THÈSE POUR OBTENIR LE GRADE DE DOCTEUR
DE L'UNIVERSITÉ DE MONTPELLIER

En Informatique

École doctorale : Information, Structures, Systèmes

Unité de recherche LIRMM

Vérification formelle d'une méthodologie pour la conception et la production de systèmes numériques critiques

Formal verification of a methodology for the design and production of safety-critical digital systems

Présentée par Vincent IAMPIETRO

Le 16 décembre 2021

Sous la direction de David Delahaye
et David Andreu

Devant le jury composé de

Marc Pouzet, Professeur, ENS, Paris	Président du jury
Sandrine Blazy, Professeure, Université de Rennes	Rapporteure
Frédéric Boniol, Maître de recherche, ONERA, Toulouse	Rapporteur
David Déharbe, Docteur, Ingénieur, Clearsy	Examinateur
David Delahaye, Professeur, Université de Montpellier	Directeur
David Andreu, MCF, Université de Montpellier	Co-directeur

UNIVERSITÉ
DE MONTPELLIER

Contents

List of Abbreviations

SITPN	**S**ynchronously executed **I**nterpreted **T**ime **P**etri **N**et with priorities
VHDL	**V**ery high speed integrated circuit **H**ardware **D**escription **L**anguage
CIS	**C**omponent **I**nstantiation **S**tatement
PCI	**P**lace **C**omponent **I**nstance
TCI	**T**ransition **C**omponent **I**nstance
GPL	**G**eneric **P**rogramming **L**anguage
HDL	**H**ardware **D**escription **L**anguage
LRM	**L**anguage **R**eference **M**anual
DSL	**D**omain **S**pecific **L**anguage
MDE	**M**odel-**D**riven **E**ngineering
FM	**F**ormal **M**ethods

Chapter 1

Preliminary notions

In this chapter, we introduce the mathematical formalisms and notations used throughout this thesis to express and formalize our ideas. Section 1.1 introduces both classical first-order logic and set theory which constitute our mathematical frameworks. Section 1.2 is a reminder on induction principles. In Section 1.3, we provide the basics to understand the Coq proof assistant, which is the system we use to write our programs and mechanize our proofs. This chapter is inspired by the book *The Formal Semantics of Programming Languages: an Introduction* [10] by Glenn Winskel, the courses of the University of Cambridge on the semantics of programming languages[1] by Neel Krishnaswami, the documentation of the Coq proof assistant[2], and the book *Certified Programming with Dependent Types* [2] by Adam Chlipala.

1.1 Mathematical formalisms

In this section, we introduce classical first-order logic and the Zermelo-Fraenkel (ZF) set theory which combination constitutes the base formalism for all our mathematical definitions, and our framework for the expression and the interpretation of logical formulas.

1.1.1 Classical first-order logic

In this section, we define the syntax of the first-order logic. Here, we already use concepts and notations that belong to set theory; set theory will be presented in the following section. For now, the reader has only to consider the *intuitive* definition of a set as a collection of elements, and a function as an entity that relates each element of a set to a unique element of another set.

To define the syntax of the first-order logic, let us first define:

- the set $\mathcal{V}$ of variables x, y, etc.
- the set $\mathcal{S}_{\mathcal{F}}$ of function symbols f, g, etc.
- the set $\mathcal{S}_{\mathcal{P}}$ of predicate symbols P, Q, etc.

[1] `https://www.cl.cam.ac.uk/teaching/2021/Semantics/`
[2] `https://coq.inria.fr/distrib/current/refman/index.html`

Let us also consider a function $a \in \mathcal{S_F} \cup \mathcal{S_P} \rightarrow \mathbb{N}$ that associates a given function or predicate symbol to an arity, i.e. the number of parameters of the function or the predicate. E.g, if $f(x,y)$ with $f \in \mathcal{S_F}$ then $a(f) = 2$; if $P(x,y,z)$ with $P \in \mathcal{S_P}$ then $a(P) = 3$. Constants are functions of arity 0, i.e. with no parameter.

The syntax of classical first-order logic [5] is divided between *terms* and *formulas*. We define a term of the classical first-order logic with the following BNF entry:

$$t ::= v \mid f(t_1, \dots, t_n)$$

This entry states that a term is either a variable $v \in \mathcal{V}$, or a function symbol $f \in \mathcal{S_F}$ of arity n with terms $t_1, \dots, t_n$ as inputs.

We define a formula of the classical first-order logic with the following BNF entry:

$$\mathcal{F} ::= \bot \mid \top \mid P(t_1, \dots, t_n) \mid \mathcal{F} \wedge \mathcal{F} \mid \mathcal{F} \vee \mathcal{F} \mid \mathcal{F} \Rightarrow \mathcal{F} \mid \mathcal{F} \Leftrightarrow \mathcal{F} \mid \neg \mathcal{F} \mid \forall x, \mathcal{F} \mid \exists x, \mathcal{F}$$

This entry states that a formula is either:

- $\bot$ (bottom), the always *false* formula, or $\top$ (top), the always *true* formula
- a predicate $P(t_1, \dots, t_n)$ (i.e. an atomic formula) of arity n, where $P \in \mathcal{S_P}$ with terms $t_1, \dots, t_n$ as inputs.
- the composition of two subformulas with one of the following binary operators: the conjunction $\wedge$, the disjunction $\vee$, the implication $\Rightarrow$, the double implication $\Leftrightarrow$
- the composition of one subformula with the negation operator $\neg$
- a subformula prefixed by the universal quantifier $\forall$ or the existential quantifier $\exists$. For instance, the formula $(\forall x, P(x))$ denotes the atomic formula $P(x)$ where the parameter x is a universally quantified variable of the formula. As a shorthand notation, we write $\forall x, y, z. \dots$ to denote $\forall x, \forall y, \forall z. \dots$. The same stands for the existential quantifier $\exists$. Variables that are introduced in a logical formula by one of the previous quantifiers are called the *bound* variables of the formula. Variables that appear in a logical formula without being introduced by a quantifier are called *free* variables. For instance, in the logical formula $(\forall x,\ P(x) \wedge Q(y))$, x is a bound variable and y is a free variable of the formula.

In this thesis, our formulas are interpreted as formulas of the *classical* logic [8]. Thus, during a proof, we can appeal to the *law of excluded middle* to reason on the truth value of a given formula.

1.1.2 ZF Set theory

In this thesis, we use the Zermelo-Fraenkel (ZF) set theory as the base formalism for all our mathematical definitions and proofs. In this section, we present the axioms of the ZF set theory, and the associated definitions and notations that will be used throughout this memoir. The reader will find further information on the ZF set theory in [6].

In the ZF set theory, a set represents a group of elements called the members of the set. For every set A, we write $a \in A$ to denote that the element a is a member of set A. The *membership* property is a basic set-theoretic property. Given a set A, we sometimes have to express a property P of the elements of A in the following form: $\forall x, x \in A \Rightarrow P(x)$. When there is no ambiguity, we equivalently write $\forall x \in A,\ P(x)$.

Now, let us define the axioms of the ZF set theory.

Axiom 1 (Existence). *There exists a set which has no elements, i.e.* $\exists A, \forall a, a \notin X$.

Definition 1 (Empty set). *We call the empty set the unique set with no element, written* $\emptyset$.

Axiom 2 (Extensionality). *If every element of* A *is an element of* B*, and vice-versa, then* $A = B$*, i.e.* $\forall A, B, x,\ (x \in A \Leftrightarrow x \in B) \Rightarrow A = B$.

Axiom 3 (Schema of comprehension). *Let* $P(x)$ *be a property of* x*. For all set* A*, there exists a set* B *such that* $x \in B$ *if and only if* $x \in A$ *and* $P(x)$ *holds. I.e.* $\forall A, B, x,\ x \in B \Leftrightarrow x \in A \wedge P(x)$.

Axiom 4 (Pair). *For all set* A *and* B*, there exist a set* C *such that* $x \in C$ *if and only if* $x = A$ *and* $x = B$*. I.e.* $\forall A, B, \exists C, \forall x,\ (x \in C \Leftrightarrow x = A \vee x = B)$.

More intuitively, Axiom 4 states that if A and B are sets then their corresponding pair C is also a set.

Axiom 5 (Union). *For all set* A *having sets as elements, there exists a set* C *corresponding to the union of the elements of* A*. I.e.* $\forall A, \exists U,\ x \in U \Leftrightarrow \exists B,\ B \in A \wedge x \in B$.

Definition 2 (Subset). *A set* A *is a subset of set* B*, written* $A \subseteq B$*, if for all* x *if* $x \in A$ *then* $x \in B$*, i.e.* $\forall x, x \in A \Rightarrow x \in B$.

Definition 3 (Union). *Given a set* A *and* B*, the set* $A \cup B$ *is the union of the members of* A *and the members of* B*, i.e.* $\forall x, x \in A \cup B \Leftrightarrow x \in A \vee x \in B$.

Axiom 6 (Infinity). *There exists a set* A *such that the empty set belongs to* A *and for all* x *such that* $x \in A$ *then* $x \cup \{x\} \in A$*. I.e.* $\exists A, \emptyset \in A \wedge (\forall x,\ x \in A \Rightarrow x \cup \{x\} \in A)$.

Axiom 7 (Power set). *For all set* A*, there exists a set* B *such that for all set* X*,* $X \in B$ *if and only if* $X \subseteq A$*. I.e.* $\forall A, \exists B, \forall X, (X \in B \Leftrightarrow X \subseteq A)$.

Axiom 8 (Schema of replacement). *Let $P(x,y)$ be a property such that for all x there exists a unique y such that $P(x,y)$ holds. For all set A, there exists a set B such that for all $x \in A$, there exists $y \in B$ for which $P(x,y)$ holds. I.e. $(\forall x, \exists y, P(x,y)) \wedge \forall A, \exists B, \forall x, (x \in A \Leftrightarrow \exists y, y \in B \wedge P(x,y))$.*

Definition 4 (Intersection). *Given a set A and B, th set $A \cap B$ denotes the set formed by the intersection of set A and B, i.e. $\forall x, x \in A \cap B \Leftrightarrow \; x \in A \wedge x \in B$.*

Axiom 9 (Foundation). *For all non-empty set A, there exists a set B such that $B \in A$ and B has no common element with A. I.e. $\forall A, x \neq \emptyset \Rightarrow \exists B, B \in A \wedge (A \cap B = \emptyset)$.*

Based on the previous axioms, we can complement the theory with following definitions and notation.

Notation 1 (Extension). *A set is defined by* extension *with the enumeration of all its members. For instance, $\{1, 0, -1\}$, $\{a, b, c\}$ or $\{p_0, \ldots, p_n\}$ are all sets defined by extension.*

Notation 2 (Intension). *Let $P(x)$ be a property of x, we write $\{x \mid P(x)\}$ the set of x for which $P(x)$ holds. The set $\{x \mid P(x)\}$ is defined by intension. For instance, here is the intensional definition of the set of even numbers: $\{n \in \mathbb{N} \mid \exists k \in \mathbb{N},\ n = 2k\}$.*

Given two sets A and B, the following sets are formed:

Definition 5 (Difference). *$A \setminus B$ denotes the set formed by the elements of set A that are not elements of set B (the difference between set A and B), i.e. $A \setminus B = \{x \mid x \in A \wedge x \notin B\}$.*

Definition 6 (Cartesian product). *$A \times B$ denotes the cartesian product between the elements of set A and set B, i.e. the set of all ordered pairs defined by $\{(x,y) \mid x \in A \wedge y \in B\}$. We generalize the definition to build the set of n-tuples $A_0 \times A_1 \times \cdots \times A_n$ defined by $\{(x_0, (x_1, \ldots (\ldots, x_n))) \mid x_0 \in A_0, x_1 \in A_1, \ldots, x_n \in A_n\}$.*

It is sometimes useful to give a name to the elements of a tuple without referring to their index. In such a case, a tuple is called a record where each element, called a field, has been given an explicit name. This formalism is useful to represent rather complex data structures. For instance, say that we want to represent the set of humans by a triplet composed of the size, weight, and eye color of a given human. We can define this set as the set of triplet $\mathbb{R} \times \mathbb{R} \times \{green, blue, brown\}$. If we want to give a concrete name to the elements of the triplet, we can equivalently define such a triplet as a record, written $<\!size, weight, eye\!>$, where $size \in \mathbb{R}$, $weight \in \mathbb{R}$ and $eye \in \{green, blue, brown\}$.

Definition 7 (Disjoint union). *$A \sqcup B$ denotes the set formed by the disjoint union of set A and set B. The disjoint union is obtained by adjoining an index i to the elements of A and an index j to the elements of B such that $i \neq j$. Then, the two sets of couples are joined together to build the disjoint union of A and B. For instance, consider that $A = \{a, b, c\}$ and $B = \{a, b\}$. To obtain the disjoint union $A \sqcup B$, we create the two sets $A_i = \{(i, a), (i, b), (i, c)\}$ and $B_i = \{(j, a), (j, b)\}$, and then join the sets together s.t. $A \sqcup B = \{(i, a), (i, b), (i, c), (j, a), (j, b)\}$.*

When the two sets A and B are disjoint, i.e. $A \cap B = \emptyset$, then $A \sqcup B$ is isomorphic to $A \cup B$. To stress the fact that we are building a set from the union of two disjoint sets, we prefer to use the disjoint union operator. For instance, we write $\mathbb{N} \sqcup \{\infty\}$, instead of $\mathbb{N} \cup \{\infty\}$, to denote the set of values ranging from the set of natural numbers with the addition of the infinite value ∞.

Definition 8 (Powerset). *$\mathcal{P}(A)$ denotes the powerset of A defined by all the possible subsets formed with the elements of set A, i.e. $\mathcal{P}(A) = \{X \mid X \subseteq Y\}$.*

Relations and functions

Definition 9 (Relation). *A binary relation R between two sets X and Y is a subset of the set of pairs $X \times Y$, i.e. $R \subseteq X \times Y$, or an element of the powerset $\mathcal{P}(X \times Y)$, i.e. $R \in \mathcal{P}(X \times Y)$. We write $R(x, y)$ to denote $(x, y) \in R$. We generalize the definition to n-ary relations. An n-ary relation between sets $X_0, \ldots, X_n$ is a subset of the set of n-tuples $X_0 \times \cdots \times X_n$, i.e. $R \subseteq X_0 \times \cdots \times X_n$, or an element of the powerset $\mathcal{P}(X_0 \times \cdots \times X_n)$, i.e. $R \in \mathcal{P}(X_0 \times \cdots \times X_n)$. We write $R(x_0, \ldots, x_n)$ to denote $(x_0, \ldots, x_n) \in R$.*

Definition 10 (Partial function). *A partial function f from set X to set Y is a binary relation from X to Y verifying that $\forall x \in X, y, y' \in Y$, $(x, y) \in f \land (x, y') \in f \Rightarrow y = y'$, i.e. x appears at most once as the first element of a pair in f. We note $f \in X \nrightarrow Y$ to denote a partial function from X to Y. The set of the first elements of the pairs defined in f is called the* domain *of f. We write it $\texttt{dom}(f) = \{x \mid \exists y \text{ s.t.} (x, y) \in f\}$. When there is no ambiguity, and given an $x \in X$ and $f \in X \nrightarrow Y$, we write $x \in f$ as a shorthand to $x \in \texttt{dom}(f)$.*

Definition 11 (Application). *A total function a, or application, from X to Y is a partial function verifying that all the elements of X appear as the first element of a pair in a, i.e. for all $x \in X$, there exists $y \in Y$ such that $(x, y) \in a$. In other words, the domain of an application from X to Y is equal to the set X. We note $a \in X \rightarrow Y$ to denote an application from X to Y.*

1.1.3 Rule-based definition of sets

All along this memoir, we define sets (especially relations) with rule instances, also called inference rules or judgments. A rule instance can take the following forms:

$$\frac{}{C} \text{ or } \frac{P_1, \ldots, P_n}{C}$$

The left form of rule instance is called an axiom. In the right form of rule instance, $P_1, \ldots, P_n$ are the premises of the rule and C is the conclusion of the rule.

Definition 12 (Rule instances)**.** *We define a set R of rule instances as a set of pairs of the form (P/C) where P is a finite (possibly empty) set of premises and C is an element called the conclusion. A pair (P/C) is a rule instance.*

Rule instances define a way to build *derivation trees*. A derivation of C takes either the form of an axiom, i.e. $\frac{}{C}$, or of a tree with C as a root and with branches composed of the derivation trees of the premises, i.e.

$$\frac{\dfrac{\vdots}{P_1} \quad \ldots \quad \dfrac{\vdots}{P_n}}{C}$$

Given a set R of rule instances, we define the set A such that if $(\emptyset/C) \in R$ then $C \in A$, and if $(\{P_1, \ldots, P_n\}/C) \in R$, then if there exists a derivation for all premises $P_1, \ldots, P_n$ then $C \in A$. In fact, the set R of rule instances define the properties that must be verified by the elements of set A. There exists an infinity of sets A that verify the properties outlined by the rule instances R. Thus, we define the set of elements defined by the rule instances R as the least set verifying the properties outlined by the rules. For instance, the two following rules, named rules Ev0 and Ev2, define the set of even natural numbers:

$$\overset{\text{Ev0}}{\frac{}{IsEven(0)}} \qquad \overset{\text{Ev2}}{\frac{IsEven(n-2)}{IsEven(n)}}$$

The rule Ev0 states as an axiom that 0 is an even number; the rule Ev2 states that for all natural number n, n is an even number if one can derive that fact that $n-2$ is an even number. Thus, we can derive from the previous rules that 4 is an even number by building the following derivation tree:

$$\dfrac{\dfrac{\dfrac{}{IsEven(0)}\,\text{Ev0}}{IsEven(2)}\,\text{Ev2}}{IsEven(4)}\,\text{Ev2}$$

Starting from $IsEven(4)$, we can apply the Ev2 rule to derive $IsEven(2)$. Then, another application of the Ev2 rule leads to $IsEven(0)$, and we can close the derivation branch by applying the Ev0 rule. Here, the only branch of the tree has reached an axiom, and thus the derivation tree is finite. To further illustrate the use of rule instances in the definition of a set, let us consider the following minimal language of arithmetic expressions expressed in the Backus-Naur form:

$$e \quad ::= \quad n \mid id \mid e_0 + e_1$$

Here, n ranges over the set of natural numbers $\mathbb{N}$; id ranges over the set `string` of non-empty strings (i.e. it is the set of identifiers). To evaluate the arithmetic expressions, we need a state $s \in \mathtt{string} ⇸ \mathbb{N}$ that maps each variable identifier to a natural number value. We assume that only a certain set of declared identifiers can appear in an arithmetic expression. Thus, the state is a partial function from the set of non-empty strings to the set of natural numbers. We define the evaluation relation for the arithmetic expressions with the three following rules:

$$\textsc{Nat}\ \frac{}{s \vdash n \to n} \qquad \textsc{Var}\ \frac{}{s \vdash id \to s(id)}\ id \in \mathtt{dom}(s) \qquad \textsc{Add}\ \frac{s \vdash e_0 \to n \qquad s \vdash e_1 \to m}{s \vdash e_0 + e_1 \to n + m}$$

Here, the evaluation relation is a subset of the set of triplets $(\mathtt{string} ⇸ \mathbb{N}) \times e \times \mathbb{N}$. In the rule instances defining the evaluation relation, the $\vdash$ symbol (pronounced *thesis*) means that the left part implies the right part, or it is involved in the evaluation of the right part. For instance, the second rule can be read: in the context of state s, id evaluates to $s(id)$ if $id \in \mathtt{dom}(s)$. When the *context* is not involved in the evaluation of the syntactic constructs on the right side of the $\vdash$ symbol, we remove the context and the $\vdash$ from the rule instances. For example, we can define to the $\textsc{Nat}$ rule by:

$$\textsc{Nat}\ \frac{}{n \to n}$$

as the state s in not involved in the evaluation of expressions that are natural numbers.

Note that in the $\textsc{Var}$ rule, there appears an extra statement, at the right of the judgment line, called a *side condition*. This is an extra condition that must hold with all the premises of the rule instance, but which does not generate a derivation tree of its own.

Finally, here is an example of a derivation tree for the evaluation of the expression $x + (y + 1)$ in the context of state $\{(x,1),(y,2)\}$:

$$\dfrac{\textsc{Var}\ \dfrac{}{\{(x,1),(y,2)\} \vdash x \to 1}\ x \in \{x,y\} \qquad \dfrac{\textsc{Var}\ \dfrac{}{\{(x,1),(y,2)\} \vdash y \to 2}\ y \in \{x,y\} \qquad \dfrac{}{1 \to 1}\ \textsc{Nat}}{\{(x,1),(y,2)\} \vdash y + 1 \to 3}\ \textsc{Add}}{\{(x,1),(y,2)\} \vdash x + (y + 1) \to 4}\ \textsc{Add}$$

Here again, all the branches of the derivation tree have reached axioms, and thus $\{(x,1),(y,2)\} \vdash x + (y + 1) \to 4$ is a member of the evaluation relation of arithmetic expressions. It states that the expression $x + (y + 1)$ evaluates to 4 in the state $\{(x,1),(y,2)\}$.

The evaluation relation can also include rule instances defining error cases. For instance, we can add an extra rule to the definition of the evaluation relation for arithmetic expressions;

the following rule states that an arithmetic expression that is an unreferenced variable in state σ results in an error:

$$\textsc{UnrefVar}\ \frac{}{s \vdash id \rightarrow \texttt{err}}\ id \notin \texttt{dom}(s)$$

The special value `err` is defined to represent error cases. Thus, the evaluation relation defines a subset of triplets $(\texttt{string} \nrightarrow \mathbb{N}) \times e \times (\mathbb{N} \sqcup \{\texttt{err}\})$.

1.2 Induction principles

In the proofs presented in this thesis, we often rely on *induction*. Here are some reminders on induction principles to help the reader understand the proofs of Chapter **??** and Appendix **??**.

1.2.1 Well-founded induction

The most general principle of induction is called *well-founded* induction. From well-founded induction derives all induction principles presented afterwards.

To introduce well-founded induction, let us define a well-founded relation.

Definition 13 (Well-founded relation)**.** *A binary relation $\prec$ over a set A is well-founded if there exist no infinite descending chain, i.e. $\cdots \prec a_i \prec \cdots \prec a_1 \prec a_0$.*

For instance, the *strictly less than* relation $<$ over the set of natural numbers is a well-founded relation.

Let $\prec$ be a well-founded binary relation on a set A. The principle of well-founded induction on the relation $\prec$ says that in order to prove that a property P holds for all elements of A, it suffices to prove that P holds of any $a \in A$ whenever P holds for all $b \in A$ such that $b \prec a$, formally:

$$\big(\forall a \in A, ([\forall b \in A, b \prec a \Rightarrow P(b)] \Rightarrow P(a))\big) \Rightarrow \forall a \in A, P(a)$$

1.2.2 Structural induction

Sometimes, reasoning by induction requires to follow the structure of a given set, i.e. the formation rules of a given set. This kind of reasoning is called structural induction.

Let us consider the formation rules of the set of natural numbers:

$$\textsc{Zero}\ \frac{}{0 \in \mathbb{N}} \qquad \textsc{Succ}\ \frac{n \in \mathbb{N}}{n+1 \in \mathbb{N}}$$

These rules state that zero is a natural number and that for every natural number, its direct successor is also a natural number. Structural induction describes a way to deduce that a property holds for the set of natural numbers, first by stating that the property holds for zero, i.e. the minimal element of the set, then by stating that if the property holds for a given number then it holds for its successor. Thus, knowing that $P(0)$ holds, we can deduce that $P(1)$ holds, $P(2)$ holds, $P(3)$ holds, etc. Following the structural induction scheme, given a property P, to prove that P holds for all natural numbers, it is sufficient to prove that:

- P holds for 0
- if P holds for a given n then it holds at $n + 1$

To take another example, if we want to prove that a given property P holds for the set of arithmetic expressions described in Section 1.1.3, we must prove that:

- P holds for all natural number n
- P holds for all identifiers *id*
- if P holds for all sub-expressions e_0 and e_1, then P holds for $e_0 + e_1$

A proof that leverages structural induction follows the structure of the elements we are reasoning upon. In this thesis, we are using structural induction to prove that a sum expression verifies a certain property. Thus, the structural induction follows the recursive definition of the sum term, which is, for any set A, function $f \in A \to \mathbb{N}$ and $X \subseteq A$ and :

$$\sum_{x \in X} f(x) = \begin{cases} 0 \texttt{ if } X = \emptyset \\ f(x) + \sum\limits_{x' \in X'} f(x') \texttt{ if } X = \{x\} \cup X' \end{cases}$$

In the second computation branch, it is left implicit that set X' is strict subset of X such that $x \notin X'$ or $X' = X \setminus \{x\}$. Given a set A and a function $f \in A \to \mathbb{N}$, to prove that for all $X \subseteq A$, the property $P(X, \sum\limits_{x \in X} f(x))$ holds, we must show that:

- $\forall X \subseteq A,\ X = \emptyset \Rightarrow P(\emptyset, 0)$
- $\forall X \subseteq A, x \in X, X' \subset X,$
  $X = \{x\} \cup X' \Rightarrow P(X', \sum\limits_{x' \in X'} f(x')) \Rightarrow P(\{x\} \cup X', f(x) + \sum\limits_{x' \in X'} f(x'))$

The induction follows the structure of the function. In this specific case, structural induction is often refered to as *functional* induction. Let us prove Proposition 1 to illustrate the use of structural induction over a sum term:

Proposition 1. *For all $X \subset \mathbb{N}$ a finite set of natural numbers, $\sum\limits_{x \in X} 2x$ is even, i.e.*

$$\exists k \in \mathbb{N} \text{ s.t. } \sum_{x \in X} 2x = 2k$$

Proof.

Let us define the property P as follows:

$$P(X, \sum_{x \in X} 2x) \equiv \exists k \in \mathbb{N} \text{ s.t. } \sum_{x \in X} 2x = 2k$$

Then, let us use structural induction to prove $P(X, \sum_{x \in X} 2x)$.

First, let us show $P(\emptyset, 0)$, i.e. $\exists k \in \mathbb{N}$ s.t. $0 = 2k$. Let us take $k = 0$ to build a tautology.

Then, given a $X' \subset X$ and a $x \in X$ s.t. $X = \{x\} \cup X'$, and assuming that $P(X', \sum_{x' \in X'} 2x')$ holds (i.e. the induction *hypothesis*), let us show $P(\{x\} \cup X', 2x + \sum_{x' \in X'} 2x')$. Appealing to the induction hypothesis, let us take a j such that $\sum_{x' \in X'} 2x' = 2j$. Rewriting $\sum_{x' \in X'} 2x'$ as $2j$:

$\Rightarrow \exists k \in \mathbb{N}$ s.t. $2x + 2j = 2k$

$\Rightarrow \exists k \in \mathbb{N}$ s.t. $2(x + j) = 2k$

$\Rightarrow$Then, let us take $k = x + j$ to obtain a tautology.

□

1.2.3 Rule induction

A specific kind of structural induction, called *rule* induction, is applied to prove properties over sets that are defined by rule instances. Let us take the evaluation relation for arithmetic expressions used in Section 1.1.3 to illustrate the principle of rule induction. To prove that a property P holds for the evaluation relation of arithmetic expressions, which is a subset of triplets $(\mathtt{string} \rightarrow \mathbb{N}) \times e \times \mathbb{N}$, we must prove that:

- For all $s \in \mathtt{string} \nrightarrow \mathbb{N}, n \in \mathbb{N}, P(s, n, n)$
- For all $s \in \mathtt{string} \nrightarrow \mathbb{N}, id \in \mathtt{string}$, if $id \in \mathtt{dom}(s)$ then $P(s, id, s(id))$
- For all $s \in \mathtt{string} \nrightarrow \mathbb{N}, e_0, e_1 \in e, n, m \in \mathbb{N}$,
  if $s \vdash e_0 \rightarrow n$ and $P(s, e_0, n)$, and $s \vdash e_1 \rightarrow m$ and $P(s, e_1, m)$
  then $P(s, e_0 + e_1, n + m)$

Rule induction states that in order to prove a property over a set defined by rule instances, the property must hold in any construction case of the considered set. The idea is that if the property is preserved from the premises of rules to the conclusions then the property holds for all the elements of the set.

Let us give an application of rule induction to prove a property over the evaluation relation of arithmetic expressions. First, we define, through the three following rules, the relation $\in_r$ stating that a given identifier id is referenced in an arithmetic expression e, written $id \in_r e$:

$$\textsc{InRId}\ \frac{}{id \in_r id} \qquad \textsc{InRAddL}\ \frac{id \in_r e_0}{id \in_r e_0 + e_1} \qquad \textsc{InRAddR}\ \frac{id \in_r e_1}{id \in_r e_0 + e_1}$$

Then, the property of Proposition 2 states that an arithmetic expression that contains references to identifiers that are not part of the current state's domain can not be evaluated.

Proposition 2. *Let $id \in \texttt{string}$. For all state s, arithmetic expression e, and natural number n,*

$$id \notin \texttt{dom}(s) \land id \in_r e \Rightarrow \neg s \vdash e \rightarrow n$$

Proof.

Let us define the property P as follows:

$$P(s, e, n) \equiv id \notin \texttt{dom}(s) \land id \in_r e \Rightarrow \neg s \vdash e \rightarrow n$$

Then, let us use rule induction to prove $P(s, e, n)$.

First, we must prove $P(s, n, n)$. Assuming $id \in_r n$, there is a contradiction as no rule instance defining the relation $\in_r$ includes the case where the considered expression is a natural number.

Then, we must prove $P(s, id', s(id'))$, assuming that $id' \in \texttt{dom}(s)$. We know that $id \in_r id'$, and thus $id = id'$. Then, there is a contradiction between $id \in \texttt{dom}(s)$ and $id \notin \texttt{dom}(s)$.

Finally, we must prove $P(s, e_0 + e_1, n + m)$, assuming that $s \vdash e_0 \rightarrow n$ and $P(s, e_0, n)$, and $s \vdash e_1 \rightarrow m$ and $P(s, e_1, m)$. We know that $id \in_r e_0 + e_1$; this hypothesis has either be constructed by applying Rule InRAddL or Rule InRAddR. If Rule InRAddL has been applied, then we know $id \in_r e_0$; thus, from $P(s, e_0, n)$, we can deduce $\neg s \vdash e_0 \rightarrow n$, which contradicts $s \vdash e_0 \rightarrow n$. We can perform the proof similarly if Rule InRAddR has been applied. □

1.3 The Coq proof assistant

In this section, we present the Coq proof assistant [9]. The Coq proof assistant constitutes our framework to encode the different semantics, programs and proofs involved in the verification of the HILECOP model-to-text transformation. Here, we give an overview of the different concepts underlying the Coq proof assistant. The aim is to give to the reader the tools to understand the different listings presenting Coq code in the following chapters. For a thorough presentation of the Coq proof assistant, the reader can refer to [2, 7, 1].

1.3.1 The Calculus of Inductive Constructions (CIC)

The kernel of the Coq proof assistant implements the Calculus of Inductive Constructions (CIC) [3]. The CIC is a typed lambda-calculus that includes polymorphism, dependent and inductive types. Thus, the CIC permits us to define programs and types similarly; both are *terms* of the language. A program is a term with a certain type, and a type is also a term with a certain type. The type of a type is called a *sort*. We can mention three basic sorts built in the Coq proof assistant: the Prop sort which is the type of logical formulas, the Set sort which is the type of *small* sets, and the Type sort which encompasses the Prop and Set sorts.

The Coq proof assistant allows us to express logic formulas and to interactively build proofs of these formulas by using a high-level tactic language. The sequence of tactics that builds a proof for a given formula is called a *proof script*. The execution of a proof script builds a proof term. In the CIC, a logic formula can be seen as a *type* and a proof of this formula is an *inhabitant* of the type denoted by the logic formula. Thus, when building a proof term by executing a proof script, the Coq kernel checks that the proof term is of the type of the logic formula by applying typing rules[3]. For instance, let us take two logical propositions A and B. In Coq, we can declare these propositions as elements of the Prop type in the Coq top-level loop[4]:

```
Coq < Variables A B : Prop.
```

The Variables keyword adds the propositions A and B to the global environment accessed by the Coq kernel. Now, say that we want to prove the *modus ponens* theorem expressed with the propositions A and B, namely that $A \Rightarrow (A \Rightarrow B) \Rightarrow B$. In Coq, we can express it as follows:

```
Coq < Theorem modus_ponens : A → (A → B) → A.
```

Here, we declare the modus ponens theorem as an element of type $A \rightarrow (A \rightarrow B) \rightarrow A$. The arrows represent functional arrows; in fact, $A \rightarrow B$ is a notation for the product type $\Pi x : A.B$ where x is not referenced in B. According to the Curry-Howard correspondence [4], there is an equivalence between a proof term and a program. This correspondence is a consequence of the Brouwer-Heyting-Kolmogorov (BHK) interpretation of the *intuitionistic* logic [5]. Intuitionistic logic is the underlying logic built-in the Coq proof assistant. Intuitionistic logic denies the use of the law of excluded middle to perform proofs. Thus, intuitionistic logic as a *constructivist* approach of proofs. In the intuitionistic setting, one has to provide a explicitly built proof to demonstrate a theorem; one can not rely on pure proof by contradiction by appealing to the law of excluded middle. Thus, a proof term of the logical implication $A \Rightarrow B$ is equivalent to an explicitly built program, or a function, of type $A \rightarrow B$, i.e. a program that takes an element of type A and yields an element of type B. Thus, the type $A \rightarrow (A \rightarrow B) \rightarrow A$ is a valid encoding of the formula $A \Rightarrow (A \Rightarrow B) \Rightarrow B$.

The Theorem keyword triggers the interactive proof mode through which the user will build a proof term for the corresponding formula. A simple proof term for the modus ponens theorem is a function that takes an element x of type A and a function f of type $A \rightarrow B$ as inputs, and yields an element of type B by applying the function f to parameter x, i.e. $(f\ x)$. The function takes the form of the following term of the typed lambda-calculus:

[3] https://coq.inria.fr/distrib/current/refman/language/cic.html

[4] Coq scripts can be either interpreted or compiled.

$$\lambda(x : \mathtt{A}).\lambda(f : \mathtt{A} \to \mathtt{B}).(f\ x)$$

While passing this *lambda-term* as a proof term of the modus ponens theorem, the Coq kernel checks the well-typedness of the term by building the following derivation tree, which is a simplified version of the full derivation tree according to the typing rules of the CIC:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathtt{A\ B : Prop}[x : \mathtt{A}, f : \mathtt{A} \to \mathtt{B}] \vdash f \ :\ \mathtt{A} \to \mathtt{B}}\ \textsc{Var} \qquad \dfrac{}{\mathtt{A\ B : Prop}[x : \mathtt{A}, f : \mathtt{A} \to \mathtt{B}] \vdash x \ :\ \mathtt{A}}\ \textsc{Var}}{\mathtt{A\ B : Prop}[x : \mathtt{A}, f : \mathtt{A} \to \mathtt{B}] \vdash (f\ x)\ :\ \mathtt{B}}\ \textsc{App}}{\mathtt{A\ B : Prop}[\mathtt{x : A}] \vdash \lambda(f : \mathtt{A} \to \mathtt{B}).(f\ x)\ :\ (\mathtt{A} \to \mathtt{B}) \to \mathtt{B}}\ \textsc{Lam}}{\mathtt{A\ B : Prop}[] \vdash \lambda(x : \mathtt{A}).\lambda(f : \mathtt{A} \to \mathtt{B}).(f\ x)\ :\ \mathtt{A} \to (\mathtt{A} \to \mathtt{B}) \to \mathtt{B}}\ \textsc{Lam}$$

In the above derivation tree, the global and the local environment are represented at the left of the thesis symbol $\vdash$. The local environment is represented by square brackets. The global environment is represented at the left of the local environment. At the root of the derivation tree, the global environment contains our two previously declared logical propositions `A` and `B`, whereas the the local environment is empty. The application of the LAM rule adds new entry to the local environment; the APP triggers the type-checking of the left and the right part of an application; the VAR rule checks that a term is well-typed if it is referenced as an element of the given type in the global or the local environment.

As said before, the `Theorem` keyword triggers the interactive proof mode. The interactive proof mode will accompany the user to an incremental building of a proof term for the current goal, i.e. the current logic formula we want to prove. Then, to prove the modus ponens theorem, the following interface is first presented to the user:

```
Coq < Theorem modus_ponens : A → (A → B) → B.
1 subgoal

============================
A → (A → B) → B
```

The term under the horizontal bar represents the current goal to prove, i.e. the current formula for which we are building a proof term. Above the horizontal bar are referenced the variables constituting the local environment. At the beginning of the proof, the local environment is empty – and so is the local environemnt at the root of the derivation tree presented above. To build a proof term in interactive mode, the user will then invoke commands called *tactics*. Each tactic invocation corresponds to the invocation of a typing rule of the CIC performed by the Coq kernel. To build a proof term for the modus ponens theorem, the first thing to do is to invoke the LAM rule; this is done by appealing to the `intros` tactic.

```
Coq < intros x.
1 subgoal

x : A
============================
(A → B) → B
```

Here, the user passes to the system the name of the variable that will be introduce in the local environment by the LAM rule, i.e the variable x. Then, we repeat the operation, applying the LAM rule a second time to introduce an element of type A → B in the environment.

```
Coq < intros f.
1 subgoal

x : A
f : A → B
============================
B
```

Then, based on the local environment, we can build an object of type B by applying f to the input x. We can do it by appealing to the apply tactic. The apply tactic invokes the APP rule.

```
Coq < apply (f x).
No more subgoals.

Coq < Qed.
```

After the invocation of the apply tactic, the proof term for the modus ponens theorem is completely built, thus, the Coq top-level loop displays the message that all goals are completed. Then, we can close the interactive proof mode and store the proof of the modus ponens theorem in the global environment by using the Qed keyword.

Another way to prove the modus ponens theorem is to directly pass the proof term with the exact tactic.

```
Coq < exact (fun (x : A) ⇒ fun (f : A → B) ⇒ f x).
No more subgoals.

Coq < Qed.
```

Here the term fun (x : A) ⇒ fun (f : A → B) ⇒ f x represents the lambda-term $\lambda(x : \mathtt{A}).\lambda(f : \mathtt{A} \rightarrow \mathtt{B}).(f\ x)$ (i.e. the proof term) in the Coq syntax.

1.3.2 Inductive types

One of the major strength of the CIC, and therefore of the Coq proof assistant, is the possibility to enrich the global type system with the definition of inductive types. For instance, here is the definition of the type of natural numbers, named nat:

```
Inductive nat : Set :=
| O : nat
| S : nat → nat.
```

The nat type is of the Set sort (remember that the type of a type is called a sort). The nat type is defined through two constructors, represented by the pipe-separated entries. The O constructor states that zero is a natural number. The S constructor takes a natural number as input and yields the succesor to this natural number. This corresponds to the structural definition of natural numbers in Peano's arithmetic. Thus, in this setting, the number 2 is represented by (S (S O)), the number 3 by (S (S (S O)), etc. The result of the evaluation of an inductive type, declared through the Inductive keyword, is the addition of this type and each of its constructors to the global environment accesible by the Coq kernel. Also, a corresponding structural induction principle is generated at the evaluation of an inductive type definition. For instance, the nat_ind induction principle is generated at the evaluation of the nat type. It is a proof term of the logical formula denoting the structural induction principle over the nat type, i.e.:

```
forall P : nat → Prop, P 0 → (forall n : nat, P n → P (S n)) → forall n : nat, P n
```

Then, the induction principle nat_ind can be used to perform structural induction in a proof involving natural numbers. For instance, say that we want to prove the following theorem stating that a natural number elevated at the power 2 is always greater than or equal to itself. We can write as follows:

```
Coq < Theorem ge_pow2 : forall n : nat, n <= n * n.
1 subgoal

============================
forall n : nat, n <= n * n
```

Then, we can use the nat_ind induction principle to prove such a theorem. Most conveniently, the built-in induction tactic chooses the appropriate induction principle based on the type of its argument. Thus, the following command invokes the nat_ind induction principle over the universally quantified variable n:

```
Coq < induction n.
2 subgoals

============================
0 <= 0 * 0
subgoal 2 is:
S n <= S n * S n
```

The result of the invocation of the induction tactic is a branching in the proof tree. Thus, the system indicates that two subgoals must be prove to complete the proof of the gt_pow2 theorem. These two subgoals corresponds to the proof of $P(0)$ and the proof that assuming $P(n)$ we can show $P(n+1)$, as agreed with structural induction. Here, the property P is defined by $P(n) \equiv n \leq n \times n$. We can use the built-in lia tactic, defined in the Lia module of the Coq standard library, to solve the two remaining subgoals. The lia tactic implements a whole decision procedure to prove theorems involving systems of equalities and inequalities over the set of natural numbers. We can combine the `induction` tactic with the `lia` tactic using the semicolon operator. Then, the `lia` tactic is applied to all the subgoals generated by the `induction` tactic.

```
Coq < induction n; lia.
No more subgoals.
Coq < Qed.
```

The Coq proof assistant permits us to define proof tactics, or procedures, in order to automatize some proof tasks. At the top-level of the Coq proof assistant, the `Ltac` and `Ltac2` languages are the supports for the definition of this kind of tactics. These languages allow us to compose sequences of tactics, to perform pattern matching over the local environment and the current goal in interactive proof mode, to define loops or recursive tactics, etc. Even though the `Ltac` and `Ltac2` languages offer a lot of possibilities, the user willing to implement complex proof tactics must turn to the OCaml language which the implementation, and thus meta-language, of the Coq proof assistant. For instance, the `lia` tactic is implemented partly with the `Ltac` language and as a OCaml program.

Leveraging the definition of inductive types, the syntactic constructs of programming languages are also easily implemented. Here is the implementation of the syntax of arithmetic expressions presented in the previous section:

```
Inductive e : Set :=
| enat : nat → e
| eid : string → e
| eadd e → e → e.
```

Each constructor corresponds to a construction case in the definition of arithmetic expressions in the Backus-Naur form. The Coq system also generates the induction principle following the structure of arithmetic expressions (thus, a structural induction principle). The induction principle is a proof term of the following logical formula:

```
  forall P : e → Prop,
      (forall n : nat, P (enat n)) →
      (forall id : string, P (eid id)) →
      (forall e0 : e, P e0 → forall e1 : e, P e1 → P (eadd e0 e1)) →
      forall e : e, P e
```

The evaluation relation for arithmetic expressions is defined in a similar way:

```
Inductive evale (s : string → option nat) : e → nat → Prop :=
| evalnat : forall n : nat, evale s (enat n) n
| evalid : forall (id : string) (n : nat),
             s id = Some n →
             evale s (eid id) n
| evaladd : forall (e0 e1 : e) (n m : nat),
              evale s e0 n →
              evale s e1 m →
              evale s (eadd e0 e1) (n + m).
```

In the above listing, the state that yields the value of identifiers present in an arithmetic expression is a named parameter of the `evale` relation, i.e. the `s` parameter. Parameters which are not varying from one construction case to another can be passed as named parameters while defining an inductive type. The state `s` takes a string identifier as input and yields an `option` to a natural number. As so, the `option` type permits to represent partial functions. The identifiers that belong to the domain of state `s` will be associated with `Some` natural number, whereas the unreferenced identifiers will be associated with the `None` value of the `option` type. The `Some` and the `None` constructors are the two constructors of the `option` type which is defined in Coq as follows:

```
Inductive option (A : Type) : Type :=
| Some : A → option A
| None : option A.
```

The `option` type is parameterized by a type `A` that will set the type of elements passed to the `Some` constructor. As so, the `option` type is an example of generic type.

1.3.3 Functional programming

As told in the presentation of the CIC, the Coq proof assistant permits to write functional programs, including the definition of recursive functions. The definition of a recursive function is performed with `Fixpoint` keyword. Here is an example of recursive function defined in Coq. The `pow` function takes two natural numbers `a` and `n` as inputs and yields `a` to the power `n`.

```
Fixpoint pow (a n : nat) {struct n} : nat :=
  match n with
  | O ⇒ 1
  | S m ⇒ a * pow a m
  end.
```

In the body of the `pow` function, the `match` construct performs pattern-matching over the structure of the input `n`. The input `n` is an element of the `nat` type, and thus it could either have been built with the `O` constructor or as the successor of another element of the `nat` type, i.e. with the `S` constructor. The `match` construct enumerates all the possible construction cases for the given input. Each construction case leads to a pipe-separated entry; for each entry, the structure of the input appears at the left of the arrow, and the result returned appears at the right the arrow. In the above example, `1` is returned if `n` equals `O`, and the result of the multiplication of `a` with the recursive call `pow a m` is returned if `n` is the successor of a certain `m`. In that case, we have `m = n − 1`, and then the recursive call `pow a m` can be read as `pow a (n − 1)`.

When declaring a recursive function, the user must specify which parameter is structurally decrementing through the recursive call. This is performed through `{struct` *id*`}` annotation, where *id* denotes one parameter of the declared function. This information permits to the Coq kernel to generate the fixpoint equation for the function, thus proving that the function is always terminating. For consistency reasons, all Coq functions must terminate and must be total. A user willing to implement a non-terminating function equivalently define the function as an inductive type where termination limitations do not apply. For instance, let us say that we want to implement this following *ill-formed* version of the `pow` function:

```
Fixpoint pow (a n : nat) {struct n} : nat :=
  match n with
  | 0 ⇒ 1
  | _ ⇒ a ∗ pow a n
  end.
```

Clearly, this function diverges for all `n` strictly superior to 0. The Coq kernel will not allow such a definition; thus, we can implement the `pow` function as the following relation `Pow` $\subseteq (\mathbb{N} \times \mathbb{N} \times \mathbb{N})$:

```
Inductive Pow (a : nat) : nat → nat → Prop :=
| Pow0 : Pow a 0 1
| Pown : forall n res, Pow a n res → Pow a n (a ∗ res).
```

The `Pow` relation takes three parameters of the `nat` type and projects a value in the `Prop` type (meaning that the `Pow` relation is a predicate). The third `nat` parameter corresponds to the result of the computation of a^n given that the two first parameters are a and n. Determining the result of the computation of a^n is equivalent to finding a natural number m that verifies that `Pow a n m` holds. In intuitionistic logic, finding a proof of the existence of a m such that `Pow a n m` holds amounts to explicitly building such a m. Here, one can notice that when the second parameter passed to the `Pow` relation is greater than zero, the formation rules of the `Pow` relation will not permit us to find a result for the computation. Thus, a tactic implementing a proof search for a m such that `Pow a n m` holds when $n > 0$ will never terminate.

1.3.4 Dependent types

In the listings that the reader will find in the following chapters, and also in the code repository associated with this thesis, some data structures are *dependently-typed* structures. Thus, we introduce here the notion of dependent type and how it is expressed with the Coq proof assistant.

A type is said to be dependent when its expression depends on one or more elements of other types. To give an example of dependent type, let us take the definition of polymorphic lists that carry their own length. In Coq, these lists are defined as follows:

```
Inductive listn (A : Type) : nat → Set :=
| niln : list A 0
| consn : forall n : nat, A → listn A n → listn A (S n).
```

The `listn` takes the type `A` of its elements as its first parameter, then its second parameter is an element of the `nat` type which represent the actual length of the list. Note that the first parameter, i.e. the `A` parameter, of the `listn` type alone is not sufficient to qualify `listn` as a dependent type. The `A` parameter is the expression of the polymorphism of the elements of the list involved in generic programming. Polymorphism relates to the fact that the `A` type is general enough to accept multiple types as the type of the list's elements. The `niln` constructor of the `listn`, i.e. the constructor of the empty list, has the type of lists of length 0. The `consn` constructor permits to add a new element at the head of an existing *tail* list to build a new list. Thus, the type of the resulting list is the type of lists of length $n + 1$, where n is the length of the tail list.

To further illustrate the use of dependent types, let us say that we want to write a function that takes two natural numbers n and m as inputs, and yields $n - m$ only if $n \geq m$. Thus, the function takes two parameters n and m, and a third parameter which is the proof that $n \geq m$. This third parameter *depends* on the two previous parameters, and thus the function is said to be a dependently-typed function. In Coq, it would be written as follows:

```
Definition my_sub (n m : nat) (pf : m <= n) : nat := n − m.
```

Even though, in its definition body, the `my_sub` function simply appeals to the Coq built-in substraction function, passing a proof that m is less than or equal to n adds a constraint to the computation of the substraction. One can see how dependent types can help check that the parameters of programs meet some properties at definition time. Constraining the type of parameters during the definition of programs reduces the proof efforts afterwards, but adds programming complexities at the moment of the definition. Thus, there is a trade-off between using dependent types to constraint the structures and programs at the moment of their definition, or letting the structures and programs as constraint loose as possible at the cost of having to prove much more properties afterwards.

To conclude the subject of dependent types, we often use *sigma* types to define a type of elements that meet a given property. Sigma types are *constructivist* versions, coming from the intuitionistic logic, of existential logic formulas. A sigma type expresses the dependence between a parameter and a proof of a given property that possibly depends on another parameter. As so, sigma types are useful to express intentional sets (cf. Section 1.1.2). In the Coq standard library, the definition of the sigma type is as follows:

```
Inductive sig (A:Type) (P: A → Prop) : Type := exist : forall x:A, P x → sig P.
```

The `sig` type only constructor takes an element `x` of type `A` along with a proof that `x` meets a certain property `P`. For instance, if we want to define the type of natural numbers that are strictly greater than zero, we can do it as follows:

```
Definition natstar := sig nat (fun n : nat ⇒ n > 0).
```

The property passed as the second argument of the `sig` type is expressed by a lambda abstraction (denoted by the `fun` keyword) that takes a parameter `n` of type `nat` and returns a proof that `n` is strictly greater than zero. The Coq standard library defines a notation to write sigma types as intensional sets. Thus, we can write the `natstar` type as follows:

```
Definition natstar2 := { x : nat | x > 0}.
```

We can leverage sigma types to rewrite the `my_sub` function presented above. In the following version, the type of the m parameter carries the proof that m is less than or equal to n:

```
Definition my_sub2 (n : nat) (m : { x : nat | x <= n }) : nat := n − (proj1_sig m).
```

Here, we can no longer directly substract n with m as the type of m is no longer `nat` but `{ x : nat | x <= n }`. We have to extract the first part of the m parameter with the help of the `proj1_sig` function. The first part of an element of the `{ x : nat | x <= n }` type corresponds to the natural number `x` verifying the following property `x <= n`, and the second part corresponds to the proof that `x` verifies the property.

Bibliography

[1] Yves Bertot and P. Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Berlin ; New York: Springer, 2004. 469 pp. ISBN: 978-3-540-20854-9.

[2] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, Dec. 2013. ISBN: 978-0-262-02665-9.

[3] Thierry Coquand and Christine Paulin. "Inductively defined types". In: *COLOG-88*. Ed. by Per Martin-Löf and Grigori Mints. Lecture Notes in Computer Science. Springer, 1990, pp. 50–66. ISBN: 978-3-540-46963-6. DOI: 10.1007/3-540-52335-9_47.

[4] William A Howard. "The formulae-as-types notion of construction". In: *To HB Curry: essays on combinatory logic, lambda calculus and formalism* 44 (1980), pp. 479–490.

[5] Joan Moschovakis. "Intuitionistic Logic". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Winter 2018. Metaphysics Research Lab, Stanford University, 2018. URL: https://plato.stanford.edu/archives/win2018/entries/logic-intuitionistic/.

[6] Yiannis Moschovakis. *Notes on Set Theory*. Undergraduate Texts in Mathematics. Springer-Verlag, 1994. ISBN: 978-1-4757-4153-7. DOI: 10.1007/978-1-4757-4153-7. URL: https://www.springer.com/gp/book/9781475741537.

[7] Christine Paulin-Mohring. "Introduction to the Coq Proof-Assistant for Practical Software Verification". In: *Tools for Practical Software Verification*. Ed. by Bertrand Meyer and Martin Nordio. Vol. 7682. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 45–95. ISBN: 978-3-642-35745-9 978-3-642-35746-6. DOI: 10.1007/978-3-642-35746-6_3. URL: http://link.springer.com/10.1007/978-3-642-35746-6_3 (visited on 10/14/2019).

[8] Stewart Shapiro and Teresa Kouri Kissel. "Classical logic". In: *The Stanford encyclopedia of philosophy*. Ed. by Edward N. Zalta. Spring 2021. Citation Key: ClassicalLogic. Metaphysics Research Lab, Stanford University, 2021. URL: https://plato.stanford.edu/archives/spr2021/entries/logic-classical/.

[9] The Coq Development Team. *Coq, version 8.13.2*. Citation Key: Coq. July 2021. URL: https://coq.inria.fr/.

[10] Glynn Winskel. *The formal semantics of programming languages: an introduction*. Citation Key: Winskel1993. MIT press, 1993.