Carlos Alberto da Silva Carvalho de Freitas

A Theory of Communicating Sequential Processes in Coq

Recife

2020

Carlos Alberto da Silva Carvalho de Freitas

A Theory of Communicating Sequential Processes in Coq

A B.Sc. Dissertation presented to the Centro de Informática of Universidade Federal de Pernambuco in partial fulfillment of the requirements for the degree of Bachelor in Computer Engineering.

Universidade Federal de Pernambuco

Centro de Informática

Bachelor in Computer Engineering

Supervisor: Gustavo Henrique Porto de Carvalho

Recife

2020

Acknowledgements

Abstract

Theories of concurrency such as Communicating Sequential Processes (CSP) allow system specifications to be expressed clearly and analyzed with precision. However, the state explosion problem, common to model checkers in general, is a real constraint when attempting to verify system properties for large systems. An alternative is to ensure these properties via proof development. This work will provide an approach on how we can develop a theory of CSP in the Coq proof assistant, and evaluate how this theory compares to other theorem prover-based frameworks for the process algebra CSP. We will implement an infrastructure for declaring syntactically and semantically correct CSP specifications in Coq, along with native support for process representation through Labelled Transition Systems (LTSs), in addition to traces refinement analysis.

Keywords: Process algebra. LTS. Traces refinement. CSP. Proof assistant. Coq. QuickChick.

Resumo

Teorias de concorrência tais como *Communicating Sequential Processes* (CSP) permitem que especificações de sistemas sejam descritas com clareza e analisadas com precisão. No entanto, o problema da explosão de estados, comum aos verificadores de modelo em geral, é uma limitação real na tentativa de verificar propriedades de um sistema complexo. Uma alternativa é garantir essas propriedades através do desenvolvimento de provas. Este trabalho fornecerá uma abordagem sobre como se pode desenvolver uma teoria de CSP no assistente de provas Coq, além de compará-la com outros frameworks baseados em provadores de teoremas para a álgebra de processos CSP. Portanto, será implementada uma infraestrutura para declarar especificações sintatica e semanticamente corretas de CSP em Coq, juntamente com um suporte nativo para a representação de processos por meio de Sistemas de Transições Rotuladas (LTSs), além de análise de refinamento no modelo de *traces*.

Palavras-chave: Álgebra de processos. LTS. Refinamento no modelo de *traces*. CSP. Assistente de provas. Coq. QuickChick.

List of Figures

List of Tables

List of abbreviations and acronyms

CSP	Communicating Sequential Processes
FDR	Failures-Divergence Refinement
LTS	Labelled Transition System
SOS	Structured Operational Semantics

Contents

1 Introduction

Concurrency is an attribute of any system that allows multiple components to perform operations at the same time. The understanding of this property is essential in modern programming because major areas, such as distributed and real-time systems, rely on this concept to work properly. As a result, the variety of applications enabled by the concurrency feature is broad: aircraft and industrial control systems, routing algorithms, peer-to-peer networks, client-server applications and parallel computation, to name a few.

Since concurrent systems may have parts that execute in parallel, the combination of ways in which these parts can interact raises the complexity in designing such systems. Phenomena like deadlock, livelock, nondeterminism and race condition can emerge from these interactions, so these issues must be addressed in order to avoid undesired behavior. Typically, testing cannot provide enough evidence to guarantee properties such as deadlock freedom, divergence freedom and determinism for a given system.

That being said, CSP (a theory for Communicating Sequential Processes) introduces a convenient notation that allows systems to be described in a clear and accurate way. More than that, it has an underlying theory that enables designs to be analysed and proven correct with respect to desired properties. The FDR (Failures-Divergence Refinement) tool is a model checker for CSP responsible for making this process algebra a practical tool for specification, analysis and verification of systems. System analysis is achieved by allowing the user to make assertions about processes and then exploring every possible behavior, if necessary, to check the truthfulness of the assertions made.

Although it is undeniable that FDR is a useful tool in the analysis of systems described in CSP, it has a limitation common to standard model checkers in general: the state explosion problem. An alternative way for deciding whether a system meets its specification is by proof development. Examples of this different approach are CSP-Prover and Isabelle/UTP, both frameworks based on the theorem prover Isabelle. Nevertheless, to the best of our knowledge, there is not a theory for CSP in the Coq proof assistant yet. Considering that, the main research question of this work is the following: how could we develop a theory of CSP in Coq, exploiting the main advantages of this proof assistant?

1.1 Objectives

The main objective (MO) of this work is to define in Coq a theory for concurrent systems, based on a limited scope of the process algebra CSP. This objective is unfolded into the following specific objectives (SO):

- SO1: study CSP and frameworks based on this process algebra.
- SO2: define a syntax for CSP in Coq, based on a restricted version of the CSP_M language (machine readable language for CSP).
- SO3: provide support for the LTS-based (Labelled Transition System) representation, considering the Structured Operational Semantics (SOS) of CSP.
- SO4: make use of the QuickChick tool to search for counterexamples of the traces refinement relation.

1.2 An overview of CSP_{Coq}

Consider the following CSP process adapted from Schneider (1999, p. 32, example 2.3). This process represents a cloakroom attendant that might help a costumer off or on with his coat, storing an retrieving coats as appropriate:

$$channel\ coat\_on, coat\_off, store, retrieve, request\_coat, eat$$

$$\begin{aligned} SYSTEM = {} & coat\_off \mathrel{-\!>} store \mathrel{-\!>} request\_coat \mathrel{-\!>} retrieve \mathrel{-\!>} coat\_on \mathrel{-\!>} SKIP \\ & [|\ \{coat\_off, request\_coat, coat\_on\}\ |] \\ & coat\_off \mathrel{-\!>} eat \mathrel{-\!>} request\_coat \mathrel{-\!>} coat\_on \mathrel{-\!>} SKIP \end{aligned}$$

We can declare such system in CSP_{Coq} by defining a specification, which consists in lists of channels and processes. This specification must also abide by a set of contextual rules that will be discussed further in this work.

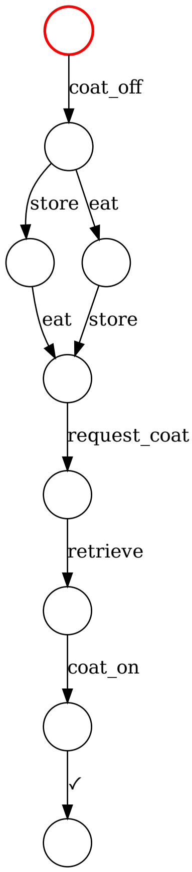
```
Definition example : specification.
Proof.
  solve_spec_ctx_rules (
    Build_Spec
    [ Channel {{"coat_off", "coat_on", "request_coat", "retrieve", "store", "eat"}} ]
    [ "SYSTEM" ::=
      "coat_off" -> "store" -> "request_coat" -> "retrieve" -> "coat_on" -> SKIP
      [| {{"coat_off", "request_coat", "coat_on"}} |]
      "coat_off" -> "eat" -> "request_coat" -> "coat_on" -> SKIP ]
  ).
Defined.
```

Furthermore, we can execute the following command to compute the process LTS and output the graph in dot language:

```
Compute generate_dot (compute_ltsR example "SYSTEM" 100).
```

The Figure 1 is a visual representation of the graph outputted by this command. The image is generated using GraphViz software.

Figure 1 – The process LTS graph.



1.3 Main contributions

The main contributions of this work are the following:

- Abstract and concrete syntax for a subset of CSP operators.
- Context rules for CSP specifications.
- Operational semantics via SOS approach.
- Inductive and functional definitions of a labelled transition system.
- Proof of correctness for the functional definition of the LTS.
- Inductive and functional definitions of traces.

- Proof of correctness for the functional definition of traces.
- Tactic macro that automates trace relation proofs.
- Formal definition of the traces refinement.
- Refinement verification using QuickChick.

1.4 Document structure

Apart from this introductory chapter, in which we discuss about the motivation behind this work and its main objective, and also take a quick look at an example that illustrates what can be done using the framework developed, this monograph contains three more chapters. The content of these chapters are detailed bellow:

Chapter 2 Discusses fundamental concepts such as CSP theory, SOS approach, trace refinement and LTS representation. Moreover, this chapter introduces the Coq proof assistant and its functional language Gallina, along with an introduction to proof development (tactics) and the Ltac language inside this tool, which gives support for developing tactic macros.

Chapter 3 Provides an in-depth look at the implementation of CSP_{Coq}, including its abstract and concrete syntax, and language semantics. Furthermore, the LTS process representation support, using the GraphViz software, is also detailed in this chapter.

Chapter 4 Concludes this monograph by presenting a comparison between the infrastructure described in this work and other interactive theorem provers based on CSP. It also addresses possible topics for future work.

2 Background

Before jumping into the specifics of the implementation of CSP_{Coq}, we need to understand some elements of the CSP language itself, such as the concrete syntax and the semantics defined in both denotational and operational models (section 2.1). Beyond that, it is also important to provide an overview of what an interactive theorem prover is: the Coq proof assistant fundamentals such as tactics, as well as the embedded Ltac language (section 2.2). We must also address the QuickChick property-based testing tool (section 2.3), which is the Coq implementation of QuickCheck (CLAESSEN; HUGHES, 2000). This chapter gives an introduction to each one of these concepts.

2.1 Communicating sequential processes

In 1978, Tony Hoare's *Communicating Sequential Processes* (HOARE, 1978) described a theory to help us understand concurrent systems, parallel programming and multiprocessing. More than that, it has introduced a way to decide whether a program meets its specification. This theory quickly evolved into what is known today as the CSP programming language. This language belongs to a class of notations known as process algebras, where concepts of communication and interaction are presented in an algebraic style.

Since the main goal of CSP is to provide a theory-driven framework for designing systems of interacting components an reasoning about them, we must introduce the concept of a component, or as we will be referencing it from now on, a *process*. Processes are self-contained entities that once combined they can describe a system, which is yet another larger process that may itself be combined as well with other processes. The way a process communicates with the environment is through its *interface*. The interface of a process is the set of all the events that the process has the potential to engage in. At last, an *event* represents the atomic part of the communication itself. It is the piece of information the processes rely on to interact with one another. A process can either participate actively or passively in a communication, depending on whether it performed or suffered the action. Events may be external, meaning they appear in the process interface; indicate termination, represented by the event $\checkmark$; or be internal, and therefore unknown for the environment, denoted by the event τ.

The most basic process one can define is $STOP$. Essentially, this process never interacts with the environment and its only purpose is to declare the end of an execution. In other words, it illustrates a deadlock: a state in which the process can not engage in any event or make any progress whatsoever. It could be used to describe a computer that failed booting because one of its components is damaged, or a camera that can no longer take pictures due to storage space shortage.

Another simple process is *SKIP*. It indicates that the process has reached a successful termination state, which also means that it has finished executing. We can use *SKIP* to illustrate an athlete that has crossed the finish line, or a build for a project that has passed.

Provided these two trivial processes, *STOP* and *SKIP*, and the knowledge of what a process interface is, we can apply a handful of CSP operators to define more descriptive processes. For example, let a be an event in the process P interface. One can write the new process P as $a \rightarrow STOP$, meaning that this process behaves as $STOP$ after performing a. This operator is known as the *event prefix*, and it is pronounced as "then".

The choice between processes can be constructed in two different ways in CSP: externally and internally. An *external choice* between two processes implies the ability to perform any event that either process can engage in. Therefore, the environment has control over the outcome of such decision. On the other hand, if the process itself is the only responsible for deciding which event from its interface will be communicated, thus which process it will resolve to, then we call it an *internal choice*. Note that this operator is essentially a source of non-deterministic behavior.

To illustrate the difference between these choice operators, consider the following scenario: a cafeteria may operate by either letting the costumers choose between ice cream and cake for desert, or by making this choice itself (employees decide), having the clients no take on what deserts they will get. In the first specification, the choice is external to the business and it might be described as $ice\_cream \rightarrow SKIP \ \Box \ cake \rightarrow SKIP$, whereas it is internal in the latter, thus $ice\_cream \rightarrow SKIP \sqcap cake \rightarrow SKIP$ would capture such business rule.

CSP introduces two approaches for describing a parallel execution between processes: the *alphabetized parallel* and the *generalized parallel*. Let A be the interface of process P, and B the interface of process Q. An alphabetized parallel combination of these processes is described as $P \ {}_{A}\|_{B} \ Q$. Events in the intersection of A and B must be simultaneously engaged in by the processes P and Q. In other words, an event that appears in both process interfaces can only be communicated if the two processes are ready to perform this event. Any other event that does not match this criteria can be engaged in by its corresponding process independently. The semantics are similar for the generalized version of the parallel operator. The only change being its constructor, that takes the synchronization alphabet alone as the interface argument the processes must agree upon. Let C be the intersection of previously defined interfaces A and B. The generalized parallel between process P and Q is written as $P \underset{C}{\|} Q$.

Both versions of the parallel operator may be used to describe a marathon where every participant is a process that runs in parallel with each other. They must all start the race at the same time, but they are not expected to cross the finish line all together. We can use the alphabetized parallel to specify the combination between two participants as $RUNNER1 \ {}_{\{start, finish1\}}\|_{\{start, finish2\}} \ RUNNER2$, or use the generalized version of the operator instead: $RUNNER1 \underset{\{start\}}{\|} RUNNER2$.

Another CSP operator that provides a concurrent execution of processes is the *interleaving* operator. Different from the parallel operators, the interleaving represents a combination of processes that do not require any synchronization at all. The processes applied to this operation execute totally independent of each other. This might be the case of two vending machines at a supermarket. They operate completely separate from each other, receiving payments, processing changes and releasing snacks. In other words, there is no dependency regarding the communication of events between the vending machines. That being said consider the process $VENDING\_MACHINE$ as $pay \rightarrow select\_snack \rightarrow return\_change \rightarrow release\_snack$. Then, the process that specifies both machines operating together is described as $VENDING\_MACHINE \mid\mid\mid VENDING\_MACHINE$.

The last two operators we will be discussing are the *sequential composition* and *event hiding*. Before we continue, the reader must be aware that there are others CSP operators for combining processes apart from the ones presented in this chapter, but they will not be supported by the framework implemented in this project.

Sometimes it is necessary to pass the control over execution from one process to another, and for that we use sequential composition. It means that the first process has reached a successful termination state and now the system is ready to behave as the second process in the composition. Parents can choose to let their children play only after completing their homework. That being the case, the process $CHILD$ could be modeled as $HOMEWORK\,; FUN$, where

$$HOMEWORK = choose\_subject \rightarrow study \rightarrow answer\_exercises \rightarrow SKIP$$
$$FUN = build\_lego \rightarrow watch\_cartoons \rightarrow play\_videogame \rightarrow SKIP$$

In this example, the process FUN can only be executed after the process $HOMEWORK$ has successfully terminated.

Last but not least, we have the event hiding operator. A system designer may choose to hide events from a process interface to prevent them from being recognized by other processes. That way, the environment can not distinguish this particular event, thus no process can engage in it. Event hiding proves to be useful when processes placed in parallel should not be allowed to synchronize on certain events. Consider, for example, that a school teacher is communicating each student individually his or her test grade. It has to be done in such way that no student gets to know other test grades besides his or her own. The process

TEACHER may be modeled as $show\_grade \rightarrow discuss\_questions \rightarrow SKIP$, so a teacher concerned with the students privacy can be described as $TEACHER \setminus \{show\_grade\}$.

2.1.1 Structured operational semantics

There are three major complementary approaches for describing and reasoning about the semantics of CSP programs. These are through *algebraic*, *denotational* (also called *behavioral*), and *operational semantics*. We will be focusing in the last one, which tries to understand all the actions and decisions that process implementations can make as they proceed.

The operational semantics for CSP language describes how a valid program is interpreted as sequences of computational steps. By evaluating the initial events of a process and finding out how it will behave immediately after performing them, this approach enables us to explore the state space of any process. All we need to do is repeat this step until we have covered the transition system picture of the process we are interested in.

It is traditional to present operational semantics as a logical inference system: Plotkin's SOS, or *Structured Operational Semantics* style. A process has a given action if, and only if, that is deducible from the rules given.

We start by analyzing the process *STOP*. Since it is unable to engage in any event whatsoever, there are no inference rules for it. Then, we move forward to the next primitive process: *SKIP*. While *STOP* has no actions of itself, *SKIP* is able to perform a single event, which is the termination event ✓. The lack of antecedents in the following rule means it is always the case that *SKIP* may perform ✓ and behave as *STOP*.

$$\frac{}{SKIP \xrightarrow{\checkmark} STOP}$$

The event prefix operation also spares the antecedents in its inference rule, so the conclusion is immediately deduced: if the process is initially able to perform a, then after performing a it behaves like P.

$$\frac{}{(a \rightarrow P) \xrightarrow{a} P}$$

The transition rules for external choice reflect the fact that the first external event resolves the choice in favor of the process performing the event. In addition, as we can see in the first two rules, the choice is not resolved on the occurrence of internal events. Control over resolution of the choice is external because the events of both choices are initially available.

$$\frac{P \xrightarrow{\tau} P'}{P \,\Box\, Q \xrightarrow{\tau} P' \,\Box\, Q}$$

$$\frac{Q \xrightarrow{\tau} Q'}{P \mathbin{\Box} Q \xrightarrow{\tau} P \mathbin{\Box} Q'}$$

$$\frac{P \xrightarrow{a} P'}{P \mathbin{\Box} Q \xrightarrow{a} P'} \quad (a \neq \tau)$$

$$\frac{Q \xrightarrow{a} Q'}{P \mathbin{\Box} Q \xrightarrow{a} Q'} \quad (a \neq \tau)$$

The internal choice is an operation that guarantees the process to behave as either of its components on any execution. This state change happens "silently", thus this transition is followed by the communication of internal event τ, as we can see in the inference rules for this operation.

$$\frac{}{P \sqcap Q \xrightarrow{\tau} P}$$

$$\frac{}{P \sqcap Q \xrightarrow{\tau} Q}$$

We can separate the rules for the alphabetized parallel into two categories: one that describes the independent execution of each process, and other defining the synchronized step performed at once by the components. The first two inference rules capture the ability of both sides performing events that are not in the common interface, thus executing them independently. The third rule dictates the joint step, where both processes are able to perform the event, so they communicate it at the same time.

$$\frac{P \xrightarrow{\mu} P'}{P \;{}_{A}\|_{B}\; Q \xrightarrow{\mu} P' \;{}_{A}\|_{B}\; Q} \quad (\mu \in (A \cup \{\, \tau \,\} \setminus B))$$

$$\frac{Q \xrightarrow{\mu} Q'}{P \;{}_{A}\|_{B}\; Q \xrightarrow{\mu} P \;{}_{A}\|_{B}\; Q'} \quad (\mu \in (B \cup \{\, \tau \,\} \setminus A))$$

$$\frac{P \xrightarrow{a} P' \qquad Q \xrightarrow{a} Q'}{P \;{}_{A}\|_{B}\; Q \xrightarrow{a} P' \;{}_{A}\|_{B}\; Q'} \quad (a \in A^{\checkmark} \cap B^{\checkmark})$$

The transition rules for the generalized parallel are very similar to the ones for the previous operation. The main difference lies in the side condition, since this version of parallelism is only interested in the interface alphabet. The same rule categories for the alphabetized parallel apply to this operation.

$$\frac{P \xrightarrow{\mu} P'}{P \underset{A}{\|} Q \xrightarrow{\mu} P' \underset{A}{\|} Q} \quad (\mu \notin A^{\checkmark})$$

$$\frac{Q \xrightarrow{\mu} Q'}{P \underset{A}{\parallel} Q \xrightarrow{\mu} P \underset{A}{\parallel} Q'} \quad (\mu \notin A^{\checkmark})$$

$$\frac{P \xrightarrow{a} P' \qquad Q \xrightarrow{a} Q'}{P \underset{A}{\parallel} Q \xrightarrow{a} P' \underset{A}{\parallel} Q'} \quad (a \in A^{\checkmark})$$

The interleave operation describes a parallel execution between processes that do not synchronize in any event except termination $\checkmark$. In other words, this operation is a particular case of the generalized parallelism, where the interface alphabet is empty, thus the event $\checkmark$ being the only event that can be performed simultaneously by the components.

$$\frac{P \xrightarrow{\mu} P'}{P \mid\mid\mid Q \xrightarrow{\mu} P' \mid\mid\mid Q} \quad (\mu \neq \checkmark)$$

$$\frac{Q \xrightarrow{\mu} Q'}{P \mid\mid\mid Q \xrightarrow{\mu} P \mid\mid\mid Q'} \quad (\mu \neq \checkmark)$$

$$\frac{P \xrightarrow{\checkmark} P' \qquad Q \xrightarrow{\checkmark} Q'}{P \mid\mid\mid Q \xrightarrow{\checkmark} P' \mid\mid\mid Q'}$$

As we already know, the hiding operator removes all events in a given alphabet from the process interface, preventing other processes to engage in them. The process to which the event hiding is applied can then behave just like it would without the operator, except the events in the given alphabet are made internal and then renamed to τ. Such behavior is capture by the inference rules:

$$\frac{P \xrightarrow{a} P'}{P \setminus A \xrightarrow{\tau} P' \setminus A} \quad (a \in A)$$

$$\frac{P \xrightarrow{\mu} P'}{P \setminus A \xrightarrow{\mu} P' \setminus A} \quad (\mu \notin A)$$

The last operational rules we need to discuss are for the sequential composition operator. Initially, this combination behaves as the process to the left of the operator until it terminates. Then, the execution control is granted to the other process in the composition. The control handover is represented by the communication of the internal event τ, as we can see in the second rule:

$$\frac{P \xrightarrow{a} P'}{P; Q \xrightarrow{a} P'; Q} \quad (a \neq \checkmark)$$

$$\frac{P \xrightarrow{\checkmark} P'}{P; Q \xrightarrow{\tau} Q}$$

2.1.2 Traces refinement

A pretty reasonable way for gathering information from a process interacting with the environment is by keeping track of the events this process engages in. This sequence of communication between process and environment, presented in a chronological order, is what we call a *trace*. Traces can either be finite or infinite, and it depends on the observation span and the nature of the process itself.

Because this record is easily observed by the environment and it represents a single interaction, it is often used to build models of CSP processes. As a matter of fact, there is one named after it: the *traces* model, represented by the symbol $\mathcal{T}$. It defines the meaning of a process expression as the set of sequences of events (traces) that the process can be observed to perform. This model is one of the three major denotational models of CSP, the other ones being the *stable failures* $\mathcal{F}$ and the *failures-divergences* model $\mathcal{N}$.

The notion of refinement is a particularly useful concept for specifying the correctness of a CSP process. If we can establish a relation between components of a system which captures the fact that one satisfies at least the same conditions as another, then we may replace a worse component by a better one without degrading the properties of the system.

Definition. (Traces Refinement) Let P and Q be two CSP processes, and *traces* be a function that yields the set of all possible traces of a given CSP process, we say that Q trace-refines P if, and only if, every trace of Q is also a trace of P:

$$P \sqsubseteq_{\mathcal{T}} Q \iff \mathit{traces}(Q) \subseteq \mathit{traces}(P)$$

If we consider P to be a specification which determines possible safe states of a system, then we can think of $P \sqsubseteq_{\mathcal{T}} Q$ as saying that Q is a safe implementation: no wrong events will be allowed.

2.1.3 Machine-readable version of CSP

In the beginning, CSP was typically used as a blackboard language. In other words, it was conceived to describe communicating and interacting processes for a human audience. Theories such as CSP have a higher chance of acceptance among the industry and academy (e.g. for teaching purposes) when they have tool support available. For that reason, the need of a notation that could actually be used with tools emerged.

The machine-readable CSP, usually denoted as CSP_M, not only provides a notation for tools such as FDR model-checker to be build upon but also extends the existing theory by using a functional programming language to describe and manipulate things like events and process parameters.

The Table 1 shows for every CSP process constructor discussed in section 2.1 the corresponding ASCII representation according to CSP_M language.

Table 1 – The ASCII representation of CSP.

Constructor	Syntax	ASCII form
Stop	$STOP$	STOP
Skip	$SKIP$	SKIP
Event prefix	$e \rightarrow P$	e -> P
External choice	$P \,\Box\, Q$	P [] Q
Internal choice	$P \sqcap Q$	P |~| Q
Alphabetized parallel	$P \; {}_A\|_B \; Q$	P [A || B] Q
Generalized parallel	$P \underset{A}{\|} Q$	P [| A |] Q
Interleave	$P \; \|\|\| \; Q$	P ||| Q
Sequential composition	$P \, ; Q$	P ; Q
Event hiding	$P \setminus A$	P \ A

2.2 The Coq proof assistant

A proof assistant is a software for helping construct proofs of logical propositions. Essentially, it is a hybrid tool that automates the more routine aspects of building proofs while relying on human intervention for more complex steps. There is a variety of proof assistants including Isabelle, Agda, ATS, Idris and Coq, among others. This work is based around the Coq proof assistant.

Coq can be viewed as a combination of a functional programming language plus a set of tools for stating and proving logical assertions. Moreover, the Coq environment provides high-level facilities for proof development, including a large library of common definitions and lemmas, powerful tactics for constructing complex proofs semi-automatically, and a special-purpose programming language for defining new proof-automation tactics for specific situations.

Coq's native functional programming language is called *Gallina*. Before we discuss about the proof development aspect of this interactive theorem prover, we need to introduce the most essential elements we may find in a Gallina program. Consider the following definition of natural numbers in Coq:

```
Inductive nat : Type :=
  | O
```

```
  | S (n : nat).
```

This declaration tells Coq that we are defining a *type*. The capital-letter O constructor represents zero. When the S constructor is applied to the representation of the natural number n, the result is the representation of $n + 1$, where S stands for "successor". An `Inductive` definition carves out a subset of the whole space of constructor expressions and gives it a name, in this case, *nat*.

Having defined *nat*, we can write functions that operate on natural numbers, such as the predecessor function:

```
Definition pred (n : nat) : nat :=
  match n with
    | O ⇒ O
    | S n' ⇒ n'
  end.
```

Note that we do not need recursion to define the predecessor function, but simple pattern matching is not enough for more interesting computations involving natural numbers. For example, to check that a number n is even, we may need to recursively check whether $n - 2$ is even. In order to do that, we use the keyword `Fixpoint` instead of `Definition`:

```
Fixpoint evenb (n:nat) : bool :=
  match n with
    | O ⇒ true
    | S O ⇒ false
    | S (S n') ⇒ evenb n'
  end.
```

Yet another way for defining evenness is through *inductive declaration*. Consider the following two rules: *the number* 0 *is even*, and *if* n *is even, then* $S\ (S\ n)$ *is even*. Lets call the first rule *ev_0* and then the second *ev_SS*. Using *ev* for the name of evenness property, we can write the following inference rules:

$$\frac{}{ev\ 0} \quad (ev\_0)$$

$$\frac{ev\ n}{ev\ (S\ (S\ n))} \quad (ev\_SS)$$

Now, we can translate these rules into a formal Coq definition. Each constructor in this definition corresponds to an inference rule:

```
Inductive ev : nat → Prop :=
  | ev_0 : ev 0
```

| *ev_SS* $(n : nat)$ $(H : ev\ n)$: *ev* $(S\ (S\ n))$.

This definition is different from previous use of `Inductive`. We are defining a function from *nat* to *Prop*, in other words, a property of numbers. The type of each constructor must be specified explicitly (after a colon), and each constructor's type must have the form *ev* n for some natural number n.

2.2.1 Building proofs

As a proof development system, Coq provides interactive proof methods, decision and semi-decision algorithms, and a tactic language for letting the user define its own proof methods. Proof development in Coq is done through a language of tactics that allows a user-guided proof process.

Recall the functional definition of evenness we introduced in the previous section, *evenb*. Suppose we want to prove that consecutive numbers have opposite parity. In other words, if $S\ n$ is even, then n is not, and if $S\ n$ is not even, then n is. One way to assert this statement is through the following proposition: $\forall\ (n : nat),\ evenb\ (S\ n) = negb\ (evenb\ n)$.

Eventually, during a proof development, one may find useful to make assertions about smaller intermediary steps of a theorem proof. This can be done either inside the main proof tree or in a completely separate one. The "divide and conquer" approach can help decreasing the number of steps in a proof and even reduce its overall complexity. In this example, we will first introduce a lemma to prove the involutive property of the negation function *negb*. This can be achieved in Coq with following commands:

```
Lemma negb_involutive : ∀ (b : bool),
  negb (negb b) = b.
Proof.
  destruct b.
  - simpl. reflexivity.
  - simpl. reflexivity.
Qed.
```

See the proof for Lemma *negb_involutive* in table 2

Table 2 – Proof of Lemma negb_involutive

Next step in Coq	Proof situation
Proof.	$\overline{\hspace{10em}}$ $forall\ b : bool,\ negb\ (negb\ b) = b$

Continuing proof of Lemma negb_involutive on the next page

Table 2 – Proof of Lemma negb_involutive continued

Next step in Coq	Proof situation
$destruct\ b.$	$\overline{\qquad\qquad}$ $negb\ (negb\ true) = true$ $\overline{\qquad\qquad}$ $negb\ (negb\ false) = false$
$-_{1/2}$	$\overline{\qquad\qquad}$ $negb\ (negb\ true) = true$
$simpl.$	$\overline{\qquad\qquad}$ $true = true$
$reflexivity.$	$-_{1/2}$ completed
$-_{2/2}$	$\overline{\qquad\qquad}$ $negb\ (negb\ false) = false$
$simpl.$	$\overline{\qquad\qquad}$ $false = false$
$reflexivity.$	$-_{2/2}$ completed, proof completed by Qed

End of proof of Lemma negb_involutive

The proof editing mode in Coq is entered whenever asserting a statement. Keywords such as `Lemma`, `Theorem`, and `Example` do so by allowing us to give the statement a name and the proposition we want to prove. Additionally, the commands `Proof` and `Qed` delimit, respectively, the beginning and the end of the sequence of tactic commands.

The keywords `destruct`, `simpl`, and `reflexivity` are examples of tactics. A tactic is a command that is used to guide the process of checking some claim we are making. The tactic `destruct` generates two sub-goals, one for each boolean value, which we must prove separately in order to prove the main goal. This strategy is also known as proof by case analysis.

The tactic `simpl` is often used in situations where we want to evaluate a compound expression, eventually reducing it to a simplified, easier-to-understand term. It facilitates our decisions in a proof development by resolving all the computations that can be done in a given state of the goal. Additionally, the tactic `reflexivity` finishes a proof by showing that both sides of an equation contain identical values.

Once we proved this lemma, it is now available to be used inside other proofs such as the one of the theorem we stated in the beginning of this section:

```
Theorem evenb_S : ∀ n : nat,
  evenb (S n) = negb (evenb n).
Proof.
  intros. induction n.
  - simpl. reflexivity.
  - simpl. simpl in IHn. rewrite IHn.
    rewrite negb_involutive. reflexivity.
Qed.
```

See the proof for Theorem *evenb_S* in table 3

Table 3 – Proof of Theorem evenb_S

Next step in Coq	Proof situation
$Proof.$	<hr> $forall\ n : nat,\ evenb\ (S\ n) = negb\ (evenb\ n)$
$intros.$	$n : nat$ <hr> $evenb\ (S\ n) = negb\ (evenb\ n)$
$induction\ n.$	<hr> $evenb\ 1 = negb\ (evenb\ 0)$
	$n : nat$ $IHn : evenb\ (S\ n) = negb\ (evenb\ n)$ <hr> $evenb\ (S\ (S\ n)) = negb\ (evenb\ (S\ n))$
$-_{1/2}$	<hr> $evenb\ 1 = negb\ (evenb\ 0)$
$simpl.$	<hr> $false = false$
$reflexivity.$	$-_{1/2}$ completed
$-_{2/2}$	$n : nat$ $IHn : evenb\ (S\ n) = negb\ (evenb\ n)$ <hr> $evenb\ (S\ (S\ n)) = negb\ (evenb\ (S\ n))$

Continuing proof of Theorem evenb_S on the next page

Table 3 – Proof of Theorem evenb_S continued

Next step in Coq	Proof situation
$simpl.$	$n : nat$ $IHn : evenb\ (S\ n) = negb\ (evenb\ n)$ ——— $evenb\ n = negb\ match\ n\ with \quad \mid 0 => false$ $\mid S\ n' => evenb\ n' \quad end$
$simpl\ in\ IHn.$	$n : nat$ $IHn : match\ n\ with \quad \mid 0 => false \quad \mid S\ n' => evenb\ n'$ $end = negb\ (evenb\ n)$ ——— $evenb\ n = negb\ match\ n\ with \quad \mid 0 => false$ $\mid S\ n' => evenb\ n' \quad end$
$rewrite\ IHn.$	$n : nat$ $IHn : match\ n\ with \quad \mid 0 => false \quad \mid S\ n' => evenb\ n'$ $end = negb\ (evenb\ n)$ ——— $evenb\ n = negb\ (negb\ (evenb\ n))$
$rewrite$ $negb\_involutive.$	$n : nat$ $IHn : match\ n\ with \quad \mid 0 => false \quad \mid S\ n' => evenb\ n'$ $end = negb\ (evenb\ n)$ ——— $evenb\ n = evenb\ n$
$reflexivity.$	$-_{2/2}$ completed, proof completed by Qed

End of proof of Theorem evenb_S

Note that we have added the quantifier $\forall\ n{:}nat$, so that our theorem talks about all natural numbers n. The tactic `intros` is responsible for moving the quantifier into the context of current assumptions.

This example demonstrates a proof by induction over natural numbers that is made possible in Coq by the `induction` tactic. Following this principle, to show that a proposition holds for all natural numbers n we must prove: the base case ($n = 0$), and then the induction step, which is, for any number n', if the proposition holds for n', then so it does for $S\ n'$.

The tactic `rewrite` tells Coq to perform a replacement in the goal, whether it is based on an assumption (hypothesis) from the proof context or a completely separate proof such as the Lemma *negb_involutive*.

Consider another theorem on natural numbers: for all n, if n is even, then the predecessor of the predecessor of n is also even. This theorem can be proved in Coq using the following commands:

```
Theorem ev_minus2 : ∀ n:nat,
  ev n → ev (pred (pred n)).
Proof.
  intros.
  destruct H.
  - simpl. apply ev_0.
  - simpl. apply H.
Qed.
```

See the proof for Theorem *ev_minus2* in table 4

Table 4 – Proof of Theorem ev_minus2

Next step in Coq	Proof situation
Proof.	—————— $forall\ n : nat,\ ev\ n -> ev\ (pred\ (pred\ n))$
intros.	$n : nat$ $H : ev\ n$ —————— $ev\ (pred\ (pred\ n))$
destruct H.	—————— $ev\ (pred\ (pred\ 0))$
	$n : nat$ $H : ev\ n$ —————— $ev\ (pred\ (pred\ (S\ (S\ n))))$
$-_{1/2}$	—————— $ev\ (pred\ (pred\ 0))$
simpl.	—————— $ev\ 0$
apply ev_0.	$-_{1/2}$ completed

Continuing proof of Theorem ev_minus2 on the next page

Table 4 – Proof of Theorem ev_minus2 continued

Next step in Coq	Proof situation
$-_{2/2}$	$n : nat$ $H : ev\ n$ ――――――――― $ev\ (pred\ (pred\ (S\ (S\ n))))$
simpl.	$n : nat$ $H : ev\ n$ ――――――――― $ev\ n$
apply H.	$-_{2/2}$ completed, proof completed by Qed

End of proof of Theorem ev_minus2

This time around, the tactic `intros` moves not only the quantifier into the context, but also the *hypothesis* that consists of the antecedent of the implication (*modus ponens*).

As we discussed before, the tactic `destruct` introduces proof by case analysis. In this example, it is responsible for generating, from the hypothesis, two sub-goals based on the inductive definition of evenness we have provided: one where $n = 0$ and the other where $n = S\ (S\ n)$. Once again, we must prove them separately so Coq accepts the theorem.

We then proceed to use the tactic `apply`, passing the term we find useful for proving each sub-goal as argument. In the first branch, where our goal is to prove $ev\ 0$, we apply the first rule of our inductive definition, *ev_0*, which concludes this sub-proof. In the second branch, we have $ev\ n$ as goal, which is already an assumption of ours (introduced to the proof context by the command `intros`). This also concludes the second sub-proof, thus proving the entire statement (theorem *ev_minus2*).

There are many other tactics and variations of them that can be used when proving a proposition. Apart from the ones we have already discussed in this subsection, other commonly used tactic commands are `unfold`, `inversion`, and `contradiction`. Further explanation on theses tactics will be provided as needed throughout the chapter 3.

2.2.2 The tactics language

Ltac is the tactic language for Coq. It provides the user with a high-level “toolbox” for tactic creation, allowing one to build complex tactics by combining existing ones with constructs such as conditionals, looping, backtracking, and error catching.

Imagine we want to prove that the number 4 does not appear in the list of consecutive natural numbers ranging from 0 to 3. By using the keyword `Example`, we can assert this statement in Coq and develop our proof:

```
Example elem_not_in_list : ¬ (In 4 [0 ; 1 ; 2 ; 3]).
Proof.
  unfold not. simpl. intros.
  destruct H.
  - inversion H.
  - destruct H.
    × inversion H.
    × destruct H.
      + inversion H.
      + destruct H.
        { inversion H. }
        { contradiction. }
Qed.
```

The first tactic unfolds the definition of ¬ (not) in the goal, replacing our initial statement by *In* 4 [0 ; 1 ; 2 ; 3] → *False*. The second tactic reduces the new goal by computing the function *In*, which leaves us with the disjunctions $0 = 4 \lor 1 = 4 \lor 2 = 4 \lor 3 = 4 \lor$ *False* in the antecedent of the implication. Then, the tactic `intros` moves this antecedent to the proof context, introducing a new hypothesis *H* and leaving the literal *False* as goal.

From this point on, the proof develops a pattern: we perform a destruction followed by an inversion of the hypothesis until we can end the proof by contradiction. The recurrence of the tactic `destruct` lets us focus in one equality from the disjunction at a time: first the hypothesis becomes $0 = 4$, then $1 = 4$ and so on. The tactic `inversion` finishes each sub proof created by the previous command by deriving all the necessary conditions that should hold for the assumption to be proved. In this case, since none of these equalities is true, there is no condition that satisfies the proposition, thus proving our goal.

Since this pattern in the sequence of tactics is now exposed, we can define a tactic macro for proving propositions of the format “not in” using the keyword `Ltac`:

```
Ltac solve_not_in := unfold not;
  let H := fresh "H" in (
    intros H; repeat (contradiction + (destruct H; [> inversion H | ]))
  ).
Example elem_not_in_list' : ¬ (In 4 [0 ; 1 ; 2 ; 3]).
Proof. solve_not_in. Qed.
```

The new tactic *solve_not_in* works by first unfolding the function *not*, therefore deriving an implication, then moving the premise to the context of assumption, and finally repeating the following steps until the proof is finished: try to finish the proof by searching for a *contradiction* in the assumptions (such as a false hypothesis), if it fails, *destruct* the hypothesis and apply *inversion* to it in order to prove the first sub-goal yielded by the previous tactic.

2.3 QuickChick

QuickChick is a set of tools and techniques for combining randomized property-based testing with formal specification and proof in the Coq ecosystem. It is the equivalent of Haskell's QuickCheck for Coq proof assistant.

There are four basic elements in property-based random testing: an *executable property* such as for deciding whether a number is even, *generators* for random inputs to the property, *printers* for converting data structures like numbers to strings when reporting counterexamples, and *shrinkers*, which are used to search for minimal counterexamples when errors occur.

Consider the following example extracted from Pierce e Lampropoulos (2018). The function *remove* takes a natural number x and a list of natural numbers l and removes x from the list.

```
Fixpoint remove (x : nat) (l : list nat) : list nat :=
  match l with
    | [] ⇒ []
    | h::t ⇒ if h =? x then t else h :: remove x t
  end.
```

We can write assertions that represent our expectations regarding this function. One possible specification for *remove* might be this property:

```
Conjecture removeP : ∀ x l, ¬ (In x (remove x l)).
```

The keyword `Conjecture` treats our property *removeP* as an axiom. This proposition claims that x never occurs in the result of *remove* x l for any x and l. Such statement turns out to be false, as we would discover if we were to try to prove it. A different — perhaps much more efficient — way to discover the discrepancy between the definition and specification is to test it:

```
QuickChick removeP.
```

The *QuickChick* command takes an "executable" property and attempts to falsify it by running it on many randomly generated inputs, resulting in output like this:

0
[0, 0]
Failed! After 17 tests and 12 shrinks

This means that, if we run *remove* with x being 0 and l being the two-element list containing two zeros, then the property *removeP* fails.

With this example in hand, we can see that the *then* branch of remove fails to make a recursive call, which means that only one occurrence of x will be removed from the list. The last line of the output records that it took 17 tests to identify some fault-inducing input and 12 "shrinks" to reduce it to a minimal counterexample.

3 A theory for CSP in Coq

3.1 Syntax

3.1.1 Abstract syntax

3.1.2 Concrete syntax

3.2 Structured operational semantics

3.3 Labelled transition systems

3.3.1 GraphViz integration

3.4 Traces refinement

3.4.1 QuickChick integration

4 Conclusions

4.1 Related work

4.2 Future work

Bibliography

CLAESSEN, K.; HUGHES, J. Quickcheck: A lightweight tool for random testing of haskell programs. In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. New York, NY, USA: Association for Computing Machinery, 2000. (ICFP '00), p. 268–279. ISBN 1581132026. Disponível em: <https://doi.org/10.1145/351240.351266>. Citado na página 13.

HOARE, C. A. R. Communicating sequential processes. *Communications of the ACM*, ACM New York, NY, USA, v. 21, n. 8, p. 666–677, 1978. Citado na página 13.

PIERCE, B. C.; LAMPROPOULOS, L. *Logical Foundations*. [S.l.]: Electronic textbook, 2018. v. 4. (Software Foundations, v. 4). Version 1.0, <http://softwarefoundations.cis.upenn.edu>. Citado na página 29.

SCHNEIDER, S. *Concurrent and Real Time Systems: The CSP Approach*. 1st. ed. USA: John Wiley & Sons, Inc., 1999. ISBN 0471623733. Citado na página 10.