UNIVERSITY OF BARCELONA

MASTER IN PURE AND APPLIED LOGIC

MASTER THESIS

When the laws of logic meet the logic of laws

Author:
Jorge DEL CASTILLO TIERZ

Tutor:
Joost J. JOOSTEN

Academic year 2017-2018

UNIVERSITAT DE BARCELONA

This page would be intentionally left blank if we would not wish to inform about that.

'If we desire respect for the law, we must first make the law respectable'.
Louis D. Brandeis, Cleveland Hill

Oh, look! Another intentionally left blank yet not blank page.

Contents

1 Introducing the thesis: three questions **1**
1.1 What is this thesis about? . . . 1
1.2 What is new? . . . 2
1.3 Who to thank? . . . 2

I The theoretical part **5**

2 Beginning at the beginning: the calculus $\lambda \rightarrow$ **7**
2.1 A brief introduction on the history of type theory . . . 7
2.2 Untyped lambda calculus . . . 8
2.3 Simply typed lambda calculus . . . 14

3 Extending $\lambda \rightarrow$: the systems $\lambda 2$, $\lambda \underline{\omega}$ and λP **19**
3.1 Second order typed lambda calculus $\lambda 2$ . . . 19
3.2 Types depending on types: $\lambda \underline{\omega}$ . . . 21
3.3 Types depending on terms: λP . . . 24

4 Putting all together: the calculus of inductive constructions **27**
4.1 The calculus λC . . . 27
4.2 Some results about λC . . . 29
4.3 Decidability of three problems . . . 32
4.4 Encoding first-order logic in type theory . . . 33

5 Applying logic: proof assistants **37**
5.1 What is a proof assistant? . . . 37
5.2 Coq: an overview . . . 38

II The case study part **41**

6 Becoming lawyers: Law 561 **43**
6.1 Logicians as lawmakers . . . 43
6.2 The law . . . 44
6.3 Some consistent yet absurd situations and some inconsistent readings . . . 46

7 Studying the law: on implication span of Reduced Weekly Rest Periods **49**
7.1 The problem . . . 49
7.2 Formalisation . . . 51

7.3 The results 55
7.4 Extended ambiguity 58

8 Changing the law: can we make it better? 61
8.1 A change in the Law: no RWRP + Compensation 61
8.2 A change in the Law: a shorter period for compensation 63
8.3 Do we have a correct ontology? 65
8.4 Final remarks 67

III The practical part 71

9 Getting our hands on Coq: a formalisation 73
9.1 Step 1: schedules 73
9.2 Step 2: interpretations 75
9.3 Step 3: legality 80
9.4 Step 4: ambiguity 91

10 Going further: the theorems 99
10.1 P-ambiguity 99
10.2 A-ambiguity 108

Bibliography 119

Chapter 1

Introducing the thesis: three questions

1.1 What is this thesis about?

Welcome, reader, to this humble thesis on types and laws. During the next pages, you will learn what type theory is and what it can be used for (in case you did not already know), specifically for using a kind of software called 'proof assistants' that can check proofs and help us formalise them correctly.

This is not intended to be a profound book on type theory, but rather an introduction for those unfamiliar with it. We only present the most important results and notions in order to understand the basics of the 'Calculus of Inductive Constructions', the formal theory that lies behind proof assistants.

Here follows a summary of the contents we will cover (spoiler alert):

- In the first part, called 'the theoretical part', we present different systems of type theory and some of their important results.
  - We first give an overview on untyped lambda calculus, which serves as a starting point for learning about λ-terms and the relations between them. We then introduce the notion of 'type' and define the system $\lambda \rightarrow$.
  - We discuss three different extensions of $\lambda \rightarrow$, each of them independent from the others: $\lambda 2$, $\lambda \underline{\omega}$ and λP. We can imagine them as the three directional vectors that start from the origin of coordinates (which corresponds to $\lambda \rightarrow$) in a 3-dimensional cube.
  - We then combine the three systems to get λC, a powerful system that can be used to encode logic.
  - Finally, we talk about proof assistants and how they relate to type theory. In particular, we present one of them, called 'Coq' and describe some of its properties and a simple example of a derivation.
- In the second part, called 'the case study part', we expose some flaws in the *Regulation (EC) No 561/2006 of the European Parliament and of the Council*, a law that regulates road transport.

  - We first present the law and point out some general problems it has.
  - We then focus on a subarticle of the law and, adopting a mathematical approach, we show that there exist situations (periods of time) of any length that are ambiguous. We shall define what this is later, but, in an informal way, it means that a police officer cannot be sure of the legality of the period.
  - Finally, we study what happens if we make some ad*j*ustments to the law with respect to the problem we found.

- In the third part, called 'the practical part', we combine parts I and II and provide a formalisation in Coq of the ontology that was defined in Chapter 7.

1.2 What is new?

In part I, most of what we present comes from different sources and the only new material we ad*d*ed are the proofs of some of the results that were too simple to be included in the other type theory presentations we consulted.

Parts II and III are completely new, with the exception of the actual text of the law. The examples from Chapter 6 largely arose within the research group on law and formal logic, the members of which I thank below.

1.3 Who to thank?

There are many people I should thank for this th*e*sis:

Joost J. Joosten for coming up with the idea for it, as well as the definition of *implication span* that all the second part is based on and the example that appears in Section 8.3. I would also want to thank him for the multiple corrections and ideas for improving the text. In particular, he conjectured the result in Theorem 7.3.6 and, thus, thought of the definition of *P-ambiguity*.

Ana Borges for being such a good and patient teacher and help me through the Coq field. I cannot thank her enough for this. Moreover, she wrote some fantastic notes for the research group in order to formalise the whole law (which is not an easy task) from which I took some of the examples that appear in Chapter 6.

Juan Conejero for being there whenever I wanted to discuss some aspect of the proofs, as well as for providing the alternative definitions for interpretation and legality that appear in Section 8.3.

The rest of the team, Aleix, Eric, Bjørn, Nika, Daniel and Mireia, for always coming up with new strange situations in which the law may fail. Also, for allowing me to be in their meetings and listen in a fascinated way how they do something that I would have thought was unachievable.

Raymond Smu*l*lyan and Pedro Buera, the reasons why I am here, studying the Master in Pure and Applied Logic.

And last, but not least, you, reader, for taking the time to go through the pages of this thesis. I hope you enjoy it. I will not say that you will not be able to separate your eyes from the text because I would be lying, but maybe you discover something that you like within these pages.

Are blank pages not boring?

Part I

The theoretical part

Blank page syndrome. Never again.

Chapter 2

Beginning at the beginning: the calculus $\lambda \rightarrow$

2.1 A brief introduction on the history of type theory

Type theory (or better, type theories) is one of the main fields in the intersection of Mathematics, Logic and Computer Science. In origin, its purposes included to fully formalise mathematics in an attempt to complete a *foundational quest* that had started in the 19th century.

At the beginning of the 20th century, Bertrand Russell writes a letter to Frege in relation to the latter's work *Begriffsschrift*, published in 1879. Russell states his disagreement with some of the concepts introduced in the book and presents a paradox. This paradox, now known as the *Russell paradox*, has become part of the common knowledge and is a frequent anecdote to break the ice during the first days of class in the degree of Mathematics in many universities. Informally, it poses a question regarding the set R of all sets that do not contain themselves, such as $\{1,2,3\}$, $\mathbb{N}$ or $\emptyset$. In symbols, $R = \{x | x \notin x\}$. The contradiction arises when we try to address the question: does R belong to R? If $R \in R$, then it is clear that R is a set that contains itself, but the elements in R are precisely the sets that do NOT contain themselves, a contradiction. If $R \notin R$, then it is a set that does not contain itself, so it should belong to R, a contradiction.

After some discussion between Russell and Frege, they both agree that they have to correct their works in order to dodge the paradox. However, as Russell's *Principia Mathematica* is already at the editor and ready to be published, he can only quickly made some amendments and write an appendix in which he starts exploring new concepts. Russell argues that R should not be an argument of the relation "belongs to R". There should be different ontological levels so that new levels can only predicate on previous levels. This paradigm is referred to as predicativism. And so, type theory is born. For the interested reader, the correspondence between Russell and Frege can be found at [Russell]. More on Russell paradox can be read at [Irvine].

Russell changes his mind several times over the subsequent years, about the better way to tackle the problem. In 1908, he finally formulates the ramified theory of types ([Urquhart]) by means of the Axiom of Reducibility: "any truth function can be expressed by a formally equivalent predicative truth function". However, some years later, in the 1920s, Leon Chwistek and Frank P. Ramsey realise that there is a simpler way to present type theory that does not require the Axiom of Reducibility. This is what we now call the simple type theory. During

the 1920s and the 1930s, different authors such as Carnap, Quine and Tarski delve into this territory and, in 1940, Alonzo Church introduces the simply typed lambda calculus ([Church]), which is a variation of the simple type theory that solves some paradoxes and is shown to be a foundation of mathematics.

Nowadays, many other type theories have been and are being developed or studied more profoundly. For example, Martin-Löf's intuitionistic type theory ([Dybjer]), which is used in some areas of constructive mathematics, the Univalent Foundations Program's *Homotopy Type Theory* ([Univalent]) or Thierry Coquand's calculus of inductive constructions, which is the basis for the proof assistant Coq, which we will talk about later.

2.2 Untyped lambda calculus

In our presentation, most of the information we are giving comes from [Nederpelt]'s Chapters 1 to 7, with some scents from [Barendregt]. We start now by showing some basic definitions and results on lambda calculus without yet taking into account types, that we shall present later. Each of the following sections are built upon the previous ones, in the same sense that first order logic is built upon propositional logic and second-order, upon first-order.

In type theory, there is something similar to the concept of functions in mathematics. We normally have an expression, e.g. $3+x^2+3x^3$, and abstract it to create a function, i.e. $f(x) = 3+x^2+3x^3$. And given a function, we apply it to a certain value, for example $f(2) = 3+2^2+3*2^3 = 3+4+24 = 31$. In type theory, we use the word *expression* (or λ*-term*) instead of *function* because we need a more general notion. A function here is an expression which starts by some $\lambda x.$ for some variable x. The λ-operator abstracts an expression M into a function $(\lambda x.M)$.

In the following chapters, we are going to deal with two basic construction principles:

1. Abstraction: given an expression M and a variable x, the abstraction of x over M is $(\lambda x.M)$.

2. Application: given two expressions M and N, the application of M to N is (MN).

Example 2.2.1. The following are some examples of the abstraction and application principles.

- The abstraction of x over $3x-z$ is $(\lambda x.3x-z)$. Note that the variable need not appear in the expression: the abstraction of y over $3x-z$ is $(\lambda y.3x-z)$. This also makes sense in mathematics (think of the constant function $f(x) = 2$).

- The abstraction of y over $\lambda x.x-y$ is $(\lambda y.(\lambda x.x-y))$.

- The application of $\lambda x.x+2$ to y and to 5 are, respectively, $((\lambda x.x+2)(y))$ and $\big((\lambda x.x+2)(5)\big)$.

- We can also apply functions to functions: the application of $\lambda x.x$ to $\lambda y.y$ is $((\lambda x.x)(\lambda y.y))$.

- It is not necessary, in principle, that we apply a function to an expression, we can apply an expression to an expression: $((x^2+1)(3))$ and (xz) are examples of this.

One could be tempted to think that an application is the same as substituting a variable by the corresponding expression but that is not the case. What would mean then $((x^2+y)(3))$? An application is only a syntactical formation rule and we shall introduce later a formal definition for a substitution (called β-reduction).

Before we give the precise definition of what an expression is, we make an important remark: as it usually happens in logic, we must assume the existence of an infinite set of variables and we denote it by $V = \{x, y, z, ...\}$.

Definition 2.2.2. *Expressions* (or λ*-terms*) are recursively defined as:

i) (Variables) If $x \in V$, then x is an expression;

ii) (Applicat*i*on) If M and N are expressions, then (MN) is also an expression;

iii) (Abstraction) If $x \in V$ and M is an expression, then $(\lambda x.M)$ is also an expression.

For example, x, (xy) and $\big((\lambda x.x)(\lambda x.x)\big)$ are all expressions. From now on, we shall try to use the alternative name for expressions, 'λ-terms', as it is more standard in type theory.

Definition 2.2.3. The set of *subterms* of a λ-term is recursively defined as:

i) $Sub(x) = \{x\}$, for all $x \in V$.

ii) $Sub((MN)) = Sub(M) \cup Sub(N) \cup \{(MN)\}$.

iii) $Sub((\lambda x.M)) = Sub(M) \cup \{(\lambda x.M)\}$.

We say that L is a subterm of M if $L \in Sub(M)$.

Example 2.2.4. The set of subterms of the λ-term $\big((\lambda x.y(\lambda z.xy))z\big)$ is $\big\{\big((\lambda x.y(\lambda z.xy))z\big)$, $(\lambda x.y(\lambda z.xy))$, $(y(\lambda z.xy))$, y, $(\lambda z.xy)$, xy, x, $z\big\}$.

Notation 2.2.5. *If two λ-terms M and N are equal, i.e. they have the same symbols and in the same order, we write $M \equiv N$. We do not write $M = N$ because it could be confused with other* equalities *we will use later, namely $=_\alpha$ and $=_\beta$. Hence, $(xy) \equiv (xy)$ but $(\lambda x.x) \not\equiv (\lambda y.y)$ and $(xy) \not\equiv (yx)$.*

Lemma 2.2.6. *Sub has the reflexivity and transitivity properties, that is, for any λ-terms M, N and L, $M \in Sub(M)$ and, if $L \in Sub(N)$ and $N \in Sub(M)$, then $L \in Sub(M)$.*

Proof. For reflexivity: $M \in Sub(M)$. We prove the result by induction on the complexity of M. If $M \equiv x$, since $Sub(x) = \{x\}$ by i) in Definition 2.2.3, we get $x \in Sub(x)$. If $M \equiv (M_1M_2)$, by ii), $\{(M_1M_2)\} \subseteq Sub((M_1M_2))$ and, hence, $(M_1M_2) \in Sub((M_1M_2))$. Finally, if $M \equiv (\lambda x.N)$, by iii), $\{(\lambda x.N)\} \subseteq Sub(\lambda x.N))$ and, hence, $(\lambda x.N) \in Sub((\lambda x.N))$.

For transit*i*vity: if $L \in Sub(N)$ and $N \in Sub(M)$, then $L \in Sub(M)$. Again, we prove this by induction on the complexity of the λ-term M. If $M \equiv x$, then $Sub(M) = \{x\}$, and it must be the case that $N \equiv x$, so $Sub(N) = \{x\}$. Hence, $L \equiv x$ and $L \in Sub(M)$.

If $M \equiv (M_1 M_2)$, then $Sub(M) = Sub(M_1) \cup Sub(M_2) \cup \{(M_1 M_2)\}$. If $N \equiv (M_1 M_2)$, it is clear that $L \in Sub((M_1 M_2))$. If $N \in Sub(M_1)$ or $N \in Sub(M_2)$, by the induction hypothesis, $L \in Sub(M_1)$ or $L \in Sub(M_2)$, respectively. In either case, $L \in Sub(M)$.

Finally, if $M \equiv (\lambda x.M_1)$, then $Sub(M) = Sub(M_1) \cup \{(\lambda x.M_1)\}$. If $N \equiv (\lambda x.M_1)$, then it is clear that $L \in Sub\big((\lambda x.M_1)\big)$. If $N \in Sub(M_1)$, by the induction hypothesis, $L \in Sub(M_1)$ and we conclude that $L \in Sub(M)$.

□

Notation 2.2.7. *As we normally do in logic, we may omit unnecessary parentheses following the convention:*

- Parentheses on the outermost position may be omitted: we can write MN instead of (MN) *and* $\lambda x.M$ *instead of* $(\lambda x.M)$.

- Successive applications may be combined in a left-associative way: we may write MNL for $(MN)L$.

- Successive abstractions may be combined in a right-associative way using a single λ*: we may write* $\lambda xy.M$ *for* $\lambda x.(\lambda y.M)$.

- Application is more binding than abstraction: we may write $\lambda x.MN$ *for* $\lambda x.(MN)$.

The next definition is analogous to the one for free variables in a first order formula, as it follows the same idea.

Definition 2.2.8. The set of *free variables* FV of a λ-term M is defined recursively as:

i) $FV(x) = \{x\}$;

ii) $FV(MN) = FV(M) \cup FV(N)$;

iii) $FV(\lambda x.M) = FV(M) \setminus \{x\}$.

For instance, the free variables in $(\lambda x.xy)(\lambda y.x)$ are $\{x, y\}$. The free variable x comes from the one in the second half, since the rest of the x's are not free; and the free variable y, from the first term $(\lambda x.xy)$, because the other y we say is a binding variable.

Definition 2.2.9. A λ-term M is *closed* if $FV(M) = \varnothing$.

In the example above, $(\lambda x.xy)(\lambda y.x)$ is not closed, but $(\lambda xy.xy)(\lambda y.y)$ is.

In the same manner that the first order formulas $\forall x(P(x))$ and $\forall y(P(y))$ are essentially equivalent, the λ-terms $\lambda x.xz$ and $\lambda y.yz$ behave very similarly. The binding variable (the one next to the λ) is only a name we give to refer to the function. We may perform a renaming of variables if necessary and we call this process α*-conversion*.

Definition 2.2.10. (α*-conversion* or α*-equivalence*) Let M be a λ-term and x, y be variables. Let $M^{x \rightarrow y}$ be the result of replacing every free occurrence of x in M by y. Define the re*l*ation $=_\alpha$ as:

i) $\lambda x.M =_\alpha \lambda y.M^{x\to y}$ if $y \notin FV(M)$ and y is not a binding variable in M.

ii) If $M =_\alpha N$, then $ML =_\alpha NL$, $LM =_\alpha LN$ and $\lambda z.M =_\alpha \lambda z.N$, for any λ-term L and any variable z.

iii) $M =_\alpha M$.

iv) If $M =_\alpha N$, then $N =_\alpha M$.

v) If $M =_\alpha N$ and $N =_\alpha L$, then $M =_\alpha L$.

If for two λ-terms M and N, it holds $M =_\alpha N$, we say they are α-equivalent or that one is an α-variant of the other.

Example 2.2.11.

1- In the λ-term $(\lambda x.x(\lambda y.(xy)z))(xz)$, we may change any of the binding variables for fresh ones: $(\lambda x.x(\lambda y.(xy)z))(xz) =_\alpha (\lambda x.x(\lambda v.(xv)z))(xz) =_\alpha (\lambda u.u(\lambda v.(uv)z))(xz)$. But we cannot replace them by a variable that appears binding or free in the term, such as y or z: $(\lambda x.x(\lambda y.(xy)z))(xz) \neq_\alpha (\lambda y.y(\lambda y.(yy)z))(xz)$ nor $(\lambda x.x(\lambda z.(xz)z))(xz)$.

2- $\lambda xy.xzy =_\alpha \lambda vy.vzy =_\alpha \lambda vu.vzu \neq_\alpha \lambda vv.vzv$.

Lemma 2.2.12. *Let M_1, N_1, M_2 and N_2 be λ-terms such that $M_1 =_\alpha N_1$ and $M_2 =_\alpha N_2$. Then, $M_1M_2 =_\alpha N_1N_2$ and $\lambda x.M_1 =_\alpha \lambda x.N_1$ for any variable x.*

Proof. Since $M_1 =_\alpha N_1$, we know that $M_1L =_\alpha N_1L$ for any λ-term L, in particular for $L = M_2$. On the other hand, we have that, since $M_2 =_\alpha N_2$, $LM_2 =_\alpha LN_2$, in particular for $L = N_1$. That is, $M_1M_2 =_\alpha N_1M_2$ and $N_1M_2 =_\alpha N_1N_2$. Therefore, $M_1M_2 =_\alpha N_1N_2$.

That $\lambda x.M_1 =_\alpha \lambda x.N_1$ follows trivially from the definition.

□

With this ingredient in our hands, we can properly define substitutions, which are the key element for the β-reductions. But, let us not anticipate too much and see the definition:

Definition 2.2.13. The *substitution* of a variable x for a λ-term N in P is denoted by $P[x := N]$ and it is defined by recursion:

i) $x[x := N] \equiv N$;

ii) $y[x : N] \equiv y$ if $x \not\equiv y$;

iii) $(PQ)[x := N] \equiv (P[x := N])(Q[x := N])$;

iv) $(\lambda y.P)[x := N] \equiv \lambda z(P^{y\to z}[x := N])$, where $P^{y\to z}$ is an α-variant of $\lambda y.P$ such that z is not free in N.

Note that we are making use of the $\equiv$ symbol and not $=_\alpha$. This means that the result of a substitution is syntactically exact to the one given in the definition. For example, $x[x := y]$ is exactly the term y.

Example 2.2.14.

1- Consider $(\lambda xy.zzx)[z := y]$. It is in the form of iv), so we have to choose an α-variant that fits the definition. Since x is not free in y, we can use $\lambda xy.zzx$ itself and $(\lambda xy.zzx)[z := y] \equiv \lambda x.((\lambda y.zzx)[z := y])$. Again, we are in the context of Condition iv), but this time we must find a different α-variant, because y is free in y. For example, take $\lambda u.zzx$ and we have that $\lambda x.((\lambda y.zzx)[z := y]) \equiv \lambda xu.((zzx)[z := y])$. We now use iii) twice to get $\lambda xu.((zzx)[z := y]) \equiv \lambda xu.((z[z := y])(z[z := y])(x[z := y])$. Finally, by i) and ii), we conclude that $(\lambda xy.zzx)[z := y] \equiv \lambda xu.yyx$.

2- We can check that $(\lambda y.yx)[x := xy] \equiv \lambda u.u(xy)$.

Lemma 2.2.15. *Let M_1, N_1, M_2 and N_2 be λ-terms such that $M_1 =_\alpha N_1$ and $M_2 =_\alpha N_2$. Then, $M_1[x := M_2] =_\alpha N_1[x := N_2]$.*

Proof. First, we show that $M[x := M_2] =_\alpha M[x := N_2]$ for any M. We proceed by induction on the complexity of M.

If $M \equiv x$, then $M[x := M_2] \equiv M_2 =_\alpha N_2 = M[x := N_2]$. If $M \equiv y$ with $y \not\equiv x$, then $M[x := M_2] \equiv y =_\alpha y = M[x := N_2]$.

If $M \equiv PQ$, then $M[x := M_2] \equiv (P[x := M_2])(Q[x := M_2])$, which, by the induction hypothesis, is α-equivalent to $(P[x := N_2])(Q[x := N_2]) \equiv M[x := N_2]$.

If $M \equiv \lambda y.P$, then $M[x := M_2] \equiv \lambda z_1(P^{y \rightarrow z_1}[x := M_2])$, where $P^{y \rightarrow z_1}$ is an α-invariant of $\lambda y.P$ such that z_1 is not free in M_2. Analogously, $M[x := N_2] \equiv \lambda z_2(P^{y \rightarrow z_2}[x := N_2])$. We can end the proof by using the induction hypothesis and the definition of α-conversion.

Similarly, we can prove that $M_1[x := N] =_\alpha N_1[x := N]$ for any N. And using both results we conclude that $M_1[x := M_2] =_\alpha M_1[x := N_2] =_\alpha N_1[x := N_2]$, which was the desired result. □

Finally, we can define β-reductions, which are the analogous in type theory of the intermediate steps that we perform when we apply a value to a function in mathematics. For example, with $f(x) = 2 + (x-1)^2$, when we compute $f(4)$, we can perform the operations one at a time, $f(4) = 2 + (4-1)^2 = 2 + 3^2 = 2 + 9 = 11$, or summarise the process, writing only some of the steps, $f(4) = 2 + 3^2 = 11$. Both strings of equalities are correct and in each one of them we *reduce* (simplify) the expression. The idea of β-reductions is the same. We first give the definition for the one-step β-reduction.

Definition 2.2.16. The relation $\rightarrow_\beta$ (or *one-step β-reduction*) is defined recursively as:

i) $(\lambda x.M)N \rightarrow_\beta M[x := N]$;

ii) If $M \rightarrow_\beta N$, then $ML \rightarrow_\beta NL$, $LM \rightarrow_\beta LN$ and $\lambda x.M \rightarrow_\beta \lambda x.N$.

Example 2.2.17. Consider the expression $(\lambda x.x)((\lambda y.zy)z)$. We can perform two different one-step β-reductions:

1- By i), $(\lambda x.x)((\lambda y.zy)z) \to_\beta (\lambda y.zy)z$. Here, we could do a further reduction, again by i): $(\lambda y.zy)z \to_\beta zz$.

2- If we look at the subexpression $(\lambda y.zy)z$, by i), we have that $(\lambda y.zy)z \to_\beta zz$. Using now ii), we get that $(\lambda x.x)((\lambda y.zy)z) \to_\beta (\lambda x.x)(zz)$. Finally, by i), $(\lambda x.x)(zz) \to_\beta zz$.

Note that in both cases, we obtain the same result. In some sense, we did the same reductions, but in reverse order.

Definition 2.2.18. The relation $\twoheadrightarrow_\beta$ (or β-*reduction*) is defined as:

$M \twoheadrightarrow_\beta N$ if there is some $n \geq 0$ and there are $M_0, ..., M_n$ such that $M \equiv M_0$, $M_n \equiv N$ and for each $0 \leq i < n$, $M_i \to_\beta M_{i+1}$.

In other words, $M \twoheadrightarrow_\beta N$ if there is a finite chain of one-step β-reductions from M to N.

Example 2.2.19. In the last example, we saw that $(\lambda x.x)((\lambda y.zy)z) \twoheadrightarrow_\beta zz$. Note that $(\lambda x.xx)z$ also reduces to zz, but this does not mean that $(\lambda x.x)((\lambda y.zy)z) \twoheadrightarrow_\beta (\lambda x.xx)z$ nor $(\lambda x.xx)z \twoheadrightarrow_\beta (\lambda x.x)((\lambda y.zy)z)$.

Remark 2.2.20. *The one-step β-reduction is a special case of the β-reduction: assume $M \to_\beta N$ and take $n = 1$, $M_0 \equiv M$ and $M_1 \equiv N$. Hence, $M \twoheadrightarrow_\beta N$.*

Lemma 2.2.21. *The relation $\twoheadrightarrow_\beta$ is reflexive and transitive.*

Proof. The relation is reflexive: let M be a λ-term and take $n = 0$ and $M_0 \equiv M$. Hence, $M \twoheadrightarrow_\beta M$.

The relation is transitive: let M, N and L be λ-terms such that $M \twoheadrightarrow_\beta N$ and $N \twoheadrightarrow_\beta L$. Then, there are $n, m \geq 0$ and $M_0, ..., M_n, N_0, ...N_m$ such that $M \equiv M_0$, $M_n \equiv M$, for every $0 \leq i < n$, $M_i \to_\beta M_{i+1}$, $N \equiv N_0$, $N_m \equiv L$ and for every $0 \leq j < m$, $N_j \to_\beta N_{j+1}$. Simply take $n + m$ and $M_0, ..., M_n, N_1, ...N_m$ and we conclude that $M \twoheadrightarrow_\beta L$.

□

Definition 2.2.22. The relation $=_\beta$ is defined as:

$M =_\beta N$ if and only if there is some $n \geq 0$ and terms $M_0, ..., M_n$ such that $M_0 \equiv M$, $M_n \equiv N$ and for each $0 \leq i < n$, either $M_i \to_\beta M_{i+1}$ or $M_{i+1} \to_\beta M_i$.

It is easy to show that the relation $=_\beta$ is, as one would expect, an equivalence relation. Moreover, it extends $\twoheadrightarrow_\beta$, that is, if $M \twoheadrightarrow_\beta N$ or $N \twoheadrightarrow_\beta M$, then $M =_\beta N$.

We finish this section by referring to an important theorem that states that when performing reductions, the order when applying the rules does not make a difference. Some of this we already hinted at the end of Example 2.2.17: we followed two different β-reduction paths from $(\lambda x.x)((\lambda y.zy)z)$ and ended up with the same term zz.

This theorem is known as the Church-Rosser Theorem or the Confluence Theorem.

Theorem 2.2.23. *(Confluence)*
Let M, N_1 and N_2 be λ-terms. If $M \twoheadrightarrow_\beta N_1$ and $M \twoheadrightarrow_\beta N_2$, then there exists some λ-term N such that $N_1 \twoheadrightarrow_\beta N$ and $N_2 \twoheadrightarrow_\beta N$.

Proof. For the proof of this theorem, see [Barendregt], result 4.19.

□

2.3 Simply typed lambda calculus

In the previous section, we talked about λ-terms in a wide sense. As it happens in mathematics, functions need to be restricted to a certain domain, e.g. the natural numbers, rational tuples of length 4 or even functions with two arguments. In type theory, this concept of a domain is grasped by the *types*.

In this section, we define simple types and construct a system $\lambda \rightarrow$, which we will extend in the subsequent pages in order to add more types to formalise different aspects.

We assume there is an infinite set of type variables $\mathbb{V} = \{\alpha, \beta, \gamma, ...\}$. Do not confuse this set with V, the infinite set of term variables x, y, z,...

Definition 2.3.1. The set $\mathbb{T}$ of *simple types* is defined recursively as:

i) $\mathbb{V} \subseteq \mathbb{T}$;

ii) If $\sigma, \tau \in \mathbb{T}$, then $(\sigma \rightarrow \tau) \in \mathbb{T}$.

Type variables intend to represent basic types, such as *int* for the integers, *nat* for the naturals, *list* for lists, etc. Types with arrows, which we unimaginatively call *arrow types*, represent functions between simpler types. For instance, the functions in $(nat \rightarrow nat)$ are all the functions from the natural numbers to the natural numbers.

Notation 2.3.2. *We may omit parentheses in such a way that arrows are right-associative. For example, $(\alpha \rightarrow \beta \rightarrow \gamma)$ stands for $(\alpha \rightarrow (\beta \rightarrow \gamma))$. We can always omit the outermost parentheses.*

Definition 2.3.3. A *statement* is an expression of the form $M : \sigma$ and it is read as 'M has type σ'.

From this moment, we assume that variables have a unique type, that is, it is not possible that σ_1 and σ_2 are different types and $x : \sigma_1$, $x : \sigma_2$ for some variable x.

Example 2.3.4.

1- Let us see what this translates into regarding the two basic construction principles from Section 2.2.

Application: from M and N, we get MN. In order to perform a correct application, we should have a function and a term that belongs to the function's domain. That is, $M : \sigma \to \tau$ ('M takes an input of type σ and outputs a term of type τ'), $N : \sigma$. After the application, the term has type the output of the function: $MN : \tau$.

Abstraction: from M, we get $\lambda x.M$ for some variable x. We are creating a function with output M and input x. Therefore, we need to know the types of x and M, say $x : \sigma$ and $M : \tau$. The abstraction has type $\lambda x.M : \sigma \to \tau$.

2- Consider the λ-term MM. We try to deduce its type: we see that there is an application, so $M : \sigma \to \tau$, $M : \sigma$ and $MM : \tau$, for some σ, τ. The following result about the uniqueness of types can be proved: every term has a unique type.

This implies that $\sigma \to \tau$ is the same type as σ, which makes no sense. Therefore, MM cannot have a type and does not belong to the theory of simply typed lambda calculus, unlike in the previous section.

Definition 2.3.5. A term M is *typable* if there is some type σ such that $M : \sigma$.

Example 2.3.6. As we saw, the term MM is not typable, neither is $x(\lambda x.x)$: assume $x : \sigma$. Then, $\lambda x.x : \sigma \to \sigma$. Since we are performing an application of x, there are types τ and γ such that $\sigma \equiv \tau \to \gamma$. But the types of the input of x and $\lambda x.x$ should match: $\tau \equiv \sigma \to \sigma$. Since $\sigma \equiv \tau \to \gamma$, we can conclude that this is not the case and the term is not typable.

What about the term $(\lambda x.M)M$? Assume $x : \tau$ and $M : \sigma$ for some types τ and σ. The term $\lambda x.M$ has type $\tau \to \sigma$ and, for performing the application, the input x and M should have the same type. Therefore, if $\sigma \equiv \tau$, $(\lambda x.M)M : \sigma$. The term is typable.

Consider now the term $(\lambda x.M)(MN)$. Since we are performing the application MN, the types should match, so $M : \sigma \to \tau$ and $N : \sigma$, for some types σ and τ. Thus, $MN : \tau$. Assume $x : \gamma$, so $\lambda x.M : \gamma \to (\sigma \to \tau)$. Finally, the types of $\lambda x.M$ and MN must match: $\gamma \equiv \tau$ and we conclude that $(\lambda x.M)(MN) : \sigma \to \tau$. This term is typable as well.

For the rest of our discussion and for the sake of clarity, we shall write $\lambda x : \sigma.M$ for some type σ instead of $\lambda x.M$. This way, we explicitly give the type of the variable x.

Definition 2.3.7.

i) Let $M : \sigma$ be a statement. We say that M is the *subject* of the statement and σ, the *type*.

ii) A *declaration* is of the form $x : \sigma$. That is, a declaration is a statement with a variable as subject.

iii) A *context* is a list of declarations with different subjects. A context may be empty.

iv) A *judgement* has the form $\Gamma \vdash M : \sigma$, where Γ is a context and $M : \sigma$ is a statement.

Example 2.3.8. An example of a statement is $(\lambda x : \tau.M)(MN) : \sigma \to \tau$. Recall that it appeared in the last page, when we were deducing the types of some terms. Of course, $(\lambda x : \tau.M)(MN) : (\sigma \to \gamma) \to (\gamma \to (\sigma \to \tau))$ is another statement with the same subject. We cannot say that a statement is correct or not, unless we are given a context (the variable x has type τ, the variable y, type γ, etc.). An example of a context is $y : \sigma, z : \sigma$. Putting both examples together, we obtain a judgement: $y : \sigma, z : \sigma \vdash (\lambda x : \tau.M)(MN) : \sigma \to \tau$.

We define the formal system $\lambda \to$ as the derivation system with the following rules:

$$\frac{}{\Gamma \vdash x : \sigma}\ \text{var} \quad \text{if } x : \sigma \in \Gamma$$

$$\frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}\ \text{appl}$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma . M : \sigma \to \tau}\ \text{abst}$$

Definition 2.3.9. A term M is *legal* if there are a context Γ and a type σ such that $\Gamma \vdash M : \sigma$.

Remark 2.3.10. *In the second part of this thesis, we are going to use the word* legal *with a completely different meaning, referring to legality with respect to a law. The notion in the previous definition only makes sense under the scope of type theory, do not confuse them, or else you could end up in prison for following different laws. We do not take responsibility for any crime the reader may commit after reading this thesis.*

Example 2.3.11. The next example provides a proof of an instance of the term we studied in the Example 2.3.8: $y : \alpha \to \beta, z : \alpha \vdash (\lambda x : \beta . y)(yz) : \alpha \to \beta$. We first show that $y : \alpha \to \beta, z : \alpha \vdash yz : \beta$:

$$\frac{\dfrac{}{y : \alpha \to \beta, z : \alpha \vdash y : \alpha \to \beta}\ \text{var} \qquad \dfrac{}{y : \alpha \to \beta, z : \alpha \vdash z : \alpha}\ \text{var}}{y : \alpha \to \beta, z : \alpha \vdash yz : \beta}\ \text{appl}$$

Using this proof, we complete our derivation:

$$\frac{\dfrac{\dfrac{}{y : \alpha \to \beta, z : \alpha, x : \beta \vdash y : \alpha \to \beta}\ \text{var}}{y : \alpha \to \beta, z : \alpha \vdash \lambda x : \beta . y : \beta \to (\alpha \to \beta)}\ \text{abst} \qquad \dfrac{}{y : \alpha \to \beta, z : \alpha \vdash yz : \beta}}{y : \alpha \to \beta, z : \alpha \vdash (\lambda x : \beta . y)(yz) : \alpha \to \beta}\ \text{appl}$$

In order to show that it follows in $\lambda \to$ we go from bottom to top. First, we check that the only rule we can start with is appl. In general, because of the formation rules, there will not be any situation in which more than one rule can be applied.

For the application rule, two premises are required: $y : \alpha \to \beta, z : \alpha \vdash \lambda x : \beta . y : \beta \to (\alpha \to \beta)$ and $y : \alpha \to \beta, z : \alpha \vdash yz : \beta$. We start with the first one and realise it is an instance of abstraction, so we use the abst rule, which needs a premise: $y : \alpha \to \beta, z : \alpha, x : \beta \vdash y : \alpha \to \beta$. But this one is a var axiom and we close this branch.

For the second one, $y : \alpha \to \beta, z : \alpha \vdash yz : \beta$, it is again an instance of abstraction, so we use the abs rule, for which we need two premises: $y : \alpha \to \beta, z : \alpha \vdash y : \alpha \to \beta$ and $y : \alpha \to \beta, z : \alpha \vdash z : \alpha$. Finally, we can close both branches by using the var axiom twice.

Proposition 2.3.12. *(Subterm lemma)*
Let M and N be λ-terms such that N is a subterm of M. Let Γ be a context and σ, a type.

If $\Gamma \vdash M : \sigma$, then there are Γ' and σ' such that $\Gamma' \vdash N : \sigma'$.

A close look at the derivation rules of the calculus tells us that the proof of the next lemma is trivial (via an induction on the length of the derivation):

Lemma 2.3.13. *(Generation Lemma)*
Let Γ be a context, M, N be terms, x be a variable and σ, τ, γ be types. Then:

i) If $\Gamma \vdash x : \sigma$, then $x : \sigma \in \Gamma$;

ii) If $\Gamma \vdash MN : \tau$, then there is some σ such that $\Gamma \vdash M : \sigma \to \tau$ and $\Gamma \vdash N : \sigma$;

iii) If $\Gamma \vdash \lambda x : \sigma.M : \tau$, then there is some γ such that $\Gamma, x : \sigma \vdash M : \gamma$ and $\tau \equiv \sigma \to \gamma$.

Corollary 2.3.14. *(Uniqueness of types)*
If $\Gamma \vdash M : \sigma$ and $\Gamma \vdash M : \tau$, then $\sigma \equiv \tau$.

Proof. We prove the result by induction on the complexity of M.

If $M \equiv x$, then, by the Generation Lemma, $x : \sigma \in \Gamma$ and $x : \tau \in \Gamma$. By the definition of context, we conclude that $\sigma \equiv \tau$.

If $M \equiv M_1 M_2$, then, by the Generation Lemma, $\Gamma \vdash M_1 : \gamma_1 \to \sigma$ and $\Gamma \vdash M_2 : \gamma_1$ for some γ_1. On the other hand, $\Gamma \vdash M_1 : \gamma_2 \to \tau$ and $\Gamma \vdash M_2 : \gamma_2$ for some γ_2. By the induction hypothesis, from $\Gamma \vdash M_2 : \gamma_1$ and $\Gamma \vdash M_2 : \gamma_2$, we deduce that $\gamma_1 \equiv \gamma_2$. Moreover, we can also deduce that $\gamma_1 \to \sigma \equiv \gamma_2 \to \tau$, from which we conclude that $\sigma \equiv \tau$.

If $M \equiv \lambda x : \rho.N$, then, by the Generation Lemma, $\Gamma, x : \rho \vdash N : \gamma_1$ and $\sigma \equiv \rho \to \gamma_1$ for some γ_1. On the other hand, $\Gamma, x : \rho \vdash N : \gamma_2$ and $\tau \equiv \rho \to \gamma_2$ for some γ_2. By the induction hypothesis, from $\Gamma, x : \rho \vdash N : \gamma_1$ and $\Gamma, x : \rho \vdash N : \gamma_2$, we deduce that $\gamma_1 \equiv \gamma_2$. Thus, $\rho \to \gamma_1 \equiv \rho \to \gamma_2$, that is, $\sigma \equiv \tau$.

□

As a final note, we would like to remark that we can prove the consistency of the system $\lambda \to$ as a corollary of the Confluence theorem for $\lambda \to$, in the sense that not all judgements $\Gamma \vdash M : \sigma$ are derivable. The proof involves β-normal forms, which we did not define in this thesis, not because it is something difficult but rather because we had to choose which concepts to include, as there are too many to fit.

Theorem 2.3.15. *(Consistency)*
There exist some context Γ, term M and type σ such that $\Gamma \vdash M : \sigma$ is not a $\lambda \to$ theorem. Therefore, the system $\lambda \to$ is consistent.

Proof. An idea of the proof can be found in [Jesse], section 6.

□

All right. I promise this one is 100% blank.

Chapter 3

Extending $\lambda \rightarrow$: the systems $\lambda 2$, $\lambda \underline{\omega}$ and λP

In the previous chapter, we defined the system $\lambda \rightarrow$ which can be used to speak about *terms that depend on terms*, e.g. $\lambda x : \sigma.M$, which depends on the term x. In this chapter, we shall extend $\lambda \rightarrow$ in three different ways to speak about *terms depending on types*, *types depending on types* and *types depending on terms*. Each of these systems is in some sense independent from the others, although they share some symbols (like '$*$') and some of the axioms are identical. In the next chapter, we shall join the three puzzle pieces to create a new system.

3.1 Second order typed lambda calculus $\lambda 2$

The first of such systems- *terms depending on types*- arises naturally and is motivated by the need of having general functions that can be applied everywhere, no matter the type. For example, in $\lambda \rightarrow$ there is no unique identity function, but rather there are various different identity functions. For each type α, the identity for x of type α is $\lambda x : \alpha.x$. But we would like to have a general function that can take as input whatever term and outputs the same term. This we can do in second-order lambda calculus.

The trick consists in allowing a type variable $*$ over which we can abstract. Thus, we would allow expressions like $\lambda \alpha : *.\lambda x : \alpha.x$. Note that now the type α has another type, $*$, used for representing the *type of all types*. Hence, the identity function is a term that depends on the type α.

This kind of abstraction is a *second order abstraction* (or *type abstraction*). Since in $\lambda \rightarrow$ we did not have any construction of this form, we need to define a new type that corresponds to the type abstraction. For this, we introduce a type binder Π.

Definition 3.1.1. (Types)
Types in $\lambda 2$ are defined recursively as:

i) If $\alpha \in \mathbb{V}$, then α is a type.

ii) The constant $*$ is a type.

iii) If σ, τ are types, then $(\sigma \rightarrow \tau)$ is a type.

iii) If σ is a type and α is a type variable, then $(\Pi \alpha : *.\sigma)$ is a type.

Definition 3.1.2. (Terms)
Terms in $\lambda 2$ are defined recursively as:

i) If $x \in V$ and σ is a type, then $(x : \sigma)$ is a term.

ii) If $(M : \sigma \to \tau)$ and $(N : \sigma)$ are terms and, then $(MN : \tau)$ is a term.

iii) If $(M : \sigma)$ is a term, x is a term variable and α is a type, then $(\lambda x : \alpha.M : \alpha \to \sigma)$ is a term.

iv) If $(M : \sigma)$ is a term and α is a type variable, then $(\lambda\alpha : *.M : \Pi\alpha : *.\sigma)$ is a term.

Remark 3.1.3. *From now on (and that includes the rest of the systems we shall present in this part of the thesis), we will adopt natural generalisations of the conventions for parentheses as in Notations 2.2.7 and 2.3.2. This way, we shall write* $\lambda\alpha : *.M : \Pi\alpha : *.\sigma$ *instead of* $(\lambda\alpha : *.M : \Pi\alpha : *.\sigma)$.

Example 3.1.4. The types of $\lambda\alpha : *.\lambda x : \alpha.x$ and $\lambda\alpha : *.\lambda\beta : *.\lambda x : \alpha \to \beta.\lambda y : \alpha.xy$ are, respectively, $\Pi\alpha : *.\alpha \to \alpha$ and $\Pi\alpha : *.\Pi\beta : *.(\alpha \to \beta) \to (\alpha \to \beta)$.

We proceed to redefine the notions of *statement*, *declaration* and *context* according to our new scene.

Definition 3.1.5.

i) A *statement* is either of the form $M : \sigma$, where M is a term and σ is a type; or $\sigma : *$, where σ is a type. The *subject* of a statement is the expression before the ':'.

ii) A *declaration* is a statement with a term variable or a type variable as subject.

iii) A $\lambda 2$-*context* is a list of declarations with different subjects such that every type that appears has been declared priorly in the list order. That is, if some declaration $x : \alpha$ occurs in Γ, then $\alpha : *$ must have appeared before in Γ.

As we did for the system $\lambda\to$, we provide a set of derivation rules to define our calculus $\lambda 2$. We start by rewriting the rules from $\lambda\to$:

$$\frac{}{\Gamma \vdash x : \sigma}\ \text{var} \quad \text{if } \Gamma \text{ is a } \lambda 2\text{-context and } x : \sigma \in \Gamma$$

$$\frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}\ \text{appl}$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma.M : \sigma \to \tau}\ \text{abst}$$

Note that we have added the condition that Γ must be a $\lambda 2$-context in the var axiom. The new rules regarding the second order aspects are:

$$\frac{}{\Gamma \vdash \sigma : *}\ \text{form} \quad \text{if } \Gamma, \sigma : * \text{ is a } \lambda 2\text{-context}$$

$$\frac{\Gamma \vdash M : (\Pi \alpha : *.\sigma) \qquad \Gamma \vdash N : *}{\Gamma \vdash MN : \sigma[\alpha := N]}\ \text{appl}_2$$

$$\frac{\Gamma, \alpha : * \vdash M : \sigma}{\Gamma \vdash \lambda \alpha : *.M : \Pi \alpha : *.\sigma}\ \text{abst}_2$$

Note that we are using the substitution we defined back in Chapter 2. In fact, we can use it in any of the systems we present, keeping in mind that the terms are different in each case.

Example 3.1.6.

1- We show that $\vdash \left(\lambda \alpha : *.\lambda x : \alpha.x\right) : \Pi \alpha : *.\alpha \to \alpha$ is a $\lambda 2$-theore*m*.

$$\frac{\dfrac{\dfrac{}{\alpha : *, x : \alpha \vdash x : \alpha}\ \text{var}}{\alpha : * \vdash \lambda x : \alpha.x : \alpha \to \alpha}\ \text{abst}}{\vdash \left(\lambda \alpha : *.\lambda x : \alpha.x\right) : \Pi \alpha : *.\alpha \to \alpha}\ \text{abst}_2$$

2- From the last example, we can derive $\vdash (\lambda \alpha : *.\lambda x : \alpha.x)(\beta : *) : \beta \to \beta$:

$$\frac{\dfrac{}{\vdash \left(\lambda \alpha : *.\lambda x : \alpha.x\right) : \Pi \alpha : *.\alpha \to \alpha} \qquad \dfrac{}{\vdash \beta : *}\ \text{form}}{\vdash (\lambda \alpha : *.\lambda x : \alpha.x)(\beta : *) : \beta \to \beta}\ \text{appl}_2$$

This was our purpose when we defined a second order identity function: when we feed it with some concrete type, it returns the correspondent identity function for that type.

3.2 Types depending on types: $\lambda \underline{\omega}$

After having generalised and obtained *terms depending on types*, it is only natural to think about *types depending on types*. Adding these to the system $\lambda \to$ results in what we will call $\lambda \underline{\omega}$. Although the constant $*$ is going to make an appearance in this section (and it is defined in the same way), remember that we are not extending $\lambda 2$, but our initial $\lambda \to$.

Consider the types $\sigma \to \sigma$, $\tau \to \tau$ and $(\gamma \to \beta) \to (\gamma \to \beta)$. They all have the form $A \to A$. We would like to have a type that, if given an input A, it outputs the type $A \to A$.

We denote this by $\lambda \alpha : *.\alpha \to \alpha$ and we say it is a *type constructor*. More examples of type constructors are $\lambda \alpha : *.\alpha$ and $\lambda \alpha : *.\lambda \beta : *.\alpha \to (\beta \to \alpha)$. When a type constructor receives a type as input, we obtain a type as desired (via the β-reduction): $(\lambda \alpha : *.\alpha \to \alpha)\gamma \to_\beta \gamma \to \gamma$.

We need to define new types, called *kinds*, to be able to express the types of the type constructors.

Definition 3.2.1. A *kind* is recursively defined as:

i) $*$ is a kind.

ii) If κ_1 and κ_2 are kinds, then $(\kappa_1 \rightarrow \kappa_2)$ is a kind.

Example 3.2.2.

1- The expression $\alpha \rightarrow \alpha$ has type $*$, since $\alpha : *$. So the type constructor $\lambda\alpha : *.\alpha \rightarrow \alpha$ has type $* \rightarrow *$.

2- Given that $\alpha, \beta : *$, the expression $\alpha \rightarrow (\beta \rightarrow \alpha)$ has type $*$. Thus, the type constructor $\lambda\alpha : *.\lambda\beta : *.\alpha \rightarrow (\beta \rightarrow \alpha)$ has type $* \rightarrow (* \rightarrow *)$.

Including kinds in our definition of types, let us have terms like $\lambda\alpha : * \rightarrow *.\alpha\beta$, which has type $(* \rightarrow *) \rightarrow *$ if $\beta : *$.

We finish this round of definitions, before starting with the calculus rules, with another supertype $\Box$, which plays a similar role for kinds that $*$ played for regular types.

Definition 3.2.3.

i) The type $\Box$ is the *type of all kinds*, that is, for all kind κ, we have that $\kappa : \Box$.

ii) A *sort* is either $*$ or $\Box$. We denote sorts by the letter s.

Note that none of the types in the previous sections has type $\Box$, except for $*$.

Definition 3.2.4. (Types)
Types in $\lambda\underline{\omega}$ are defined recursively as:

i) If $\alpha \in \mathbb{V}$, then α is a type.

ii) The constants $*$ and $\Box$ are types.

iii) If σ, τ are types, then $(\sigma \rightarrow \tau)$ is a type.

Definition 3.2.5. (Terms)
Terms in $\lambda\underline{\omega}$ are defined recursively as:

i) If σ_1 and σ_2 are types, then $\sigma_1 : \sigma_2$ is a term.

ii) If $x \in V$ and σ is a type, then $(x : \sigma)$ is a term.

iii) If $(M : \sigma \rightarrow \tau)$ and $(N : \sigma)$ are terms and, then $(MN : \tau)$ is a term.

iv) If $(M : \sigma)$ is a term, x is a term variable and α is a type, then $(\lambda x : \alpha.M : \alpha \rightarrow \sigma)$ is a term.

v) If $(M : \sigma)$ is a term and α is a type variable, then $(\lambda\alpha : *.M : * \rightarrow \sigma)$ is a term.

And, finally, as it has become customary, we define the system $\lambda\underline{\omega}$, which is an extension of $\lambda\,{\to}$, with the following set of rules.

$$\frac{}{\emptyset \vdash * : \Box}\ \text{sort}$$

$$\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}\ \text{var}\quad \text{if } x \notin \Gamma$$

$$\frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B}\ \text{weak}\quad \text{if } x \notin \Gamma$$

$$\frac{\Gamma \vdash A : s \qquad \Gamma \vdash B : s}{\Gamma \vdash A \to B : s}\ \text{form}$$

$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}\ \text{appl}$$

$$\frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash A \to B : s}{\Gamma \vdash \lambda x : A.M : A \to B}\ \text{abst}$$

$$\frac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s}{\Gamma \vdash A : B'}\ \text{conv}\quad \text{if } B =_\beta B'$$

The sort axiom is a way of ensuring that the supertype $*$ has type $\Box$. This together with the rest of the rules make it possible to derive a proof of $\vdash \kappa : \Box$, for all kind κ.

Note that we are using the relation $=_\beta$ in the conv rule. This is the same $=_\beta$ we defined in Chapter 2, but using the appropriate terms.

Example 3.2.6. We illustrate some of the rules of $\lambda\underline{\omega}$ by showing the derivation of $\beta : * \vdash (\lambda\alpha : *.\alpha \to \alpha)\beta : *$.

First, we derive $\beta : * \vdash * : \Box$ in the following way:

$$\frac{\dfrac{}{\vdash * : \Box}\ \text{sort} \qquad \dfrac{}{\vdash * : \Box}\ \text{sort}}{\beta : * \vdash * : \Box}\ \text{weak}$$

And we use this short result to *p*rove $\beta : * \vdash (\lambda\alpha : *.\alpha \to \alpha) : * \to *$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\beta:*\vdash *:\Box}}{\beta:*,\alpha:*\vdash\alpha:*}\text{var}\quad\dfrac{\dfrac{}{\beta:*\vdash *:\Box}}{\beta:*,\alpha:*\vdash\alpha:*}\text{var}}{\beta:*,\alpha:*\vdash\alpha\to\alpha:*}\text{form}\quad\dfrac{\dfrac{}{\beta:*\vdash *:\Box}\quad\dfrac{}{\beta:*\vdash *:\Box}}{\beta:*\vdash *\to *:\Box}\text{form}}{\beta:*\vdash(\lambda\alpha:*.\alpha\to\alpha):*\to *}\text{abst}$$

Finally, we show $\beta:*\vdash(\lambda\alpha:*.\alpha\to\alpha)\beta:*$, finishing our example:

$$\dfrac{\dfrac{}{\beta:*\vdash(\lambda\alpha:*.\alpha\to\alpha):*\to *}\quad\dfrac{\dfrac{\dfrac{}{\vdash *:\Box}\text{sort}}{\beta:*\vdash\beta:*}\text{var}}{\beta:*\vdash(\lambda\alpha:*.\alpha\to\alpha)\beta:*}\text{appl}$$

Note that most of the rules of $\lambda\underline{\omega}$ have a dual nature, since we do not specify the sort s. For example, let us look at the var rule:

$$\dfrac{\Gamma\vdash A:s}{\Gamma,x:A\vdash x:A}\text{ var }\quad\text{if } x\notin\Gamma$$

This is a compact way to express two different var rules:

$$\dfrac{\Gamma\vdash A:*}{\Gamma,x:A\vdash x:A}\text{ var1 }\quad\text{if } x\notin\Gamma$$

$$\dfrac{\Gamma\vdash A:\Box}{\Gamma,x:A\vdash x:A}\text{ var2 }\quad\text{if } x\notin\Gamma$$

In our last example, both times we used the var rule, we actually used var2.

3.3 Types depending on terms: λP

We are left with only one more possible extension of $\lambda\to$, the one that speaks about *types depending on terms*, λP. We will briefly mention it in this last section of the chapter before moving on to the calculus of inductive constructions, which is the common extension of $\lambda 2$, $\lambda\underline{\omega}$ and λP, the three systems that appear in this chapter. As before, remember that even if λP shares some similarities with $\lambda 2$ and $\lambda\underline{\omega}$, they are all independent extensions of $\lambda\to$ (i.e. none is an extension of another).

We shall use type constructors of the form $\lambda x:A.M$, where M is a type, x is a term variable and A is a type. For example, $\lambda x:\alpha.\beta x$ is a type that depends on the term x.

This extension is important because it has many applications in mathematics and logic. For instance, denoting the type of natural numbers by *nat* and the set of numbers below n by S_n (which is also a type), the expression $\lambda n:nat.S_n$ refers to the function that takes a natural number and outputs the set S_n. The type of this function is $nat\to *$, since $S_n:*$. And why is it

true that $S_n : *$? You caught us, it is merely a convention to give sets the type $*$. Do not think too much about this now as we will explore this further in the next chapter.

The definitions of types and terms are analogous to the ones we have already seen. This time, we add the possibility of terms of the form $\lambda x : A.M : \Pi x : A.B$, where x is a term variable, A and B, types and M is a term.

The rules of the calculus λP are quite similar to the ones we gave in the previous section, but with some modifications:

$$\frac{}{\emptyset \vdash * : \Box}\ \text{sort}$$

$$\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}\ \text{var}\quad \text{if } x \notin \Gamma$$

$$\frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B}\ \text{weak}\quad \text{if } x \notin \Gamma$$

$$\frac{\Gamma \vdash A : * \qquad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Pi x : A.B : s}\ \text{form}$$

$$\frac{\Gamma \vdash M : \Pi x : A.B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B[x := N]}\ \text{appl}$$

$$\frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash \Pi x : A.B : s}{\Gamma \vdash \lambda x : A.M : \Pi x : A.B}\ \text{abst}$$

$$\frac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s}{\Gamma \vdash A : B'}\ \text{conv}\quad \text{if } B =_\beta B'$$

Again, we use substitutions and $=_\beta$ as in Chapter 2, but for our current terms.

Note that in the form, appl and abst rules, instead of writing $A \to B$, we return to the $\Pi x : A.B$ from the second order lambda calculus. We do so because the variable x may occur freely in B. Thus, the type of the output is not fixed, but changes depending on x.

Example 3.3.1. We show a derivation of $S : * \vdash \left(\Pi x : S.\Pi y : S. *\right) : \Box$ in λP.

We first prove two short derivations: $S : *, x : S \vdash S : *$ and $S : *, x : S \vdash * : \Box$:

$$\dfrac{\dfrac{\dfrac{}{\vdash * : \Box}\ \text{sort}}{S : * \vdash S : *}\ \text{var} \qquad \dfrac{\dfrac{}{\vdash * : \Box}\ \text{sort}}{S : * \vdash S : *}\ \text{var}}{S : *, x : S \vdash S : *}\ \text{weak}$$

$$\dfrac{\dfrac{\dfrac{}{\vdash * : \Box}\,\text{sort} \quad \dfrac{}{\vdash * : \Box}\,\text{sort}}{S : * \vdash * : \Box}\,\text{weak} \quad \dfrac{\dfrac{}{\vdash * : \Box}\,\text{sort}}{S : * \vdash S : *}\,\text{var}}{S : *, x : S \vdash * : \Box}\,\text{weak}$$

And we use this results to show that $S : * \vdash \left(\Pi x : S.\Pi y : S.*\right) : \Box$ in λP:

$$\dfrac{\dfrac{\dfrac{}{\vdash * : \Box}\,\text{sort}}{S : * \vdash S : *}\,\text{var} \quad \dfrac{S : *, x : S \vdash S : *}{\dfrac{S : *, x : S \vdash * : \Box \qquad S : *, x : S \vdash S : *}{S : *, x : S, y : S \vdash * : \Box}\,\text{weak}}{S : *, x : S \vdash \Pi y : S.* : \Box}\,\text{form}}{S : * \vdash \left(\Pi x : S.\Pi y : S.*\right) : \Box}\,\text{form}$$

Chapter 4

Putting all together: the calculus of inductive constructions

4.1 The calculus λC

It is time for us to combine the three systems $\lambda 2$, $\lambda\underline{\omega}$ and λP to create an extension of $\lambda \rightarrow$ that deals with terms that depend on types, types that depend on types and types that depend on terms. Note that the remaining possibility, terms that depend on terms, is basically what $\lambda \rightarrow$ does. This new system is called λC, *λ-cube* or *calculus of inductive constructions* (CoIC). As a warning note, we should remark that the λ-cube and the CoIC are not exactly the same: the λ-cube is the framework in which all $\lambda \rightarrow$, $\lambda 2$, $\lambda\underline{\omega}$, λP and λC (i.e., the CoIC) are in. It is usually displayed as a cube in which the three directional vectors correspond to the three extensions of $\lambda \rightarrow$. However, we take the liberty of dealing with them as synonyms. May the reader forgive us for this license.

Let us be slightly more formal than in the previous chapter and state the key definitions:

Definition 4.1.1. The set of *expressions* in λC is recursively defined as:

i) If $x \in V$, i.e., x is a variable, then x is an expression.

ii) $*$ and $\Box$ are expressions.

iii) If e_1 and e_2 are expressions, then $e_1 e_2$, $\lambda V : e_1.e_2$ and $\Pi V : e_1.e_2$ are expressions.

Notation 4.1.2. *We shall write $A \rightarrow B$ for $\Pi x : A.B$, whenever x does not occur freely in B. This is usually done in some of the systems presented in the previous chapter, but, for simplicity, we omitted this notation until now.*

The rules for the calculus λC are:

$$\frac{}{\emptyset \vdash * : \Box}\ \text{sort}$$

$$\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}\ \text{var}\quad \text{if } x :\notin \Gamma$$

$$\frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B} \text{ weak} \quad \text{if } x :\notin \Gamma$$

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash A \to B : s_2} \text{ form}$$

$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B[x := N]} \text{ appl}$$

$$\frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash A \to B : s}{\Gamma \vdash \lambda x : A.M : A \to B} \text{ abst}$$

$$\frac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s}{\Gamma \vdash A : B'} \text{ conv} \quad \text{if } B =_\beta B'$$

Note that the rules are almost identical to the ones we provided for $\lambda\underline{\omega}$ and λP, with the notable exception of the form rule. In $\lambda\underline{\omega}$, the form rule was:

$$\frac{\Gamma \vdash A : s \qquad \Gamma \vdash B : s}{\Gamma \vdash A \to B : s}\ \lambda\underline{\omega} \text{ form}$$

The two sorts had to be equal, in order to avoid having mixed expressions (like types depending on terms). On the other hand, in λP:

$$\frac{\Gamma \vdash A : * \qquad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Pi x : A.B : s}\ \lambda P \text{ form}$$

Here, we did not allow A to have type $\Box$ in order to be sure that we did not end up with types depending on types when performing $\Gamma \vdash \Pi x : A.B : s$. In λC, however, as we do not have to make these distinctions, we allow A to have any sort, independent of B's.

Example 4.1.3. We show that $S : *, P : S \to * \vdash \lambda x : S.(Px \to \Pi\alpha : *.\alpha) : S \to *$ is derivable in λC. First, note that we can derive $S : *, P : S \to * \vdash S : *$ and $S : *, P : S \to * \vdash * : \Box$ by applying the weak rule a couple of times (and the sort an*d* var rules in the upper parts of the proof).

Using this, we can show that $S : *, P : S \to * \vdash S \to * : \Box$. We label this proof as Π_1.

$$\dfrac{\dfrac{}{S : *, P : S \to * \vdash S : *} \qquad \dfrac{\dfrac{}{S : *, P : S \to * \vdash * : \Box} \qquad \dfrac{}{S : *, P : S \to * \vdash S : *}}{S : *, P : S \to *, x : S \vdash * : \Box}\ \text{weak}}{S : *, P : S \to * \vdash S \to * : \Box}\ \text{form}$$

On the other hand, we can prove that $S:*, P:S\to *, x:S, y:Px \vdash *:\Box$ by applying repeatedly the weak rule (plus the var and sort), assuming we have a proof of $S:*, P:S\to *, x:S \vdash Px:*$. But this fact can be derived by abstraction from $S:*, P:S\to *, x:S \vdash P:S\to *$ and $S:*, P:S\to *, x:S \vdash x:S$, both of which follow from using the weak, sort and var rules. Using the same idea, we can also show that $S:*, P:S\to *, x:S, y:Px, \alpha:* \vdash \alpha:*$. From the two proofs, we get that $S:*, P:S\to *, x:S, y:Px \vdash \Pi\alpha:*.\alpha:*$. We label this proof as Π_2:

$$\frac{S:*, P:S\to *, x:S, y:Px \vdash *:\Box \qquad S:*, P:S\to *, x:S, y:Px, \alpha:* \vdash \alpha:*}{S:*, P:S\to *, x:S, y:Px \vdash \Pi\alpha:*.\alpha:*}\ \text{form}$$

Moreover, using Π_2 and the proof of $S:*, P:S\to *, x:S \vdash Px:*$, we can derive $S:*, P:S\to *, x:S \vdash Px\to \Pi\alpha:*.\alpha:*$. We label this proof as Π_3.

$$\frac{S:*, P:S\to *, x:S \vdash Px:* \qquad \dfrac{\Pi_2}{S:*, P:S\to *, x:S, y:Px \vdash \Pi\alpha:*.\alpha:*}}{S:*, P:S\to *, x:S \vdash Px\to \Pi\alpha:*.\alpha:*}\ \text{form}$$

Finally, we put Π_1 and Π_3 together and conclude our proof.

$$\frac{\dfrac{\Pi_3}{S:*, P:S\to *, x:S \vdash Px\to \Pi\alpha:*.\alpha:*} \qquad \dfrac{\Pi_1}{S:*, P:S\to * \vdash S\to *:\Box}}{S:*, P:S\to * \vdash \lambda x:S.(Px\to \Pi\alpha:*.\alpha):S\to *}\ \text{abst}$$

4.2 Some results about λC

As before, a context is a list of declarations $e_1:e_1', e_2:e_2', \ldots$ with different subjects.

Definition 4.2.1. A context Γ is *well-formed* if there exist expressions A and B such that $\Gamma \vdash A:B$.

Proposition 4.2.2. *(Thinning)*
Let $\Gamma \subseteq \Gamma'$ be two well-formed contexts. If $\Gamma \vdash A:B$, then $\Gamma' \vdash A:B$.

Proof. We show this result by induction on the proof $\Gamma \vdash A:B$.

- If it is the result of applying a sort rule, i.e., $\Gamma = \emptyset$, $A \equiv *$ and $B \equiv \Box$, then we can show that $\Gamma' \vdash *:\Box$ by using repeatedly the weak rule. Each time we do so, we can find the proper second premise of the rule because of the fact that Γ' is well-formed. This should be proved by another induction, but an example suffices: we show that $S:*, P:*\to S \vdash *:\Box$. We first prove that $S:*, x:* \vdash S:*$:

$$\dfrac{\dfrac{\dfrac{}{\vdash * : \Box}\,\text{sort}}{S : * \vdash S : *}\,\text{var} \qquad \dfrac{\dfrac{\dfrac{}{\vdash * : \Box}\,\text{sort} \quad \dfrac{}{\vdash * : \Box}\,\text{sort}}{S : * \vdash * : \Box}\,\text{weak}}{S : *, x : * \vdash S : *}\,\text{weak}$$

And, finally,

$$\dfrac{\dfrac{\dfrac{}{\vdash * : \Box}\,\text{sort} \quad \dfrac{}{\vdash * : \Box}\,\text{sort}}{S : * \vdash * : \Box}\,\text{weak} \qquad \dfrac{\dfrac{\dfrac{}{\vdash * : \Box}\,\text{sort} \quad \dfrac{}{\vdash * : \Box}\,\text{sort}}{S : * \vdash * : \Box}\,\text{weak} \qquad \dfrac{}{S : *, x : * \vdash S : *}}{S : * \vdash * \to S : \Box}\,\text{form}}{S : *, P : * \to S \vdash * : \Box}\,\text{weak}$$

We see that the order in which we use the weak rule is important. If we had started by eliminating $S : *$ instead of $P : * \to S$, we would have not been able to show that $\vdash * \to S : \Box$. This idea is the key to the proof that $\Gamma' \vdash * : \Box$ for a general well-formed Γ'.

- If it is the result of applying a var rule, i.e. $\Delta, x : A \vdash x : A$ if $x \notin \Delta$, then for showing $\Delta, x : A, \Delta' \vdash x : A$, we must get rid of all the appearances of x in Δ'. For doing so, we perform as before, repeatedly using the weak rule. Once we get some Δ'' in which x does not appear, we apply the var rule and the induction hypothesis and end our proof.

- The rest of the rules are analogous.

□

The next result shows that the order of the declarations in a context makes zero difference.

Proposition 4.2.3. *(Permutation)*
Let Γ and Γ' be two well-formed contexts such that Γ' is a permutation of Γ. If $\Gamma \vdash A : B$, then $\Gamma' \vdash A : B$.

Proof. As in the last proposition, we show the result by induction on the proof of $\Gamma \vdash A : B$.

- If it is the result of applying the sort rule, then $\Gamma = \emptyset$, $A \equiv *$ and $B \equiv \Box$. Γ' must be also empty and the result follows trivially.

- If it is the result of applying the form rule, then $\Gamma \vdash A' \to B' : s_2$. By induction hypothesis, both $\Gamma' \vdash A : s_1$ and $\Gamma', x : A' \vdash B' : s_2$ hold. Thus, by the weak rule, $\Gamma' \vdash A' \to B' : s_2$.

- The rest of the rules are analogous.

□

Proposition 4.2.4. *(Condensing)*
If $\Gamma, x : A, \Gamma' \vdash B : C$ *and* x *does not occur in* Γ', B *or* C, *then* $\Gamma, \Gamma' \vdash B : C$.

In λC, there is a Generation Lemma, similar to the one for $\lambda \rightarrow$, that somehow establishes an equivalence between the premises in the rules of the calculus and the conclusions. Using this result, one can prove the *uniqueness of types up to conversion*. Note that in the following theorem we are using substitutions and the relation $=_\beta$ as in Chapter 2, but for the terms in λC.

Theorem 4.2.5.

i) If $\Gamma \vdash x : A$, *then there are a sort* s *and an expression* B *such that* $B =_\beta A$, $x : B \in \Gamma$ *and* $\Gamma \vdash B : s$.

ii) If $\Gamma \vdash A_1 A_2 : C$, *then there are expressions* A *and* B *such that* $\Gamma \vdash A_1 : \Pi x : A.B$, $\Gamma \vdash A_2 : A$ *and* $C =_\beta B[x := A_2]$.

iii) If $\Gamma \vdash \lambda x : A_1.A_2 : C$, *then there are a sort* s *and an expression* B *such that* $C =_\beta \Pi x : A_1.B$, $\Gamma \vdash \Pi x : A_1.B : s$ *and* $\Gamma, x : A_1 \vdash A_2 : B$.

iv) If $\Gamma \vdash \Pi : A_1.A_2 : B$, *then there are sorts* s_1, s_2 *such that* $B \equiv s_2$, $\Gamma \vdash A_1 : s_1$ *and* $\Gamma, x : A_1 \vdash A_2 : s_2$.

Corollary 4.2.6. *(Uniqueness of types)*
If $\Gamma \vdash A : B$ *and* $\Gamma \vdash A : C$, *then* $B =_\beta C$.

Proof. The result can be proved by induction on the complexity of A.

- If $A \equiv x$, then by the Generation Lemma, there are sorts s_1, s_2 and expressions B' and C' such that $B' \equiv_\beta B$, $C =_\beta C'$, $\Gamma \vdash B' : s_1$, $\Gamma \vdash C' : s_2$ and $x : B', x : C' \in \Gamma$. By the definition of context, we deduce that $B' \equiv C'$. Since the relation $=_\beta$ is transitive, $B =_\beta C$.

- If $A \equiv A_1 A_2$, then by the Generation Lemma, there are expressions M, N, M' and N' such that $\Gamma \vdash A_1 : \Pi x : M.N$, $\Gamma \vdash A_1 : \Pi x : M'.N'$, $\Gamma \vdash A_2 : M$, $\Gamma \vdash A_2 : M'$, $B =_\beta N[x := A_2]$ and $C =_\beta N'[x := A_2]$. By the induction hypothesis, we get that $M =_\beta M'$ and, thus, $N =_\beta N'$. Finally, by transitivity, we conclude that $B =_\beta C$.

- The rest of the cases are analogous.

□

The following is an useful lemma that can be used for simplifying the expressions via a substitution.

Lemma 4.2.7. *(Substitution lemma)*
If $\Gamma, x : A, \Gamma' \vdash B : C$ *and* $\Gamma \vdash D : A$, *then* $\Gamma, \Gamma'[x := D] \vdash B[x := D] : C[x := D]$.

In λC, we also have a Confluence theorem. Recall that $\twoheadrightarrow_\beta$ was defined in Definition 2.2.18, but here we use the correct expressions in λC.

Theorem 4.2.8. *(Confluence)*
If $A \twoheadrightarrow_\beta B_1$ *and* $A \twoheadrightarrow_\beta B_2$, *then there exists some* C *such that* $B_i \twoheadrightarrow_\beta C$ *for* $i = 1, 2$.

4.3 Decidability of three problems

There are three main kinds of problems that a logician can come across in any of the systems we have presented: $\lambda\rightarrow$, $\lambda 2$, $\lambda\underline{\omega}$, λP and λC.

1- (Well-typedness) Given a term M, find a context Γ and a type σ such that $\Gamma \vdash M : \sigma$. This problem is usually referred to as $? \vdash M :?$. A particular case of this is when a context Γ is already given and we only want to find the type σ.

2- (Type Checking) Given a context Γ, a term M and a type σ, we prove that $\Gamma \vdash M : \sigma$. Most of the examples that we solved are of this type.

3- (Term Finding) Given a context Γ and a type σ, we find a term M such that $\Gamma \vdash M : \sigma$. This problem is usually referred to as $\Gamma \vdash ? : \sigma$.

It should not come as a surprise that all of these three problems are decidable in $\lambda\rightarrow$ because, by construction, it is easy to select the correct rule each time.

Let us solve a Well-typedness problem in $\lambda\rightarrow$ as an example: $?_1 \vdash \big(\lambda x : (\alpha \rightarrow \beta) \rightarrow \beta . x(yz)\big) :?_2$. The only rule it can come from is the abst rule: the second question mark must be of the form $((\alpha \rightarrow \beta) \rightarrow \beta) \rightarrow ?'_2$ and we get the hypothesis $?_1, x : (\alpha \rightarrow \beta) \rightarrow \beta \vdash x(yz) :?'_2$. Now, it must be the result of applying the appl rule: $?_1, x : (\alpha \rightarrow \beta) \rightarrow \beta \vdash x :?_3 \rightarrow ?'_2$ and $?_1, x : (\alpha \rightarrow \beta) \rightarrow \beta \vdash yz :?_3$. If we consider the first one, it is clear that it is the result of a var rule and that $?_3$ must be $\alpha \rightarrow \beta$ and $?'_2$, β. Thus, we have to solve $?_1, x : (\alpha \rightarrow \beta) \rightarrow \beta \vdash yz : \alpha \rightarrow \beta$. This is, again, a result of an appl rule: $?_1, x : (\alpha \rightarrow \beta) \rightarrow \beta \vdash y :?_4 \rightarrow (\alpha \rightarrow \beta)$ and $?_1, x : (\alpha \rightarrow \beta) \rightarrow \beta \vdash z :?_4$. Both come from the var rule. Giving an arbitrary type to the variable z, we find the solution to the problem: $z : \gamma, y : \gamma \rightarrow (\alpha \rightarrow \beta) \vdash \big(\lambda x : (\alpha \rightarrow \beta) \rightarrow \beta . x(yz)\big) : ((\alpha \rightarrow \beta) \rightarrow \beta) \rightarrow \beta$.

Type Checking follows a similar procedure, and it is also decidable. Moreover, both Well-typedness and Type-checking are not only decidable in $\lambda\rightarrow$, but in all of our systems, including λC. We take these results from [Barendregt].

Theorem 4.3.1. *(Curry)*
Let λX be one of $\lambda\rightarrow$, $\lambda 2$, $\lambda\underline{\omega}$, λP and λC. Then:

i) Well-typedness is decidable for λX.

ii) If a term M is typable (i.e. there exists a type σ and a context Γ such that $\Gamma \vdash M : \sigma$), then M has a type σ', called the principal type, such that every possible type for M is a substitution instance of σ' and σ' is computable from M.

Corollary 4.3.2. *Let λX be one of $\lambda\rightarrow$, $\lambda 2$, $\lambda\underline{\omega}$, λP and λC. Then, Type Checking is decidable for λX.*

Proof. Let Γ be a context, M a term and σ a type. For checking whether $\Gamma \vdash M : \sigma$, it is enough to verify that M is well-typed (which is a decidable process, by the previous theorem) and that σ is a substitution instance of its principal type.

□

Term Finding, however, is a different matter. Because of the simplicity of the system, it is decidable in $\lambda\,{\to}$. For example, let us solve $\vdash ? : ((\alpha \to \alpha) \to \gamma) \to (\alpha \to (\beta \to \gamma))$. It clearly must come from the abst rule: $x : (\alpha \to \alpha) \to \gamma \vdash ?_1 : \alpha \to (\beta \to \gamma)$. Again, we must apply the abst rule: $x(\alpha \to \alpha) \to \gamma, y : \alpha \vdash ?_2 : \beta \to \gamma$. A third use of the abst rule gives us: $x(\alpha \to \alpha) \to \gamma, y : \alpha, z : \beta \vdash ?_3 : \gamma$. So far the algorithm is similar to previous ones but here is where it differs. We cannot keep going 'up' using the rules. From the variables x, y and z we have to construct a term of type γ. We observe that γ only appears in x, so we guess that we need an appl rule using a term of type $\alpha \to \alpha$. It is easy to obtain such a term, $\lambda u : \alpha.y$ (note that $\lambda u : \alpha.u$ also works). Thus, $x(\alpha \to \alpha) \to \gamma, y : \alpha, z : \beta \vdash x(\lambda u : \alpha.y) : \gamma$ and the solution to our initial problem is $\vdash \big(\lambda x : (\alpha \to \alpha) \to \gamma, y : \alpha, z : \beta.x(\lambda u : \alpha.y)\big) : ((\alpha \to \alpha) \to \gamma) \to (\alpha \to (\beta \to \gamma))$.

As we see there is no general algorithm for this, we have to make guesses at some stage. This is why, in more complex systems, Term Finding is not decidable. Only in $\lambda\,{\to}$ and $\lambda\underline{\omega}$ it is decidable. In the next section, we will present a way for encoding logic in λC, interpreting types as formulas and terms as proofs. If Term Finding were decidable, it would imply that there is an algorithm for proving or disproving any first-order logical proposition, which we sadly know is impossible.

4.4 Encoding first-order logic in type theory

As we advanced at the end of the previous section, it is possible to encode the main logical connectives using tools in lambda calculus. This implies that type theory is powerful enough to be able to write proofs in first-order logic.

For this, we interpret propositions as types and proofs as terms. In other words, the expression $a : A$ means that a is a proof of the proposition A. We can also interpret sets as types: for example, 3 has type nat, because $3 \in \mathbb{N}$.

The following is a list of the codings of the important elements in logic:

- Implication: for coding the logical implication, we can use the arrow $\to$ because it fulfills the i*m*plication rules $\to$ elim and $\to$ intro:

$$\frac{A, B : *, \Gamma \vdash c : A \to B \qquad A, B : *, \Gamma \vdash a : A}{A, B : *, \Gamma \vdash ca : B} \to \text{elim}$$

We see that the only rule we need to use is the appl rule. For the introduction rule:

$$\frac{A, B : *, \Gamma, x : A \vdash b : B \qquad A, B : *, \Gamma \vdash A \to B : s}{A, B : *, \Gamma \vdash \lambda x : A.b : A \to B} \to \text{intro}$$

This time, it is the abst rule that we are using.

- Falsehood: if there exists a proof of $\bot$, then there exist a proof of anything. Coding it by $\Pi\alpha : *.\alpha$, we get this property (efq rule) as an instance of the appl rule:

$$\frac{A : *, \Gamma \vdash f : \Pi\alpha : *.\alpha \qquad A : *, \Gamma \vdash A : *}{A : *, \Gamma \vdash fA : A}\ \text{efq}$$

- Negation: once we have $\bot$ and implication, it is easy to code $\neg A$ as $A \to \bot$, that is, $A \to \Pi\alpha : *.\alpha$.

- Conjunction: for the conjunction, we are tempted to use the logical equivalence $A \wedge B \equiv \neg(A \to \neg B)$. But this equivalence only holds in classical logic, and we want to have an intuitionistic logic so computers can work with it. Because of this, we need the more general expression $\Pi C : *.(A \to B \to C) \to C$. Intuitively, it can be read as "for all C, if A and B imply C, then C holds on its own". In other words, the condition "A and B" is superfluous, it must be the case that A and B always hold. Here, the introduction and elimination rules for the conjunction hold as well. For example:

$$\frac{A, B : *, \Gamma \vdash a : A \qquad A, B : *, \Gamma \vdash b : B}{A, B : *, \Gamma \vdash \big(\lambda C : *.(\lambda x : (A \to B \to C).xab)\big) : \Pi C : *.(A \to B \to C) \to C}\ \wedge\ \text{intro}$$

The proof goes as follows: it is easy to write a proof of $A : *, B : *, \Gamma \vdash * \to ((A \to B \to C) \to C) : *$ by repeating the form and weak rules and ending each branch with the sort rule. The same happens for $A : *, B : *, \Gamma, C : * \vdash (A \to B \to C) \to C : *$. Thus, if we abbreviate $A \to B \to C$ by P, we start the proof by:

$$\frac{A, B : *, \Gamma, C : *, x : P \vdash xab : C \qquad A, B : *, \Gamma, C : * \vdash P \to C : *}{A, B : *, \Gamma, C : * \vdash \lambda x : P.xab : P \to C}\ \text{abst}$$

$$\frac{A, B : *, \Gamma, C : * \vdash \lambda x : P.xab : P \to C \qquad A, B : *, \Gamma \vdash * \to (P \to C) : *}{A, B : *, \Gamma \vdash \big(\lambda C : *.(\lambda x : P.xab)\big) : \Pi C : *.P \to C}\ \text{abst}$$

We are left to show $A, B : *, \Gamma, C : *, x : A \to B \to C \vdash xab : C$. From the hypothesis $A, B : *, \Gamma \vdash b : B$, we can derive $A, B : *, \Gamma, C : *, x : A \to B \to C \vdash b : B$. Analogously, $A, B : *, \Gamma, C : *, x : A \to B \to C \vdash a : A$. And, clearly, $A, B : *, \Gamma, C : *, x : A \to B \to C \vdash x : A \to B \to C$.

$$\frac{A, B : *, \Gamma, C : *, x : P \vdash x : P \qquad A, B : *, \Gamma, C : *, x : P \vdash a : A}{A, B : *, \Gamma, C : *, x : P \vdash xa : B \to C}\ \text{appl}$$

$$\frac{A, B : *, \Gamma, C : *, x : P \vdash xa : B \to C \qquad A, B : *, \Gamma, C : *, x : P \vdash b : B}{A, B : *, \Gamma, C : *, x : P \vdash xab : C}\ \text{appl}$$

- Disjunction: similar to the *p*revious case, we cannot use the equivalence $A \vee B \equiv \neg A \to B$. We code it instead in the form $\Pi C : *.(A \to C) \to (B \to C) \to C$. The reader can check that this construction works for the respective introduction and elimination rules.

- Universal quantifier: we write $\Pi x : S.Px$ for $\forall x \in S(P(x))$. Similarly to the previous cases, the universal rules also hold for this expression. We leave the details to the reader.

- Existential quantifier: again, one may write $\exists \equiv \neg\forall\neg$, but this only works in classical logic. We write instead $\Pi\alpha : *.((\Pi x : S.(Px \to \alpha)) \to \alpha)$ for $\exists x \in S(P(x))$. The existential rules can be derived in λC as well.

Remark 4.4.1. *In fact, one can already code the implication and the universal quantifier in λP, but the rest of connectives only appear in λC.*

Example 4.4.2. We derive in λC the tautology $A \to (B \to A)$. It is easy to check that $A, B : *, x : A \vdash B \to A : *$: this is an instance of the form rule and its premises can be derived using the weak rule several times. From this, we can derive $A, B : * \vdash A \to (B \to A) : *$ by a form rule, using the fact that $A, B : * \vdash A : *$. The rest of the proofs involved are almost trivial:

$$\dfrac{\dfrac{A,B:*,x:A,y:B \vdash x:A \qquad A,B:*,x:A \vdash B \to A:*}{A,B:*,x:A \vdash \lambda y:B.x:B \to A}\,\text{abst} \qquad A,B:* \vdash A \to (B \to A):*}{A,B:* \vdash \left(\lambda x:A, y:B.x\right) : A \to (B \to A)}\,\text{abst}$$

A blank page.

Chapter 5

Applying logic: proof assistants

5.1 What is a proof assistant?

Using the coding scheme we presented in the last chapter, it is possible to develop formal proofs without having to allude to a metalevel: if a proposition is true, we find a constructive proof as an inhabitant of the type corresponding to the proposition. This makes it easy to implement an algorithm in a computer to check whether a proof is correct or not. The formalism behind this is the Curry-Howard isomorphism between computer programs and formal proofs. We already mentioned it without giving it a name: 'types correspond to propositions and terms to proofs'.

Such programs are called *automated proof checking softwares*. If a software is able to build up new proofs, we say it is a *automated theorem prover*. Somewhere in between are *proof assistants*, which are tools that help the user to check the validity of a proof and provide a general guidance on how to perform them. They can even, in some simple cases, prove a theorem or complete a proof on their own.

We can find an example of the importance of proof assistants in mathematics in the four-colour theorem. The statement of the theorem is the following: 'any map in a plane can be coloured using four colours in such a way that adjacent regions do not have the same colour'. In 1976 and 1996, some proofs of said theorem were presented but they involved a computer checking an extremely large amount of cases (with the correspondent extremely large computational time). Many mathematicians did not accept any of the proofs because the computations could not be checked. A decade later, in 2005, a mathematician called George Gonthier managed to write a proof using a proof assistant (concretely, Coq). This cleared the doubts about the correctness and the theorem is now fully accepted. If the reader is interested, a short article on this is [Gonthier]. Another example of this nature is Kepler's conjecture ('No arrangement of equally sized spheres filling space has a greater average density than that of the cubic close packing and hexagonal close packing arrangements'), that was finally proved in 2014 using the proof assistants HOL light and Isabelle.

We already mentioned a couple of proof assistants. Some more are Mizar, Lean and LEGO. Although each one has different features, we choose to focus on Coq for illustrative means.

5.2 Coq: an overview

Coq (which means 'rooster' in French), was released in 1989 by the French Institute for Research in Computer Science and Automation (INRIA). The project was led by Gérard Huet and Thierry Coquand (hence, the name 'Coq'). Based on the calculus of constructions, it can represent proofs in higher-order logic and is written in the OCaml language.

Coq's most important part is the kernel. It is a type checker, that is, its function is to check the validity of the proofs. In other words, trusting Coq is the same as trusting its kernel. Besides, it is an organ independent of the rest of the program.

Let us see a detailed example of a small proof derived in Coq. Our theorem will sound familiar: 'For any natural number n, either $n = 0$ or it is a successor'.

```
Theorem example : forall n:nat, (n=0 \/ exists m:nat, (n=S(m))).
```

We first give a name to the theorem: *example*. As we saw, sets are interpreted as types, so instead of $n \in \mathbb{N}$, we write $n : nat$. At this point, Coq returns a single goal:

```
1 subgoal
______________________________________(1/1)
forall n : nat, n = 0 \/ (exists m : nat, n = S m)
```

The main structure we see in the goal is of the form *forall A:B, C*, so we introduce a variable n of type *nat* in our first line of the proof:

```
intros n.
```

We have added the condition $n : nat$ to our context. The current goal is:

```
1 subgoal
n : nat
______________________________________(1/1)
n = 0 \/ (exists m : nat, n = S m)
```

We need now to distinct two cases, when $n = 0$ and when $n > 0$. The tactic *case* does this for us. Tactics are the tools in Coq that allow us to replace goals for different ones. For example, the tactic *intros* we used takes a goal of the form *forall n, P(n)* and replaces it for the goal *P(n)* while adding the variable n to the context. The tactic *case n* splits the subgoal in two: in the first one, it replaces the variable n for 0 and, in the second, for a number greater than 0, i.e. a successor number $S(n')$.

```
case n.
```

Indeed, Coq creates two su*b*goals, one for each case:

```
2 subgoals
n : nat
______________________________________(1/2)
0 = 0 \/ (exists m : nat, 0 = S m)
______________________________________(2/2)
forall n0 : nat, S n0 = 0 \/ (exists m : nat, S n0 = S m)
```

We see that the first one is very easy to prove, since $0 = 0$. We must tell Coq that we are going to prove the left term and finally, using the reflexivity property of equality, we provide a proof of $0 = 0$:

```
left.
exact (eq_refl 0).
```

Once the subgoal is closed, we are only left to prove the second one, for the case that $n > 0$. The beginning of the proof is similar to the one we just did:

```
intros n0.
right.
```

The current goal is:

```
1 subgoal
n, n0 : nat
______________________________________(1/1)
exists m : nat, S n0 = S m
```

We must find a witness for the existential quantifier. The correct m is, of course, $n0$.

```
refine (ex_intro _ n0 _).
```

The goal has been simplified to:

```
1 subgoal
n, n0 : nat
______________________________________(1/1)
S n0 = S n0
```

But this one we already know how to solve:

```
exact (eq_refl (S n0)).
```

For finishing the proof, we must write *Qed.* at the end. The whole Coq code was:

```
Theorem example : forall n:nat, (n=0 \/ exists m:nat, (n=S(m))).
Proof.
  intros n.
  case n.
  left.
  exact (eq_refl 0).

  intros n0.
  right.
  refine (ex_intro _ n0 _).
  exact (eq_refl (S n0)).

Qed.
```

However, can we be sure of the validity of a proof derived in Coq? We already mentioned that we must trust the kernel to do so, but there is more to this. Here are some of the several factors adressed in [Adams] and [Schnider] that we have to take into account:

- Is Coq correctly implemented? In other words, can we trust the kernel?
- Are we working with what we really want to be working with? We will provide an example in Chapter 8 of an attempt of implementation of an ontology that fails to capture said ontology, although it looks like it does at first sight.
- Have we added any axioms that make the system inconsistent? We have to be very careful when adding axioms to systems, because if there is an inconsistency, anything can be proved!
- Is Coq consistent? This question reduces to 'Is the Calculus of Inductive constructions consistent?'. In the paper [Werner], Benjamin Werner presents a surprising result: that the Calculus of Inductive Constructions is equivalent to the ZF set theory. The proof is not very hard to read and we encourage the reader to do so. Therefore, the question that remains is 'Is Set Theory consistent?' And we know we have to take a leap of faith on this.

Part II

The case study part

Blank page syndrome. Never again.

Chapter 6

Becoming lawyers: Law 561

6.1 Logicians as lawmakers

In this second part of the thesis, we are going to study the *Regulation (EC) No 561/2006 of the European Parliament and of the Council*, as published in the Official Journal of the European Union (see [Law 561]). We shall refer to it as *Law 561* for short.

Why is it interesting to consider a law? Not a mathematical law but a legal one. Because laws are written in a natural language that is subject to interpretations, unlike the mathematical and logical theories we are used to. A single word can have two different meanings (man, right), or be diffuse (good, many). Even some constructions can be ambiguous, like "A or B", which can be interpreted as an exclusive or inclusive disjunction. Lawmakers are constantly trying to improve their legal texts by eliminating these ambiguities. But it is a never-ending story, laws keep being readable in various ways and then lawyers have to defend their different positions in court.

Imagine that the Law would be one and only: if A is written, A is done. No possible misunderstandings or loopholes. Judges would only have to press a button and a verdict would be output. A possibility for this is to have logical laws (and by "logical", we mean "based on Logic").

This can be achieved by two ways: formalising an existing law into mathematics or logic or creating a new law that already belongs to a formal system. The two directions have issues: in the first one, how can we be sure that we are faithfully formalising the spirit of the law? It is possible (and has happened multiple times) that something seems like a good representation of an ontology but it turns out to have a slightly different detail that can be interpreted in a distinct way. We will talk about this in Chapter 8. Moreover, we already mentioned that laws are often ambiguous, so if there is more than one possibility, which interpretation should we take? We are bound to have to make assumptions, like we will do in Chapter 7, and be very careful.

In the second way we referred to, writing a law directly in a mathematical style, the problem is evident. Laws apply to the world and must have some *meaning*. Even if we have not written a legal law yet, we have a notion of what we want (that some employees must not work on Sundays and that they cannot take a free Saturday more than once per month, for example). So, in reality we are not creating a law from scratch, but formalising a primitive law that has not yet been written. Thus, we are back to were we started.

As a real-life example of an attempt to achieve this, the Dutch Tax and Customs Administration started in 1990 a project whose purpose is to take care of all the process, from creating legal texts to implementing the normative. The name of the project is POWER, for 'Programme for an Ontology-based Working Environment for design and implementation of Rules and regulations'. Its aim was to extract knowledge from different juridical sources and implement it in order to reduce costs, time and effort. More about POWER and its verification and validation techniques (called VALENS) can be found in [Spreeuwenberg].

In addition to this, a research was done on automated norm extraction, i.e. given a law text in natural language, they extract the norms in it. For the interested reader, [Engers] provides an explanation on the state of this norm extraction programme.

In the last years, more projects and companies have been created with the goal of formally verifying the legislation.

6.2 The law

Law 561 is a legislation regarding *road transport*. It affects drivers that carry passengers or goods, like truck drivers and bus drivers. We shall only consider parts of Articles 4, 6, 7 and 8, which establish for how long should a driver work and for how long should they rest.

Article 4 contains definitions of the principal terms that will appear in the rest of the articles, such as *driver*, *vehicle* and *rest*. We reproduce some of them in the following lines. We shall omit some irrelevant parts of the articles.

> **Article 4:**
>
> For the purposes of this Regulation the following definitions shall apply:
>
> [...]
>
> (f) 'rest' means any uninterrupted period during which a driver may freely dispose of his time;
>
> (g) 'daily rest period' means the daily period during which a driver may freely dispose of his time and covers a 'regular daily rest period' and a 'reduced daily rest period':
>
> — 'regular daily rest period' means any period of rest of at least 11 hours. Alternatively, this regular daily rest period may be taken in two periods, the first of which must be an uninterrupted period of at least 3 hours and the second an uninterrupted period of at least nine hours,
>
> — 'reduced daily rest period' means any period of rest of at least nine hours but less than 11 hours;
>
> (h) 'weekly rest period' means the weekly period during which a driver may freely dispose of his time and covers a 'regular weekly rest period' and a 'reduced weekly rest period':

— 'regular weekly rest period' means any period of rest of at least 45 hours,

— 'reduced weekly rest period' means any period of rest of less than 45 hours, which may, subject to the conditions laid down in Article 8(6), be shortened to a minimum of 24 consecutive hours;

[...]

(q) 'driving period' means the accumulated driving time from when a driver commences driving following a rest period or a break until he takes a rest period or a break. The driving period may be continuous or broken.

Article 6 regulates driving times, so *t*hat drivers do not work for too long:

Article 6:

1. The daily driving time shall not exceed nine hours.

However, the daily driving time may be extended to at most 10 hours not more than twice during the week.

2. The weekly driving time shall not exceed 56 hours [...].

3. The total accumulated driving time during any two consecutive weeks shall not exceed 90 hours.

[...]

Article 7 deals with break times. We do not reproduce it here, simply know that break times are regulated. Article 8 is about rest times and it is the most crucial one, specifically Article 8.6, which we will study in more detail in the next chapter.

Article 8:

1. A driver shall take daily and weekly rest periods.

2. Within each period of 24 hours after the end of the previous daily rest period or weekly rest period a driver shall have taken a new daily rest period.

[...]

4. A driver may have at most three reduced daily rest periods between any two weekly rest periods.

[...]

6. In any two consecutive weeks a driver shall take at least:

— two regular weekly rest periods, or

— one regular weekly rest period and one reduced weekly rest period of at least 24 hours. However, the reduction shall be compensated by an equivalent period of rest taken en bloc before the end of the third week following the week in question.

> A weekly rest period shall start no later than at the end of six 24-hour periods from the end of the previous weekly rest period.
>
> 7. Any rest taken as compensation for a reduced weekly rest period shall be attached to another rest period of at least nine hours.
>
> [...]
>
> 9. A weekly rest period that falls in two weeks may be counted in either week, but not in both.

6.3 Some consistent yet absurd situations and some inconsistent readings

In this section, we shall become more familiar with Law 561 by analisying simple cases and existing holes.

- What is 'time'? What is 'period'? They are not defined in the law and they are sometimes used as 'intervals' and sometimes as 'duration of intervals'. For example,

  > (f) 'rest' means any uninterrupted period during which a driver may freely dispose of his time;

  Here 'period' means 'interval of time'. But consider this other definition:

  > (g) [...] — 'reduced daily rest period' means any period of rest of at least nine hours but less than 11 hours;

  'Period' means now 'the duration of the interval of resting'. We humans understand what the law means in each case, but if we were to implement a computational law, we would have to be careful about these details.

- In Article 4, the word *rest* is defined, but we are not able to find a definition of *drive* or *driving*. Moreover, a *driver* is defined as someone who drives the vehicle, which seems like a circular description.

- A *week* is defined in Article 4 as:

  > (i) 'week' means the period of time between 00.00 on Monday and 24.00 on Sunday;

  Does this mean that the period of time from a Wednesday to a Tuesday is not a week? It seems so. This could have been written in this way so police officers only check the legality of weeks starting on Monday and ending on Sunday (indeed, it would be substantially more complex to check every single 168-hour period of the tachograph). We would be happy to accept this sense of the word if the following issue did not arise.

In later parts of the law, the adjective 'weekly' appears. In case you are not good with words, 'weekly' here means 'once in a week', that is, 'once in a period of time from 00.00 on a Monday and 24.00 on the following Sunday'. Therefore, a period that goes from 15:51 on a Saturday to 13.45 on the following Monday cannot be considered as a weekly rest period. In fact, no weekly rest period can fall in two weeks, making Article 8.9 unnecessary. What really happens is that, besides having fixed a meaning for 'week', Law 561 uses both of its primitive meanings in different parts.

- Exceptions are poorly expressed. For example, in Article 6,

  > 1. The daily driving time shall not exceed nine hours.
  >
  > However, the daily driving time may be extended to at most 10 hours not more than twice during the week.

  It tells us that the driving time cannot exceed 9 hours AND that it can exceed 9 hours, which is an inconsistency. It should be rephrased in a way like 'If A, then B; otherwise C'.

- Consider a situation in which a driver goes on a holiday for a month. They are obviously resting, and by Article 8.9, this 30-day rest is assigned to the first week of the month, leaving the remaining ones restless. Of course, we know this situation is not illegal, but the tachograph (the machine that records the schedules of the drivers) does not know how to interpret this in the correct way, and so police officers if we were to model them in a naive way, in which they would stick to the letter of the law.

- In the same spirit, assume a driver starts a weekly rest at midday on a Sunday and ends it on the next Saturday. By Article 8.9, this rest corresponds to the first week (the one in which the driver started the weekly rest), and so the driver is fined because they have not rested enough during the second week (although they did rest for almost a week). Indeed, they did something wrong: not resting enough on the first week, but the reason the policer officer provides when giving the fine is 'You have not rested during the SECOND week'. This structure can be repeated for several weeks and the result is that the driver gets fined for a week but because of another week that may have taken place months before.

- What if a driver gets stuck in a traffic jam and cannot park for several hours? According to the law, they should get a fine, since they are driving for too long (remember that 'driving' is not defined).

- This is a quite curious aspect of the law: if a driver just got their license, they can drive for as long as they want, given that a driving period is something between two rests/breaks and there was none of them before the driver started driving.

- According to Article 8.6, if a driver takes a reduced weekly rest period,

  > the reduction shall be compensated by an equivalent period of rest taken en bloc before the end of the third week following the week in question.

Which means that if a driver starts a compensation in the weekend of the third week after a reduction and ends it on Monday, for example, they are committing an illegality because they did not take the whole rest before the end of the third week after the reduction, although, by Article 8.7, the whole rest is assigned to the correct week.

- If there is a power cut and the tachograph stops recording, it may assign the activity 'resting' to the period of unknown data, and we have already seen some cases in which too much rest might be a problem; or it may assign the activity 'driving', which can obviously be even more problematic (think of a power cut of two days).

- Article 6.4 states that

  > Daily and weekly driving times shall include all driving time on the territory of the Community or of a third country.

  By 'Community', we mean the European Co*m*munity. So, what if a driver goes from Finland to Russia (or they have to take a detour in Russia but then they return to Finland)? They are not going to a *third country* and the driving time in Russia should be not included, since the law only contemplates the EU and a third country. The same if there are more than three countries.

All of these examples fall into one of two categories: situations that are illegal (according to the law) but obey its spirit and situations that are legal but do not obey the spirit of the law. For example, having a month of holidays falls in the first category and driving for as long as we want right after getting our driving license, in the second. There is, though, a more problematic third category: situations that have unsatisfiable requirements. We have not found any so far (the exception in Article 6.1 would be an example, but it is so easily avoided that we disregard it), but if we did, the catastrophic state of the law would become evident.

These are only some of the multiple problems that Law 561 generates. In Chapter 7, we shall mention a couple more related to Article 8.6. Of course, in most of these examples, we are taking a literal reading and we understand the spirit of the law. No one is going to jail for having a month-long holiday in Egypt, unless for example, they murder someone in the Nile or try to smuggle some ancient relics. It is the role of the lawyers, to interpret the law and serve justice according to it. But, again, we defend a flawless and logical law that does not depend on someone's mood, but on a computer's software.

Chapter 7

Studying the law: on implication span of Reduced Weekly Rest Periods

7.1 The problem

In all the laws that depend on time, like the one we are dealing with, there is an important property to take into account, as they should aim to have: locality. Consider the following situation: a period of time is being checked to determine whether it is legal or not. Is it enough to only check that period? Maybe the legality there depends on what was done a month before, or two days later. For example, in a certain company each worker has to work for a minimum number of days each month. If a manager checks only the last two weeks of a month, it may be unclear whether a worker has followed the rules. The manager should check the whole month.

We call this period of time that has to be looked at to determine the legality of an interval the *implication span*. An implication span can be either finite or infinite (unbounded). Ideally, laws should be local, that is, their implication span should be finite and preferably small, but the real world is not perfect and there are some laws with infinite implication spans. What happens then? Do we have to examine every single moment in History, even before the humans walked the Earth? The answer is, obviously, no. At some point, there must be a cut in time, either because the law was not in force yet, the records are finite or there is a complementary law to regulate this. Therefore, we are bound to miss some data.

For example, if we look at Article 8.4:

> A driver may have at most three reduced daily rest periods between any two weekly rest periods.

Using the definition of 'week' the law provides, we have that the implication span for daily rest periods is just seven days, finite. Any police officer can check whether there are more than three reduced daily rest periods within a week or not without having to look at the following week or the one before.

In this chapter, we are going to study Articles 8.6 and 8.7 of the law and discuss the corresponding implication span (for weekly rest periods). Remember that 8.6 says:

> In any two consecutive weeks a driver shall take at least:

> - two regular weekly rest periods, or
>
> - one regular weekly rest period and one reduced weekly rest period of at least 24 hours. However, the reduction shall be compensated by an equivalent period of rest taken en bloc before the end of the third week following the week in question.

In other words, a driver can*n*ot take two consecutive reduced weekly rest periods. Recall that a weekly rest period is of at least 45 hours and a reduced one, of less than 45 but of at least 24 hours. The expression "en bloc" means that the driver must perform the compensation without any interruption, it cannot be distributed along a week, for example.

Our second focus point, Article 8.7, states:

> Any rest taken as a compensation for a reduced weekly rest period shall be attached to another rest period of at least nine hours.

In the following discussion, we disregard the cases in which a reduction is attached to a daily rest period or a break, that is, we will only consider weekly rest periods and reduced weekly rest periods.

From now on, we shall sometimes write WRP and RWRP for 'weekly rest period' and 'reduced weekly rest period', respectively. Another important remark is that if a driver takes a RWRP of x hours, the compensation must be the difference between the RWRP and what a minimal WRP should be, that is $45 - x$ hours. For example, if the driver rested only for 35 hours, they should compensate 10 hours.

Let us examine the implication span for these two articles. If a police officer takes a tachograph and checks a period I of, say, two weeks, if there is a RWRP in the last week, at least three more weeks have to be checked. But what happens if there is another RWRP in those three weeks? More weeks must be added to the list and we can see there is no bound to the right.

But the problems do not end here. What if there was a RWRP a couple of weeks before the start of the considered interval? What if one of the rest periods in I is actually a compensation plus a rest period? In this case, the real period of rest for that week is smaller, and may not be long enough to be a legal rest anymore. We have to check what happened three weeks before too to avoid cases like this. But, again, to check these three new weeks, we must also check the three previous weeks and so on. There is no bound to the left either. Therefore, the implication span is infinite. But the tachograph only records a finite amount of data: about a year.

And now we wonder: is it possible that there is some interval I of two weeks that looks legal on its own but in fact there exists another interval J that happened right after I and such that the concatenated interval IJ is illegal but it only became illegal in the very last week of J? If such an example exists, what can we say about the length of J? If it is small enough, there will be no problem, since it can fit inside the one-year tachograph and we do not miss any relevant information. But if, on the contrary, it is too long or arbitrarily large, there will be a big ISSUE: the tachograph will not be enough to provide all the information the police officer needs. If that is the case, some decisions should be taken: should we make a new law to avoid the problem or, after analisying its likelihood in the real world, keep the law as it is hoping this case never happens?

The reader might be wondering why we chose I to be of two weeks. In principle, it is a parameter that could have any value, but, in order to mirror the real world, we decided it to be two because that is the length of the random interval the police checks when they examine a tachograph.

In the following sections, we take a mathematical approach to formalise the situation described above and solve the question we posed.

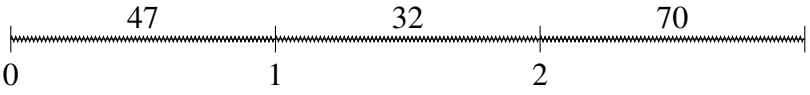
7.2 Formalisation

When Article 8.6 talks about when the compensation shall be taken, it says "before the end of the third week following the week in question". There are no bounds to the left, the compensation could be taken before the reduction is made. For instance, a week before or even three years before. For this reason, we can believe that this article does not create any illegal cases: since the tachograph is finite, if a police officer checks it and finds any uncompensated reduced weekly rest period, the driver can excuse himself alleging that he had taken that compensation in 1960. But this, of course, makes no sense. In this chapter, and according to law practice, we shall assume that the compensations must be taken after the reductions (and so, we may have illegalities). This is not entirely satisfactory because what happens if a driver knows that in the next weeks they are going to work too much and decides to compensate now that they have time? They should be able to do so. But then, where is the limit between "too early" and "acceptable" for taking the compensations in advance? The notion of pre-compensation makes sense and so, the law should be rephrased to take into account these cases.

In the following lines, we present an intuitive definition of intervals of weeks with associated rest times.

Definition 7.2.1. A *schedule* is a function $f : \{0,...,n\} \to [0,168] \cap \mathbb{Q}$, for some $n \in \mathbb{N}$. We say that $n+1$ is the *length* of the schedule.

For example, the function $f : \{0,1,2\} \to [0,168] \cap \mathbb{Q}$ such that $f(0) = 47$, $f(1) = 32$ and $f(2) = 70$ is a schedule. We may express it by the following picture:



This intends to represent an interval of three weeks. For instance, the first week, labeled 0, has a rest period of 47 hours. Note that we could have defined a schedule as a function from $\{0,n\}$ to $[0,168] \cap \mathbb{N}$, or even to $[0,168] \cap \mathbb{R}$. It depends on the type of Time we use: if we allow drivers to rest for any number of hours (like $\sqrt{2}$), or consider only integer numbers, rationals,... Since the tachograph uses seconds, we may have fractions of hours and that is why we wrote $\mathbb{Q}$ in the definition.

Another remark we should mention is that we use initial segments of the natural nu*m*bers. It is possible to change the definition and work with any interval of natural numbers. But the important things about an interval are its length and the outcome of f, i.e., the duration of the

rest periods. It does not matter if the first week is labeled as 0 or as 103. For this reason, and without loss of generality, we use initial segments of $\mathbb{N}$.

We made an implicit assumption when we said that f is a function, that is, that there is an image $f(i)$ to each number i of the interval. But, in the real world, it is possible that a driver takes two or more weekly rests. In this case, it would not be very clear how to perform: should we consider only the first one, the longest one, the sum of all the hours,...? Thus, we assume that there is one and only one weekly rest period associated to each week. Note that a driver may not take any weekly rests within a week (this case is, of course, illegal), but then $f(i) = 0$ for the corresponding i.

Before we move on to the next definition, let us point out one final assumption that we made: Article 8.9 states that:

> A weekly rest period that falls in two weeks may be counted in either week, but not in both.

And so we assume that this prior step has already been made. Each rest period belongs to only one week.

A given interval of weeks could be interpreted in different ways. For example, consider

We do not care yet about legality, but it is clear that there are RWRPs in Weeks 0 and 2. In Week 1, there are more readings, for example: there could be a WRP of 65 hours, or a RWRP of 43 hours plus a 22-hour compensation. The legality of the case may depend on the interpretation we take. This motivates the following definition:

Definition 7.2.2. Given a schedule $f : \{0,...,n\} \to [0,168] \cap \mathbb{Q}$, an *interpretation* of f is a pair $(\tilde{f}, g)$, where $\tilde{f}$ is a schedule of length $n+1$ such that $\tilde{f}(i) \leq f(i)$ for all $i \in \{0,...,n\}$ and $g : \{0,...,n\} \to \{0,...,n+3\}$ such that $g(i) \in \{i, i+1, i+2, i+3\}$ for all $i \in \{0,...,n\}$.

Intuitively, the function g assigns a value to each week according to when it has a compensation. If $g(i) = i$ for some i, it means that there is no compensation (either because there is no RWRP or because the driver decides not to compensate, in which case, that is illegal). Note that a compensation may fall after the schedule ends: for example, $g(i) = n+1$. The function $\tilde{f}$ is a new schedule where the compensations have disappeared and the rest periods become more evident. Now, we can jump right into defining what a legal schedule is:

Definition 7.2.3. A schedule $f : \{0,...,n\} \to [0,168] \cap \mathbb{Q}$ is *legal* (or consistent) if there exists an interpretation $(\tilde{f}, g)$ of f such that:

i) For any $i \in \{0,...,n\}$, we have $24 \leq \tilde{f}(i)$;

ii) For any $i \in \{0,...,n-1\}$, it is not the case that $\tilde{f}(i) < 45$ and $\tilde{f}(i+1) < 45$;

iii) For any $i \in \{0,...,n\}$, we have $\tilde{f}(i) < 45$ if and only if $g(i) > i$.

iv) For all $i \in \{0,...,n\}$, we have $\tilde{f}(i) + \sum_{j \in g^{-1}(i) \setminus \{i\}} \left(45 - \tilde{f}(j)\right) = f(i)$.

A schedule is *illegal* if it is not legal.

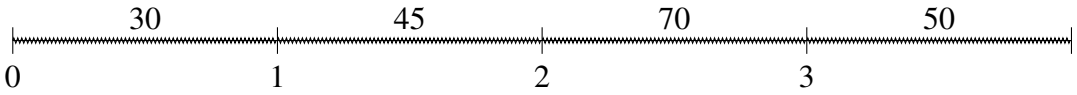
We may use the expressions 'I is a legal interpretation for the schedule f' and 'I makes f legal' for 'f is a legal schedule and I is an interpretation of f that fulfills conditions i) to iv)'.

Note that this definition is a formalisation of Articles 8.6 and 8.7 of the Law. In particular, Condition iii) assures that all reductions have a compensation and all compensations come from a prior reduction. This may seem superfluous, but in Section 8.3 we will see that it is also important, or else we could have negative compensations.

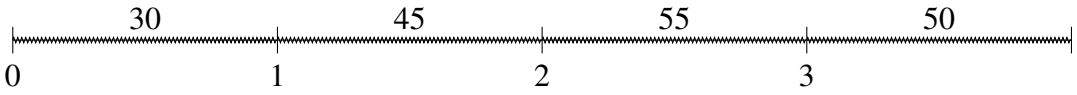
The last definition follows the *In dubio pro reo* rule: when there are several interpretations of the events, the one that favours the defendant should be chosen. For example, if a period of compensation has not ended yet, we assume the corresponding reduction to be legal (by "In dubio pro reo" it is possible that the driver compensates the following week after the present one). How is this grasped in the previous definition? There is some $i \in \{0,...n\}$ such that $f(i) < 45$ and $i+3 > n$. We take an interpretation with $g(i) = i+3$ and Condition iii) holds. Part iv) also holds trivially, since $i+3 \notin \{0,...,n\}$.

Moreover, this definition takes into account the cases in which several compensations may occur within the same week. And, finally, it also assures that each compensation comes from a unique reduction: in principle, the Law does not exclude the possibility that a single 5-hour compensation serves for two 40-hour RWRPs, but since it does not seem reasonable, we assumed it not to be legal.

Example 7.2.4. The following schedule is legal:

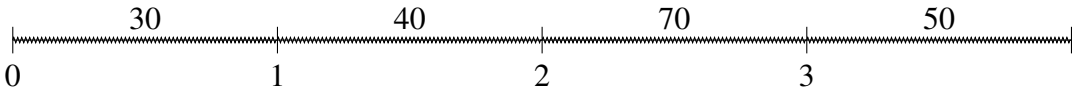


A possible interpretation that makes it legal is:



with $g(i) = i$ for all $i > 0$ and $g(0) = 3$.

This schedule, however, is illegal:



All possible interpretations have a RWRP in both Weeks 0 and 1.

Definition 7.2.5.

1) Let f and g be two schedules of lengths $n+1$ and $m+1$, respectively. The *concatenation* of f and g, denoted by $f^\frown g : \{0,...,n,n+1,...,n+m+1\} \to [0,168] \cap \mathbb{Q}$, is the schedule defined as:

$$f^\frown g(i) = \begin{cases} f(i), & \text{if } i \leq n, \\ g(i-n-1), & \text{if } i > n. \end{cases}$$

2) Let $f : \{0,...,n\} \to [0,168] \cap \mathbb{Q}$ be a schedule and let $m < n+1$. We denote by f_{-m} the schedule that results after eliminating the rightmost m weeks of the schedule, that is, $f_{-m} : \{0,...,n-m\} \to [0,168] \cap \mathbb{Q}$ such that $f_{-m}(i) = f(i)$ for $i \leq n-m$.

3) Let $f : \{0,...,n\} \to [0,168] \cap \mathbb{Q}$ be a schedule and let $m < n+1$. We denote by f^{-m} the schedule that results after eliminating the leftmost m weeks of the schedule, that is, $f^{-m} : \{0,...,n-m\} \to [0,168] \cap \mathbb{Q}$ such that $f^{-m}(i) = f(i+m)$ for $i \leq n-m$.

And, finally, we arrive to the c*r*ucial definition of this chapter, the one that formalises the intervals we discussed in the introductory section.

Definition 7.2.6. Let f be a legal schedule of length 2. Then:

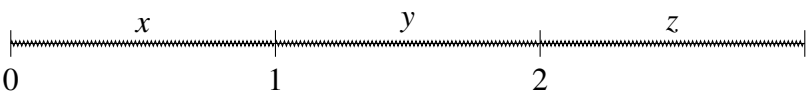
i) f has the *P-ambiguity property* (PAP), or is *P-ambiguous*, if there exists some $n \in \mathbb{N}$ and a legal schedule h of length $n+1$ such that the schedule $h^\frown f$ is illegal but $(h^\frown f)^{-1}$ is legal. We may also say that the schedule is P-ambiguous for n.

ii) f has the *A-ambiguity property* (AAP), or is *A-ambiguous*, if there exists some $n \in \mathbb{N}$ and a legal schedule h of length $n+1$ such that the schedule $f^\frown h$ is illegal but $(f^\frown h)_{-1}$ is legal. We may also say that the schedule is A-ambiguous for n.

The P and the A in the definitions come from "prepending" and "appending", since in the PAP definition, we prepend a new schedule and in the AAP definition, we append it. The length of f is assumed to be 2 because, in real life, police officers check for intervals of two weeks. This way, we can study when one of such intervals is ambiguous or not. Of course, the value 2 is just a parameter, we could have defined ambiguity for a general length m.

In the following examples, we are examining some schedules that have ambiguity properties and some that do not.

Example 7.2.7. For n=0, there is no schedule with either the PAP or the AAP if we want the illegality of the extended schedule to happen because of a violation of iii) and iv) in the definition of legality.



For any f of length 2 and h of length 1, the concatenations $f^\frown h$ and $h^\frown f$ have length 3. And no schedule s of length 3 can violate any of Conditions iii) and iv): take an interpretation $(\tilde{s}, g)$ of s in such a way that $\tilde{s} = s$ and if $s(i) < 45$, then $g(i) = i+3$, and if $s(i) \geq 45$, then

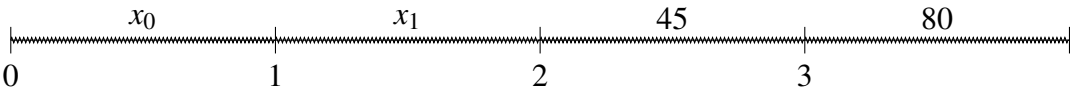
$g(i) = i$. It is easy to check that this interpretation fulfills both Conditions iii) and iv). The reason behind this is that the period of compensation has not ended yet and, by *In dubio pro reo*, the compensation falls out of the picture.

Nonetheless, if we also consider Condition ii), we can find an easy example of AAP: take the schedules $f : \{0,1\} \to [0,168] \cap \mathbb{Q}$ with $f(0) = 45$ and $f(1) = 40$ and $h : \{0\} \to [0,168] \cap \mathbb{Q}$ with $h(0) = 40$. Separately, both are legal schedules, but when we concatenate them, there are two consecutive weeks with RWRPs. However, when we erase the last week, we end up with only f, which was legal.

For PAP, an analogous example suffices: $f : \{0,1\} \to [0,168] \cap \mathbb{Q}$ with $f(0) = 40$ and $f(1) = 45$ and $h : \{0\} \to [0,168] \cap \mathbb{Q}$ with $h(0) = 40$.

Remark 7.2.8. *In the real world, there are more examples for $n = 0$ with the general notion of legality, involving the whole Law 561. For instance, imagine an interval of two weeks that ends with a weekly rest period (regular or reduced). Then, append a week that starts with another weekly rest period. Now, there is no interruption between both rests and the tachograph interprets them as a single rest. But, in that case, technically there is a week with no weekly rest, so the situation is illegal. And, when we erase the last week, the problem no longer occurs. Therefore, this would be an example of A-ambiguity, if we took the whole law for the definition of legality.*

Example 7.2.9. The following schedule is not P-ambiguous: $f : \{0,1\} \to [0,168] \cap \mathbb{Q}$ with $f(0) = 45$ and $f(1) = 80$. Assume the contrary: there is some legal $h : \{0,...,n\} \to [0,168] \cap \mathbb{Q}$ such that $h^\frown f$ is illegal and $(h^\frown f)_{-1}$ is legal. We should prove this by induction on n, but since this is one of the first examples and we do not want to present an intricate proof this early, we will prove it for $n = 1$ and the reader can check the rest (or believe us when we say it is true).



To begin with, both $h(0) = x_0$ and $h(1) = x_1$ must be greater or equal to 24, or otherwise h would not be legal. If $x_0 \geq 45$, take the interpretation $(\widetilde{(h^\frown f)}, g)$ such that $\widetilde{(h^\frown f)}(0) = h(0)$, $\widetilde{(h^\frown f)}(1) = h(1)$, $\widetilde{(h^\frown f)}(2) = f(0)$ and $\widetilde{(h^\frown f)}(3) = f(1)$ and $g(i) = i$, for all $i \neq 1$, and $g(1)$ is either 1 or 4, depending on whether $h(1) < 45$ or not. This interpretation makes the schedule legal, a contradiction, since we assumed it was not. On the contrary, if $h(0) < 45$, since h is legal, it cannot be the case that $h(1) < 45$. Hence, we can use the same interpretation as before, but with $\widetilde{(h^\frown f)}(3) = f(1) - (45 - h(0))$ and $g(i) = i$, for all $i \neq 0$, and $g(0) = 2$. Note that $f(1) - (45 - h(0))$ will always be greater or equal to 24, since $f(1) = 80$ and $24 \geq h(0) < 45$. This interpretation makes the schedule legal, a contradiction. Therefore, we checked that there is no schedule h of length 2 such that $h^\frown f$ is illegal, so we conclude that f is not P-ambiguous.

7.3 The results

Lemma 7.3.1. *Let $f : \{0,...,n\} \to [0,168] \cap \mathbb{Q}$ be a schedule.*

i) If f is illegal, then $h^\frown f$ is illegal, for any schedule h.

ii) If $h : \{0,...,m\} \to [0,168] \cap \mathbb{Q}$ is a schedule such that $h(i) = 45$ for all $i \in \{0,...,m\}$, then $h^\frown f$ has the same legality as h, i.e., it is legal if and only if f is legal.

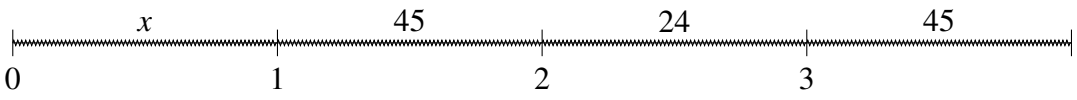
Proof. Trivial.

□

Lemma 7.3.2. *Let $n \geq 3$. Let $f : \{0,...,n\} \to [0,168] \cap \mathbb{Q}$ be a schedule such that $f(0) < 45$, $f(n-1) = 24$ and $f(i) = 45$ for all $i \neq 0, n-1$. Then, f is illegal.*

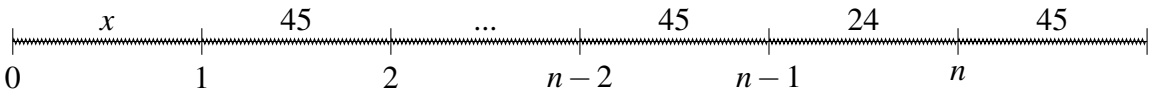
Proof. We prove the lemma by induction on n.

For $n = 3$, the schedule is:



Let us assume it is legal. Then, there is some interpretation $(\tilde{f}, g)$ that fulfills i)-iv) of the definition. We note that, by assumption, $\tilde{f}(0) < 45$, and we have that $g(0) \in \{1,2,3\}$. If $g(0) = 1$, then it is clear that $\tilde{f}(1) < f(1) = 45$, but then Condition ii) is not satisfied. The same happens for the case $g(0) = 3$, since $\tilde{f}(2) \leq 24 < 45$. If $g(0) = 2$, then $\tilde{f}(2) < f(2) = 24$ and it is now Condition i) that is not satisfied. In any case, we reach a contradiction: f must be illegal.

For $n > 3$, the picture is similar, but has $n-2$ weeks of 45 hours between the first and penultimate weeks:



As before, assume it is legal: there is some interpretation $(\tilde{f}, g)$ that fulfills i)-iv) of Definition 7.2.3. The value of $\tilde{f}(0)$ must be less than 45, so $g(0) \in \{1,2,3\}$. It cannot be $g(0) = 1$ because then $\tilde{f}(1) < 45$ and Condition ii) would not be satisfied. If $g(0) = 2$, we have that $\tilde{f}(2) < 45$. It is clear that $\tilde{f}^{-2}$ together with g restricted to $\{2,...,n\}$ is a possible interpretation of the schedule $f' : \{0,...,n-2\} \to [0,168] \cap \mathbb{Q}$ defined as $f'(0) = \tilde{f}(2)$, $f(n-3) = 24$ and $f(i) = 45$ for all $i \neq 0, n-3$. By induction hypothesis, this schedule is illegal. And, therefore, so is the interpretation we just considered. By the previous lemma, f is illegal, a contradiction. By an analogous argument, we can also discard the case $g(0) = 3$ and conclude that f is illegal.

□

Lemma 7.3.3. *Let $n \geq 3$. Let f be as in Lemma 7.3.2. Then, f_{-1} is legal if and only if $n \neq 4$.*

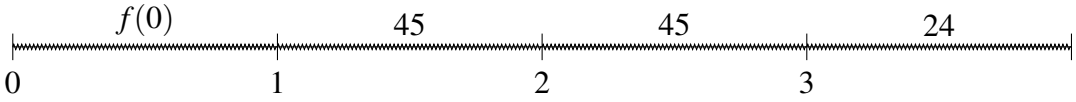
Proof. Let $n \neq 4$. We find a convenient interpretation of f depending on the parity of n.

If n is odd, take the interpretation $(\widetilde{f_{-1}}, g)$, where $\widetilde{f_{-1}}(2i) = f(0)$ and $\widetilde{f_{-1}}(2i+1) = f(2i+1)$, for $0 \leq i \neq n-1$, and $\widetilde{f_{-1}}(n-1) = 24$; and $g(2i) = 2i+2$ and $g(2i+1) = 2i+1$, for $i \geq 0$.

If n is even, take the interpretation $(\widetilde{f_{-1}}, g)$, where $\widetilde{f_{-1}}(0) = f(0)$, $\widetilde{f_{-1}}(1) = 45$, $\widetilde{f_{-1}}(2i) = f(2i)$ and $\widetilde{f_{-1}}(2i+1) = f(0)$, for $1 \leq i \neq n-1$, and $\widetilde{f_{-1}}(n-1) = 24$; and $g(0) = 3$, $g(1) = 1$, $g(2i) = 2i$ for $1 \leq i$, $g(2i+1) = 2i+2$, for $1 \leq i$ such that $2i+1 \neq n-3$ and $g(n-3) = n$.

We see that f is a legal schedule, independently of the parity of n.

If $n = 4$, the schedule f_{-1} is:

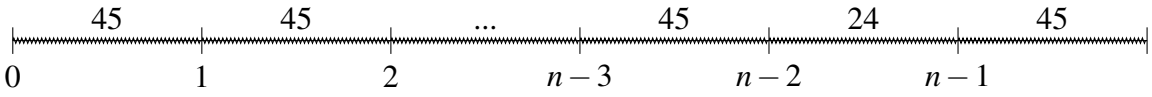


We prove it is illegal: assume there is some interpretation $(\widetilde{f_{-1}}, g)$ that makes it legal. It must be the case that $\widetilde{f_{-1}}(0) < 45$ and, hence, there is some $j \in \{1,2,3\}$ such that $g(i) = j$. If $j = 1$, then $\widetilde{f_{-1}}(1) + (45 - \widetilde{f_{-1}}(0)) = 45$, that is, $\widetilde{f_{-1}}(0) = \widetilde{f_{-1}}(1)$. In particular, $\widetilde{f_{-1}}(1) < 45$, which violates legality, since $\widetilde{f_{-1}}(0)$. If $j = 2$, we use the same argument, except for the fact that the legality does not hold because of the following week: $\widetilde{f_{-1}}(3) = 24$. Finally, if $j = 3$, then we can deduce that $\widetilde{f_{-1}}(3) < 24$, which is also illegal. In conclusion, there is no such interpretation and f_{-1} is illegal.

□

Lemma 7.3.4. *Let $n \geq 3$. Let f be as in Lemma 7.3.2. Then, f^{-1} is legal.*

Proof. The schedule f^{-1} has the following structure:



Take the interpretation $(\widetilde{f^{-1}}, g)$, where $\widetilde{f^{-1}}(i) = 45$ for all $i \neq n-2$, $\widetilde{f^{-1}}(n-2) = 24$ and $g(i) = i$ for all $i \neq n-2$ and $g(n-2) = n$.

□

Theorem 7.3.5. *For any $n \geq 0$, there exists a schedule $f : \{0,1\} \to [0,168] \cap \mathbb{Q}$ of length 2 such that f is A-ambiguous for n.*

Proof. The case $n = 0$ was already checked in the examples.

For $n = 2$, take the schedule $f : \{0,1\} \to [0,168] \cap \mathbb{Q}$ such that $f(0) = 45$ and $f(1) = 44$ and take $h : \{0,1,2\} \to [0,168] \cap \mathbb{Q}$ such that $h(0) = 45$, $h(1) = 24$ and $h(2) = 45$. But the schedule $(f^\frown h)^{-1}$ has the same structure as the one in Lemma 7.3.2 and, hence, it is illegal. By

Lemma 7.3.1 ii), $f^\frown h$ is illegal. But, by the Lemma 7.3.3, $(f^\frown h)^{-1}_{-1}$ is legal and we conclude by Lemma 7.3.1 ii) that $(f^\frown h)_{-1}$ is legal. In other words, f has the A-ambiguity property for $n = 2$.

For $n \neq 0, 2$, take the schedule $f : \{0,1\} \to [0,168] \cap \mathbb{Q}$ such that $f(0) = 44$ and $f(1) = 45$ and take $h : \{0,...,n\} \to [0,168] \cap \mathbb{Q}$ such that $h(n-1) = 24$, $h(i) = 45$ for all $i \neq n-1$. Now, $f^\frown h$ has the structure of the schedules in Lemma 7.3.2 so it is illegal. And, finally, $(f^\frown h)_{-1}$ is legal. Therefore, f has the A-ambiguity property for n.

In conclusion, we have shown that for any $n \geq 0$, there is some f of length 2 such that f has the A-ambiguity property for n.

□

The proof of existence of schedules that have the P-am*b*iguity property follows the same idea.

Theorem 7.3.6. *For any $n \geq 0$, there exists a schedule $f : \{0,1\} \to [0,168] \cap \mathbb{Q}$ of length 2 such that f is P-ambiguous for n.*

Proof. The case $n = 0$ was already checked in the examples.

For $n > 0$, consider the schedule $f : \{0,1\} \to [0,168] \cap \mathbb{Q}$ as $f(0) = 24$ and $f(1) = 45$. Now, take $h : \{0,...,n\} \to [0,168] \cap \mathbb{Q}$ such that $h(0) = 44$ and $h(i) = 45$ for all $i > 0$. The schedule $h^\frown f$ is of the form of the one in Lemma 7.3.2, so it is illegal. And, finally, by Lemma 7.3.4, $(h^\frown f)^{-1}$ is legal. That is, the schedule f has the P-ambiguity property for n.

□

7.4 Extended ambiguity

We can now turn our attention to the extended definitions of ambiguity:

Definition 7.4.1. Let f be a legal schedule of length 2 and let $m \in \mathbb{N}$. Then:

i) f has the *m-P-ambiguity property*, or is *m-P-ambiguous*, if there exists some $n \in \mathbb{N}$ and a legal schedule h of length $n+1$ such that the schedule $h^\frown f$ is illegal but $(h^\frown f)^{-m}$ is legal. We may also say that the schedule is m-P-ambiguous for n.

ii) f has the *m-A-ambiguity property*, or is *m-A-ambiguous*, if there exists some $n \in \mathbb{N}$ and a legal schedule h of length $n+1$ such that the schedule $f^\frown h$ is illegal but $(f^\frown h)_{-m}$ is legal. We may also say that the schedule is m-A-ambiguous for n.

According to this definition, being PAP or AAP is just being 1PAP or 1AAP, respectively. Let us see for what m's a schedule can have these properties.

Theorem 7.4.2.

i) If a schedule f of length 2 has the m-A-ambiguity property for n, then it has the k-A-ambiguity property for n for any $m \leq k \leq n+2$.

ii) If a schedule f of length 2 has the m-P-ambiguity property for n, then it has the k-P-ambiguity property for n for any $m \leq k \leq n+2$.

Proof. i) Assume the schedule f is m-A-ambiguous for n. Then, there exists a legal schedule h of length $n+1$ such that $f^\frown h$ is illegal but $(f^\frown h)_{-m}$ is legal. Let $m \leq k \leq n+2$. Of course, $(f^\frown h)$ remains illegal. Take the same h as before. Consider an interpretation of $(f^\frown h)_{-m}$ that makes it legal and restrict it to the domain $\{0,...,n-k\}$. It clearly is an interpretation that makes $(f^\frown h)_{-k}$ legal, so f is k-A-ambiguous for n.

Of course, k must be less than the total length of the interval, which is $n+3$.

ii) Analogous.

□

Corollary 7.4.3. *For any $n \geq 0$ and $1 \leq m \leq n+2$, there exists a schedule f of length 2 such that f has the m-A-ambiguity property for n.*

Corollary 7.4.4. *For any $n \geq 0$ and $1 \leq m \leq n+2$, there exists a schedule f of length 2 such that f has the m-P-ambiguity property for n.*

This blank page congratulates you for overcoming the last chapter.

Chapter 8

Changing the law: can we make it better?

8.1 A change in the Law: no RWRP + Compensation

What would happen if the Law is cha*n*ged and we are not allowed to take a compensation attached to a RWRP anymore? That is, when a driver takes a RWRP, they have to compensate it with a rest attached to a WRP. The theorem*s* we proved in the last sections are no longer valid, since the schedules we used do not work (we used the trick of attaching RWRP+Comp). Can we still find examples of ambiguous intervals for any length? Or is the law too strict now for that? If the latter is the case, we might be able to solve the problem we talked about in the introduction of Chapter 7 after all.

We need a new version of legality:

Definition 8.1.1. A schedule $f : \{0,...,n\} \to [0,168] \cap \mathbb{Q}$ is *s-legal* if there exists an interpretation $(\tilde{f}, g)$ of f such that:

i) For any $i \in \{0,...,n\}$, we have $24 \leq \tilde{f}(i)$;

ii) For any $i \in \{0,...,n-1\}$, it is not the case that $\tilde{f}(i) < 45$ and $\tilde{f}(i+1) < 45$;

iii) For any $i \in \{0,...,n\}$, we have $\tilde{f}(i) < 45$ if and only if $\tilde{f}(g(i)) \geq 45$;

iv) For all $i \in \{0,...,n\}$, we have $\tilde{f}(i) + \sum_{j \in g^{-1}(i) \setminus \{i\}} \big(45 - \tilde{f}(j)\big) = f(i)$.

A schedule is *s-illegal* if it is not s-legal.

The 's' in the definition stands for 'strict', as it is a stricter notion of what is legal. The following are the new definitions of ambiguity:

Definition 8.1.2. Let f be an s-legal schedule of length 2. Then:

i) f has the *strict P-ambiguity property*, or is *strictly P-ambiguous*, if there exists some $n \in \mathbb{N}$ and an s-legal schedule h of length $n+1$ such that the schedule $h^\frown f$ is s-illegal but $(h^\frown f)^{-1}$ is s-legal. We may also say that the schedule is strictly P-ambiguous for n.

ii) f has the *strict A-ambiguity property*, or is *strictly A-ambiguous*, if there exists some $n \in \mathbb{N}$ and an s-legal schedule h of length $n+1$ such that the schedule $f^\frown h$ is s-illegal but $(f^\frown h)_{-1}$ is s-legal. We may also say that the schedule is strictly A-ambiguous for n.

Theorem 8.1.3. *For any $n \geq 0$, there exists a schedule $f : \{0,1\} \to [0,168] \cap \mathbb{Q}$ of length 2 such that f has the strict A-ambiguity property for n.*

Proof. Similar to the proof of Theorem 7.3.5 but using the following schedules:

- If n is odd, take $f : \{0,1\} \to [0,168] \cap \mathbb{Q}$ as $f(0) = 44$ and $f(1) = 45$. In this case, we can pick $h : \{0,...,n\} \to [0,168] \cap \mathbb{Q}$ as $h(2i) = 44$, $h(2i+1) = 46$ for $i \in \{0,..., \frac{n-1}{2} - 1\}$, $h(n-1) = 44$ and $h(n) = 45$.

- If n is even, take $f : \{0,1\} \to [0,168] \cap \mathbb{Q}$ as $f(0) = 45$ and $f(1) = 44$. In this case, we can pick $h : \{0,...,n\} \to [0,168] \cap \mathbb{Q}$ as $h(0) = 45$, $h(2i+1) = 44$ and $h(2i+2) = 46$ for $i \in \{0,..., \frac{n}{2} - 2\}$, $h(n-1) = 44$ and $h(n) = 45$. □

Theorem 8.1.4. *For any $n \geq 0$, there exists a schedule $f : \{0,1\} \to [0,168] \cap \mathbb{Q}$ of length 2 such that f has the strict P-ambiguity property for n.*

Proof. Similar to the proof of Theorem 7.3.6 but using the following schedules:

- If n is odd, take $f : \{0,1\} \to [0,168] \cap \mathbb{Q}$ as $f(0) = 44$ and $f(1) = 45$. In this case, we can pick $h : \{0,...,n\} \to [0,168] \cap \mathbb{Q}$ as $h(0) = 44$, $h(1) = 45$, $h(2i) = 44$, $h(2i+1) = 46$ for $i \in \{1,..., \frac{n-1}{2}\}$.

- If n is even, take $f : \{0,1\} \to [0,168] \cap \mathbb{Q}$ as $f(0) = 44$ and $f(1) = 45$. In this case, we can pick $h : \{0,...,n\} \to [0,168] \cap \mathbb{Q}$ as $h(0) = 45$, $h(1) = 44$, $h(2) = 45$, $h(2i+1) = 44$ and $h(2i) = 46$ for $i \in \{1,..., \frac{n}{2} - 1\}$. □

In the same sense, we can change the generalisations of ambiguity:

Definition 8.1.5. Let f be a legal schedule of length 2 and let $m \in \mathbb{N}$. Then:

i) f has the *strict m-P-ambiguity property*, or is *strictly m-P-ambiguous*, if there exists some $n \in \mathbb{N}$ and an s-legal schedule h of length $n+1$ such that the schedule $h^\frown f$ is s-illegal but $(h^\frown f)^{-m}$ is s-legal. We may also say that the schedule is strictly m-P-ambiguous for n.

ii) f has the *strict m-A-ambiguity property*, or is *strictly m-A-ambiguous*, if there exists some $n \in \mathbb{N}$ and an s-legal schedule h of length $n+1$ such that the schedule $f^\frown h$ is s-illegal but $(f^\frown h)_{-m}$ is s-legal. We may also say that the schedule is strictly m-A-ambiguous for n.

And we have the same corollaries, since their proofs used only the results in Theorems 7.3.5 and 7.3.6 and not the definition of legality.

Corollary 8.1.6. *For any $n \geq 0$ and $1 \leq m \leq n+2$, there exists a schedule f of length 2 such that f has the strict m-A-ambiguity property for n.*

Corollary 8.1.7. *For any $n \geq 0$ and $1 \leq m \leq n+2$, there exists a schedule f of length 2 such that f has the strict m-P-ambiguity property.*

In conclusion, this "solution" would not solve our problem. The law should be even stricter or changed somewhere else so that it has a finite implication span.

8.2 A change in the Law: a shorter period for compensation

Some years ago, Law 561 was slightly different and the drivers had to take the compensation for a reduction during the following week, instead of having three whole weeks for doing so. Would our problem be solved if we did not allow such a long compensation period? Consider the following modification in the law: '[...] However, the reduction shall be compensated by an equivalent period of rest taken en bloc before the end of the first/second week following the week in question'.

As in the last section, we need to make some changes in the definitions.

Definition 8.2.1. Let $p \in \{1,2\}$. Given a schedule $f : \{0,...,n\} \to [0,168] \cap \mathbb{Q}$, an *interpretation*^{p} of f is a pair $(\tilde{f}, g)$, where $\tilde{f}$ is a schedule of length $n+1$ such that $\tilde{f}(i) \leq f(i)$ for all $i \in \{0,...,n\}$ and $g : \{0,...,n\} \to \{0,...,n+3\}$ such that $g(i) \in \{i,...,i+p\}$ for all $i \in \{0,...,n\}$.

The notion of legality remains unchanged, except for the fact that we need to use the *interpretation*^{p} instead of *interpretation*. Hence, we shall also write *legal*^{p} instead of *legal*. The concepts of ambiguity become:

Definition 8.2.2. Let $p \in \{1,2\}$ and let f be a legal^p schedule of length 2. Then:

i) f has the *P-ambiguity*^{p} *property*, or is *P-ambiguous*^{p}, if there exists some $n \in \mathbb{N}$ and a legal^p schedule h of length $n+1$ such that the schedule $h^\frown f$ is illegal^p but $(h^\frown f)^{-1}$ is legal^p. We may also say that the schedule is P-ambiguous^p for n.

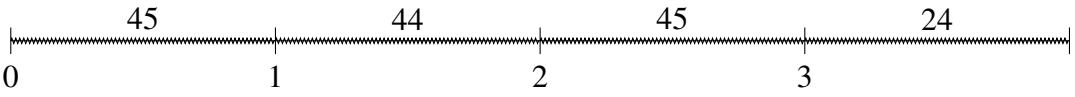
ii) f has the *A-ambiguity*^{p} *property*, or is *A-ambiguous*^{p}, if there exists some $n \in \mathbb{N}$ and a legal^p schedule h of length $n+1$ such that the schedule $f^\frown h$ is illegal^p but $(f^\frown h)_{-1}$ is legal^p. We may also say that the schedule is A-ambiguous^p for n.

As the following results show, Theorems 7.3.5 and 7.3.6 are still conserved for $p = 2$:

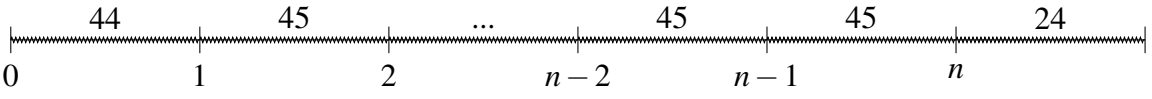
Theorem 8.2.3. *For any $n \geq 0$, there exists a schedule $f : \{0,1\} \to [0,168] \cap \mathbb{Q}$ of length 2 such that f is A-ambiguous^2 for n.*

Proof. For $n = 0$, the same schedule that was used in Theorem 7.3.5 works.

For $n = 1$, take the schedule $f : \{0,1\} \to [0,168] \cap \mathbb{Q}$ such that $f(0) = 45$ and $f(1) = 44$. Take also the schedule $h : \{0,1\} \to [0,168] \cap \mathbb{Q}$ such that $h(0) = 45$ and $h(1) = 24$.



For $n > 1$, take $f : \{0,1\} \to [0,168] \cap \mathbb{Q}$ such that $f(0) = 44$ and $f(1) = 45$ and $.h : \{0,...,n\} \to [0,168] \cap \mathbb{Q}$ such that $h(n) = 24$ and $h(i) = 45$ for all $i \neq n$.

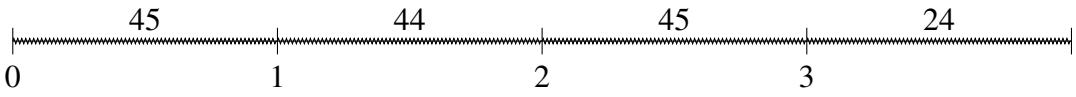


□

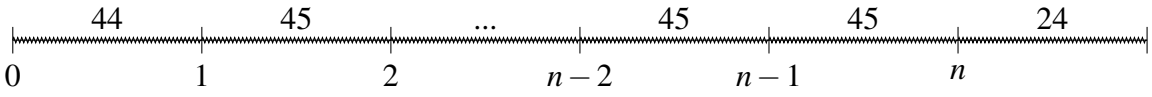
Theorem 8.2.4. *For any $n \geq 0$, there exists a schedule $f : \{0,1\} \to [0,168] \cap \mathbb{Q}$ of length 2 such that f is P-ambiguous[2] for n.*

Proof. For $n = 0$, the same schedule that was used in Theorem 7.3.6 works.

For $n = 1$, take the schedule $f : \{0,1\} \to [0,168] \cap \mathbb{Q}$ such that $f(0) = 45$ and $f(1) = 24$. Take also the schedule $h : \{0,1\} \to [0,168] \cap \mathbb{Q}$ such that $h(0) = 45$ and $h(1) = 44$.



For $n > 1$, take $f : \{0,1\} \to [0,168] \cap \mathbb{Q}$ such that $f(0) = 45$ and $f(1) = 24$ and $h : \{0,...,n\} \to [0,168] \cap \mathbb{Q}$ such that $h(0) = 44$ and $h(i) = 45$ for all $i \neq 0$.



□

However, for $p = 1$, the situation changes. In this case, the implication span of a RWRP is only a week: if it is to be maintained, a new RWRP must appear within the period for compensation. But, that would mean that there is some RWRP in the week right after the original RWRP. Impossible, since there cannot be two consecutive reductions. The only possible case is when $n = 0$:

Theorem 8.2.5. *Let $n \geq 0$. There exists a schedule $f : \{0,1\} \to [0,168] \cap \mathbb{Q}$ of length 2 such that f is A-ambiguous[1] for n if and only if $n = 0$.*

Proof. Let $n = 0$. Take $f : \{0,1\} \to [0,168] \cap \mathbb{Q}$ such that $f(0) = 45$ and $f(1) = 44$ and $h\{0\} \to [0,168] \cap \mathbb{Q}$ such that $h(0) = 45$.

Assume it was legal[1]: there exists some interpretation^p $(\widetilde{f^\frown h}, g)$ such that conditions i)-iv) from Definition 8.2.1 are fulfilled. It is clear that $\widetilde{f^\frown h}(1) < 45$ and, therefore, it must be the case that $g(1) = 2$. But then, $\widetilde{f^\frown h}(2) + (45 - \widetilde{f^\frown h}(1)) = 45$, from which we deduce that $\widetilde{f^\frown h}(2) = \widetilde{f^\frown h}(1)$. But if one was less than 45, the other must be less than 45 too, and condition ii) is violated. Contradiction, the schedule is illegal[1].

To show that $(f^\frown h)_{-1}$ is legal[1]: take the interpretation^p $((\widetilde{f^\frown h})_{-1}, g)$ such that $(\widetilde{f^\frown h})_{-1}(i) = (\widetilde{f^\frown h})_{-1}(i)$ for all $i \in \{0,1\}$ and $g(0) = 0$ and $g(1) = 2$.

For $n \neq 0$, the idea of the proof was explained before the statement of the theorem.

□

Theorem 8.2.6. *Let $n \geq 0$. There exists a schedule $f : \{0,1\} \to [0,168] \cap \mathbb{Q}$ of length 2 such that f is P-ambiguous[1] for n if and only if $n = 0$.*

In other words, some years ago, the Law did not have the problem we are studying in this chapter. It was, in some sense, a better law. Should we then go back and use the previous version? From this point on, it is not a mathematical discussion. We should check the reasons that drove the European Union to change the article: maybe they considered it was too short for a compensation period, or they opted for giving the driver more flexibility,... Once the reasons are clear, the lawyers should weigh both arguments and decide which one is a better option: giving drivers more freedom or being able to check whether an interval of time is legal. What is clear is that we cannot have our cake and eat it.

8.3 Do we have a correct ontology?

In the first draft of this thesis, we used the following versions of the definitions of *interpretation* and *legal*. *R*ead them carefully and compare them to the ones we saw in Chapter 7.

Definition 8.3.1. Given a schedule $f : \{0,...,n\} \to [0,168] \cap \mathbb{Q}$, an *interpretation* of f is a pair $(\tilde{f}, g)$, where $\tilde{f}$ is a schedule of length $n+1$ such that $\tilde{f}(i) \leq f(i)$ for all $i \in \{0,...,n\}$ and $g : \{0,...,n\} \to \{0,...,n+3\}$ such that $g(i) \geq i$ for all $i \in \{0,...,n\}$.

Definition 8.3.2. A schedule $f : \{0,...,n\} \to [0,168] \cap \mathbb{Q}$ is *legal* (or *consistent*) if there exists an interpretation $(\tilde{f}, g)$ of f such that:

i) For any $i \in \{0,...,n\}$, we have $24 \leq \tilde{f}(i)$;

ii) For any $i \in \{0,...,n-1\}$, it is not the case that $\tilde{f}(i) < 45$ and $\tilde{f}(i+1) < 45$;

iii) If there is some $i \in \{0,...,n\}$ such that $\tilde{f}(i) < 45$, then either $i+3 > n$, or there is some $j \in \{i+1, i+2, i+3\}$ such that $g(i) = j$;

iv) For all $i \in \{0,...,n\}$, we have $\tilde{f}(i) + \sum_{j \in g^{-1}(i) \setminus \{i\}} \left(45 - \tilde{f}(j)\right) = f(i)$.

A schedule is *illegal* if it is not legal.

They look like an alternative to Definitions 7.2.2 and 7.2.3. However, in that case, the title of this section would be something like "Alternative definitions" and not "Do we have a correct ontology?" Consider the following schedule, proposed by Joost J. Joosten:

Let us study it, without using any of our definitions, only the plain Law 561. There is a RWRP in Week 0 and we have to compensate 2 hours. We cannot compensate them in Week 1,

for there would be two (in fact, three) consecutive weeks with RWRP. It cannot be compensated in Week 2, since there is already a minimal rest period. And it cannot be compensated in Week 3, for there would be two consecutive weeks with RWRP. Conclusion: the schedule is illegal.

But, according to the definitions we just gave, it is legal. Let us see why. Consider the interpretation $(\tilde{f}, g)$ of f with $\tilde{f}(0) = 43$, $\tilde{f}(1) = 46$, $\tilde{f}(2) = 24$ and $\tilde{f}(3) = 45$ and $g(0) = 3$, $g(1) = 3$, $g(2) = 2$ and $g(3) = 3$.

Condition i) holds trivially: for any $i \in \{0,1,2,3\}$, we have $24 \leq \tilde{f}$.

Condition ii) also holds trivially: for any $i \in \{0,1,2,3\}$, it is not the case that $\tilde{f}(i) < 45$ and $\tilde{f}(i+1) < 45$.

For condition iii), we check it for $i = 0, 2$. In both cases, $j = 3$ works, since $j \in \{i+1, i+2, i+3\}$ and $g(i) = j$.

And now comes the key point, condition iv). It holds trivially for $i = 0, 1, 2$. For $i = 3$, the left part of the equation becomes $45 + \big((45-43) + (45-46)\big) = 45 + (2-1) = 46$, which is equal to $f(3)$.

Notice that one of the summands yielded a negative number because $\tilde{f}(j) > 45$. That means that a week that did not have a RWRP had a compensation (a negative compensation), which makes no sense. In Definition 7.2.3, this problem was excluded, since we assured that only weeks with RWRPs have a compensation.

The problem becomes evident: how can we know that our ontology is correct? We first had one that looked so, but then an example we had not thought of teared it down. Can the same happen to our current system? Is there a way to check whether we have truly formalised Law 561? The answer, as in most cases in science, is 'No'. We only have the experience in our hands, we assume a formalisation is correct if it passes an empirical exam with many cases. For double checking, we should define various equivalent formalisations and different implementations for each one of them and show with Coq that they are indeed equivalent. The purpose for this is to have different ways of expressing the same, which may make easier to spot flaws or holes and, if we cannot show an equivalence, we may be able to find small aspects in which they differ and decide which one is better for our interest.

For example, the following definition (suggested by Juan Conejero) is equivalent to the one we have worked with but, instead of using two functions $\tilde{f}$ and g, consists of only one, I, of two arguments.

Definition 8.3.3. Given a schedule $f : \{0,...,n\} \to [0,168] \cap \mathbb{Q}$, an *interpretation* of f is a function $I : \{0,...,n\} \times \{0,...,n+3\} \to [0,168] \cap \mathbb{Q}$ such that for all $i \in \{0,...,n\}$:

i) $I(i,i) \leq f(i)$;

ii) Either $I(i,j) = 0$ for all $j \neq i$ or there is a unique $j \in \{i+1, i+2, i+3\}$ such that $I(i,j) \neq 0$ and $I(i,k) = 0$ for all $k \neq i, j$.

The interpretation I tells us how many hours from Week i are in Week j. The value $I(i,i)$ corresponds to our old $\tilde{f}$ and the unique j in part ii) is $g(i)$. Let us see an example to clarify:

Example 8.3.4. Consider the following schedule f:

A possible inter*p*retation of f is the function $I : \{0,1,2,3\}x\{0,1,2,3,4,5,6\}$ such that $I(0,0) = 43$, $I(0,1) = 2$, $I(1,1) = 48$, $I(2,2) = 30$, $I(2,4) = 15$, $I(3,3) = 45$ and the rest of combinations are all zero. In other words, for Week 0, the driver rests 43 hours in Week 0 and 2 hours in Week 1; for Week 1, the driver rests 48 hours in Week 1; for Week 2, the driver rests 30 hours in Week 2 and 15 hours in Week 4 and, for Week 3, the driver rests 45 hours in Week 3. In fact, as the reader will be able to check after the next definition, this interpretation makes the schedule legal.

Finally, the notion of legality translates into:

Definition 8.3.5. A schedule $f : \{0,...,n\} \to [0,168] \cap \mathbb{Q}$ is *legal* (or consistent) if there exists an interpretation $I : \{0,...,n\}x\{0,...,n+3\} \to [0,168] \cap \mathbb{Q}$ of f such that:

i) For any $i \in \{0,...,n\}$, we have $24 \leq I(i,i)$;

ii) For any $i \in \{0,...,n-1\}$, it is not the case that $I(i,i) < 45$ and $I(i+1,i+1) < 45$;

iii) For any $i \in \{0,...,n\}$, we have $I(i,i) < 45$ if and only if there is some $j \neq i$ such that $I(i,j) \neq 0$.

iv) For all $i \in \{0,...,n\}$, we have $I(i,i) + \sum_{j \in J \setminus \{i\}} \big(45 - I(j,i)\big) = f(i)$, where $J = \{j | I(j,i) \neq 0\}$

A schedule is *illegal* if it is not legal.

We save ourselves working with two different functions by using a more compact one, but the intuitions may not be as clear. This is why we opted to work with the interpretations of the form $(\tilde{f}, g)$.

8.4 Final remarks

Note that all the main theorems in this chapter were of the form 'for all n, there is a schedule f of length 2 such that there is a schedule h of length $n+1$ such that [...]'. Does the order of the quantifiers matter? If we had taken the approach 'for all n, there is a schedule h of length $n+1$ such that there is a schedule f of length 2 such that [...]', the results would still follow. Of course, in that case, we would have to change the definitions of ambiguity to 'a schedule h of length $n+1$ is A/P-ambiguous if there is some legal schedule f of length 2 [...]'. The same proofs work, since it does not matter which of f and h we define first.

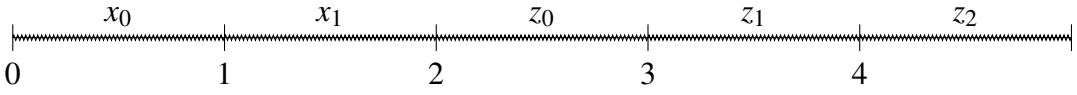
However, the third and last approach we could take: 'there is a schedule f of length 2 such that for all n there is a schedule h of length $n+1$ such that [...]' is erroneous. There is no schedule of length 2 which is A-ambiguous for all n. Remember the proof of Theorem 7.3.5: we differentiated the case $n = 2$ from the rest of the cases. This was because the general

schedule we defined in the lemmas did not directly work for $n = 2$. But, maybe we did not look hard enough and there is some f that works in each case? The answer is, as we said a couple of lines above, 'No'. We present the main ideas for the proof of the following theorem. Nonetheless, for P-ambiguity, we can find a schedule of length 2 which is P-ambiguous for all n. Please note that the case $n = 0$ was a very special one, as it used a different tactic, and we cannot aim to include it in these general results. Hence, when we say 'for all n' here, we refer to 'for all $n \geq 1$'.

Theorem 8.4.1. *There does not exist a schedule $f : \{0,1\} \to [0,168] \cap \mathbb{Q}$ of length 2 such that, for any $n \geq 1$, f is A-ambiguous for n.*

Proof. Assume there is such a schedule f. In particular, it is A-ambiguous for $n = 1$ and $n = 2$. For $n = 1$, the resulting schedule (after appending h) looks like the following picture:

And for $n = 3$, the picture is:



The first two weeks correspond to our initial schedule f and the last two and three weeks, respectively, to a fitting h. By assumption, this whole schedule is illegal, but f, h and $(f^\frown h)_{-1}$ are all legal. The illegality must occur because of conditions iii) or iv) in Definition 7.2.3, or otherwise, one of the other schedules would be illegal as well. If both $x_0, x_1 \geq 45$, then we can always find an interpretation to make it legal (or, again, otherwise, some of f, h or $(f^\frown h)_{-1}$ would be illegal). Hence, $x_0 < 45$ or $x_1 < 45$ (not both because of condition iii)).

Consider the case $n = 1$. If $x_1 < 45$, the period of compensation has not ended yet. Thus, we can find an interpretation that makes the schedule legal, making $g(1) = 4$. Contradiction, it must be the case that $x_0 < 45$.

Consider now the case $n = 2$. We prove that we can always find an interpretation that makes $f^\frown h$ legal, unless $(f^\frown h)_{-1}$ is also illegal. Since $x_0 < 45$, then for any interpretation $\widetilde{f^\frown h}(0) < 45$. Assume there is some $j \in \{1,2,3\}$ such that $g(0) = j$. If $j = 1$, then $\widetilde{f^\frown h}(1) \geq 45$ (or otherwise there would be two consecutive weeks with RWRPs). But then, we have 'closed' the RWRP and, if there is some illegality, it must happen because h is illegal, which is not. If $j = 2,3$ and $\widetilde{f^\frown h}(j) \geq 45$, the reduction is closed and the schedule is legal, unless h is illegal. But, if $\widetilde{f^\frown h}(j) < 45$, then a new reduction opens. But, then $j + 3 > 4$ and there is no illegality, again unless h is illegal. Therefore, the only possibility for $f^\frown h$ to be illegal is that there is no $j \in \{1,2,3\}$ such that $g(0) = j$. But, in that case, $(f^\frown h)_{-1}$ would also *b*e illegal. Contradiction.

Hence, it cannot be neither $x_0 < 45$ nor $x_1 < 45$, which is a contradiction. There is no such an schedule f.

□

Theorem 8.4.2. *There exists a schedule* $f : \{0,1\} \to [0,168] \cap \mathbb{Q}$ *of length* 2 *such that, for any* $n \geq 1$*,* f *is P-ambiguous for n.*

Proof. The schedule we used in the proof of Theorem 7.3.6 works: $f : \{0,1\} \to [0,168] \cap \mathbb{Q}$ with $f(0) = 24$ and $f(1) = 45$.

□

BLANK: *(of a surface or background) unrelieved by decorative or other features; bare, empty, or plain.*

Part III

The practical part

Blank page syndrome. Never again.

Chapter 9

Getting our hands on Coq: a formalisation

9.1 Step 1: schedules

As the title of the chapter hints, we are going to write some Coq co*d*e in order to formalise the ontology we defined in Chapter 7.

Before anything else, like in most programming languages, we need to import some libraries, so our code must start like this.

```
Require Import PeanoNat Even NAxioms.
Require Import ZArith.
```

These two libraries have results on arithmetic and natural numbers.

We have to go back to the definitions and formalise them one by one. The first one we encounter, and the most basic notion, is *schedule*:

Definition 9.1.1. A *schedule* is a function $f : \{0,...,n\} \to [0,168] \cap \mathbb{Q}$, for some $n \in \mathbb{N}$. We say that $n+1$ is the *length* of the schedule.

For simplicity, we may assume that the type of Time is nat. The proof works fine when we have rational numbers, but makes the Coq proofs more difficult. In principle, since the tachograph records seconds and not fractions of seconds, the type nat suffices. In the previous chapters, we worked with hours instead of seconds because "45 hours" is more easy to understand than "162000 seconds". Hence, we should have all our periods expressed in seconds. The problem is that this causes a Coq overflow, so we will work with hours expressed as natural numbers.

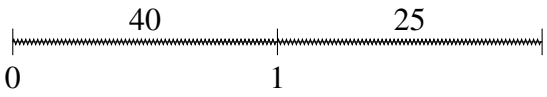
Each schedule has a number associated with it: the length. If we define a schedule as a function from the naturals to the naturals, it might take some number as input that does not belong to the interval, i.e. that is greater or equal to the length. For this reason, we need to check that the input is correct and we do so by providing a proof that it is less than the length.

We use the command *Record* which lets us write lists of functions in a recursive way.

```
Record schedule := {
length : nat;
f (k : nat) (proof : Is_true (k <? length)) : nat
}.
```

That is, a schedule consists of a length $n+1$ and a function $f : \{0,...,n\} \to \mathbb{N}$.

For example, we can define the schedule:




```
Definition a_schedule := {|
  length := 2;
  f := fun (k : nat) (proof : Is_true (k <? 2)) =>
    match k with
    | 0 => 40
    | _ => 25
    end
|}.
```

We can easily get both the length and the f of the schedule by writting *length a_schedule* and *f a_schedule*. We can alternatively write *a_schedule.(length)* and *a_schedule.(f)*.

We need to know how to prove that a number is less than another, because whenever we use a schedule, a proof of such type is required. Since we used the boolean inequality $<?$, it is very easy.

```
Lemma less_0_1 : Is_true (0 <? 1).
Proof.
  exact I.
Qed.
```

The only thing Coq asks us to prove it *True*, which has a proof by the name *I*. Proofs that simple can be solved by Coq on its own. We only need to write the tactic *auto* and the program works for us. We shall use this tactic a couple of times in this chapter when we feel too lazy to write the whole proof.

One must not confuse this proof with this one, which can also be solved by *auto*:

```
Lemma lt_0_1 : 0 < 1.
Proof.
  unfold lt.
  apply le_n.
Qed.
```

Here the inequality is the regular one and first we must unfold its definition. After the *unfold lt* line in the proof, the goal is $1 <= 1$, which is proved by applying a theorem called *le_n*, which refers to the reflexivity of $<=$.

In general, proving *Is_true(n<?m)* takes only one step, *exact I* (of course, if it is true that $n < m$). But for showing $n < m$, we need to change the scheme:

```
Lemma lt_0_3 : 0 < 3.
Proof.
  unfold lt.
  apply le_S.
  apply le_S.
  apply le_n.
Qed.
```

The *le_S* command makes the goal reduce in 1 unit: from $1 <= 3$ to $1 <= 2$. Repeatedly applying it, we end up with $1 <= 1$, which we already know how to solve. However, this proof can be shortened:

```
Lemma lt_0_3 : 0 < 3.
Proof.
  repeat (apply le_S; try (apply le_n)).
Qed.
```

With this cleared up, we can get, for example, the value of $f(0)$ with *Compute (a_schedule.(f) 0 (less_0_2))*.

Another example of schedule, which we shall use later for illustrating interpretations, is:

```
Definition b_schedule := {|
  length := 4;
  f := fun (k : nat) (proof : Is_true(k <? 4)) =>
    match k with
    | 0 => 60
    | 1 => 50
    | 2 => 45
    | _ => 35
    end
|}.
```

9.2 Step 2: interpretations

And now that we know how to write schedules and retrieve their lengths and the duration of the rest periods, we proceed to formalise what an interpretation is.

Definition 9.2.1. Given a schedule $f : \{0,...,n\} \to [0,168] \cap \mathbb{Q}$, an *interpretation* of f is a pair $(\tilde{f}, g)$, where $\tilde{f}$ is a schedule of length $n+1$ such that $\tilde{f}(i) \leq f(i)$ for all $i \in \{0,...,n\}$ and $g : \{0,...,n\} \to \{0,...,n+3\}$ such that $g(i) \in \{i, i+1, i+2, i+3\}$ for all $i \in \{0,...,n\}$.

Instead of writing $\tilde{f}$, we shall use f' in Coq. An interpretation of f is a pair (f', g) such that the functions fulfill some properties. In particular, f' is another schedule.

```
Record inter := {
  f' : schedule;
  g (k : nat) (proof : Is_true(k <? length f')) : nat
}.
```

Note that we had to do the same trick for assuring that the input that the function g gets belongs to the interval: provide a proof that the value is indeed less than the length of the schedule. However, our definition is not yet complete. We say that an *inter* $I = (f', g)$ interprets a schedule f if the lengths of f and f' match, $f'(i) \leq f(i)$ for all i and $i \leq g(i) \leq i+3$ for all i.

```
Definition interprets (s:schedule) (I:inter): Prop :=
   eq (length s) (length (f' I)) /\
   (forall k:nat, forall proof1:Is_true(k<?length s),
        forall proof2:Is_true(k<?length I.(f')),
        (Is_true((f I.(f') k (proof2)) <=? f s k (proof1)))) /\
   (forall k:nat, forall proof:Is_true(k<?length I.(f')),
        (Is_true(k <=? I.(g) k (proof)) /\ Is_true(I.(g) k (proof) <=? k+3)))
.
```

When some s and some I fulfill *interprets s I*, we say that I interprets s for short.

Let us see an example of interpretation in Coq. We are going to find one for the *b_schedule* of the examples. First, we define a new schedule:

```
Definition b_interf := {|
  length := 4;
  f := fun (k : nat) (proof : Is_true(k <? 4)) =>
    match k with
    | 0 => 60
    | 1 => 30
    | 2 => 15
    | _ => 35
    end
|}.
```

And then we properly define the interpretation.

```
Definition b_interpretation := {|
  f' := b_interf;
  g := fun (k : nat) (proof : Is_true(k <? 4)) =>
```

```
    match k with
    | 0 => 1
    | 1 => 1
    | 2 => 3
    | _ => 4
    end
|}.
```

At a glance, we see that it is indeed an interpretation of *b_schedule*, but Coq does not. We have to formally prove it:

```
Lemma b_interpretation_interprets_b : interprets b_schedule b_interpretation.
Proof.
  unfold interprets.
  simpl (length b_schedule).
  simpl (length (f' b_interpretation)).
  refine (conj _ _).

  exact (eq_refl 4).
```

After performing some simplifications, we close the first subgoal: proving that the lengths are equal. At this point the state of proof is:

```
1 subgoal
______________________________________(1/1)
(forall (k : nat) (proof1 proof2 : Is_true (k <? 4)),
 Is_true (f (f' b_interpretation) k proof2 <=? f b_schedule k proof1)) /\
(forall (k : nat) (proof : Is_true (k <? 4)),
 Is_true (k <=? g b_interpretation k proof) /\
 Is_true (g b_interpretation k proof <=? k + 3))
```

Let us break the conjunction into two subgoals, and start solving one. For this, we introduce a variable k and check, beginning a proof by cases, what happens when $k = 0$.

```
  refine (conj _ _).
  intros k.
  case k.
```

The goals now are:

```
3 subgoals
k : nat
______________________________________(1/3)
forall proof1 proof2 : Is_true (0 <? 4),
Is_true (f (f' b_interpretation) 0 proof2 <=? f b_schedule 0 proof1)
______________________________________(2/3)
forall (n : nat) (proof1 proof2 : Is_true (S n <? 4)),
```

```
Is_true (f (f' b_interpretation) (S n) proof2 <=? f b_schedule (S n) proof1)
______________________________________(3/3)
forall (k : nat) (proof : Is_true (k <? 4)),
Is_true (k <=? g b_interpretation k proof) /\
Is_true (g b_interpretation k proof <=? k + 3)
```

The fist case is straightforward, given that $60 \leq 60$. We just need to perform some simplifications in order for Coq to recognise that it can solve it.

```
  intros.
  simpl.
  tauto.
```

The Coq status right now is:

```
2 subgoals
k : nat
______________________________________(1/2)
forall (n : nat) (proof1 proof2 : Is_true (S n <? 4)),
Is_true (f (f' b_interpretation) (S n) proof2 <=? f b_schedule (S n) proof1)
______________________________________(2/2)
forall (k : nat) (proof : Is_true (k <? 4)),
Is_true (k <=? g b_interpretation k proof) /\
Is_true (g b_interpretation k proof <=? k + 3)
```

Here we need to keep checking more cases. We are proving that the statement holds for $0, 1, 2$ and 3. For $k > 3$, we will use a different strategy. Up to $k = 3$, the code is analogous to the case we just saw:

```
  intros n.
  case n.
  intros.
  simpl.
  tauto.

  intros n0.
  case n0.
  intros.
  simpl.
  tauto.

  intros n1.
  case n1.
  intros.
  simpl.
  tauto.
```

Finally, the goals are:

```
2 subgoals
k, n, n0, n1 : nat
______________________________________(1/2)
forall (n2 : nat) (proof1 proof2 : Is_true (S (S (S (S n2))) <? 4)),
Is_true
  (f (f' b_interpretation) (S (S (S (S n2)))) proof2 <=?
   f b_schedule (S (S (S (S n2)))) proof1)
______________________________________(2/2)
forall (k : nat) (proof : Is_true (k <? 4)),
Is_true (k <=? g b_interpretation k proof) /\
Is_true (g b_interpretation k proof <=? k + 3)
```

That is, when $k \geq 4$. But it makes to sense no give a schedule a number greater than its length. Precisely, we added the *Is_true* condition to the definition to prevent this. Coq solves this problem for us with just one tactic:

```
  contradiction.
```

The remaining goal, *forall (k : nat) (proof : Is_true (k <? 4)), Is_true (k <=? g b_interpretation k proof) / Is_true (g b_interpretation k proof <=? k + 3)*, can be proved in a totally analogous way:

```
  intros k.
  case k.
  intros.
  simpl.
  tauto.

  intros n.
  case n.
  intros.
  simpl.
  tauto.

  intros n0.
  case n0.
  intros.
  simpl.
  tauto.

  intros n1.
  case n1.
  intros.
  simpl.
  tauto.
```

```
  contradiction.

Qed.
```

And this completes our proof that *b_interpretation* interprets *b_schedule*.

The schedule of this interpretation is obviously illegal, since they are three RWRPs in a row. But *b_schedule* is clearly legal, so there is some 'good' interpretation. Let us define now a second interpretation that we will use for proving the legality of *b_schedule* (of course, after defining what legality means in Coq).

```
Definition b_interf2 := {|
  length := 4;
  f := fun (k : nat) (proof : Is_true(k <? 4)) =>
    match k with
    | 0 => 60
    | 1 => 30
    | 2 => 45
    | _ => 35
    end
|}.

Definition b_interpretation2 := {|
  f' := b_interf2;
  g := fun (k : nat) (proof : Is_true(k <? 4)) =>
    match k with
    | 0 => 0
    | 1 => 4
    | 2 => 2
    | _ => 4
    end
|}.
```

An easy way to show that *b_interpretation* interprets *b_schedule* is by copying and pasting the proof of lemma *b_interpretation_interprets_b*.

9.3 Step 3: legality

Let us refresh the definition of a *legal schedule* from Chapter 7:

Definition 9.3.1. A schedule $f : \{0,...,n\} \to [0,168] \cap \mathbb{Q}$ is *legal* (or consistent) if there exists an interpretation $(\tilde{f}, g)$ of f such that:

i) For any $i \in \{0,...,n\}$, we have $24 \leq \tilde{f}(i)$;

ii) For any $i \in \{0,...,n-1\}$, it is not the case that $\tilde{f}(i) < 45$ and $\tilde{f}(i+1) < 45$;

iii) For any $i \in \{0,...,n\}$, we have $\tilde{f}(i) < 45$ if and only if $g(i) > i$.

iv) For all $i \in \{0,...,n\}$, we have $\tilde{f}(i) + \sum_{j \in g^{-1}(i) \setminus \{i\}} \left(45 - \tilde{f}(j)\right) = f(i)$.

A schedule is *illegal* if it is not legal.

The problem in that definition is the finite sum that appears in Condition iv). Depending on the case, it may consists of one summand, two, or even none. So we need to clearly tell Coq what we want. Luckily for us, Coq uses the natural truncated sum. Why is it lucky for us? We will see in a minute.

If we perform *Compute 2-40*, Coq outputs 0. But if we do *Compute 2-40+1*, the result is 1. In other words, Coq reads from left to right and keeps simplifying things. Let us try a more complicated one: what happens if we ask Coq the result of $58-42-(58-42-1)$? First, $58-42=16$ and $58-42-1=15$, so Coq does $16-15=1$, nothing special. What about $58-67-(58-67-1)$? We have that $58-67=0$ and $58-67-1=0-1=0$, so $0-0=0$. Does the reader see how to use this in our benefit?

Imagine that we have a set, $\{20,105,43,67,12,3,75\}$ for instance, and we want to sum all the numbers in the set that are below 58. The easiest way: we could look at each one and determine which numbers we want and sum them. But what if we do not know which numbers are in the set, they are variables $\{i,j,k,...\}$? Would it not be perfect to have a factor that equals 0 or 1 depending on if the number is below 58? According to the paragraph above, $58-i-(58-i-1)$ does the job. Therefore, the solution to our toy problem for the set $\{i,j\}$ is $(58-i-(58-i-1))*i+(58-j-(58-j-1))*j$.

How about summing up all the elements of a set that are *equal* to 58? We can perform this trick twice with a slight modification and each summand is of the form $(58-i-(58-i-1))*(i-57-(i-57-1))*i$. The whole factor $(58-i-(58-i-1))*(i-57-(i-57-1))$ is null if either $i<58$ or $58<i$ and is only 1 when $i=58$.

Returning to our formalisation, we want to know which numbers from $\{g(i-3),g(i-2),g(i-1)\}$ are equal to i. In other words, which $\{i-3,i-2,i-1\}$, if any, is an antiimage of i via g. After we know this, we want to sum $45-f'(j)$ for all those j. Thus, we can perform:

$$\sum_{j=i-3}^{i-1} (g(j)-i-(g(j)-i-1))*(i-g(j)-1-(i-g(j)-2))*(45-f'(j)).$$

However, if $i=0$, there are no possible j's before i; if $i=1$, only $j=0$ could belong to the set of antiimages; and if $i=2$, we should only consider $j=0,1,2$. For this reason, we define the following function depending on the case. It is a formalisation in Coq of Condition iv) using the formula we just saw.

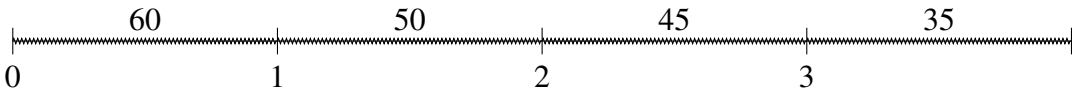
```
Definition legal_sum := fun (I:inter) (k:nat)
     (proof:Is_true(k<?length I.(f'))) (proof1:Is_true(k-1<?length I.(f')))
     (proof2:Is_true(k-2<?length I.(f'))) (proof3:Is_true(k-3<?length I.(f')))
     =>
  match k with
```

```
  | 0 => f I.(f') k (proof)
  | 1 => (f I.(f') k (proof) +
     (I.(g) (k-1) proof1 -k - (I.(g) (k-1) proof1 -k-1))*
       (k -I.(g) (k-1) proof1 -1 - (k-I.(g) (k-1) proof1 -2))*
       (45-f I.(f') (k-1) (proof1)))
  | 2 => (f I.(f') k (proof) +
     (I.(g) (k-2) proof2 -k - (I.(g) (k-2) proof2 -k-1))*
       (k -I.(g) (k-2) proof2 -1 - (k-I.(g) (k-2) proof2 -2))*
       (45-f I.(f') (k-2) (proof2)) +
     (I.(g) (k-1) proof1 -k - (I.(g) (k-1) proof1 -k-1))*
       (k -I.(g) (k-1) proof1 -1 - (k-I.(g) (k-1) proof1 -2))*
       (45-f I.(f') (k-1) (proof1)))
  | _ => (f I.(f') k (proof) +
     (I.(g) (k-3) proof3 -k - (I.(g) (k-3) proof3 -k-1))*
       (k -I.(g) (k-3) proof3 -1 - (k-I.(g) (k-3) proof3 -2))*
       (45-f I.(f') (k-3) (proof3)) +
     (I.(g) (k-2) proof2 -k - (I.(g) (k-2) proof2 -k-1))*
       (k -I.(g) (k-2) proof2 -1 - (k-I.(g) (k-2) proof2 -2))*
       (45-f I.(f') (k-2) (proof2)) +
     (I.(g) (k-1) proof1 -k - (I.(g) (k-1) proof1 -k-1))*
       (k -I.(g) (k-1) proof1 -1 - (k-I.(g) (k-1) proof1 -2))*
       (45-f I.(f') (k-1) (proof1)))
  end
.
```

And with this powerful definition in our hands, it is trivial to write a formalisation of *legality*:

```
Definition legal (s:schedule): Prop :=
   exists I:inter, (interprets s I /\
   (forall k:nat, forall proof:Is_true(k<?length I.(f')),
      (Is_true(24 <=? f I.(f') k (proof)))) /\
   (forall k:nat, forall proof1:Is_true(k<?length I.(f')),
      forall proof2:Is_true(k+1<?length I.(f')),
      (Is_true(45 <=? f I.(f') k (proof1)) \/
      Is_true(45 <=? f I.(f') (k+1) (proof2)))) /\
   (forall k:nat, forall proof:Is_true(k<?length I.(f')),
      (Is_true(45 <=? f I.(f') k (proof)) <->
      Is_true((I.(g) k (proof)) =? k))) /\
   (forall k:nat, forall proof:Is_true(k<?length I.(f')),
      forall proofs:Is_true(k<?length s),
      forall proof1:Is_true(k-1<?length I.(f')),
      forall proof2:Is_true(k-2<?length I.(f')),
      forall proof3:Is_true(k-3<?length I.(f')),
      (Is_true(legal_sum I k  proof proof1 proof2 proof3 =? f s k (proofs))))
   )
.
```

Let us see an example. Remember *b_schedule* from the previous section?



We show in the next lemma that it is a legal schedule. The witness we are using is *b_interpretation2*, also defined in the previous section:

```
Definition b_interf2 := {|
  length := 4;
  f := fun (k : nat) (proof : Is_true(k <? 4)) =>
    match k with
    | 0 => 60
    | 1 => 30
    | 2 => 45
    | _ => 35
    end
|}.

Definition b_interpretation2 := {|
  f' := b_interf2;
  g := fun (k : nat) (proof : Is_true(k <? 4)) =>
    match k with
    | 0 => 0
    | 1 => 4
    | 2 => 2
    | _ => 4
    end
|}.
```

The statement of the lemma is thus the following:

```
Lemma b_schedule_is_legal : legal b_schedule.
```

The first step is to unfold the definition of *legal*, so the 4-term conjunction becomes visible. Before performing any other step, it is convenient to simplify things:

```
Proof.
  unfold legal.
  simpl (length b_schedule).
```

The current goal is:

```
1 subgoal
______________________________________(1/1)
exists I : inter,
```

```
interprets b_schedule I /\
(forall (k : nat) (proof : Is_true (k <? length (f' I))),
 Is_true (24 <=? f (f' I) k proof)) /\
(forall (k : nat) (proof1 : Is_true (k <? length (f' I)))
   (proof2 : Is_true (k + 1 <? length (f' I))),
 Is_true (45 <=? f (f' I) k proof1) \/
    Is_true (45 <=? f (f' I) (k + 1) proof2)) /\
(forall (k : nat) (proof : Is_true (k <? length (f' I))),
 Is_true (45 <=? f (f' I) k proof) <-> Is_true (g I k proof =? k)) /\
(forall (k : nat) (proof : Is_true (k <? length (f' I)))
   (proofs : Is_true (k <? 4)) (proof1 : Is_true (k - 1 <? length (f' I)))
   (proof2 : Is_true (k - 2 <? length (f' I)))
   (proof3 : Is_true (k - 3 <? length (f' I))),
 Is_true (legal_sum I k proof proof1 proof2 proof3 =? f b_schedule k proofs))
```

In order to take care of the existential quantifier, we need to use the tactics *pose* and *refine (ex_intro)* for providing a witness. And once the existential disappears, we can break the conjunction into smaller parts.

The first subgoal is to prove that *b_interpretation* is indeed an interpretation of *b_schedule*. But we had already shown this result in a lemma.

```
pose (witness := b_interpretation2).
refine (ex_intro _ witness _).
refine (conj _ _).
apply b_interpretation2_interprets_b.
```

We must break the conjunction again, to prove Condition ii). The proof of this subgoal is similar to some that we have already seen:

```
refine (conj _ _).
intros k.
case k.
intros.
simpl.
tauto.

intros n.
case n.
intros.
simpl.
tauto.

intros n0.
case n0.
intros.
simpl.
tauto.
```

```
  intros n1.
  case n1.
  intros.
  simpl.
  tauto.

  contradiction.
```

We prove the result for each of $k = 0, 1, 2, 3$ and show that $k > 3$ leads to a contradiction. Now, we have to break again the conjunction and show that Condition iii) holds. The proof uses exactly the same lines of code as the one we just saw, so we are not repeating them. Identically, we prove Conditions iii) and iv) and finish our lemma.

Let us now see an example of a proof of illegality. We define *c_schedule*:

Which in Coq is defined as:

```
  Definition c_schedule := {|
  length := 5;
  f := fun (k : nat) (proof : Is_true(k <? 5)) =>
    match k with
    | 0 => 60
    | 1 => 20
    | 2 => 45
    | 3 => 45
    | _ => 35
    end
|}.
```

First, we prove a lemma that will be the core of the illegality: for all interpretations of *c_schedule*, $f'(1) \leq 20$.

```
  Lemma c_illegal_reason : forall I:inter,
     (interprets c_schedule I -> forall proof:Is_true(1<?length I.(f')),
     Is_true((f I.(f') 1 proof <=? 20))).
```

The first steps of the proof should be understandable by now:

```
Proof.
  intros I.
  unfold interprets.
  simpl.
```

```
  intros interprets.
  destruct interprets.
  destruct H0.
```

We have introduced an *inter* I, simplified some terms and assumed I interprets the schedule. Since the definition of *interprets* is a conjunction of three terms, we have destructed it in two steps in order to have three simpler hypotheses. Right now, the goal is:

```
1 subgoal
I : inter
H : 5 = length (f' I)
H0 : forall k : nat,
     Is_true (k <? 5) ->
     forall proof2 : Is_true (k <? length (f' I)),
     Is_true
       (f (f' I) k proof2 <=?
        match k with
        | 0 => 60
        | 1 => 20
        | 2 => 45
        | 3 => 45
        | S (S (S _))) => 35
        end)
H1 : forall (k : nat) (proof : Is_true (k <? length (f' I))),
     Is_true (k <=? g I k proof) /\ Is_true (g I k proof <=? k + 3)
______________________________________(1/1)
forall proof : Is_true (1 <? length (f' I)), Is_true (f (f' I) 1 proof <=? 20)
```

The next step is clear: *intros proof*. Now, we would like to use the extra information that *Is_true(1<?length c_schedule)*. Coq allows us to do so with the *assert* tactic. It creates a new subgoal, precisely *Is_true(1<?length c_schedule)* and, when we have already proved it, we can use it in the main proof.

```
intros proof.
assert (Is_true(1<?length c_schedule)).
  simpl.
  auto.
```

This extra proof was pretty easy, we only needed to simplify and ask Coq to prove it for us. Next, we need another extra result:

```
assert (Is_true(f (f' I) 1 proof <=? f c_schedule 1 H2)).
      apply H0.
      auto.
```

The *H2* in the statement refers to the first assertion we made, which serves as an input for making *f c_schedule 1* work. The first step, *apply H0*, uses the hypothesis *H0*, which

corresponds to the condition of interpretation that $f'(k) \leq f(k)$ for all k. This creates the subgoal *Is_true (1 <? 5)*, easily solved with *auto*.

Finally, the after adding these new hypotheses, the goal is:

```
1 subgoal
I : inter
H : 5 = length (f' I)
H0 : forall k : nat,
     Is_true (k <? 5) ->
     forall proof2 : Is_true (k <? length (f' I)),
     Is_true
       (f (f' I) k proof2 <=?
        match k with
        | 0 => 60
        | 1 => 20
        | 2 => 45
        | 3 => 45
        | S (S (S _))) => 35
        end)
H1 : forall (k : nat) (proof : Is_true (k <? length (f' I))),
     Is_true (k <=? g I k proof) /\ Is_true (g I k proof <=? k + 3)
proof : Is_true (1 <? length (f' I))
H2 : Is_true (1 <? length c_schedule)
H3 : Is_true (f (f' I) 1 proof <=? f c_schedule 1 H2)
______________________________________(1/1)
Is_true (f (f' I) 1 proof <=? 20)
```

Since $f(1) = 20$, we only need to *apply H3* and we are done.

```
apply H3.
Qed.
```

Before we move on into the proof that *c_schedule* is illegal, let us see two small lemmas that will prove useful. They let us reach inside the black box that is *Is_true* for some of Coq tactics and get rid of it when needed.

```
Lemma Is_true1: forall n m : nat, Is_true(n <? m) <-> n < m.
Proof.
   assert (forall n m : nat, (n <? m) = true <-> n < m).
      exact Nat.ltb_lt.
   assert (forall x : bool, Is_true x -> x = true).
      exact Is_true_eq_true.
   intros n m.
   specialize (H0 (n<?m)).
   specialize (H n).
   specialize (H m).
```

```
   destruct H as [H H'].
   intuition.

Qed.
```

This lemma shows that for all natural numbers n, m, the expressions *Is_true(n<?m)* and $n<m$ are equivalent. For this, we use two results: *Nat.ltb_lt* and *Is_true_eq_true*.

Nat.ltb_lt is a lemma whose statement is *forall n m : nat, (n <? m) = true <-> n < m*. On the other hand, *Is_true_eq_true* is *forall x : bool, Is_true x -> x = true*. So, our first step is to have these results close by asserting and providing their proofs.

Next, we introduce n and m and specify that we want to have the first assertion for these n and m and the second one, for $x = (n <?m)$. This simplifies the hypotheses and, after breaking the biconditional from *Nat.ltb_lt* into two conditionals, we are able to use *intuituion* and let Coq finish the proof. The tactic *intuition* is another of Coq's auto tactics, like *auto*, *tauto* and *firstorder*.

The second lemma we mentioned is proved in an analogous way, but using *Nat.leb_le* instead of *Nat.ltb_lt*. It is the same lemma but with an $\leq$ in place of $<$.

```
Lemma Is_true2: forall n m : nat, Is_true(n <=? m) <-> n <= m.
Proof.
   assert (forall n m : nat, (n <=? m) = true <-> n <= m).
      exact Nat.leb_le.
   assert (forall x : bool, Is_true x -> x = true).
      exact Is_true_eq_true.
   intros n m.
   specialize (H0 (n<=?m)).
   specialize (H n).
   specialize (H m).
   destruct H as [H H'].
   intuition.
Qed.
```

And now we can show *c_schedule*'s illegality! But, be warned, it is not a short proof.

```
Lemma c_schedule_is_illegal : ~(legal c_schedule).
Proof.
  unfold not.
  unfold legal.
  intros exists_I.
  destruct exists_I as [witness proof_of_exists_I].
  destruct proof_of_exists_I as [interp  HH].
  destruct HH as [H H0C].
  destruct H0C as [H0 H1C].
  destruct H1C as [H1 H2].
```

The first tactic is for rewriting the negation as *legal c_schedule -> False*. Then, we unfold the definition of *legal* and introduce a proof that there exists a legal interpretation. We are left to proving *False*. Since we assume there is such a proof of the existence of an interpretation, we can introduce a witness for getting rid of the existential. Now, *legal* involves many conjunctions, so we break it into smaller pieces with *destruct*.

This is the moment for bringing *c_illegal_reason* into this proof:

```
assert (forall I:inter, (interprets c_schedule I ->
   forall proof:Is_true(1<?length I.(f')), Is_true((f I.(f') 1 proof <=? 20)))).
     apply c_illegal_reason.
```

Let us take a look at the goal:

```
1 subgoal
witness : inter
interp : interprets c_schedule witness
H : forall (k : nat) (proof : Is_true (k <? length (f' witness))),
    Is_true (24 <=? f (f' witness) k proof)
H0 : forall (k : nat) (proof1 : Is_true (k <? length (f' witness)))
       (proof2 : Is_true (k + 1 <? length (f' witness))),
     Is_true (45 <=? f (f' witness) k proof1) \/
     Is_true (45 <=? f (f' witness) (k + 1) proof2)
H1 : forall (k : nat) (proof : Is_true (k <? length (f' witness))),
     Is_true (45 <=? f (f' witness) k proof) <->
     Is_true (g witness k proof =? k)
H2 : forall (k : nat) (proof : Is_true (k <? length (f' witness)))
       (proofs : Is_true (k <? length c_schedule))
       (proof1 : Is_true (k - 1 <? length (f' witness)))
       (proof2 : Is_true (k - 2 <? length (f' witness)))
       (proof3 : Is_true (k - 3 <? length (f' witness))),
     Is_true
       (legal_sum witness k proof proof1 proof2 proof3 =? f c_schedule k proofs)
H3 : forall I : inter,
     interprets c_schedule I ->
     forall proof : Is_true (1 <? length (f' I)),
     Is_true (f (f' I) 1 proof <=? 20)
______________________________________(1/1)
False
```

We have many hypotheses, but we still need some more.

```
assert (forall I:inter, (interprets c_schedule I ->
  forall proof:Is_true(1<?length I.(f')), Is_true((f I.(f') 1 proof <? 24)))).
   intros.
   assert (Is_true(f I.(f') 1 proof <=? 20) -> Is_true(f I.(f') 1 proof <? 24)).
      intros.
```

```
      apply Is_true1.
      apply Is_true2 in H5.
      intuition.
   refine (H5 _).
   apply H3.
   exact H4.
```

This one is similar to *c_illegal_reason*, but less restrictive: for any interpretation of the schedule, $f'(1) < 24$. Easy to prove when we already have $f'(1) \leq 20$. The main tool here is the assertion, in which we make use of *Is_true1* and *Is_true2* from before. Once we have proved this subassertion, since our subgoal is the conclusion in that assertion, we *refine* and prove the condition *Is_true(f I.(f') 1 proof <=? 20)*. But this is almost the same as *c_illegal_reason*, here labeled as *H3*. We only need to tell Coq that *I* is an interpretation of *c_schedule*, which we assumed at the beginning of this assertion (when we did the first *intros* and Coq labeled it as *H4*.

Let us refresh the current goal after introducing a new hypothesis.

```
1 subgoal
witness : inter
interp : interprets c_schedule witness
H : forall (k : nat) (proof : Is_true (k <? length (f' witness))),
    Is_true (24 <=? f (f' witness) k proof)
H0 : forall (k : nat) (proof1 : Is_true (k <? length (f' witness)))
       (proof2 : Is_true (k + 1 <? length (f' witness))),
     Is_true (45 <=? f (f' witness) k proof1) \/
     Is_true (45 <=? f (f' witness) (k + 1) proof2)
H1 : forall (k : nat) (proof : Is_true (k <? length (f' witness))),
     Is_true (45 <=? f (f' witness) k proof) <->
     Is_true (g witness k proof =? k)
H2 : forall (k : nat) (proof : Is_true (k <? length (f' witness)))
       (proofs : Is_true (k <? length c_schedule))
       (proof1 : Is_true (k - 1 <? length (f' witness)))
       (proof2 : Is_true (k - 2 <? length (f' witness)))
       (proof3 : Is_true (k - 3 <? length (f' witness))),
     Is_true
       (legal_sum witness k proof proof1 proof2 proof3 =? f c_schedule k proofs)
H3 : forall I : inter,
     interprets c_schedule I ->
     forall proof : Is_true (1 <? length (f' I)),
     Is_true (f (f' I) 1 proof <=? 20)
H4 : forall I : inter,
     interprets c_schedule I ->
     forall proof : Is_true (1 <? length (f' I)),
     Is_true (f (f' I) 1 proof <? 24)
______________________________________(1/1)
False
```

We would like to have the corresponding *H* and *H4* for $k = 1$ and *I*=*witness*, respectively.

```
specialize (H4 witness).
specialize (H 1).
intuition.
```

The tactic *intuition* let us erase the condition *interprets c_schedule witness* from *H4*, given that it is one of the hypotheses, *interp*. And let us add a new one, we promise it is the last hypothesis we add.

```
assert (forall proof : Is_true (1 <? length (f' witness)), False).
    intros.
    specialize(H proof).
    specialize(H5 proof).
    assert (Is_true (24 <=? f (f' witness) 1 proof) ->
      ~Is_true (f (f' witness) 1 proof <? 24)).
       intros P1.
       rewrite Is_true1.
       rewrite Is_true2 in P1.
       intuition.

    intuition.
```

We introduce a proof of *Is_true (1 <? length (f' witness))* and use it in *H* and *H5*, which is the new *H4* after erasing the condition as we discussed. We then bring a result into the proof: if $24 \leq f'(1)$, then it is not the case that $f'(1) < 24$. Finally, *intuition* saves us some work.

At this moment, we have an hypothesis *Is_true (1 <? length (f' witness)) -> False* and want to prove *False*. If we show that *Is_true (1 <? length (f' witness))*, we are done. But we know that *length (f' witness)=length c_schedule* from one of the conditions of *interprets* (we use a *destruct* for getting it). And *length c_schedule=5*, so indeed *Is_true (1 <? 5)*.

```
unfold interprets in interp.
destruct interp as [interp1 interp2].
symmetry in interp1.
rewrite interp1 in H4.
simpl in H4.
apply H4.
exact I.

Qed.
```

After simplifying, *H4* becomes *True -> False*, and since our goal is *False*, using *apply H4* replaces the goal for *True*, which is provable with *exact I*. Hooray, we ended our example.

9.4 Step 4: ambiguity

In this fourth step of our formalisational quest, we will explain Coq what an ambiguous schedule is. But there is some previous work to do, as *ambiguity* relies on three concepts that we have not yet defined in Coq:

Definition 9.4.1.

1) Let f and g be two schedules of lengths $n+1$ and $m+1$, respectively. The concatenation of f and g, denoted by $f^\frown g : \{0,...,n,n+1,...,n+m+1\} \to [0,168] \cap \mathbb{Q}$, is the schedule defined as:

$$f^\frown g(i) = \begin{cases} f(i), & \text{if } i \leq n, \\ g(i-n-1), & \text{if } i > n. \end{cases}$$

2) Let $f : \{0,...,n\} \to [0,168] \cap \mathbb{Q}$ be a schedule and let $m < n+1$. We denote by f_{-m} the schedule that results after eliminating the rightmost m weeks of the schedule, that is, $f_{-m} : \{0,...,n-m\} \to [0,168] \cap \mathbb{Q}$ such that $f_{-m}(i) = f(i)$ for $i \leq n-m$.

3) Let $f : \{0,...,n\} \to [0,168] \cap \mathbb{Q}$ be a schedule and let $m < n+1$. We denote by f^{-m} the schedule that results after eliminating the leftmost m weeks of the schedule, that is, $f^{-m} : \{0,...,n-m\} \to [0,168] \cap \mathbb{Q}$ such that $f^{-m}(i) = f(i+m)$ for $i \leq n-m$.

Before we jump right in, let us say that the two last notions are fairly simpler to write in Coq than the first one.

For *concatenation*, given two schedules f and g, if we are to perform $f^\frown g(k)$, we feed the function with a proof that $k < length(f^\frown g)$ and, depending the case, we need a proof of $k < length(f)$ or a proof of $k < length(g)$. Of course, we know that if there is a proof of $f^\frown g(k)$, then it is true both $k < length(f)$ and $k < length(g)$, because the lengths are shorter. But Coq does not know it, so we must show him some result on this.

```
Lemma helper (n m k : nat) : Is_true (k <? n + m) ->
    {Is_true (k <? n)} + {Is_true (k - n <? m)}.
```

The sum that appears at the right hand side means the disjunction of the boolean values. However, we cannot prove this lemma right now. We need two preliminary results. The first one is:

```
Lemma true_to_Is_true : forall k n : nat, {(k<?n)=true} +
    {(n<=?k)=true} -> {Is_true(k<?n)} + {Is_true(n<=?k)}.
```

Can the reader imagine why we are going to use it? Because, in general, Coq works better with things like *(k<?n)=true* than with *Is_true(k<?n)*. Hence, we will have to prove at some point *Is_true(k<?n) + Is_true(n<=?k)* and, applying this lemma, we will only have to show *(k<?n)=true + (n<=?k)=true*. But, wait a minute, does *Is_true(k<?n) + Is_true(n<=?k)* remind of anything? It is an instance of *excluded middle*. Even if we are working with constructive mathematics, we can prove it for booleans in Coq.

Let us begin the proof.

```
Proof.
  intros.
  destruct H.
```

We introduce the variables *k*, *n* and the proof of *(k<?n)=true + (n<=?k)=true*. Next, we break the boolean sum in the hypothesis (remember it behaves as a disjunction). We have to prove *Is_true(k<?n) + Is_true(n<=?k)* in both cases: when *(k<?n)=true* and when *(n<=?k)=true*. But these two subgoals are easy enough for Coq to prove them on its own.

```
intuition.
intuition.
Qed.
```

As we advanced before, we are going to prove the following version of excluded middle:

```
Lemma Is_true_excluded_middle : forall k n : nat,
    {Is_true(k<?n)} + {Is_true(n<=?k)}.
Proof.
 intros k n.
 apply true_to_Is_true.
 induction n; induction k.
```

We apply the previous result, so we are left to prove *(k<?n)=true + (n<=?k)=true*. For this, we perform a double induction on *n* and *k*. This creates four subgoals:

```
4 subgoals
______________________________________(1/4)
{(0 <? 0) = true} + {(0 <=? 0) = true}
______________________________________(2/4)
{(S k <? 0) = true} + {(0 <=? S k) = true}
______________________________________(3/4)
{(0 <? S n) = true} + {(S n <=? 0) = true}
______________________________________(4/4)
{(S k <? S n) = true} + {(S n <=? S k) = true}
```

The first three are easy to prove:

```
right.
intuition.

right.
intuition.

left.
intuition.
```

The fourth case is a bit more complex. We have two induction hypotheses:

```
1 subgoal
k, n : nat
IHn : {(S k <? n) = true} + {(n <=? S k) = true}
IHk : {(k <? n) = true} + {(n <=? k) = true} ->
      {(k <? S n) = true} + {(S n <=? k) = true}
______________________________________(1/1)
{(S k <? S n) = true} + {(S n <=? S k) = true}
```

We are going to use a tactic we applied in the previous theorem, for breaking a boolean sum: *destruct*. This generates two subgoals. Besides, we transform *(S k <? S n) = true* into *S k < S n* via a theorem called *Nat.ltb_lt*. Finally, we close the first subgoal by transitivity.

```
destruct IHn.
left.
rewrite Nat.ltb_lt.
rewrite Nat.ltb_lt in e.
transitivity n.
intuition.
intuition.
```

For the last case, the goal is:

```
1 subgoal
k, n : nat
e : (n <=? S k) = true
IHk : {(k <? n) = true} + {(n <=? k) = true} ->
      {(k <? S n) = true} + {(S n <=? k) = true}
______________________________________(1/1)
{(S k <? S n) = true} + {(S n <=? S k) = true}
```

We do as before and rewrite *(n <=? S k) = true* as *n <= S k* (this time, the result we use is called *Nat.leb_le*. We know that *n <= S k* implies that either $n < Sk$ or $n = Sk$. This is exactly the result *le_lt_eq_dec*, which we use to generate two goals, one for each case.

```
rewrite Nat.leb_le in e.
destruct (le_lt_eq_dec n (S k) e).
```

If *n < S k*, we do something similar to what we have already done and let Coq prove that if *n < S k*, then *S n <= S k*.

```
right.
rewrite Nat.leb_le.
intuition.
```

If *n = S k*, it is clear that *n < S n*, so *S k < S n*. Thus, we aim to prove the left part of the boolean sum. We only need to rewrite our equality and simplify as before.

```
left.
rewrite e0.
rewrite Nat.ltb_lt.
intuition.
Qed.
```

Two lemmas proved, one to go before being able to define *concatenation*. We can now prove the *helper* lemma we stated at the beginning of this section:

```
Lemma helper (n m k : nat) : Is_true (k <? n + m) ->
      {Is_true (k <? n)} + {Is_true (k - n <? m)}.
Proof.
  intro H.
  destruct (Is_true_excluded_middle k n).
```

As usual, we introduce the proof of *Is_true (k <? n + m)* as a hypothesis. We also have *n*, *m* and *k* as parameters. Then, we use the version of excluded middle we proved to create two cases: when *Is_true (k <? n)* and when *Is_true (n <=? k)*. The extra hypothesis is labeled as *i* by Coq. The goals at this moment are:

```
2 subgoals
n, m, k : nat
H : Is_true (k <? n + m)
i : Is_true (k <? n)
______________________________________(1/2)
{Is_true (k <? n)} + {Is_true (k - n <? m)}
______________________________________(2/2)
{Is_true (k <? n)} + {Is_true (k - n <? m)}
```

The first case is straightforward, we already have the left part as a hypothesis.

```
intuition.
```

For the second one, the goal is:

```
1 subgoal
n, m, k : nat
H : Is_true (k <? n + m)
i : Is_true (n <=? k)
______________________________________(1/1)
{Is_true (k <? n)} + {Is_true (k - n <? m)}
```

We want to prove the right part and, and by rewriting each instance of *Is_true* with the lemmas *Is_true1* and *Is_true2*, we simplify it enough for Coq to be able to solve it.

```
right.
rewrite Is_true1.
rewrite Is_true2 in i.
rewrite Is_true1 in H.
intuition.
Qed.
```

And, finally, we can define *concatenation*. Let us refresh once more the definition we had:

Definition 9.4.2. Let f and g be two schedules of lengths $n+1$ and $m+1$, respectively. The concatenation of f and g, denoted by $f^\frown g : \{0,...,n,n+1,...,n+m+1\} \to [0,168] \cap \mathbb{Q}$, is the schedule defined as:

$$f^\frown g(i) = \begin{cases} f(i), & \text{if } i \leq n, \\ g(i-n-1), & \text{if } i > n. \end{cases}$$

We need the *helper* to distinguish between the two cases:

```
Definition concat := fun (f1 f2 : schedule) => {|
       length := length f1 + length f2;
       f := fun (k : nat) (proof : Is_true(k <? length f1 + length f2)) =>
          match helper (length f1) (length f2) k proof with
          | left proof1 => f f1 k proof1
          | right proof2 => f f2 (k - length f1) proof2
          end
       |}.
```

Let us see an example. We are going to concatenate the schedules we defined at the beginning of the chapter: *a_schedule* and *b_schedule*. To check that the process goes well, we compute the value of the resulting schedule in 0. For this, we need a proof that *Is_true(0 <? 6)*. We already saw how to do this:

```
Lemma less_0_6 : Is_true (0 <? 6).
Proof.
  exact I.
Qed.
```

And if we write in Coq *Compute (concat a_schedule b_schedule).(f) 0 less_0_6*, we get the desired result, 40:

```
= if helper 2 4 0 less_0_6 then 40 else 60
     : nat
```

Hmm... Indeed, 40 appears in the result, but it is quite different from what we expected. What is happening?

The problem is that Coq interprets the lemmas as black boxes and cannot access to their contents, only their statements. However, in the definition of *concatenation*, we use a lemma

and need to compute the body of the proof. For this reason, if instead of ending the proofs of this section with *Qed*, we put *Defined*, Coq will be able to retrieve the information and compute the value correctly.

Writing *Compute (concat a_schedule b_schedule).(f) 3 less_3_6* (with the correct proof that *Is_true(3 <? 6)*), yields to 30, the value that we should expect.

That was a long way for defining only the first part of Definition 7.2.5! Luckily for us, the other two are much shorter and only precise one lemma each. Moreover, the lemmas are so similar that we can use the exact same proof!

```
Lemma helper2 (n k : nat) : Is_true (k <? n -1) ->
  Is_true (k <? n).
Proof.
  intros.
  apply Is_true1.
  apply Is_true1 in H.
  intuition.
Qed.

Lemma helper3 (n k : nat) : Is_true (k <? n -1) ->
  Is_true (k+1 <? n).
Proof.
  intros.
  apply Is_true1.
  apply Is_true1 in H.
  intuition.
Qed.
```

There should not be any problem to understand these results. The key point is simplifying *Is_true* with the auxiliar lemma and then Coq can finish the proof.

Finally, we define the schedules that result after removing the first/last week of a given schedule:

```
Definition minus_one_last := fun (s : schedule) => {|
       length := length s -1;
       f := fun (k : nat) (proof : Is_true(k <? length s -1)) =>
            s.(f) k (helper2 (length s) k proof)
       |}.

Definition minus_one_first := fun (s : schedule) => {|
       length := length s -1;
       f := fun (k : nat) (proof : Is_true(k <? length s -1)) =>
            s.(f) (k+1) (helper3 (length s) k proof)
       |}.
```

We use the lemmas we just proved to show Coq that he can compute the values in the original schedule because the lengths are fine.

And we finish this section by formalising the most important concept: *ambiguity*:

Definition 9.4.3. Let f be a legal schedule of length 2. Then:

i) f has the *P-ambiguity property* (PAP), or is *P-ambiguous*, if there exists some $n \in \mathbb{N}$ and a legal schedule h of length $n+1$ such that the schedule $h^\frown f$ is illegal but $(h^\frown f)^{-1}$ is legal. We may also say that the schedule is P-ambiguous for n.

ii) f has the *A-ambiguity property* (AAP), or is *A-ambiguous*, if there exists some $n \in \mathbb{N}$ and a legal schedule h of length $n+1$ such that the schedule $f^\frown h$ is illegal but $(f^\frown h)_{-1}$ is legal. We may also say that the schedule is A-ambiguous for n.

It is easy once we already have all the definitions at our hand:

```
Definition A_ambig (s:schedule) (n:nat): Prop :=
   (Is_true(length s =? 2)) /\
   exists h:schedule, (Is_true(length h =? n+1) /\
   ~(legal (concat s h)) /\ legal(minus_one_last(concat s h))
.


Definition P_ambig (s:schedule) (n:nat): Prop :=
   (Is_true(length s =? 2)) /\
   exists h:schedule, (Is_true(length h =? n+1) /\
   ~(legal (concat h s)) /\ legal(minus_one_first(concat h s)))
.
```

And we completed our formalisation of the ontology in Chapter 7! We can conclude that writing things in Coq is not as straightforward as it is in regular mathematics, but the sense of achievement is much greater.

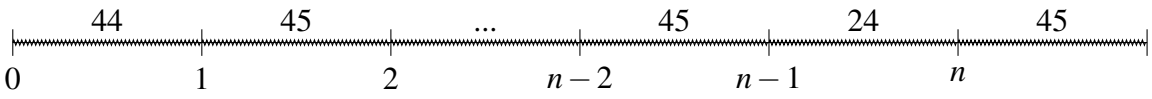
Chapter 10

Going further: the theorems

10.1 P-ambiguity

Since we stated that Coq is used to prove theorems, we will go one step further and show in this chapter the main results from Chapter 7 using these new definitions in Coq. However, due to their complexity, extension and lack of interest for a first-time approach to Coq, we have decided to ommit some of the proofs of the preliminary lemmas. If the reader is trying to reproduce our steps, we suggest using the tactic *admit* and ending the proofs with *Admitted* instead of *Qed*.

The first thing we need is to define the general schedule that we will use in the proof, as in Lemma 7.3.2.



This, in Coq, is translated as:

```
Definition main_schedule (n : nat) :=
      {|
        length := n + 3;
        f := fun (k : nat) (proof : Is_true (k <? n + 3)) =>
          if k =? 0 then
            44
          else if k =? n+1 then
            24
          else
            45
      |}.
```

We put $n+3$ as the length rather than $n+1$ because we are not going to work with the same n as in the picture. Instead, $n+1$ is going to be the length of a shorter interval, which corresponds to the initial segment of the schedule:

```
Definition start_main_schedule (n : nat) :=
      {|
        length := n + 1;
        f := fun (k : nat) (proof : Is_true (k <? n + 1)) =>
          if k =? 0 then
            44
          else
            45
      |}.
```

And the remaining part of the schedule is:

```
Definition end2_main_schedule :=
      {|
        length := 2;
        f := fun (k : nat) (proof : Is_true (k <? 2)) =>
          match k with
          | 0 => 24
          | _ => 45
          end
      |}.
```

It is possible to prove the following result, which shows that concatenating *start_main_schedule* and *end2_main_schedule* yields to *main_schedule*, as expected:

```
Lemma P_same_schedule : forall n,
      (concat (start_main_schedule n) end2_main_schedule = (main_schedule n)).
```

We will now first study Theorem 7.3.6, the one for P-ambiguity. The corresponding statements of Lemmas 7.3.2 and 7.3.4 are:

```
Lemma illegal : forall n:nat, (n>0 -> ~(legal (main_schedule n))).

Lemma minus_one_first_legal : forall n:nat,
     (legal(minus_one_first (concat(start_main_schedule n) end2_main_schedule))).
```

Let us prove the second one, *minus_one_first_legal*. Since we want to prove that an schedule is legal, we must provide an interpretation:

```
Definition mainP_interf (n:nat) := {|
  length := n+2;
  f := fun (k : nat) (proof : Is_true(k <? n+2)) =>
    if k =? n then
            24
          else
            45
|}.
```

```
Definition mainP_interpretation (n:nat) := {|
  f' := mainP_interf n;
  g := fun (k : nat) (proof : Is_true(k <? n+2)) =>
    if k =? n then
            n+2
          else
            k
|}.
```

First, we show that this interpretation indeed interprets the schedule. We leave this lemma as an exercise for the reader.

```
Lemma mainP_interpretation_interprets_main_schedule : forall n:nat,
interprets (minus_one_first (main_schedule n)) (mainP_interpretation n).
```

However, in the way we will prove the legality of the *main_schedule*, we will not even use the whole result, but only the base case of the induction:

```
Lemma mainP_interpretation_interprets_main_schedule :
interprets (minus_one_first (main_schedule 0)) (mainP_interpretation 0).
```

This is a simpler lemma to prove and we can reuse the code from one of the examples in the interpretations section. We will not describe much of the process, given that it is exactly the same as in lemma *b_interpretation_interprets_b*.

```
Proof.
  unfold interprets.
  simpl (length (minus_one_first (main_schedule 0))).
  simpl (length (f' (mainP_interpretation 0))).
  refine (conj _ _).
  auto.

  refine (conj _ _).
  intros k.
  case k.
  intros.
  simpl.
  exact I.

  intros n.
  case n.
  intros.
  simpl.
  exact I.

  intros n0.
```

```
  case n0.
  intros.
  simpl.
  exact I.

  intros n1.
  case n1.
  intros.
  simpl.
  exact I.

  contradiction.

  intros k.
  case k.
  intros.
  simpl.
  tauto.

  intros n.
  case n.
  intros.
  simpl.
  tauto.

  intros n0.
  case n0.
  intros.
  simpl.
  tauto.

  intros n1.
  case n1.
  intros.
  simpl.
  tauto.

  contradiction.

Qed.
```

And we proceed to the proof of Lemma 7.3.4 as follows:

```
Lemma minus_one_first_legal : forall n:nat,
     (legal(minus_one_first (concat(start_main_schedule n) end2_main_schedule))).
Proof.
  intros.
  rewrite P_same_schedule.
  induction n.
```

As we did in Chapter 7, we prove the result by induction on n. For the base case, we prove each of the properties of *legal* for the interpretation *mainP_interpretation 0*. But first we must provide a proof of the fact that it interprets the schedule:

```
unfold legal.
simpl (length (minus_one_first (main_schedule 0))).
pose (witness := (mainP_interpretation 0)).
refine (ex_intro _ witness _).
refine (conj _ _).
apply mainP_interpretation_interprets_main_schedule0.
simpl (length (f' witness)).
refine (conj _ _).
```

Our current goals are:

```
3 subgoals
witness := mainP_interpretation 0 : inter
______________________________________(1/3)
forall (k : nat) (proof : Is_true (k <? 2)),
Is_true (24 <=? f (f' witness) k proof)
______________________________________(2/3)
(forall (k : nat) (proof1 : Is_true (k <? 2)) (proof2 : Is_true (k + 1 <? 2)),
 Is_true (45 <=? f (f' witness) k proof1) \/
 Is_true (45 <=? f (f' witness) (k + 1) proof2)) /\
(forall (k : nat) (proof : Is_true (k <? 2)),
 Is_true (45 <=? f (f' witness) k proof) <-> Is_true (g witness k proof =? k)) /\
(forall (k : nat) (proof proofs : Is_true (k <? 2))
   (proof1 : Is_true (k - 1 <? 2)) (proof2 : Is_true (k - 2 <? 2))
   (proof3 : Is_true (k - 3 <? 2)),
 Is_true
   (legal_sum witness k proof proof1 proof2 proof3 =?
    f (minus_one_first (main_schedule 0)) k proofs))
______________________________________(3/3)
legal (minus_one_first (main_schedule (S n)))
```

The first condition of *legality* is easy to check and we have already seen a couple of proofs with the same idea: we examine $k = 0$ and $k = 1$ and then for the rest of the cases, since the schedule cannot take any $k > 1$, we get a contradiction.

```
intros.
simpl.
case k.
simpl.
exact I.
intros n.
case n.
simpl.
```

```
exact I.
intuition.
refine (conj _ _).
```

The second condition is analogous. The only difference is that we tell Coq which side of the disjunction we are going to prove:

```
intros.
simpl.
case k.
right.
simpl.
exact I.
intros n.
case n.
left.
simpl.
exact I.
intuition.
refine (conj _ _).
```

Condition iii) is also similar. Here the difference is that in the last case we have to prove *Is_true (n0 =? n0)*. The strategy we follow is: we transform it into *(n0 =? n0) = true* with a result called *Is_true_eq_left* and then use another result, *Nat.eqb_refl*, which states exactly what we want to prove.

```
intros.
simpl.
case k.
simpl.
intuition.
intros n.
case n.
simpl.
intuition.
simpl.
intuition.
apply Is_true_eq_left.
apply Nat.eqb_refl.
```

At this point, we have reduced the number of goals considerably:

```
2 subgoals
witness := mainP_interpretation 0 : inter
______________________________________(1/2)
forall (k : nat) (proof proofs : Is_true (k <? 2))
  (proof1 : Is_true (k - 1 <? 2)) (proof2 : Is_true (k - 2 <? 2))
```

```
  (proof3 : Is_true (k - 3 <? 2)),
Is_true
  (legal_sum witness k proof proof1 proof2 proof3 =?
   f (minus_one_first (main_schedule 0)) k proofs)
______________________________________(2/2)
legal (minus_one_first (main_schedule (S n)))
```

The last condition of *legality* involves the same reasoning, but we must first retrive what *legal_sum* was:

```
intros.
unfold legal_sum.
simpl.
destruct k.
simpl.
exact I.
destruct k.
simpl.
exact I.
contradiction.
```

And the base case of the induction is completed. We are left with the induction step:

```
1 subgoal
n : nat
IHn : legal (minus_one_first (main_schedule n))
______________________________________(1/1)
legal (minus_one_first (main_schedule (S n)))
```

For this, we are going to admit two simple lemmas that the reader can see are very intuitive. The first one: that prepending a week of 45 hours to a schedule does not modify its legality.

```
Definition schedule_45 :=
      {|
        length := 1;
        f := fun (k : nat) (proof : Is_true (k <? 1)) =>
          45
      |}.
Lemma prepend_45_is_fine : forall s:schedule,
      legal(concat schedule_45 s) <-> legal s.
```

The second result is that *minus_one_first(main_schedule (S n))* is precisely the concatenation of a week with 45 hours and *minus_one_first(main_schedule n)*.

```
Lemma prepend_45_to_minus_one_first : forall n:nat,
      minus_one_first (main_schedule (S n))=
      concat schedule_45 (minus_one_first (main_schedule n)).
```

The rest of the proof is now almost trivial:

```
assert (H: minus_one_first (main_schedule (S n))=
       concat schedule_45 (minus_one_first (main_schedule n))).
  apply prepend_45_to_minus_one_first.

rewrite H.
apply prepend_45_is_fine.
apply IHn.
Qed.
```

We bring the second result and rewrite the equality in our goal. After that, all we need to do is apply the first lemma and the induction hypothesis.

It is time to construct a proof of Theorem 7.3.6:

```
Lemma P_theorem : forall n:nat, (n>0 -> exists f:schedule, P_ambig f n).
```

We begin by providing the correct witness, *end2_main_schedule*:

```
Proof.
  intros.
  pose (witness := end2_main_schedule).
  refine (ex_intro _ witness _).
```

We have to unfold *ambig* and, since the length of the schedule is 2, Coq can prove the left part of the conjunction on its own with *intuition*.

```
unfold P_ambig.
intuition.
```

The current goal is:

```
1 subgoal
n : nat
H : n > 0
witness := end2_main_schedule : schedule
______________________________________(1/1)
exists h : schedule,
  Is_true (length h =? n + 1) /\
  (legal (concat h witness) -> False) /\
  legal (minus_one_first (concat h witness))
```

The witness for the new existential quantifier is clear: *start_main_schedule n*.

```
pose (witness2 := start_main_schedule n).
refine (ex_intro _ witness2 _).
intuition.
```

We have broken the goal into three simpler subgoals:

```
3 subgoals
n : nat
H : n > 0
witness := end2_main_schedule : schedule
witness2 := start_main_schedule n : schedule
______________________________________(1/3)
Is_true (length witness2 =? n + 1)
______________________________________(2/3)
False
______________________________________(3/3)
legal (minus_one_first (concat witness2 witness))
```

Since *length witness2* is indeed $n+1$, we can apply a procedure we used before for closing the subgoal:

```
apply Is_true_eq_left.
apply Nat.eqb_refl.
```

We have now to prove a contradiction:

```
2 subgoals
n : nat
H : n > 0
witness := end2_main_schedule : schedule
witness2 := start_main_schedule n : schedule
H0 : legal (concat witness2 witness)
______________________________________(1/2)
False
______________________________________(2/2)
legal (minus_one_first (concat witness2 witness))
```

But this is easy, we only need to bring the *illegal* lemma into our proof:

```
assert (~(legal (concat(start_main_schedule n) end2_main_schedule))).
  rewrite P_same_schedule.
  apply illegal.

contradiction.
```

And the last subgoal is exactly the lemma proved before this theorem:

```
apply minus_one_first_legal.
Qed.
```

10.2 A-ambiguity

Analogously to what we did for proving P-ambiguity, we can define the following two schedules:

```
Definition start2_main_schedule :=
      {|
        length := 2;
        f := fun (k : nat) (proof : Is_true (k <? 2)) =>
          match k with
          | 0 => 44
          | _ => 45
        end
      |}.

Definition end_main_schedule (n : nat) :=
      {|
        length := n + 1;
        f := fun (k : nat) (proof : Is_true (k <? n + 1)) =>
          if k =? n-1 then
            24
          else
            45
      |}.
```

And the corresponding lemma:

```
Lemma A_same_schedule : forall n,
      (concat start2_main_schedule (end_main_schedule n) = (main_schedule n)).
```

The proof of Lemma 7.3.3 relied on whether n was odd or even. Thus, we will need two different interpretations.

```
Definition odd_interf (n:nat) := {|
  length := n;
  f := fun (k : nat) (proof : Is_true(k <? n)) =>
       if k mod 2 =? 0 then
            44
       else
            45
|}.

Definition odd_interpretation (n:nat) := {|
  f' := (odd_interf n);
  g := fun (k : nat) (proof : Is_true(k <? n)) =>
    if k mod 2 =? 0 then
            k+2
```

```
        else
             k
|}.

Definition even_interf (n:nat) := {|
  length := n;
  f := fun (k : nat) (proof : Is_true(k <? n)) =>
        if k =? 0 then
             44
        else if k =? 1 then
             45
        else if k mod 2 =? 0 then
             45
        else if k =? n-1 then
             24
        else
             44
|}.

Definition even_interpretation (n:nat) := {|
  f' := (even_interf n);
  g := fun (k : nat) (proof : Is_true(k <? n)) =>
        if k =? 0 then
             3
        else if k =? 1 then
             1
        else if k mod 2 =? 0 then
             k
        else if k =? n-3 then
             k+3
        else
             k+2
|}.
```

These were the ones that appeared within the proof. For using them, we can use another "excluded middle":

```
Lemma odd_or_even : forall n,
      {Is_true(n mod 2 =? 0)} + {Is_true(n mod 2 =? 1)}.
```

We leave as an exercise for the reader to complete the proof of Lemma 7.3.3 using these interpretations and the previous result as a hint.

```
Lemma minus_one_last_legal : forall n:nat,
      ((n>0 /\ ~n=2) -> legal(minus_one_last (concat start2_main_schedule
      (end_main_schedule n)))).
```

Moreover, in Theorem 7.3.5, we distinguished two cases: when $n = 2$ and when $n \neq 2$. In the latter, we used the lemma above, but when $n = 2$, we had to define a different interpretation. Let us start by defining the right schedule when $n = 2$:

```
Definition A2_start :=
      {|
        length := 2;
        f := fun (k : nat) (proof : Is_true (k <? 2)) =>
          match k with
          | 0 => 45
          | _ => 44
        end
      |}.


Definition A2_end :=
      {|
        length := 3;
        f := fun (k : nat) (proof : Is_true (k <? 3)) =>
          match k with
          | 0 => 45
          | 1 => 24
          | _ => 45
        end
      |}.
```

When concatenated, they correspond to the schedule:

We also need two results: that the schedule is illegal and that the schedule minus the last week is legal. For the first one, we use the fact that the schedule is one of the instances of *main_schedule* but prepending a week of 45 hours.

```
Lemma A2_is_main : concat A2_start A2_end = concat schedule_45 (main_schedule 1).
```

Once we have realised this fact, the illegality lemma becomes quite trivial:

```
Lemma A2_illegal : ~(legal (concat A2_start A2_end)).
Proof.
  rewrite A2_is_main.
  rewrite -> prepend_45_is_fine.
  apply illegal.
Qed.
```

We rewrite the equality from *A2_is_main* and then apply the biconditional from lemma *prepend_45_is_fine*, which stated that the legality of an schedule that starts with a week of 45 hours is the same as the one of the schedule without the first week. Finally, the *ilegallity* lemma ends our proof.

For the legality lemma, we unsurprisingly need an intepretation:

```
Definition A2_interf := {|
  length := 4;
  f := fun (k : nat) (proof : Is_true(k <? 4)) =>
         match k with
         | 0 => 45
         | 1 => 44
         | 2 => 45
         | _ => 24
       end
|}.

Definition A2_interpretation := {|
  f' := A2_interf;
  g := fun (k : nat) (proof : Is_true(k <? 4)) =>
    if k =? 1 then
           k+3
    else if k =? 3 then
           k+1
         else
           k
|}.
```

The next steps are the same as always: prove that this interpretation interprets the schedule and that, moreover, it is a legal interpretation. We are not going into much detail because we have already seen several identical proofs.

```
Lemma A2_interpretation_interprets_A2 :
      interprets (minus_one_last (concat A2_start A2_end)) A2_interpretation.
Proof.
  unfold interprets.
  simpl (length (minus_one_last (concat A2_start A2_end))).
  simpl (length (f' A2_interpretation)).
  refine (conj _ _).
  exact (eq_refl 4).

  refine (conj _ _).

  intros k.
  case k.
  intros.
  simpl.
```

```
  tauto.

  intros n.
  case n.
  intros.
  simpl.
  tauto.

  intros n0.
  case n0.
  intros.
  simpl.
  tauto.

  intros n1.
  case n1.
  intros.
  simpl.
  tauto.

  contradiction.

  intros k.
  case k.
  intros.
  simpl.
  tauto.

  intros n.
  case n.
  intros.
  simpl.
  tauto.

  intros n0.
  case n0.
  intros.
  simpl.
  tauto.

  intros n1.
  case n1.
  intros.
  simpl.
  tauto.

  contradiction.
Qed.
```

```
Lemma A2_minus_one_last_legal : legal(minus_one_last (concat A2_start A2_end)).
Proof.
  unfold legal, minus_one_last, concat, A2_start, A2_end.
  simpl.
  pose (witness := A2_interpretation).
  refine (ex_intro _ witness _).
  simpl.
  refine (conj _ _).
  apply A2_interpretation_interprets_A2.

  refine (conj _ _).
  intros k.
  case k.
  intros.
  simpl.
  tauto.

  intros n.
  case n.
  intros.
  simpl.
  tauto.

  intros n0.
  case n0.
  intros.
  simpl.
  tauto.

  intros n1.
  case n1.
  intros.
  simpl.
  tauto.

  contradiction.

  refine (conj _ _).
  intros k.
  case k.
  intros.
  simpl.
  tauto.

  intros n.
  case n.
```

```
  intros.
  simpl.
  tauto.

  intros n0.
  case n0.
  intros.
  simpl.
  tauto.

  intros n1.
  case n1.
  intros.
  simpl.
  tauto.

  contradiction.

  refine (conj _ _).
  intros k.
  case k.
  intros.
  simpl.
  tauto.

  intros n.
  case n.
  intros.
  simpl.
  tauto.

  intros n0.
  case n0.
  intros.
  simpl.
  tauto.

  intros n1.
  case n1.
  intros.
  simpl.
  tauto.

  contradiction.

  intros k.
  case k.
  intros.
```

```
  simpl.
  tauto.

  intros n.
  case n.
  intros.
  simpl.
  tauto.

  intros n0.
  case n0.
  intros.
  simpl.
  tauto.

  intros n1.
  case n1.
  intros.
  simpl.
  tauto.

  contradiction.
Qed.
```

With this last proof, we have all the ingredients for the A-ambiguity theorem and ending the chapter (and the thesis).

```
Lemma A_theorem : forall n:nat, (n>0 -> exists f:schedule, A_ambig f n).
```

In Chapter 7, we distinguished between the cases $n = 2$ and $n \neq 2$. In Coq, we are going to use the tactic *destruct* for checking the cases $n = 0$, $n = 1$, $n = 2$ and $n > 2$. The first one is trivial, because one of the hypotheses is that $n > 0$, so we can end the case by a contradiction.

```
Proof.
  intros n.
  intros H.
  unfold A_ambig.
  destruct n.
  (*n=0*)
  assert (~ 0>0).
    intuition.
  contradiction.
```

For $n = 1$, we work with *main_schedule*, similarly to what we did for the P-ambiguity theorem. We first specify the two parts of the schedule for the existential quantifiers and then apply the legal and illegal lemmas.

```
destruct n.
(*n=1*)
pose (witness := start2_main_schedule).
refine (ex_intro _ witness _).
intuition.

pose (witness2 := end_main_schedule 1).
refine (ex_intro _ witness2 _).
intuition.

assert (~(legal (concat start2_main_schedule (end_main_schedule 1)))).
  rewrite A_same_schedule.
  apply illegal.

contradiction.

apply (minus_one_last_legal 1).
auto.
```

For $n = 2$, it is the *A2_schedule* that we must use, but the process is the same.

```
destruct n.
(*n=2*)
pose (witness := A2_start).
refine (ex_intro _ witness _).
intuition.

pose (witness2 := A2_end).
refine (ex_intro _ witness2 _).
intuition.
apply A2_illegal.
exact H0.

apply A2_minus_one_last_legal.
```

And, finally, for $n > 2$, we repeat the same proof we did for $n = 1$:

```
(*n>2*)
pose (witness := start2_main_schedule).
refine (ex_intro _ witness _).
intuition.

pose (witness2 := end_main_schedule (S (S (S n)))).
refine (ex_intro _ witness2 _).
intuition.
simpl.
apply Is_true_eq_left.
apply Nat.eqb_refl.
```

```
assert (~(legal (concat start2_main_schedule (end_main_schedule (S (S (S n)))))))).
  rewrite A_same_schedule.
  apply illegal.

contradiction.

apply (minus_one_last_legal (S (S (S n)))).
auto.
Qed.
```

We finished our theorem! It was not an easy task, though. Now the reader should take a break and pat himself on the back, for we managed to prove a real-life theorem using type theory. But we did more than that: we also proved our assertion that (at least some) laws can be logically formalised and verified, making verdicts easier to be handed down.

Like a blank canvas

Bibliography

[Adams] M. ADAMS *'Flyspecking Flyspeck'. In: Hong H., Yap C. (eds) Mathematical Software – ICMS 2014. ICMS 2014. Lecture Notes in Computer Science, vol 8592. Springer, Berlin, Heidelberg).*

[Barendregt] HENK BARENDREGT & ERIK BARENDSEN *'Introduction to Lambda Calculus', 2000.*

[Bertot] YVES BERTOT *'Coq in a Hurry'. 3rd cycle. Types Summer School, also used at the University of Goteborg, Nice,Ecole Jeunes Chercheurs en Programmation,Universite de Nice, 2010.*

[Coq Homepage] COQ HOMEPAGE `https://coq.inria.fr/`.

[Coq Manual] COQ REFERENCE MANUAL `https://coq.inria.fr/refman/`.

[Church] ALONZO CHURCH *'A formulation of the simple theory of types', Journal of Symbolic Logic, 1940.*

[Dybjer] PETER DYBJER & ERIK PALMGREN *'Intuitionistic Type Theory', The Stanford Encyclopedia of Philosophy (Winter 2016 Edition), Edward N. Zalta (ed.)*, available at `https://plato.stanford.edu/archives/win2016/entries/type-theory-intuitionistic/`.

[Engers] TOM M. VAN ENGERS, RON VAN GOG & KAMAL SAYAH *'A case study on Automated Norm Extraction' in T. Gordon (ed.), Legal Knowlegde and Information Systems. Jurix 2004: the Seventeenth Annual Conference. Amsterdam: IOS Press, 2004.*

[Gonthier] GEORGES GONTHIER *'Formal Proof— The Four Color Theorem'*, available at `https://www.ams.org/notices/200811/tx081101382p.pdf`.

[Irvine] ANDREW DAVID IRVINE & HARRY DEUTSCH *'Russell's Paradox'. The Stanford Encyclopedia of Philosophy (Winter 2016 Edition), Edward N. Zalta (ed.)*, available at `https://plato.stanford.edu/archives/win2016/entries/russell-paradox/`.

[Jesse] ALAMA, JESSE & KORBMACHER, JOHANNES *'The Lambda Calculus', The Stanford Encyclopedia of Philosophy (Summer 2018 Edition), Edward N. Zalta (ed.).*

[Law 561] OFFICIAL JOURNAL OF THE EUROPEAN UNION *'Regulation (EC) No 561/2006 of the European Parliament and of the Council', Official Journal of the European Union*, available at `https://eur-lex.europa.eu/legal-content/EN/ALL/?uri=CELEX%3A32006R0561`.

[Nahas] MIKE NAHAS *'Coq tutorial', 2014*, available at `https://coq.inria.fr/tutorial-nahas`.

[Nederpelt] BOB NEDERPELT & HERMAN GEUVERS *'Type Theory and Formal Proof- An introduction'. Cambrigde University Press, 2014.*

[Paulin-Mohring] CHRISTINE PAULIN-MOHRING *'Introduction to the Calculus of Inductive Constructions'. Bruno Woltzenlogel Paleo; David Delahaye. All about Proofs, Proofs for All, 55, College Publications, 2015.*

[Russell] BERTRAND RUSSELL *'Letter to Frege', in Jean van Heijenoort (ed.), From Frege to Gödel, Cambridge, Mass.: Harvard University Press, 1967.*

[Schnider] PATRICK SCHNIDER *'An introduction to proof assistants'. Student Seminar in Combinatorics: Mathematical Software, ETH Zurich.*

[Spreeuwenberg] SILVIE SPREEUWENBERG, TOM VAN ENGERS & RIK GERRITS *'The Role of Verification in Improving the Quality of Legal Decision-Making' in Bart Verheij, Arno R. Lodder, Ronald P. Loui and Antoinette J. Muntjewerff (eds.), Legal Knowledge and Information Systems. Jurix 2001: The Fourteenth Annual Conference. Amsterdam: IOS Press, 2001.*

[Stampoulis] ANTONIS STAMPOULIS *'Theorem proving with Coq'. Section Notes. Dept. of Computer Science, Yale University, 2009*, available at `http://flint.cs.yale.edu/cs430/sectionNotes/`.

[Univalent] UNIVALENT FOUNDATIONS PROGRAM *'Homotopy Type Theory: Univalent Foundations of Mathematics'. Institute for Advanced Study, Princeton, 2013*, available at `http://homotopytypetheory.org/book`.

[Urquhart] ALASDAIR URQUHART *'Russell's Zig-Zag Path to the Ramified Theory of Types', Russell, 1988.*

[Werner] BENJAMIN WERNER *'Sets in Types, Types in Sets'. Proceedings of TACS'97. Springer-Verlag, 1997.*