "gratuitous" ones (our deep-embedding of hardware circuits), or "necessary" ones to build reflexive tactics (the proofs leading to Kozen's theorem). We summarise our contributions below, which interleave these two lines of works through the definition of reflexive decision procedures.

An efficient decision procedure for Kleene algebras. First, we presented a reflexive tactic for deciding the equational theory of Kleene algebras. This tactic relies on a careful implementation of efficient finite automata algorithms, so that it solves casual equations instantaneously and properly scales to larger expressions. The underlying decision procedure is proved correct and complete: correctness is established w.r.t. any model by formalising Kozen's initiality theorem; and a counter-example is returned when the given equation does not hold. The correctness proof is challenging: it involves both a precise analysis of the underlying automata algorithms and a lot of algebraic reasoning. In particular, we had to formalise the theory of matrices over a Kleene algebra.

Tools for rewriting modulo AC. Second, we presented a set of tools for rewriting AC, solving a long-standing practical problem. We use two building blocks: an extensible reflexive decision procedure for equality modulo AC; and an OCaml plug-in for pattern matching modulo AC. We handle associative only operations, neutral elements, uninterpreted function symbols, and user-defined equivalence relations. We defined a new reification method based on the use of type-classes: we can infer these properties automatically, so that end-users do not need to specify which operation is A or AC, or which constant is a neutral element.

A library to verify hardware circuits. Third, we presented a new library to model and verify hardware circuits in Coq. This library allows one to easily build circuits by following the usual pen-and-paper diagrams. The main novelty of our contribution is that we define a deep-embedding: we use a (dependently typed) data-type that models the architecture of circuits, and a meaning function. We proposed tactics that ease the reasoning about the behaviour of the circuits, and we demonstrated that our approach is practicable by proving the correctness of various circuits.

Perspectives

The use of combinational reflection. In his survey and critique on the use of reflection, Harrison [81] wrote:

> *"The idea of computational reflection is not to make a formal system stronger but rather to make its deductive process more efficient by utilising information which avoids having to construct formal proofs in full detail."*

We shall ponder this statement. First, consider the particular example of equality modulo AC. Indeed, our decision procedure alleviates the need to produce a proof of equality in full detail, but this would be tractable (regardless of the size of the proofs). However, the above quote makes perfect sense in the case of our decision procedure for Kleene algebras: constructing a formal proof of a given arbitrary theorem directly from the rules of Kleene algebras is difficult. Even if, from a theoretical point of view, an algorithm to compute derivations could be extracted from a proof of correction of the decision procedure, it is not straightforward to use our development to do so; moreover, such an algorithm may be unusable in practice.

It might be the case that this particular use of combinational reflection is actually simpler than exhibiting proof-trees using the rules of Kleene algebras, but we look forward to investigate to what extent the scales could be tipped in favour of the use of an external solver coupled with some traces verification.

An automata library. We could expand our library of automata constructions to provide a complete library dedicated to formal languages. First, we should improve on our preliminary implementations of minimisation algorithms. Then, our development could be supplemented with theoretical results such as the pumping lemma, or the Myhill-Nerode theorem. We could also implement decision procedures for emptiness or finiteness of regular sets, or certified `grep`-like tools.

A more ambitious project would be to implement an automata-based decision procedure for the Weak Second-Order Theory of 1 Successor, WS1S [42]. Note that Presburger arithmetic (with existential and univeral quantifiers) is definable as a restricted fragment of WS1S, and that we could revisit *en cours de route* the decision procedure implemented by Berghofer and Reiter [18] in Isabelle. In another decidable fragment of WS1S, Basin and Klarlund [15] demonstrated how to represent classes of combinational or sequential circuits (for instance, n-bit adders), and perform automatic verification of some correctness properties. Although the decision problem for WS1S formula is of non-elementary complexity, Basin and Klarlund report that their decision procedure may be used in practice. The usability of a Coq implementation is an open question.

Formal behavioural synthesis. By contrast with our structural description of circuits, behavioural descriptions aim to specify circuits by their input/output behaviours without describing their implementation details. Then, hardware implementations are synthesised through optimising compilation. There is empirical evidence [83] that, on big designs, such synthesis of high-level description yields more efficient implementations than hand-optimised low-level description circuits. Indeed, hand-optimisations have diminishing returns, and do not scale well beyond small examples. Moreover, inter-block synchronisation mechanisms must be implemented on an ad-hoc basis—and are usually scattered throughout the whole circuit, thus difficult to maintain. Among behavioural description languages, term-rewriting systems [83], statically allocated functional languages [114] or synchronous languages [19] have been endowed with formal semantics. We look forward to study to what extent behavioural synthesis from one

of these languages is amenable to formal verification. While most surely, optimising compilation should be out of reach, simpler synthesis schemes and simpler circuits-to-circuits transformations (for instance, elasticisation [58]) are more likely to be amenable to formal proof.

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] A. V. Aho and J. D. Ullman. *Foundations of Computer Science*. Computer Science Press, W. H. Freeman and Company, 1992.

[3] J. Alglave and L. Maranget. "Stability in Weak Memory Models". In: *Proc. CAV*. Vol. 6806. LNCS. Springer, 2011, pp. 50–66.

[4] S. F. Allen, R. L. Constable, D. J. Howe, and W. E. Aitken. "The Semantics of Reflected Proof". In: *Proc. LICS*. IEEE Computer Society, 1990, pp. 95–105.

[5] J. Bacelar Almeida, N. Moreira, D. Pereira, and S. Melo de Sousa. "Partial Derivative Automata Formalized in Coq". In: *Proc. CIAA*. Vol. 6482. LNCS. Springer, 2010, pp. 59–68.

[6] C. Alvarado. "Réflexion pour la réécriture dans le calcul des constructions inductives". PhD thesis. Université Paris XI, 2002.

[7] C. Alvarado and Q.-H. Nguyen. "ELAN for Equational Reasoning in Coq". In: *Proc. LFM*. INRIA, 2000.

[8] L. Arditi. "Spécification et preuve de microprocesseurs". PhD thesis. Université de Nice - Sophia Antipolis, 1996.

[9] M. Armand, B. Grégoire, A. Spiwack, and L. Théry. "Extending Coq with Imperative Features and Its Application to SAT Verification". In: *Proc. ITP*. Vol. 6172. LNCS. Springer, 2010, pp. 83–98.

[10] L. Bachmair, I. V. Ramakrishnan, A. Tiwari, and L. Vigneron. "Congruence Closure modulo Associativity-Commutativity". In: *Proc. FroCos*. Vol. 1794. LNCS. Springer, 2000, pp. 242–256.

[11] D. Baelde. "Private communication". 2011.

[12] B. Barras, J.-P. Jouannaud, P.-Y. Strub, and Q. Wang. "CoqMTU: a higher-order type theory with a predicative hierarchy of universes parametrized by a decidable first-order theory". In: *Proc. LICS*. 2011.

[13] G. Barthe. "Implicit Coercions in Type Systems". In: *Proc. TYPES*. Vol. 1158. LNCS. Springer, 1995, pp. 1–15.

[14] G. Barthe, M. Ruys, and H. Barendregt. "A Two-Level Approach Towards Lean Proof-Checking". In: *Proc. TYPES*. LNCS. Springer, 1995, pp. 16–35.

[15] D. Basin and N. Klarlund. "Automata Based Symbolic Reasoning in Hardware Verification". In: *Formal Methods In System Design* 13 (1998). Extended version of: "Hardware verification using monadic second-order logic," *CAV '95*, LNCS 939, pp. 255–288.

[16] D. Benanav, D. Kapur, and P. Narendran. “Complexity of Matching Problems”. In: *Proc. RTA*. Vol. 202. LNCS. Springer, 1985, pp. 417–429.

[17] N. Benton, C.-K. Hur, A. Kennedy, and C. McBride. “Strongly Typed Term Representations in Coq”. In: *Journal of Automated Reasoning* published on-line March 2011 (), pp. 1–19.

[18] S. Berghofer and M. Reiter. “Formalizing the Logic-Automaton Connection”. In: *Proc. TPHOLs*. 2009, pp. 147–163.

[19] G. Berry. “The foundations of Esterel”. In: *Proof, Language, and Interaction, Essays in Honour of Robin Milner*. The MIT Press, 2000. ISBN: 978-0-262-16188-6.

[20] G. Berry, M. Kishinevsky, and S. Singh. “System Level Design and Verification Using a Synchronous Language”. In: *Proc. ICCAD*. IEEE Computer Society / ACM, 2003, pp. 433–440.

[21] Y. Bertot. *Coq in a Hurry*. Tech. rep. 2010. URL: http://cel.archives-ouvertes.fr/inria-00001173/en/.

[22] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.

[23] Y. Bertot, G. Gonthier, S. Ould Biha, and I. Pasca. “Canonical Big Operators”. In: *Proc. TPHOL*. Vol. 5170. LNCS. Springer, 2008, pp. 86–101. URL: http://dx.doi.org/10.1007/978-3-540-71067-7_11.

[24] G. Betarte and A. Tasistro. “Formalisation of systems of algebras using dependent record types and subtyping: an example”. In: *Proc. 7th Nordic workshop on Programming Theory*. 1995.

[25] S. Beyer, C. Jacobi, D. Kroening, and D. Leinenbach. “Correct Hardware by Synthesis from PVS”. 2002. URL: http://www-wjp.cs.uni-saarland.de/publikationen/BJKL02.pdf.

[26] S. Beyer, C. Jacobi, D. Kröning, D. Leinenbach, and W. J. Paul. “Putting it all together - Formal verification of the VAMP”. In: *STTT* 8.4-5 (2006), pp. 411–430.

[27] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. “Lava: Hardware Design in Haskell”. In: *Proc. ICFP*. ACM Press, 1998, pp. 174–184.

[28] F. Blanqui, S. Coupet-Grimal, W. Delobel, and A. Koprowski. *CoLoR: a Coq Library on Rewriting and Termination*. 2006.

[29] A. Boudet, J.-P. Jouannaud, and M. Schmidt-Schauß. “Unification in Boolean Rings and Abelian Groups”. In: *J. Symb. Comput.* 8.5 (1989), pp. 449–477.

[30] S. Boutin. “Using Reflection to Build Efficient and Certified Decision Procedures”. In: *Proc. TACS*. Vol. 1281. LNCS. Springer, 1997, pp. 515–529.

[31] R. S. Boyer and J S. Moore. *A Computational Logic*. New York, NY: Academic Press, 1979.

[32] R. S. Boyer and J. S. Moore, eds. *The Correctness Problem in Computer Science*. Academic Press, 1981.

[33] T. Braibant and D. Pous. *Coq library: ATBR, Algebraic tools for working with binary relations*. http://sardes.inrialpes.fr/~braibant/atbr/. 2009.

[34] T. Braibant and D. Pous. *Tactics for working modulo AC in Coq*. Coq library available at http://sardes.inrialpes.fr/~braibant/aac_tactics/. 2010.

[35] S. Briais. *Coq development: Finite Automata Theory*. http://www.prism.uvsq.fr/~bris/tools/Automata_080708.tar.gz. 2008.

[36] B. Brock and W. A. Hunt Jr. "The DUAL-EVAL Hardware Description Language and Its Use in the Formal Specification and Verification of the FM9001 Microprocessor". In: *Formal Methods in System Design* 11.1 (1997), pp. 71–104.

[37] C. Brown and G. Hutton. "Categories, Allegories and Circuit Design". In: *Proc. LICS*. IEEE Computer Society, 1994, pp. 372–381.

[38] A. Brüggemann-Klein. "Regular expressions into finite automata". In: *Theoretical Computer Science* 120.2 (1993), pp. 197–213.

[39] N. G. De Bruijn. "Telescopic mappings in typed lambda calculus". In: *Information and Computation* 91 (1991), pp. 189–204.

[40] R. E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". In: *IEEE Trans. Computers* 35.8 (1986), pp. 677–691.

[41] J. A. Brzozowski. "Derivatives of Regular Expressions". In: *J. ACM* 11.4 (1964), pp. 481–494.

[42] J. R. Büchi. "Weak Second-Order Arithmetic and Finite Automata". In: *Mathematical Logic Quarterly* 6.1-6 (1960), pp. 66–92.

[43] J.-M. Champarnaud, F. Nicart, and D. Ziadi. "From the ZPC Structure of a Regular Expression to its Follow Automaton". In: *IJAC* 16.1 (2006), pp. 17–34.

[44] J.-M. Champarnaud, F. Ouardi, and D. Ziadi. "Normalized Expressions and Finite Automata". In: *IJAC* 17.1 (2007), pp. 141–154.

[45] J.-M. Champarnaud and D. Ziadi. "Computing the Equation Automaton of a Regular Expression in Space and Time". In: *Proc. CPM*. Vol. 2089. LNCS. Springer, 2001, pp. 157–168.

[46] A. Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2011.

[47] E. M. Clarke, E. A. Emerson, and J. Sifakis. "Model checking: algorithmic verification and debugging". In: *Commun. ACM* 52 (11 2009), pp. 74–84. ISSN: 0001-0782.

[48] M. Clavel et al. "The Maude 2.0 System". In: *Proc RTA*. Vol. 2706. LNCS. Springer, 2003.

[49] C. Sacerdoti Coen and E. Tassi. "Working with Mathematical Structures in Type Theory". In: *Proc. TYPES*. Vol. 4941. LNCS. Springer, 2007, pp. 157–172.

[50] S. Conchon and J.-C. Filliâtre. "A Persistent Union-Find Data Structure". In: *ACM SIGPLAN Workshop on ML*. Freiburg, Germany, 2007, pp. 37–45.

[51] E. Contejean. "A Certified AC Matching Algorithm". In: *Proc. RTA*. Vol. 3091. LNCS. Springer, 2004, pp. 70–84.

[52] E. Contejean and H. Devie. "An Efficient Algorithm for Solving Systems of Diophantine Equations". In: *Information and Computation* 113.1 (1994), pp. 143–172.

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

[54] T. Coquand and G. P. Huet. "The Calculus of Constructions". In: *Inf. Comput.* 76.2/3 (1988), pp. 95–120.

[55] T. Coquand and C. Paulin. "Inductively defined types". In: *Conference on Computer Logic*. Vol. 417. LNCS. Springer, 1988, pp. 50–66.

[56] T. Coquand and V. Siles. “A Decision Procedure for Regular Expression Equivalence in Type Theory”. In: *Proc. CPP*. Vol. 7086. LNCS. Springer, 2011, pp. 119–134.

[57] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. Second. MIT Press, 2001.

[58] J. Cortadella, M. Kishinevsky, and B. Grundmann. “Synthesis of synchronous elastic architectures”. In: *Proc. DAC*. ACM, 2006, pp. 657–662.

[59] S. Coupet-Grimal and L. Jakubiec. “Certifying circuits in Type Theory”. In: *Formal Asp. Comput.* 16.4 (2004), pp. 352–373.

[60] H. Doornbos, R. Backhouse, and J. van der Woude. “A Calculational Approach to Mathematical Induction”. In: *Theoretical Computer Science* 179.1-2 (1997), pp. 103–135.

[61] P. J. Downey, R. Sethi, and R. E. Tarjan. “Variations on the Common Subexpression Problem”. In: *J. ACM* 27.4 (1980), pp. 758–771.

[62] S. Eker. “Associative-Commutative Rewriting on Large Terms”. In: *Proc. RTA*. Vol. 2706. LNCS. Springer, 2003, pp. 14–29.

[63] S. Eker. “Single Elementary Associative-Commutative Matching”. In: *Journal of Automated Reasoning* 28.1 (2002), pp. 35–51.

[64] J.-C. Filliâtre. *Finite Automata Theory in Coq: A constructive proof of Kleene’s theorem*. Research Report 97–04. LIP - ENS Lyon, 1997. URL: ftp://ftp.ens-lyon.fr/pub/LIP/Rapports/RR/RR97/RR97-04.ps.Z.

[65] S. Foster, G. Struth, and T. Weber. “Automated Engineering of Relational and Algebraic Methods in Isabelle/HOL - (Invited Tutorial)”. In: *Proc. RAMICS*. Vol. 6663. LNCS. Springer, 2011, pp. 52–67.

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

[67] F. Garillot, G. Gonthier, A. Mahboubi, and L. Rideau. “Packaging Mathematical Structures”. In: *Proc. TPHOL*. Vol. 5674. LNCS. Springer, 2009, pp. 327–342.

[68] H. Geuvers, R. Pollack, F. Wiedijk, and J. Zwanenburg. “A Constructive Algebraic Hierarchy in Coq”. In: *Journal of Symbolic Computation* 34.4 (2002), pp. 271–286.

[69] D. R. Ghica. “Geometry of synthesis: a structured approach to VLSI design”. In: *Proc. POPL*. 2007, pp. 363–375.

[70] G. Gonthier. “Point-Free, Set-Free Concrete Linear Algebra”. In: *Proc. ITP*. Vol. 6898. LNCS. Springer, 2011, pp. 103–118.

[71] G. Gonthier. “The Four Colour Theorem: Engineering of a Formal Proof”. In: *ASCM*. Vol. 5081. LNCS. Springer, 2007, p. 333.

[72] G. Gonthier and A. Mahboubi. “An introduction to small scale reflection in Coq”. In: *Journal of Formalized Reasoning* 3.2 (2010), pp. 95–152.

[73] G. Gonthier, B. Ziliani, A. Nanevski, and D. Dreyer. “How to make ad hoc proof automation less ad hoc”. In: *Proc. ICFP*. ACM, 2011, pp. 163–175.

[74] M. Gordon. *Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware*. Tech. rep. UCAM-CL-TR-77. Cambridge Univ., Computer Lab, 1985.

[75] B. Grégoire and X. Leroy. “A compiled implementation of strong reduction”. In: *Prox. ICFP*. 2002, pp. 235–246.

[76] B. Grégoire and A. Mahboubi. “Proving equalities in a commutative ring done right in Coq”. In: *Proc. TPHOL*. Vol. 3603. LNCS. Springer, 2005, pp. 98–113.

[77] D. Gries. “Describing an algorithm by Hopcroft”. In: *Acta Informatica* 2 (1973), pp. 97–109.

[78] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. “The synchronous dataflow programming language Lustre”. In: *Proceedings of the IEEE* 79.9 (1991), pp. 1305–1320.

[79] F. K. Hanna, N. Daeche, and M. Longley. “Veritas$^+$: A Specification Language Based on Type Theory”. In: *Hardware Specification, Verification and Synthesis*. LNCS. Springer, 1989, pp. 358–379.

[80] J. R. Harrison. “A HOL Theory of Euclidean space”. In: *Proc. TPHOL*. Vol. 3603. LNCS. Springer, 2005, pp. 114–129.

[81] J. Harrison. *Metatheory and Reflection in Theorem Proving: A Survey and Critique*. Tech. rep. CRC-053. SRI Cambridge, 1995.

[82] R. Hinze. “An Algebra of Scans”. In: *MPC*. LNCS. Springer, 2004, pp. 186–210.

[83] J. C. Hoe and Arvind. “Hardware Synthesis from Term Rewriting Systems”. In: *Proc. VLSI*. Vol. 162. IFIP Conference Proceedings. Kluwer, 1999, pp. 595–619.

[84] P. Höfner and G. Struth. “Automated Reasoning in Kleene Algebra”. In: *Proc. CADE*. Vol. 4603. LNCS. Springer, 2007, pp. 279–294.

[85] P. Höfner and G. Struth. “On Automating the Calculus of Relations”. In: *Proc. IJCAR*. Vol. 5195. LNCS. Springer, 2008, pp. 50–66.

[86] J. E. Hopcroft. *An nlog n algorithm for minimizing states in a finite automaton*. Tech. rep. Stanford University, 1971.

[87] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to automata theory, languages, and computation, 2nd edition*. Vol. 32. New York, NY, USA: ACM, 2001, pp. 60–65.

[88] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Reading, Massachusets, USA: Adison-Wesley Publishing Company, 1979.

[89] R. Hosabettu, M. Srivas, and G. Gopalakrishnan. “Proof of Correctness of a Processor with Reorder Buffer Using the Completion Functions Approach”. In: *Proc. CAV*. Vol. 1633. LNCS. Springer, 1999, pp. 686–686.

[90] J. M. Hullot. “Associative Commutative pattern matching”. In: *Proc. IJCAI*. 1979, pp. 406–412.

[91] L. Ilie and S. Yu. “Follow automata”. In: *Information and Computation* 186.1 (2003), pp. 140–162.

[92] P. Jipsen. “From Semirings to Residuated Kleene Lattices”. In: *Studia Logica* 76.2 (2004), pp. 291–303.

[93] P. Johnstone. *Topos theory*. Academic Press, 1977.

[94] J.-P. Jouannaud and H. Kirchner. “Completion of a Set of Rules Modulo a Set of Equations”. In: *SIAM J. Comput.* 15.4 (1986), pp. 1155–1194.

[95] W. A. Hunt Jr. and B. Brock. “The Verification of a Bit-slice ALU”. In: *Hardware Specification, Verification and Synthesis*. Vol. 408. LNCS. Springer, 1989, pp. 282–306.

[96] W. A. Hunt Jr. and E. Reeber. “Formalization of the DE2 Language”. In: *Proc. CHARME*. Vol. 3725. LNCS. Springer, 2005, pp. 20–34.

[97] W. Kahl. “Calculational Relation-Algebraic Proofs in Isabelle/Isar”. In: *Proc. RelMiCS*. Vol. 3051. LNCS. Springer, 2003, pp. 178–190.

[98] S. C. Kleene. “Representation of Events in Nerve Nets and Finite Automata”. In: *Automata Studies*. Princeton University Press, 1956, pp. 3–41.

[99] G. Klein et al. “seL4: Formal Verification of an OS Kernel”. In: *Proc. SOSP*. ACM, 2009, pp. 207–220.

[100] D. Kozen. “A completeness theorem for Kleene algebras and the algebra of regular events”. In: *Information and Computation* 110.2 (1994), pp. 366–390.

[101] D. Kozen. “Kleene algebra with tests”. In: *Transactions on Programming Languages and Systems* 19.3 (1997), pp. 427–443.

[102] D. Kozen. “On Kleene Algebras and Closed Semirings”. In: *Proc. MFCS*. Vol. 452. LNCS. Springer, 1990, pp. 26–47.

[103] D. Kozen. *Typed Kleene algebra*. TR98-1669, CS Dpt. Cornell University. 1998.

[104] A. Krauss and T. Nipkow. “Proof Pearl: Regular Expression Equivalence and Relation Algebra”. In: *Journal of Automated Reasoning* ?? (published online March 2011), ??–??

[105] D. Krob. “Complete Systems of B-Rational Identities”. In: *Theoretical Computer Science* 89.2 (1991), pp. 207–343.

[106] D. Kroening and O. Strichman. *Decision Procedures – an Algorithmic Point of View*. EATCS. Springer, 2008.

[107] Y. Lafont. “Towards an algebraic theory of Boolean circuits”. In: *Journal of Pure and Applied Algebra* 184 (2003), pp. 257–310.

[108] P. LeGuernic, T. Gautier, M. Le Borgne, and C. Le Maire. “Programming real-time applications with SIGNAL”. In: *Proceedings of the IEEE* 79.9 (1991), pp. 1321 –1336.

[109] X. Leroy. “A formally verified compiler back-end”. In: *Journal of Automated Reasoning* 43.4 (2009), pp. 363–446.

[110] C. Marché. “On Ground AC-Completion”. In: *Proc. RTA*. Vol. 488. LNCS. Springer, 1991, pp. 411–422.

[111] U. Martin and T. Nipkow. “Ordered Rewriting and Confluence”. In: *Proc. CADE*. Vol. 449. LNCS. Springer, 1990, pp. 366–380.

[112] T. Melham. *Higher Order Logic and Hardware Verification*. Vol. 31. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1993.

[113] A.R. Meyer and L. J. Stockmeyer. “Word problems requiring exponential time”. In: *Proc. STOC*. ACM, 1973, pp. 1–9. URL: http://doi.acm.org/10.1145/800125.804029.

[114] A. Mycroft and R. Sharp. “A Statically Allocated Parallel Functional Language”. In: *Proc. ICALP*. Vol. 1853. LNCS. Springer, 2000, pp. 37–48.

[115] J. Narboux. “Formalisation et automatisation du raisonnement géométrique en Coq”. PhD thesis. Université Paris Sud, 2006.

[116] G. Nelson and D. C. Oppen. “Fast Decision Procedures Based on Congruence Closure”. In: *J. ACM* 27.2 (1980), pp. 356–364.

[117] A. Nerode. “Linear automaton transformations”. In: *Proc. of the AMS*. Vol. 9. 1958, pp. 541–544.

[118] Q. H. Nguyen, C. Kirchner, and H. Kirchner. “External Rewriting for Skeptical Proof Assistants”. In: *Journal of Automated Reasoning* 29.3-4 (2002), pp. 309–336.

[119] T. Nipkow. “Equational Reasoning in Isabelle”. In: *Sci. Comp. Prg.* 12.2 (1989), pp. 123–149.

[120] T. Nipkow. “Proof Transformations for Equational Theories”. In: *Proc. LICS*. IEEE Computer Society, 1990, pp. 278–288.

[121] D. von Oheimb and T.F. Gritzner. “RALL: Machine-Supported Proofs for Relation Algebra”. In: *Proc. CADE*. Vol. 1249. LNCS. Springer, 1997, pp. 380–394.

[122] C. Paulin-Mohring. “Circuits as Streams in Coq: Verification of a Sequential Multiplier”. In: *Proc. TYPES*. 1995, pp. 216–230.

[123] G. Peterson and M. Stickel. “Complete Sets of Reductions for Some Equational Theories”. In: *J. ACM* 28.2 (1981), pp. 233–264.

[124] F. Pfenning. “Unification and Anti-Unification in the Calculus of Constructions”. In: *Proc. LICS*. IEEE Computer Society, 1991, pp. 74–85.

[125] B. C. Pierce, C. Casinghino, M. Greenberg, V. Sjoberg, and B. Yorgey. *Software Foundations*. Distributed electronically, 2011.

[126] G. Plotkin. “Building in Equational Theories”. In: *Machine Intelligence 7* (1972).

[127] A. Pnueli, M. Siegel, and E. Singerman. “Translation Validation”. In: *Proc. TACAS*. LNCS. London, UK: Springer, 1998, pp. 151–166.

[128] H. Poincaré. *La science et l'hypothèse*. Flammarion, 1902.

[129] D. Pous. “Untyping Typed Algebraic Structures and Colouring Proof Nets of Cyclic Linear Logic”. In: *Proc. CSL*. Vol. 6247. LNCS. Springer, 2010, pp. 484–498.

[130] M.O. Rabin and D. Scott. “Finite Automata and Their Decision Problems”. In: *IBM Journal of Research and Development* 3(2) (1959), pp. 114–125.

[131] B. Razet. “Machines d'Eilenberg Effectives”. PhD thesis. Université Paris Diderot, 2009.

[132] V. Redko. “On defining relations for the algebra of regular events”. In: *Ukrain. Mat. Z.* 16 (1964), pp. 120–126.

[133] J. C. Reynolds. “Syntactic Control of Interference”. In: *Proc. POPL*. 1978, pp. 39–46.

[134] David M. Russinoff. “A Case Study in Fomal Verification of Register-Transfer Logic with ACL2: The Floating Point Adder of the AMD Athlon$^{\text{TM}}$ Processor”. In: *Proc. FMCAD*. Vol. 1954. LNCS. Springer, 2000, pp. 3–36.

[135] J. J. M. M. Rutten. “Automata and Coinduction (An Exercise in Coalgebra)”. In: *Proc. CONCUR*. Vol. 1466. LNCS. Springer, 1998, pp. 194–218.

[136] J. Sakarovitch. “Kleene's theorem revisited”. In: *Trends, Techniques, and Problems in Theoretical Computer Science*. Vol. 281. LNCS. Springer Berlin / Heidelberg, 1987, pp. 39–50.

[137] A. Salomaa. “Two Complete Axiom Systems for the Algebra of Regular Events”. In: *J. ACM* 13 (1 1966), pp. 158–169. ISSN: 0004-5411.

[138] J. Sawada and W. A. Hunt Jr. “Verification of FM9801: An Out-of-Order Microprocessor Model with Speculative Execution, Exceptions, and Program-Modifying Capability”. In: *Formal Methods in System Design* 20.2 (2002), pp. 187–222.

[139] K. Schneider. “Embedding Imperative Synchronous Languages in Interactive Theorem Provers”. In: *Proc. ACSD*. IEEE Computer Society, 2001, pp. 143–.

[140] P. Selinger. “A Survey of Graphical Languages for Monoidal Categories”. In: *New Structures for Physics*. Vol. 813. Lecture Notes in Physics. Springer, 2011, pp. 289–355.

[141] M. Sheeran. “Hardware Design and Functional Programming: a Perfect Match”. In: *J. UCS* 11.7 (2005), pp. 1135–1158.

[142] M. Sheeran. "μFP, A Language for VLSI Design". In: *LISP and Functional Programming*. 1984, pp. 104–112.

[143] T. R. Shiple, G. Berry, and H. Touati. "Constructive Analysis of Cyclic Circuits". In: *Proc. EDTC*. IEEE Computer Society, 1996, pp. 328–333.

[144] K. Slind. "AC Unification in HOL90". In: *Proc. HUG*. Vol. 780. LNCS. Springer, 1993, pp. 436–449.

[145] K. Slind, G. Li, and S. Owens. "Compiling Higher Order Logic by Proof". In: *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Springer, 2010, pp. 193–220. DOI: 10.1007/978-1-4419-1539-9_7.

[146] K. Slind, S. Owens, J. Iyoda, and M. Gordon. "Proof producing synthesis of arithmetic and cryptographic hardware". In: *Formal Asp. Comput.* 19.3 (2007), pp. 343–362.

[147] M. Sozeau and N. Oury. "First-Class Type Classes". In: *Proc. TPHOL*. Vol. 4732. LNCS. Springer, 2008, pp. 278–293.

[148] B. Spitters and E. van der Weegen. "Type Classes for Mathematics in Type Theory". In: *MSCS, special issue on 'Interactive theorem proving and the formalization of mathematics'* 21 (2011). (A four pages abstract appeared in Proc. ITP 2010.), pp. 1–31.

[149] J. Michael Spivey. "Algebras for combinatorial search". In: *J. Funct. Program.* 19.3-4 (2009), pp. 469–487.

[150] P.-Y. Strub. "Coq Modulo Theory". In: *Proc. CSL*. Vol. 6247. LNCS. Brno, Czech Republic: Springer, 2010, pp. 529–543.

[151] G. Struth. "Calculating Church-Rosser Proofs in Kleene Algebra". In: *Proc. RelMiCS*. Vol. 2561. LNCS. Springer, 2001, pp. 276–290.

[152] A. Tarski and S. Givant. *A Formalization of Set Theory without Variables*. Vol. 41. Colloquium Publications. AMS, 1987.

[153] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.3*. 2010.

[154] R. Thiemann and C. Sternagel. "Certification of Termination Proofs Using CeTA". In: *Proc. TPHOL*. Vol. 5674. LNCS. Springer, 2009, pp. 452–468.

[155] K. Thompson. "Regular Expression Search Algorithm". In: *C. ACM* 11 (1968), pp. 419–422.

[156] J. Vuillemin. "On circuits and numbers". In: *IEEE Transactions on Computers* 43 (1994), pp. 868–879.

[157] F. Wiedijk, ed. *The Seventeen Provers of the World*. Vol. 3600. LNCS. Springer, 2006.

[158] C. Wu, X. Zhang, and C. Urban. "A Formalisation of the Myhill-Nerode Theorem Based on Regular Expressions (Proof Pearl)." In: *Proc. ITP*. Vol. 6898. LNCS. Springer, 2011, pp. 341–356.

Abstract

This thesis describes three formalisations in Coq. The first chapter is devoted to the implementation of an efficient decision procedure for Kleene algebras : as regular languages are the initial model of Kleene algebras, we can resort to finite automata algorithms to solve equations in an arbitrary Kleene algebra. The second chapter presents a set of tools for rewriting modulo associativity and commutativity built using two components: a reflexive decision procedure for equality modulo AC and an OCaml plug-in for pattern matching modulo AC. The third chapter defines a deep-embedding of hardware circuits using dependent types that is used to model and prove the functional correctness of parametrised circuits.

Résumé

Cette thèse décrit trois travaux de formalisation en Coq. Le premier chapitre s'intéresse à l'implémentation d'une procédure de décision efficace pour les algèbres de Kleene, pour lesquelles le modèle des langages réguliers est initial : il est possible de décider la théorie équationelle des algèbres de Kleene via la construction et la comparaison d'automates finis. Le second chapitre est consacré à la définition de tactiques pour la réécriture modulo associativité et commutativité en utilisant deux composants : une procédure de décision réflexive pour l'égalité modulo AC, ainsi qu'un greffon OCaml implémentant le filtrage modulo AC. Le dernier chapitre esquisse une formalisation des circuits digitaux via un plongement profond utilisant les types dépendants de Coq ; on s'intéresse ensuite à prouver la correction totale de circuits paramétriques.