

# Hydras & Co.

*Formalized mathematics in Coq for inspiration and entertainment*

*Pierre Castéran, LaBRI, Univ. Bordeaux, CNRS UMR 5800*

*email: pierre dot casteran arobas gmail dot com.*

*With contributions by Yves Bertot, Ilmārs Cīrulis, Évelyne Contejean, Jérémy Damour, Stéphane Desarzens, Florian Hatat, Pascal Manoury, Karl Palmkog, Clément Pit-Claudel, and Théo Zimmermann.*

*The formalization of primitive recursive functions and Peano Arithmetic was originally authored by Russel O'Connor [O'C05b].*

February 19, 2024

`Set Implicit Arguments.`  
`Require Import ORDINAL.`  
`Require Import Ord_Complete.`  
`Require Import EO_Base.`  
  
`Module Epsilon0 := More_Ord Epsilon0_Base.`  
  
`Check Epsilon0_Base.omega.`  
`Check Epsilon0.lt.`  
`Check Epsilon0.omega.`

$\epsilon_0 = \varphi(1, 0)$   
 $\epsilon = \varphi(\varphi(0, 0), 0)$   
 $(1, 0) (0, 0)$   
 $\alpha_1 \leq \alpha_2$   $\omega * 2 + 3$   $(2)$   
 $\alpha_1 < \alpha_2$   $\beta_1 < (\alpha_2, \beta_2) \Rightarrow (\alpha_1, \beta_1) < (\alpha_2, \beta_2)$

*Ordinal numbers in Veblen normal form*

*(Aprieta el bastón con las dos manos, se yergue un tanto, casi con entusiasmo) ¡Caramba! Claro ... los números transfinitos, Kantor ...*  
[ Jorge Luis Borges]

Nessun senso percepisce l'infinito. Nessun senso permette di concludere ch'esso esista. L'infinito, in effetti, non puo' essere l'oggetto dei sensi.

[Giordano Bruno] *De l'infinito, universo e mondi*

I start from one point and go as far as possible.

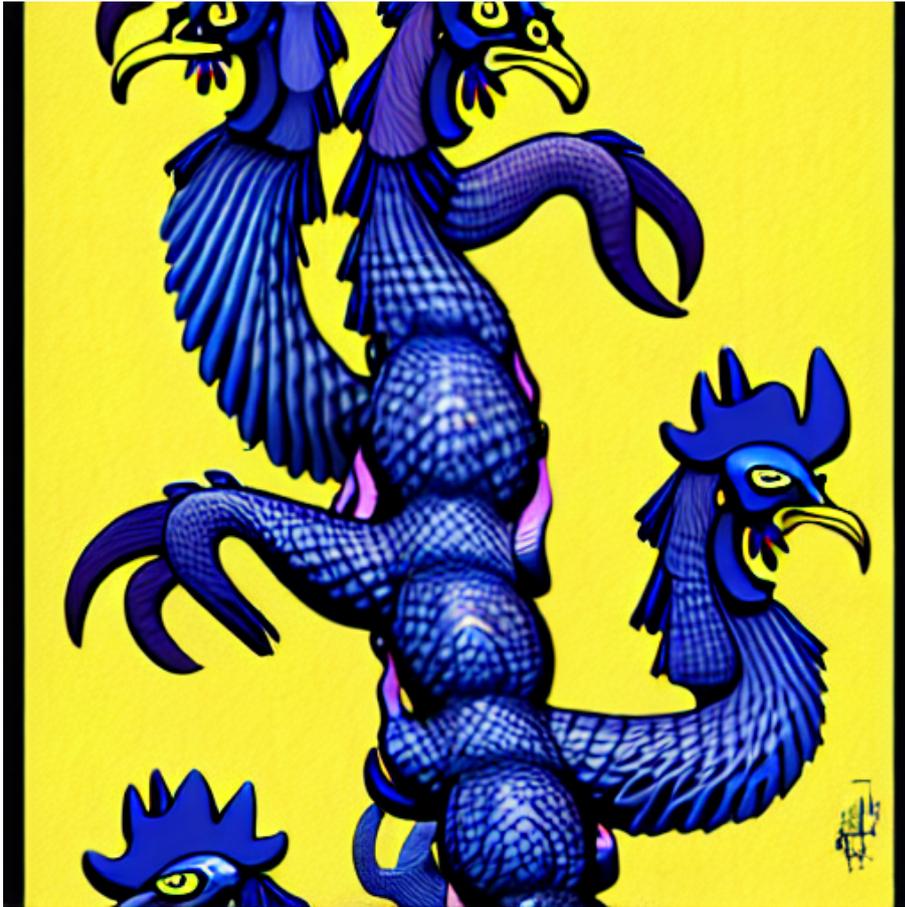
[John Coltrane]



*"The hydra in black and white": Watercolor by Pierrette Cassou-Noguès*



*"The blue hydra": Watercolor by Pierrette Cassou-Noguès*



*"The incomplete rooster hydra": AI art generated by Karl Palmkog*

# Contents

<b>1</b>	<b>Introduction</b>	<b>9</b>
1.1	Generalities . . . . .	9
1.2	How to install the libraries . . . . .	15
1.3	Comments on exercises and projects . . . . .	15
1.4	Acknowledgements . . . . .	16
<b>I</b>	<b>Hydras and ordinals</b>	<b>17</b>
<b>2</b>	<b>Hydras and hydra games</b>	<b>21</b>
2.1	Hydras and their representation in <i>Coq</i> . . . . .	25
2.2	Relational description of hydra battles . . . . .	30
2.3	A long battle . . . . .	36
2.4	Generic properties . . . . .	44
<b>3</b>	<b>Introduction to ordinal numbers and ordinal notations</b>	<b>49</b>
3.1	The mathematical point of view . . . . .	50
3.2	Ordinal numbers in <i>Coq</i> . . . . .	51
3.3	Ordinal Notations . . . . .	52
3.4	Example: the ordinal $\omega$ . . . . .	54
3.5	Sum of two ordinal notations . . . . .	54
3.6	Limits and successors . . . . .	56
3.7	Product of ordinal notations . . . . .	58
3.8	The ordinal $\omega^2$ . . . . .	59
3.9	A notation for finite ordinals . . . . .	65
3.10	Comparing two ordinal notations . . . . .	68
3.11	Comparing an ordinal notation with Schütte's model . . . . .	69
3.12	Isomorphism of ordinal notations . . . . .	70
3.13	Other ordinal notations . . . . .	71
<b>4</b>	<b>The ordinal <math>\epsilon_0</math></b>	<b>73</b>
4.1	The ordinal $\epsilon_0$ . . . . .	73
4.2	Well-foundedness and transfinite induction . . . . .	87
4.3	An ordinal notation for $\omega^\omega$ . . . . .	91
4.4	A variant for hydra battles . . . . .	95

<b>5</b>	<b>The Ketonen-Solovay machinery</b>	<b>101</b>
5.1	Introduction . . . . .	101
5.2	Canonical Sequences . . . . .	102
5.3	Accessibility inside $\epsilon_0$ : paths . . . . .	105
5.4	A proof of impossibility . . . . .	108
5.5	The case of standard battles . . . . .	111
<b>6</b>	<b>Large sets and rapidly growing functions</b>	<b>119</b>
6.1	Definitions . . . . .	119
6.2	Length of minimal large sequences . . . . .	121
6.3	A variant of the Hardy hierarchy . . . . .	129
6.4	A variant of the Wainer hierarchy (functions $F_\alpha$ ) . . . . .	136
6.5	More about rapidly growing functions . . . . .	139
<b>7</b>	<b>Gaia and the hydra (draft)</b>	<b>141</b>
7.1	Introduction . . . . .	141
7.2	Library structure . . . . .	142
7.3	Importing Definitions and theorems from Hydra-battles . . . . .	147
7.4	Rapidly growing arithmetic functions . . . . .	155
7.5	Importing a theorem from Gaia . . . . .	159
<b>8</b>	<b>Countable ordinals (after Schütte)</b>	<b>161</b>
8.1	Declarations and axioms . . . . .	162
8.2	Additional axioms . . . . .	163
8.3	The successor function . . . . .	167
8.4	Finite ordinals . . . . .	169
8.5	The definition of <b>omega</b> . . . . .	169
8.6	The exponential of basis $\omega$ . . . . .	174
8.7	More about $\epsilon_0$ . . . . .	178
8.8	Critical ordinals . . . . .	179
8.9	Cantor normal form . . . . .	180
8.10	An embedding of <b>T1</b> into <b>Ord</b> . . . . .	182
8.11	Related work . . . . .	183
<b>9</b>	<b>The Ordinal <math>\Gamma_0</math> (first draft)</b>	<b>185</b>
9.1	Introduction . . . . .	185
9.2	The type <b>T2</b> of ordinal terms . . . . .	186
9.3	A strict order on <b>T2</b> . . . . .	187
9.4	Veblen normal form . . . . .	189
9.5	Main functions on <b>T2</b> . . . . .	191
9.6	An ordinal notation for $\Gamma_0$ . . . . .	193
<b>II</b>	<b>Ackermann, Gödel, Peano ...</b>	<b>195</b>
<b>10</b>	<b>General presentation (draft)</b>	<b>197</b>
10.1	Introduction . . . . .	197
10.2	File contents . . . . .	198
10.3	Warning . . . . .	200

<b>11 Primitive recursive functions</b>	<b>201</b>
11.1 Introduction . . . . .	201
11.2 Mathematical definition . . . . .	201
11.3 First look at the Ackermann library . . . . .	203
11.4 Abstract syntax for primitive recursive functions . . . . .	203
11.5 Proving that a given Coq arithmetic function is primitive recursive	211
11.6 Proofs by induction over all primitive recursive functions . . . . .	219
11.7 Ackermann function is not primitive recursive . . . . .	222
11.8 The length of standard hydra battles . . . . .	228
<b>12 First Order Logic (in construction)</b>	<b>231</b>
12.1 Introduction . . . . .	231
12.2 Data types . . . . .	231
12.3 A notation scope for first-order terms and formulas . . . . .	235
12.4 Computing and reasoning on first-order formulas . . . . .	238
12.5 Proofs . . . . .	249
12.6 Concluding remarks . . . . .	256
<b>13 Natural Deduction (in construction)</b>	<b>257</b>
13.1 Contexts as sets . . . . .	257
13.2 The Deduction theorem . . . . .	259
13.3 Derived rules and natural deduction . . . . .	260
<b>14 Languages for Arithmetic (in construction)</b>	<b>265</b>
14.1 Notations for Formulas (experimental) . . . . .	267
<b>15 Gödel's Encoding (in construction)</b>	<b>269</b>
15.1 Cantor pairing function . . . . .	269
15.2 First order logic and Gödel encoding . . . . .	275
<b>16 Every Primitive Recursive Function is representable</b>	<b>277</b>
<b>III A few certified algorithms</b>	<b>279</b>
<b>17 Smart computation of <math>x^n</math></b>	<b>281</b>
17.1 Introduction . . . . .	281
17.2 Some basic implementations . . . . .	281
17.3 Representing monoids in Coq . . . . .	288
17.4 Computing powers in any EMonoid . . . . .	295
17.5 Comparing exponentiation algorithms with respect to efficiency .	300
17.6 Addition chains . . . . .	302
17.7 Proving a chain's correctness . . . . .	307
17.8 Certified chain generators . . . . .	318
17.9 Euclidean Chains . . . . .	321
17.10Projects . . . . .	343

<b>IV Appendices</b>	<b>349</b>
<b>18 Index and tables</b>	<b>359</b>
Links to Gaia Library . . . . .	360
Coq, plug-ins and standard library . . . . .	361
Mathematical notions and algorithmics . . . . .	362
Library hydras: Ordinals and hydra battles . . . . .	363
Library hydras.Ackermann: Primitive recursive functions, Gödel en- coding . . . . .	364
Library additions: Addition chains . . . . .	365

# Chapter 1

## Introduction

### 1.1 Generalities

Proof assistants are excellent tools for exploring the structure of mathematical proofs, studying which hypotheses are really needed, and which proof patterns are useful and/or necessary. Since the development of a theory is represented as a bunch of computer files, everyone is able to read the proofs with an arbitrary level of detail, or to play with the theory by writing alternate proofs or definitions.

If a formal development is large (at least 10 KLOCs), we believe that a human-readable document containing explanations, diagrams, code snippets, examples, exercises, etc.) would be useful for a better understanding of both the mathematical contents and the formalization techniques used in the development [CPC23].

This document has been generated with Alectryon (see Sect. 1.1.2 on the following page), which ensures the pdf is consistent with the last compiled version of the Coq project.

Among all the theorems proved with the help of proof assistants like Coq [Coq, BC04a], HOL [GM93], Isabelle [NPW02], etc., several statements and proofs share some interesting features:

- Their statements are easy to understand, even by non-mathematicians
- Their proof requires some non-trivial mathematical tools
- Their mechanization on computer presents some methodological interest.

This is obviously the case of the four-color theorem [Gon08] and the Kepler conjecture [HAB<sup>+</sup>17]. We do not mention impressive works like the proof of the odd-order theorem [GAA<sup>+</sup>13], since understanding its statement requires a quite good mathematical culture.

In this document, we present two examples which seem to have the above properties.

- Hydra games (a.k.a. *Hydra battles*) appear in an article published in 1982 by two mathematicians: L. Kirby and J. Paris [KP82]: *Accessible Independence Results for Peano Arithmetic*. Although the mathematical contents

of this paper are quite advanced, the rules of hydra battles are very easy to understand<sup>1</sup>. There are now several sites on the Internet where you can find tutorials on hydra games, together with simulators you can play with. See, for instance, the blogpost and source code written by Andrej Bauer [Bau08, Bau].

Hydra battles, as well as Goodstein sequences [Goo44, KP82] are a nice way to present complex termination problems. The article by Kirby and Paris presents a proof of termination based on ordinal numbers, as well as a proof that this termination is not provable in Peano arithmetic. In the book dedicated to J.P. Jouannaud [CLKK07], N. Dershowitz and G. Moser give a thorough survey on this topic [DM07].

We present a (still partial, under continuous development) implementation in Coq of the various techniques shown in Kirby & Paris' and Ketonen & Solovay's [KS81] article.

Our library Gaia-hydras is dedicated to make compatible our lemmas with José Grimm's Gaia project (designed for SSReflect/MathComp) (please look at Sect. I on page 19 and the paragraphs signalled with ).

- In the second part, we are interested in computing  $x^n$  with the least number of multiplications as possible. We use the notion of *addition chains* [Bra39, BB87], to generate efficient certified exponentiation functions.

**Warning:** This document is *not* an introductory text for Coq, and there are many aspects of this proof assistant that are not covered. The reader should already have some basic experience with the Coq system. The Reference Manual and several tutorials are available on the Coq website [Coq]. The first chapters of textbooks like *Interactive Theorem Proving and Program Development* [BC04a], *Software Foundations* [P<sup>+</sup>], *Programs and Proofs* [Ser14], or *Certified Programming with Dependent Types* [Ch11] will give you the right background.

### 1.1.1 Structure of Hydras & Co.

Hydras & Co. is made of three main packages: Hydra-battles, Gaia-hydras, and Addition-chains. Figure 1.1 illustrates the complex relationships: inheritance from historical contributions to Coq, and dependency with other Coq packages. Many thanks to Karl Palmkog and Théo Zimmermann for the CI/CD design of Hydras & Co. and the automation of documentation maintenance. Please look for a more detailed description in [CDP<sup>+</sup>22].

### 1.1.2 Documenting theories with Alectryon

Quotations of Coq source and answers are progressively replaced from copy-pasted *verbatim* to automatically generated *LaTeX* blocks, using Clément Pit-Claudel's Alectryon tool [PC20, PC]. Many thanks to Jérémy Damour, Clément

---

<sup>1</sup>Let us underline the analogy between hydra battles and interactive theorem proving. Hercules is the user (you!), and hydra's heads are the subgoals: you may think that applying a tactic would solve a subgoal, but it results often in the multiplication of such tasks.

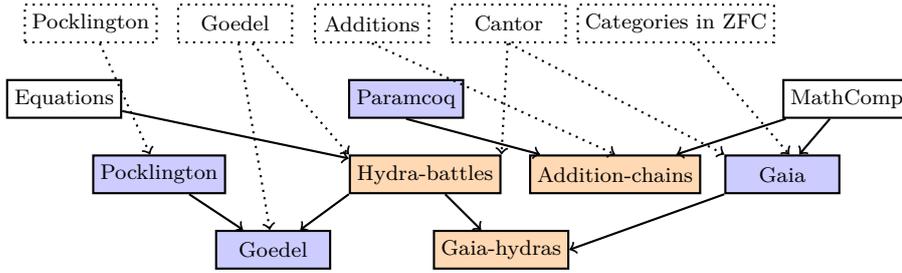


Figure 1.1: Genealogy and dependencies for Hydras & Co. packages. Dotted boxes represent historical Coq contributions, while regular boxes represent maintained Coq packages. Orange packages are maintained in the Hydras & Co. GitHub repository, while light blue packages are maintained in other Coq-community repositories. Dotted lines represent Coq code ancestry, while regular lines represent direct code dependencies.

Pit-Claudel and Théo Zimmermann who designed tools for maintaining consistency between the always evolving Coq modules and documentation written in *LaTeX*.

Besides the guarantee of consistency between theories and documentation, we hope to give a corpus for experimenting new ways of documenting the implementation of non-trivial mathematics on a proof assistant.

### 1.1.3 Trust in our proofs

Unlike mathematical literature, where definitions and proofs are spread out over many articles and books, the whole proof is now inside your computer. It is composed from the `.v` files you downloaded, parts of Coq’s standard library, and required Coq packages (see Fig. 1.1). Thus, there is no ambiguity in our definitions and the premises of the theorems. Furthermore, you will be able to navigate through the development, using your favorite text editor or IDE, and some commands like `Search`, `Locate`, etc.

### 1.1.4 Assumed redundancy

It may often happen that several definitions of a given concept, or several proofs of a given theorem are possible. If all the versions present some interest, we will make them available, since each one may be of some methodological interest (by illustrating some tactic of proof pattern, for instance). We use Coq’s module system to make several proofs of a given theorem co-exist in our libraries (see also Sect 1.1.9 on page 15). After some discussions of the pros and cons of each solution, we develop only one of them, leaving the others as exercises or projects (i.e., big or difficult exercises). In order to discuss which assumptions are really needed for proving a theorem, we will also present several aborted proofs. Of course, do not hesitate to contribute nice proofs or alternative definitions!

It may also happen that some direct proof looks to be useless, because the proven theorem is a trivial consequence of another (also proven) result. For instance, let us consider the three following statements:

1. There is no measure into  $\mathbb{N}$  for proving the termination of all hydra battles (Sect 2.4.3 on page 45).
2. There is no measure into the interval<sup>2</sup>  $[0, \omega^2)$  for proving the termination of all hydra battles (Sect. 3.8.2 on page 62).
3. There is no measure into  $[0, \mu)$  for proving the termination of all hydra battles, for any  $\mu < \epsilon_0$  (Sect.5.4.1 on page 109).

Obviously, the third theorem implies the second one, which implies the first one. So, theoretically, a library would contain only a proof of (3) and remarks for (2) and (1). But we found it interesting to make all the three proofs available, allowing the reader to compare their common structure and notice their technical differences. In particular, the proof of (3) uses several non-trivial combinatorial properties of ordinal numbers up to  $\epsilon_0$  [KS81], whilst (1) and (2) use simple properties of  $\mathbb{N}$  and  $\mathbb{N}^2$ .

### 1.1.5 About logic

Most of the proofs we present are *constructive*. Whenever possible, we provide the user with an associated function, which she or he can apply in Gallina or OCaml in order to get a “concrete” feeling of the meaning of the considered theorem. For instance, in Chapter 5 on page 101, the notion of *limit ordinal* is made more “concrete” thanks to a function `canon` which computes every item of a sequence which converges on a given limit ordinal  $\alpha$ . This simply typed function allows the user/reader to make her/his own experimentations. For instance, one can very easily compute the 42-nd item of a sequence which converges towards  $\omega^{\omega^{\omega}}$ .

Except in the `Schutte` library, dedicated to an axiomatic presentation of the set of countable ordinal numbers, all of our development is axiom-free, and respects the rules of intuitionistic logic. Note that we also use the `Equations` plug-in [SM19] in the definition of several rapidly growing hierarchy of functions, in Chap. 6. This plug-in imports several known-as-harmless axioms.

At any place of our development, you may use the `Print Assumptions` *ident* command in order to verify on which axiom the theorem *ident* may depend. The following example is extracted from Library `hydras.Epsilon0.F_alpha`, where we use the `coq-equations` plug-in (see Sect. 6.4 on page 136).

**Print Assumptions** F\_zero\_eqn.

```
Axioms:
functional_extensionality_dep
: forall (A : Type) (B : A -> Type)
  (f g : forall x : A, B x),
  (forall x : A, f x = g x) -> f = g
Eqdep.Eq_rect_eq.eq_rect_eq
: forall (U : Type) (p : U) (Q : U -> Type)
  (x : Q p) (h : p = p), x = eq_rect p Q x p h
```

<sup>2</sup>We use the notation  $[a, b)$  for denoting the set of ordinals greater or equal than  $a$  and strictly less than  $b$ .

## 1.1.6 Typographical Conventions

### 1.1.6.1 Using Alectryon

Whenever possible, we use Alectryon to display Coq code (definition, proof scripts) and answers. Here are two examples from Chapters 11 and 17.

```
Fixpoint Ack (m:nat) : nat -> nat :=
  match m with
  | 0 => S
  | n.+1 => fun k => iterate (Ack n) k.+1 1
  end.
```

**Compute** Ack 3 2.

```
= 29
: nat
```

```
Definition fib_eucl gamma `{Hgamma: Strategy gamma} n :=
  let c := make_chain gamma n
  in let r := chain_apply c (M:=Mul2) (1,0) in
    fst r + snd r.
```

**Time Compute** fib\_eucl dicho 153.

```
= 68330027629092351019822533679447
: N
Finished transaction in 0.014 secs (0.014u,0.s) (successful)
```

**Time Compute** fib\_eucl two 153.

```
= 68330027629092351019822533679447
: N
Finished transaction in 0.011 secs (0.011u,0.s) (successful)
```

**Time Compute** fib\_eucl half 153.

```
= 68330027629092351019822533679447
: N
Finished transaction in 0.01 secs (0.007u,0.003s) (successful)
```

### 1.1.6.2 Verbatim quotations

In some situations, we replace Alectryon snippets with verbatim blocks.

- When the quoted source belongs to some library on which we do not have the write permission, we cannot include directives for generating snippets. For instance, the following code belongs to Coq's standard library.

```
Inductive CompareSpec (Peq Plt Pgt : Prop) :
  comparison -> Prop :=
  CompEq : Peq -> CompareSpec Peq Plt Pgt Eq
  | CompLt : Plt -> CompareSpec Peq Plt Pgt Lt
  | CompGt : Pgt -> CompareSpec Peq Plt Pgt Gt.
```

- We use also verbatim code inclusions when the examples would lead to too long computations during the compilation and the documentation generation.

```
Example C87_ok_slow : chain_correct 87 C87.
Proof.
Time slow_chain_correct_tac.
```

```
Finished transaction in 49.927 secs (49.445u,0.079s) (successful)
```

```
Qed.
```

### 1.1.7 Remark

In general, we do not include full proof scripts in this document. The only exceptions are very short proofs (*e.g.*, proofs by computation, or by application of automatic tactics). Likewise, we may display only the important steps on a long interactive proof, for instance, in the following lemma (5.5.1.1 on page 114):

```
Lemma Lemma2_6_1 (alpha : T1) :
  nf alpha ->
  forall beta, beta t1< alpha ->
    {n:nat | const_path (S n) alpha beta}.
```

**Proof.**

```
transfinite_induction alpha.
```

```
(* ... *)
```

**Defined.**

The reader may consult the full proof scripts with Proof General or CoqIDE, for instance.

### 1.1.8 Active links

The links which appear in this pdf document lead are of three possible kinds of destination:

- Local links to the document itself,
- External links, mainly to Coq's website,
- Local links to pages generated by `coqdoc`. According to the current makefile (through the commands `make html` and `make pdf`), the pages generated by `coqdoc` are stored at the relative address `../theories/html/*.html` (from the location of the pdf). Thus, active links to our Coq modules may be incorrect if you did not get this pdf document by compiling the distribution available at <https://github.com/coq-community/hydra-battles>.

### 1.1.9 Alternative or bad definitions

Finally, we decided to include definitions or lemma statements, as well as tactics, that lead to dead-ends or too complex developments, with the following coloring. Bad definitions are "masked" inside modules called `Bad`, `Bad1`, etc.

**Module** `Bad`.

**Definition** `bottom` := `the_least Empty_set`.

**Lemma** `le_zero_bottom` : `zero <= bottom`.

**Proof.** `apply zero_le. Qed.`

**Lemma** `bottom_eq` : `bottom = bottom`.

**Proof.** `trivial. Qed.`

**Lemma** `le_bottom_zero` : `bottom <= zero`.

**Proof.**

`unfold bottom, the_least, the; apply iota_ind.`

---

```
exists ! x : Ord, least_member lt Empty_set x
```

---

```
forall a : Ord,
unique (least_member lt Empty_set) a -> a <= zero
```

---

**Abort.**

**End** `Bad`.

Likewise, alternative, but still unexplored definitions will be presented in modules `Alt`, `Alt1`, etc. Using these definitions is left as an implicit exercise.

**Module** `Alt`.

**Inductive** `Hydra` : `Set` :=  
| `hnode` (`daughters` : `list Hydra`).

**End** `Alt`.

## 1.2 How to install the libraries

The present distribution has been checked with versions up to 8.18 of the Coq proof assistant, with a few plug-ins. *Please refer to the README file of the project.*

## 1.3 Comments on exercises and projects

Although we do not plan to include complete solutions to the exercises, we think it would be useful to include comments and hints, and questions/answers from the users. In contrast, "projects" are supposed, once completed, to be included in the repository.

Please consult the sub-directory `exercises/` of the project (in construction).

## 1.4 Acknowledgements

Many thanks to Yves Bertot, Ilmārs Cīrulis, Évelyne Contejean, Jérémy Damour, Stéphane Desarzens, Florian Hatat, David Ilcinkas, Pascal Manoury, Karl Palmiskog, Clément Pit-Claudel, Sylvain Salvati, Alan Schmitt and Théo Zimmermann for their help on the elaboration of this library and document, and to the members of the *Formal Methods* team and the *Coq working group* at laBRI for their helpful comments on oral presentations of this work.

Many thanks also to the Coq development team and the members of the *Coq Club* for interesting discussions about the Coq system and the Calculus of Inductive Constructions.

The author of the present document wishes to express his gratitude to the late Patrick Dehornoy, whose talk was determinant for our desire to work on this topic. I owe my interest in discrete mathematics and their relation to formal proofs and functional programming to Srečko Brlek. Equally, there is W. H. Burge's book "*Recursive Programming Techniques*" [Bur75] which was a great source of inspiration.

Last but not least, many thanks to Pierrette Cassou-Noguès for the watercolor on pages 2 and 3. Thanks to Karl Palmiskog for his *rooster hydra*, page 4.

### 1.4.1 Contributions

Yves Bertot made nice optimizations to algorithms presented in Chapter 17. Évelyne Contejean contributed libraries on the recursive path ordering (*rpo*) for proving the well-foundedness of our representation of  $\epsilon_0$  and  $\Gamma_0$ . Florian Hatat proved many useful lemmas on countable sets, which we used in our adaptation of Schütte's formalization of countable ordinals. Pascal Manoury is integrating the ordinal  $\omega^\omega$  into our hierarchy of ordinal notations.

The formalization of primitive recursive functions was originally a part of Russel O'Connor's work on Gödel's incompleteness theorems [O'C05b].

Any form of contribution is welcome: correction of errors (typos and more serious mistakes), improvement of Coq scripts, proposition of inclusion of new chapters, and generally any comment or proposition that would help us. The text contains several *projects* which, when completed, may improve the present work. Please do not hesitate to share your contributions, for instance using pull requests and issues on GitHub. Thank you in advance!

## Part I

# Hydras and ordinals



## Introduction

In this part, we present a development for the Coq proof assistant, after the work of Kirby and Paris [KP82]. This formalization contains the following main parts:

- Representation in Coq of hydras and hydra battles.
- A proof that every battle is finite and won by Hercules. This proof is based on a *variant* which maps any hydra to an ordinal strictly less than  $\epsilon_0$  and is strictly decreasing along any battle.
- Using a combinatorial toolkit designed by J. Ketonen and R. Solovay [KS81], we prove that, for any ordinal  $\mu < \epsilon_0$ , there exists no such variant mapping any hydra to an ordinal strictly less than  $\mu$ . Thus, the complexity of  $\epsilon_0$  is really needed in the previous proof.
- We prove a relation between the length of a “classic” kind of battles<sup>3</sup> and the Wainer-Hardy hierarchy of “rapidly growing functions”  $H_\alpha$  [Wai70]. The considered class of battles, which we call *standard*, is the most considered one in the scientific literature (including popularization).

Simply put, this document tries to combine the scientific interest of two articles [KP82, KS81] and a book [Sch77] with the playful activity of truly proving theorems. We hope that such a work, besides exploring a nice piece of discrete mathematics, will show how Coq and its standard library are well fitted to help us to understand some non-trivial mathematical developments, and also to experiment the constructive parts of the proof through functional programming.

We also hope to provide a little clarification on infinity (both potential and actual) through the notions of function, computation, limit, type and proof.

### Compatibility with Gaia (in progress)

The Gaia project [GQS] by José Grimm, Alban Quadrat, and Carlos Simpson, aims to formalize mathematics in Coq in the style of Nicolas Bourbaki. It contains many definitions and results about ordinal numbers. In Chapter 7, we present some modules which allow Hydra-battles’ users to apply lemmas proven in Gaia, and vice versa. Remarks about compatibility with Gaia are signalled with the picture . A special index is in construction (page 361).

### Difference from Kirby and Paris’s work

In [KP82], Kirby and Paris show that there is no proof of termination of all hydra battles in Peano Arithmetic (PA). Since we are used to writing proofs in higher order logic, the restriction to PA was quite unnatural for us. So we chose to prove another statement without any reference to PA, by considering a class of proofs indexed by ordinal numbers up to  $\epsilon_0$ .

---

<sup>3</sup>This class is also called *standard* in this document (text and proofs). The *replication factor* of the hydra is exactly  $i$  at the  $i$ -th round of the battle (see Sect 2.0.1 on page 22).

## State of the development

The Coq scripts herein are in constant development since our contribution [CC06] on notations for the ordinals  $\epsilon_0$  and  $\Gamma_0$ . We added new material: axiomatic definitions of countable ordinals after Schütte [Sch77], combinatorial aspects of  $\epsilon_0$ , after Ketonen and Solovay [KS81] and Kirby and Paris [KP82], recent Coq technology: type classes, function definition by equations, etc.

We are now working in order to make clumsy proofs more readable, simplify definitions, and “factorize” proofs as much as possible. Many possible improvements are suggested as “todo”s or “projects” in this text.

## Future work (projects)

This document and the proof scripts are far from being complete.

First, there must be a lot of typos to correct, references and index items to add. Many proofs are too complex and should be simplified, etc.

The following extensions are planned, but help is needed:

- Semiautomatic tactics for proving inequalities  $\alpha < \beta$ , even when  $\alpha$  and  $\beta$  are not closed terms.
- More lemmas about hierarchies of rapidly growing functions, and their relationship with primitive recursive functions and provability in Peano arithmetic (following [KS81, KP82]).
- From Coq’s point of view, this development could be used as an illustration of the evolution of the software, every time new libraries and sets of tactics could help to simplify the proofs.

## Main references

In our development, we adapt the definitions and prove many theorems which we found in the following articles.

- “Accessible independence results for Peano arithmetic” by Laurie Kirby and Jeff Paris [KP82]
- ”Rapidly growing Ramsey Functions” by Jussi Ketonen and Robert Solovay [KS81]
- “The Termite and the Tower”, by Will Sladek [Sla07]
- Chapter V of “Proof Theory” by Kurt Schütte [Sch77]

## Chapter 2

# Hydras and hydra games

This chapter is dedicated to the representation of hydras and rules of the hydra game in Coq's specification language: Gallina.

Technically, a *hydra* is just a finite ordered tree, each node of which has any number of sons. Contrary to the computer science tradition, we display the hydras with the heads up and the foot (i.e., the root of the tree) down. Fig. 2.1 represents such a hydra, which will be referred to as  $\mathfrak{H}_y$  in our examples (please look at the module `Hydra.Hydra_Examples`). *For a less formal description of hydras, please see <https://www.smbc-comics.com/comic/hydra>.*

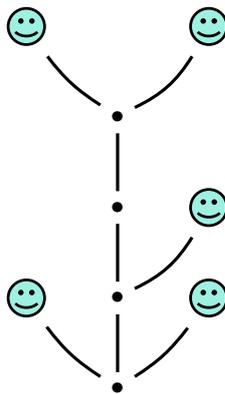


Figure 2.1: The hydra  $\mathfrak{H}_y$

We use a specific vocabulary for talking about hydras. Table 2.2 shows the correspondence between our terminology and the usual vocabulary for trees in computer science.

The hydra  $\mathfrak{H}_y$  has a *foot* (below), five *heads*, and eight *segments*. We leave it to the reader to define various parameters such as the height, the size, the highest arity (number of sons of a node) of a hydra. In our example, these parameters have the respective values 4, 9 and 3.

Hydras	Finite rooted trees
foot	root
head	leaf
node	node
segment	(directed) edge
sub-hydra	subtree
daughter	immediate subtree

Figure 2.2: Translation from hydras to trees

### 2.0.1 The rules of the game

A *hydra battle* is a fight between Hercules and the Hydra. More formally, a battle is a sequence of *rounds*. At each round:

- If the hydra is composed of just one head, the battle is finished and Hercules is the winner.
- Otherwise, Hercules chops off *one* head of the hydra,
  - If the head is at distance 1 from the foot, the head is just lost by the hydra, with no more reaction.
  - Otherwise, let us denote by  $r$  the node that was at distance 2 from the removed head in the direction of the foot, and consider the sub-hydra  $h'$  of  $h$ , whose root is  $r$ <sup>1</sup>. Let  $n$  be some natural number. Then  $h'$  is replaced by  $n + 1$  of copies of  $h'$  which share the same root  $r$ . The *replication factor*  $n$  may be different (and generally is) at each round of the fight. It may be chosen by the hydra, according to its strategy, or imposed by some particular rule. In many presentations of hydra battles, this number is increased by 1 at each round. In the following presentation, we will also consider battles where the hydra is free to choose its replication factor at each round of the battle<sup>2</sup>.

Note that the description given in [KP82] of the replication process in hydra battles is also semi-formal.

“From the node that used to be attached to the head which was just chopped off, traverse one segment towards the root until the next node is reached. From this node sprout  $n$  replicas of that part of the hydra (after decapitation) which is “above” the segment just traversed, i.e., those nodes and segments from which, in order to reach the root, this segment would have to be traversed. If the head just chopped off had the root of its nodes, no new head is grown. ”

Moreover, we note that this description is in *imperative* terms. In order to formally study the properties of hydra battles, we prefer to use a mathematical vocabulary, i.e., graphs, relations, functions, etc. Thus, the replication process

<sup>1</sup> $h'$  will be called “the wounded part of the hydra” in the subsequent text. In Figures 2.4 on the next page and 2.6 on page 24, this sub-hydra is displayed in red.

<sup>2</sup>Let us recall that, if the chopped-off head was at distance 1 from the foot, the replication factor is meaningless.

will be represented as a binary relation on a data type `Hydra`, linking the state of the hydra *before* and *after* the transformation. A battle will thus be represented as a sequence of terms of type `Hydra`, respecting the rules of the game. In other terms, we consider hydra battles as *transition systems*.

## 2.0.2 Example

Let us start a battle between Hercules and the hydra `Hy` of Fig. 2.1.

At the first round, Hercules chooses to chop off the rightmost head of `Hy`. Since this head is near the floor, the hydra simply loses this head. Let us call `Hy'` the resulting state of the hydra, represented in Fig. 2.3.

Next, assume Hercules chooses to chop off one of the two highest heads of `Hy'`, for instance the rightmost one. Fig. 2.4 represents the broken segment in dashed lines, and the part that will be replicated in red. Assume also that the hydra decides to add 4 copies of the red part<sup>3</sup>. We obtain a new state `Hy''` depicted in Fig. 2.5.

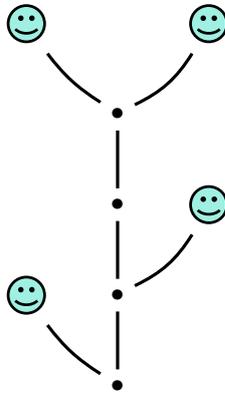


Figure 2.3: `Hy'`: the state of `Hy` after one round

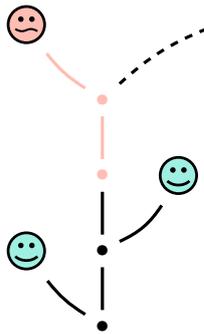
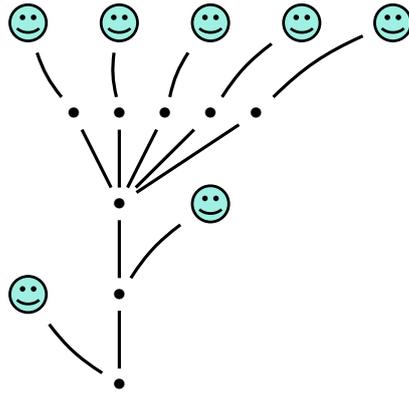


Figure 2.4: A second beheading

<sup>3</sup>In other words, the replication factor at this round is equal to 4.

Figure 2.5:  $\text{Hy}''$ : the state of  $\text{Hy}$  after two rounds

Figs. 2.6 and 2.7 on the next page represent a possible third round of the battle, with a replication factor equal to 2. Let us call  $\text{Hy}'''$  the state of the hydra after that third round.

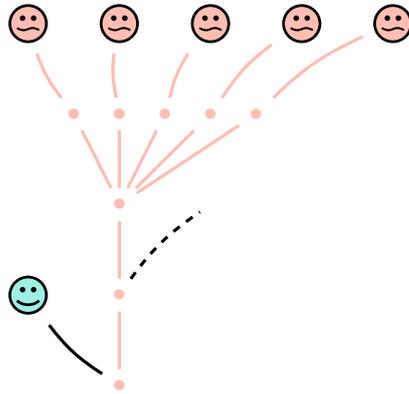
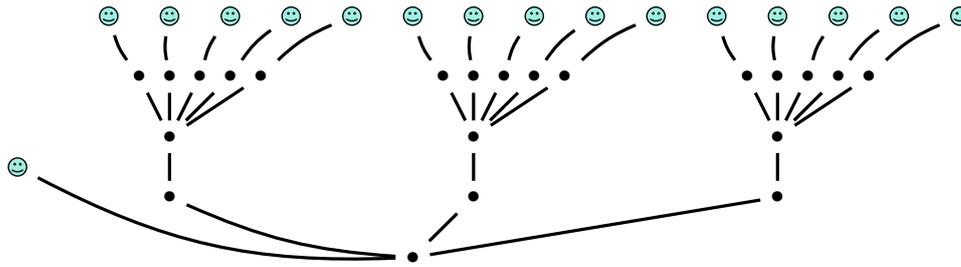


Figure 2.6: A third beheading (wounded part in red)

Figure 2.7: The configuration  $\text{Hy}'''$  of  $\text{Hy}$ 

We leave it to the reader to guess the following rounds of the battle ...

## 2.1 Hydras and their representation in *Coq*

In order to describe trees where each node can have an arbitrary (but finite) number of sons, it is usual to define a type where each node carries a *forest*, *i.e.* a list of trees (see for instance Chapter 14, pages 400-406 of [BC04a], also available as [BC04b]).

For this purpose, we define two mutual *ad-hoc* inductive types, where `Hydra` is the main type, and `Hydrae` is a helper for describing finite sequences of hydras.

From Module `Hydra.Hydra_Definitions`

```

Inductive Hydra : Set :=
| node : Hydrae -> Hydra
with Hydrae : Set :=
| hnil : Hydrae
| hcons : Hydra -> Hydrae -> Hydrae.

```

**Project 2.1** Look for an existing library on trees with nodes of arbitrary arity, in order to replace this ad-hoc type with something more generic.

**Remark 2.1 (Mutually inductive types vs lists of hydras)** Another very similar representation could use the `list` type family instead of the specific type `Hydrae`:

```

Module Alt.
  Inductive Hydra : Set :=
  | hnode (daughters : list Hydra).
End Alt.

```

Using this representation, one can re-define all the constructions of this chapter, which is left as an exercise. You will probably have to use patterns described for instance in [BC04a, BC04b] or the archives of the Coq communication channels (please consult <https://coq.inria.fr/community.html>).

**Project 2.2** Our type `Hydra` describes hydras as *plane oriented trees*, i.e., as drawn on a sheet of paper or computer screen. Thus, it is appropriate to consider a *leftmost* or *rightmost* head of the beast. It could be interesting to consider another representation, in which every non-leaf node has a *multi-set* – not an ordered list – of daughters.

### 2.1.0.1 Abbreviations

We provide several notations for hydra patterns which occur often in our developments.

*From Module Hydra.Hydra\_Definitions*

```
(** *** Hydra with 0, 1, 2 or 3 daughters *)
```

```
Notation head := (node hnil).
```

```
Notation hyd1 h := (node (hcons h hnil)).
```

```
Notation hyd2 h h' := (node (hcons h (hcons h' hnil))).
```

```
Notation hyd3 h h' h'' := (node (hcons h (hcons h' (hcons h'' hnil)))).
```

For instance, the hydra `Hy` of Figure 2.1 on page 21 is defined as follows:

*From Module Hydra.Hydra\_Examples*

```
Example Hy := hyd3 head
              (hyd2
               (hyd1
                (hyd2 head head))
               head)
              head.
```

Hydras quite frequently contain multiple adjacent copies of the same subtree. The following functions will help us to describe and reason about replications in hydra battles.

*From Module Hydra.Hydra\_Definitions*

```
Fixpoint hcons_mult (h:Hydra) (n:nat) (s:Hydrae) :Hydrae :=
  match n with
  | 0 => s
  | S p => hcons h (hcons_mult h p s)
  end.
```

```
(** *** Hydra with n equal daughters *)
```

```
Definition hyd_mult h n := node (hcons_mult h n hnil).
```

Let us consider for instance the hydra `Hy''` of Fig 2.5 on page 24.

*From Module Hydra.Hydra\_Examples*

```

Example Hy'' :=
  hyd2 head
    (hyd2
      (hyd_mult (hyd1 head) 5)
      head).

```

### 2.1.0.2 Recursive functions on type Hydra

In order to define a recursive function over the type `Hydra`, one has to consider the three constructors `node`, `hnil` and `hcons` of the mutually inductive types `Hydra` and `Hydrae`. Let us define for instance the function which computes the number of nodes of any hydra:

From Module `Hydra.Hydra_Definitions`

```

Fixpoint hsize (h:Hydra) : nat :=
  match h with
  | node l => S (lhsizel l)
  end
with lhsizel l : nat :=
  match l with
  | hnil => 0
  | hcons h hs => hsize h + lhsizel hs
  end.

```

**Compute** hsize Hy.

```

= 9
: nat

```

Likewise, the *height* (maximum distance between the foot and a head) is defined by mutual recursion:

```

Fixpoint height (h:Hydra) : nat :=
  match h with
  | node l => lheight l
  end
with lheight l : nat :=
  match l with
  | hnil => 0
  | hcons h hs => Nat.max (S (height h)) (lheight hs)
  end.

```

**Compute** height Hy.

```

= 4
: nat

```

**Exercise 2.1** Define a function `max_degree: Hydra → nat` which returns the highest degree of a node in any hydra. For instance, the evaluation of the term `(max_degree Hy)` should return 3.

### 2.1.1 Induction principles for hydras

In this section, we show how induction principles are used to prove properties on the type Hydra. Let us consider for instance the following statement:

“ The height of any hydra is strictly less than its size. ”

#### 2.1.1.1 A failed attempt

One may try to use the default tactic of proof by induction, which corresponds to an application of the automatically generated induction principle for type Hydra:

**Check** Hydra\_ind.

```
Hydra_ind
  : forall P : Hydra -> Prop,
    (forall h : Hydrae, P (node h)) ->
      forall h : Hydra, P h
```

Let us start a simple proof by induction.

*From Module Hydra.Hydra\_Examples*

**Module** Bad.

**Lemma** height\_lt\_size (h:Hydra) : height h < hsize h.

**Proof.**

induction h as [s].

```
s: Hydrae
-----
height (node s) < hsize (node s)
```

We might be tempted to do an induction on the sequence s:

induction s as [| h s'].

```
height head < hsize head
h: Hydra
s': Hydrae
IHs': height (node s') < hsize (node s')
-----
height (node (hcons h s')) < hsize (node (hcons h s'))
```

The first subgoal is trivial.

```
height head < hsize head
```

simpl; auto with arith.

Let us look now at the second subgoal of the induction.

```

h: Hydra
s': Hydrae
IHs': height (node s') < hsize (node s')
-----
height (node (hcons h s')) < hsize (node (hcons h s'))

```

cbn.

```

h: Hydra
s': Hydrae
IHs': height (node s') < hsize (node s')
-----
match lheight s' with
| 0 => S (height h)
| S m' => S (Nat.max (height h) m')
end < S (hsize h + lheight s')

```

We notice immediately that the context of this sub-goal does not allow to infer its conclusion. Let's stop.

**Abort.**

**End Bad.**

### 2.1.1.2 A Principle of mutual induction

In order to get an appropriate induction scheme for the types `Hydra` and `Hydrae`, we can use Coq's command `Scheme`.

```

Scheme Hydra_rect2 := Induction for Hydra Sort Type
with   Hydrae_rect2 := Induction for Hydrae Sort Type.

```

**Check** `Hydra_rect2`.

```

Hydra_rect2
: forall (P : Hydra -> Type)
  (P0 : Hydrae -> Type),
  (forall h : Hydrae, P0 h -> P (node h)) ->
  P0 hnil ->
  (forall h : Hydra,
   P h ->
   forall h0 : Hydrae, P0 h0 -> P0 (hcons h h0)) ->
  forall h : Hydra, P h

```

### 2.1.1.3 A Correct proof

Let us now use `Hydra_rect2` for proving that the height of any hydra is strictly less than its size. Using this scheme requires an auxiliary predicate, called `P0` in `Hydra_rect2`'s statement.

*From Module `Hydra.Hydra_Definitions`*

```

(** All elements of s satisfy P *)
Fixpoint h_forall (P: Hydra -> Prop) (s: Hydrae) :=
  match s with
  | hnil => True

```

```
| hcons h s' => P h /\ h_forall P s'
end.
```

From Module *Hydra.Hydra\_Examples*

**Lemma** `height_lt_size` ( $h$ :Hydra) : height  $h$  < hsize  $h$ .

**Proof.**

```
induction h using Hydra_rect2 with
  (P0 := h_forall (fun h => height h < hsize h)).
```

```
h: Hydrae
IHh: h_forall (fun h : Hydra => height h < hsize h) h
-----
height (node h) < hsize (node h)
-----
h_forall (fun h : Hydra => height h < hsize h) hnil

h: Hydra
h0: Hydrae
IHh: height h < hsize h
IHh0: h_forall (fun h : Hydra => height h < hsize
  h)
  h0
-----
h_forall (fun h : Hydra => height h < hsize h)
  (hcons h h0)
```

The first subgoal is easily solvable, using some arithmetic. The second and third ones are almost trivial. We let the reader look at the source.

**Qed.**

**Exercise 2.2** It happens very often that, in the proof of a proposition of the form  $(\forall h:\text{Hydra}, P h)$ , the predicate  $P0$  is  $(h\_forall P)$ . Design a tactic for induction on hydras that frees the user from binding explicitly  $P0$ , and solves trivial subgoals. Apply it for writing a shorter proof script of `height_lt_size`.

## 2.2 Relational description of hydra battles

In this section, we represent the rules of hydra battles as a binary relation associated with a *round*<sup>4</sup>, i.e., an interaction composed of the two following actions:

1. Hercules chops off one head of the hydra.
2. Then, the hydra replicates the wounded part (if the head is at distance  $\geq 2$  from the foot).

The relation associated with each round of the battle is parameterized by the *expected* replication factor (irrelevant if the chopped head is at distance 1 from the foot, but present for consistency's sake).

<sup>4</sup>usually called a *small step semantics*

In our description, we will apply the following naming convention: if  $h$  represents the configuration of the hydra before a round, then the configuration of  $h$  after this round will be called  $h'$ . Thus, we are going to define a proposition (`round_n n h h'`) whose intended meaning will be “the hydra  $h$  is transformed into  $h'$  in a single round of a battle, with the expected replication factor  $n$ ”.

Since the replication of parts of the hydra depends on the distance of the chopped head from the foot, we decompose our description into two main cases, under the form of a bunch of [mutually] inductive predicates over the types `Hydra` and `Hydrae`.

The mutually exclusive cases we consider are the following:

- **R1**: The chopped off head was at distance 1 from the foot.
- **R2**: The chopped off head was at a distance greater than or equal to 2 from the foot.

### 2.2.1 Chopping off a head at distance 1 from the foot (relation R1)

If Hercules chops off a head next to the root, there is no replication at all. We use an auxiliary predicate `S0`, associated with the removing of one head from a sequence of hydras.

*From Module Hydra.Hydra\_Definitions*

```
Inductive S0 : relation Hydrae :=
| S0_first : forall s, S0 (hcons head s) s
| S0_rest : forall h s s', S0 s s' -> S0 (hcons h s) (hcons h s').
```

```
Inductive R1 : relation Hydra :=
| R1_intro : forall s s', S0 s s' -> R1 (node s) (node s').
```

#### 2.2.1.1 Example

Let us represent in Coq the transformation of the hydra of Fig. 2.1 on page 21 into the configuration represented in Fig. 2.3 on page 23.

*From Module Hydra.Hydra\_Examples*

```
Example Hy_1 : R1 Hy Hy'.
Proof. repeat constructor. Qed.
```

### 2.2.2 Chopping off a head at distance $\geq 2$ from the foot (relation R2)

Let us now consider beheadings where the chopped-off head is at distance greater than or equal to 2 from the foot. All the following relations are parameterized by the replication factor  $n$ .

Let  $s$  be a sequence of hydras. The proposition (`S1 n s s'`) holds if  $s'$  is obtained by replacing some element  $h$  of  $s$  by  $n + 1$  copies of  $h'$ , where the proposition (`R1 h h'`) holds, in other words,  $h'$  is just  $h$ , without the chopped-off head. `S1` is an inductive relation with two constructors that allow us to choose the position in  $s'$  of the wounded sub-hydra  $h$ .

From Module *Hydra.Hydra\_Definitions*

```
Inductive S1 (n:nat) : relation Hydrae :=
| S1_first : forall s h h' ,
    R1 h h' ->
    S1 n (hcons h s) (hcons_mult h' (S n) s)
| S1_next : forall h s s' ,
    S1 n s s' ->
    S1 n (hcons h s) (hcons h s').
```

The rest of the definition is composed of two mutually inductive relations on hydras and sequences of hydras. The first constructor of **R2** describes the case where the chopped head is exactly at height 2. The others constructors allow us to consider beheadings at height strictly greater than 2.

From Module *Hydra.Hydra\_Definitions*

```
Inductive R2 (n:nat) : relation Hydra :=
| R2_intro : forall s s' , S1 n s s' -> R2 n (node s) (node s')
| R2_intro_2 : forall s s' , S2 n s s' -> R2 n (node s) (node s')

with S2 (n:nat) : relation Hydrae :=
| S2_first : forall h h' s ,
    R2 n h h' -> S2 n (hcons h s) (hcons h' s)
| S2_next : forall h r r' ,
    S2 n r r' -> S2 n (hcons h r) (hcons h r').
```

### 2.2.2.1 Example

Let us prove the transformation of  $Hy'$  into  $Hy''$  (see Fig. 2.5 on page 24). We use an experimental set of tactics<sup>5</sup> for specifying the place where the interaction between Hercules and the hydra holds.

From Module *Hydra.Hydra\_Examples*.

**Example** *R2\_example*:  $R2\ 4\ Hy'\ Hy''$ .

**Proof.**

```
R2 4 Hy' Hy''
```

```
(** move to 2nd sub-hydra (0-based indices) *) r2_up 1.
```

```
R2 4 (hyd2 (hyd1 (hyd2 head head)) head)
(hyd2 (hyd_mult (hyd1 head) 5) head)
```

```
(** move to first sub-hydra *) r2_up 0.
```

```
R2 4 (hyd1 (hyd2 head head)) (hyd_mult (hyd1 head) 5)
```

```
(** we're at distance 2 from the to-be-chopped-off head
let's go to the first daughter,
then chop-off the leftmost head *) r2_d2 0 0.
```

**Qed.**

<sup>5</sup>See the Ltac definitions in *Hydra.Hydra\_Definitions*.

The reader is encouraged to look at all the successive subgoals of this example. *Please consider also exercise 2.5 on the next page.*

### 2.2.3 Binary relation associated with a round

Let us merge R1 and R2 into a single relation. First, we define the relation (`round_n n h h'`) where `n` is the expected number of replications (irrelevant in the case of an R1-transformation). Then, we define a *round* (small step) of a battle by abstraction over `n`,

*From Module Hydra.Hydra\_Definitions*

**Definition** `round_n n h h' := R1 h h' \\/ R2 n h h'.`

**Definition** `round h h' := exists n, round_n n h h'.`

**Infix** `"-1->" := round (at level 60).`

**Project 2.3** Give a direct translation of Kirby and Paris’s description of hydra battles (quoted on page 22) and prove that our relational description is consistent with theirs.

### 2.2.4 Rounds and battles

Using library `Relations.Relation_Operators`, we define `round_plus`, the transitive closure of `round`, and `round_star`, the reflexive and transitive closure of `round`.

**Definition** `round_plus := clos_trans_1n Hydra round.`

**Definition** `round_star h h' := h = h' \\/ round_plus h h'.`

**Infix** `"-+->" := round_plus (at level 60).`

**Infix** `"-*->" := round_star (at level 60).`

**Remark 2.2** Coq’s library `Coq.Relations.Relation_Operators` contains three logically equivalent definitions of the transitive closure of a binary relation. This equivalence is proved in `Coq.Relations.Operators_Properties`.

Why three definitions for a single mathematical concept? Each definition generates an associated induction principle. According to the form of statement one would like to prove, there is a “best choice”:

- To prove  $\forall y, x R^+ y \rightarrow P y$ , prefer `clos_trans_n1`
- To prove proving  $\forall x, x R^+ y \rightarrow P x$ , prefer `clos_trans_1n`
- To prove  $\forall x y, x R^+ y \rightarrow P x y$ , prefer `clos_trans`,

But there is no “wrong choice” at all: the equivalence lemmas in `Coq.Relations.Operators_Properties` allow the user to convert any one of the three closures into another one before applying the corresponding elimination tactic. The same remark also holds for reflexive and transitive closures.

**Exercise 2.3** Prove that if  $h \dashrightarrow h'$ , then the height of  $h'$  is less or equal than the height of  $h$ .

**Exercise 2.4** Define a restriction of **round**, where Hercules always chops off the leftmost among the lowest heads.

Prove that, if  $h$  is not a simple head, then there exists a unique  $h'$  such that  $h$  is transformed into  $h'$  in one round, according to this restriction.

**Exercise 2.5 (Interactive battles)** Given a hydra  $h$ , the specification of a hydra battle for  $h$  is the type  $\{h' : \text{Hydra} \mid h \dashrightarrow h'\}$ . In order to avoid long sequences of `split`, `left`, and `right`, design a set of dedicated tactics for the interactive building of a battle. Your tactics will have the following functionalities:

- Choose to stop a battle, or continue
- Choose an expected number of replications
- Navigate in a hydra, looking for a head to chop off.

Use your tactics for simulating a small part of a hydra battle, for instance the rounds which lead from  $H_y$  to  $H_y''$  (Fig. 2.7 on page 25).

**Hints:**

- Please keep in mind that the last configuration of your interactively built battle is known only at the end of the battle. Thus, you will have to create and solve subgoals with existential variables. For that purpose, the tactic `eeexists`, applied to the goal  $\{h' : \text{Hydra} \mid h \dashrightarrow h'\}$  generates the subgoal  $h \dashrightarrow ?h'$ .
- You may use Gérard Huet's *zipper* data structure [Hue97] for writing tactics associated with Hercules's interactive search for a head to chop off.

## 2.2.5 Classes of battles

In some presentations of hydra battles, e.g. [KP82, Bau08], the transformation associated with the  $i$ -th round may depend on  $i$ . For instance, in these articles, the replication factor at the  $i$ -th round is equal to  $i$ . In other examples, one can allow the hydra to apply any replication factor at any time. In order to be the most general as possible, we define the type of predicates which relate the state of the hydra before and after the  $i$ -th round of a battle.

*From Module Hydra.Hydra\_Definitions*

```
Definition round_t := nat -> Hydra -> Hydra -> Prop.
```

```
Class Battle :=
  { battle_rel : round_t;
    battle_ok : forall i h h', battle_rel i h h' -> round h h' }.
```

```
Arguments battle_rel : clear implicits.
```

The most general class of battles is `free`, which allows the hydra to choose any replication factor at every step:

*From Module Hydra.Hydra\_Definitions*

```
#[ global, refine ] Instance free : Battle
:= Build_Battle (fun _ h h' => round h h') _ .
Proof. easy. Defined.
```

We chose to call *standard*<sup>6</sup> the kind of battles which appear most often in the literature and correspond to an arithmetic progression of the replication factor : 0, 1, 2, 3, ...

*From Module Hydra.Hydra\_Definitions*

```
#[ global, refine] Instance standard : Battle :=
  Build_Battle round_n _ .
Proof.
  intros i h h' H; now exists i.
Defined.
```

## 2.2.6 Big steps

Let  $B$  be any instance of class `Battle`. It is easy to define inductively the relation between the  $i$ -th and the  $j$ -th steps of a battle of type  $B$ .

*From Module Hydra.Hydra\_Definitions*

```
Inductive rounds (B:Battle)
  : nat -> Hydra -> nat -> Hydra -> Prop :=
  rounds_1 : forall i h h',
    battle_rel B i h h' -> rounds B i h (S i) h'
| rounds_n : forall i h j h' h'',
    battle_rel B i h h'' ->
    rounds B (S i) h'' j h' ->
    rounds B i h j h'.
```

(\*\* number of steps leading to the hydra's death \*)

```
Definition battle_length B k h l :=
  rounds B k h (Nat.pred (k + l)%nat) head.
```

The following property allows us to build battles by composition of smaller ones.

*From Module Hydra.Hydra\_Lemmas*

```
Lemma rounds_trans {B:Battle} :
  forall i h j h', rounds B i h j h' ->
    forall k h0, rounds B k h0 i h ->
      rounds B k h0 j h'.
```

**Proof.**

---

<sup>6</sup>This appellation is ours. If there is a better one, we will change it.

```

intros i h j h' H k h0. induction 1 /dr.
- intros h'0 ? ?; now right with h'0.
- intros ? ? ? h'' ? ? ? ; right with h'';auto.
Qed.

```

## 2.3 A long battle

In this section we consider a simple example of battle, starting with a small hydra, shown on figure 2.8, with a simple strategy for both players:

- At each round, Hercules chops off the rightmost head of the hydra.
- The battle is standard: at the round number  $i$ , the expected replication factor is  $i$ .

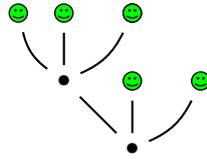


Figure 2.8: The hydra hinit

*From Module Hydra.BigBattle*

```

#[local] Notation h3 := (hyd_mult head 3).
Definition hinit := hyd3 h3 head head.

```

The lemma we would like to prove is “The considered battle lasts exactly  $N$  rounds”, with  $N$  being a natural number we have to guess.

But the battle is so long that no *test* can give us any estimation of its length. Nevertheless, in order to guess this length, we made some experiments, computing with Gallina, Coq’s functional programming language. Thus, we can consider this development as a collaboration of proof with computation. In the rest of this section, we show how we found experimentally the value of the number  $N$ . The complete proof is in file `../theories/html/hydras.Hydra.BigBattle.html`.

### 2.3.1 First rounds

During the two first rounds, our hydra loses its two rightmost heads. Figure 2.9 on the facing page shows the state of the hydra just before the third round.

The following lemma is a formal description of these first rounds, in terms of the `rounds` predicate.

```

Lemma L_0_2 : rounds standard 0 hinit 2 (hyd1 h3).
Proof.
  eapply rounds_trans with (h := hyd2 h3 head) (i:=1).
  (* ... *)
Qed.

```

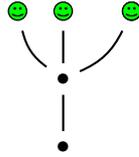


Figure 2.9: The hydra (hyd1 h3)

### 2.3.2 Looking for regularities

A first study with pencil and paper suggested us that, after three rounds, the hydra always looks like in figure 2.10 (with a variable number of subtrees of height 1 or 0). Thus, we introduce a few handy abbreviations.

**Notation** `h2` := (hyd\_mult head 2).

**Notation** `h1` := (hyd1 head).

**Notation** `hyd a b c` :=  
 (node (hcons\_mult h2 a  
 (hcons\_mult h1 b  
 (hcons\_mult head c hnil)))).

For instance, the hydra shown in Fig 2.10 is (hyd 3 4 2). The hydra (hyd 0 0 0) is the “final” hydra of any terminating battle, i.e., a tree with exactly one node and no edge.

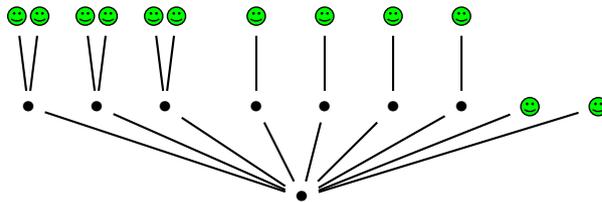


Figure 2.10: The hydra (hyd 3 4 2)

With these notations, we get a formal description of the first three rounds.

**Lemma** `L_2_3` : rounds standard 2 (hyd1 h3) 3 (hyd 3 0 0).

**Proof.**

```
left; trivial; right ; simpl; left; left.
split; right; right; left.
```

**Qed.**

**Lemma** `L_0_3` : rounds standard 0 hinit 3 (hyd 3 0 0).

**Proof.**

```
apply rounds_trans with (1:= L_2_3) (2:= L_0_2).
```

**Qed.**

### 2.3.3 Testing ...

In order to make the study of this battle easier, we will use a simple data type for representing a configuration (*round*, *hyd*  $n_2$   $n_1$   $n_h$ ) as the 4-tuple

```
Record state : Type :=
  mks {round : nat ; n2 : nat ; n1 : nat ; nh : nat}.
```

The following function returns the next configuration of the game. Note that this function is defined only for making experiments and is not “certified”. Formal proofs about our battle only start with the lemma `step_rounds`, page 40.

```
Definition next (s : state) :=
  match s with
  | mks round a b (S c) => mks (S round) a b c
  | mks round a (S b) 0 => mks (S round) a b (S round)
  | mks round (S a) 0 0 => mks (S round) a (S round) 0
  | _ => s
  end.
```

We can make bigger steps through iterations of `next`. The functional `iterate`, similar to Standard Library’s `Nat.iter`, is defined and studied in `Prelude.Iterates`.

```
Fixpoint iterate {A:Type}(f : A -> A) (n: nat)(x:A) :=
  match n with
  | 0 => x
  | S p => f (iterate f p x)
  end.
```

The following function computes the state of the battle at the  $n$ -th round.

```
Definition test n := iterate next (n-3) (mks 3 3 0 0).
```

**Compute** test 3.

```
= {| round := 3; n2 := 3; n1 := 0; nh := 0 |}
: state
```

**Compute** test 4.

```
= {| round := 4; n2 := 2; n1 := 4; nh := 0 |}
: state
```

**Compute** test 5.

```
= {| round := 5; n2 := 2; n1 := 3; nh := 5 |}
: state
```

**Compute** test 2000.

```
= {|
  round := 2000; n2 := 1; n1 := 90; nh := 1102
|}
: state
```

The battle we are studying looks to be awfully long. Let us concentrate our tests on some particular events : the states where  $n_h = 0$ . From the value of test 5, it is obvious that at the 10-th round, the counter `nh` is equal to zero.

**Compute** test 10.

```
= {| round := 10; n2 := 2; n1 := 3; nh := 0 |}
: state
```

Thus,  $(1 + 11)$  rounds later, the `n1` field is equal to 2, and `nh` to 0.

**Compute** test 22.

```
= {| round := 22; n2 := 2; n1 := 2; nh := 0 |}
: state
```

**Compute** test 46.

```
= {| round := 46; n2 := 2; n1 := 1; nh := 0 |}
: state
```

**Compute** test 94.

```
= {| round := 94; n2 := 2; n1 := 0; nh := 0 |}
: state
```

Next round, we decrement `n2` and set `n1` to 95.

**Compute** test 95.

```
= {| round := 95; n2 := 1; n1 := 95; nh := 0 |}
: state
```

We now have some intuition of the sequence. It looks like the next “`nh=0`” event will happen at the  $192 = 2(95 + 1)$ -th round, then at the  $2(192 + 1)$ -th round, etc.

**Definition** `doubleS (n : nat) := 2 * (S n)`.

**Compute** test `(doubleS 95)`.

```
= {| round := 192; n2 := 1; n1 := 94; nh := 0 |}
: state
```

**Compute** test `(iterate doubleS 2 95)`.

```
= {| round := 386; n2 := 1; n1 := 93; nh := 0 |}
: state
```

### 2.3.4 Proving ...

We are now able to reason about the sequence of transitions defined by our hydra battle.

Let us define a binary relation associated with every round of the battle. In the following definition  $i$  is associated with the round number (or date, if we consider a discrete time), and  $a, b, c$  respectively associated with the number of occurrences of  $h_2, h_1$  and heads connected to the hydra's foot. For convenience<sup>7</sup>, we do not use the type `state` of the preceding section, but consider the round numbers and the number of hydras  $h_2, h_1$  and heads as separate arguments of the relation (which is no more —formally— “binary”).

```
Inductive one_step (i: nat) :
  nat -> nat -> nat -> nat -> nat -> nat -> Prop :=
| step1: forall a b c, one_step i a b (S c) a b c
| step2: forall a b, one_step i a (S b) 0 a b (S i)
| step3: forall a, one_step i (S a) 0 0 a (S i) 0.
```

The relation between `one_step` and the rules of hydra battles is asserted by the following lemma.

```
Lemma step_rounds : forall i a b c a' b' c',
  one_step i a b c a' b' c' ->
  rounds standard i (hyd a b c) (S i) (hyd a' b' c').
```

Next, we define “big steps” as the transitive closure of `one_step`, and reachability (from the initial configuration of figure 2.8 at time 0).

```
Inductive steps : nat -> nat -> nat -> nat ->
  nat -> nat -> nat -> nat -> Prop :=
| steps1 : forall i a b c a' b' c',
  one_step i a b c a' b' c' -> steps i a b c (S i) a' b' c'
| steps_S : forall i a b c j a' b' c' k a'' b'' c'',
  steps i a b c j a' b' c' ->
  steps j a' b' c' k a'' b'' c'' ->
  steps i a b c k a'' b'' c''.
```

```
(** reachability (for i > 0) *)
```

```
Definition reachable (i a b c : nat) : Prop :=
  steps 3 3 0 0 i a b c.
```

The following lemma establishes a relation between `steps` and the predicate `rounds`.

```
Lemma steps_rounds : forall i a b c j a' b' c',
  steps i a b c j a' b' c' ->
  rounds standard i (hyd a b c) j (hyd a' b' c').
```

Thus, any result about `steps` will be applicable to standard battles. Using the predicate `steps`, our study of the length of the considered battle can be decomposed into three parts:

---

<sup>7</sup>In a few words, the type `state` was designed for performing *computations*, and `steps` for writing *interactive proofs*, inspired by the aforementioned computations.

1. Characterization and proofs of regularities of some events (inspired by our experiments of Sect. 2.3.3).
2. Study of the beginning of the battle
3. Computing the exact length of the battle.

First, we prove that, if at round  $i$  the hydra is equal to  $(\text{hyd } a \ (S \ b) \ 0)$ , then it will be equal to  $(\text{hyd } a \ b \ 0)$  at the  $2(i+1)$ -th round.

**Lemma LS** : forall c a b i, steps i a b (S c) (i + S c) a b 0.

**Proof.**

```

induction c.
- intros; replace (i + 1) with (S i) by ring.
  repeat constructor.
- intros; eapply steps_S.
  + eleft; apply step1.
  + replace (i + S (S c)) with (S i + S c) by ring; apply IHc.

```

**Qed.**

(\*\* The law that relates two consecutive events with  $(nh = 0)$  \*)

**Lemma doubleS\_law** : forall a b i, steps i a (S b) 0 (doubleS i) a b 0.

**Proof.**

```

intros; eapply steps_S.
+ eleft; apply step2.
+ unfold doubleS; replace (2 * S i) with (S i + S i) by ring;
  apply LS.

```

**Qed.**

**Lemma reachable\_S** : forall i a b, reachable i a (S b) 0 ->  
 reachable (doubleS i) a b 0.

**Proof.** intros; right with (1 := H); apply doubleS\_law. **Qed.**

From now on, the lemma `reachable_S` allows us to watch larger and larger steps of the battle.

**Lemma L4** : reachable 4 2 4 0.

**Proof.** left; constructor. **Qed.**

**Lemma L10** : reachable 10 2 3 0.

**Proof.** change 10 with (doubleS 4); apply reachable\_S, L4. **Qed.**

**Lemma L22** : reachable 22 2 2 0.

**Proof.** change 22 with (doubleS 10); apply reachable\_S, L10. **Qed.**

**Lemma L46** : reachable 46 2 1 0.

**Proof.** change 46 with (doubleS 22); apply reachable\_S, L22. **Qed.**



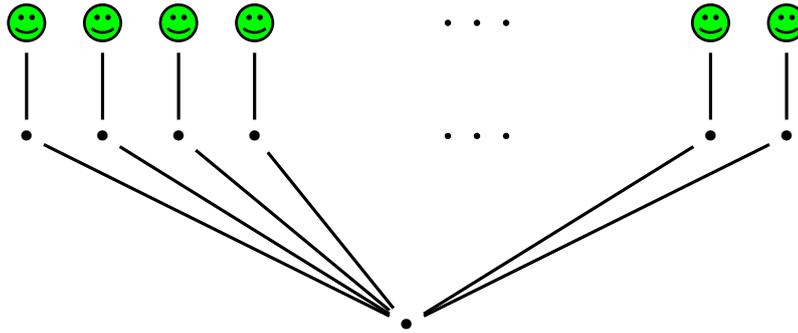


Figure 2.12

The state of the hydra after  $M + 1$  rounds (with  $M + 1$  heads).

We are now able to prove formally that the considered battle is composed of  $N$  steps.

**Lemma Almost\_done :**  
 rounds standard 3 (hyd 3 0 0) N (hyd 0 0 0).

**Proof.** apply steps\_rounds, SuperbigStep. Qed.

**Theorem Done :**  
 rounds standard 0 hinit N head.

**Proof.** eapply rounds\_trans with (1:= Almost\_done) (2:= L\_0\_3). Qed.

### 2.3.6 A minoration lemma

Now, we would like to get an intuition of how big the number  $N$  is. For that purpose, we use a minoration of the function `doubleS` by the function `(fun n => 2 * n)`.

```
Fixpoint exp2 (n:nat) : nat :=
  match n with
  | 0 => 1
  | S i => 2 * exp2 i
  end.
```

Using a few facts (proven in `hydras.Hydra.BigBattle`), we get several minorations.

**Lemma minoration\_0 :** forall n, 2 \* n <= doubleS n.

**Lemma minoration\_1 :** forall n x, exp2 n \* x <= iterate doubleS n x.

**Lemma minoration\_2 :** exp2 95 \* 95 <= M.

**Lemma minoration\_3 :** exp2 (S M) \* S M <= N.

**Lemma minoration :** exp2 (exp2 95 \* 95) <= N.

The number  $N$  is greater than or equal to  $2^{2^{95} \times 95}$ . If we write  $N$  in base 10,  $N$  would require at least  $10^{30}$  digits!

## 2.4 Generic properties

The example we just studied shows that the termination of any battle may take a very long time. If we want to study hydra battles in general, we have to consider any hydra and any strategy, both for Hercules and the hydra itself. So, we first give some definitions, generally borrowed from transition systems vocabulary (see [Tel00] for instance).

### 2.4.1 Liveliness

Let  $B$  be an instance of `Battle`. We say that  $B$  is *alive* if for any configuration  $(i, h)$ , where  $h$  is not a head, there exists a further step in class  $B$ .

*From Module Hydra.Hydra\_Definitions*

```
Definition Alive (B : Battle) :=
  forall i h, h <> head -> {h' : Hydra | battle_rel B i h h'}.
```

The theorems `Alive_free` and `Alive_standard` of the module `Hydra.Hydra_Theorems` show that the classes `free` and `standard` satisfy this property.

*From Module Hydra.Hydra\_Lemmas*

```
(** If the hydra is not reduced to a head, there exists at
   least one head that Hercules can chop off **)
```

```
Definition next_round_dec n (h: Hydra) :
  (h = head) + {h' : Hydra & {R1 h h'}} + {R2 n h h'}.
```

*From Module Hydra.Hydra\_Theorems*

**Corollary** `Alive_free` : Alive free.

**Corollary** `Alive_standard` : Alive standard.

### 2.4.2 Termination

The termination of *any* battle is naturally expressed by the predicate `well_founded` defined in the module `Coq.Init.Wf` of the Standard Library.

```
Definition Termination := well_founded (transp _ round).
```

Let  $B$  be an instance of class `Battle`. A *variant* for  $B$  consists in a well-founded relation  $<$  on some type  $A$ , and a function (also called a *measure*)  $m : \text{Hydra} \rightarrow A$  such that for any successive steps  $(i, h)$  and  $(1 + i, h')$  of a battle in  $B$ , the inequality  $m(h') < m(h)$  holds.

*From Module Hydra.Hydra\_Definitions*

```

Class Hvariant {A:Type}{Lt:relation A}
  (Wf: well_founded Lt)(B : Battle)
  (m: Hydra -> A): Prop :=
{variant_decr: forall i h h',
  h <> head -> battle_rel B i h h' -> Lt (m h') (m h)}.

```

**Exercise 2.6** Prove that, if there exists some instance of `(Hvariant Lt wf_Lt B m)`, then there exists no infinite battle in  $B$ .

### 2.4.3 A small proof of impossibility

When one wants to prove a termination theorem with the help of a variant, one has to consider first a well-founded set  $(A, <)$ , then a strictly decreasing measure on this set. The following lemma shows that, if the order structure  $(A, <)$  is too simple, it is useless to look for a convenient measure, which simply no exists. Such kind of result is useful, because it saves you time and effort.

The best known well-founded order is the natural order on the set  $\mathbb{N}$  of natural numbers (the type `nat` of Standard Library). It would be interesting to look for some measure  $m : \text{nat} \rightarrow \text{nat}$  and prove it is a variant.

Unfortunately, we can prove that *no* instance of class `(WfVariant round Peano.lt m)` can be built, where  $m$  is *any* function of type `Hydra  $\rightarrow$  nat`.

Let us present the main steps of that proof, the script of which is in the module `Hydra/Omega_Small.v`<sup>8</sup>.

Let us assume there exists some variant  $m$  from `Hydra` into  $(\text{nat}, <)$  for proving the termination of all hydra battles.

#### Section Impossibility\_Proof.

```

(** Let us assume there exists a variant from Hydra into nat
    for proving the termination of all hydra battles
    [Omega] is an ordinal notation for the least infinite ordinal
    [omega], whose members are the natural numbers.
*)

```

```

Variable m : Hydra -> nat.
Context (Hvar : Hvariant Omega free m).

```

We define an injection  $\iota$  from the type `nat` into `Hydra`. For any natural number  $i$ ,  $\iota(i)$  is the hydra composed of a foot and  $i + 1$  heads at height 1. For instance, Fig. 2.13 represents the hydra  $\iota(3)$ .

```

Let iota (i: nat) := hyd_mult head (S i).

```

Let us consider now some hydra `big_h` out of the range of the injection  $\iota$  (see Fig. 2.14 on the following page).

```

Let big_h := hyd1 (hyd1 head).

```

<sup>8</sup>The name of this file means “the ordinal  $\omega$  is too small for proving the termination of [free] hydra battles”. In effect, the elements of  $\omega$ , considered as a set, are just the natural numbers (see next chapter for more details)

Figure 2.13: The hydra  $\iota(3)$ Figure 2.14  
The hydra `big_h`.

Using the functions  $m$  and  $\iota$ , we define a second hydra `small_h`, and show there is a one-round battle that transforms `big_h` into `small_h`. Please note that, due to the hypothesis `Hvar`, we are interested in the termination of *free* battles. There is no problem to consider a round with  $(m \text{ big\_h})$  as the replication factor.

```
Let small_h := iota (m big_h).
```

```
Fact big_to_small: forall i, battle_rel free i big_h small_h.
```

```
Proof.
```

```
exists (m big_h); right; repeat constructor.
```

```
Qed.
```

But, by hypothesis,  $m$  is a variant. Hence, we infer the following inequality.

```
Lemma m_lt : m small_h < m big_h.
```

```
Proof.
```

```
apply (variant_decr 0); auto with hydra.
```

```
discriminate.
```

```
Qed.
```

In order to get a contradiction, it suffices to prove the inequality  $m(\text{big\_h}) \leq m(\text{small\_h})$  i.e.,  $m(\text{big\_h}) \leq m(\iota(m(\text{big\_h})))$ .

```
Lemma m_ge : m big_h <= m small_h.
```

Intuitively, it means that, from any hydra of the form  $(\text{iota } i)$ , the battle will take (at least)  $i$  rounds. Thus the associated measure cannot be less than  $i$ . Technically, we prove this lemma by Peano induction on  $i$ .

- The base case  $i = 0$  is trivial
- Otherwise, let  $i$  be any natural number and assume the inequality  $i \leq m(\iota(i))$ .

1. But the hydra  $\iota(S(i))$  can be transformed in one round into  $\iota(i)$  (by losing its rightmost head, for instance)
2. Since  $m$  is a variant, we have  $m(\iota(i)) < m(\iota(S(i)))$ , hence  $i < m(\iota(S(i)))$ , which implies  $S(i) \leq m(\iota(S(i)))$ .

We are now ready to complete our impossibility proof.

```

induction i.
- auto with arith.
- apply Nat.le_lt_trans with (m (iota i)).
  (* ... *)
Qed.
Theorem Contradiction : False.
Proof. generalize m_lt, m_ge; intros; lia. Qed.

End Impossibility_Proof.

```

**Exercise 2.7** Prove that there exists no variant  $m$  from Hydra into nat for proving the termination of all *standard* battles.

### 2.4.3.1 Conclusion

In order to build a variant for proving the termination of all hydra battles, we need to consider order structures more complex than the usual order on type nat. The notion of *ordinal number* provides a catalogue of well-founded order types. For a reasonably large bunch of ordinal numbers, *ordinal notations* are data-types which allow the Coq user to define functions, to compute and prove some properties, for instance by reflection.

The next chapter is dedicated to a generic formalization of ordinal notations, and chapter 4 to a proof of termination of all hydra battles with the help of an ordinal notation for the interval  $[0, \epsilon_0]$ <sup>9</sup>.

---

<sup>9</sup>We use the mathematical notation  $[a, b)$  for the interval  $\{x \mid a \leq x < b\}$ .



## Chapter 3

# Introduction to ordinal numbers and ordinal notations

The proof of termination of all hydra battles presented in [KP82] is based on *ordinal numbers*. From a mathematical point of view, an ordinal is a representative of an equivalence class for isomorphisms of totally ordered well-founded sets.

For the computer scientist, ordinals are tools for proving the totality of a given recursive function, or termination of a transition system. *Ordinal arithmetic* provides a set of functions whose properties, like *monotony*, allow to define *variants*, *i.e.* strictly decreasing measures used in proofs of termination.

Let us have a look at Figure 3.1. It presents a few items of a sequence of ordinal numbers, which extends the sequence of natural numbers.

Let us comment some features of this figure:

- The ordinals are listed in a strictly increasing order.
- Dots : “...” stand for infinite sequences of ordinals, not shown for lack of space. For instance, the ordinal  $\omega^2$  is not shown in the first line, but it exists, somewhere between 17 and  $\omega$ .
- Each ordinal printed in black is the immediate successor of another ordinal. We call it a *successor* ordinal. For instance, 12 is the successor of 11, and  $\omega^4 + 1$  the successor of  $\omega^4$ .
- Ordinals (displayed in red) that follow immediately dots are called *limit ordinals*. With respect to the order induced by this sequence, any limit ordinal  $\alpha$  is the least upper bound of the set  $\mathbb{O}_\alpha$  of all ordinals strictly less than  $\alpha$ . For instance,  $\omega$  is the least upper bound of the set of all finite ordinals (in the first line). It is also the first limit ordinal, and the first *infinite ordinal number*, in the sense that the set  $\mathbb{O}_\omega$  is infinite.
- The ordinal  $\epsilon_0$  is the first number which is equal to its own exponential of base  $\omega$ . It plays an important role in proof theory, and is particularly studied in chapters 4 to 6.

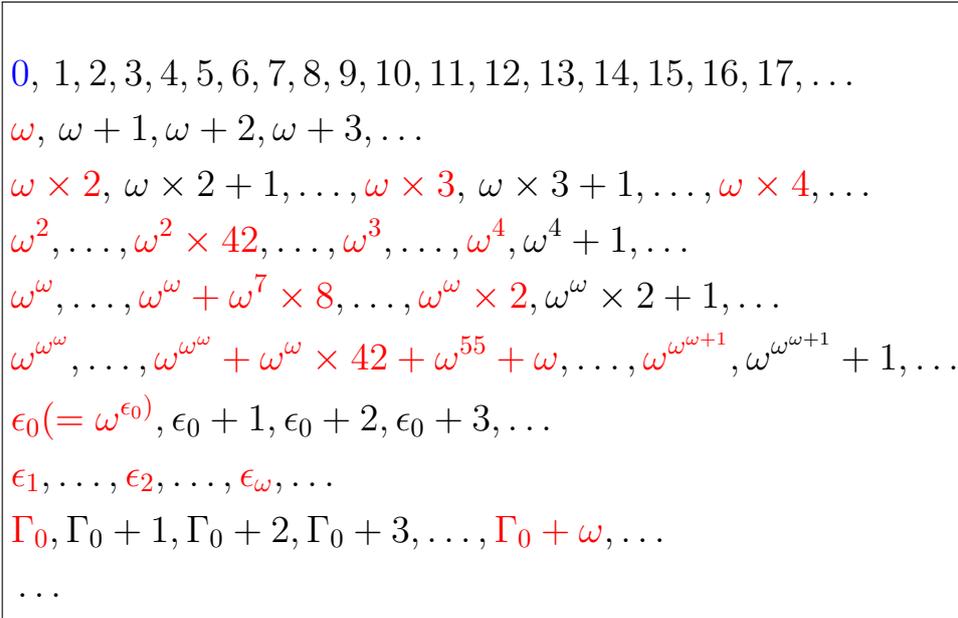


Figure 3.1: A short overview of the sequence of ordinal numbers

- Any ordinal is either the ordinal  $0$ , a successor ordinal, or a **limit ordinal**.

### 3.1 The mathematical point of view

We cannot cite all the literature published on ordinals since Cantor's book [Can55], and leave it to the reader to explore the bibliography. The introduction of José Grimm's report [Gri13] contains also a nice presentation of the main properties of ordinals.

For simplicity's sake, we will only give the definitions which are useful for understanding our Coq development.

#### 3.1.1 Well-ordered sets

Let us start with some definitions. A *well-ordered set* is a set provided with a binary relation  $<$  which has the following properties.

**irreflexivity** :  $\forall x \in A, x \not< x$

**transitivity** :  $\forall x y z \in A, x < y \Rightarrow y < z \Rightarrow x < z$

**trichotomy** :  $\forall x y \in A, x < y \vee x = y \vee y < x$

**well foundedness** :  $<$  is well-founded (every element of  $A$  is accessible)<sup>1</sup>.

<sup>1</sup>In classical mathematics, we would say that there is no infinite sequence  $a_1 > a_2 > \dots a_n > a_{n+1} \dots$  in  $A$ . In contrast, Coq's standard library contains an inductive definition of a predicate `Acc` which allows us to write constructive proofs of accessibility (See `Coq.Init.Wf`).

The best known examples of well-ordered sets are the set  $\mathbb{N}$  of natural numbers (with the usual order  $<$ ), as well as any finite segment  $[0, i) = \{j \in \mathbb{N} \mid j < i\}$ . The disjoint union of two copies of  $\mathbb{N}$ , *i.e.* the set  $\{0, 1\} \times \mathbb{N}$  is also well-ordered, with respect to the order below:

$$(i, j) < (i, k) \text{ \textbf{if} } j < k \\ (0, k) < (1, l) \text{ \textbf{for any } } k \text{ \textbf{and} } l$$

### 3.1.2 Ordinal numbers

Let  $(A, <_A)$  and  $(B, <_B)$  two well-ordered sets.  $A$  and  $B$  are said to have *the same order type* if there exists a strictly monotonous bijection  $b$  from  $A$  to  $B$ , *i.e.* which verifies the proposition  $\forall x y \in A, x <_A y \Rightarrow b(x) <_B b(y)$ .

Having the same order type is an equivalence relation between well-ordered sets. Ordinal numbers (in short: *ordinals*) are descriptions (*names*) of the associated equivalence classes. For instance, the order type of  $(\mathbb{N}, <)$  is associated with the ordinal called  $\omega$ , and the order we considered on the disjoint union of two copies of  $\mathbb{N}$  is associated with  $\omega \times 2$  (a.k.a.  $\omega + \omega$ ).

In a set-theoretic framework, one can consider any ordinal  $\alpha$  as a well-ordered set, whose elements are just the ordinals strictly less than  $\alpha$ , *i.e.* the *segment*  $\mathbb{O}_\alpha = [0, \alpha)$ . So, one can speak about *finite*, *infinite*, *countable*, etc., ordinals. Nevertheless, since we work within type theory, we do not identify ordinals as sets of ordinals, but consider the correspondence between ordinals and sets of ordinals as the function that maps  $\alpha$  to  $\mathbb{O}_\alpha$ . For instance  $\mathbb{O}_\omega = \mathbb{N}$ , and  $\mathbb{O}_7 = \{0, 1, 2, 3, 4, 5, 6\}$ .

## 3.2 Ordinal numbers in Coq

Two kinds of representation of ordinals are defined in our development.

- A “mathematical” representation of the set of countable ordinal numbers, after Kurt Schütte [Sch77]. This representation uses several (hopefully harmless) axioms. We use it as a reference for proving the correctness of ordinal notations.
-  We are also progressively importing Gaia’s [GQS] definitions and theorems, based on Bourbaki’s set theory [Gri16]. Chapter 7 is dedicated to the connexion between libraries Hydra-battles and Gaia.
- A family of *ordinal notations* (also called [*ordinal*] *notation systems*), *i.e.* data types used to represent segments  $[0, \mu)$ , where  $\mu$  is some countable ordinal. Each ordinal notation is defined inside the Calculus of Inductive Constructions (without axioms). Many functions are defined, allowing proofs by computation. Note that proofs of correctness of a given ordinal notation with respect to Schütte’s model obviously use axioms. Please execute the `Print Assumptions` command in case of doubt.

It is interesting to compare proofs of a given property (for instance the associativity of addition) both in the computational framework of some ordinal notation, and in mathematical models of Schütte or Gaia.

### 3.3 Ordinal Notations

Fortunately, the ordinals we need for studying hydra battles are much simpler than Schütte's, and can be represented as quite simple data types in Gallina.

Let  $\alpha$  be some (countable) ordinal; we call *ordinal notation for  $\alpha$*  any structure composed of:

- A data type  $A$  for representing all ordinals strictly below  $\alpha$ ,
- A well founded order  $<$  on  $A$ ,
- A correct function for comparing two ordinals. Note that the reflexive closure of  $<$  is thus a *total order*.

Such a structure should be proved correct relatively to a bigger ordinal notation or to Schütte's or Gaia's model.

#### 3.3.1 Classes for ordinal notations

From the Coq user's point of view, an ordinal notation is a structure allowing to compare two ordinals by computation, and proving by well-founded induction.

##### 3.3.1.1 The Comparable class

The `Comparable` class, contributed by Jérémy Damour and Théo Zimmermann, allows us to apply generic lemmas and tactics about decidable strict orders. The correctness of the comparison function is expressed through Stdlib's type `Datatypes.CompareSpec` and predicate `Datatypes.CompSpec`.

```

Inductive CompareSpec (Peq Plt Pgt : Prop) :
  comparison -> Prop :=
  | CompEq : Peq -> CompareSpec Peq Plt Pgt Eq
  | CompLt : Plt -> CompareSpec Peq Plt Pgt Lt
  | CompGt : Pgt -> CompareSpec Peq Plt Pgt Gt.

Definition CompSpec {A} (eq lt : A -> A -> Prop) (x y : A) :=
  CompareSpec (eq x y) (lt x y) (lt y x).

```

*From Module Prelude.Comparable*

```
Class Compare (A:Type) := compare : A -> A -> comparison.
```

```
Class Comparable {A:Type} (lt: relation A) (cmp : Compare A) :=
{
  comparable_sto := StrictOrder lt;
  comparable_comp_spec : forall (a b : A), CompSpec eq lt a b (compare a b)
}.

```

##### 3.3.1.2 The ON class

The following class definition, parameterized with a type  $A$ , a binary relation  $lt$  on  $A$ , specifies that  $lt$  is a well-founded strict order, provided with a correct comparison function.

*From Library OrdinalNotations.ON\_Generic*

```

Class ON {A:Type} (lt: relation A) (cmp: Compare A) :=
{
  ON_comp :=> Comparable lt cmp;
  ON_wf : well_founded lt;
}.

#[global] Existing Instance ON_comp.
Coercion ON_wf : ON >-> well_founded.

Definition rep {A:Type} {lt: relation A} {cmp: Compare A}
  (on : ON lt cmp) := A.

#[global] Coercion rep : ON >-> Sortclass.

```

We give also a few handy definitions and lemmas for any ordinal notation.

#### Section Definitions.

```

Context {A:Type} {lt : relation A} {cmp : Compare A} {on: ON lt cmp}.

#[using="All"] Definition ON_t := A.

#[using="All"] Definition ON_compare : A -> A -> comparison := compare.

#[using="All"] Definition ON_lt := lt.

#[using="All"] Definition ON_le: relation A := leq lt.

#[using="All"]
Definition measure_lt {B : Type} (m : B -> A) : relation B :=
  fun x y => ON_lt (m x) (m y).

#[using="All"]
Lemma wf_measure {B : Type} (m : B -> A) :
  well_founded (measure_lt m).

#[using="All"]
Definition ZeroLimitSucc_dec :=
  forall alpha,
    {Least alpha} +
    {Limit alpha} +
    {beta: A | Successor alpha beta}.

(** The segment called [0 alpha] in Schutte's book *)

#[using="All"]
Definition big0 (a: A) : Ensemble A := fun x: A => lt x a.

```

#### End Definitions.

```

Infix "o<" := ON_lt : ON_scope.

```

```
Infix "o<=" := ON_le : ON_scope.
Infix "o?=" := ON_compare (at level 70) : ON_scope.
```

**Remark 3.1** The infix notations `o<` and `o<=` are defined in order to make apparent the distinction between the various notation scopes that may co-exist in a same statement. So the infix `<` and `<=` are reserved to the natural numbers. In mathematical formulas, however, we still use `<` and `≤` for inequalities between ordinals.

### 3.4 Example: the ordinal $\omega$

The simplest example of ordinal notation is built over the type `nat` of Coq's standard library. We just have to apply already proven lemmas about Peano numbers.

*From Library OrdinalNotations.ON\_Omega*

```
#[global] Instance Omega_comp : Comparable Peano.lt compare_nat.
```

```
Proof.
```

```
split.
- apply Nat.lt_strorder.
- apply Nat.compare_spec.
```

```
Qed.
```

```
#[global] Instance Omega : ON Peano.lt compare_nat.
```

```
Proof.
```

```
split.
- apply Omega_comp.
- apply Wf_nat.lt_wf.
```

```
Qed.
```

```
#[local] Open Scope ON_scope.
```

```
Compute 6 o?= 9.
```

```
= Lt
: comparison
```

### 3.5 Sum of two ordinal notations

Let `NA` and `NB` be two ordinal notations, on the respective types `A` and `B`.

We consider a new strict order on the disjoint sum of the associated types, by putting all elements of `A` before the elements of `B` (thanks to Standard Library's relation operator `le_ASB`).

*From Library Relations.Relation\_Operators.*

```
Inductive
le_ASB (A B : Type) (leA : A -> A -> Prop) (leB : B -> B -> Prop)
: A + B -> A + B -> Prop :=
| le_aa : forall x y : A, leA x y -> le_ASB A B leA leB (inl x) (inl y)
```

```
| le_ab : forall (x : A) (y : B), le_AsB A B leA leB (inl x) (inr y)
| le_bb : forall x y : B, leB x y -> le_AsB A B leA leB (inr x) (inr y)
```

From Library OrdinalNotations.ON\_plus

### Section Defs.

```
Context `(ltA: relation A)
          (cmpA : Compare A)
          (NA: ON ltA cmpA).
```

```
Context `(ltB: relation B)
          (cmpB : Compare B)
          (NB: ON ltB cmpB).
```

```
Definition t := (A + B)%type.
```

```
Arguments inl {A B} _.
```

```
Arguments inr {A B} _.
```

```
Definition lt : relation t := le_AsB _ _ ltA ltB.
```

In order to build an instance of Comparable, we have to define a correct comparison function.

```
#[global] Instance compare_plus : Compare t :=
```

```
fun (alpha beta: t) =>
  match alpha, beta with
  | inl _, inr _ => Lt
  | inl a, inl a' => compare a a'
  | inr b, inr b' => compare b b'
  | inr _, inl _ => Gt
  end.
```

```
Lemma compare_correct alpha beta :
  CompSpec eq lt alpha beta (compare alpha beta).
(* ... *)
```

```
#[global] Instance plus_comp : Comparable lt compare_plus.
```

```
Proof. split; [apply lt_strorder | apply compare_correct]. Qed.
```

Standard library's lemma Wellfounded.Disjoint\_Union.wf\_disjoint\_sum is applied to prove that our order lt is well-founded, allowing us to build an instance of ON:

```
Lemma lt_wf : well_founded lt.
```

```
Proof.
```

```
  destruct NA, NB; apply wf_disjoint_sum; [apply ON_wf | apply ON_wf0].
```

```
Qed.
```

```
#[global] Instance ON_plus : ON lt compare_plus.
```

```
Proof. split; [apply plus_comp | apply lt_wf]. Qed.
```

### 3.5.1 Example: The ordinal $\omega + \omega$

The ordinal  $\omega + \omega$  (also known as  $\omega \times 2$ ) may be represented as the concatenation of two copies of  $\omega$  (Figure 3.2). It is also represented by the two first lines of Figure 3.1. We define this notation in Coq as an instance of `ON_plus`.



Figure 3.2:  $\omega + \omega$

From Module `OrdinalNotations.ON_Omega_plus_omega`

```
#[global] Instance compare_nat_nat : Compare t :=
  compare_plus compare_nat compare_nat.

#[global] Instance Omega_plus_Omega : ON _ compare_nat_nat :=
  ON_plus Omega Omega.

Definition t := ON_t.

Compute inl 42 o? = inr 0.

Example ex1 : inl 7 o < inr 8.
Proof. apply compare_lt_iff. trivial. Qed.
```

We can now define abbreviations. For instance, the finite ordinals are represented by terms built with the constructor `inl`, and the first infinite ordinal  $\omega$  by the term `(inr 0)`.

```
Definition fin (i:nat) : t := inl i.
Coercion fin : nat -> t.
```

```
Notation omega := (inr 0:ON_t).
```

```
Compute fin 8 o? = omega.
```

```
= Lt
: comparison
```

```
Lemma lt_omega alpha :
  alpha o < omega <-> exists n:nat, alpha = fin n.
```

## 3.6 Limits and successors

Let us look again at our implementation of  $\omega + \omega$ . We can classify its elements into three categories:

- The least ordinal,  $(\text{inl } 0)$ , also known as  $(\text{fin } 0)$ .
- The first infinite ordinal  $\omega$ .
- The remaining ordinals, either of the form  $(\text{inl } (S i))$  or  $(\text{inr } (S i))$  (in black on Figure 3.1), called *successor ordinals*.

### 3.6.1 Definitions

It would be interesting to specify at the most generic level, what is a zero, a successor or a limit ordinal. Let  $<$  be a strict order on a type  $A$ .

- A *least* element is a minorant (in the large sense) of the full set on  $A$ ,
- $y$  is a *successor* of  $x$  if  $x < y$  and there is no element between  $x$  and  $y$ . We will also say that  $x$  is a *predecessor* of  $y$ .
- $x$  is a *limit* if  $x$  is not a least element, and for any  $y$  such that  $y < x$ , there exists some  $z$  such that  $y < z < x$ .

The following definitions are in Library Prelude.MoreOrders.

#### Section Defs.

```

Variables (A : Type)
          (lt: relation A).

#[local] Infix "<" := lt.
#[local] Infix "<=" := (leq lt).

Definition Least {sto : StrictOrder lt} (x : A) :=
  forall y, x <= y.

Definition Successor {sto : StrictOrder lt} (y x : A) :=
  x < y /\ (forall z, x < z -> z < y -> False).

Definition Limit {sto : StrictOrder lt} (x:A) :=
  (exists w:A, w < x) /\
  (forall y:A, y < x -> exists z:A, y < z /\ z < x).

Definition Omega_limit
  {sto : StrictOrder lt} (s: nat -> A) (x:A) :=
  (forall i: nat, s i < x) /\
  (forall y, y < x -> exists i:nat, y < s i).

```

**Exercise 3.1** Prove, that, in any ordinal notation system, every ordinal has at most one predecessor, and at most one successor.

*You may start this exercise with the file `exercises/ordinals/predSuccUnicity.v`.*

**Exercise 3.2** Prove, that, in any ordinal notation system, if  $\beta$  is a successor of  $\alpha$ , then for any  $\gamma$ ,  $\gamma < \beta$  implies  $\gamma \leq \alpha$ .

*You may start this exercise with the file `exercises/ordinals/lt_succ_le.v`.*

### 3.6.2 Limits and successors in $\omega + \omega$

Using the definitions above, we can prove the following lemma:

*From Module OrdinalNotations.ON\_Omega\_plus\_omega*

**Lemma** `limit_iff` (`alpha` : `t`) : `Limit alpha <-> alpha = omega`.

Regarding successors, let us define the following function and prove its correctness:

**Definition** `succ` (`alpha` : `t`) :=  
`match alpha with`  
`inl n => inl (S n)`  
`| inr n => inr (S n)`  
`end.`

**Lemma** `succ_correct` `alpha beta` :  
`Successor beta alpha <-> beta = succ alpha`.

We can also check whether an ordinal is a successor by a simple computation:

**Definition** `succb` (`alpha`: `t`) : `bool`  
`:= match alpha with`  
`| inr (S _) | inl (S _) => true`  
`| _ => false`  
`end.`

**Lemma** `succb_correct` (`alpha`: `t`) :  
`succb alpha <-> exists beta: t, alpha = succ beta`.

Finally, the nature of any ordinal is decidable (inside this notation system) :  
*From Module OrdinalNotations.ON\_Omega\_plus\_omega*

**Definition** `Zero_limit_succ_dec` : `ON_Generic.ZeroLimitSucc_dec`.  
`(* ... *)`

## 3.7 Product of ordinal notations

Let `NA` and `NB` be two ordinal notations, on the respective ordered types `A` and `B`. The product of `NA` and `NB` is considered as the concatenation of `B` copies of `A`, ordered by the lexicographic order on  $B \times A$ .

In Coq, we build an instance of class `ON` through a sequence of steps as for the sum of ordinal notations.

*From Module OrdinalNotations.ON\_mult*

**Section** `Defs`.

**Context** `(ltA: relation A)  
(cmpA : Compare A)  
(NA: ON ltA cmpA)

```

      `(ltB : relation B)
      (cmpB : Compare B)
      (NB: ON ltB cmpB).

Definition t := (B * A)%type.
Definition lt : relation t := lexico ltB ltA.
Definition le := clos_refl _ lt.

#[global] Instance compare_mult : Compare t :=
  fun (alpha beta: t) =>
    match compare (fst alpha) (fst beta) with
    | Eq => compare (snd alpha) (snd beta)
    | c => c
    end.

#[global] Instance mult_comp: Comparable lt compare_mult.
Proof.
  split.
  - apply lt_strorder.
  - apply compare_correct.
Qed.

#[global] Instance ON_mult: ON lt compare_mult.
Proof.
  split.
  - apply mult_comp.
  - apply lt_wf.
Qed.

End Defs.

```

### 3.8 The ordinal $\omega^2$

The ordinal  $\omega^2$  (also called  $\phi_0(2)$ , see Chap. 8), is an instance of the multiplication presented in the preceding section.

*From Module OrdinalNotations.ON\_Omega2*

```

#[ global ] Instance compare_omega2 : Compare ON_mult.t :=
  compare_mult compare_nat compare_nat.

#[ global ] Instance Omega2: ON _ compare_omega2 := ON_mult Omega Omega.

Definition t := ON_t.
Notation omega := (1,0).
Definition zero: t := (0,0).

Definition fin (i:nat) : t := (0,i).
Coercion fin : nat >-> t.

Example ex1 : (5,8) o< (5,10).
Proof. right; auto with arith. Qed.

```

### 3.8.1 Arithmetic on $\omega^2$

#### 3.8.1.1 Successor

The successor of any ordinal is defined by a simple pattern-matching.

**Definition** `succ (alpha : t) := (fst alpha, S (snd alpha))`.

This function is proved to be correct w.r.t. the Successor predicate.

**Lemma** `succ_ok alpha beta :`  
 Successor beta alpha  $\leftrightarrow$  beta = succ alpha.  
 (\* ... \*)

**Lemma** `lt_succ_le alpha beta :`  
 alpha o< beta  $\leftrightarrow$  succ alpha o<= beta.  
 (\* ... \*)

**Lemma** `lt_succ alpha :` alpha o< succ alpha.

**Proof.**

`destruct alpha; right; cbn; abstract lia.`

**Qed.**

#### 3.8.1.2 Addition

We can define on `Omega2` an addition which extends the addition on `nat`. Please note that this operation is not commutative:

**Definition** `plus (alpha beta : t) : t :=`  
`match alpha,beta with`  
`| (0, b), (0, b') => (0, b + b')`  
`| (0,0), y => y`  
`| x, (0,0) => x`  
`| (0, _b), (S n', b') => (S n', b')`  
`| (S n, b), (S n', b') => (S n + S n', b')`  
`| (S n, b), (0, b') => (S n, b + b')`  
`end.`

**Infix** `"+" := plus : ON_scope.`

**Lemma** `plus_compat (n p: nat) :`  
`fin (n + p)%nat = fin n + fin p.`

**Proof.** `destruct n; reflexivity. Qed.`

**Compute** `3 + omega.`

```
= omega
: t
```

**Compute** `omega + 3.`

```
= (1, 3)
: t
```

**Example** `non_commutativity_of_plus`:  $\omega + 3 \neq 3 + \omega$ .

**Proof.** `discriminate. Qed.`

### 3.8.1.3 Finite multiplication

The restriction of ordinal multiplication to the segment  $[0, \omega^2)$  is not a total function. For instance  $\omega \times \omega = \omega^2$  is outside the set of represented values. Nevertheless, we can define two operations mixing natural numbers and ordinals.

`(** multiplication of an ordinal by a natural number *)`

```
Definition mult_fin_r (alpha : t) (p : nat): t :=
  match alpha, p with
  | (0,0), _ => zero
  | _, 0 => zero
  | (0, n), p => (0, n * p)
  | (n, b), n' => (n * n', b)
  end.
```

**Infix** "\*" := mult\_fin\_r : ON\_scope.

`(** multiplication of a natural number by an ordinal *)`

```
Definition mult_fin_l (n:nat)(alpha : t) : t :=
  match n, alpha with
  | 0, _ => zero
  | _, (0,0) => zero
  | n , (0,n') => (0, (n*n')%nat)
  | n, (n',p') => (n', (n * p')%nat)
  end.
```

**Example** `e1` :  $(\omega * 7 + 15) * 3 = \omega * 21 + 15$ .

**Proof.** `reflexivity. Qed.`

**Example** `e2` :  $\text{mult\_fin\_l } 3 (\omega * 7 + 15) = \omega * 7 + 45$ .

**Proof.** `reflexivity. Qed.`

Multiplication with a finite ordinal and addition are related through the following lemma:

**Lemma** `unique_decomposition` (alpha: t) :  
exists! i j : nat, alpha = omega \* i + j.

**Proof.**

`destruct alpha as [i j]; exists i; split.`

```
i, j: nat
-----
exists ! j0 : nat, (i, j) = omega * i + j0
-----
i, j: nat
-----
forall x' : nat,
(exists ! j0 : nat, (i, j) = omega * x' + j0) ->
i = x'
```

```
(* ... *)
```

**Qed.**

### 3.8.1.4 Example: Ackermann function

The famous Ackermann function can be defined with the `Equations` plug-in [SM19], with a measure towards  $\omega^2$ .

From `OrdinalNotations.ON_Omega2`.

```
From Equations Require Import Equations.
Section A_def.
```

```
Let m (x : nat * nat) : t := omega * fst x + snd x.
```

```
#[ local ] Instance WF : WellFounded (measure_lt m) :=
  wf_measure m.
```

```
Equations A (p : nat * nat) : nat by wf p (measure_lt m) :=
  A (0, j) := S j;
  A (S i, 0) := A(i, 1);
  A (S i, S j) := A(i, A(S i, j)).
```

**Next Obligation.**

```
(* ... *)
```

## 3.8.2 Yet another proof of impossibility

In Sect. 2.4.3 on page 45, we proved that there exists no variant from `Hydra` to `(nat, <)` (*i.e.* the ordinal  $\omega$ ) for proving the termination of all hydra battles. We prove now that the ordinal  $\omega^2$  is also insufficient for this purpose.

The proof we are going to comment has exactly the same structure as in Section 2.4.3. Nevertheless, the proof of technical lemmas is a little more complex, due to the structure of the lexicographic order on  $\mathbb{N} \times \mathbb{N}$ . Consider for instance that there exists an infinite number of ordinals between  $\omega$  and  $\omega \times 2$ .

The detailed proof script is in the file `theories/ordinals/Hydra/Omega2_Small.v`.

### 3.8.2.1 Preliminaries

Let us assume there is a variant from `Hydra` into  $\omega^2$  for proving the termination of all hydra battles.

*From Module `Hydra.Omega2_Small`*

```
Section Impossibility_Proof.
```

```
Variable m : Hydra -> rep Omega2.
```

```
Context
```

```
(Hvar: Hvariant Omega2 free m).
```

We follow the same pattern as in Sect. 2.4.3. First, we define an injection  $\iota$  from type `ON_Omega2.t` into `Hydra`, by associating to each ordinal  $\omega \times i + j = (i, j)$  the hydra with  $i$  branches of length 2 and  $j$  branches of length 1.

From Module *Hydra.Omega2\_Small*

```
Let iota (p: ON_Omega2.t) :=
  node (hcons_mult (hyd1 head) (fst p)
        (hcons_mult head (snd p) hnil)).
```

For instance, Figure 3.3 shows the hydra associated to the ordinal  $(3, 5)$ , a.k.a.  $\omega \times 3 + 5$ .

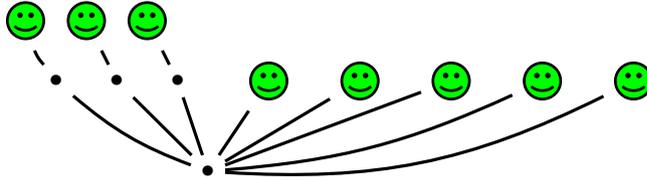


Figure 3.3: The hydra  $\iota(\omega \times 3 + 5)$

Like in Sect. 2.4.3, we build a hydra out of the range of `iota` (represented in Fig. 3.4).



Figure 3.4  
The hydra `big_h`.

```
Let big_h := hyd1 (hyd2 head head).
```

In a second step, we build a “smaller” hydra<sup>2</sup>.

```
Let small_h := iota (m big_h).
```

Like in Sect. 2.4.3, we prove the inequality  $m \text{ big\_h } o \leq m \text{ small\_h } o \leq m \text{ big\_h}$ , which is obviously false.

### 3.8.2.2 Proof of the inequality $m \text{ small\_h } o < m \text{ big\_h}$

In order to prove the inequality `m_lt`:  $m \text{ small\_h } o < m \text{ big\_h}$ , it suffices to build a battle transforming `big_h` into `small_h`.

First we prove that `small_h` is reachable from `big_h` in one or two steps. Let us decompose  $m \text{ big\_h}$  as  $(i, j)$ . If  $j = 0$ , then one round suffices to transform `big_h` into  $\iota(i, j)$ . If  $j > 0$ , then a first round transforms `big_h` into  $\iota(i + 1, 0)$  and a second round into  $\iota(i, j)$ . So, we have the following result.

<sup>2</sup>With respect to the measure  $m$ .

**Lemma** `big_to_small` : `big_h`  $\dashv\vdash$  `small_h`.

Since  $m$  is a variant, we infer the following inequality:

**Corollary** `m_lt` : `m small_h`  $\circ <$  `m big_h`.

**Proof.** `apply m_strict_mono with (1:=Hvar) (2:=big_to_small).` **Qed.**

### 3.8.2.3 Proof of the inequality `m big_h` $\circ \leq$ `m small_h`

The proof of the inequality `m big_h`  $\circ \leq$  `m small_h` is quite more complex than in Sect 2.4.3. If we consider any ordinal  $\alpha = (i, j)$ , where  $i > 0$ , there exists an infinite number of ordinals strictly less than  $\alpha$ , and there exists an infinite number of battles that start from  $\iota(\alpha)$ . Indeed, at any configuration  $\iota(k, 0)$ , where  $k > 0$ , the hydra can freely choose any replication number. Intuitively, the measure of such a hydra must be large enough for taking into account all the possible battles issued from that hydra. Let us now give more technical details.

The first steps of our proof start a well-founded induction over  $\omega^2$ .

**Lemma** `m_ge` : `m big_h`  $\circ \leq$  `m small_h`.

**Proof.**

```
unfold small_h;
pattern (m big_h);
  apply well_founded_induction with (R := ON_lt) (1:= ON_wf);
  intros (i,j) IHij.
```

```
m: Hydra -> Omega2
Hvar: Hvariant Omega2 free m
big_h: Hydra
iota: t -> Hydra
small_h: Hydra
i, j: nat
IHij: forall y : Omega2,
      y < (i, j) -> y <= m (iota y)
-----
(i, j) <= m (iota (i, j))
```

A case analysis on  $i$  and  $j$  leads us to consider three cases :

- $i = j = 0$ : the inequality is trivial.
- $i = 1 + l, j = 0$  ( $(i, j)$  is a limit ordinal): By the induction hypothesis `IHij`,  $(l, k) \circ \leq m(\iota(l, k))$  for any  $k$ . But (by the rules of the hydra game),  $\iota(i, 0)$  is transformed into any  $\iota(l, k)$  in one round. Thus  $m(\iota(l, k)) < m(\iota(i, 0))$  for any  $k$ . Therefore,  $(l, k) < m(\iota(i, 0))$  for any  $k$ , thus  $(i, 0) \circ \leq m(\iota(i, 0))$ .
- $j = l + 1$  ( $(i, j)$  is a successor). We apply the induction hypothesis to the pair  $(i, l)$ .

Please look at the proof script for more details.

(\* ... \*)

**Qed.**

### 3.8.2.4 End of the proof

From `m_ge`, we get  $m \text{ big\_h } \leq m \text{ small\_h } = m (\text{iota } (m \text{ big\_h}))$ . Since  $<$  is a strict order (irreflexive and transitive), this inequality is incompatible with the strict inequality  $m \text{ small\_h } < m \text{ big\_h}$  (lemma `m_lt`).

From Module *Hydra.Omega2\_Small*

**Theorem Impossible** : `False`.

**Proof.**

```
destruct (StrictOrder_Irreflexive (R:=ON_lt) (m big_h));
  eapply le_lt_trans; [apply m_ge | apply m_lt].
```

**Qed.**

**End Impossibility\_Proof.**

**Exercise 3.3** Prove that there exists no variant  $m$  from *Hydra* into  $\omega^2$  for proving the termination of all *standard* battles.

**Remark 3.2** We prove in Chapter 5 a generalization of the impossibility lemmas of Sect. 2.4.3 and this section, with the same proof structure, but with much more complex technical details.

## 3.9 A notation for finite ordinals

Let  $n$  be some natural number. The segment associated with  $n$  is the interval  $[0, n) = \{0, 1, \dots, n - 1\}$ . One may represent the ordinal  $n$  by a sigma type.

From Module *OrdinalNotations.ON\_Finite*

**Definition** `t` ( $n:\text{nat}$ ) :=  $\{i:\text{nat} \mid \text{Nat.ltb } i \ n\}$ .

**Definition** `lt`  $\{n:\text{nat}\}$  : relation (`t`  $n$ ) :=

```
fun alpha beta => Nat.ltb (proj1_sig alpha) (proj1_sig beta).
```

### 3.9.0.1 Examples

For instance, let us build two elements of the segment  $[0, 7)$ , *i.e.* two inhabitants of type `(t 7)`, and prove a simple inequality (see Fig. 3.5).

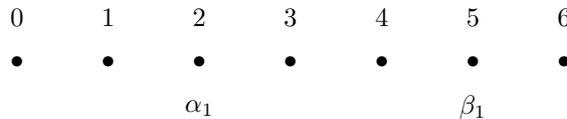


Figure 3.5: The segment  $\mathbb{O}_7$

**Program Example** `alpha1` : `t 7` := 2.

**Program Example** `beta1` : `t 7` := 5.

**Example i1** : lt alpha beta1.

**Proof.** reflexivity. Qed.

Note that the type  $(t \ 0)$  is empty, and that, for any natural number  $n$ ,  $n$  does not belong to  $(t \ n)$ .

**Lemma t0\_empty** (alpha: t 0): False.

**Proof.**

destruct alpha ; discriminate.

**Qed.**

**Definition bad** : t 10.

exists 10.

```
10 <? 10
```

**Abort.**

### 3.9.0.2 Comparison function

In order to build an instance of **ON**, we define a comparison function, then prove its correctness.

```
#[global] Instance compare_fin {n:nat} : Compare (t n) :=
  fun (alpha beta : t n) => Nat.compare (proj1_sig alpha) (proj1_sig beta).
```

**Lemma compare\_correct** {n} (alpha beta : t n) :

CompSpec eq lt alpha beta (compare alpha beta).

**Remark 3.3** The proof of `compare_correct` uses a well-known pattern of Coq. Let us consider the following subgoal.

```
n, x0: nat
i, i0: x0 <? S n
-----
exist (fun i : nat => i <=? n) x0 i =
exist (fun i : nat => i <=? n) x0 i0
```

Applying the tactic `f_equal` generates a simpler subgoal.

```
f_equal.
```

```
n, x0: nat
i, i0: x0 <? S n
-----
i = i0
```

We have now to prove that there exists at most one proof of  $(\text{Nat.ltb } x0 \ (S \ n))$ . This is not obvious, but a consequence of the following lemma of library `Coq.Logic.Eqdep_dec`.

```
eq_proofs_unicity_on :
forall (A : Type) (x : A),
(forall y : A, x = y \ / x <=> y) ->
forall (y : A) (p1 p2 : x = y), p1 = p2
```

Thus unicity of proofs of `Nat.ltb x0 (S n)` comes from the decidability of equality on type `bool`. This is why we used the boolean function `Nat.ltb` instead of the inductive predicate `Nat.lt` in the definition of type `t n` (see page 65). For more information about this pattern, please look at the numerous mailing lists and FAQs on Coq).

```
    apply eq_proofs_unicity_on; decide equality.
  (* ... *)
```

**Remark 3.4** Please note that attempting to compare a term of type `(t n)` with a term of type `(t p)` leads to an error if `n` and `p` are not convertible.

**Program Example** `gamma1 : t 8 := 7.`

**Fail Goal** `alpha1 < gamma1.`

```
The command has indeed failed with message:
The term "gamma1" has type "t 8"
while it is expected to have type "t 7".
```

### 3.9.0.3 Building an instance of `ON`

Applying lemmas of the libraries `Coq.Wellfounded.Inverse_Image`, `Coq.Wellfounded.Inclusion`, and `Coq.Arith.Wf_nat`, we prove that our relation `lt` is well founded.

**Lemma** `lt_wf (n:nat) : well_founded (@lt n).`

Now we can build our instance of `OrdinalNotation`.

```
#[global] Instance sto n : StrictOrder (@lt n).
```

```
#[global] Instance comp n : Comparable (@lt n) compare.
```

```
#[global] Instance FinOrd n : ON (@lt n) compare.
```

**Remark 3.5** It is important to keep in mind that the integer `n` is not an “element” of `FinOrd n`. In set-theoretic presentations of ordinals, the set associated with the ordinal `n` is  $\{0, 1, \dots, n - 1\}$ . In our formalization, the interpretation of an ordinal as a set is realized by the function `big0` (see Section 3.3.1.2 on page 53).

**Remark 3.6** There is no interesting arithmetic on finite ordinals, since functions like successor, addition, etc., cannot be represented in Coq as *total* functions.

### 3.9.0.4 See also ...

 Finite ordinals are also formalized in `SSReflect/MathComp` [MT18]. In Module `gaia_hydras.ON_gfinite`, we build an instance of class `ON`.

```

Definition finord_lt (n:nat) (alpha beta: 'I_n): bool :=
  (alpha < beta)%N.

#[global] Instance finord_compare (n:nat) : Compare ('I_n) :=
  fun alpha beta => Nat.compare alpha beta.
#[global] Instance finord_ON n : ON (@finord_lt n) (@finord_compare n).

#[program] Example o_33_of_42: 'I_42 := @Ordinal 42 33 _.

#[program] Example o_36_of_42: 'I_42 := @Ordinal 42 36 _.
Compute compare o_33_of_42 o_36_of_42.

```

```

= Lt
: comparison

```

### 3.10 Comparing two ordinal notations

It is sometimes useful to compare two ordinal notations with respect to expressive power (the segment of ordinals they represent).

The following class specifies a strict inclusion of segments. The ordinal notation **OA** describes a segment  $[0, \alpha)$ , and **OB** a larger segment (which contains a notation for  $\alpha$ , whilst  $\alpha$  is not represented in **OA**) (like in Fig. 3.6). We require also the comparison functions of the two notation systems to be compatible.

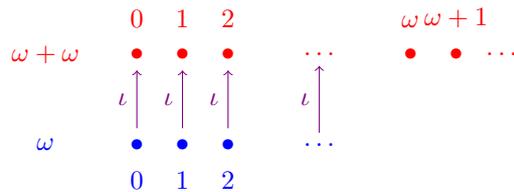


Figure 3.6:  $\omega$  is a sub-segment of  $\omega + \omega$

From ON\_Generic.

```

Class SubON
  `(OA: @ON A ltA compareA)
  `(OB: @ON B ltB compareB)
  (alpha: B) (iota: A -> B):=
{
  SubON_compare: forall x y : A,
    compareB (iota x) (iota y) = compareA x y;
  SubON_incl : forall x, ltB (iota x) alpha;
  SubON_onto : forall y, ltB y alpha -> exists x:A, iota x = y}.

```

For instance, we prove that **Omega** is a sub-notation of **Omega\_plus\_Omega** (with  $\omega$  as the first “new” ordinal, and **fin** as the injection).

From Module OrdinalNotations.ON\_Omega\_plus\_omega

```
#[global] Instance Incl : SubON Omega Omega_plus_Omega omega fin.
```

We can also show that, if  $i < j$ , then the segment  $[0, i)$  is a “sub-segment” of  $[0, j)$ . Since the terms  $(t\ i)$  and  $(t\ j)$  are not convertible, we consider a “cast” function  $\iota$  from  $(t\ i)$  into  $(t\ j)$ , and prove that this function is a monotonous bijection from  $(t\ i)$  to the segment  $[0, i)$  of  $(t\ j)$ .

*From Module OrdinalNotations.ON\_Finite*

**Section** Inclusion\_ij.

```
Variables i j : nat.
```

```
Hypothesis Hij : i < j.
```

```
Remark Ltb_ij : Nat.ltb i j.
```

```
#[program] Definition iota_ij (alpha: t i) : t j := alpha.
```

```
Let b : t j := exist _ i Ltb_ij.
```

```
#[global]
```

```
Instance F_incl_ij: SubON (Fin0rd i) (Fin0rd j) b iota_ij.
```

```
Lemma iota_compare_commute alpha beta:
```

```
  compare alpha beta =
```

```
  compare (iota_ij alpha) (iota_ij beta).
```

```
Lemma iota_mono : forall alpha beta,
```

```
  lt alpha beta <->
```

```
  lt (iota_ij alpha) (iota_ij beta).
```

```
End Inclusion_ij.
```

**Exercise 3.4** Prove that  $\text{Omega\_plus\_Omega}$  cannot be a sub-notation of  $\text{Omega}$ .

**Project 3.1** Adapt the definition of  $\text{Hvariant}$  (Sect. 2.4.2) in order to have an ordinal notation as argument. Prove that if  $O_A$  is a sub-notation of  $O_B$ , then any variant defined on  $O_A$  can be automatically transformed into a variant on  $O_B$ .

## 3.11 Comparing an ordinal notation with Schütte’s model

Finally, it may be interesting to compare an ordinal notation with the more theoretical model from Schütte (well, at least with our formalization of that model). This would be a relative proof of correctness of the considered ordinal notation.

The following class specifies that a notation  $OA$  describes a segment  $[0, \alpha)$ , where  $\alpha$  is a countable ordinal *à la* Schütte.

```
Class ON_correct `(alpha : Ord)
  `(OA : @ON A ltA compareA)
  (iota : A -> Ord) :=
```

```

{ ON_correct_inj : forall a, lt (iota a) alpha;
  ON_correct_onto : forall beta, lt beta alpha ->
    exists b, iota b = beta;
  On_compare_spec : forall a b:A,
    match compareA a b with
    | Datatypes.Lt => lt (iota a) (iota b)
    | Datatypes.Eq => iota a = iota b
    | Datatypes.Gt => lt (iota b) (iota a)
    end
}.

```

For instance, the following theorem tells that `Epsilon0`, our notation system for the segment  $[0, \epsilon_0)$  is a correct implementation of the theoretically defined ordinal  $\epsilon_0$  (see chapter 8 for more details).

*From Module Schutte.Correctness\_E0*

```

#[ global ] Instance Epsilon0_correct :
  ON_correct epsilon0 Epsilon0 (fun alpha => inject (cnf alpha)).

```

**Project 3.2** Same work, but replace Schütte’s model with Gaia’s.

**Project 3.3** When you have read Chapter 8, prove that the sum of two ordinal notations `ON_plus` implements the addition of ordinals.

## 3.12 Isomorphism of ordinal notations

In some cases we want to show that two notation systems describe the same segment (for instance  $[0, 3 + \omega)$  and  $[0, \omega)$ ). For this purpose, one may prove that the two notation systems are order-isomorphic.

```

Class ON_Iso
  `(OA : @ON A ltA compareA)
  `(OB : @ON B ltB compareB)
  (f : A -> B) (g : B -> A) :=
{
  iso_compare : forall x y : A,
    compareB (f x) (f y) = compareA x y;
  iso_inv1 : forall a, g (f a) = a;
  iso_inv2 : forall b, f (g b) = b
}.

```

**Exercise 3.5** Let  $i$  be some natural number. Prove that the notation systems `Omega` and `(ON_plus (OrdFin i) Omega)` are isomorphic.

*Note:* This property reflects the equality  $i + \omega = \omega$ , that we prove also in larger notation systems, as well as in Schütte’s model. This exercise is partially solved for  $i = 3$  (in `OrdinalNotations.Example_3PlusOmega`).

**Project 3.4** This exercise is about the non-commutativity of the multiplication of ordinals, reflected in ordinal notations.

For instance, the elements of the product  $(\mathbf{ON\_mult\ \Omega\ (FinOrd\ 3)})$  are ordered as follows.

$(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), \dots, (1, 0), (1, 1), (1, 2), \dots, (2, 0), (2, 1), (2, 2), \dots$

Note that the elements of  $(\mathbf{ON\_mult\ (FinOrd\ 3)\ \Omega})$  are differently ordered (without limit ordinals):

$(0, 0), (1, 0), (2, 0), (0, 1), (1, 1), (2, 1), (0, 2), (1, 2), (2, 2), (0, 3), \dots$

Prove formally that  $(\mathbf{ON\_mult\ (FinOrd\ i)\ \Omega})$  is isomorphic to  $\Omega$  whilst  $\Omega$  is a sub-notation of  $(\mathbf{ON\_mult\ \Omega\ (FinOrd\ i)})$ , for any strictly positive  $i$ .

**Note:** Like Exercise 3.5, this project corresponds to the [in]equalities  $i + \omega = \omega < \omega + i$ , for any natural number  $i$ .

**Project 3.5** Consider two isomorphic ordinal notations  $\mathbf{OA}$  and  $\mathbf{OB}$ . Prove that, if  $\mathbf{OA}$  [resp.  $\mathbf{OB}$ ] is a correct implementation of  $\alpha$  [resp.  $\beta$ ], then  $\alpha = \beta$ .

**Project 3.6** Add to the class  $\mathbf{ON}$  the requirement that for any  $\alpha$  it is decidable whether  $\alpha$  is 0, a successor or a limit ordinal.

**Hint:** Beware of the instances associated with sum and product of notations! You may consider additional fields to make the sum and product of notations “compositional”.

**Project 3.7** Reconsider the class  $\mathbf{ON}$ , with an equivalence instead of Leibniz equality.

### 3.13 Other ordinal notations

**Project 3.8** Let  $N_A$  be a notation system for ordinals strictly less than  $\alpha$ , with the strict order  $(A, <_A)$ . Please build the notation system  $\mathbf{ON\_Expl\ } N_A$ , on the type of multisets of elements of  $A$  (or, if preferred, the type of non-increasing finite sequences on  $A$ , provided with the lexicographic ordering on lists).

For instance, let us take  $N_A = \Omega$ , and take  $\alpha = \langle 4, 4, 2, 1, 0 \rangle$ ,  $\beta = \langle 4, 3, 3, 3, 3, 2 \rangle$ , and  $\gamma = \langle 5 \rangle$ . Then  $\beta < \alpha < \gamma$ .

In contrast the list  $\langle 5, 6, 3, 3 \rangle$  is not non-increasing (*i.e.* sorted w.r.t.  $\geq$ ), so it is not to be considered.

Note that if the notation  $N_A$  implements the ordinal  $\alpha$ , the new notation  $\omega^{N_A}$  must implement the ordinal  $\phi_0(\alpha)$ , a.k.a.  $\omega^\alpha$  (see chapter 8)

**Remark 3.7** The set of ordinal terms in Cantor normal form (see Chap. 4) and in Veblen normal form (see  $\mathbf{Gamma0.Gamma0}$ ) are shown to be ordinal notation systems, but there is a lot of work to be done in order to unify ad-hoc definitions and proofs which were written before the definition of the  $\mathbf{ON}$  type class.

In Section 4.3 on page 91, we present a notation system for the ordinal  $\omega^\omega$  as a *refinement* of the ordinal notation for  $\epsilon_0$ .



# Chapter 4

## The Ordinal $\epsilon_0$

In this chapter, we adapt to Coq the well-known proof [KP82] that Hercules eventually wins every battle, whichever the strategy of each player. In other words, we present a formal and self-contained proof of termination of all [free] hydra battles. First, we take from Manolios and Vroon [MV05] a representation of the ordinal  $\epsilon_0$  as terms in Cantor normal form. Then, we define a variant for hydra battles as a measure that maps any hydra to some ordinal strictly less than  $\epsilon_0$ .

### 4.1 The ordinal $\epsilon_0$

The ordinal  $\epsilon_0$  is the least ordinal number that satisfies the equation  $\alpha = \omega^\alpha$ , where  $\omega$  is the least infinite ordinal<sup>1</sup>. Thus, we can intuitively consider  $\epsilon_0$  as an *infinite*  $\omega$ -tower.

#### 4.1.1 Cantor normal form

Any ordinal strictly less than  $\epsilon_0$  can be finitely represented by a unique *Cantor normal form*, that is, an expression which is a sum  $\omega^{\alpha_1} \times n_1 + \omega^{\alpha_2} \times n_2 + \dots + \omega^{\alpha_p} \times n_p$  where  $p \in \mathbb{N}$ , all the  $\alpha_i$  are ordinals in Cantor normal form,  $\alpha_1 > \alpha_2 > \dots > \alpha_p$  and all the  $n_i$  are positive integers.

An example of Cantor normal form is displayed in Fig 4.1: Note that any ordinal of the form  $\omega^0 \times i + 0$  is just written  $i$ .

$$\omega(\omega^\omega + \omega^2 \times 8 + \omega) + \omega^\omega + \omega^4 + 6$$

Figure 4.1: An ordinal in Cantor normal form

In the rest of this section, we define an inductive type for representing in Coq all the ordinals strictly less than  $\epsilon_0$ , then extend some arithmetic operations to this type, and finally prove that our representation fits well with the expected

---

<sup>1</sup>For a precise — *i.e.* mathematical — definition of  $\omega^\alpha$ , please see Sect. 8.6 on page 174.

mathematical properties: the order we define is a well order, and the decomposition into Cantor normal form is consistent with usual definition of ordinals, for instance in Gaia [GQS], Schütte’s book [Sch77], or larger ordinal notations 9 on page 185.

**Remark** Unless explicitly mentioned, the term “ordinal” will be used instead of “ordinal strictly less than  $\epsilon_0$ ” (except in Chapter 8 where it stands for “countable ordinal”).

### 4.1.2 A data type for ordinals in Cantor normal form

Let us define an inductive type whose constructors are respectively associated with the ways to build Cantor normal forms:

- the ordinal 0
- the construction  $(\alpha, n, \beta) \mapsto \omega^\alpha \times (n + 1) + \beta$  ( $n \in \mathbb{N}$ )

From Module *Epsilon0.T1*

```
Inductive T1 : Set :=
| zero
| cons (alpha : T1) (n : nat) (beta : T1) .
```

#### 4.1.2.1 Example

For instance, the ordinal  $\omega^\omega + \omega^3 \times 5 + 2$  is represented by the following term of type T1:

```
Example alpha_0 : T1 :=
  cons (cons (cons zero 0 zero)
            0
        zero)
    0
  (cons (cons zero 2 zero)
        4
    (cons zero 1 zero)).
```

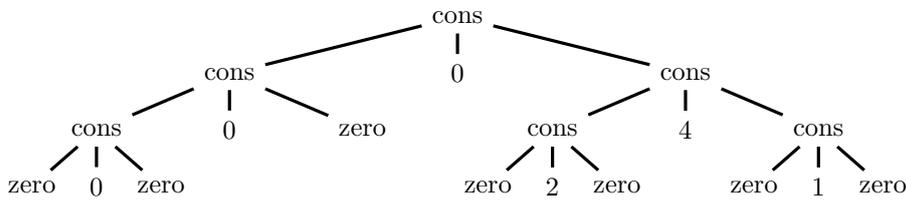


Figure 4.2: The tree-like representation of the ordinal  $\omega^\omega + \omega^3 \times 5 + 2$

**4.1.2.1.1 Remark** For simplicity's sake, we chose to forbid expressions of the form  $\omega^\alpha \times 0 + \beta$ . Thus, the construction `(cons  $\alpha$   $n$   $\beta$ )` is intended to represent the ordinal  $\omega^\alpha \times (n + 1) + \beta$  and not  $\omega^\alpha \times n + \beta$ . In a future version, we would like to replace the type `nat` with standard library's type `positive` in `T1`'s definition. But this replacement would take a lot of time ...

**4.1.2.1.2 Remark** The name `T1` we gave to this data-type is proper to this development and refers to a hierarchy of ordinal notations. For instance, in Library `Gamma0.T2`, the following type is used to represent ordinals strictly less than  $\Gamma_0$ , in Veblen normal form (see also [GQS, Sch77]).

```
Declare Scope T2_scope.
Delimit Scope T2_scope with t2.
```

```
Open Scope T2_scope.
```

```
Inductive T2 : Set :=
| zero : T2
| gcons : T2 -> T2 -> nat -> T2 -> T2.
```

### 4.1.3 About Gaia

 Chapter 7 on page 141 describes the present state of a project of making compatible the libraries `Hydra-battles` and `Gaia`. At present, both libraries contain their own version of the inductive type `T1`, with the same base name for the constructors.

```
Inductive T1 : Set :=
  zero : T1
| cons : T1 -> nat -> T1 -> T1.
```

Thus, many examples of this and following chapters can be run in `Gaia`'s context. We will signal any difference (notation, name) which may appear. A closer integration (same data type and functions) is still in project.

**Remark 4.1** In future releases, we plan to make some `Hydra-battles` identifiers progressively deprecated, in favour of `Gaia`'s names.

### 4.1.4 Abbreviations

Some abbreviations may help to write more concisely complex ordinal terms.

#### 4.1.4.1 Finite ordinals

For representing finite ordinals, *i.e.* natural numbers, we first introduce a notation for terms of the form  $n + 1$ , then define a coercion from type `nat` into `T1`.

```
(** The [(S n)]-th ordinal *)
Notation FS n := (cons zero n zero).
```

**Notation** `one` := (FS 0).

(\*\* the [n]-th (finite) ordinal \*\*)

**Definition** `Tlnat` (n:nat) := match n with 0 => zero | S p => FS p end.

**Notation** `"\F n"` := (Tlnat n) (at level 29): t1\_scope.

**Coercion** `Tlnat` : nat >-> T1.

**Example** `ten` : T1 := 10.

**G** In `gaia.ssete9`, the  $n$ -th finite ordinal is also written `\F n`. There is no coercion in Gaia from `nat` to `T1`.

From `gaia_hydras.HydraGaia_Examples`.

**Check** `\F 42`.

```
\F 42
  : T1
```

**Fail Check** `(42 : T1)`.

```
The command has indeed failed with message:
The term "42" has type "nat"
while it is expected to have type "T1".
```

#### 4.1.4.2 The ordinal $\omega$

Since  $\omega$ 's Cantor normal form is i.e.  $\omega^{\omega^0} \times 1 + 0$ , we can define the following abbreviation:

**Notation** `Tlomega` := (cons (cons zero 0 zero) 0 zero).

Note that `Tlomega` is not an identifier in Hydra-battles, thus any tactic like `unfold Tlomega` would fail.

**G** In `gaia.ssete9`, the ordinal  $\omega$  is bound to the *constant* `Tlomega` (not a notation).

#### 4.1.4.3 The ordinal $\omega^\alpha$ , a.k.a. $\phi_0(\alpha)$

We provide also a notation for ordinals of the form  $\omega^\alpha$ .

**Notation** `phi0 alpha` := (cons alpha 0 zero).

**Remark 4.2** The name  $\phi_0$  comes from ordinal numbers theory. In [Sch77], Schütte defines  $\phi_0$  as the ordering (i.e. enumerating) function of the set of *additive principal ordinals* i.e. strictly positive ordinals  $\alpha$  that verify  $\forall \beta < \alpha, \beta + \alpha = \alpha$ . For Schütte,  $\omega^\alpha$  is just a notation for  $\phi_0(\alpha)$ . See also Chapter 8 of this document.

**G** In `gaia.ssete9` the identifier `phi0` is bound to a plain constant (not a notation).

#### 4.1.4.4 The hierarchy of $\omega$ -towers:

The ordinal  $\epsilon_0$ , although not represented by a finite term in Cantor normal form, is approximated by the sequence of  $\omega$ -towers (see also Sect 8.6.3 on page 175 ).

*From Module `Epsilon0.T1`*

```
Fixpoint omega_tower (height:nat) : T1 :=
  match height with
  | 0 => one
  | S h => phi0 (omega_tower h)
  end.
```

**Compute** omega\_tower 7.

```
= phi0 (phi0 (phi0 (phi0 (phi0 (phi0 T1omega))))))
: T1
```

For instance, Figure 4.3 represents the ordinal returned by the evaluation of the term `omega_tower 7`.

Figure 4.3: The  $\omega$ -tower of height 7

#### 4.1.5 Pretty-printing ordinals in Cantor normal form

Let us consider again the ordinal  $\alpha_0$  defined in section 4.1.2.1 on page 74. If we ask Coq to print its normal form, we get a hardly readable term of type `T1`.

**Compute** alpha\_0.

```
= cons T1omega 0 (cons (FS 2) 4 (FS 1))
: T1
```

The following data type defines a more readable abstract syntax for ordinals terms in Cantor normal form:

```
Declare Scope ppT1_scope.
Delimit Scope ppT1_scope with pT1.
```

```
Inductive ppT1 :=
  | PP_fin (_ : nat)
  | PP_add (_ _ : ppT1)
  | PP_mult (_ : ppT1) (_ : nat)
```

```
| PP_exp ( _ _ : ppT1)
| PP_omega.
```

**Coercion** `PP_fin` : `nat`  $\rightarrow$  `ppT1`.

**Notation** `"alpha + beta"` := `(PP_add alpha beta)` : `ppT1_scope`.

**Notation** `"alpha * n"` := `(PP_mult alpha n)` : `ppT1_scope`.

**Notation** `"alpha ^ beta"` := `(PP_exp alpha beta)` : `ppT1_scope`.

**Notation** `ω` := `PP_omega`.

**Check** `(ω ^ ω * 2 + 1)%ppT1`.

```
(ω ^ ω * 2 + 1)%ppT1
  : ppT1
```

The function `pp: T1 -> ppT1` converts any closed term of type `T1` into a human-readable expression. For instance, let us convert the term `alpha_0`.

**Compute** `pp alpha_0`.

```
= (ω ^ ω + ω ^ 3 * 5 + 2)%ppT1
  : ppT1
```

 In `gaia.T1Bridge`, we define a variant of `pp` for pretty-printing terms of type `ssete9.CantorOrdinal.T1` (see Sect. 7.2.2.1 on page 144).

**Project 4.1** Design tools for systematically pretty printing ordinal terms in Cantor normal form.

## 4.1.6 Comparison between ordinal terms

In order to compare two terms of type `T1`, we define a recursive function `compare` that maps two ordinal terms  $\alpha$  and  $\beta$  to a value of type `comparison`. This type is defined in Coq's standard library `Init.Datatypes` and contains three constructors: `Lt` (less than), `Eq` (equal), and `Gt` (greater than).

*From Module `Epsilon0.T1`*

```
#[global] Instance compare_T1 : Compare T1 :=
  fix cmp (alpha beta:T1) :=
    match alpha, beta with
    | zero, zero => Eq
    | zero, cons a' n' b' => Lt
    | _ , zero => Gt
    | (cons a n b),(cons a' n' b') =>
      (match cmp a a' with
      | Lt => Lt
```

```

    | Gt => Gt
    | Eq => (match n ?= n' with
            | Eq => cmp b b'
            | comp => comp
            end)
    end)
end.

```

**Definition** `lt` (`alpha beta` : T1) : Prop :=  
`compare alpha beta = Lt.`

**Notation** `le` := (leq lt).

**[global] Instance** `t1_strorder`: StrictOrder lt.

**[global] Instance**: Comparable lt `compare`.

Please note that this definition of `lt` makes it easy to write proofs by computation, as shown by the following examples.

**Example E1** : `lt (cons Tlomega 56 zero) (omega_tower 3).`

**Proof.** `reflexivity. Qed.`

**Example E2** : `~ lt (omega_tower 3) (cons Tlomega 5 (omega_tower 3))%t1.`

**Proof.** `discriminate. Qed.`

 In Gaia, the strict order `T1lt` on ordinal terms is directly defined as a boolean function (in `SSReflect`'s style).

```

Fixpoint T1lt x y {struct x} :=
  if x is cons a n b then
    if y is cons a' n' b' then
      if a < a' then true
      else if a == a' then
        if (n < n')%N then true
        else if (n == n') then b < b' else false
        else false
      else false
    else if y is cons a' n' b' then true else false
where "x < y" := (T1lt x y) : cantor_scope.

```

**Definition** `T1le` (x y :T1) := (x == y) || (x < y).

**Notation** "`x <= y`" := (T1le x y) : cantor\_scope.

**Notation** "`x >= y`" := (y <= x) (**only parsing**) : cantor\_scope.

**Notation** "`x > y`" := (y < x) (**only parsing**) : cantor\_scope.

In section 7.2.3.3 on page 145, we show that both definitions are mutually equivalent.

### 4.1.6.1 A Predicate for Characterizing Normal Forms

Our data-type `T1` allows us to write expressions that are not properly in Cantor normal form as specified in Section 4.1. For instance, consider the following term of type `T1`.

**Example** `bad_term`: `T1 := cons 1 1 (cons T1omega 2 zero)`.

This term would have been written  $\omega^1 \times 2 + \omega^\omega \times 3$  in the usual mathematical notation. We note that the exponents of  $\omega$  are not in the right (strictly decreasing) order. The following boolean function determines whether a given ordinal term is well formed.

*From Module Epsilon0.T1*

```
Fixpoint nf_b (alpha : T1) : bool :=
  match alpha with
  | zero => true
  | cons a n zero => nf_b a
  | cons a n ((cons a' n' b') as b) =>
    (nf_b a && nf_b b && (bool_decide (lt a' a)))%bool
  end.
```

**Definition** `nf alpha : Prop := nf_b alpha`.

**Compute** `nf_b alpha_0`.

```
= true
: bool
```

**Compute** `nf_b bad_term`.

```
= false
: bool
```

 In Gaia, the boolean function which characterizes ordinal terms in normal form is defined as follows:

```
Fixpoint T1nf x :=
  if x is cons a _ b then [ && T1nf a, T1nf b & b < phi0 a ]
  else true.
```

In Sect. 7.2.3.6 on page 146, we show that Gaia's `T1nf` is extensionally equivalent with Hydra-battles' `nf`.

## 4.1.7 Making normality implicit

We would like to get rid of terms of type `T1` which are not in Cantor normal form. A simple way to do this is to consider statements of the form `forall alpha: T1, nf alpha -> P alpha`, where  $P$  is a predicate over type `T1`, like in the following lemma <sup>2</sup>.

<sup>2</sup>Ordinal addition is formally defined a little later (page 4.1.9.3)

```

Lemma plus_is_zero alpha beta :
  nf alpha -> nf beta ->
  alpha + beta = zero -> alpha = zero /\ beta = zero.

```

But this style leads to clumsy statements, and generates too many subgoals in interactive proofs (although often solved with `auto` or `eauto`).

One may encapsulate conditions of the form  $(nf \alpha)$  in the most used predicates. For instance, we introduce the restriction of `lt` to terms in normal form, and provide a handy notation for this restriction.

*From Module `hydras.Prelude.Restriction`*

```

Definition restrict {A:Type}(E: Ensemble A)(R: relation A) :=
  fun a b => E a /\ R a b /\ E b.

```

*From Module `Epsilon0.T1`*

```

Definition LT := restrict nf lt.
Infix "t1<" := LT : t1_scope.

```

```

Definition LE := restrict nf (leq lt).
Infix "t1<=" := LE : t1_scope.

```

For instance, in the following lemma, the condition that  $\alpha$  is in normal form is included in the condition  $\alpha < 1$ .

```

Lemma LT_one alpha :
  alpha t1< one -> alpha = zero.

```

**Proof.**

```

  intros [H1 [H2 _]]; destruct alpha; auto.

```

```

alpha1: T1
n: nat
alpha2: T1
H1: nf (cons alpha1 n alpha2)
H2: lt (cons alpha1 n alpha2) one
-----
cons alpha1 n alpha2 = zero

```

(\* ... \*)

**Qed.**

#### 4.1.7.1 $\mathbf{E0}$ : a sigma-type for $\epsilon_0$

As we noticed in Sect. 4.1.6.1, the type `T1` is not a correct ordinal notation, since it contains terms that are not in Cantor normal form. In certain contexts (for instance in Sections 6.2.4, 6.3, and 6.4), we need to define total recursive functions on well-formed ordinal terms less than  $\epsilon_0$ , using the `Equations` plugin [SM19]. In order to define a type whose inhabitants represent just ordinals, we build a type gathering a term of type `T1` and a proof that this term is in normal form.

*From Module `Epsilon0.E0`*

```
Class E0 : Type := mkord {cnf : T1; cnf_ok : nf cnf}.
```

```
Arguments cnf : clear implicits.
```

```
#[export] Hint Resolve cnf_ok : E0.
```

Many constructs : types, predicates, functions, notations, etc., on type T1 are adapted to E0.

First, we declare a notation scope for E0, then we redefine the predicates of comparison.

```
Declare Scope E0_scope.
```

```
Delimit Scope E0_scope with e0.
```

```
Open Scope E0_scope.
```

```
Definition E0lt (alpha beta : E0) := T1.LT (@cnf alpha) (@cnf beta).
```

```
Definition E0le := leq E0lt.
```

```
Infix "<" := E0lt : E0_scope.
```

```
Infix "<=" := E0le : E0_scope.
```

Equality in E0 is just Leibniz equality. Note that, since nf is defined by a Boolean function, for any term  $\alpha : T1$ , there exists at most one proof of  $\text{nf } \alpha$ , thus two ordinals of type E0 are equal if and only if their projection to T1 are equal (see also Sect. 3.3 on page 66).

```
Lemma nf_proof_unicity :
```

```
  forall (alpha:T1) (H H' : nf alpha), H = H'.
```

```
Proof.
```

```
  intros; red in H, H'; apply eq_proofs_unicity_on; decide equality.
```

```
Qed.
```

```
Corollary E0_eq_iff (alpha beta : E0) :
```

```
  alpha = beta <-> cnf alpha = cnf beta.
```

In order to upgrade constants and functions from type T1 to E0, we have to prove that the term they build is in normal form. For instance, let us represent the ordinals 0 and  $\omega$  as instances of the class E0.

```
#[export] Instance E0zero : E0 := @mkord zero refl_equal.
```

```
#[export] Instance E0_omega : E0 := @mkord T1omega refl_equal.
```

 Our library `gaia_hydras.T1Bridge` also defines a type E0 (which doesn't exist in `Gaia-ssete9`).

### 4.1.8 Syntactic definition of limit and successor ordinals

Pattern matching and structural recursion allow us to define boolean characterizations of successor and limit ordinals.

From Module *Epsilon0.T1*

```
Fixpoint Tlis_succ alpha :=
  match alpha with
  | zero => false
  | cons zero _ _ => true
  | cons _alpha _n beta => Tlis_succ beta
  end.
```

```
Fixpoint Tllimit alpha :=
  match alpha with
  | zero => false
  | cons zero _ _ => false
  | cons _ _ zero => true
  | cons _ _ beta => Tllimit beta
  end.
```

**Compute** Tllimit Tlomega.

```
= true
: bool
```

**Compute** Tllimit 42.

```
= false
: bool
```

**Compute** Tlis\_succ 42.

```
= true
: bool
```

**Compute** Tlis\_succ Tlomega.

```
= false
: bool
```

The correctness of these definitions with respect to the mathematical notions of limit and successor ordinals is established through several lemmas. For instance, Lemma `canonS_limit_lub`, page 105, shows that if  $\alpha$  is (syntactically) a limit ordinal, then it is the least upper bound of a strictly increasing sequence of ordinals.

 In Gaia, the boolean functions associated with limit and successor ordinals are also called `Tllimit` and `Tlis_succ`.

From `gaia_hydras.HydraGaia_Examples`.

```

Compute (Tlimit Tlomega, Tlis_succ (omega_tower 2), Tlis_succ (\F 42)).
= (true, false, true)
: bool * bool * bool

```

## 4.1.9 Arithmetic on $\epsilon_0$

### 4.1.9.1 Successor

The successor of any ordinal  $\alpha < \epsilon_0$  is defined by structural recursion on its Cantor normal form.

*From Module Epsilon0.T1*

```

Fixpoint succ (a: T1) : T1 :=
  match a with
  | zero => Tlnat 1
  | cons zero n _ => cons zero (S n) zero
  | cons b n c => cons b n (succ c)
  end.

```

The following lemma establishes the connection between the function `succ` and the Boolean predicate `Tlis_succ`.

```

Lemma Tlis_succ_iff alpha (Halpha : nf alpha) :
  Tlis_succ alpha <-> exists beta : T1, nf beta /\ alpha = succ beta.

```

**Exercise 4.1**  Look for the Gaia-theorem which corresponds to `Tlis_succ_iff`.

### 4.1.9.2 Successor function on $E0$

The function `succ` on `T1` is extended to `E0` the following way:

*From Module Epsilon0.E0*

```

#[global, program] Instance E0_succ (alpha : E0) : E0 :=
@mkord (T1.succ (@cnf alpha)) _.
Next Obligation. apply succ_nf, cnf_ok. Defined.

```

**Exercise 4.2** Prove in Coq that for any ordinal  $0 < \alpha < \epsilon_0$ ,  $\alpha$  is a limit if and only if for all  $\beta < \alpha$ , the interval  $[\beta, \alpha)$  is infinite.

*You may start this exercise with the file `exercises/ordinals/Limit_Infinity.v`.*

### 4.1.9.3 Addition and multiplication

Ordinal addition and multiplication are also defined by structural recursion over the type `T1`. Please note that they use the `compare` function on some subterms of their arguments.

```

Fixpoint Tladd (a b : T1) :T1 :=
  match a, b with
  | zero, y => y
  | x, zero => x

```

```

| cons a n b, cons a' n' b' =>
  (match compare a a' with
  | Lt => cons a' n' b'
  | Gt => (cons a n (Tladd b (cons a' n' b')))
  | Eq => (cons a (S (n+n')) b'))
  end)
end
where "alpha + beta" := (Tladd alpha beta) : t1_scope.

Fixpoint T1mul (a b : T1) :T1 :=
  match a, b with
  | zero, _ => zero
  | _, zero => zero
  | cons zero n _, cons zero n' b' =>
    cons zero (Peano.pred((S n) * (S n'))) b'
  | cons a n b, cons zero n' _ =>
    cons a (Peano.pred ((S n) * (S n'))) b
  | cons a n b, cons a' n' b' =>
    cons (a + a') n' ((cons a n b) * b')
  end
where "a * b" := (T1mul a b) : t1_scope.

```

 We keep Gaia's base names for addition and multiplication of ordinal terms below  $\epsilon_0$ . Please refer to Sect. 7.2.3.5 about compatibility of both arithmetics.

#### 4.1.9.4 Examples

The following examples are instances of *proofs by computation*. Please note that addition and multiplication on T1 are not commutative. Moreover, both operations fail to be strictly monotonous in their first argument.

**Example Ex1** :  $42 + \text{Tlomega} = \text{Tlomega}$ .

**Proof.** reflexivity. Qed.

**Example Ex2** :  $\text{Tlomega} \text{ t1} < \text{Tlomega} + 42$ .

**Proof.** now compute. Qed.

**Example Ex3** :  $5 * \text{Tlomega} = \text{Tlomega}$ .

**Proof.** reflexivity. Qed.

**Example Ex4** :  $\text{Tlomega} \text{ t1} < \text{Tlomega} * 5$ .

**Proof.** now compute. Qed.

**Lemma Tladd\_not\_monotonous\_l** :

exists a b c : T1, a t1 < b /\ a + c = b + c.

**Proof.** exists 3, 5, Tlomega; now compute. Qed.

**Lemma T1mul\_not\_monotonous** :

exists a b c : T1, c <> zero /\ a t1 < b /\ a \* c = b \* c.

**Proof.** exists 3, 5, Tlomega; split; [discriminate] now compute]. Qed.

The function `succ` is related with addition through the following lemma:

**Lemma `succ_is_plus_one`** (`a : T1`) : `succ a = a + 1`.

**Proof.**

```
induction a as [ |a IHa n b IHb]; [trivial |].
(* ... *)
```

**Qed.**

#### 4.1.9.5 Arithmetic on type `E0`

We define an addition in type `E0`, since the sum of two terms in normal form is in normal form too.

**Lemma `plus_nf`:**

```
forall a, nf a -> forall b, nf b -> nf (a + b).
```

**#[global, program] Instance `E0add`** (`alpha beta : E0`) : `E0 :=`

```
@mkord (T1add (@cnf alpha) (@cnf beta))_ .
```

**Next Obligation.** `apply plus_nf; apply cnf_ok.` **Defined.**

**Infix `"+"`** := `E0add` : `E0_scope`.

**Check** `E0_omega + E0_omega`.

```
E0_omega + E0_omega
: E0
```

**Remark 4.3** In all this development, two representations of ordinals co-exist: ordinal terms (type `T1`, notation scope `t1_scope`, for reasoning on the tree-structure of Cantor normal forms), and ordinal terms *known to be in normal form* (type `E0`, notation scope `E0_scope`). Looking at the contexts displayed by Coq prevents you from any risk of confusion.

**Exercise 4.3** Prove that for any ordinal  $\alpha : E0$ ,  $\omega \leq \alpha$  if and only if, for any natural number  $i$ ,  $i + \alpha = \alpha$ .

*You may start this exercise with the file `exercises/ordinals/ge_omega_iff.v`.*

#### 4.1.10 A proof by computation

It is interesting to compare the following proof of the equality  $\omega + 42 + \omega^2$  with the more theoretical proof in Sect 8.6.5 on page 177.

**Example Ex42:** `E0_omega + 42 + E0_omega^2 = E0_omega^2`.

**Proof.**

```
rewrite <- Comparable.compare_eq_iff.
```

```
compare (E0_omega + 42 + E0_phi0 2) (E0_phi0 2) = Eq
```

```
reflexivity.
```

**Qed.**

## 4.2 Well-foundedness and transfinite induction

### 4.2.1 About well-foundedness

In order to use `T1` for proving termination results, we need to prove that our order `<` is well-founded. Then we will get *transfinite induction* for free.

The proof of well-foundedness of the strict order `<` on Cantor normal forms is already available in the Cantor contribution by Castéran and Contejean [CC06]. That proof relies on a library on recursive path orderings written by E. Contejean. We present here a direct proof of the same result, which does not require any knowledge on r.p.o.s.

**Exercise 4.4** Prove that the *total* order `lt` on `T1` is not well-founded. **Hint:** You will have to build a counter-example with terms of type `T1` which are not in Cantor normal form.

*You may start this exercise with the file `exercises/ordinals/T1_ltNotWf.v`.*

#### 4.2.1.1 A first attempt

It is natural to try to prove by structural induction over `T1` that every term in normal form is `LT`-accessible.

Unfortunately, it won't work. Let us consider some well-formed term  $\alpha = \text{cons } \beta \ n \ \gamma$ , and assume that  $\beta$  and  $\gamma$  are `LT`-accessible. In order to prove the accessibility of  $\alpha$ , we have to consider any well formed term  $\delta$  such that  $\delta < \alpha$ .

**Section** `First_attempt`.

**Lemma** `wf_LT` : forall alpha: T1, nf alpha -> Acc LT alpha.

**Proof.**

```
induction alpha as [| beta IHbeta n gamma IHgamma].
- split. intros y H0; inversion H0 as [_ [H3 _]];
  destruct (not_lt_zero H3).
- split; intros delta Hdelta.
```

```
IHbeta: nf beta -> Acc LT beta
IHgamma: nf gamma -> Acc LT gamma
delta: T1
Hdelta: delta t1< cons beta n gamma
-----
Acc LT delta
```

The problem comes from the too weak hypothesis `Hdelta`. It does not prevent  $\delta$  to be bigger than  $\beta$  or  $\gamma$ ; for instance  $\delta$  may be of the form `cons beta p' gamma'`, where  $p' < n$ . Thus, the induction hypotheses `IHbeta` and `IHgamma` are useless for finishing our proof.

**Abort.**

**End** `First_attempt`.

#### 4.2.1.2 Using a stronger inductive predicate.

Instead of trying to prove directly that any ordinal term  $\alpha$  in normal form is accessible through `LT`, we propose to consider the following (stronger) predicate:

```

Let Acc_strong (alpha:T1) :=
  forall n beta,
    nf (cons alpha n beta) -> Acc LT (cons alpha n beta).

```

The following lemma is an application of the strict inequality  $\alpha < \omega^\alpha$ . If  $\alpha$  is strongly accessible, then, by definition,  $\omega^\alpha$  is accessible, thus  $\alpha$  is *a fortiori* accessible.

```

Lemma Acc_strong_stronger : forall alpha,
  nf alpha -> Acc_strong alpha -> Acc LT alpha.

```

**Proof.**

```

intros alpha H H0; apply acc_impl with (phi0 alpha).
- repeat split; trivial.
+ now apply lt_a_phi0_a.
- apply H0; now apply single_nf.

```

**Qed.**

Thus, it remains to prove that every ordinal strictly less than  $\epsilon_0$  is strongly accessible.

**4.2.1.2.1 A helper** First, we prove that, for any LT-accessible term  $\alpha$ ,  $\alpha$  is strongly accessible too. The following proof is structured as an induction on  $\alpha$ 's accessibility. Let us consider any accessible term  $\alpha$ .

```

Lemma Acc_implies_Acc_strong : forall alpha,
  Acc LT alpha -> Acc_strong alpha.

```

**Proof.**

```

(* main induction (on alpha's accessibility) *)
unfold Acc_strong; intros alpha Aalpha.

```

```

alpha: T1
Aalpha: Acc LT alpha
-----
forall (n : nat) (beta : T1),
nf (cons alpha n beta) -> Acc LT (cons alpha n beta)

pattern alpha;
eapply Acc_ind with (R:= LT); [| assumption];
clear alpha Aalpha; intros alpha Aalpha IAlpha.

```

```

alpha: T1
Aalpha: forall y : T1, y t1< alpha -> Acc LT y
IAlpha: forall y : T1,
  y t1< alpha ->
  forall (n : nat) (beta : T1),
  nf (cons y n beta) -> Acc LT (cons y n beta)
-----
forall (n : nat) (beta : T1),
nf (cons alpha n beta) -> Acc LT (cons alpha n beta)

```

First, we prove that, for any  $n$  and  $\beta$ , if  $(\text{cons } \alpha \ n \ \beta)$  is in normal form, then  $\beta$  is accessible.

```

assert(beta_Acc:
  forall beta, lt beta (phi0 alpha) -> nf alpha -> nf beta
  -> Acc LT beta).
(* ... *)

```

The new hypothesis `beta_Acc` allows us to prove by well-founded induction on  $\beta$ , and natural induction on  $n$  that  $(\text{cons } \alpha \ n \ \beta)$  is accessible.

```

alpha: T1
Aalpha: forall y : T1, y t1< alpha -> Acc LT y
IHalpha: forall y : T1,
  y t1< alpha ->
  forall (n : nat) (beta : T1),
  nf (cons y n beta) -> Acc LT (cons y n beta)
beta_Acc: forall beta : T1,
  lt beta (phi0 alpha) ->
  nf alpha -> nf beta -> Acc LT beta
-----
forall (n : nat) (beta : T1),
nf (cons alpha n beta) -> Acc LT (cons alpha n beta)

```

The proof, quite long, can be consulted in `Epsilon0.T1`.

**4.2.1.2.2 Accessibility of any well-formed ordinal term** Our goal is still to prove accessibility of any well formed ordinal term. Thanks to our previous lemmas, we are almost done (by a simple structural induction!).

**Theorem** `nf_Acc (alpha : T1): nf alpha -> Acc LT alpha.`

**Proof.**

```

induction alpha.
- intro; apply Acc_zero.
- intros; eapply Acc_implies_Acc_strong; auto.
  apply IHalpha1; eauto.

```

```

alpha1: T1
n: nat
alpha2: T1
IHalpha1: nf alpha1 -> Acc LT alpha1
IHalpha2: nf alpha2 -> Acc LT alpha2
H: nf (cons alpha1 n alpha2)
-----
nf alpha1

```

```

  apply nf_inv1 in H; auto.

```

**Qed.**

**Corollary** `T1_wf : well_founded LT.`

## 4.2.2 Transfinite induction

Traditionnally, well-founded induction on ordinals is called *transfinite induction*.

**Definition** `transfinite_recurzor` := `well_founded_induction_type T1_wf`.

**Check** `transfinite_recurzor`.

```
transfinite_recurzor
  : forall P : T1 -> Type,
    (forall x : T1,
     (forall y : T1, y t1< x -> P y) -> P x) ->
    forall a : T1, P a
```

**Ltac** `transfinite_induction` `alpha` :=  
`pattern alpha; apply transfinite_recurzor`.

As a corollary, the order `Lt` on type `E0` is well-founded too.

**Lemma** `E0lt_wf` : `well_founded E0lt`.

**Proof.**

```
split; intros [t Ht] H;
  apply (Acc_inverse_image _ _ LT (@cnf)
        {| cnf := t; cnf_ok := Ht |});
  now apply T1.nf_Acc.
```

**Defined.**

**Remark 4.4 (Related work)** A proof of well-foundedness using Évelyne Contejean’s work on recursive path ordering [Der82, CPU<sup>+</sup>10] is also available in the library `Epsilon0.Epsilon0rpo`.

In [MV05], Manolios and Vroom prove the well-foundedness of ordinal terms below  $\epsilon_0$  by reduction to the natural order on the set of natural numbers.

### 4.2.3 An ordinal notation for $\epsilon_0$

We are now able to build an instance of `ON`.

*From Module* `Epsilon0.E0`

```
#[global] Instance E0_comp: Comparable E0lt compare.
Proof. split; [apply E0_sto | apply compare_correct]. Qed.

#[global] Instance Epsilon0 : ON E0lt compare.
Proof. split; [apply E0_comp | apply E0lt_wf]. Qed.
```

We prove also that this notation is correct w.r.t. Schutte’s model (see Chapter 8).

*From Module* `Schutte.Correctness_E0`

```
Fixpoint inject (t:T1) : Ord :=
  match t with
  | T1.zero => zero
  | T1.cons a n b => AP._phi0 (inject a) * S n + inject b
  end.

#[ global ] Instance Epsilon0_correct :
  ON_correct epsilon0 Epsilon0 (fun alpha => inject (cnf alpha)).
```

### 4.2.4 An ordinal notation for Gaia's ordinals

**G** Module `gaia.T1Bridge` contains an instance of class `ON E0lt compare`, where `E0lt` is the order on the well-formed ordinal terms below  $\epsilon_0$  defined in `Gaia-hydras` library (please see Chapter 7).

```
#[global] Instance Epsilon0 : ON E0lt compare.
Proof. split; [apply: E0_comp | apply: gE0lt_wf]. Qed.
```

**Project 4.2** *This exercise is a continuation of Project 3.12 on page 70.* Use `ON_mult` to define an ordinal notation `0omega2` for  $\omega^2 = \omega \times \omega$ .

Prove that `0omega2` is a sub-notation of `Epsilon0`.

Define on `0omega2` an addition compatible with the addition on `Epsilon0`.

**Hint.** You may use the following definition (in `OrdinalNotations.ON_Generic`).

```
Definition SubON_same_op `{OA : @ON A ltA compareA}
  `{OB : @ON B ltB compareB}
  {iota : A -> B} {alpha: B}
  {_ : SubON OA OB alpha iota}
  (f : A -> A -> A)
  (g : B -> B -> B)
:= forall x y, iota (f x y) = g (iota x) (iota y).
```

**Project 4.3** The class `ON` of ordinal notations has been defined long after this chapter, and is not used in the development of the type `E0` yet. A better integration of both notions should simplify the development on ordinals in Cantor normal form. This integration is planned for the future versions.

## 4.3 An ordinal notation for $\omega^\omega$

In Module `theories/ordinals/OrdinalNotations/OmegaOmega.v`, we represent ordinals below  $\omega^\omega$  by lists of pairs of natural numbers (with the same coefficient shift as in `T1`). For instance, the ordinal  $\omega^4 \times 10 + \omega^3 + \omega + 5$  is represented by the list `(4,9)::(3,0)::(1,0)::(0,4)::nil`.

**Module L0.**

```
Definition t := list (nat*nat).
```

```
Definition zero : t := nil.
```

```
(** omega^ i * S n + alpha *)
```

```
Notation cons i n alpha := ((i,n)::alpha).
```

```
(** Finite ordinals *)
```

```
Notation FS n := (cons 0 n zero: t).
```

**Definition** `fin` (`n:nat`): `t := match n with 0 => zero | S p => FS p end.`

**Coercion** `fin` : `nat -> t.`

`(** [omega ^i ] *)`

**Notation** `phi0` `i := (cons i 0 nil).`

**Notation** `omega := (phi0 1).`

The usual operations `: succ`, `+`, `*` are simple variants of the same operations in T1.

**Fixpoint** `succ` (`a : t`) : `t :=`  
`match a with`  
`| nil => fin 1`  
`| cons 0 n _ => cons 0 (S n) nil`  
`| cons a n b => cons a n (succ b)`  
`end.`

**Fixpoint** `plus` (`a b : t`) : `t :=`  
`match a, b with`  
`| nil, y => y`  
`| x, nil => x`  
`| cons a n b, cons a' n' b' =>`  
`(match Nat.compare a a' with`  
`| Datatypes.Lt => cons a' n' b'`  
`| Gt => (cons a n (plus b (cons a' n' b')))`  
`| Eq => (cons a (S (n+n')) b')`  
`end)`  
`end`

**where** `"a + b" := (plus a b) : lo_scope.`

**Fixpoint** `mult` (`a b : t`) : `t :=`  
`match a, b with`  
`| nil, _ => zero`  
`| _, nil => zero`  
`| cons 0 n _, cons 0 n' b' =>`  
`cons 0 (Peano.pred((S n) * (S n'))) b'`  
`| cons a n b, cons 0 n' _ =>`  
`cons a (Peano.pred ((S n) * (S n'))) b`  
`| cons a n b, cons a' n' b' =>`  
`cons (a + a')%nat n' ((cons a n b) * b')`  
`end`

**where** `"a * b" := (mult a b) : lo_scope.`

**Compute** `omega * omega.`

```
= phi0 2
: t
```

**Compute** `fin 1 * omega.`

```
= omega
: t
```

```
Compute fin 42 * omega.
```

```
= omega
: t
```

We establish this representation as a *refinement* of the data types we used to represent ordinals less than  $\epsilon_0$ . Thus, many properties like well-foundedness of  $<$  and associativity of  $+$ , of this ordinal notations have very short proofs.

```
Fixpoint refine (a : t) : T1.T1 :=
  match a with
  | nil => T1.zero
  | cons i n b => T1.cons (\F i)%t1 n (refine b)
  end.
Lemma phi0_ref (i:nat) : refine (phi0 i) = T1.phi0 (\F i).
Proof. reflexivity. Qed.
Lemma succ_ref (alpha : t) :
  refine (succ alpha) = T1.succ (refine alpha).
Lemma plus_ref : forall alpha beta: t,
  refine (alpha + beta) = T1.T1add (refine alpha) (refine beta).
Lemma mult_ref : forall alpha beta: t,
  refine (alpha * beta) =
  T1.T1mul (refine alpha) (refine beta).
```

In order to make an ordinal notation for  $\omega^\omega$ , we follow the same steps as for  $\epsilon_0$ :

1. Define an order `lt`, which refines the order `lt` on `T1`.

```
#[ global ] Instance compare_oo : Compare t :=
fix cmp (a b : t) :=
  match a, b with
  | nil, nil => Eq
  | nil, cons a' n' b' => Datatypes.Lt
  | _ , nil => Gt
  | (cons a n b), (cons a' n' b') =>
    (match Nat.compare a a' with
    | Datatypes.Lt => Datatypes.Lt
    | Gt => Gt
    | Eq => (match Nat.compare n n' with
              | Eq => cmp b b'
              | comp => comp
            end)
    end)
  end)
end.
```

```
Lemma compare_ref (a b : t) :
  compare a b = compare (refine a) (refine b).
```

```

Definition lt (a b : t) : Prop :=
  compare a b = Datatypes.Lt.
Lemma lt_ref (a b : t) :
  lt a b <-> T1.lt (refine a) (refine b).

```

2. Define the predicate “to be in normal form”.

```

Fixpoint nf_b (alpha : t) : bool :=
  match alpha with
  | nil => true
  | cons a n nil => true
  | cons a n ((cons a' n' b') as b) =>
    (nf_b b && Nat.ltb a' a)%bool
  end.

```

```

Definition nf alpha : Prop := nf_b alpha.

```

3. Define a class `00` of terms in normal form, and an embedding from `E0` into `00`.

```

Class 00 : Type := mkord {data: L0.t ; data_ok : L0.nf data}.

Arguments data : clear implicit.
#[local] Hint Resolve data_ok : core.

Definition lt (alpha beta: 00) := L0.lt (data alpha) (data beta).
Definition le := leq lt.
#[ global ] Instance compare_00 : Compare 00 :=
  fun (alpha beta: 00) => L0.compare_oo (data alpha) (data beta).

#[ global ] Instance embed (alpha: 00) : E0.E0.
Proof.
  destruct alpha as [data Hdata].
  refine (@E0.mkord (L0.refine data) _).
  now apply nf_ref.
Defined.

```

4. Infer well-foundedness of the order on `00`.

```

Lemma lt_wf : well_founded lt.

#[ global ] Instance ON_00 : ON lt compare.
End 00.

```

Let us show a few examples.

```

Import 00.
#[local] Open Scope 00_scope.

Check phi0 7.

```

```
phi0 7
  : 00
```

```
#[global] Coercion Fin : nat -> 00.
```

```
Example Ex42: omega + 42 + omega^ 2 = omega^ 2.
rewrite <- Comparable.compare_eq_iff.
```

```
compare (omega + 42 + phi0 2) (phi0 2) = Eq
```

```
reflexivity.
```

```
Qed.
```

### 4.3.1 Related work

The article [BMR16] defines another representation of ordinals below  $\omega^\omega$  based on lists of natural numbers.

**Exercise 4.5** It may be interesting to write a *direct* proof of well-foundedness of the order in  $\omega^\omega$  (*i.e.* without using properties of  $\epsilon_0$ ). This exercise may help to understand better the proof structure of Sect. 4.2.1.2 on page 87.

## 4.4 A variant for hydra battles

In order to prove the termination of any hydra battle, we try to define a variant mapping hydras to ordinals strictly less than  $\epsilon_0$ . In order to make such a variant easy to define (for instance by a structural recursion), we introduce a variant of addition, which, contrary to  $+$ , is commutative and strictly monotonous in both of its arguments. This last property makes it possible to prove that our function is truly a variant for hydra battles (in Sect. 4.4.3 on page 99).

### 4.4.1 Natural sum (a.k.a. Hessenberg's sum)

Natural sum (Hessenberg sum) is a commutative and monotonous version of addition. It is used as an auxiliary operation for defining variants for hydra battles, where Hercules is allowed to chop off any head of the hydra.

In the literature, the natural sum of ordinals  $\alpha$  and  $\beta$  is often denoted by  $\alpha\#\beta$  or  $\alpha\oplus\beta$ . Thus we called `oplus` the associated *Coq* function.

#### 4.4.1.1 Definition of `oplus`

The definition of `oplus` is recursive in both of its arguments and uses the same pattern as for the `merge` function on lists of library `Coq.Sorting.Mergesort`.

1. Define a nested recursive function, using the `Fix` construct
2. Build a principle of induction dedicated to `oplus`
3. Establish equations associated to each case of the definition.

**4.4.1.1.1 Nested recursive definition** The following definition is composed of

- A main function `oplus`, structurally recursive in its first argument `alpha`
- An auxiliary function `oplus_aux` within the scope of `alpha`, structurally recursive in its argument `beta`; `oplus_aux beta` is supposed to compute `oplus alpha beta`.

*From Module `Epsilon0.Hessenberg`*

```

Fixpoint oplus (alpha beta : T1) : T1 :=
  let fix oplus_aux beta {struct beta} :=
    match alpha, beta with
    | zero, _ => beta
    | _, zero => alpha
    | cons a1 n1 b1, cons a2 n2 b2 =>
      match compare a1 a2 with
      | Gt => cons a1 n1 (oplus b1 beta)
      | Lt => cons a2 n2 (oplus_aux b2)
      | Eq => cons a1 (S (n1 + n2)%nat) (oplus b1 b2)
      end
    end
  in oplus_aux beta.

Infix "o+" := oplus (at level 50, left associativity).

```

The reader will note that each recursive call of the functions `oplus` and `oplus_aux` satisfies *Coq*'s constraint on recursive definitions. The function `oplus` is recursively called on a sub-term of its first argument, and `oplus_aux` on a sub-term of its unique argument. Thus, `oplus`'s definition is accepted by *Coq* as a structurally recursive function.

#### 4.4.1.2 Rewriting lemmas

*Coq*'s constraints on recursive definitions result in the quite complex form of `oplus`'s definition. Proofs of properties of this function can be simpler if we derive a few rewriting lemmas that will help to simplify expressions of the form  $(\text{oplus } \alpha \beta)$ .

A first set of lemmas correspond to the various cases of `oplus`'s definition. They can be proved almost immediately. Here are a few examples.

```

Lemma oplus_0_r (alpha : T1) : alpha o+ zero = alpha.

```

**Proof.**

```

  destruct alpha; reflexivity.

```

**Qed.**

```

Lemma oplus_0_l (beta : T1): zero o+ beta = beta.

```

**Proof.**

```

  destruct beta; reflexivity.

```

**Qed.**

```

Lemma oplus_compare_Lt:
  forall a n b a' n' b',
  compare a a' = Lt ->
  cons a n b o+ cons a' n' b' = cons a' n' (cons a n b o+ b').

```

**Project 4.4** Compare `oplus`'s definition (with inner fixpoint) with other possibilities (coq-equations, `Function`, etc.).

#### 4.4.2 More theorems on Hessenberg sum

We need to prove some properties of  $\oplus$ , particularly about its relation with the order  $<$  on `T1`.

##### 4.4.2.1 Commutativity, associativity

We prove the commutativity of  $\oplus$  in two steps. First, we prove by transfinite induction on  $\alpha$  that the restriction of  $\oplus$  to the interval  $[0, \alpha)$  is commutative.

```

Lemma oplus_comm_0 (gamma: T1):
  nf gamma ->
  forall alpha beta, nf alpha -> nf beta ->
    T1.lt alpha gamma ->
    T1.lt beta gamma ->
    alpha o+ beta = beta o+ alpha.

```

```

Proof.
  transfinite_induction gamma.
  (* ... *)

```

Then, we infer  $\oplus$ 's commutativity for any pair of ordinals: Let  $\alpha$  and  $\beta$  be two ordinals strictly less than  $\epsilon_0$ . Both ordinals  $\alpha$  and  $\beta$  are strictly less than  $\max(\alpha, \beta) + 1$ . Thus, we have just to apply the lemma `oplus_comm_0`.

```

Lemma oplus_comm (alpha beta: T1):
  nf alpha -> nf beta -> alpha o+ beta = beta o+ alpha.

```

```

Proof.
  intros ? ?; apply oplus_comm_0 with (T1.succ (max alpha beta));
  trivial.
  (* ... *)

```

Associativity of Hessenberg sum is proved the same way.

```

Lemma oplus_assoc_0 (alpha: T1):
  nf alpha ->
  forall a b c, nf a -> nf b -> nf c ->
    T1.lt a alpha ->
    T1.lt b alpha -> T1.lt c alpha ->
    a o+ (b o+ c) = (a o+ b) o+ c.

```

```

Proof.
  transfinite_induction_lt alpha.
  (* ... *)

```

```

Lemma oplus_assoc (alpha beta gamma:T1) :
  nf alpha -> nf beta -> nf gamma ->
  alpha o+ (beta o+ gamma) =
  alpha o+ beta o+ gamma.
Proof with eauto with T1.
  intros.
  apply oplus_assoc_0 with (T1.succ (max alpha (max beta gamma)));
  trivial.
  (* ... *)

```

#### 4.4.2.2 Monotony

At last, we prove that  $\oplus$  is strictly monotonous in both of its arguments.

```

Lemma oplus_strict_mono_LT_l (alpha beta gamma : T1) :
  nf gamma -> alpha t1< beta ->
  alpha o+ gamma t1< beta o+ gamma.

```

```

Lemma oplus_strict_mono_LT_r (alpha beta gamma : T1) :
  nf alpha -> beta t1< gamma ->
  alpha o+ beta t1< alpha o+ gamma.

```

**Project 4.5** The library `Hessenberg` looks too long (proof scripts and compilation). Please try to make it simpler and more efficient! Thanks!

 The module `gaia_hydras.GHessenberg` defines a version of Hessenberg sum compatible with Gaia's type `T1`.

#### 4.4.3 A termination measure for hydra battles

Let us define a measure from type `Hydra` into `T1`.

*From Module Hydra.Hydra\_Termination*

```

Fixpoint m (h:Hydra) : T1 :=
  match h with head => T1.zero
  | node hs => ms hs
end
with ms (s:Hydrae) : T1 :=
  match s with hnil => T1.zero
  | hcons h s' => T1.phi0 (m h) o+ ms s'
end.

```

First, we prove that the measure  $m(h)$  of any hydra  $h$  is a well-formed ordinal term of type `T1`.

```

Lemma m_nf : forall h, nf (m h).
Proof.
  induction h using Hydra_rect2
  with (P0 := fun s => nf (ms s)).
  (* ... *)

```

**Lemma** `ms_nf` : forall s, nf (ms s).

For proving the termination of all hydra battles, we have to prove that `m` is a variant. First, a few technical lemmas follow the decomposition of `round` into several relations. Then the lemma `round_decr` gathers all the cases.

**Lemma** `S0_decr`:

```
forall s s', S0 s s' -> ms s' t1 < ms s.
```

**Lemma** `R1_decr` :

```
forall h h', R1 h h' -> m h' t1 < m h.
```

**Lemma** `S1_decr n`:

```
forall s s', S1 n s s' -> ms s' t1 < ms s.
```

**Lemma** `R2_decr n` : forall h h', R2 n h h' -> m h' t1 < m h.

```
repeat split; auto with T1; now eapply R2_decr_0 with n. (* none *)
```

**Lemma** `round_decr` : forall h h', h -1-> h' -> m h' t1 < m h.

**Proof.**

```
destruct l as [n [H | H]].
- now apply R1_decr.
- now apply R2_decr with n.
```

**Qed.**

Finally, we prove termination of all (free) battles.

**#[global] Instance** `HVariant` : Hvariant Epsilon0 free var.

**Proof.**

```
split; intros; eapply round_decr; eauto.
```

**Qed.**

**Theorem** `every_battle_terminates` : Termination.

**Proof.**

```
red; apply Inclusion.wf_incl with (R2 := fun h h' => m h t1 < m h').
red; intros; now apply round_decr.
apply Inverse_Image.wf_inverse_image, T1_wf.
```

**Qed.**

 The module `gaia_hydras.GHydra` contains a proof of Gaia-hydras version of termination of all battles.

**Fixpoint** `m` (h:Hydra) : T1 :=

```
if h is node (hcons _ _ as hs) then ms hs else zero
```

**with** `ms` (s : Hydrae) : T1 :=

```
if s is hcons h s' then phi0 (m h) o+ ms s' else zero.
```

**Compute** `T1pp` (m Examples.Hy).

```
= (ω ^ (ω ^ ω ^ 2 + 1) + 2)%pT1
: ppT1
```

**Lemma** `mVariant`: Hvariant `nf_Wf` free `m` .

**Theorem** `every_battle_terminates` : Termination.



## Chapter 5

# Accessibility inside $\epsilon_0$ : The Ketonen-Solovay Machinery

### 5.1 Introduction

The reader may think that our proof of termination in the previous chapter requires a lot of mathematical tools and may be too complex. So, the question is “is there any simpler proof” ?

In their article [KP82], Kirby and Paris show that this result cannot be proved in Peano arithmetic. Their proof uses some knowledge about model theory and non-standard models of Peano arithmetic. In this chapter, we focus on a specific class of proofs of termination of hydra battles: construction of some variant mapping the type `Hydra` into a given initial segment of ordinals. Our proof relies only on the Calculus of Inductive Constructions and is a natural complement of the results proven in the previous chapters.

- There is no variant mapping the type `Hydra` into the interval  $[0, \omega^2)$  (section 3.8.2 on page 62), and *a fortiori*  $[0, \omega)$  (section 2.4.3 on page 45).
- There exists a variant which maps the type `Hydra` into the interval  $[0, \epsilon_0)$  (theorem `every_battle_terminates`, in section 4.4.3 on page 99).

Thus, a very natural question is the following one:

“ Is there any variant from `Hydra` into some interval  $[0, \mu)$ , where  $\mu < \epsilon_0$ , for proving the termination of all hydra battles ?”

We prove in Coq the following result:

There is no variant for proving the termination of all hydra battles from `Hydra` into the interval  $[0, \mu)$ , where  $\mu < \epsilon_0$ . The same impossibility holds even if we consider only standard battles (with the successive replication factors  $0, 1, 2, \dots, t, t + 1, \dots$ ).

Our proofs are constructive and require no axioms: they are closed terms of the CIC, and are mainly composed on function definitions and proofs of properties of these functions. They borrow much theoretical material from Kirby

and Paris, although they do not use any knowledge about Peano arithmetic nor about model theory. The combinatorial arguments we use and implement come from an article by J. Ketonen and R. Solovay [KS81], already cited in the work by L. Kirby and J. Paris. Section 2 of this article: "A hierarchy of probably recursive functions", contains a systematic study of *canonical sequences*, which are closely related to rounds of hydra battles. Nevertheless, they have the same global structure as the simple proofs described in sections 2.4.3 on page 45 and 3.8.2 on page 62. We invite the reader to compare the three proofs step by step, lemma by lemma.

## 5.2 Canonical Sequences

Canonical sequences are functions that associate an ordinal  $\{\alpha\}(i)$  to every ordinal  $\alpha < \epsilon_0$  and positive integer  $i$ . They satisfy several nice properties:

- If  $\alpha \neq 0$ , then  $\{\alpha\}(i) < \alpha$ . Thus canonical sequences can be used in proofs by transfinite induction or function definition by transfinite recursion
- If  $\lambda$  is a limit ordinal, then  $\lambda$  is the least upper bound of the set  $\{\{\lambda\}(i) \mid i \in \mathbb{N}_1\}$
- If  $\beta < \alpha < \epsilon_0$ , then there is a "path" from  $\alpha$  to  $\beta$ , *i.e.* a sequence  $\alpha_0 = \alpha, \alpha_1, \dots, \alpha_n = \beta$ , where for every  $k < n$ , there exists some  $i_k$  such that  $\alpha_{k+1} = \{\alpha_k\}(i_k)$

**Remark 5.1** Canonical sequences are defined for any ordinal  $\alpha < \epsilon_0$ , by stating that if  $\alpha$  is a successor ordinal  $\beta+1$ , the sequence associated with  $\alpha$  is simply the constant sequence whose terms are equal to  $\beta$ . Likewise, the canonical sequence of 0 maps any natural number to 0.

Thus, the function that maps any ordinal  $\alpha$  and natural number  $i$  to the ordinal  $\{\alpha\}(i)$  is total.

### 5.2.1 Canonical sequences and hydra battles

Canonical sequences correspond tightly to rounds of hydra battles: if  $\alpha \neq 0$ , then  $\iota(\alpha)$  is transformed into  $\iota(\{\alpha\}(i+1))$  in one round with the replication factor  $i$  (Lemma Hydra.O2H.canonS\_iota\_i). Thus, whenever  $\beta < \alpha < \epsilon_0$ , there exists a (free) battle from  $\iota(\alpha)$  to  $\iota(\beta)$ .

### 5.2.2 Definitions

First, let us recall how canonical sequences are defined in [KS81]. For efficiency's sake, we decided not to implement directly K.&S's definitions, but to define in Gallina simply typed structurally recursive functions which share the abstract properties which are used in the mathematical proofs.

#### 5.2.2.1 Mathematical definition of canonical sequences

In [KS81] the definition of  $\{\alpha\}(i)$  is based on the following remark:

Any non-zero ordinal  $\alpha$  can be decomposed in a unique way as the product  $\omega^\beta \times (\gamma + 1)$ .

Thus the  $\{\alpha\}(i)$  s are defined in terms of this decomposition:

**Definition 5.1 (Canonical sequences: mathematical definition)**

- Let  $\lambda < \epsilon_0$  be a limit ordinal
  - If  $\lambda = \omega^{\alpha+1} \times (\beta + 1)$ , then  $\{\lambda\}(i) = \omega^{\alpha+1} \times \beta + \omega^\alpha \times i$
  - If  $\lambda = \omega^\gamma \times (\beta + 1)$ , where  $\gamma < \lambda$  is a limit ordinal, then  $\{\lambda\}(i) = \omega^\gamma \times \beta + \omega^{\{\gamma\}(i)}$
- For successor ordinals, we have  $\{\alpha + 1\}(i) = \alpha$
- Finally,  $\{0\}(i) = 0$ .

**5.2.2.2 Canonical sequences in Coq**

Our definition may look more complex than the mathematical one, but uses plain structural recursion over the type **T1**. Thus, tactics like **cbn**, **simpl**, **compute**, etc., are applicable.

*From Module Epsilon0.Canon*

```

Fixpoint canon alpha (i:nat) : T1 :=
  match alpha with
  | zero => zero
  | cons zero 0 zero => zero
  | cons zero (S k) zero => FS k
  | cons gamma 0 zero => (match T1.pred gamma with
    Some gamma' =>
      match i with
      | 0 => zero
      | S j => cons gamma' j zero
      end
    | None => cons (canon gamma i) 0 zero
    end)

  | cons gamma (S n) zero =>
    (match T1.pred gamma with
    Some gamma' =>
      (match i with
      0 => cons gamma n zero
      | S j => cons gamma n (cons gamma' j zero)
      end)
    | None => cons gamma n (cons (canon gamma i) 0 zero)
    end)

  | cons alpha n beta => cons alpha n (canon beta i)
end.

```

**G** The translation of `canon` compatible with Gaia's data-types is defined in `gaia_hydras.GCanon` (please see Sect. 7.3.2).

For instance Coq's computing facilities allow us to verify the equalities

$$\{\omega^\omega\}(3) = \omega^3 \quad \text{and} \quad \{\omega^{\omega^{\omega+1}+1}\}(42) = \omega^{\omega^{\omega+1}} \times 42$$

**Compute** `pp (canon (phi0 Tlomega) 3).`

```
= ω ^ 3
: ppT1
```

**Compute** `pp (canon (Tlomega ^ (Tlomega ^ (Tlomega + 1) + 1))%t1 42).`

```
= ω ^ ω ^ (ω + 1) * 42
: ppT1
```

### 5.2.3 Basic properties of canonical sequences

We did not try to prove that our definition truly implements Ketonen and Solovay's [KS81]'s canonical sequences. The most important is that we were able to prove the abstract properties of canonical sequences that are really used in our proof. The complete proofs are in the module `Epsilon0.Canon`. For instance, the equality  $\{\alpha + 1\}(i) = \alpha$  can be proved by structural induction on  $\alpha$ .

**Lemma** `canon_succ i alpha :`

```
nf alpha -> canon (succ alpha) i = alpha.
```

**Proof.**

```
revert i; induction alpha.
```

```
forall i : nat, nf zero -> canon (succ zero) i = zero
alpha1: T1
n: nat
alpha2: T1
IHalpha1: forall i : nat,
  nf alpha1 -> canon (succ alpha1) i = alpha1
IHalpha2: forall i : nat,
  nf alpha2 -> canon (succ alpha2) i = alpha2
-----
forall i : nat,
nf (cons alpha1 n alpha2) ->
canon (succ (cons alpha1 n alpha2)) i =
cons alpha1 n alpha2

(* ... *)
```

#### 5.2.3.1 Canonical sequences and the order $<$

We prove by transfinite induction over  $\alpha$  that  $\{\alpha\}(i)$  is an ordinal strictly less than  $\alpha$  (assuming  $\alpha \neq 0$ ). This property allows us to use the function `canonS` and its derivatives in function definitions by transfinite recursion.

**Lemma** `canon_LT i alpha :` `nf alpha -> alpha <> zero ->`  
`canon alpha i t1< alpha.`

 This lemma is also available in Library `gaia_hydras.GCanon`:

```
Lemma canon_lt (i : nat) [a : T1]: T1nf a -> a <> zero -> canon a i < a.
```

### 5.2.3.2 Limit ordinals are truly limits

The following theorem states that any limit ordinal  $\lambda < \epsilon_0$  is the limit of the sequence  $\{\lambda\}(i)$  ( $1 \leq i$ ).

*From Module `Epsilon0.Canon`*

```
Lemma canonS_limit_strong lambda :
  nf lambda -> T1limit lambda ->
  forall beta, beta t1< lambda -> {i:nat | beta t1< canon lambda (S i)}.
```

**Proof.**

```
transfinite_induction lambda.
(* ... *)
```

**Defined.**

Note the use of Coq's `sig` type in the theorem's statement, which expresses a constructive view of the limit of a sequence: for any  $\beta < \lambda$ , we can compute an item of the canonical sequence of  $\lambda$  which is greater than  $\beta$ . We can also state directly that  $\lambda$  is a (strict) least upper bound of the elements of its canonical sequence.

```
Lemma canonS_limit_lub (lambda : T1) :
  nf lambda -> T1limit lambda -> strict_lub (fun i => canon lambda (S i)) lambda.
```

 In Gaia-hydras, the statement use a slightly different vocabulary:

```
Lemma canon_limit_of lambda (Hnf : T1nf lambda) (Hlim : T1limit lambda) :
  limit_of (canon lambda) lambda.
```

**Exercise 5.1** Instead of using the `sig` type, define a simply typed function that, given two ordinals  $\alpha$  and  $\beta$ , returns a natural number  $i$  such that, if  $\alpha$  is a limit ordinal and  $\beta < \alpha$ , then  $\beta < \{\alpha\}(i + 1)$ . Of course, you will have to prove the correctness of your function.

**Hint:** You may add to your function a third argument usually called `fuel` for allowing you to give a structurally recursive function (*cf* the post of Guillaume Melquiond on Coq-club (Dec 21, 2020) <https://sympa.inria.fr/sympa/arc/coq-club/2020-12/msg00069.html>). The type `fuel`, an alternative to `nat` is available on `Prelude.Fuel` .

## 5.3 Accessibility inside $\epsilon_0$ : paths

Let us consider a kind of accessibility problem inside  $\epsilon_0$ : given two ordinals  $\alpha$  and  $\beta$ , where  $\beta < \alpha < \epsilon_0$ , find a *path* consisting of a finite sequence  $\gamma_0 = \alpha, \dots, \gamma_l = \beta$ , where, for every  $i < l$ ,  $\gamma_i \neq 0$  and there exists some strictly positive integer  $s_i$  such that  $\gamma_{i+1} = \{\gamma\}(s_i)$ .

Let  $s$  be the sequence  $\langle s_0, s_1, \dots, s_{l-1} \rangle$ . We describe the existence of such a path with the notation  $\alpha \xrightarrow{s} \beta$ . We say also that the considered path from  $\alpha$  to  $\beta$  starts at [index]  $s_0$  and ends at  $s_l$ .

For instance, we have  $\omega * 2 \xrightarrow{2,2,2,4,5} 3$ , through the path  $\langle \omega \times 2, \omega + 2, \omega + 1, \omega, 4, 3 \rangle$ .

**Remark 5.2** Note that, given  $\alpha$  and  $\beta$ , where  $\beta < \alpha$ , the sequence  $s$  which leads from  $\alpha$  to  $\beta$  is not unique.

For instance, we have  $\omega \times 2 \xrightarrow{2} \omega$  and  $\omega \times 2 \xrightarrow{3,4,5,6} \omega$ . Likewise,  $\omega \times 2 \xrightarrow{1,2,1,4} 0$  and  $\omega \times 2 \xrightarrow{3,3,3,3,3,3,3} 0$ .

### 5.3.1 Formal definition

In Coq, the notion of path can be simply defined as an inductive predicate parameterized by the destination  $\beta$ .

*From Module Epsilon0.Paths*

```
Definition transition_S i : relation T1 :=
  fun alpha beta => alpha <> zero /\ beta = canon alpha (S i).
```

```
Definition transition i : relation T1 :=
  match i with 0 => fun _ _ => False | S j => transition_S j end.
```

```
Inductive path_to (beta: T1) : list nat -> T1 -> Prop :=
| path_to_1 : forall (i:nat) alpha ,
  i <> 0 -> transition i alpha beta -> path_to beta (i::nil) alpha
| path_to_cons : forall i alpha s gamma,
  i <> 0 -> transition i alpha gamma ->
  path_to beta s gamma -> path_to beta (i::s) alpha.
```

```
Definition path alpha s beta := path_to beta s alpha.
```

**Remark 5.3** In the present version of our library, we use a variant `path_to5` of `path_to`, where the proposition  $(\text{path\_to5 } \beta \ s \ \alpha)$  is equivalent to  $(\text{path\_to } \beta \ (\text{List.map } S \ s) \ \alpha)$ . This variant is scheduled to be deprecated.

 The library `gaia_hydras.GPaths` transposes the notion of path into Gaia's type `T1` (please see Sect. 7.3.3).

**Exercise 5.2** Write a tactic for solving goals of the form  $(\text{path\_to } \beta \ s \ \alpha)$  where  $\alpha$ ,  $\beta$  and  $s$  are closed terms. You should solve automatically the following goals:

```
Example ex_path1: path_to Tlomega (2::2::2::nil) (Tlomega * 2).
Proof. path_tac. Qed.
```

```
Example ex_path2: path_to Tlomega (3::4::5::6::nil) (Tlomega * 2).
Proof. path_tac. Qed.
```

**Example ex\_path3:** path\_to zero (interval 3 14) (Tlomega \* 2).

**Proof.** cbn; path\_tac. Qed.

**Example ex\_path4:** path\_to zero (List.repeat 3 8) (Tlomega \* 2).

**Proof.** cbn; path\_tac. Qed.

### 5.3.2 Existence of a path

By transfinite induction on  $\alpha$ , we prove that for any  $\beta < \alpha$ , one can build a path from  $\alpha$  to  $\beta$  (in other terms,  $\beta$  is accessible from  $\alpha$ ).

**Lemma LT\_path\_to** (alpha beta : T1) :

beta t1< alpha -> {s : list nat | path\_to beta s alpha}.

By the lemma canon\_LT (Sct.5.2.3.1 on page 104), we can convert any path into an inequality on ordinals (by induction on paths).

**Lemma path\_to\_LT** beta s alpha :

path\_to beta s alpha -> nf alpha -> beta t1< alpha.

**Exercise 5.3 (continuation of exercise 5.1 on page 105)** Define a simply typed function for computing a path from  $\alpha$  to  $\beta$ .

### 5.3.3 Paths and hydra battles

In order to apply our knowledge about ordinal numbers less than  $\epsilon_0$  to the study of hydra battles, we define an injection from the interval  $[0, \epsilon_0)$  into the type Hydra.

*From Module Hydra.O2H*

**Fixpoint iota** (a : T1) : Hydra :=

```
match a with
| zero => head
| cons c n b => node (hcons_mult (iota c) (S n) (iotas b))
end
```

**with iotas** (a : T1) : Hydrae :=

```
match a with
| zero => hnil
| cons a0 n b => hcons_mult (iota a0) (S n) (iotas b)
end.
```

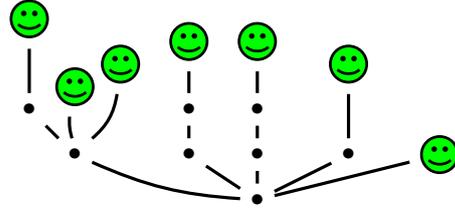
For instance Fig. 5.1 shows the image by  $\iota$  of the ordinal  $\omega^{\omega+2} + \omega^\omega \times 2 + \omega + 1$

The following lemma (proved in Hydra.O2H.v) maps canonical sequences to rounds of hydra battles.

**Lemma canonS\_iota** i a :

nf a -> a <> 0 -> iota a -1-> iota (canon a (S i)).

The next step of our development extends this relationship to the order  $<$  on  $[0, \epsilon_0)$  on one side, and hydra battles on the other side.

Figure 5.1: The hydra  $\iota(\omega^{\omega+2} + \omega^\omega \times 2 + \omega + 1)$ 

**Lemma** `path_to_round_plus a s b :`  
`path_to b s a -> nf a -> iota a --> iota b.`

As a corollary, we are now able to transform any inequality  $\beta < \alpha < \epsilon_0$  into a (free) battle.

**Lemma** `LT_to_round_plus a b : b t1< a -> iota a --> iota b.`

## 5.4 A proof of impossibility

We now have the tools for proving that there exists no variant bounded by some  $\mu < \epsilon_0$  for proving the termination of all battles. The proof we are going to show is a proof by contradiction. It can be considered as a generalization of the proofs described in sections 2.4.3 on page 45 and 3.8.2 on page 62.

In the module `Hydra.Epsilon0_Needed_Generic`, we assume there exists some variant  $m$  bounded by some ordinal  $\mu < \epsilon_0$ . This part of the development is parameterized by some class  $B$  of battles, which will be instantiated later to `free` or `standard`.

From `Hydra.Hydra_Definitions`:

```
Class BoundedVariant {A:Type}{Lt:relation A}
  {Wf: well_founded Lt}{B : Battle}
  {m: Hydra -> A} (Var: Hvariant Wf B m)(mu:A):=
  { m_bounded: forall h, Lt (m h) mu }.
```

Let us assume there exists such a variant:

**Section** `Bounded`.

```
Context (B: Battle)
  (mu: T1)
  (Hmu: nf mu)
  (m : Hydra -> T1)
  (Var : Hvariant T1_wf B m)
  (Hy : BoundedVariant Var mu).
```

**Hypothesis** `m_decrease` : forall i h h', round\_n i h h' -> m h' t1< m h.

**Remark 5.4** The hypothesis `m_decrease` is not provable in general, but is satisfied by the `free` and `standard` kinds of battles. This trick allows to “factorize” our proofs of impossibility.

First, we prove that  $m(\iota(\alpha))$  is always greater than or equal to  $\alpha$ , by transfinite induction over  $\alpha$ .

**Lemma `m_ge_0 alpha`:** `nf alpha -> alpha t1<= m (iota alpha)`.

- If  $\alpha = 0$ , the inequality trivially holds
- If  $\alpha$  is the successor of some ordinal  $\beta$ , the inequality  $\beta \leq m(\iota(\beta))$  holds (by induction hypothesis). But the hydra  $\iota(\alpha)$  is transformed in one round into  $\iota(\beta)$ , thus  $m(\iota(\beta)) < m(\iota(\alpha))$ . Hence  $\beta < m(\iota(\alpha))$ , which implies  $\alpha \leq m(\iota(\alpha))$
- If  $\alpha$  is a limit ordinal, then  $\alpha$  is the least upper bound of the set of all the  $\{\alpha\}(i)$ . Thus, we have just to prove that  $\{\alpha\}(i) < m(\iota(\alpha))$  for any  $i$ .
  - Let  $i$  be some natural number. By the induction hypothesis, we have  $\{\alpha\}(i) \leq m(\iota(\{\alpha\}(i)))$ . But the hydra  $\iota(\alpha)$  is transformed into  $\iota(\{\alpha\}(i))$  in one round, thus  $m(\iota(\{\alpha\}(i))) < m(\iota(\alpha))$ , by our hypothesis `m_decrease`.

Please note that the impossibility proofs of sections 2.4.3 on page 45 and 3.8.2 on page 62 contain a similar lemma, also called `m_ge`. We are now able to build a counter-example.

**Definition `big_h`:** `:= iota mu`.

**Definition `beta_h`:** `:= m big_h`.

**Definition `small_h`:** `:= iota beta_h`.

**Lemma `mu_beta_h`:** `acc_from mu beta_h`.

**Proof.** `apply LT_acc_from, m_bounded. Qed.`

**Corollary `m_ge_generic`:** `m big_h t1<= m small_h`.

**Proof.** `apply m_ge_0, nf_m. Qed.`

**End Bounded.**

The (big) rest of the proof is dedicated to prove formally the converse inequality `m small_h t1< m big_h`.

### 5.4.1 The case of free battles

Let us now consider that  $B$  is instantiated to `free` (which means that we are considering proofs of termination of *all* battles). The following lemmas are proved in Module `Hydra.Epsilon0_Needed_Free`. The case  $B = \text{standard}$  is studied in section 5.5 on page 111.

**Section `Impossibility_Proof`.**

**Context** (`mu`: T1)

```

(Hmu: nf mu)
(m : Hydra -> T1)
(Var : Hvariant T1_wf free m)
(Hy : BoundedVariant Var mu).

```

```

Let big_h := big_h mu.
Let small_h := small_h mu m.

```

1. The following lemma is an application of `m_ge_generic`, since `free` satisfies trivially the hypothesis `m_decrease` (see page 108).

```

Lemma m_ge : m big_h t1<= m small_h.
Proof.
  apply m_ge_generic with (l := Hy).
  (* ... *)

```

2. From the hypothesis `Hy`, we have `m big_h t1< mu`
3. By Lemma `LT_to_round_plus`, we get a (free) battle from `big_h = iota mu` to `small_h = iota (m big_h)`.

```

Lemma big_to_small : big_h ->= small_h.
Proof.
  unfold big_h, small_h. apply LT_to_round_plus; auto.
  unfold beta_h. apply (m_bounded big_h); auto.
Qed.

```

4. From the hypotheses on `m`, we infer:

```

Lemma m_lt : m small_h t1< m big_h.
Proof. apply m_variant_LT, big_to_small. Qed.

```

5. From lemmas `m_ge` and `m_lt`, and the irreflexivity of `<`, we get a contradiction.

```

Fact self_lt_free : m big_h t1< m big_h .
Proof.
  apply LE_LT_trans with (m small_h).
  - apply m_ge.
  - apply m_lt.
Qed.

```

```

Theorem Impossibility_free : False.
Proof. apply (LT_irrefl self_lt_free). Qed.

```

```

End Impossibility_Proof.

```

We have now proved there exists no bounded variant for the class of free battles.

**Check** `Impossibility_free`.

```
Impossibility_free
: forall (mu : T1) (m : Hydra -> T1)
  (Var : Hvariant T1_wf free m),
  BoundedVariant Var mu -> False
```

 Please look at the Gaia version of this theorem in Sect. 7.4.5 on page 159.

## 5.5 The case of standard battles

One may wonder if our theorem holds also in the framework of standard battles. Unfortunately, its proof relies on the lemma `LT_to_round_plus` of Module `Hydra.O2H`.

**Lemma** `LT_to_round_plus a b` :  $b \text{ t1} < a \rightarrow \text{iota } a \text{ --> } \text{iota } b$ .

This lemma builds a battle out of any inequality  $\beta < \alpha$ . It is a straightforward application of `LT_path_to` of Module `Epsilon0.Paths`:

**Lemma** `LT_path_to (alpha beta : T1)` :  
 $\text{beta t1} < \text{alpha} \rightarrow \{s : \text{list nat} \mid \text{path\_to } \text{beta } s \text{ alpha}\}$ .

The sequence  $s$ , used to build the sequence of replication factors of the battle, depends on  $\alpha$  and  $\beta$ , so we cannot be sure that the generated battle is a genuine standard battle.

The solution to this issue comes once again from Ketonen and Solovay’s article [KS81]. Instead of considering plain paths, i.e. sequences  $\alpha_0 = \alpha, \alpha_1, \dots, \alpha_k = \beta$  where  $\alpha_{j+1}$  is equal to  $\{\alpha_j\}(i_j)$  where  $i_j$  is *any* natural number, we consider various constraints on these sequences. In particular, a path is called *standard* if  $i_{j+1} = i_j + 1$  for every  $j < k$ . It corresponds to a “segment” of some standard battles. Please note that the vocabulary on paths is ours, but all the concepts come really from [KS81].

In Coq, standard paths can be defined as follows.

*From Module `Epsilon0.Paths`*

```
Inductive standard_path_to (j:nat)(beta : T1): nat -> T1 -> Prop :=
| std_1 : forall i alpha,
  alpha <> zero ->
  beta = canon alpha i -> j = i -> i <> 0 ->
  standard_path_to j beta i alpha
| std_S : forall i alpha,
  standard_path_to j beta (S i) (canon alpha i) ->
  standard_path_to j beta i alpha.
```

**Definition** `standard_path i alpha j beta` :=  
`standard_path_to j beta i alpha`.

In the mathematical text and figures, we shall use the notation  $\alpha \xrightarrow{i,j} \beta$  for the proposition (standard\_path  $i$   $\alpha$   $j$   $\beta$ ). In [KS81] the notation is  $\alpha \xrightarrow{i}^* \beta$  for the proposition  $\exists j, i < j \wedge \alpha \xrightarrow{i,j} \beta$ .

Our goal is now to transform any inequality  $\beta < \alpha < \epsilon_0$  into a standard path  $\alpha \xrightarrow{i,j} \beta$  for some  $i$  and  $j$ , which corresponds to a standard battle from  $\iota(\alpha)$  (at time  $i$ ) to  $\iota(\beta)$  (at time  $j$ ). Following [KS81], we proceed in two stages:

1. we simulate plain (free) paths from  $\alpha$  to  $\beta$  with paths made of steps  $(\gamma, \{\gamma\}(n))$ , with the same  $n$  all along the path.
2. we simulate any such path by a standard path.

### 5.5.1 Paths with a constant index

First of all, paths with a constant index enjoy nice properties. They are defined as paths where all the  $i_j$  are equal to the same natural number  $i$ , for some  $i > 0$ .

Like in [KS81], we shall use the notation  $\alpha \xrightarrow{i} \beta$  for denoting such a path, also called an  $i$ -path.

**Definition const\_pathS  $i$  :=**

```
clos_trans_ln T1 (fun alpha beta => alpha <> zero /\
  beta = canon alpha (S i)).
```

**Definition const\_path  $i$  alpha beta :=**

```
match i with
  0 => False
| S j => const_pathS j alpha beta
end.
```

A most interesting property of  $i$ -paths is that we can “upgrade” their index, as stated by K.&S.’s Corollary 12.

*From Module Epsilon0.Paths*

**Corollary Cor12** (alpha : T1) : nf alpha ->  
 forall beta i n, beta t1< alpha ->  
 i < n ->  
 const\_path (S i) alpha beta ->  
 const\_path (S n) alpha beta.

**Proof.**

```
transfinite_induction_lt alpha.
(* ... *)
```

We also use a version of Cor12 with large inequalities.

**Corollary Cor12\_1** (alpha : T1) :

```
nf alpha ->
forall beta i n, beta t1< alpha ->
  i <= n ->
  const_path (S i) alpha beta ->
  const_path (S n) alpha beta.
```

Cor12 is a consequence of the following theorem (numbered 2.4 in Ketonen and Solovay’s article), proven by transfinite induction on  $\alpha$ .

```
Theorem KS_thm_2_4 (lambda : T1) :
  nf lambda -> Tllimit lambda ->
  forall i j, (i < j)%nat ->
    const_path 1 (canon lambda (S j)) (canon lambda (S i)).
```

```
Proof.
  transfinite_induction lambda.
  (* ... *)
```

### 5.5.1.1 Sketch of proof of Cor12

Cor12 is also proved by transfinite induction on  $\alpha$ . Let us give a sketch of its proof<sup>1</sup>

Let us consider a path  $\alpha \xrightarrow{i} \beta$  ( $i > 0$ ). Its first step is the pair  $(\alpha, \{\alpha\}(i))$ . We have  $\{\alpha\}(i) < \alpha$  and  $\{\alpha\}(i) \xrightarrow{i} \beta$ . Let  $n$  be any natural number such that  $n > i$ . By the induction hypothesis, there exists a path  $\{\alpha\}(n) \xrightarrow{i} \beta$ .

- If  $\alpha$  is a successor ordinal  $\gamma + 1$ , then  $\{\alpha\}(n) = \{\alpha\}(i) = \gamma$ . Thus we have a path  $\alpha \xrightarrow{n} \gamma \xrightarrow{n} \beta$
- If  $\alpha$  is a limit ordinal, we apply Theorem KS\_thm\_2\_4 (see above).

The proof of the limit case, is decomposed into a sequence of path constructions leading to  $\alpha \xrightarrow{n} \beta$ .

1.  $\alpha \xrightarrow{n} \{\alpha\}(n)$  (single step path)
2.  $\{\alpha\}(n) \xrightarrow{1} \{\alpha\}(i)$  (by Theorem\_2\_4),
3.  $\{\alpha\}(n) \xrightarrow{n} \{\alpha\}(i)$  (applying the induction hypothesis to the preceding path);
4.  $\{\alpha\}(i) \xrightarrow{n} \beta$  (applying the induction hypothesis)
5.  $\alpha \xrightarrow{n} \beta$  (by composition of 1, 3, and 4).

**Remark 5.5** Cor12 “casts”  $i$ -paths into  $n$ -paths for any  $n > i$ . But the obtained  $n$ -path can be much longer than the original  $i$ -path. The following exercise will give an idea of this increase.

**Exercise 5.4** Prove that the length of the  $i + 1$ -path from  $\omega^\omega$  to  $\omega^i$  is  $1 + (i + 1)^{(i+1)}$ , for any  $i$ . Note that the  $i$ -path from  $\omega^\omega$  to  $\omega^i$  is only one step long.

<sup>1</sup>This proof sketch is a slight simplification of the formal proof script: The strictly positive indexes  $i$  and  $n$  stand for the terms (S i) and (S n). We do not explicit the (simpler) case where the considered path is made of only one step.

Why is Cor12 so useful? Let us consider two ordinals  $\beta < \alpha < \epsilon_0$ . By induction on  $\alpha$ , we decompose any inequality  $\beta < \alpha$  into  $\beta < \{\alpha\}(i) < \alpha$ , where  $i$  is some natural number. Applying corollary Cor12' we build a  $n$ -path from  $\beta$  to  $\alpha$ , where  $n$  is the maximum of the indices  $i$  met in the induction.

Lemma 1, Section 2.6 of [KS81] is naturally expressed in terms of Coq's sig construct.

```
Lemma Lemma2_6_1 (alpha : T1) :
  nf alpha ->
  forall beta, beta t1< alpha ->
    {n:nat | const_path (S n) alpha beta}.
```

**Proof.**

```
transfinite_induction alpha.
(* ... *)
```

**Defined.**

Intuitively, Lemma 2\_6\_1 shows that if  $\beta < \alpha < \epsilon_0$ , then there exists a battle from  $\iota(\alpha)$  to  $\iota(\beta)$  where the replication factor is constant, although large enough.

 Corollary Cor12 and Lemma Lemma2\_6\_1 are also available in gaia\_hydras.GPaths (please see Sect. 7.3.3 on page 151).

## 5.5.2 Casting paths with a constant index into a standard path

The article [KS81] contains the following lemma, which allows us to simulate  $i$ -paths by  $[i + 1, j]$ -paths, where  $j$  is large enough.

```
Lemma constant_to_standard (alpha beta : T1) (n : nat):
  nf alpha -> const_pathS n alpha beta ->
  {l : nat | standard_gnaw (S n) alpha l = beta}.
```

### 5.5.2.1 Sketch of proof of constant\_to\_standard\_path

Our proof follows the proof by Ketonen and Solovay, including its organization as a sequence of lemma. Since it is a non-trivial proof, we will comment its main steps below (see Figure 5.2 on the next page to Figure 5.5 on page 116).

#### Preliminaries

Please note that, given an ordinal  $\alpha : T1$ , and two natural numbers  $i$  and  $l$ , there exists at most a standard path  $\alpha \xrightarrow[i, i+l]{*} \beta$ . The following function computes  $\beta$  from  $\alpha$ ,  $i$  and  $l$ .

```
Fixpoint standard_gnaw (i:nat)(alpha : T1)(l:nat): T1 :=
  match l with
  | 0 => alpha
  | S m => standard_gnaw (S i) (canon alpha i) m
  end.
```

By transfinite induction over  $\alpha$ , we prove that the ordinal 0 is reachable from any ordinal  $\alpha < \epsilon_0$  by some standard path.

**Lemma standard\_path\_to\_zero:**

```
forall alpha i, nf alpha -> alpha <> zero ->
  {j: nat | standard_path (S i) alpha j zero}.
```

Now, let us consider two ordinals  $\beta < \alpha < \epsilon_0$ . Let  $p$  be some  $(n + 1)$ -path from  $\alpha$  to  $\beta$ .

**Section Constant\_to\_standard\_Proof.**

**Variables** (alpha beta: T1) (n : nat).

**Hypotheses** (Halpha: nf alpha) (Hpos : zero t1< beta)

(Hpa : const\_pathS n alpha beta).

Applying standard\_path\_to\_zero, 0 is reachable from  $\alpha$  by some standard path.



Figure 5.2: A nice proof (1)

Since comparison on T1 is decidable, one can compute the last step  $\gamma$  of the standard path from  $(\alpha, n + 1)$  such that  $\beta \leq \gamma$ . Let  $l$  be the length of the path from  $\alpha$  to  $\gamma$ . This step of the proof is illustrated in figure 5.3 <sup>2</sup>.

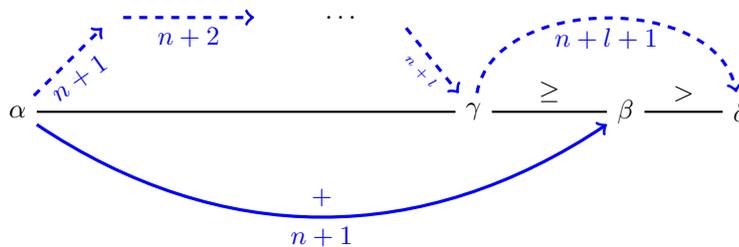


Figure 5.3: A nice proof (2)

- If  $\beta = \gamma$ , it's OK! We have got a standard path from  $\alpha$  to  $\beta$  with successive indices  $n + 1, n + 2, \dots, n + l + 1$

<sup>2</sup>Please note that in these figures, smaller ordinals are represented on the right side!

- Otherwise,  $\beta < \gamma$ . Let us consider  $\delta = \{\gamma\}(n+l+1)$ . By applying several times lemma Cor12, one converts every path of Fig 5.3 into a  $n+l+1$ -path (see figure 5.4).

But  $\gamma$  is on the  $n+l+1$ -path from  $\alpha$  to  $\beta$ . As shown by figure 5.5, the ordinal  $\delta$ , reachable from  $\gamma$  in one single step, must be greater than or equal to  $\beta$ , which contradicts our hypothesis  $\beta < \gamma$ .

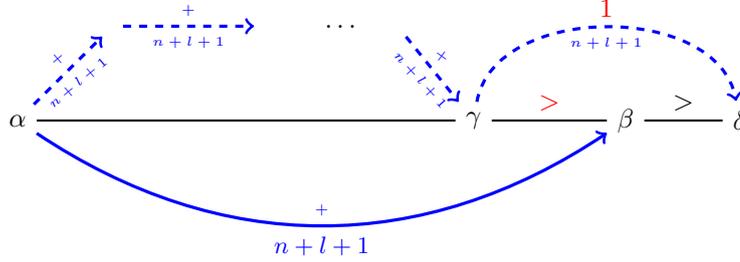


Figure 5.4: A nice proof (3)

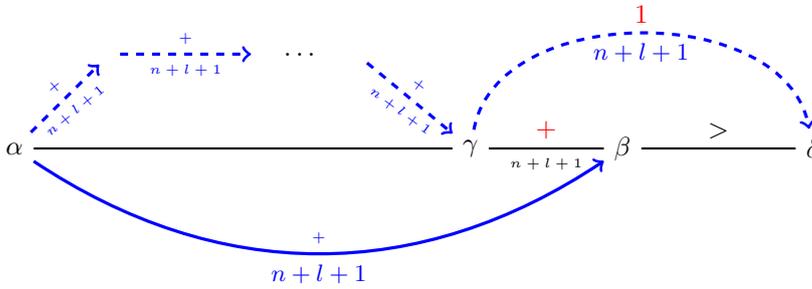


Figure 5.5: A nice proof (4)

The only possible case is thus  $\beta = \gamma$ , so we have got a standard path from  $\alpha$  to  $\beta$ .

**Lemma constant\_to\_standard\_0 :**  
 $\{l : \text{nat} \mid \text{standard\_gnaw } (S \ n) \ \text{alpha } l = \text{beta}\}.$

**End Constant\_to\_standard\_Proof.**

Here is the full statement of the conversion from constant to standard paths.

**Lemma constant\_to\_standard\_path**  
 $(\text{alpha } \text{beta} : \text{T1}) \ (\text{i} : \text{nat}) :$   
 $\text{nf } \text{alpha} \rightarrow \text{const\_paths } i \ \text{alpha } \text{beta} \rightarrow \text{zero } \text{t1} < \text{alpha} \rightarrow$   
 $\{j : \text{nat} \mid \text{standard\_path } (S \ i) \ \text{alpha } j \ \text{beta}\}.$

Applying Lemma2\_6\_1 and constant\_to\_standard\_path, we get the following corollary.

**Corollary LT\_to\_standard\_path**  $(\text{alpha } \text{beta} : \text{T1}) :$   
 $\text{beta } \text{t1} < \text{alpha} \rightarrow$   
 $\{n : \text{nat} \ \& \ \{j : \text{nat} \mid \text{standard\_path } (S \ n) \ \text{alpha } j \ \text{beta}\}\}.$

### 5.5.3 Back to hydras

We are now able to complete our proof that there exists no bounded variant for proving the termination of standard hydra battles. This proof can be consulted in the module `Hydra.Epsilon0_Needed_Std`. Please note that it has the same global structure as in section 5.4.1

Applying the lemmas `Lemma2_6_1` of the module `Epsilon0.pathS` and `constant_to_standard_path`, we can convert any inequality  $\beta < \alpha < \epsilon_0$  into a standard path from  $\alpha$  to  $\beta$ , then into a fragment of a standard battle from  $\iota(\alpha)$  to  $\iota(\beta)$ , hence the inequality  $m(\iota(\beta)) < m(\iota(\alpha))$ .

*From Module Hydra.Epsilon0\_Needed\_Std*

```
Lemma LT_to_standard_battle :
  forall alpha beta,
    beta t1< alpha ->
      exists n i, rounds standard n (iota alpha) i (iota beta).
```

Now, please consider the following context:

**Section** `Impossibility_Proof`.

```
Context (mu: T1)
  (m : Hydra -> T1)
  (Var : Hvariant T1_wf standard m)
  (Hy : BoundedVariant Var mu).
```

```
Let big_h := big_h mu.
Let small_h := small_h mu m.
```

In the same way as for free battles, we import a large inequality from the module `Epsilon0_Needed_Generic`. (see Sect. 5.4 on page 109).

```
Lemma m_ge : m big_h t1<= m small_h.
```

If remains to prove the following strict inequality, in order to have a contradiction.

```
Lemma m_lt : m small_h t1< m big_h.
```

**Sketch of proof:** Let us recall that  $\text{big\_h} = \iota(\mu)$  and  $\text{small\_h} = \iota(m(\text{big\_h}))$ .

Since  $m(\text{big\_h}) < \mu$ , there exists a standard path from  $\mu$  to  $m(\text{big\_h})$ , hence a standard battle from  $\iota(\mu)$  to  $\iota(m(\text{big\_h}))$ , i.e. from  $\text{big\_h}$  to  $\text{small\_h}$ .

Since  $m$  is assumed to be a variant for standard battles, we get the inequality  $m(\text{small\_h}) < m(\text{big\_h})$ .

```
Fact self_lt_standard : m big_h t1< m big_h.
```

**Proof.**

```
apply LE_LT_trans with (m small_h);[apply m_ge | apply m_lt].
```

**Qed.**

```
Theorem Impossibility_std: False.
```

**Proof.** `apply (LT_irrefl self_lt_standard).` **Qed.**

**End Impossibility\_Proof.**

**Check** `Impossibility_std.`

```
Impossibility_std
: forall (mu : T1) (m : Hydra -> T1)
  (Var : Hvariant T1_wf standard m),
  BoundedVariant Var mu -> False
```



Please look at the Gaia version of this theorem in Sect. 7.4.5 on page 159.

### 5.5.4 Remarks

We are grateful to J. Ketonen and R. Solovay for the high quality of their explanations and proof details. Our proof follows tightly the sequence of lemmas in their article, with a focus on constructive aspects. Roughly speaking, our implementation *builds*, out of a hypothetical variant  $m$ , bounded by some ordinal  $\mu < \epsilon_0$ , a hydra `big_h` which verifies the impossible inequality  $m(\text{big\_h}) < m(\text{big\_h})$ .

One may ask whether the preceding results are too restrictive, since they refer to a particular data type `T1`. In fact, our representation of ordinals strictly less than  $\epsilon_0$  is faithful to their mathematical definition, at least Kurt Schütte's [Sch77], as proved in Chapter 8 on page 161. (please see also the module `hydras.Schutte.Correctness_E0`).

Thus, we can infer that our theorems can be applied to any well order.

**Project 5.1** Study a possible modification of the definition of a variant (for standard battles).

- The variant is assumed to be strictly decreasing *on configurations reachable from some initial configuration where the replication factor is equal to 0*
- The variant may depend on the number of the current round.

In other words, its type should be `nat -> Hydra -> T1`, and it must verify the inequality  $m(Si)h' < mih$  whenever the configuration  $(i, h)$  is reachable from some initial configuration  $(0, h_0)$  and  $h$  is transformed into  $h'$  in the considered round. Can we still prove the theorems of section 5.5 with this new definition?

## Chapter 6

# Large sets and rapidly growing functions

**Remark 6.1** Some notations (mainly names of fast-growing functions) of our development may differ slightly from the literature. Although this fact does not affect our proofs, we are preparing a future version where the names  $F_\alpha$ ,  $f_\alpha$ ,  $H_\alpha$ , etc., are fully consistent with the cited articles.

In this chapter, we try to feel how long a standard battle can be. To be precise, for any ordinal  $\alpha < \epsilon_0$  and any positive integer  $k$ , we give a minoration of the number of steps of a standard battle which starts with the hydra  $\iota(\alpha)$  and the replication factor  $k$ .

We express this number in terms of the Hardy hierarchy of fast-growing functions [WB87, Wai70, KS81, Prö13]. From the Coq user’s point of view, such functions are very attractive: they are defined as functions in Gallina, and we can apply them *in theory*, but they are so complex that you will never be able to look at the result of the computation. Thus, our knowledge on these functions must rely on *proofs*, not tests. In our development, we use often the rewriting rules generated by Coq’s Equations plug-in.

### 6.1 Definitions

**Definition 6.1** Let  $0 < \alpha < \epsilon_0$  be any ordinal, and  $s = \langle s_1, s_2, \dots, s_N \rangle$  a finite sequence of strictly positive natural numbers.

We say that  $s$  is  $\alpha$ -large if the sequence  $\langle \alpha_0 = \alpha, \dots, \alpha_{i+1} = \{\alpha_i\}(i+1), \dots \rangle$  leads to 0. We say also that  $s$  is minimally  $\alpha$ -large (in short:  $\alpha$ -mlarge) if  $s$  is  $\alpha$ -large and every strict prefix of  $s$  leads to a non-zero ordinal (cf Sect. 5.3.1 on page 106).

**Remark 6.2** Ketonen and Solovay [KS81] consider large finite *sets* of natural numbers, but they are mainly used as sequences. Thus, we chose to represent them explicitly as (sorted) lists.

The following function “gnaws” an ordinal  $\alpha$ , following a sequence of indices (ignoring the 0s).

From Module *Epsilon0.Paths*

```
Fixpoint gnaw (alpha : T1) (s : list nat) :=
  match s with
  | nil => alpha
  | (0::s') => gnaw alpha s'
  | (S i :: s') => gnaw (canon alpha (S i)) s'
  end.
```

From Module *Epsilon0.Large\_Sets*

```
Definition largeb (alpha : T1) (s : list nat) :=
  match (gnaw alpha s)
  with zero => true | _ => false end.
```

```
Definition large (alpha : T1) (s : list nat) : Prop :=
  largeb alpha s.
```

Minimal large sequences can be directly defined in terms of the predicate `path_to` (5.3.1 on page 106) which already prohibits paths containing non-final zeros.

From Module *Epsilon0.Large\_Sets*

```
Definition mlarge alpha (s : list nat) := path_to zero s alpha.
```

Let us consider two integers  $k$  and  $l$ , such that  $0 < k < l$ . In order to check whether the interval  $[k, l]$  is minimally large for  $\alpha$ , it is enough to follow from  $\alpha$  the path associated with the interval  $[k, l]$  and verify that the last ordinal we obtain is equal to 1.

### 6.1.1 Examples

For instance the interval  $[6, 70]$  leads  $\omega^2$  to  $\omega \times 2 + 56$ . Thus this interval is not  $\omega^2$ -large.

From Module *Epsilon0.Large\_Sets\_Examples*

```
Compute pp (gnaw (T1omega * T1omega) (interval 6 70)).
```

```
= (ω * 2 + 56)%pT1
: ppT1
```

The interval  $[6, 700]$  is  $\omega^2$ -large, but not  $\omega^2$ -mlarge, since  $[6, 699]$  is also  $\omega^2$ -large.

```
Compute (gnaw (T1omega * T1omega) (interval 6 700)).
```

```
= zero
: T1
```

```
Compute (gnaw (T1omega * T1omega) (interval 6 699)).
```

```
= zero
: T1
```

The following lemma relates minimal largeness with the function `gnaw`.

```
Lemma mlarge_iff alpha x (s:list nat) :
  s <> nil -> ~ In 0 (x::s) ->
  mlarge alpha (x::s) <-> gnaw alpha (but_last x s) = one.
```

*From Module Epsilon0.Large\_Sets\_Examples*

```
Example Ex1 : mlarge (Tlomega * Tlomega) (interval 6 510).
Proof with try (auto with arith || discriminate ).
  unfold interval; simpl Peano.minus.
  do 2 rewrite iota_from_unroll; rewrite mlarge_iff ...
  repeat rewrite not_in_cons ...
Qed.
```

## 6.2 Length of minimal large sequences

Now, let us consider some natural number  $k > 0$  and an ordinal  $0 < \alpha < \epsilon_0$ . We would like to compute a number  $l$  such that the interval  $[k, l]$  is  $\alpha$ -mlarge. So, the standard battle starting with  $\iota(\alpha)$  and the replication factor  $k$  will end after  $(l - k + 1)$  steps.

First, we notice that this number  $l$  exists, since the segment  $[0, \epsilon_0)$  is well-founded and  $\{\alpha\}(i) < \alpha$  for any  $i$  and  $\alpha > 0$ . Moreover, it is unique:

*From Module Epsilon0.Large\_Sets*

```
Lemma mlarge_unicity alpha k l l' :
  mlarge alpha (interval (S k) l) ->
  mlarge alpha (interval (S k) l') ->
  l = l'.
```

Thus, we would like to define a function, parameterized by  $\alpha$  which associates to any strictly positive integer  $k$  the number  $l$  such that the interval  $[k, l]$  is  $\alpha$ -mlarge. It would be fine to write in Gallina a definition like this:

```
Function L_ (alpha: E0) (i:nat) : nat := ...
```

But we do not know how to fill the dots yet ... In the next section, we will use Coq to reason about the *specification* of  $L$ , prove properties of any function which satisfies this specification. In Sect. 6.2.4, we use the `coq-equations` plug-in to define  $L$ , then prove its correctness w.r.t. its specification.

### 6.2.1 Formal specification

Let  $0 < \alpha < \epsilon_0$  be an ordinal term. We consider any function which maps any strictly positive integer  $k$  to the number  $l$ , where the interval  $[k, l]$  is  $\alpha$ -mlarge.

**Remark 6.3** In [KS81] Ketonen and Solovay consider the least natural number  $l$  where the interval  $[k, l]$  ( $l$  included) is  $\alpha$ -large, and call  $H_\alpha$  the function which maps  $k$  to  $l$ . We chose to consider intervals  $[l, k)$  instead of  $[l, k]$  in order to simplify some statements and proofs in composition lemmas associated with the ordinals of the form  $\alpha \times i$  and  $\omega^\alpha \times i + \beta$ . Clearly, both approaches are related through the equality  $L_\alpha(k) = H_\alpha(k) + 1$ , for any non-null  $\alpha$  and  $k$ .

Our specification of the function  $L$  is as follows:

*From Module `Epsilon0.Large_Sets`*

```
Inductive L_spec : T1 -> (nat -> nat) -> Prop :=
  L_spec0 :
    forall f, (forall k, f (S k) = S k) -> L_spec zero f
| L_spec1 : forall alpha f,
  alpha <> zero ->
  (forall k,
    mlarge alpha (interval (S k) (Nat.pred (f (S k)))) ->
    L_spec alpha f.
```

**To do 6.1** Check if the functions  $L_\alpha$  are the same as [KS81]' functions  $f_\alpha$  (p. 297).

Note that, for  $\alpha \neq 0$ , the value of  $f(0)$  is not specified. Nevertheless, the restriction of  $f$  to the set of strictly positive integers is unique (up to extensionality).

```
Lemma L_spec_unicity alpha f g :
  L_spec alpha f -> L_spec alpha g -> forall k, f (S k) = g (S k).
```

## 6.2.2 Abstract properties

Let us now prove properties of any function  $f$  (if any) which satisfies `L_spec`. We are looking for properties which could be used for writing *equations* and prove the correctness of the function generated by the `coq-equations` plug-in. Moreover, they will give us some examples (for small values of  $\alpha$ ).

The properties we consider are defined in `Prelude.Iterates`.

```
Definition strict_mono f := forall n p, n < p -> f n < f p.
```

```
Definition dominates_from n g f := forall p, n <= p -> f p < g p.
```

```
Definition fun_le f g := forall n:nat, f n <= g n.
```

```
Infix "<=<=" := fun_le (at level 60).
```

```
Definition dominates g f := exists n : nat, dominates_from n g f .
```

```
Infix ">>" := dominates (at level 60).
```

```
Definition dominates_strong g f := {n : nat | dominates_from n g f}.
```

```
Infix ">>s" := dominates_strong (at level 60).
```

Our exploration of the  $L_\alpha$ s considers the usual cases of a proof by transfinite induction: zero, successors and limit ordinals. The lemmas we are going to prove will be applied in a big proof by induction in Sect 6.2.4.1 on page 128.

### 6.2.2.1 The ordinal zero

The base case is directly a consequence of the specification.

```
Lemma L_zero_inv f : L_spec zero f -> forall k, f (S k) = S k.
```

### 6.2.2.2 Successor ordinals

Let  $\beta$  be some ordinal, and assume the arithmetic function  $f$  satisfies the specification  $(L\_spec\ \beta)$ . Let  $k$  be any natural number. Any path from  $\text{succ}\ \beta$  to 0 starting at  $k + 1$  can be decomposed into a first step from  $\text{succ}\ \beta$  to  $\beta$ , then a path from  $\beta$  at  $k + 2$  to 0. By hypothesis the interval  $[k + 2, f(k + 2) - 1]$  is  $\beta$ -mlarge. But the interval  $[k + 1, f(k + 2) - 1]$  is the concatenation of the singleton  $\{k + 1\}$  and the interval  $[k + 2, f(k + 2) - 1]$ . So, the function  $\lambda k. f(k + 1)$  satisfies the specification  $L\_spec\ \beta$ .

Note that our decomposition of intervals works only if the intervals we consider are not empty. In order to ensure this property, we assume that  $f\ k$  is always greater than  $k$ , which we note  $S \ll= f$ , or  $(\text{fun\_le}\ S\ f)$ .

*From Module `Epsilon0.Large_Sets`*

#### Section `succ`.

**Variables** `(beta : T1) (f : nat -> nat)`.

**Hypotheses** `(Hbeta : nf beta)`  
`(f_mono : strict_mono f)`  
`(f_Sle : S <= f)`  
`(f_ok : L_spec beta f)`.

**Definition** `L_succ := fun k => f (S k)`.

**Lemma** `L_succ_mono : strict_mono L_succ`.

**Lemma** `L_succ_Sle : S <= L_succ`.

**Lemma** `L_succ_ok : L_spec (succ beta) L_succ`.

#### End `succ`.

### 6.2.2.3 Limit ordinals

Let  $\lambda < \epsilon_0$  be any limit ordinal. In a similar way as for successors, we decompose any path from  $\lambda$  into a first step to  $\{\lambda\}(k)$ , followed by a path to 0. In the following section, we assume that there exists a correct function computing  $L_{\{\lambda\}}(k)$  for any strictly positive  $k$ .

#### Section `lim`.

**Variables** `(lambda : T1)`  
`(Hnf : nf lambda)`  
`(Hlim : Tlimit lambda)`  
`(f : nat -> nat -> nat)`  
`(H : forall k, L_spec (canon lambda (S k)) (f (S k)))`.

**Remark** `canon_not_null : forall k, canon lambda (S k) <> zero`.

**Definition** `L_lim k := f k (S k)`.

**Lemma** `L_lim_ok : L_spec lambda L_lim`.

End `lim`.

### 6.2.3 First results

Applying the previous lemmas on successors and limit ordinals, we obtain a few correct implementations of  $(L\_spec\ \alpha)$  for small values of  $\alpha$ .

#### 6.2.3.1 Finite ordinals

By iterating the functional `L_succ`, we get a realization of  $(L\_spec\ i)$  for any natural number  $i$ .

**Definition** `L_fin i := fun k => (i + k)%nat.`

**Lemma** `L_fin_ok i : L_spec (\F i) (L_fin i).`

**Proof.**

```
induction i.
(* ... *)
```

#### 6.2.3.2 The first limit ordinal $\omega$

The lemmas `L_fin_ok` and `L_lim_ok` allow us to get by diagonalization a correct implementation for  $L\_spec\ \omega$ .

**Definition** `L_omega k := S (2 * k)%nat.`

**Lemma** `L_omega_ok : L_spec Tlomega L_omega.`

**Proof.**

```
pose (H:= L_lim_ok Tlomega nf_omega refl_equal L_fin
      (fun i => L_fin_ok (S i))).
eapply L_spec_compat with (l:=H);
intro ; unfold L_lim, L_fin, L_omega; abstract lia.
```

**Qed.**

#### 6.2.3.3 Towards $\omega^2$

We would like to get exact formulas for the ordinal  $\omega^2$ , a.k.a.  $\phi_0(2)$ . This ordinal is the limit of the sequence  $\omega \times i$  ( $i \in \mathbb{N}$ ). Thus, we have to study ordinals of this form, then use our lemma on limits.

The following lemma establishes a path from  $\omega \times (i + 1)$  to  $\omega \times i$ .

**Lemma** `path_to_omega_mult (i k:nat) :`

```
path_to (Tlomega * i)
  (interval (S k) (2 * (S k))%nat)
  (Tlomega * (S i)).
```

Let us consider a path from  $\omega \times (i + 1)$  to 0 starting at  $k + 1$ . A first “big step” will lead to  $\omega \times i$  at  $2(k + 1)$ . If  $i > 0$ , the next jump leads to  $\omega \times (i - 1)$  at  $2(2(k + 1)) + 1$ , etc.

The following lemma expresses the length of the `mlarge` sequences associated with any finite multiple of  $\omega$ .

```

Lemma omega_mult_mlarge_0 i : forall k,
  mlarge (Tlomega * (S i))
    (interval (S k)
      (Nat.pred (iterate (fun p => S (2 * p)%nat)
        (S i)
        (S k))))).

```

From Module *Epsilon0.Large\_Sets*

```

Definition L_omega_mult i (x:nat) := iterate L_omega i x.

```

```

Compute L_omega_mult 8 5.

```

```

= 1535
: nat

```

More generally, we prove the equality  $L_{\omega \times i}(k) = 2^i \times (k + 1) - 1$ .

```

Lemma L_omega_mult_eqn (i : nat) :
  forall (k : nat),
    (0 < k)%nat -> L_omega_mult i k = (exp2 i * S k - 1)%nat.

```

Correctness of the function `L_omega_mult` is asserted through the following lemma.

```

Lemma L_omega_mult_ok (i: nat) :
  L_spec (Tlomega * i) (L_omega_mult i).

```

By diagonalization, we obtain a simple formula for  $L_{\omega^2}$ .

```

Definition L_omega_square k :=
  iterate (fun z => S (2 * z)%nat) k (S k).

```

```

Compute L_omega_square 8.

```

```

= 2559
: nat

```

```

Lemma L_omega_square_eqn k :
  (0 < k)%nat ->
  L_omega_square k = (exp2 k * (k + 2) - 1)%nat.

```

```

Lemma L_omega_square_ok:
  L_spec (Tlomega * Tlomega) L_omega_square.

```

### 6.2.3.4 Going further

Let us consider a last example, “computing”  $L_{\omega^3}$ . Let us study the canonical sequence associated with  $\omega^3$ , the elements of which are  $\omega^2 \times i$  ( $i \in \mathbb{N}_1$ ).

To this end, we prove a generic lemma, which expresses  $L_{\omega^\alpha \times i}$  as an iterate of  $L_{\omega^\alpha}$ . Note that in this lemma, we assume that the function associated with  $\alpha$  is strictly monotonous and greater or equal than the successor function, and prove that  $L_{\omega^\alpha \times i}$  satisfies the same properties.



This number is quite big. Using `ocaml`'s float arithmetic, we can under-approximate it by  $2^{3.8 \times 10^{30}} \times 3.8 \times 10^{30}$ .

```
# let exp2 x = 2.0 ** x;;

val exp2 : float -> float = <fun>
# exp2 95.0 *. 97.0 -. 1.0;;
- : float = 3.84256588194182037e+30
# let n = exp2 95.0 ;;
# let p = n *. 97.0 -. 1.0;;
val p : float = 3.84256588194182037e+30

Estimation :
2 ** (3.84 e+30) * 3.84 e+30.
```

## 6.2.4 Using Equations

Note that we did not define any function  $L_\alpha$  for any  $\alpha < \epsilon_0$  yet. We have got no more than a collection of proved realizations of `L_spec`  $\alpha$  for a few values of  $\alpha$ .

Using the `coq-equations` plug-in by M. Sozeau [SM19], we will now define a function `L_` which maps any ordinal  $\alpha < \epsilon_0$  to a proven realization of `L_spec`  $\alpha$ . To this end, we represent ordinals as inhabitants of the type `E0` of well-formed ordinal terms (see Sect 4.1.7.1 on page 81). So, we define a total function `L_` of type `E0 -> nat -> nat`, by transfinite recursion, considering the usual three cases :  $\alpha = 0$ ,  $\alpha$  is a successor,  $\alpha$  is a limit ordinal.

### 6.2.4.1 Definition

*From Module `L_alpha`.*

```
From Equations Require Import Equations.
Import RelationClasses Relations.
```

```
#[global] Instance 0lt : WellFounded E0lt := E0lt_wf.
#[global] Hint Resolve 0lt : E0.
```

```
(** Using Coq-Equations for building a function which satisfies
   [Large_sets.L_spec] **)
```

```
Equations L_ (alpha: E0) (i:nat) : nat by wf alpha E0lt :=
  L_alpha i with E0_eq_dec alpha E0zero :=
  { | left _zero => i ;
    | right _nonzero
      with Utils.dec (E0limit alpha) :=
      { | left _limit => L_ (Canon alpha i) (S i) ;
        | right _successor => L_ (E0_pred alpha) (S i)} }.
Solve All Obligations with auto with E0.
```

This definition results in a bunch of automatically generated lemmas. For instance:

**About** `L_equation_1`.

```
L_equation_1 :
forall (alpha : E0) (i : nat),
L_alpha i =
L_unfold_clause_1 alpha (E0_eq_dec alpha E0zero) i

L_equation_1 is not universe polymorphic
Arguments L_equation_1 alpha i%nat_scope
L_equation_1 is transparent
Expands to: Constant L_alpha.L_equation_1
```

In most cases, it may be useful to write human-readable paraphrases of these statements.

**Lemma** `L_zero_eqn` : forall i, L\_E0zero i = i.

**Proof.** `intro i; now rewrite L_equation_1. Qed.`

**Lemma** `L_eq2 alpha i` :

`E0is_succ alpha -> L_alpha i = L_ (E0_pred alpha) (S i).`

**Lemma** `L_succ_eqn alpha i` :

`L_ (E0_succ alpha) i = L_alpha (S i).`

**Lemma** `L_lim_eqn alpha i` :

`E0limit alpha ->
L_alpha i = L_ (Canon alpha i) (S i).`

Using these three lemmas as rewrite rules, we can prove more properties of the functions  $L_\alpha$ .

**Lemma** `L_finite` : forall i k :nat, L\_i k = (i+k)%nat.

**Lemma** `L_omega` : forall k, L\_E0\_omega k = S (2 \* k)%nat.

By well-founded induction on  $\alpha$ , we prove the following properties:

**Lemma** `L_ge_S alpha` :

`alpha <> E0zero -> S <= L_alpha.`

**Theorem** `L_correct alpha` : L\_spec (cnf alpha) (L\_alpha).

Please note that the proof of `L_correct` applies the lemmas proven in Sections 6.2.2.1, 6.2.2.2 and 6.2.2.3. Our previous study of `L_spec` allowed us to pave the way for the definition by Equations and the correctness proof.



The module `gaia_hydras.GL_alpha` contains an adaptation of `Epsilon0/L_alpha`.

### 6.2.4.2 Back to hydra battles

Lemma `battle_length_std` of Module `Hydra.Battle_length` relates the length of standard battles with the functions  $L_\alpha$ .

**Lemma `battle_length_std`:**

`battle_length standard k (iota (cnf alpha)) (l-k)%nat.`

**Exercise 6.1** Instead of considering standard paths and battles, consider battles where the replication factor is a constant  $k$ . Please use `Equations` in order to define the function that computes the length of the  $k$ -path which leads from  $\alpha$  to 0. Prove a few exact formulas and minoration lemmas.

## 6.3 A variant of the Hardy hierarchy

In order to give a feeling on the complexity of the functions  $L_\alpha$ s, we compare them with a better known family of functions, the *Hardy hierarchy* of fast growing functions, presented for instance in [Prö13].

**Remark 6.4** Indeed, the functions presented in this section are a *variant* of the Hardy hierarchy of functions. In the future versions of this development, we will correct the references to the literature. For the time being, we call our functions  $H'_\alpha$  in order to underline the difference from “classic” Hardy functions.

For each ordinal  $\alpha$  below  $\epsilon_0$ ,  $H'_\alpha$  is a total arithmetic function, defined by transfinite recursion on  $\alpha$ , according to three cases:

- If  $\alpha = 0$ , then  $H'_\alpha(k) = k$  for any natural number  $k$ .
- If  $\alpha = \text{succ}(\beta)$ , then  $H'_\alpha(k) = H'_\beta(k + 1)$  for any  $k \in \mathbb{N}$
- If  $\alpha$  is a limit ordinal, then  $H'_\alpha(k) = H'_{\{\alpha\}(k+1)}(k)$  for any  $k \in \mathbb{N}$ .

**Remark 6.5** The “classic” definition of the Hardy hierarchy differs in the third equation.

- If  $\alpha = 0$ , then  $H_\alpha(k) = k$  for any natural number  $k$ .
- If  $\alpha = \text{succ}(\beta)$ , then  $H_\alpha(k) = H_\beta(k + 1)$  for any  $k \in \mathbb{N}$
- If  $\alpha$  is a limit ordinal, then  $H_\alpha(k) = H_{\{\alpha\}(k)}(k)$  for any  $k \in \mathbb{N}$ .

### 6.3.1 Definition in `Coq`

We define a function `H'_` of type `E0 -> nat -> nat` by transfinite induction over the type `E0` of the well formed ordinals below  $\epsilon_0$ .

*From Module `Epsilon0.Hprime`*

```

Equations H'_ (alpha: E0) (i:nat) : nat by wf alpha E0lt :=
  H'_ alpha i with E0_eq_dec alpha E0zero :=
  { | left _zero => i ;
    | right _nonzero
      with Utils.dec (E0limit alpha) :=
      { | left _limit => H'_ (Canon alpha (S i)) i ;
        | right _successor => H'_ (E0_pred alpha) (S i)}}.

```

**Solve All Obligations with** `auto with E0`.

**Lemma** `H'_eq1` : forall i, H'\_ E0zero i = i.

**Proof.**

```
intro i; now rewrite H'__equation_1.
```

**Qed.**

**Lemma** `H'_eq2_0` alpha i :  
`E0is_succ alpha ->`  
`H'_ alpha i = H'_ (E0_pred alpha) (S i).`

**Lemma** `H'_eq3` alpha i :  
`E0limit alpha -> H'_ alpha i = H'_ (Canon alpha (S i)) i.`

**Lemma** `H'_eq2` alpha i :  
`H'_ (E0_succ alpha) i = H'_ alpha (S i).`

## 6.3.2 First steps of the H' hierarchy

Using rewrite rules from `H'_eq1` to `H'_succ_eqn`, we can explore the functions  $H'_\alpha$  for small values of  $\alpha$ .



The module `gaia_hydras.GHprime` contains an adaptation of `Epsilon0.Hprime`.

### 6.3.2.1 Finite ordinals

By induction on  $i$ , we prove a simple expression of `H'_ (Fin i)`, where `Fin i` is the  $i$ -th finite ordinal.

**Lemma** `H'_Fin` : forall i k : nat, H'\_ (E0fin i) k = (i+k)%nat.

**Proof with** `eauto with E0`.

```

induction i.
- intros; simpl E0fin; simpl; autorewrite with H'_rw E0_rw ...
- intros ;simpl; autorewrite with H'_rw E0_rw ...
  rewrite IHi; abstract lia.

```

**Qed.**

### 6.3.2.2 Multiples of $\omega$

Since the canonical sequence of  $\omega$  is composed of finite ordinals, it is easy to get the formula associated with  $H'_\omega$ .

**Lemma H'\_omega** : forall k, H'\_ E0\_omega k = S (2 \* k)%nat.

**Proof with** auto with E0.

```
intro k; rewrite H'_eq3 ...
- replace (Canon E0_omega (S k)) with (E0fin (S k)).
+ rewrite H'_Fin; abstract lia.
+ now autorewrite with E0_rw.
```

**Qed.**

Before going further, we prove a useful rewriting lemma:

**Lemma H'\_Plus\_Fin alpha** : forall i k : nat,  
H'\_ (alpha + i)%e0 k = H'\_ alpha (i + k)%nat.

**Proof.**

```
induction i.
(* ... *)
```

Then, we get easily formulas for  $H'_{\omega+i}$ , and  $H'_{\omega \times i}$  for any natural number  $i$ .

**Lemma H'\_omega\_double k** :

H'\_ (E0\_omega \* 2)%e0 k = (4 \* k + 3)%nat.

**Proof.**

```
rewrite H'_eq3; simpl Canon; [ | now compute].
ochange (Canon (E0_omega * E0finS 1)%e0 (S k)) (E0_omega + (S k))%e0;
rewrite H'_Plus_Fin, H'_omega; abstract lia.
```

**Qed.**

**Lemma H'\_omega\_3 k** : H'\_ (E0\_omega \* 3)%e0 k = (8 \* k + 7)%nat.

**Lemma H'\_omega\_4 k** : H'\_ (E0\_omega \* 4)%e0 k = (16 \* k + 15)%nat.

**Lemma H'\_omega\_i (i:nat)** : forall k,

H'\_ (E0\_omega \* i)%e0 k = (exp2 i \* k + Nat.pred (exp2 i))%nat.

**Proof.**

```
induction i.
(* ... *)
```

Since  $\omega^2$  is a limit ordinal, we prove the following equality:

$$H'_{\omega^2}(k) = 2^{k+1} \times (k + 1) - 1$$

**Lemma H'\_omega\_sqr** : forall k,

H'\_ (E0\_phi0 2)%e0 k = (exp2 (S k) \* (S k) - 1)%nat.

### 6.3.2.3 New limits

Our next step would be to prove an exact formula for  $H'_{\omega^\omega}(k)$ . Since the canonical sequence of  $\omega^\omega$  is composed of all the  $\omega^i$ , we first need to express  $H'_{\omega^i}$  for any natural number  $i$ .

Let  $i$  and  $k$  be two natural numbers. The ordinal  $\{\omega^{(i+1)}\}(k)$  is the product  $\omega^i \times k$ , so we need also to consider ordinals of this form.

1. First, we express  $H'_{\omega^\alpha \times (i+2)}$  in terms of  $H'_{\omega^\alpha \times (i+1)}$ .

**Lemma H'\_Omega\_term\_1** :  $\alpha \lt \text{E0zero} \rightarrow \text{forall } k,$   
 $\text{H}'_ (\text{Omega\_term } \alpha \text{ (S } i)) k =$   
 $\text{H}'_ (\text{Omega\_term } \alpha \text{ } i) (\text{H}'_ (\text{E0\_phi0 } \alpha) k).$

2. Then, we prove by induction on  $i$  that  $H'_{\omega^\alpha \times (i+1)}$  is just the  $(i+1)$ -th iterate of  $H'_{\omega^\alpha}$ .

**Lemma H'\_Omega\_term** ( $\alpha : \text{E0}$ ) :  
 $\text{forall } i k,$   
 $\text{H}'_ (\text{Omega\_term } \alpha \text{ } i) k =$   
 $\text{iterate } (\text{H}'_ (\text{E0\_phi0 } \alpha)) \text{ (S } i) k.$

3. In particular, we derive a formula for  $H'_{\omega^{i+1}}$ .

**Definition H'\_succ\_fun**  $f k := \text{iterate } f \text{ (S } k) k.$

**Lemma H'\_Phi0\_Si** :  $\text{forall } i k,$   
 $\text{H}'_ (\text{E0\_phi0 } \text{ (S } i)) k = \text{iterate } \text{H}'_ \text{succ\_fun } i \text{ (H}'_ \text{E0\_omega}) k.$

4. We get now a formula for  $H'_{\omega^3}$ :

**Lemma H'\_omega\_cube** :  $\text{forall } k,$   
 $\text{H}'_ (\text{E0\_phi0 } 3) k = \text{iterate } (\text{H}'_ (\text{E0\_phi0 } 2)) \text{ (S } k) k.$

#### 6.3.2.4 A numerical example

It is hard to capture the complexity of this function by looking only at this “exact” formula. Let us consider a simple example: the number  $H'_{\omega^3}(3)$ .

**Let**  $f k := (\text{exp2 } \text{(S } k) * \text{(S } k) - 1) \text{nat}.$

**Remark R0**  $k : \text{H}'_ (\text{E0\_phi0 } 3) k = \text{iterate } f \text{ (S } k) k.$

**Proof.**

$\text{ochange } (\text{E0\_phi0 } 3) (\text{E0\_phi0 } (\text{E0\_succ } 2)); \text{rewrite } \text{H}'_ \text{Phi0\_succ}.$   
 $\text{unfold } \text{H}'_ \text{succ\_fun}; \text{apply } \text{iterate\_ext}.$   
 $- \text{intro } x; \text{now } \text{rewrite } \text{H}'_ \text{omega\_sqr}.$

**Qed.**

Thus, the number  $H_{\omega^3}(3)$  can be written as four nested applications of  $f$ .

**Fact F0** :  $\text{H}'_ (\text{E0\_phi0 } 3) 3 = f (f (f (f 3))).$

**Proof.**  $\text{rewrite } \text{R0}; \text{reflexivity. Qed}.$

In a more classical writing, this number is displayed as follows:

$$H'_{\omega^3}(3) = 2^{(2^{N+1}(N+1))} (2^{N+1} (N + 1)) - 1$$

In order to make this statement more readable, we introduce a local definition.

**Let**  $N := (\text{exp2 } 64 * 64 - 1)\%nat.$

This number looks quite big; let us compute an approximation with `0caml`:

```
# (2.0 ** 64.0 *. 64.0 -. 1.0);;
```

```
- : float = 1.1805916207174113e+21
```

**Fact**  $F1 : H'_\omega(E0\_phi0\ 3)\ 3 = f\ (f\ N).$

**Proof.**

`rewrite F0; reflexivity.`

**Qed.**

**Fact**  $F1\_simpl :$

$H'_\omega(E0\_phi0\ 3)\ 3 =$   
 $(\text{exp2 } (\text{exp2 } (S\ N) * S\ N) * (\text{exp2 } (S\ N) * S\ N) - 1)\%nat.$

We leave as an exercise to determine the best approximation as possible of the size of this number (for instance its number of digits). For instance, if we do not take into account the multiplications in the formula above, we obtain that, in base 2, the number  $H'_{\omega^3}(3)$  has at least  $2^{10^{21}}$  digits. But it is still an under-approximation !

**Fact**  $F3 : (\text{exp2 } (\text{exp2 } N) <= H'_\omega(E0\_phi0\ 3 + 3)\ 0).$

### 6.3.2.5 A formula for $H'_{\omega^\omega}$

Now, we can get at last an exact formula for  $H'_{\omega^\omega}$ .

**Lemma**  $H'\_Phi0\_omega :$

`forall k, H'_\omega(E0\_phi0\ E0\_omega)\ k =`  
`iterate H'_succ_fun k (H'_\omega E0\_omega)\ k.`

Using extensionality of the functional `iterate`, we also get a closed formula.

**Lemma**  $H'\_Phi0\_omega\_exact\_formula\ k :$

$H'_\omega(E0\_phi0\ E0\_omega)\ k =$   
`let F f i := iterate f (S i) i`  
`in let g k := S (2 * k)%nat`  
`in iterate F k g k.`

Note that this formula contains two occurrences of the functional `iterate`, the second one is in fact a second-order iteration (on type `nat -> nat`) and the first one first-order (on type `nat`).

### 6.3.3 Abstract properties of $H'_\alpha$

Since pure computation seems to be useless for dealing with expressions of the form  $H'_\alpha(k)$ , even for small values of  $\alpha$  and  $k$ , we need to prove theorems for comparing  $H'_\alpha(k)$  and  $H'_\beta(l)$ , in terms of comparison between  $\alpha$  and  $\beta$  on the one hand,  $k$  and  $l$  on the other hand.

But beware of fake theorems! For instance, one could believe that  $H'$  is monotonous in its first argument. The following proof shows this is false.

**Remark** `H'_non_monol` :

```
~ (forall alpha beta k,
   (alpha <= beta)%e0 ->
   (H'_ alpha k <= H'_ beta k)%nat).
```

**Proof.**

```
intros H ;specialize (H 42 E0_omega 3).
(* ... *)
```

On the contrary, the functions of the  $H'$  hierarchy have the following five properties [KS81]: for any  $\alpha < \epsilon_0$ ,

- the function  $H'_\alpha$  is strictly monotonous : For all  $n, p \in \mathbb{N}, n < p \Rightarrow H'_\alpha(n) < H'_\alpha(p)$ .
- If  $\alpha \neq 0$ , then for every  $n, n < H'_\alpha(n)$ .
- The function  $H'_\alpha$  is pointwise less or equal than  $H'_{\alpha+1}$
- For any  $n \geq 1, H'_\alpha(n) < H'_{\alpha+1}(n)$ . We say that  $H'_{\alpha+1}$  dominates  $H'_\alpha$  from 1.
- For any  $n$  and  $\beta$ , if  $\alpha \xrightarrow[n]{} \beta$ , then  $H'_\beta(n) \leq H'_\alpha(n)$ .

In Coq, we follow the proof of [KS81]. This proof is mainly a single proof by transfinite induction on  $\alpha$  of the conjunction of the five properties. For each  $\alpha$ , the three cases :  $\alpha = 0$ ,  $\alpha$  is a limit, and  $\alpha$  is a successor are considered. Inside each case, the five sub-properties are proved sequentially, using the abstract properties defined in Sect 6.2.2 on page 122

**Section** `Proof_of_Abstract_Properties`.

```
Record P (alpha:E0) : Prop :=
mkP {
  PA : strict_mono (H'_ alpha);
  PB : alpha <> E0zero -> forall n, (n < H'_ alpha n)%nat;
  PC : H'_ alpha <= H'_ (E0_succ alpha);
  PD : dominates_from 1 (H'_ (E0_succ alpha)) (H'_ alpha);
  PE : forall beta n, Canon_plus n alpha beta ->
      (H'_ beta n <= H'_ alpha n)%nat}.
```

**Theorem** `P_alpha` : forall alpha, P alpha.

**Proof.**

```
intro alpha; apply well_founded_induction with E0lt.
```

```
alpha: E0
-----
well_founded E0lt
alpha: E0
-----
forall x : E0, (forall y : E0, y < x -> P y) -> P x
```

```
(* ... *)
```

**Qed.**

**End** `Proof_of_Abstract_Properties`.

Using a few lemmas *à la* Ketonen-Solovay, we prove that if  $\alpha < \beta$ , then  $H'_\beta$  eventually dominates  $H'_\alpha$ . We let the reader look at the proof (Section `Proof_of_H'_mono_l` of `Epsilon0.Hprime`).

**About** `H'_dom`.

```
H'_dom :
forall alpha beta : E0,
alpha o< beta -> H'_beta >>s H'_alpha

H'_dom is not universe polymorphic
Arguments H'_dom alpha beta H_alpha_beta
H'_dom is opaque
Expands to: Constant Hprime.H'_dom
```

### 6.3.4 Comparison between `L_` and `H'_`

By well-founded induction on  $\alpha$ , we prove that  $H'_\alpha(i) \leq L_\alpha(i+1)$  for any  $i$ .  
From Module `Epsilon0.L_alpha`

**Theorem** `H'_L_alpha` :

```
forall i: nat, (H'_alpha i <= L_alpha (S i))%nat.
```

#### 6.3.4.1 Back to hydras

The following theorem relates the length of (standard) battles with the the  $H'$  family of fast growing functions.

From Module `Hydra.Hydra_Theorems`

**Theorem** `battle_length_std_Hardy` (`alpha` : `E0`) :

```
alpha <> E0zero ->
forall k , 1 <= k ->
exists l: nat,
H'_alpha k - k <= l /\
battle_length standard k (iota (cnf alpha)) l.
```

**Proof.**

```
intros H k H0; exists (L_alpha (S k) - k); split.
- generalize (H'_L_alpha k); lia.
- now apply battle_length_std.
```

**Qed.**

Since  $H'_{\omega^3}(3) = H'_{\omega^3+3}(0)$ , the big number shown in Section 6.3.2.4 on page 132 is less or equal than the number of steps of the battle with the initial hydra of Figure 6.1, and 0 as initial replication factor.

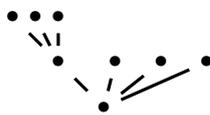


Figure 6.1: The hydra corresponding to the ordinal  $\omega^3 + 3$

## 6.4 A variant of the Wainer hierarchy (functions $F_\alpha$ )

Ketonen and Solovay introduce in [KS81] a “trivial” variant of the Wainer hierarchy [WB87, Wai70] of fast growing functions, indexed by ordinals below  $\epsilon_0$ . The functions  $F_\alpha$  are defined by the following equations.

- $F_0(i) = i + 1$
- $F_{\beta+1}(i) = (F_\beta)^{(i+1)}(i)$ , where  $f^{(i)}$  is the  $i$ -th iterate of  $f$ .
- $F_\alpha(i) = F_{\{\alpha\}}(i)$  if  $\alpha$  is a limit ordinal.

**Remark 6.6** The difference with the “classic” Wainer hierarchy  $f_\alpha$  ( $\alpha < \epsilon_0$ ) lies in the second equation:  $f_{\beta+1}(i) = (f_\beta)^{(i)}(i)$  and not  $f_{\beta+1}(i) = (f_\beta)^{(i+1)}(i)$ .

A module about the classic Wainer hierarchy is in preparation.

Note also that [KS81] defines also  $F_{\epsilon_0}$  (by the third equation). Since  $\epsilon_0$  is not representable in type  $\mathbf{E0}$ , our implementation in Coq does not take  $F_{\epsilon_0}$  into account.

A first attempt is to write a definition of  $F_\alpha$  by equations, in the same way as for  $H_\alpha$  (the functional iterate has already been used in Sect.2.3.3 on page 38).

```
Fixpoint iterate {A:Type}(f : A -> A) (n: nat)(x:A) :=
  match n with
  | 0 => x
  | S p => f (iterate f p x)
  end.
```

*FromEpsilon0.F\_alpha .*

*(\* Works with Dev*

*Fail Equations F\_ (alpha: E0) (i:nat) : nat by wf alpha E0lt :=*

*F\_ alpha i with E0\_eq\_dec alpha E0zero :=*

*{ | left\_zero => i ;*

*| right\_nonzero*

*with Utils.dec (E0limit alpha) :=*

*{ | left\_limit => F\_ (Canon alpha i) i ;*

*| right\_notlimit => iterate (F\_ (E0\_pred alpha)) (S i) i}}.*

*\*)*

We presume that this error comes from the recursive call of  $F_$  inside an application of `iterate`. The workaround we propose is to define first the iteration of  $F_$  as an helper  $F^*$ , then to define the function  $F$  as a “iterating  $F^*$  once”.

`Equations` accepts the following definition, relying on lexicographic ordering on pairs  $(\alpha, n)$ .

```
Definition call_lt (c c' : E0 * nat) :=
  lexico E0lt (Peano.lt) c c'.
```

```
Lemma call_lt_wf : well_founded call_lt.
```

```

  unfold call_lt; apply Inverse_Image.wf_inverse_image, wf_lexico.
- apply E0lt_wf.
- unfold Peano.lt; apply Nat.lt_wf_0.
Qed.

#[ global ] Instance WF : WellFounded call_lt := call_lt_wf.

(*  $F\_star$  ( $\alpha, i$ ) is intended to be the  $i$ -th iterate of  $F_\alpha$  *)

Equations  $F\_star$  (c: E0 * nat) (i:nat) : nat by wf c call_lt :=
   $F\_star$  ( $\alpha$ , 0) i := i;
   $F\_star$  ( $\alpha$ , 1) i
    with E0_eq_dec  $\alpha$  E0zero :=
    { | left _zero => S i ;
      | right _nonzero
        with Utils.dec (E0limit  $\alpha$ ) :=
        { | left _limit =>  $F\_star$  (Canon  $\alpha$  i,1) i ;
          | right _notlimit =>
             $F\_star$  (E0_pred  $\alpha$ , S i) i };
    }
   $F\_star$  ( $\alpha$ , (S (S n))) i :=
     $F\_star$  ( $\alpha$ , 1) ( $F\_star$  ( $\alpha$ , (S n)) i).

```

**Definition**  $F\_alpha$  i :=  $F\_star$  ( $\alpha$ , 1) i.

It is quite easy to prove that our functional  $F\_$  satisfies the equations on page 136.

**Lemma**  $F\_zero\_eqn$  : forall i,  $F\_ E0zero$  i = S i.

**Lemma**  $F\_lim\_eqn$  : forall  $\alpha$  i,  
 E0limit  $\alpha$  ->  
 $F\_ alpha$  i =  $F\_ (Canon \alpha i)$  i.

**Lemma**  $F\_succ\_eqn$  : forall  $\alpha$  i,  
 $F\_ (E0\_succ \alpha)$  i = iterate ( $F\_ alpha$ ) (S i) i.

As for the Hardy functions, we can use these equalities as rewrite rules for “computing” some values of  $F_\alpha(i)$ , for small values of  $\alpha$ .

**Lemma**  $LF1$  : forall i,  $F\_ 1$  i = S (2 \* i).

**Lemma**  $LF2$  : forall i,  $\exp2$  i \* i <  $F\_ 2$  i.

Like in Sect 6.3.3, we prove by induction the following properties (see [KS81]).

**Theorem**  $F\_alpha\_mono$   $\alpha$  : strict\_mono ( $F\_ alpha$ ).

**Theorem**  $F\_alpha\_gt$   $\alpha$  : forall n, n <  $F\_ alpha$  n.

**Corollary**  $F\_alpha\_positive$   $\alpha$  : forall n, 0 <  $F\_ alpha$  n.

**Theorem F\_alpha\_dom alpha :**  
 dominates\_from 1 (F\_ (E0\_succ alpha)) (F\_ alpha).

**Theorem F\_restricted\_mono\_l alpha :**  
 forall beta n, Canon\_plus n alpha beta ->  
 F\_ beta n <= F\_ alpha n.

As a corollary, we prove that, if  $\beta < \alpha$ , then  $F_\alpha$  dominates  $F_\beta$  (p. 284 of [KS81]).

**Section F\_monotony\_l.**

**Variables alpha beta :** E0.  
**Hypothesis H'\_beta\_alpha :** E0lt beta alpha.

**Lemma F\_mono\_l:** dominates (F\_ alpha) (F\_ beta).

**End F\_monotony\_l.**

**Exercise 6.2** Prove the following property:

Lemma LF3 : dominates\_from 2 (F\_ 3) (fun n => iterate exp2 n n).

*You may start this exercise with the file exercises/ordinals/F\_3.v.*

**Exercise 6.3** Prove that, for any  $\alpha \geq 3$  and  $n \geq 2$ ,  $F_\alpha(1 + n) \geq 2^{F_\alpha(n)}$ .

*You may start this exercise with the file exercises/ordinals/F\_3.v.*

**Exercise 6.4** It is tempting to prove a simple property of monotony of the function  $F_-$ .

Let  $\alpha \leq \beta < \epsilon_0$ . For any  $n \geq 2$ ,  $F_\alpha(n) \leq F_\beta(n)$ .

Prove or disprove this statement.

*You may start this exercise with the file exercises/ordinals/is\_F\_monotonous.v.*

**Exercise 6.5** Prove that for any  $n \geq 2$ ,  $\text{Ack } n n \leq F_\omega(n)$ , where Ack is the Ackermann function. Next, prove that  $F_\alpha$  is not primitive recursive, for any  $\alpha \geq \omega$  (please see Sect. 11.7 on page 222). On the other hand, please show that for any natural number  $n$ , the function  $F_n$  is primitive recursive. Thus  $F_\alpha$  is primitive recursive if and only if  $\alpha$  is finite.

*You may start this exercise with the file exercises/ordinals/F\_Omega.v. Properties of the Ackermann function are studied in theories/ordinals/MoreAck/Ack.v and theories/ordinals/MoreAck/AckNotPR.v .*



Library `gaia_hydras.GF_alpha` defines the functions  $F_\alpha$  for Gaia's world.

## 6.5 More about rapidly growing functions

In Sect. 11.8 on page 228, we prove that the length of hydra-battles (for a given hydra, according to the initial replication factor) is not primitive recursive in general. This proof uses properties of the Ackermann function, and the  $H'_\alpha$ ,  $F_\alpha$ ,  $L_\alpha$  families of functions.



# Chapter 7

## Gaia and the hydra (draft)

### 7.1 Introduction

The Gaia project [GQS] by José Grimm aimed to formalize mathematics in Coq in the style of Nicolas Bourbaki. The formalization of the first book in the Elements of Mathematics series by Bourbaki, on the theory of sets, was initially described in a technical report in July 2009 [Gri09a]. The set-theoretic axioms and basic definitions in Gaia were derived from an earlier development by Carlos Simpson [Sim04b, Sim04a]. Grimm then wrote (and continually updated) technical reports describing the formalization of Bourbaki’s two subsequent books [Gri09b, Gri16] and additional topics in number theory [Gri13, Gri14], before he passed away in 2019.

In 2020, members of Coq-community transferred the Gaia source code to GitHub and adapted it for recent releases of the Mathematical Components library, which Gaia heavily relies on. Anonymous volunteers (“collaborators of Nicolas Bourbaki”) then finished the only in-progress proof left by Grimm. At around 155,000 LOC, Gaia is currently one of the largest maintained open source Coq projects.

Gaia contains definitions of ordinals in Cantor and Veblen normal form [Gri13], adapted from the historical Cantor contribution [CC06]. The data types for ordinals are essentially defined the same way as in Hydras & Co., but they are not identical inside Coq, e.g., due to residing in different modules. There are also minor differences in how ordinal arithmetic is implemented, due to the different evolutionary paths taken since divergence from the ancestor library.

The main objective of the `theories/gaia/*.v` files is to provide Gaia’s users definitions and lemmas on some combinatorial aspects of ordinal numbers less than  $\epsilon_0$  [KS81], already proven in Hydras & Co.: canonical sequences, accessibility properties, large sets and rapidly growing arithmetical functions. We may also complete Hydras & Co. with some lemmas proven in Gaia.

For these purposes, we write modules dedicated to the importation of definitions and lemmas from each of both libraries into the other one. Please note that by “import” we do not mean “duplicate” nor “repairing” Hydra-battles’ proofs. Whenever possible, we proceed by rewriting steps over the statements and not the proofs. In a future version, it would be nice to have our proofs in the same context, after porting Hydra-battles’ proofs under Gaia’s definition,

following proof re-use or repair techniques [BP01, Mag03, RPY<sup>+</sup>21].

This initial draft bridge code mostly uses the SSReflect proof language and idioms from the Mathematical Components library. We made this design decision since we believe it is less challenging to reason about Hydra-battles code using SSReflect and MathComp than to reason about Gaia without SSReflect<sup>1</sup>.

## 7.2 Library structure

The Gaia-hydras library is designed for users of Gaia and/or Hydra-battles. Whenever possible, we managed to respect the notations and naming conventions of both libraries.

The modules are named `theories/gaia/*.v` and coqdocumented in `theories/html/gaia_hydras.*.html`. The main entry point is `gaia_hydras.T1Bridge`, which requires both Gaia’s `ssete9` and a few modules of `theories/ordinals/Epsilon0`. Other modules are dedicated to the adaptation of several notions developed in Hydra-battles to MathComp’s and Gaia’s vocabulary and structures. By default, we give priority to Gaia’s notation and vocabulary.

### 7.2.1 The T1Bridge Module

`T1Bridge` starts with a few requirements.

```
From mathcomp Require Import all_ssreflect zify.
```

```
From Coq Require Import Logic.Eqdep_dec.
From hydras Require Import DecPreOrder ON_Generic T1 E0.
From gaia Require Export ssete9.
```

Because of the common origin of Hydra-battles and Gaia, both libraries share the common name `T1` (data type of ordinal terms below  $\epsilon_0$ ). As mentioned before, the bare name `T1` is given by priority to Gaia. Hydra-battles’ type `T1` may be called `hT1` (h for “hydras”) or simply `T1.T1`, `Epsilon0.T1.T1`, etc.

```
#[global] Notation hT1 := T1.T1.
#[global] Notation T1 := ssete9.CantorOrdinal.T1.
```

Likewise, many constants are common to Hydra-battles and Gaia. For instance, the name `zero` may refer to Gaia’s `gaia.ssete9.CantorOrdinal.zero` and Hydra-battles’ `hydras.Epsilon0.T1.zero`. By default, the former will be simply called `zero` and the latter `T1.zero`.

The following notations are not defined in Gaia; we add them to `T1Bridge` for simplicity’s sake.

```
(** Restrictions to terms in normal form *)

#[global] Notation LT := (restrict T1nf T1lt).
#[global] Notation LE := (restrict T1nf T1le).
```

---

<sup>1</sup>The author of these proof scripts is still a beginner in Ssreflect. Please forgive the clumsiness of the current proof scripts.

**Remark 7.1** Please keep in mind that, in this module, we favour Gaia's notations over hydra's. For instance, the mathematical inequality  $42 < \omega$  is expressed in Gaia's terms as follows:

**Check** `T1.lt (\F 42) T1.omega.`

**Check** `(\F 42 < T1.omega)%ca.`

**Check** `\F 42 < T1.omega. (* within cantor_scope *)`

The same inequality, with Hydra-battles vocabulary:

**Check** `T1.lt (T1.T1nat 42) T1.T1omega.`

**Check** `T1.lt (\F 42)%t1 T1.T1omega.`

## 7.2.2 Translation functions and data refinement

The bridge between both libraries is made of two straightforward bijections.

**Fixpoint** `h2g (a : hT1) : T1 :=`  
`if a is T1.cons a n b then cons (h2g a) n (h2g b) else zero.`

**Fixpoint** `g2h (a : T1) : hT1 :=`  
`if a is cons a n b then T1.cons (g2h a) n (g2h b) else T1.zero.`

**Example**  `$\alpha$  : T1 :=`  
`T1.omega + phi0 T1.omega * \F 3 + phi0 (\F 5) * \F 4 + T1.omega * T1.omega.`

**Example**  `$\beta$  : T1 := phi0 (phi0 (\F 2)).`

**Compute** `g2h  $\alpha$ .`

```
= T1.cons T1.T1omega 2
  (T1.cons (FS 4) 3 (T1.phi0 (FS 1)))
: hT1
```

**Compute**  `$\alpha$  == h2g (g2h  $\alpha$ ).`

```
= true
: bool
```

The following cancel lemmas will be often applied in order to simplify sub-terms of the form  $(g2h (h2g t))$  and  $(h2g (g2h t))$ , which appear in many proofs by rewriting.

**Lemma** `h2g_g2hK : cancel g2h h2g.`

**Proof.** `elim => // => a1 IH1 n t2 IH2 /:=; by rewrite IH1 IH2. Qed.`

**Lemma** `g2h_h2gK : cancel h2g g2h.`

`(* ... *)`

**Lemma** `h2g_eqE (a b : hT1): h2g a = h2g b <-> a = b.`

**Lemma** `g2h_eqE (a b : T1): g2h a = g2h b <-> a = b.`

### 7.2.2.1 Pretty printing Cantor normal forms

The following function provides us with a more readable printing of Cantor normal forms (please see Sect. 4.1.5 on page 77).

**Definition** `Tlpp` (`a:T1`) : `ppT1 := pp (g2h a)`.

**Compute** `β + α`.

```
= cons (cons (cons zero 1 zero) 0 zero) 0
      (cons (cons (cons zero 0 zero) 0 zero) 2
            (cons (cons zero 4 zero) 3
                  (cons (cons zero 1 zero) 0 zero)))
: T1
```

**Compute** `Tlpp (β + α)`.

```
= (ω ^ ω ^ 2 + ω ^ ω * 3 + ω ^ 5 * 4 + ω ^ 2)%pT1
: ppT1
```

### 7.2.2.2 Refinements: Definitions

Functions `h2g` and `g2h` allow us to define a notion of “data-refinement” for constants, functions, predicates and relations. The following definitions express that a given constant, function, relation defined in Hydra-battles “implements” the same concept of Gaia.

From `gaia_hydras.T1Bridge`.

**Definition** `refines0` (`x:hT1`)(`y:T1`) := `y = h2g x`.

**Definition** `refines1` (`f:hT1 -> hT1`) (`f': T1 -> T1`) :=  
`forall x: hT1, f' (h2g x) = h2g (f x)`.

**Definition** `refines2` (`f:hT1 -> hT1 -> hT1`) (`f': T1 -> T1 -> T1`) :=  
`forall x y : hT1, f' (h2g x) (h2g y) = h2g (f x y)`.

**Definition** `refinesPred` (`hP: hT1 -> Prop`) (`gP: T1 -> Prop`) :=  
`forall x : hT1, hP x <-> gP (h2g x)`.

**Definition** `refinesRel` (`hR: hT1 -> hT1 -> Prop`)  
(`gR: T1 -> T1 -> Prop`) :=  
`forall x y : hT1, hR x y <-> gR (h2g x) (h2g y)`.

Refinement lemmas can be easily “reversed”.

**Lemma** `refines1_R f f'` :  
`refines1 f f' <-> forall y: T1, f (g2h y) = g2h (f' y)`.

**Lemma** `refines2_R f f'` :  
`refines2 f f' <-> forall y z: T1, f (g2h y) (g2h z) = g2h (f' y z)`.

### 7.2.3 Examples of refinement

Both of our libraries define constants like 0, 1,  $\omega$ , and arithmetic functions: successor, addition, multiplication, and exponential of base  $\omega$  (function  $\phi_0$ ). We prove that these definitions are mutually consistent. Finally, we prove that the boolean predicates “to be in normal form” are equivalent.

#### 7.2.3.1 A few constants

For each constant: 0, 1, ...,  $n$  and  $\omega$ , we prove that hydras’ definition refines gaia’s.

```
Lemma zero_ref : refines0 T1.zero zero.
Proof. done. Qed.
```

```
Lemma one_ref : refines0 T1.one one.
Proof. done. Qed.
```

```
Lemma Finite_ref (n:nat) : refines0 (T1.T1nat n) (\F n).
Proof. by case: n. Qed.
```

```
Lemma omega_ref : refines0 T1.T1omega T1omega.
Proof. done. Qed.
```

#### 7.2.3.2 Unary functions

```
Lemma succ_ref : refines1 T1.succ T1succ.
Lemma phi0_ref x : refines0 (T1.phi0 x) (phi0 (h2g x)).
```

#### 7.2.3.3 Order and comparison

The strict orders on types T1 and hT1 are compatible. Note that Hydra-battles’ comparison on ordinal terms uses a Stdpp-like trichotomy, hence the first two lemmas below:

```
Lemma compare_ref (x y : hT1) :
  match T1.compare_T1 x y with
  | Datatypes.Lt => T1lt (h2g x) (h2g y)
  | Datatypes.Eq => h2g x = h2g y
  | Datatypes.Gt => T1lt (h2g y) (h2g x)
  end.

Lemma decide_hltE (a b : hT1):
  bool_decide (T1.lt a b) = (h2g a < h2g b).
```

```
Lemma lt_ref : refinesRel T1.lt T1lt.
```

```
Lemma le_ref : refinesRel T1.le T1le.
```

#### 7.2.3.4 More rewriting lemmas

```
Lemma T1lt_iff a b: T1nf a -> T1nf b ->
  a < b <-> g2h a t1< g2h b.
```

**Lemma T1le\_iff** (a b: T1):  
 a <= b <-> T1.le (g2h a) (g2h b).

**Lemma hnf\_g2h a** : T1.nf (g2h a) = T1nf a.  
**Proof.** by rewrite /T1.nf (nf\_ref (g2h a)) h2g\_g2hK. **Qed.**

**Lemma g2h\_succ a** : g2h (T1succ a) = T1.succ (g2h a).  
**Proof.** by rewrite -(h2g\_g2hK a) succ\_ref !g2h\_h2gK. **Qed.**

**Lemma hlt\_iff a b**: T1.lt a b <-> h2g a < h2g b.  
**Proof.** by rewrite lt\_ref. **Qed.**

**Lemma T1limit\_ref** (a:Epsilon0.T1.T1) : T1.T1limit a = T1limit (h2g a).

**Lemma T1is\_succ\_ref** (a:Epsilon0.T1.T1): T1.T1is\_succ a = T1is\_succ (h2g a).

### 7.2.3.5 Addition and multiplication

The definition of the binary operations  $+$  and  $\times$  requires the comparison of ordinal terms, which is not implemented in Gaia-hydras the same way as in Hydra-battles. Thus, the proof of the following lemmas applies compatibility lemmas like `compare_ref` (see Section 7.2.3.3 on the preceding page).

**Lemma plus\_ref** : refines2 T1.T1add T1add.  
**Lemma mult\_ref** : refines2 T1.T1mul T1mul.

### 7.2.3.6 Well formed ordinal terms (Cantor normal form)

Both definitions of “being in Cantor normal form” are compatible.

**Lemma nf\_ref** (a: hT1) : T1.nf\_b a = T1nf (h2g a).

## 7.2.4 Looking for a lemma

Coq’s command `Search` and notation scopes allow you to explore both libraries. For instance, let us look for lemmas whose conclusion is  $\alpha \times \beta = \beta$ .

The following command lists us ‘gaia’s lemmas (thanks to Gaia’s `cantor_scope`).

**Search** ( \_ \* ?beta = ?beta)%ca.

```
T1mul_a0E: forall c : T1, c * zero = zero
T1mul1n: forall x : T1, one * x = x
T1muln0: forall x : T1, x * zero = zero
mul_fin_omega:
  forall n : nat, \F n.+1 * T1omega = T1omega
mul_int_limit:
  forall (n : nat) [y : T1],
    T1limit y -> \F n.+1 * y = y
exp_F0:
  forall (z : T1) (n : nat) [v : T1_eqType],
    v != zero -> exp_F z n * exp_0 z v = exp_0 z v
```

Within `t1_scope`:

**Search** ( `_ * ?beta = ?beta` )%t1.

```

mult_a_0: forall a : hT1, (a * T1.zero)%t1 = T1.zero
mult_1_a: forall [a : hT1], nf a -> (\F 1 * a)%t1 = a
mult_fin_omega:
  forall n : nat, (FS n * T1.Tlomega)%t1 = T1.Tlomega
L7:
  forall (n : nat) [c : hT1] (p : nat),
  c <> T1.zero ->
  (FS n * T1.cons c p T1.zero)%t1 =
  T1.cons c p T1.zero
Ex3: (\F 5 * T1.Tlomega)%t1 = T1.Tlomega

```

## 7.3 Importing Definitions and theorems from Hydra-battles

Some constructions of Hydra-battles were (to our knowledge) not implemented in Gaia yet, for instance, commutative (“Hessenberg”) addition, canonical sequences and Ketonen-Solovay machinery (Chapters 4, 5 and 6). In order not to copy long proofs, we chose to *derive* several constructions from Hydra-battles into Gaia’s world, and rewrite Hydra-battles statements in order to make them Gaia-compatible.

### 7.3.0.1 Remark

We try to respect the following convention: Let us consider some module `hydras.Epsilon0.Foo`. Its adaptation to Gaia will be a new module called `gaia_hydras.GFoo` (“G” for Gaia).

Let `bar` be a symbol defined in `Foo`. Inside `GFoo`, the identifier `bar` is reserved in priority to Gaia’s adaptation. Hydra-battles’ definition is still available through qualified names, like `Epsilon0.Foo.bar`, `Foo.bar`, or even a short notation like `hbar` (“h” for Hydra-battles).

### 7.3.1 Hessenberg sum

Natural sum (a.k.a. Hessenberg sum) is defined in `hydras.Epsilon0.Hessenberg` and is used for proving termination of all hydra battles (See Sect. 4.4.1).

Instead of defining this operation *ex nihilo* in `theories/gaia`, we chose to *import* the definition from `hydras.Epsilon0.Hessenberg` as follows:

```
#[local] Notation hoplus := Epsilon0.Hessenberg.oplus.
```

```
Definition oplus alpha beta := h2g (hoplus (g2h alpha) (g2h beta)).
```

```
Infix "o+" := oplus: cantor_scope.
```

```
Fixpoint o_finite_mult n alpha :=
```

```

if n is p.+1 then alpha o+ (o_finite_mult p alpha)
else zero.

```

**Compute** Tlpp (Tlomega o+ Tlomega).

```

= (ω * 2)%pT1
: ppT1

```

**Compute** Tlpp (o\_finite\_mult 5 (Tlomega + \F 1)).

```

= (ω * 5 + 5)%pT1
: ppT1

```

**Exercise 7.1** We could also have defined `oplus` by a structurally recursive definition, as in 4.4.1.1.1 on page 96.

```

Fixpoint oplus (alpha beta : T1) : T1 :=
  let fix oplus_aux beta {struct beta} :=
    match alpha, beta with
    | zero, _ => beta
    | _, zero => alpha
    | cons a1 n1 b1, cons a2 n2 b2 =>
      match compare a1 a2 with
      | Gt => cons a1 n1 (oplus b1 beta)
      | Lt => cons a2 n2 (oplus_aux b2)
      | Eq => cons a1 (S (n1 + n2)%nat) (oplus b1 b2)
      end
    end
  in oplus_aux beta.

```

Prove that `Hessenberg.oplus` refines the new function.

### 7.3.1.1 A few lemmas

Many properties of Hessenberg's sum can be obtained from `hydras.Epsilon0.Hessenberg` just by sequences of rewritings.

```

Lemma oplusE (a b :T1) :
  a o+ b =
  match a, b with
  | zero, _ => b
  | _, zero => a
  | cons a1 n1 b1, cons a2 n2 b2 =>
    match compare a1 a2 with
    | Gt => cons a1 n1 (b1 o+ b)
    | Eq => cons a1 (S (n1 + n2)) (b1 o+ b2)
    | Lt => cons a2 n2 (a o+ b2)
    end
  end.

```

**Proof.**

```

rewrite /oplus oplus_eqn; case: a.
cbn ; by rewrite h2g_g2hK.
case: b. move => ? ? ? ;by rewrite !h2g_g2hK.
move => t n t0 t1 n0 t2; rewrite !g2h_cons compare_g2h.
case (compare t1 t); by rewrite h2g_cons !h2g_g2hK.
Qed.

```

**Lemma oplus0b:** left\_id zero oplus.

**Proof.** rewrite /oplus; case => // /= ? ? ?; by rewrite !h2g\_g2hK. Qed.

**Lemma oplusa0:** right\_id zero oplus.

**Lemma oplus\_nf** (a b : T1) : T1nf a -> T1nf b -> T1nf (a o+ b).

Hessenberg sum is associative, commutative and strictly monotonous (on ordinal terms in normal form).

**Lemma oplusC** (a b : T1): T1nf a -> T1nf b -> a o+ b = b o+ a.

**Proof.** move => ? ?; by rewrite /oplus oplus\_comm ?hnf\_g2h. Qed.

**Lemma oplusA** (a b c : T1) :

T1nf a -> T1nf b -> T1nf c -> a o+ (b o+ c) = a o+ b o+ c.

**Proof.**

move => ? ? ?; by rewrite /oplus !g2h\_h2gK oplus\_assoc ?hnf\_g2h.

Qed.

**Lemma oplus\_lt1** (a b:T1):

T1nf a -> T1nf b -> zero < a -> b < b o+ a.

**Lemma oplus\_lt2** (a b : T1):

T1nf a -> T1nf b -> zero < b -> a < b o+ a.

**Lemma oplus\_strict\_mono\_l** (a b c : T1):

T1nf a -> T1nf b -> T1nf c -> a < b -> a o+ c < b o+ c.

**Lemma oplus\_strict\_mono\_r** (a b c : T1):

T1nf a -> T1nf b -> T1nf c -> b < c -> a o+ b < a o+ c.

**Lemma oplus\_lt\_phi0** (a b c : T1):

T1nf a -> T1nf b -> T1nf c ->  
a < c -> b < c -> phi0 a o+ phi0 b < phi0 c.

### 7.3.2 Canonical sequences

Canonical sequences are described in Chapter 5 on page 101, and implemented in `hydras.Epsilon0.Canon`. Module `gaia_hydras.GCanon`, imports definitions and results from that library.

Like for Hessenberg sum, we define a function `canon` *via* the translations `g2h` and `h2g`.

**[global] Notation** `hcanon` := `Epsilon0.Canon.canon`.

**Definition** `canon` (a : T1) (i:nat) : T1 := `h2g (hcanon (g2h a) i)`.

**Lemma** `g2h_canon a i`: `g2h (canon a i) = hcanon (g2h a) i`.

**Proof.** by `rewrite /canon g2h_h2gK`. **Qed.**

The following lemmas are proved by rewriting from corresponding statements proved in Hydra-battles. The first six lemmas correspond to rewriting rules derived from the body of the definition of `Epsilon0.Canon.canon`.

**Lemma** `gcanon_zero i`: `canon zero i = zero`.

**Proof.** `rewrite /canon => //`. **Qed.**

**Lemma** `canon_succ i a` (`Ha`: `T1nf a`):  
`canon (T1succ a) i = a`.

**Lemma** `canon_SSn_zero (i : nat) (a : T1) (n : nat)`:  
`T1nf a ->`  
`canon (cons a n.+1 zero) i = cons a n (canon (phi0 a) i)`.

**Lemma** `canon_lim1 i (lambda: T1) :`  
`T1nf lambda ->`  
`T1limit lambda -> canon (phi0 lambda) i = phi0 (canon lambda i)`.

**Lemma** `canon_lim2 i n (lambda : T1) (Hnf: T1nf lambda) (Hlim: T1limit lambda)`:  
`canon (cons lambda n.+1 zero) i = cons lambda n (phi0 (canon lambda i))`.

**Lemma** `canon_lim3 i n a lambda (Ha: T1nf a)`  
`(Hlambda: T1nf lambda) (Hlim :T1limit lambda) :`  
`canon (cons a n lambda) i = cons a n (canon lambda i)`.

### 7.3.2.1 Canonical sequences and the order `T1lt`

The following lemmas are also borrowed from `hydras.Epsilon0.Canon`.

**Lemma** `canon_lt (i : nat) [a : T1]`: `T1nf a -> a <> zero -> canon a i < a`.

**Lemma** `canon_limit_mono lambda i j (Hnf : T1nf lambda)`  
`(Hlim : T1limit lambda) (Hij : (i < j)%N) :`  
`canon lambda i < canon lambda j`.

**Lemma** `canon_limit_strong lambda :`  
`T1nf lambda -> T1limit lambda ->`  
`forall b, T1nf b ->`  
`T1lt b lambda -> {i : nat | b < canon lambda i}`.

Let us recall Gaia's definition of  $\omega$ -limit.

**Definition** `limit_v2 (f: Tf) x :=`  
`(forall n, f n < x) /\ (forall y, T1nf y -> y < x -> (exists n, y <= f n))`.

**Definition** `limit_of (f: Tf) x :=`  
`[/\ (forall n m, (n < m)%N -> f n < f m), limit_v2 f x & T1nf x]`.

We can prove the following statements, in Gaia's style.

**Lemma** `gcanon_limit_v2 (lambda: T1)`:  
`T1nf lambda -> T1limit lambda -> limit_v2 (canon lambda) lambda`.

**Lemma** `canon_limit_of lambda (Hnf : T1nf lambda) (Hlim : T1limit lambda) :`  
`limit_of (canon lambda) lambda`.

### 7.3.3 Accessibility in $\epsilon_0$

The library `gaia_hydras.GPaths` imports definitions and lemmas from `hydras.Epsilon0.Paths` (described in Sect. 5.3).

#### 7.3.3.1 Transitions and paths

Let us consider a kind of transition system, the states of which are ordinals below  $\epsilon_0$ . The elementary transitions (“small steps”) are jumps from an ordinal  $\alpha$  to  $\{\alpha\}(i)$ , for some  $i \neq 0$ .

```
#[global] Notation htransition := Epsilon0.Paths.transition.
#[global] Notation hbounded_transitionS := Paths.bounded_transitionS.
```

```
Definition transition i (a b: T1) :=
  [/\ i != 0 , a!= zero & b == canon a i].
```

```
Definition bounded_transitionS n (a b: T1) :=
  exists i, (i <= n)%N /\ transition (S i) a b.
```

Paths (sequences of transitions) are defined by delegation to `hydras.Epsilon0.Paths`.

```
(** [path_to b s alpha] : [b] is accessible from [alpha]with trace [s] *)
```

```
Definition path_to (to: T1)(s: seq nat) (from:T1) : Prop :=
  hpath_to (g2h to) s (g2h from).
```

```
Notation path from s to := (path_to to s from).
```

```
Definition acc_from a b := exists s, path a s b.
```

Paths with constant index and paths whose index is incremented by 1 at each step play an important role in the so-called Ketonen-Solovay machinery [KS81].

```
Definition const_path i a b := hconst_path i (g2h a) (g2h b).
```

```
Definition standard_path i a j b :=
  Paths.standard_path i (g2h a) j (g2h b).
```

The following examples are proved using the `Epsilon0.Paths.path_tac` tactic.

```
Example ex_path1 : path (Tlomega * (\F 2)) [:: 2; 2; 2] Tlomega.
```

```
Proof. rewrite /path_to; Epsilon0.Paths.path_tac. Qed.
```

```
Example ex_path2: path (Tlomega * \F 2) [:: 3; 4; 5; 6] Tlomega.
```

```
Proof. rewrite /path_to; path_tac. Qed.
```

```
Example ex_path3: path (Tlomega * \F 2) (index_iota 3 15) zero.
```

```
Proof. rewrite /path_to /index_iota => /=; path_tac. Qed.
```

```
Example ex_path4: path (Tlomega * \F 2) (List.repeat 3 8) zero.
```

```
Proof. rewrite /path_to => /=; path_tac. Qed.
```

### 7.3.3.2 Main theorems about paths

$\beta$  is (strictly) accessible from  $\alpha$ , if and only iff  $\beta < \alpha$ .

**Lemma** `path_to_LT b s a`:

```
path_to b s a -> T1nf a -> T1nf b -> b < a.
```

**Lemma** `LT_path_to (a b : T1)` :

```
T1nf a -> T1nf b -> b < a -> {s : list nat | path_to b s a}.
```

Constant index paths can be simulated with constant paths with larger index.

**Lemma** `Cor12 (a: T1) : T1nf a ->`

```
forall b i n, T1nf b -> b < a -> (i < n)%N ->
  const_path i.+1 a b -> const_path n.+1 a b.
```

If  $\beta < \alpha$ , then  $\beta$  is accessible from  $\alpha$  through some constant-index path.

**Lemma** `Lemma2_6_1 (a:T1) :`

```
T1nf a -> forall b, T1nf b -> b < a -> {n:nat | const_path n.+1 a b}.
```

Any constant-index path can be simulated by a “standard” path (with index incremented by 1 at each step).

**Lemma** `constant_to_standard_path (a b : T1) (i : nat):`

```
T1nf a -> const_path i.+1 a b -> zero < a ->
  {j:nat | standard_path i.+1 a j b}.
```

As a corollary, we relate ordinal inequality to standard paths.

**Theorem** `LT_to_standard_path (a b : T1) :`

```
T1nf a -> T1nf b -> b < a ->
  {n : nat & {j:nat | standard_path n.+1 a j b}}.
```

**Proof.**

## 7.3.4 A type for well-formed ordinal terms

For compatibility’s sake, we add a clone of `hydra-battles`’ type `E0` defined in `Epsilon0.E0`. This type is mainly used in the definition with `Equations` of rapidly growing functions indexed by ordinals (see Sect. 7.4 on page 155).

**Record** `E0 := mkE0 { cnf : T1 ; _ : T1nf cnf == true}.`

**Coercion** `cnf: E0 -> T1.`

**Definition** `E0_eq_mixin : Equality.mixin_of E0.`

**Definition** `E0_eqtype := Equality.Pack E0_eq_mixin.`

**Canonical Structure** `E0_eqtype.`

## 7.3.4.1 Definitions

Many functions on type T1 are “lifted” to E0.

**Definition** ppE0 (a: E0) := T1pp (cnf a).

**Definition** E0lt (a b: E0) := cnf a < cnf b.

**Definition** E0le (a b: E0) := cnf a <= cnf b.

#[global, program] **Definition** E0zero: E0 := @mkE0 zero \_.

#[global, program]

**Definition** E0\_succ (a: E0): E0 := @mkE0 (T1succ (cnf a)) \_.

**Next Obligation.**

rewrite nf\_succ => //; case: a => ? i //:=; by apply /eqP.

**Defined.**

#[global, program]

**Definition** E0\_pred (a:E0) : E0:= @mkE0 (T1pred (cnf a)) \_.

**Next Obligation.**

case: a => ? ?; rewrite nf\_pred => //:= ; by apply /eqP.

**Defined.**

**Fixpoint** E0fin (n:nat) : E0 :=

if n is p.+1 then E0\_succ (E0fin p) else E0zero.

#[program] **Definition** E0\_omega: E0 := @mkE0 T1omega \_.

#[program] **Definition** E0\_phi0 (a: E0) : E0 := @mkE0 (phi0 (cnf a)) \_.

#[program] **Definition** E0plus (a beta: E0) : E0 :=

@mkE0 (T1add (cnf a) (cnf beta)) \_.

**Next Obligation.**

rewrite nf\_add => //.

case :a; cbn => t Ht; apply /eqP => //.

case :beta; cbn => t Ht; apply /eqP => //.

**Defined.**

#[program] **Definition** E0mul (a beta: E0) : E0 :=

@mkE0 (T1mul (cnf a) (cnf beta)) \_.

(\* ... \*)

Canonical sequences are lifted to type E0 in gaia\_hydras.GCanon.

#[program] **Definition** E0Canon (a: E0) (i: nat): E0 :=

@mkE0 (canon (cnf a) i) \_.

**Lemma** E0\_canon\_lt (a: E0) i:

cnf a <> zero -> E0lt (E0Canon a i) a.

(\* ... \*)

In order to import definitions and lemmas from `Epsilon0.E0`, we define a pair of translations.

```
#[program] Definition E0_h2g (a: hE0): E0 := @mkE0 (h2g (E0.cnf a)) _.
Next Obligation.
  rewrite -nf_ref; case: a => /= cnf cnf_ok; by rewrite cnf_ok.
Defined.
```

```
#[program] Definition E0_g2h (a: E0): hE0 := @E0.mkord (g2h (cnf a)) _.
Next Obligation.
  case: a => /= cnf0 /eqP; by rewrite hnf_g2h.
Defined.
```

#### 7.3.4.2 Main lemmas about `E0`

These tools allow us, for instance, to import `E0.lt`'s well-foundedness.

**Lemma** `gE0lt_wf` : `well_founded E0lt`.

**Proof.**

```
move => ?; apply Acc_incl with (fun x y => hE0lt (E0_g2h x) (E0_g2h y)).
(* ... *)
```

#### 7.3.4.3 `E0` as an ordinal notation

The notion of ordinal notation is defined in Chapter 3. In `gaia_hydras.T1Bridge`, we define an instance of class `ON E0lt compare`.

First, we define an instance of `(Compare T1)`:

```
#[global] Instance T1compare : Compare T1 :=
  fun a beta => compare (g2h a) (g2h beta).
```

**Compute** `compare (\F 6 + T1omega) T1omega`.

```
= Eq
: comparison
```

```
Lemma T1compare_correct (a b: T1):
  CompSpec eq T1lt a b (compare a b).
```

Then, we build an instance of `(ON E0lt compare)`.

```
#[global] Instance E0compare: Compare E0 :=
  fun (alpha beta: E0) => T1compare (cnf alpha) (cnf beta).
```

```
Lemma E0compare_correct (alpha beta : E0) :
  CompSpec eq E0lt alpha beta (compare alpha beta).
(* ... *)
```

```
#[global] Instance E0_sto : StrictOrder E0lt.
```

```
#[global] Instance E0_comp : Comparable E0lt compare.
```

**Proof.** `split; [apply E0_sto | apply E0compare_correct]. Qed.`

```

Compute compare (E0_phi0 (E0fin 2)) (E0mul (E0_succ E0_omega) E0_omega).
= Eq
: comparison

```

```

#[global] Instance Epsilon0 : ON E0lt compare.
Proof. split; [apply: E0_comp | apply: gE0lt_wf]. Qed.

```

## 7.4 Rapidly growing arithmetic functions

In this section, we consider a few families of arithmetic functions, indexed by ordinal numbers below  $\epsilon_0$ . The type of such families is  $E0 \rightarrow \text{nat} \rightarrow \text{nat}$  where  $E0$  is the type of ordinal terms in Cantor normal form, strictly below  $\epsilon_0$ . In order to compare such functions, we define a few abstract properties in `gaia_hydras.T1Bridge` (adapted from `hydras.Prelude.Iterates`).

```

Definition strict_mono (f: nat -> nat) :=
  forall n p, (n < p)%N -> (f n < f p)%N.

```

```

Definition dominates_from (n : nat) (g f : nat -> nat) :=
  forall p : nat, (n <= p)%N -> (f p < g p)%N.

```

```

Definition dominates g f := exists n : nat, dominates_from n g f .

```

```

Definition dominates_strong g f := {n : nat | dominates_from n g f}.

```

```

Definition fun_le f g := forall n:nat, (f n <= g n)%N.

```

In Hydra-battles, these functions are defined by transfinite induction, using the `coq-equations` plug-in [SM19]. For that purpose, we defined a sigma-type `hydras.Epsilon0.E0.E0` of ordinal-terms in Cantor normal form.

### 7.4.1 The $H'_\alpha$ family

This variant of the Hardy hierarchy is described in Sect. 6.3 on page 129, and adapted to Gaia in Module `gaia_hydras.GHprime`.

```

From mathcomp Require Import all_ssreflect zify.
From gaia Require Export ssete9.
From Coq Require Import Logic.Eqdep_dec.
From hydras Require Import DecPreOrder T1 E0.
From hydras Require Paths.
From hydras Require Import Iterates Hprime L_alpha.
From gaia_hydras Require Import T1Bridge GCanon GPaths.
Definition H'_alpha i := Hprime.H'_ (E0_g2h alpha) i.

```

In the rest of this section, we prove a few lemmas by rewriting from Hydra-battles version (`Epsilon0.Hprime`).

### 7.4.1.1 Equations for $H'$

In Hydra-battles',  $H'_\alpha(i)$  is defined by transfinite induction over  $\alpha$ , using co-equations. In the present module, equations for  $H'$  are just imported as lemmas.

**Lemma  $H'_{eq1}$**  ( $i$ : nat) :  $H'_{E0zero} i = i$ .

**Proof.** by rewrite / $H'_{g2h_{E0zero}}$  Epsilon0.Hprime.H'\_{eq1}. **Qed.**

**Lemma  $H'_{eq2}$**  alpha i :

$H'_{(E0_{succ} \alpha)} i = H'_{\alpha} (S i)$ .

**Proof.**

case alpha => ? ?; by rewrite / $H'_{g2h_{E0_{succ}}}$  H'\_{eq2}.

**Qed.**

**Lemma  $H'_{eq3}$**  alpha i :

$E0_{limit} \alpha \rightarrow H'_{\alpha} i = H'_{(E0_{Canon} \alpha)} (S i) i$ .

(\* ... \*)

### 7.4.1.2 Examples

**Lemma  $H'_{\omega}$**  k :  $H'_{E0_{\omega}} k = (2 * k).+1 \%nat$ .

**Proof.**

rewrite H'\_{eq3} ?/E0\_{limit} => //.

(\* ... \*)

**Lemma  $H'_{\omega\_double}$**  (k: nat) :

$H'_{(E0_{mul} E0_{\omega} (E0_{fin} 2))} k = (4 * k + 3)\%coq\_nat$ .

**Proof.**

by rewrite / $H'_{-H'_{\omega\_double} E0_{g2h\_mul} E0_{g2h_{\omega}} E0_{g2h\_Fin}}$ .

**Qed.**

### 7.4.1.3 Order and monotony properties

**Lemma  $H'_{dom}$**  alpha beta :

$E0_{lt} \alpha \beta \rightarrow \text{dominates\_strong} (H'_{\beta}) (H'_{\alpha})$ .

**Lemma  $H'_{\alpha\_mono}$**  (alpha : E0) : strict\_mono ( $H'_{\alpha}$ ).

**Proof.**

generalize (Hprime.H'\_{\alpha\\_mono} (E0\_{g2h} alpha)) => H x y /ltP.

move /H; by rewrite / $H'_{\alpha}$  => /ltP.

**Qed.**

**Theorem  $H'_{\alpha\_gt}$**  alpha (Halpha: alpha <> E0zero) n :

(n <  $H'_{\alpha} n$ )%N.

**Proof.**

move: (H'\_{\alpha\\_gt} (E0\_{g2h} alpha)) => H.

rewrite / $H'_{\alpha}$  ; apply /ltP; apply H => H0; apply Halpha.

apply gE0\_{eq\\_intro}; case: alpha H H0 Halpha => // ? ? ? H0 ?.

injection H0 => Heq; by rewrite -g2h\_{eq} ?Heq.

**Qed.**

**Lemma  $H'_{\omega\_cube\_min}$**  :

```
forall k : nat,
  0 <> k -> (hyper_exp2 k.+1 <= H'_ (E0_phi0 (E0fin 3)) k)%N.
```

**Proof.**

```
move => k Hk; apply /leP; transitivity (Hprime.H'_ (hE0phi0 3) k).
- by apply H'_omega_cube_min.
- by rewrite /H'_ E0g2h_phi0 E0g2h_Fin.
```

**Qed.**

### 7.4.2 The $F_\alpha$ hierarchy (Library `GF_alpha`)

The functions  $F_\alpha$  are described in Section 6.4 on page 136.

From `gaia_hydras.GF_alpha`.

**Definition** `F_ (alpha : E0) := F_alpha.F_ (E0_g2h alpha)`.

The following equalities correspond to the equations of `Epsilon0.F_alpha`

**Lemma** `F_zeroE i: F_ E0zero i = i.+1`.

**Lemma** `F_alpha_0_eq (alpha : E0): F_ alpha 0 = 1`.

**Lemma** `F_succE alpha i :`

```
F_ (E0_succ alpha) i = Iterates.iterate (F_ alpha) i.+1 i.
```

**Lemma** `F_limE alpha i:`

```
Tllimit (cnf alpha) -> F_ alpha i = F_ (E0Canon alpha i) i.
```

By rewriting, we import a few lemmas from `Epsilon0.F_alpha`.

**Lemma** `F_alpha_gt (alpha : E0) (n : nat): (n < F_ alpha n)%N`.

**Proof.** `apply /ltP; apply Epsilon0.F_alpha.F_alpha_gt. Qed.`

**Lemma** `F_alpha_mono (alpha: E0): strict_mono (F_ alpha)`.

**Proof.**

```
rewrite /strict_mono /F_ => n p Hnp; apply /ltP.
apply F_alpha_mono; move: Hnp => /ltP //.
```

**Qed.**

**Lemma** `F_alpha_dom alpha:`

```
dominates_from 1 (F_ (E0_succ alpha)) (F_ alpha).
```

**Proof.**

```
rewrite /dominates_from /F_ g2h_E0_succ => p Hp.
apply /ltP; apply F_alpha_dom; by apply /leP.
```

**Qed.**

The  $F_\alpha$  and  $H'_\alpha$  hierarchies satisfy the inequality  $F_\alpha(n) \leq H'_{\omega^\alpha}(n)$  for any  $\alpha$  and  $n > 0$ .

**Lemma** `H'_F alpha n : (F_ alpha n.+1 <= H'_ (E0_phi0 alpha) n.+1)%N`.

With the same technique, we prove that if  $\alpha \geq \omega$ , then  $F_\alpha$  is not primitive recursive.

**Lemma** `F_alpha_not_PR_E0 alpha:`

```
E0le E0_omega alpha -> isPR 1 (F_ alpha) -> False.
```

### 7.4.3 The $L_\alpha$ hierarchy (Library `GL_alpha`)

The functions  $L_\alpha$ , described in Sect. 6.2 on page 121 allow us to “compute” the length of standard hydra battles. In the same way as for the hierarchies  $F_\alpha$  and  $H'_\alpha$ , we import definitions and lemmas from the module `Epsilon0.L_alpha`.

From `gaia_hydras.GF_alpha`.

```
Definition L_ (alpha:E0) (i:nat): nat :=
  L_alpha.L_ (E0_g2h alpha) i.
```

```
Lemma L_zeroE i : L_ E0zero i = i.
```

```
Proof. by rewrite /L_. Qed.
```

```
Lemma L_eq2 alpha i :
```

```
  E0is_succ alpha -> L_ alpha i = L_ (E0_pred alpha) (S i).
```

```
Lemma L_succE alpha i : L_ (E0_succ alpha) i = L_ alpha i.+1.
```

```
Lemma L_limE alpha i :
```

```
  E0limit alpha -> L_ alpha i = L_ (E0Canon alpha i) (S i).
```

The following theorem compares the  $L_\alpha$  and  $H'_\alpha$  hierarchies.

```
Theorem H'_L_ a i: (H'_ a i <= L_ a i.+1)%N.
```

```
Proof. rewrite /L_ /H'_; apply /leP; by apply L_alpha.H'_L_. Qed.
```

### 7.4.4 Back to hydras

In `gaia_hydras.GHydra`, we import a few theorem about hydra battles (see `Hydra.Hydra_Theorems`).

#### 7.4.4.1 Termination of all battles

The termination measure is re-defined with Gaia’s ordinals.

```
Fixpoint m (h:Hydra) : T1 :=
```

```
  if h is node (hcons __ as hs) then ms hs else zero
```

```
with ms (s : Hydrae) : T1 :=
```

```
  if s is hcons h s' then phi0 (m h) o+ ms s' else zero.
```

```
Compute T1pp (m Examples.Hy).
```

```
= (ω ^ (ω ^ ω ^ 2 + 1) + 2)%pT1
: ppT1
```

```
Lemma mVariant: Hvariant nf_Wf free m .
```

```
Theorem every_battle_terminates : Termination.
```

### 7.4.5 Impossibility Theorems

The theorems described in Sect. 5.4 on page 108 and 5.5.3 on page 117 are adapted to Gaia's world.

**Lemma** `Impossibility_free mu m (Var: Hvariant nf_Wf free m):`  
`~ BoundedVariant Var mu.`

**Proof.**

`move => bvar; refine (Impossibility_free _ _ _ (bVar bvar)).`

**Qed.**

**Lemma** `Impossibility_std mu m (Var: Hvariant nf_Wf standard m):`  
`~ BoundedVariant Var mu.`

**Proof.**

`move => bvar; refine (Impossibility_std _ _ _ (bVar bvar)).`

**Qed.**

#### 7.4.5.1 Length of standard battles

Finally, we relate the length of standard-battles with the  $L_\alpha$  fast growing functions.

**Definition** `l_std alpha k := (L_alpha (S k) - k)%nat.`

**Definition** `T1toH (a: T1) : Hydra := O2H.iota (g2h a).`

**Lemma** `l_std_ok : forall alpha : E0,`  
`alpha != E0zero ->`  
`forall k : nat,`  
`(1 <= k)%N -> battle_length standard k (T1toH (cnf alpha))`  
`(l_std alpha k).`

## 7.5 Importing a theorem from Gaia

The Gaia library already contains many lemmas about ordinal arithmetic. In this section, we give two examples of porting such a lemma to Hydra-battles' vocabulary.

### 7.5.1 Associativity of ordinal multiplication (below $\epsilon_0$ )

Gaia already contains a proof that the multiplication of ordinals less than  $\epsilon_0$  is associative. From *gaia.ssete9.v*

**Lemma** `mulA: associative T1mul.`

This lemma was missing from *hydra-battles*. Nevertheless, we could adapt this lemma to *hydra-battles*' ordinals, by a small sequence of rewritings.

**Lemma** `g2h_multE (a b : T1) : g2h (a * b) = T1.T1mul (g2h a) (g2h b).`

**Proof.** `apply symmetry, refines2_R, mult_ref. Qed.`

The module `gaia_hydras.GaiaToHydra` shows a small example of importation of `multA` into `hydra-battles`' world.

```
From mathcomp Require Import all_ssreflect zify.
From gaia_hydras Require Import T1Bridge .
From hydras Require Import T1.
```

```
From gaia Require Import ssete9.
Import Epsilon0.T1.
```

In the rest of this module, names like `T1`, `omega`, etc. are bound to `Hydra-battles`' meaning.

**Locate** `T1`.

```
Inductive hydras.Epsilon0.T1.T1
Inductive gaia.ssete9.CantorOrdinal.T1
  (shorter name to refer to it in current context is CantorOrdinal.T1)
Notation gaia_hydras.T1Bridge.T1
  (shorter name to refer to it in current context is T1Bridge.T1)
Module hydras.Epsilon0.T1
```

**Lemma** `hmultA` : associative `T1mul`.

**Proof.**

```
move => a b c.
by rewrite -(g2h_h2gK a) -(g2h_h2gK b) -(g2h_h2gK c)
           -!g2h_multE multA.
```

**Qed.**

**Example** `Ex1` (`a`: `T1`): `T1omega * (a * T1omega) = T1omega * a * T1omega`.

**Proof.** `by rewrite hmultA. Qed.`

## 7.5.2 Right Distributivity, etc.

Likewise, we prove almost for free that ordinal multiplication is right distributive over addition (with `Hydra-battles`' definitions).

**Lemma** `hmult_dist` : right\_distributive `T1mul` `T1add`.

**Proof.**

```
move => a b c; move :(mul_distr (h2g a) (h2g b) (h2g c)) => H.
rewrite -(g2h_h2gK a) -(g2h_h2gK b) -(g2h_h2gK c).
by rewrite -!g2h_plusE -!g2h_multE H -g2h_plusE.
```

**Qed.**

We plan to automatize as soon as possible the proof of this kind of transfer lemmas from `Gaia` to `Hydra-battles`.

## Chapter 8

# Kurt Schütte's axiomatic definition of countable ordinals

In the present chapter, we compare our implementation of the segment  $[0, \epsilon_0)$  with a mathematical text in order to “validate” our constructions. Our reference here is the axiomatic definition of the set of countable ordinals, in chapter V of Kurt Schütte's book “ Proof Theory ” [Sch77].

**Remark 8.1** *In all this chapter, the word “ordinal” will be considered as a synonymous of “countable ordinal”*

Schütte's definition of countable ordinals relies on the following three axioms: There exists a strictly ordered set  $\mathbb{O}$ , such that

1.  $(\mathbb{O}, <)$  is well-ordered
2. Every bounded subset of  $\mathbb{O}$  is countable
3. Every countable subset of  $\mathbb{O}$  is bounded.

Starting with these three axioms, Schütte re-defines the vocabulary about ordinal numbers: the null ordinal 0, limits and successors, the addition of ordinals, the infinite ordinals  $\omega$ ,  $\epsilon_0$ ,  $\Gamma_0$ , etc.

This chapter describes an adaptation to Coq of Schütte's axiomatization. Unlike the rest of our libraries, our library `hydras.Schutte` is not constructive, and relies on several axioms.

- First, please keep in mind that the set of countable ordinals is not countable. Thus, we cannot hope to represent all countable ordinals as finite terms of an inductive type, which was possible with the set of ordinals strictly less than  $\epsilon_0$  (resp.  $\Gamma_0$ )
- We tried to be as close as possible to K. Schütte's text, which uses “classical” mathematics : excluded middle, Hilbert's  $\epsilon$  (choice) and Russel's  $\iota$  (definite description) operators. Both operators allow us to write definitions close to the natural mathematical language, such as “succ is *the* least ordinal strictly greater than  $\alpha$ ”

- Please note that only the library `Schutte/*`.v is “contaminated” by axioms, and that the rest of our libraries remain constructive.

## 8.1 Declarations and axioms

Let us declare a type `Ord` for representing countable ordinals, and a binary relation `lt`. Note that, in our development, `Ord` is a type, while the set of countable ordinals (called  $\mathbb{O}$  by Schütte) is the full set over the type `Ord`.

A set  $A$  is countable if there is an injective function from  $A$  to  $\mathbb{N}$  (see Library `Schutte.Countable`).

This library was initially written by Florian Hatat, as he was a student of *École Normale Supérieure de Lyon*. Stéphane Desarzens adapted it in order to make it compatible with the `Topology/ZornsLemma` project [Sch].

*From Module `Schutte.Schutte_basics`*

```
Parameter Ord : Type.
```

```
Parameter lt : relation Ord.
```

```
Infix "<" := lt : schutte_scope.
```

```
Notation ordinal := (@Full_set Ord).
```

```
Definition big0 alpha : Ensemble Ord := fun beta => beta < alpha.
```

Schütte’s first axiom tells that `lt` is a well order on the set `ordinal` (The class `WO` is defined in Module `Schutte.Well_Orders.v`).

```
Variables (M:Type)
          (Lt : relation M).
```

```
Definition Le (a b:M) := a = b \ / Lt a b.
```

```
Definition least_member (X:Ensemble M) (a:M) :=
  In X a /\ forall x, In X x -> Le a x.
```

```
Class WO : Type:=
{
  Lt_trans : Transitive Lt;
  Lt_irreflexive : forall a:M, ~ Lt a a;
  well_order : forall (X:Ensemble M)(a:M),
    In X a ->
      exists a0:M, least_member X a0
}.

```

```
Axiom AX1 : WO lt.
```

The second and third axioms say that a subset  $X$  of  $\mathbb{O}$  is (strictly) bounded if and only if it is countable.

```
Axiom AX2 :
forall X: Ensemble Ord,
  (exists a, (forall y, In X y -> y < a)) ->
```

Countable X.

**Axiom AX3 :**

```
forall X : Ensemble Ord,
  Countable X ->
  exists a, forall y, In X y -> y < a.
```

AX2 and AX3 could have been replaced by a single axiom (using the `iff` connector), but we decided to respect as most as possible the structure of Schütte's definitions.

## 8.2 Additional axioms

The adaptation of Schütte's mathematical discourse to Coq led us to import a few axioms from the standard library. We encourage the reader to consult Coq's FAQ about the safe use of axioms <https://github.com/coq/coq/wiki/The-Logic-of-Coq#axioms>.

### 8.2.0.1 Classical logic

In order to work with classical logic, we import the module `Coq.Logic.Classical` of Coq's standard library, specifically the following axiom:

```
Axiom classic : forall P:Prop, P ∨ ~P.
```

### 8.2.0.2 Description operators

In order to respect Schütte's style, we imported also the library `Coq.Logic.Epsilon`. The rest of this section presents a few examples of how Hilbert's choice operator and Church's definite description allow us to write understandable definitions (close to the mathematical natural language).

### 8.2.0.3 The definition of zero

According to the definition of a well order, every non-empty subset of `Ord` has a least element. Furthermore, this least element is unique. We would like to call this element `zero`.

**Remark R :** `exists! z : Ord, least_member lt ordinal z.`

**Proof.**

```
destruct inh_Ord as [a ];
  apply least_member_ex_unique with a.
- apply AX1.
- split.
```

**Qed.**

**Definition zero :** `Ord.`

**Proof.**

```
Fail destruct R.
```

```
The command has indeed failed with message:
Case analysis on sort Type is not allowed for
inductive definition ex.
```

```
Ord
```

**Abort.**

Indeed, the basic logic of Coq does not allow us to eliminate a proof of a proposition  $\exists! x : A, P(x)$  for building a term whose type lies in the sort `Type`. The reasons for this impossibility are explained in many documents [BC04a, Ch11, Coq].

Let us import the library `Coq.Logic.Epsilon`, which contains the following axiom and lemmas.

**Print** `epsilon_statement`.

```
*** [ epsilon_statement :
forall (A : Type) (P : A -> Prop),
inhabited A -> {x : A | (exists x0 : A, P x0) -> P x} ]

Arguments epsilon_statement [A]%type_scope
P%function_scope _
```

Hilbert's  $\epsilon$  operator is derived from this axiom.

**Print** `epsilon`.

```
epsilon =
fun (A : Type) (i : inhabited A) (P : A -> Prop) =>
proj1_sig (epsilon_statement P i)
: forall A : Type,
inhabited A -> (A -> Prop) -> A

Arguments epsilon [A]%type_scope i P%function_scope
```

**Check** `constructive_indefinite_description`.

```
constructive_indefinite_description
: forall (A : Type) (P : A -> Prop),
(exists x : A, P x) -> {x : A | P x}
```

If we consider the *unique existential* quantifier  $\exists!$ , we obtain Church's *definite description operator*.

**Check** `iota_statement`.

```
iota_statement
: forall (A : Type) (P : A -> Prop),
inhabited A ->
{x : A | (exists ! x0 : A, P x0) -> P x}
```

**Print** `iota`.

```

iota =
fun (A : Type) (i : inhabited A) (P : A -> Prop) =>
proj1_sig (iota_statement P i)
  : forall A : Type,
    inhabited A -> (A -> Prop) -> A

Arguments iota [A]%type_scope i P%function_scope

```

**Print** `iota_spec`.

```

iota_spec =
fun (A : Type) (i : inhabited A) (P : A -> Prop) =>
proj2_sig (iota_statement P i)
  : forall (A : Type) (i : inhabited A)
    (P : A -> Prop),
    (exists ! x : A, P x) -> P (iota i P)

Arguments iota_spec [A]%type_scope i P%function_scope
-

```

Indeed, the operators `epsilon` and `iota` allowed us to make our definitions quite close to Schütte's text. Our libraries `Schutte.MoreEpsilonIota` and `Schutte.PartialFun` are extensions of `Coq.logic.Epsilon` for making easier such definitions. See also an article in french [Cas07].

```

Class InH (A : Type) : Prop :=
  InHWit : inhabited A.

```

```

Definition some {A:Type} {H : InH A} (P: A -> Prop)
:= epsilon (@InHWit A H) P.

```

```

Definition the {A:Type} {H : InH A} (P: A -> Prop)
:= iota (@InHWit A H) P.

```

In order to use these tools, we have to tell Coq that the declared type `Ord` is not empty:

```

Axiom inh_Ord : inhabited Ord.

```

```

#[ global ] Instance InH_Ord : InH Ord.

```

```

Proof.
  exact inh_Ord.

```

```

Qed.

```

We are now able to define `zero` as the least ordinal. For this purpose, we define a function returning the least element of any [non-empty] subset.

*From ModuleSchutte.Well\_Orders*

```

Variables (M:Type)
  (Lt : relation M).

```

```

Definition Le (a b:M) := a = b \ / Lt a b.

```

```

Definition least_member (X:Ensemble M) (a:M) :=
  In X a /\ forall x, In X x -> Le a x.

```

```

Definition the_least {M: Type} {Lt}
  {inh : InH M} {WO: WO Lt} (X: Ensemble M) : M :=
  the (least_member Lt X ).

```

From Module Schutte.Schutte\_basics

```

Definition zero := the_least ordinal.

```

We want to prove now that zero is less than or equal to any ordinal number.

```

Lemma zero_le (alpha : Ord) : zero <= alpha.

```

**Proof.**

```

  unfold zero, the_least, the; apply iota_ind.

```

<pre> <b>alpha</b>: Ord exists ! x : Ord, least_member lt ordinal x </pre>
<pre> <b>alpha</b>: Ord forall a : Ord, unique (least_member lt ordinal) a -&gt; a &lt;= alpha </pre>

```

- apply the_least_unicity, Inh_ord.
- destruct 1 as [[_ H1] _]; apply H1; split.

```

**Qed.**

#### 8.2.0.4 Remarks on epsilon and iota

What would happen in case of a misuse of epsilon or iota? For instance, one could give a unsatisfiable specification to epsilon or a specification for iota that admits several realizations.

Let us consider an example:

```

Module Bad.

```

```

Definition bottom := the_least Empty_set.

```

Since we won't be able to prove the proposition  $(\text{exists! } a: \text{Ord}, \text{least\_member } (\text{Empty\_set } \text{Ord}) a)$ , the only properties we would be able to prove about bottom would be *trivial* properties, *i.e.*, satisfied by *any* element of type Ord, like for instance  $\text{bottom} = \text{bottom}$ , or  $\text{zero} \leq \text{bottom}$ .

```

Lemma le_zero_bottom : zero <= bottom.

```

```

Proof. apply zero_le. Qed.

```

```

Lemma bottom_eq : bottom = bottom.

```

```

Proof. trivial. Qed.

```

On the other hand, the following attempt fails, because of the unprovable first subgoal (please notice that the second subgoal is easy to solve!).

```

Lemma le_bottom_zero : bottom <= zero.
Proof.
  unfold bottom, the_least, the; apply iota_ind.

```

```

exists ! x : Ord, least_member lt Empty_set x

forall a : Ord,
unique (least_member lt Empty_set) a -> a <= zero

```

**Abort.**

**End Bad.**

In short, using `epsilon` and `iota` in our implementation of countable ordinals after Schütte has two main advantages.

- It allows us to give a *name* (using `Definition`) to witnesses of existential quantifiers (let us recall that, in classical logic, one may consider non-constructive proofs of existential statements)
- By separating definitions from proofs of [unique] existence, one may make definitions more concise and readable. Look for instance at the definitions of `zero`, `succ`, `plus`, etc. in the rest of this chapter.

### 8.3 The successor function

The definition of the function `succ:Ord -> Ord` is very concise. The successor of any ordinal  $\alpha$  is the smallest ordinal strictly greater than  $\alpha$ .

```

Definition succ (alpha : Ord)
:= the_least (fun beta => alpha < beta).

```

Using `succ`, we define the following predicates.

```

Definition is_succ (alpha:Ord)
:= exists beta, alpha = succ beta.

```

```

Definition is_limit (alpha:Ord)
:= alpha <> zero /\ ~ is_succ alpha.

```

How do we prove properties of the successor function? First, we make its specification explicit.

```

Definition succ_spec (alpha:Ord) :=
least_member lt (fun z => alpha < z).

```

Then, we prove that our function `succ` meets this specification.

```

Lemma succ_ok :
forall alpha, succ_spec alpha (succ alpha).

```

```

Proof.
intro alpha; unfold succ, the_least, the; apply iota_spec.

```

```

alpha: Ord
-----
exists ! x : Ord, succ_spec alpha x

```

We have now to prove that the set of all ordinals strictly greater than  $\alpha$  has a unique least element. But the singleton set  $\{\alpha\}$  is countable, hence bounded (by the axiom AX3). Hence; the set  $\{\beta \in \mathbb{O} \mid \alpha < \beta\}$  is not empty and therefore has a unique least element.

The rest of the Coq proof script is quite short.

```

destruct (@AX3 (Singleton alpha)).
- apply countable_singleton.
- unfold succ_spec; apply the_least_unicity; exists x.
  apply H; split.

```

**Qed.**

We can “uncap” the description operator for proving properties of the succ function.

**Lemma** `lt_succ` (`alpha` : Ord): `alpha < succ alpha`.

**Proof.**

```

destruct (succ_ok alpha); tauto.

```

**Qed.**

**Lemma** `lt_succ_le` (`alpha beta` : Ord):

```

alpha < beta -> succ alpha <= beta.

```

**Proof with** `eauto` with `schutte`.

```

intros H; pattern (succ alpha); apply the_least_ok ...
exists (succ alpha); red;apply lt_succ ...

```

**Qed.**

**Lemma** `lt_succ_le_2` (`alpha beta` : Ord):

```

alpha < succ beta -> alpha <= beta.

```

**Lemma** `succ_mono` (`alpha beta` : Ord):

```

alpha < beta -> succ alpha < succ beta.

```

**Lemma** `succ_monoR` (`alpha beta` : Ord) :

```

succ alpha < succ beta -> alpha < beta.

```

**Lemma** `succ_injection` (`alpha beta` : Ord) :

```

succ alpha = succ beta -> alpha = beta.

```

**Lemma** `succ_zero_diff` (`alpha` : Ord): `succ alpha <> zero`.

**Lemma** `zero_lt_succ` : forall `alpha`, `zero < succ alpha`.

**Lemma** `lt_succ_lt` (`alpha beta` : Ord) :

```

is_limit beta -> alpha < beta -> succ alpha < beta.

```

## 8.4 Finite ordinals

Using `succ`, it is now easy to define recursively all the finite ordinals.

```
Fixpoint finite (i:nat) : Ord :=
  match i with 0 => zero
            | S i => succ (finite i)
  end.
```

```
Notation F i := (finite i).
```

```
Coercion finite : nat -> Ord.
```

## 8.5 The definition of omega

In order to define  $\omega$ , the first infinite ordinal, we use an operator which “returns” the least upper bound (if it exists) of a subset  $X \subseteq \mathbb{O}$ . For that purpose, we first use a predicate: `(is_lub D lt X a)` if  $a$  belongs to  $D$  and is the least upper bound of  $X$  (with respect to  $lt$ ).

```
Definition upper_bound (M:Type)
  (D: Ensemble M)
  (lt: relation M)
  (X:Ensemble M)
  (a:M) :=
  forall x, In _ D x -> In _ X x -> x = a \ / lt x a.
```

```
Definition is_lub (M:Type)
  (D : Ensemble M)
  (lt : relation M)
  (X:Ensemble M)
  (a:M) :=
  In _ D a /\ upper_bound D lt X a /\
  (forall y, In _ D y ->
    upper_bound D lt X y ->
    y = a \ / lt a y).
```

```
Definition sup_spec X lambda := is_lub ordinal lt X lambda.
```

```
Definition sup (X: Ensemble Ord) : Ord := the (sup_spec X).
```

```
Notation "'|_|' X" := (sup X) (at level 29) : schutte_scope.
```

Then, we define the function `omega_limit` which returns the least upper bound of the (denumerable) range of any sequence  $s: \text{nat} \rightarrow \text{Ord}$ . By AX3 this range is bounded, hence the set of its upper bounds is not empty and has a least element. Then we define `omega` as the limit of the sequence of finite ordinals.

```
Definition omega_limit (s:nat->Ord) : Ord
```

```
:= |_| (seq_range s).
```

```
Definition _omega := omega_limit finite.
```

```
Notation omega := (_omega).
```

Among the numerous properties of the ordinal  $\omega$ , let us quote the following ones (proved in Module `Schutte.Schutte_basics`)

```
Lemma finite_lt_omega (i : nat) : i < omega.
```

```
Lemma zero_lt_omega : zero < omega.
```

```
Proof.
```

```
change zero with (F 0); apply finite_lt_omega.
```

```
Qed.
```

```
Lemma lt_omega_finite (alpha : Ord) :  
alpha < omega -> exists i:nat, alpha = i.
```

```
Lemma is_limit_omega : is_limit omega.
```

### 8.5.1 Ordering functions and ordinal addition

After having defined the finite ordinals and the infinite ordinal  $\omega$ , we define the sum  $\alpha + \beta$  of two countable ordinals. Schütte's definition looks like the following one:

“ $\alpha + \beta$  is the  $\beta$ -th ordinal greater than or equal to  $\alpha$ ”

The purpose of this section is to give a meaning to the construction “the  $\alpha$ -th element of  $X$ ” where  $X$  is any non-empty subset of  $\mathbb{O}$ . We follow Schütte's approach, by defining the notion of *ordering functions*, a way to associate a unique ordinal to each element of  $X$ . Complete definitions and proofs can be found in Module `Schutte.Ordering_Functions`).

### 8.5.2 Definitions

A *segment* is a set  $A$  of ordinals such that, whenever  $\alpha \in A$  and  $\beta < \alpha$ , then  $\beta \in A$ ; a segment is *proper* if it strictly included in  $\mathbb{O}$ .

```
Definition segment (A: Ensemble Ord) :=  
forall alpha beta, In A alpha -> beta < alpha -> In A beta.
```

```
Definition proper_segment (A: Ensemble Ord) :=  
segment A /\ ~ Same_set A ordinal.
```

Let  $A$  be a segment, and  $B$  a subset of  $\mathbb{O}$ : an *ordering function for  $A$  and  $B$*  is a strictly increasing bijection from  $A$  to  $B$ . The set  $B$  is said to be an *ordering segment* of  $A$ . Our definition in `Coq` is a direct translation of the mathematical text of [Sch77].

```

Class ordering_function (f : Ord -> Ord)
  (A B : Ensemble Ord) : Prop :=
  Build_OF {
    OF_segment : segment A;
    OF_total : forall a, In A a -> In B (f a);
    OF_onto : forall b, In B b -> exists a, In A a /\ f a = b;
    OF_mono : forall a b, In A a -> In A b -> a < b -> f a < f b
  }.

Definition ordering_segment (A B : Ensemble Ord) :=
  exists f : Ord -> Ord, ordering_function f A B.

```

We are now able to associate with any subset  $B$  of  $\mathbb{O}$  its ordering segment and ordering function.

```

Definition the_ordering_segment (B : Ensemble Ord) :=
  the (fun x => ordering_segment x B).

Definition ord (B : Ensemble Ord) :=
  some (fun f => ordering_function f (the_ordering_segment B) B).

```

Thus  $(\text{ord } B \alpha)$  is the  $\alpha$ -th element of  $B$ . Please note that the last definition uses the epsilon-based operator `some` and not `the`. This is due to the fact that we cannot prove the unicity (w.r.t. Leibniz' equality) of the ordering function of a given set. By contrast, we admit the axiom `Extensionality_Ensembles`, from the library `Coq.Sets.Ensembles`, so we use the operator `the` in the definition of `the_ordering_segment`.

One of the main theorems of `Ordering_Functions` associates a unique segment and a unique (up to extensionality) ordering function to every subset  $B$  of  $\mathbb{O}$ .

**About** `ordering_function_ex`.

```

ordering_function_ex :
forall B : Ensemble Ord,
exists ! S : Ensemble Ord,
  exists f : Ord -> Ord, ordering_function f S B

ordering_function_ex is not universe polymorphic
Arguments ordering_function_ex B
ordering_function_ex is opaque
Expands to: Constant
Ordering_Functions.ordering_function_ex

```

**About** `ordering_function_unicity`.

```

ordering_function_unicity :
forall [B A1 A2 : Ensemble Ord] [f1 f2 : Ord -> Ord],
ordering_function f1 A1 B ->
ordering_function f2 A2 B -> fun_equiv f1 f2 A1 A2

ordering_function_unicity is not universe polymorphic
Arguments ordering_function_unicity [B A1 A2]
  [f1 f2]%function_scope 01 02
ordering_function_unicity is opaque
Expands to: Constant
Ordering_Functions.ordering_function_unicity

```

Thus, our function `ord` which enumerates the elements of  $B$  is defined in a non-ambiguous way. Let us quote the following theorems (see `Library Schutte.Ordering_Functions` for more details).

(\* Theorem 13.3 of Schutte's book \*)

```

Theorem ordering_le : forall f A B,
  ordering_function f A B ->
  forall alpha, In A alpha -> alpha <= f alpha.

```

(\* Theorem 13.5.2 by Schutte \*)

**About** Th\_13\_5\_2.

```

Th_13_5_2 :
forall [A B : Ensemble Ord] [f : Ord -> Ord],
ordering_function f A B ->
Closed B -> continuous f A B

Th_13_5_2 is not universe polymorphic
Arguments Th_13_5_2 [A B] [f]%function_scope f_ord
  B_closed
Th_13_5_2 is opaque
Expands to: Constant Ordering_Functions.Th_13_5_2

```

### 8.5.3 Ordinal addition

We are now ready to define and study addition on the type `Ord`. The following definitions and proofs can be consulted in `Module Schutte.Addition.v`.

**Definition** `plus alpha` := `ord (ge alpha)`.

**Infix** `"+"` := `plus : schutte_scope`.

In other words,  $\alpha + \beta$  is the  $\beta$ -th ordinal greater than or equal to  $\alpha$ . Thanks to generic properties of ordering functions, we can show the following properties of addition on  $\mathbb{O}$ . First, we prove a useful lemma:

```

Lemma plus_elim (alpha : Ord) :
  forall P : (Ord->Ord)->Prop,
  (forall f: Ord->Ord,

```

```

    ordering_function f ordinal (ge alpha)-> P f) ->
  P (plus alpha).

```

**Proof.**

```

  intros P H0; now apply H0, plus_ordering.

```

**Qed.**

As a use-case, let us prove that 0 is a right neutral element of +.

**Lemma alpha\_plus\_zero** (alpha: Ord): alpha + zero = alpha.

**Proof.**

```

  pattern (plus alpha); apply plus_elim; eauto.

```

```

alpha: Ord
-----
forall f : Ord -> Ord,
ordering_function f ordinal (ge alpha) ->
f zero = alpha

```

(\* ... \*)

The following lemmas are proved the same way.

**Lemma zero\_plus\_alpha** (alpha : Ord): zero + alpha = alpha.

**Lemma le\_plus\_l** (alpha beta : Ord) : alpha <= alpha + beta.

**Lemma le\_plus\_r** (alpha beta : Ord) : beta <= alpha + beta.

**Lemma plus\_mono\_r** (alpha beta gamma : Ord) :  
beta < gamma -> alpha + beta < alpha + gamma.

**Lemma plus\_of\_succ** (alpha beta : Ord) :  
alpha + (succ beta) = succ (alpha + beta).

The following lemmas are not direct applications of plus\_elim.

**#[global] Instance plus\_assoc**: Assoc eq plus.

```

Assoc eq plus

```

**Lemma finite\_plus\_infinite** (n : nat) (alpha : Ord) :  
omega <= alpha -> n + alpha = alpha.

It is interesting to compare the proof of these lemmas with the computational proofs of the corresponding statements in Module `Epsilon0.T1`. For instance, the proof of the lemma `one_plus_omega` uses the continuity of ordering functions (applied to `(plus 1)`) and compares the limit of the  $\omega$ -sequences  $i_{(i \in \mathbb{N})}$  and  $(1 + i)_{(i \in \mathbb{N})}$ , whereas in the library `Epsilon0/T1`, the equality  $1 + \omega = \omega$  is just proved with reflexivity!

### 8.5.3.1 Multiplication by a natural number

The multiplication of an ordinal by a natural number is defined in terms of addition. This operation is useful for the study of Cantor normal forms.

```

Fixpoint mult_Sn (alpha:Ord)(n:nat){struct n} :Ord :=
  match n with 0 => alpha
              | S p => mult_Sn alpha p + alpha
  end.

```

```

Definition mult_fin_r alpha n :=
  match n with
    0 => zero
  | S p => mult_Sn alpha p
  end.

```

```

Infix "*" := mult_fin_r : schutte_scope.

```

## 8.6 The exponential of basis $\omega$

In this section, we define the function which maps any  $\alpha \in \mathbb{O}$  to the ordinal  $\omega^\alpha$ , also written  $\phi_0(\alpha)$ . It is an opportunity to apply the definitions and results of the preceding section. Indeed, Schütte first defines a subset of  $\mathbb{O}$ : the set of additive principal ordinals, and  $\phi_0$  is just defined as the ordering function of this set.

### 8.6.1 Additive principal ordinals

**Definition 8.1** *A non-zero ordinal  $\alpha$  is said to be additive principal if, for all  $\beta < \alpha$ ,  $\beta + \alpha$  is equal to  $\alpha$ . We call AP the set of additive principal ordinals.*

*From Module Schutte.AP*

```

Definition AP : Ensemble Ord :=
  fun alpha =>
    zero < alpha /\
    (forall beta, beta < alpha -> beta + alpha = alpha).

```

### 8.6.2 The function $\phi_0$

Let us call  $\phi_0$  the ordering function of AP. In the mathematical text, we shall use indifferently the notations  $\omega^\alpha$  and  $\phi_0(\alpha)$ .

```

Definition _phi0 := ord AP.

```

```

Notation phi0 := _phi0.

```

```

Notation "'omega^'" := phi0 (only parsing) : schutte_scope.

```

### 8.6.3 Omega-towers and the ordinal $\epsilon_0$

Using  $\phi_0$ , we can define recursively the set of finite omega-towers.

```

Fixpoint omega_tower (i : nat) : Ord :=
  match i with
  | 0 => 1
  | S j => phi0 (omega_tower j)
  end.

```

Then, the ordinal  $\epsilon_0$  is defined as the limit of the sequence of all finite towers (a kind of infinite tower).

```

Definition epsilon0 := omega_limit omega_tower.

```

The rest of our library `AP` is devoted to the proof of properties of additive principal ordinals, hence of the ordering function  $\phi_0$  and the ordinal  $\epsilon_0$  (which we could not express within the type `T1`).

### 8.6.4 Properties of the set `AP`

The set of additive principal ordinals is not empty: it contains at least the ordinals 1 and  $\omega$ .

```

Lemma AP_one : In AP 1.
Lemma least_AP : least_member lt AP 1.
Lemma AP_omega : In AP omega.
Lemma omega_second_AP :
  least_member lt
    (fun alpha => 1 < alpha /\ In AP alpha)
    omega.

```

The set `AP` is *closed* under addition, and unbounded.

```

Lemma AP_plus_closed (alpha beta gamma : Ord):
  In AP alpha -> beta < alpha -> gamma < alpha ->
  beta + gamma < alpha.

```

```

Theorem AP_unbounded : Unbounded AP.

```

```

Proof.

```

```

  intro x.

```

```

x: Ord
-----
exists y : Ord, In AP y /\ x < y

```

```

exists (omega_limit
  (fix seq (n : nat) : Ord :=
    match n with
    | 0 => succ x
    | S p => seq p + seq p
    end)).
(* ... *)

```

Finally, `AP` is (topologically) *closed* and ordered by the segment of all countable ordinals.

From Module `Schutte.Schutte_basics`

**Definition** `Closed` (`B : Ensemble Ord`) : `Prop` :=  
`forall M, Included M B -> Inhabited M -> Countable M ->`  
`In B (|_| M).`

**Theorem** `AP_closed` : `Closed AP`.

**Theorem** `AP_o_segment` : `the_ordering_segment AP = ordinal`.

#### 8.6.4.1 Properties of the function $\phi_0$

The ordering function  $\phi_0$  of the set `AP` is defined on the full set  $\mathbb{O}$  and is continuous (Schütte calls such a function *normal*).

`Theorem normal_phi0 : normal phi0 AP.`

The following properties come from the definition of  $\phi_0$  as the ordering function of `AP`. It may be interesting to compare these proofs with the computational ones described in Chapter 4.

**Lemma** `phi0_elim` : `forall P : (Ord->Ord)->Prop,`  
`(forall f: Ord->Ord,`  
`ordering_function f ordinal AP -> P f) ->`  
`P phi0.`

**Proof.**

`intros P H; apply H, phi0_ordering.`

**Qed.**

**Lemma** `AP_phi0` (`alpha : Ord`) : `In AP (phi0 alpha)`.

**Proof.**

`pattern phi0; apply phi0_elim.`  
`destruct 1 as [H H0 H1 H2]; apply H0;auto; split.`

**Qed.**

**Lemma** `phi0_zero` : `phi0 zero = 1`.

**Lemma** `phi0_mono` (`alpha beta : Ord`) :  
`alpha < beta -> phi0 alpha < phi0 beta.`

**Lemma** `phi0_mono_weak` (`alpha beta : Ord`) :  
`alpha <= beta -> phi0 alpha <= phi0 beta.`

**Lemma** `phi0_mono_R` (`alpha beta : Ord`) :  
`phi0 alpha < phi0 beta -> alpha < beta.`

**Lemma** `phi0_mono_R_weak` (`alpha beta: Ord`):  
`phi0 alpha <= phi0 beta -> alpha <= beta.`

**Lemma** `phi0_inj` (`alpha beta : Ord`) :  
`phi0 alpha = phi0 beta -> alpha = beta.`

**Lemma phi0\_positive** (alpha : Ord): zero < phi0 alpha.

**Lemma plus\_lt\_phi0** (ksi alpha: Ord):  
ksi < phi0 alpha -> ksi + phi0 alpha = phi0 alpha.

**Lemma phi0\_alpha\_phi0\_beta** (alpha beta: Ord) :  
alpha < beta -> phi0 alpha + phi0 beta = phi0 beta.

**Lemma phi0\_sup** : forall U: Ensemble Ord,  
Inhabited U ->  
Countable U ->  
phi0 (|\_| U) = |\_| (image U phi0).

**Lemma phi0\_of\_limit** (alpha : Ord) :  
is\_limit alpha ->  
phi0 alpha = |\_| (image (members alpha) phi0).

**Lemma AP\_to\_phi0** (alpha : Ord) :  
AP alpha -> exists beta, alpha = phi0 beta.

**Lemma AP\_plus\_AP** (alpha beta gamma : Ord) :  
zero < beta ->  
phi0 alpha + beta = phi0 gamma ->  
alpha < gamma /\ beta = phi0 gamma.

**Lemma is\_limit\_phi0** (alpha : Ord) :  
zero < alpha -> is\_limit (phi0 alpha).

**Lemma omega\_eqn** : omega = phi0 1.

**Lemma le\_phi0** (alpha : Ord) : alpha <= phi0 alpha.

### 8.6.5 A last example

Let us prove again the equality  $\omega + 42 + \omega^2 = \omega^2$ . Let us recall that  $\omega^2$  is an abbreviation of  $\phi_0(2)$ , *i.e.* the third additive principal ordinal.

**Example Ex42:**  $\omega + 42 + \omega^2 = \omega^2$ .

Our proof is very different from the computational proof of Sect 4.1.10 on page 86. By definition of additive principal ordinals, it suffices to prove the inequality  $\omega + 42 < \phi_0(2)$ .

```
assert (HAP:= AP_phi0 2).
elim HAP; intros _ H0; apply H0; clear H0.
```

```
HAP: In AP (phi0 (F 2))
-----
omega + F 42 < phi0 (F 2)
```

Since the set AP of additive principals is closed under addition (by Lemma AP\_plus\_closed, page 175), it suffices to prove the inequalities  $\omega < \phi_0(2)$  and  $42 < \phi_0(2)$ .

**Check** AP\_plus\_closed.

```
AP_plus_closed
: forall alpha beta gamma : Ord,
  In AP alpha ->
  beta < alpha ->
  gamma < alpha -> beta + gamma < alpha
```

```
assert (Hlt: omega < omega^2) by
  (rewrite omega_eqn; apply phi0_mono, finite_mono;
  auto with arith).
```

```
HAP: In AP (phi0 (F 2))
Hlt: omega < phi0 (F 2)
-----
omega + F 42 < phi0 (F 2)
```

```
apply AP_plus_closed; trivial.
```

```
F 42 < phi0 (F 2)
```

```
(* ... *)
```

## 8.7 More about $\epsilon_0$

Let us recall that the limit ordinal  $\epsilon_0$  cannot be written within the type T1. Since we are now considering the set of all countable ordinals, we can now prove some properties of this ordinal.

We prove the inequality  $\alpha < \omega^\alpha$  whenever  $\alpha < \epsilon_0$ . *Note that this condition was implicit in Module Epsilon0.T1.*

**Lemma** lt\_phi0 (alpha : Ord):  
alpha < epsilon0 -> alpha < phi0 alpha.

The proof is as follows:

1. Since  $\alpha < \epsilon_0$ , consider the least  $i$  such that  $\alpha$  is strictly less than the omega-tower of height  $i$ .
2.
  - If  $i = 0$ , then the result is trivial (because  $\alpha = 0$ )
  - Otherwise let  $i = j + 1$ ;  $\alpha$  is greater than or equal to the omega-tower of height  $j$ . By monotony,  $\phi_0(\alpha)$  is greater than or equal to the omega-tower of height  $j + 1$ , thus strictly greater than  $\alpha$

Moreover,  $\epsilon_0$  is the least ordinal  $\alpha$  that verifies the equality  $\alpha = \omega^\alpha$ , in other words, the least fixpoint of the function  $\phi_0$ .

**Theorem** `epsilon0_lfp` : `least_fixpoint lt phi0 epsilon0`.

## 8.8 Critical ordinals

For any (countable) ordinal  $\alpha$ , the set  $Cr(\alpha)$  is inductively defined as follows by Schütte (p.81 of [Sch77]).

- $Cr(0)$  is the set  $AP$  of additive principal ordinals.
- If  $0 < \alpha$ , then  $Cr(\alpha)$  is the intersection of all the sets of fixpoints of the  $Cr(\beta)$  for  $\beta < \alpha$ .

This definition is translated in Coq in Module `Schutte.Critical`, as the least fixpoint of a functional.

```
Definition Cr_fun : forall alpha : Ord,
  (forall beta : Ord, beta < alpha -> Ensemble Ord) ->
  Ensemble Ord
:=
  fun (alpha: Ord)
    (Cr : forall beta, beta < alpha -> Ensemble Ord)
    (x : Ord) => (
      (alpha = zero /\ AP x) \/
      (zero < alpha /\
        forall beta (H:beta < alpha),
          In (the_ordering_segment (Cr beta H)) x /\
            ord (Cr beta H) x = x)).
```

```
Definition Cr (alpha : Ord) : Ensemble Ord :=
  (Fix all_ord_acc
    (fun (_:Ord) => Ensemble Ord) Cr_fun) alpha.
```

Lets us denote by  $\phi_\alpha$  the ordering function of the set  $Cr(\alpha)$  and by  $A_\alpha$  its ordering segment.

```
Definition phi (alpha : Ord) : Ord -> Ord := ord (Cr alpha).
```

```
Definition A_ (alpha : Ord) : Ensemble Ord := the_ordering_segment (Cr alpha).
```

For instance, we prove that  $Cr(0)$  is the set of additive principals and that  $\epsilon_0$  belongs to  $Cr(1)$ .

```
Lemma Cr_zero_AP : Cr zero = AP.
```

```
Lemma epsilon0_Cr1 : In (Cr 1) epsilon0.
```

**Exercise 8.1** Prove that  $\epsilon_0$  is the least element of  $Cr(1)$ .

### 8.8.1 A flavor of infinity

The family of the  $Cr(\alpha)$ s is made of infinitely many unbounded (hence infinite) sets. Let us quote Lemma 5, p. 82 of [Sch77]:

For all  $\alpha$ , the set  $Cr(\alpha)$  is closed (for the least upper bound of non-empty countable sets) and unbounded.

We prove this result by a transfinite induction on  $\alpha$  of the conjunction of both properties.

**Theorem** `Unbounded_Cr alpha` : Unbounded (Cr alpha).

**Theorem** `Closed_Cr alpha` : Closed (Cr alpha).

## 8.9 Cantor normal form

The notion of Cantor normal form is defined for all countable ordinals. Nevertheless, note that, contrary to the implementation based on type `T1`, the Cantor normal form of an ordinal  $\alpha$  may contain  $\alpha$  as a sub-term<sup>1</sup>.

We represent Cantor normal forms as lists of ordinals. A list  $l$  is a Cantor normal form of a given ordinal  $\alpha$  if it satisfies two conditions:

- The list  $l$  is sorted (in decreasing order) w.r.t. the order  $\leq$
- The sum of all the  $\omega^{\beta_i}$  where the  $\beta_i$  are the terms of  $l$  (in this order) is equal to  $\alpha$ .

*From Schutte.CNF*

**Definition** `cnf_t` := list Ord.

**Fixpoint** `eval` (l : cnf\_t) : Ord :=  
`match l with`  
`| nil => zero`  
`| beta :: l' => phi0 beta + eval l'`  
`end.`

**Definition** `sorted` (l : cnf\_t) :=  
 LocallySorted (fun alpha beta => beta <= alpha) l.

**Definition** `is_cnf_of` (alpha : Ord)(l : cnf\_t) : Prop :=  
 sorted l /\ alpha = eval l.

By transfinite induction on  $\alpha$ , we prove that every countable ordinal  $\alpha$  has at least a Cantor normal form.

**Theorem** `cnf_exists` (alpha : Ord) :  
`exists l: cnf_t, is_cnf_of alpha l.`

---

<sup>1</sup>This would prevent us from trying to represent Cantor normal forms as finite trees (like in Sect. 4.1.2)

By structural induction on lists, we prove that this normal form is unique.

```
Lemma cnf_unicity l alpha:
  is_cnf_of alpha l ->
  forall l', is_cnf_of alpha l' -> l = l'.
```

```
Theorem cnf_exists_unique (alpha:Ord) :
  exists! l: cnf_t, is_cnf_of alpha l.
```

**Proof.**

```
destruct (cnf_exists alpha) as [l Hl]; exists l; split; auto.
now apply cnf_unicity.
```

**Qed.**

Finally, the following two lemmas relate  $\epsilon_0$  with Cantor normal forms.

If  $\alpha < \epsilon_0$ , then the Cantor normal form of  $\alpha$  is made of ordinals strictly less than  $\alpha$ .

```
Lemma cnf_lt_epsilon0 : forall l alpha,
  is_cnf_of alpha l ->
  alpha < epsilon0 ->
  Forall (fun beta => beta < alpha) l.
```

**Exercise 8.2** Please consider the following statement :

```
Lemma cnf_lt_epsilon0_iff :
  forall l alpha,
  is_cnf_of alpha l ->
  (alpha < epsilon0 <-> Forall (fun beta => beta < alpha) l).
```

Is it true? *You may start this exercise with the file `exercises/ordinals/schutte_cnf_counter_example.v`.*

Finally, the Cantor normal form of  $\epsilon_0$  is just  $\omega^{\epsilon_0}$ .

```
Lemma cnf_of_epsilon0 : is_cnf_of epsilon0 [epsilon0].
```

**Proof.**

```
split; [constructor | cbn].
```

```
epsilon0 = phi0 epsilon0 + zero
```

```
now rewrite alpha_plus_zero, epsilon0_fxp.
```

**Qed.**

**Project 8.1** Implement pages 82 to 85 of [Sch77] (critical, strongly critical, maximal critical ordinals, Feferman's ordinal  $\Gamma_0$ ).

**Remark 8.2** The sub-directory `theories/ordinals/Gamma0` contains an (incomplete, still undocumented) implementation of the set of ordinals below  $\Gamma_0$ , represented in Veblen normal form.

## 8.10 An embedding of T1 into Ord

Our library `Schutte.Correctness_E0` establishes the link between two very different modelizations of ordinal numbers. In other words, it “validates” a data structure in terms of a classical mathematical discourse considered as a model. First, we define a function from `T1` into `Ord` by structural recursion.

```
Fixpoint inject (t:T1) : Ord :=
  match t with
  | T1.zero => zero
  | T1.cons a n b => AP._phi0 (inject a) * S n + inject b
  end.
```

This function enjoys good commutation properties with respect to the main operations which allow us to build Cantor normal forms.

**Theorem** `inject_of_zero` : `inject T1.zero = zero`.

**Proof.** `reflexivity. Qed.`

**Theorem** `inject_of_finite` (n : nat):

`inject (\F n) = n`.

**Theorem** `inject_of_omega` :

`inject T1.omega = Schutte_basics._omega`.

**Theorem** `inject_of_phi0` (alpha : T1):

`inject (T1.phi0 alpha) = AP._phi0 (inject alpha)`.

**Theorem** `inject_plus` (alpha beta : T1):

`nf alpha -> nf beta ->`  
`inject (alpha + beta)%t1 = inject alpha + inject beta`.

**Theorem** `inject_mult_fin_r` (alpha : T1) :

`nf alpha ->`  
`forall n:nat,`  
`inject (alpha * n)%t1 = inject alpha * n`.

Finally, we prove that `inject` is a bijection from the set of all terms of `T1` in normal form to the set `(members epsilon0)` of the elements of `Ord` strictly less than  $\epsilon_0$ .

**Theorem** `inject_lt_epsilon0` (alpha : T1):

`inject alpha < epsilon0`.

**Theorem** `embedding` : `fun_bijection` (nf: Ensemble T1)

`(members epsilon0)`

`inject`.

### 8.10.1 Remarks

Let us recall that the library `Schutte` depends on five *axioms* and lies explicitly in the framework of classical logic with a weak version of the axiom of choice (please look at the documentation of `Coq.Logic.ChoiceFacts`).

On the other hand, the other libraries: `Epsilon0`, `Hydra`, et `Gamma0` do not import any axioms and are really constructive.

**Project 8.2** There is no construction of ordinal multiplication in [Sch77]. It would be interesting to derive this operation from Schütte's axioms, and prove its consistence with multiplication in ordinal notations for  $\epsilon_0$  and  $\Gamma_0$ .

## 8.11 Related work

In [Gri13], José Grimm establishes the consistency between our ordinal notations  $T1$  and  $T2$  (Veblen normal form) and his implementation of ordinal numbers after Bourbaki's set theory.

The Gaia project <https://github.com/coq-community/gaia> maintains Grimm's theory of ordinals as part of `coq-community` on GitHub. Integration of the present ordinal theory with Gaia, i.e., relating the different notions of ordinals and transferring relevant results, is an interesting project. First experiments in that direction are developed in the `theories/gaia/` directory.



# Chapter 9

## The Ordinal $\Gamma_0$ (first draft)

*This chapter and the files it presents are still very incomplete, considering the impressive properties of  $\Gamma_0$  [Gal91]. We hope to add new material soon, and accept contributions!*

**To do 9.1** *Build a T2Bridge! (note: Gamma0's wellfoundedness is missing in Gaia (?)).*

### 9.1 Introduction

We present a notation system for the ordinal  $\Gamma_0$ , following Chapter V, Section 14 of [Sch77]: “A notation system for the ordinals  $< \Gamma_0$ ”. We try to be as close as possible to Schütte’s text and usual practices of Coq developments.

The ordinal  $\Gamma_0$  is defined in Section 13 of [Sch77] as the least *strongly critical ordinal*. It is widely known as the *Feferman-Schütte ordinal*.

Section V, 13 of [Sch77] defines *strongly critical* and *maximal  $\alpha$ -critical* ordinals:

- $\alpha$  is strongly critical if  $\alpha$  is  $\alpha$ -critical,
- $\gamma$  is maximal  $\alpha$ -critical if  $\gamma$  is  $\alpha$ -critical, and, for all  $\xi > \alpha$ ,  $\gamma$  is not  $\xi$ -critical.

*From Schutte.Critical*

**Definition** `strongly_critical alpha` := In (Cr alpha) alpha.

**Definition** `maximal_critical alpha` : Ensemble Ord :=  
  fun gamma =>  
    In (Cr alpha) gamma /\  
    forall xi, alpha < xi -> ~ In (Cr xi) gamma.

**Definition** `Gamma0` := the\_least strongly\_critical.

**Project 9.1** Prove that a (countable) ordinal  $\alpha$  is strongly critical iff  $\phi_\alpha(0) = \alpha$  (Theorem 13.13 of [Sch77]).

**Project 9.2** Prove that the set of strongly critical ordinals is unbounded and closed (Theorem 13.14 of [Sch77]). Thus this set is not empty, hence has a least element. Otherwise, the definition of  $\Gamma_0$  above would be useless.

In the present version of this development, we only study  $\Gamma_0$  as a notation system, much more powerful than the ordinal notation for  $\epsilon_0$ .

## 9.2 The type T2 of ordinal terms

The notation system for ordinals less than  $\gamma_0$  comes from the following theorem of [Sch77], where  $\psi_\alpha$  is the ordering function of the set of maximal  $\alpha$ -critical ordinals.

Any ordinal  $\neq 0$  which is not strongly critical can be expressed in terms of  $+$  and  $\psi$ .

**Project 9.3** This theorem is not formally proved in this development yet. It should be!

Like in Chapter 4, we define an inductive type with two constructors, one for 0, the other for the construction  $\psi(\alpha, \beta) \times (n + 1) + \gamma$ , adapting a Manolios-Vroon-like notation [MV05] to *Veblen normal forms*.

From *Gamma0.T2*

```
Declare Scope T2_scope.
```

```
Delimit Scope T2_scope with t2.
```

```
Open Scope T2_scope.
```

```
Inductive T2 : Set :=
```

```
| zero : T2
```

```
| gcons : T2 -> T2 -> nat -> T2 -> T2.
```

```
Notation "[ alpha , beta ]" := (gcons alpha beta 0 zero)
      (at level 0): T2_scope.
```

```
Definition psi alpha beta := [alpha, beta].
```

```
Definition psi_term alpha :=
```

```
  match alpha with zero => zero
```

```
    | gcons a b n c => [a, b]
```

```
  end.
```

Like in chapter 4, we get familiar with the type T2 by recognizing simple constructs like finite ordinals,  $\omega$ , etc., as inhabitants of T2.

```
Notation one := [zero, zero].
```

```
Notation FS n := (gcons zero zero n zero).
```

```
Definition fin (n:nat) := match n with 0 => zero | S p => FS p end.
```

**Coercion** `fin` : `nat`  $\rightarrow$  `T2`.

**Notation** `omega` := `[zero,one]`.

**Notation** `epsilon0` := `[one,zero]`.

**Definition** `epsilon alpha` := `[one, alpha]`.

### 9.3 A strict order on T2

Let us define a strict order on type `T2`. The following definition is an adaptation of Schütte's, taking into account the multiplications by a natural number (inspired by [MV05], and also present in `T1`).

```

Inductive lt : T2  $\rightarrow$  T2  $\rightarrow$  Prop :=
| (* 1 *)
  lt_1 : forall alpha beta n gamma, zero t2 < gcons alpha beta n gamma
| (* 2 *)
  lt_2 : forall alpha1 alpha2 beta1 beta2 n1 n2 gamma1 gamma2,
    alpha1 t2 < alpha2  $\rightarrow$ 
    beta1 t2 < gcons alpha2 beta2 0 zero  $\rightarrow$ 
    gcons alpha1 beta1 n1 gamma1 t2 <
    gcons alpha2 beta2 n2 gamma2
| (* 3 *)
  lt_3 : forall alpha1 beta1 beta2 n1 n2 gamma1 gamma2,
    beta1 t2 < beta2  $\rightarrow$ 
    gcons alpha1 beta1 n1 gamma1 t2 <
    gcons alpha1 beta2 n2 gamma2

| (* 4 *)
lt_4 : forall alpha1 alpha2 beta1 beta2 n1 n2 gamma1 gamma2,
  alpha2 t2 < alpha1  $\rightarrow$ 
  [alpha1, beta1] t2 < beta2  $\rightarrow$ 
  gcons alpha1 beta1 n1 gamma1 t2 <
  gcons alpha2 beta2 n2 gamma2

| (* 5 *)
lt_5 : forall alpha1 alpha2 beta1 n1 n2 gamma1 gamma2,
  alpha2 t2 < alpha1  $\rightarrow$ 
  gcons alpha1 beta1 n1 gamma1 t2 <
  gcons alpha2 [alpha1, beta1] n2 gamma2

| (* 6 *)
lt_6 : forall alpha1 beta1 n1 n2 gamma1 gamma2,
  (n1 < n2)%nat  $\rightarrow$ 
  gcons alpha1 beta1 n1 gamma1 t2 <
  gcons alpha1 beta1 n2 gamma2

```

```
| (* 7 *)
lt_7 : forall alpha1 beta1 n1 gamma1 gamma2,
  gamma1 t2< gamma2 ->
  gcons alpha1 beta1 n1 gamma1 t2<
  gcons alpha1 beta1 n1 gamma2
```

**where** "o1 t2< o2" := (lt o1 o2): T2\_scope.

Seven constructors! In order to get accustomed with this definition, let us look at a small set of examples, covering all the constructors of `lt`.

**Example Ex1:**  $0 \ t2< \ \text{epsilon}_0$ .

**Proof.** constructor 1. **Qed.**

**Example Ex2:**  $\omega \ t2< \ \text{epsilon}_0$ .

**Proof.** info\_auto with T2. (\* uses lt\_1 and lt\_2 \*) **Qed.**

**Example Ex3:**  $\text{gcons } \omega \ 8 \ 12 \ 56 \ t2< \ \text{gcons } \omega \ 8 \ 12 \ 57$ .

**Proof.**

constructor 7; constructor 6; auto with arith.

**Qed.**

**Example Ex4:**  $\text{epsilon}_0 \ t2< \ [2,1]$ .

**Proof.**

apply lt\_2; auto with T2.

- apply lt\_6; auto with arith.

**Qed.**

**Example Ex5 :**  $[2,1] \ t2< \ [2,3]$ .

**Proof.**

constructor 3; auto with T2.

- constructor 6; auto with arith.

**Qed.**

**Example Ex6 :**  $\text{gcons } 1 \ 0 \ 12 \ \omega \ t2< \ [0, [2,1]]$ .

**Proof.**

constructor 4.

- constructor 1.

- constructor 2.

+ constructor 6; auto with arith.

+ constructor 1.

**Qed.**

**Example Ex7 :**  $\text{gcons } 2 \ 1 \ 42 \ \text{epsilon}_0 \ t2< \ [1, [2,1]]$ .

**Proof.**

constructor 5.

constructor 6; auto with arith.

**Qed.**

**Project 9.4** Write a tactic that solves automatically goals of the form  $(\alpha \text{ t2} < \beta)$ , where  $\alpha$  and  $\beta$  are closed terms of type T2.

## 9.4 Veblen normal form

**Definition 9.1** A term of the form  $\psi(\alpha_1, \beta_1) \times n_1 + \psi(\alpha_2, \beta_2) \times n_2 + \dots + \psi(\alpha_k, \beta_k) \times n_k$  is said to be in [Veblen] normal form if for every  $i < n$ ,  $\psi(\alpha_i, \beta_i) < \psi(\alpha_{i+1}, \beta_{i+1})$ , all the  $\alpha_i$  and  $\beta_i$  are in normal form, and all the  $n_i$  are strictly positive integers.

```

Inductive nf : T2 -> Prop :=
| zero_nf : nf zero
| single_nf : forall a b n,
  nf a ->
  nf b -> nf (gcons a b n zero)
| gcons_nf : forall a b n a' b' n' c',
  [a', b'] t2< [a, b] ->
  nf a -> nf b ->
  nf(gcons a' b' n' c')->
  nf(gcons a b n (gcons a' b' n' c')).

```

```

#[global] Hint Constructors nf : T2.

```

Let us look at some positive examples (we have to prove some inversion lemmas before proving counter-examples).

```

Lemma nf_fin i : nf (fin i).

```

**Proof.**

```

  destruct i.
  - auto with T2.
  - constructor 2; auto with T2.

```

**Qed.**

```

Lemma nf_omega : nf omega.

```

**Proof.** compute; auto with T2. **Qed.**

```

Lemma nf_epsilon0 : nf epsilon0.

```

**Proof.** constructor 2; auto with T2. **Qed.**

```

Lemma nf_epsilon : forall alpha, nf alpha -> nf (epsilon alpha).

```

**Proof.** compute; auto with T2. **Qed.**

```

Example Ex8: nf (gcons 2 1 42 epsilon0).

```

**Proof.**

```

  constructor 3; auto with T2.
  - apply Ex4.
  - apply nf_fin.
  - apply nf_fin.

```

**Qed.**

### 9.4.1 Length of a term

The notion of *term length* is introduced by Schütte as a helper for proving (at least) the *trichotomy* property and transitivity of the strict order `lt` on `T2`. These properties are proved by induction on length.

### 9.4.2 Trichotomy

*Trichotomy* is another name for the well-known property of decidable total ordering (like Standard Library's `Compare_dec.lt_eq_lt_dec`).

We first prove by induction on  $l$  the following lemma:

*From `Gamma0.Gamma0`*

```
Lemma tricho_aux (l: nat) : forall t t': T2,
  t2_length t + t2_length t' < l ->
  {t t2< t'} + {t = t'} + {t' t2< t}.
Definition lt_eq_lt_dec (t t': T2) : {t t2< t'}+{t = t'}+{t' t2< t}.
```

**Proof.**

```
eapply tricho_aux.
eapply Nat.lt_succ_diag_r.
```

**Defined.**

```
#[ global ] Instance compare_T2 : Compare T2 :=
fun (t1 t2 : T2) =>
  match lt_eq_lt_dec t1 t2 with
  | inleft (left _) => Lt
  | inleft (right _) => Eq
  | inright _ => Gt
  end.
```

```
Compute compare (gcons 2 1 42 epsilon0) [2,2].
```

```
= Lt
: comparison
```

With the help of `compare`, we get a boolean version of `nf` (being in Veblen normal form).

```
Fixpoint nfb (alpha : T2) : bool :=
  match alpha with
  zero => true
  | gcons a b n zero => andb (nfb a) (nfb b)
  | gcons a b n ((gcons a' b' n' c') as c) =>
    match compare [a', b'] [a, b] with
    Lt => andb (nfb a) (andb (nfb b) (nfb c))
    | _ => false
    end
  end.
```

```
Compute nfb (gcons 2 1 42 epsilon0).
```

```
= true
: bool
```

**Compute** nfb (gcons 2 1 42 (gcons 2 2 4 epsilon0)).

```
= false
: bool
```

## 9.5 Main functions on T2

### 9.5.1 Successor

The successor function is defined by structural recursion.

*From Gamma0.T2*

```
Fixpoint succ (a:T2) : T2 :=
  match a with zero => one
            | gcons zero zero n c => fin (S (S n))
            | gcons a b n c => gcons a b n (succ c)
  end.
```

### 9.5.2 Addition

Like for Cantor normal forms (see Sect. 4.1.9.3), the definition of addition in T2 requires comparison between ordinal terms.

*From Gamma0.Gamma0*

```
Fixpoint plus (t1 t2 : T2) {struct t1} : T2 :=
  match t1,t2 with
  | zero, y => y
  | x, zero => x
  | gcons a b n c, gcons a' b' n' c' =>
    (match compare (gcons a b 0 zero)
      (gcons a' b' 0 zero) with
    | Lt => gcons a' b' n' c'
    | Gt => gcons a b n (c + gcons a' b' n' c')
    | Eq => gcons a b (S(n+n')) c'
    end)
  end
where "alpha + beta" := (plus alpha beta): T2_scope.
```

**Example Ex7** : 3 + epsilon0 = epsilon0.

**Proof.** trivial. **Qed.**

### 9.5.3 The Veblen function $\phi$

The enumeration function of critical ordinals, presented in Sect. 8.8 on page 179, is recursively defined in type T2.

```

Definition phi (alpha beta : T2) : T2 :=
  match beta with
  zero => [alpha, beta]
  | [b1, b2] =>
    (match compare alpha b1
     with Datatypes.Lt => [b1, b2 ]
     | _ => [alpha,[b1, b2]]
     end)
  | gcons b1 b2 0 (gcons zero zero n zero) =>
    (match compare alpha b1
     with Datatypes.Lt =>
       [alpha, (gcons b1 b2 0 (fin n))]
     | _ => [alpha, (gcons b1 b2 0 (fin (S n)))]
     end)
  | any_beta => [alpha, any_beta]
  end.

```

**Example Ex8:**  $\text{phi } 1 (\text{succ } \text{epsilon}0) = [1, [1,0] + 1]$ .

**Proof.** reflexivity. Qed.

Despite its complexity, the function `phi` is well adapted to proofs by simplification or computation.

The relation between the constructor  $\psi$  and the function  $\phi$  is studied in [Sch77], and partially implemented in this development. *Please contribute!*

For instance, the following theorem states that, if  $\gamma$  is the sum of a limit ordinal  $\beta$  and a finite ordinal  $n$ , and  $\beta$  is a fixpoint of  $\phi(\alpha)$ , then  $\psi(\alpha, \gamma) = \phi_\alpha(\gamma + 1)$ .

```

Lemma phi_psi : forall beta gamma n,
  nf gamma ->
  limit_plus_fin beta n gamma ->
  phi alpha beta = beta ->
  [alpha, gamma] = phi alpha (succ gamma).

```

**Example Ex9 :**  $[\text{zero}, \text{epsilon } 2 + 4] = \text{phi } 0 (\text{epsilon } 2 + 5)$ .

**Proof.** trivial. Qed.

On the other hand,  $\phi$  can be expressed in terms of  $\psi$ .

```

Theorem phi_of_psi : forall a b1 b2,
  phi a [b1, b2] =
  if (lt_ge_dec a b1)
  then [b1, b2]
  else [a ,[b1, b2]].

```

**Example Ex10 :**  $\text{phi } \text{omega } [\text{epsilon}0, 5] = [\text{epsilon}0, 5]$ .

**Proof.** reflexivity. Qed.

**Project 9.5** Please study a way to pretty print ordinal terms in Veblen normal form (see Section 4.1.5 on page 77).

## 9.6 An ordinal notation for $\Gamma_0$

In order to consider type `T2` as an ordinal notation, we have to build an instance of class `ON` (See Definition page 52).

First, we define a type that contains only terms in Veblen normal form, and redefine `lt` and `compare` by delegation (see for comparison the construction of type `E0` in Sect. 4.1.7.1 on page 81).

**Module** `G0`.

**Definition** `LT` := restrict nf lt.

**Class** `G0` := mkg0 {vnf : T2; vnf\_ok : nfb vnf}.

**Definition** `lt` (alpha beta : G0) := T2.lt (@vnf alpha) (@vnf beta).

**#[ global ] Instance** `compare_G0` : Compare G0 :=  
 fun alpha beta => compare (@vnf alpha) (@vnf beta).

Then, we build an instance of class `ON`. function `compare` is correct.

**#[ global ] Instance** `lt_sto` : StrictOrder lt.

**Lemma** `lt_wf` : well\_founded lt.

**#[ global ] Instance** `Gamma0_comp`: Comparable lt compare.

**#[ global ] Instance** `Gamma0`: ON lt compare.

**End** `G0`.

**Remark 9.1** The proof of `lt_wf` has been written by Évelyne Contejean, using her library on the recursive path ordering (see also remark 4.4 on page 90).

**Project 9.6** Prove that `Epsilon0` (page 90) is a sub-notation system of `Gamma0`.

Prove that the implementations of `succ`, `+`,  `$\phi_0$` , etc. are compatible in both notation systems.

Note that a function `T1_inj` from `T1` to `T2` has already been defined. It may help to complete the task.

From `Gamma0.T2`

(\* injection from T1 \*)

```
Fixpoint T1_to_T2 (alpha :T1) : T2 :=
  match alpha with
  | T1.zero => zero
  | T1.cons a n b => gcons zero (T1_to_T2 a) n (T1_to_T2 b)
  end.
```

**Project 9.7** Prove that the notation system `Gamma0` is a correct implementation of the segment  $[0, \Gamma_0)$  of the set of countable ordinals.



## Part II

# Ackermann, Gödel, Peano ...



# Chapter 10

## General presentation (draft)

### 10.1 Introduction

This part contains comments, examples and exercises about Russel O'Connor's work on Gödel's first incompleteness theorem [G86]. O'Connor's work was published in 2005 [O'C05b], and released as a user-contribution of the Coq proof assistant. This work is now maintained by Coq community [CCK] volunteers and split into two projects: Goedel [O'C05a] and Hydra-battles [HBk].

The main reference to this work is Russel O'Connor's article [O'C05b], which we strongly encourage the reader to consult regularly.

It was the first computer verified proof of the essential incompleteness<sup>1</sup> of Peano arithmetic. The main reference to this work is Russel O'Connor's article [O'C05b], which we strongly encourage the reader to consult regularly.

Several reasons — enumerated below — led us to maintain and document this work in the framework of Coq community [CCK].

**To do 10.1** *Cite [Dow23].*

- Historical interest in Gödel's proof and its mechanizations, as shown by the abundant literature on this topic (for instance [Smu92, Hof99, CN04]).
- O'Connor's proof was written at the end of Coq V7, then rewritten at the beginning of Coq V8. Since then, Coq and its ecosystem evolved a lot (new styles, tactics, documentation tools, etc.). We think this evolution should benefit to the original proof-scripts and make their understanding easier.
- Finally, let us quote Efim Zelmanov.

“The purpose of a proof is *understanding*” [BBB<sup>+</sup>22]

We hope that underlying some points of the proof will make easier to understand this large and technical work.

---

<sup>1</sup>Todo: explain “essential”

For technical reasons — mainly in order to simplify the installation and use of its sub-libraries, we split the project into two main parts: Goedel [O’C05a] and Hydra-battles [HBk].

Some changes are made to the aforementioned libraries, mainly because of the recent evolution of Coq and its libraries. Nevertheless, the definitions, lemmas and theorems of the original contribution have been preserved in this new release.

*These maintenance and documentation jobs have just started, and will probably be long to complete. Help is welcome!*

## 10.2 File contents

All Russel O’Connor’s files have been stored in two directories, in order to simplify packages maintenance.

- `theories/goedel/`: Proofs which depend on `CoqPrime` package.
- `theories/ordinals/Ackermann/` : all the rest: definition of primitive recursive functions, first-order logic, Peano Arithmetic, Gödel’s encoding.
- Some additions we made: examples, exercices, new notations, etc., are stored in a specific directory `theories/ordinal/MoreAck/`.

### 10.2.1 The Ackermann sub-directory

The following main topics are studied in `theories/ordinals/Ackermann/`: The following list presents the main modules in a dependance-compatible order.

#### Primitive Recursive Functions

This topic is discussed in Chapter 11 on page 201.

- `Ackermann.extEqualNat`
  - `Ackermann.primRec`
  - `Ackermann.cPair`
  - `MoreAck.PrimRecExamples`
  - `MoreAck.Ack`
  - `MoreAck.AckNotPR`
- Ackermann function is not primitive recursive

#### First Order Logic

This part presents Coq definitions and basic properties of first-order languages and proofs.

- `Ackermann.fol`
- `Ackermann.folProp`
- `Ackermann.folProof`
- `Ackermann.model`

- Ackermann.code
- Ackermann.prLogic
- Ackermann.codeList
- Ackermann.codeFreeVar
- Ackermann.checkPrf
- Ackermann.wellFormed
- Ackermann.codeSubTerm
- Ackermann.codeSubFormula

**Natural Deduction** Thanks to the *Deduction Theorem*, we prove many lemmas about provability in first-order-logic, many of them can be considered as natural deduction rules.

- Ackermann.Deduction
- Ackermann.folLogic
- Ackermann.folLogic2
- Ackermann.folLogic3
- Ackermann.folReplace
- Ackermann.subProp
- Ackermann.subAll
- MoreAck.FolExamples

### Languages of Arithmetic

- Ackermann.Languages
- Ackermann.LNN
- Ackermann.LNT
- Ackermann.NN Axioms for Natural Numbers and basic properties.
- Ackermann.NNtheory
- Ackermann.PA Peano Arithmetic: axioms and first properties.
- Ackermann.LNN2LNT
- Ackermann.NN2PA
- Ackermann.PAtheory
- Ackermann.PAconsistent
- Ackermann.codePA
- Ackermann.codeNat2Term
- Ackermann.wConsistent
- Ackermann.expressible
- MoreAck.LNN\_Examples

### Modules dependent on CoqPrime

- `Goedel.PRrepresentable`
- `Goedel.fixPoint`
- `Goedel.codeSysPrf`
- `Goedel.rosser`
- `Goedel.goedel1`
- `Goedel.rosserPA`
- `Goedel.goedel2`

**To do 10.2** *Add information on recent developments on formal proofs of Gödel incompleteness theorems. Justify the decision of working on this development.*

### 10.3 Warning

Russel O’Connors contribution contains more than 42 KLoc. Since its construction, Coq, its libraries and recommended style have evolved a lot. We have just started to “modernize” this code. We apologize for provisional inconsistencies of presentation (code and documentation).

# Chapter 11

## Primitive recursive functions

### 11.1 Introduction

*Primitive recursive functions* are a small class of total arithmetic functions from  $\mathbb{N}^n$  to  $\mathbb{N}$ , for some  $n \in \mathbb{N}$ , corresponding to the expressive power of a simple imperative programming language without **while** loops, in which every program execution terminates. Please note that not all total  $n$ -ary recursive functions are primitive recursive (see for instance Sect. 11.7 on page 222).

Primitive recursive *relations* are boolean total functions whose *characteristic function* — obtained by mapping the returned value to 1 (**true**) or 0 (**false**) — is primitive recursive.

### 11.2 Mathematical definition

The traditional definition of primitive recursive functions is structured as an inductive definition in five rules: three base cases, and two recursive construction rules.

**zero** the natural number 0 is a primitive recursive function without arguments (in other words, a *constant*, or a *nullary* function).

**S** The successor function  $S : \mathbb{N} \rightarrow \mathbb{N}$  is primitive recursive.

**projections** For any  $i$  and  $n$ , such that  $0 < i \leq n$ , the projection  $\pi_{i,n} : \mathbb{N}^n \rightarrow \mathbb{N}$ , defined by  $\pi_{i,n}(x_1, x_2, \dots, x_n) = x_i$ , is primitive recursive.

**composition** For any  $n$  and  $m$ , if  $h : \mathbb{N}^m \rightarrow \mathbb{N}$ , and  $g_0, \dots, g_{m-1} : \mathbb{N}^n \rightarrow \mathbb{N}$  are primitive recursive of  $n$  arguments, then the function which maps any tuple  $(x_0, \dots, x_{n-1})$  to  $h(g_0(x_0, \dots, x_{n-1}), \dots, g_{m-1}(x_0, \dots, x_{n-1})) : \mathbb{N}^n \rightarrow \mathbb{N}$  is primitive recursive.

**primitive recursion** If  $g : \mathbb{N}^n \rightarrow \mathbb{N}$  and  $h : \mathbb{N}^{n+2} \rightarrow \mathbb{N}$  are primitive recursive,

then the function from  $\mathbb{N}^{n+1}$  into  $\mathbb{N}$  defined by

$$f(0, x_1, \dots, x_n) = g(x_1, \dots, x_n) \quad (11.1)$$

$$f(S(p), x_1, \dots, x_n) = h(p, f(p, x_1, \dots, x_n), x_1, \dots, x_n) \quad (11.2)$$

is primitive recursive.

Please note the use of dots:  $\dots$  in the definition above. Dots are not part of Gallina's syntax. Thus, the formal definition of the set of primitive recursive function will have to overcome this representation problem.

### 11.2.1 A few (informal) examples

Before playing with primitive recursive functions in Coq, let us get familiar with their mathematical definition, with the help of a few simple examples, which will be considered again as Coq terms in Section 11.4.4.1 on page 207.

#### 11.2.1.1 Projections

For instance, the projection  $\pi_{2,3}$  satisfies the equation  $\pi_{2,3}(x, y, z) = y$  for any natural numbers  $x, y$  and  $z$ .

#### 11.2.1.2 Constant functions

The *nullary* constant function which returns 0 is simply the **zero** construction.

If we want to consider the *unary* function which maps any natural number  $i$  to 0, we may built it through the *composition* construction, instanciated with  $n = 1$ ,  $m = 0$ , and  $h = \mathbf{zero}$ .

**Exercise 11.1** Let  $m$  and  $k$  be two natural numbers; please build the primitive recursive function which maps any tuple  $t \in \mathbb{N}^m$  to  $k$ .

#### 11.2.1.3 Addition on natural natural numbers

Addition may be defined by primitive recursion:

$$\begin{aligned} 0 + x_1 &:= x_1 \\ S x_0 + x_1 &:= S(x_0 + x_1) \end{aligned}$$

Both equations can be rewritten as follows:

$$\begin{aligned} 0 + x_1 &:= g(x_1) \\ S p + x_1 &:= h(p, p + x_1, x_1) \\ \mathbf{where} \quad g(x_1) &:= x_1 \\ \mathbf{and} \quad h(p, x, x_1) &:= S x \end{aligned}$$

It remains to show that  $g$  and  $h$  are primitive recursive, which is almost immediate:

- $g$  is the projection  $\pi_{1,1}$ ,
- $h$  is the composition (with  $n = 3$  and  $m = 1$ ) of the function **S** and the projection  $\pi_{2,3}$ .

#### 11.2.1.4 Multiplication on natural natural numbers

The following equations define the product of two natural numbers:

$$\begin{aligned} 0 \times x_1 &:= 0 \\ Sx_0 \times x_1 &:= (x_0 \times x_1) + x_1 \end{aligned}$$

This function is an instance of the primitive recursion scheme, with  $n = 1$ ,  $g$  is the constant unary function which returns 0 (see subsection 11.2.1.2 on the preceding page), and  $h$  the function defined by  $h(p, x, n_1) = x + n_1$ , which can be written as the composition of  $+$  and the projections  $\pi_{2,3}$  and  $\pi_{3,3}$  (the three of them being primitive recursive).

**Exercise 11.2** Build a primitive recursive definition of the factorial function, using the constructions of Section 11.2, addition and multiplication.

### 11.3 First look at the Ackermann library

We present here a formalization of primitive recursive functions, taken from Russel O’Connor’s formalization in Coq of Gödel’s incompleteness theorems [O’C05b].

A few additions and/or small changes (mainly notations, and adaptation to the continuously evolving practices of Coq development) have been made to O’Connor’s original library. Contributions (under the form of comments, new examples or exercises) are welcome!

O’Connor’s library on Gödel’s incompleteness theorems contains a little more than 45K lines of code. The part dedicated to primitive recursive functions and Peano arithmetic is 32K lines long and is originally structured in 38 modules. Thus, we propose a partial exploration of this library, through examples and exercises. Our additions to the original library — mainly examples and counter-examples —, are stored in the directory `theories/ordinals/MoreAck`.

In particular, the library `MoreAck.AckNotPR` contains the well-known proof that the Ackermann function is not primitive recursive (see Section 11.7 on page 222). Moreover, the library `Hydra.Hydra_Theorems` contains a proof that the length of an hydra battle (according to the initial replication factor) is not primitive recursive in general.

### 11.4 Abstract syntax for primitive recursive functions

The formal definition of primitive recursive functions lies in the library `Ackermann.primRec`, with preliminary definitions in `Ackermann.extEqualNat` and `Ackermann.misc`.

#### 11.4.1 Functions of arbitrary arity

The `Ackermann.extEqualNat` library allows us to consider primitive functions on `nat`, with any number of arguments, in curried form. This is made possible in by the following definition:

```

Fixpoint naryFunc (n : nat) : Set :=
  match n with
  | 0 => nat
  | S n => nat -> naryFunc n
  end.

```

For instance (naryFunc 1) is convertible to `nat -> nat` and (naryFunc 3) to `nat -> nat -> nat -> nat`.

*From MoreAck.PrimRecExamples.*

**Compute** naryFunc 3.

```

= nat -> nat -> nat -> nat
: Set

```

**Check** plus: naryFunc 2.

**Check** 42: naryFunc 0.

**Check** (fun n p q : nat => n \* p + q): naryFunc 3.

## 11.4.2 Extensional equality

Dependent types make it possible to define recursively extensional equality between functions of the same arity.

*From Ackermann.extEqualNat*

```

Fixpoint extEqual (n : nat) : forall (a b : naryFunc n), Prop :=
  match n with
  | 0 => fun a b => a = b
  | S p => fun a b => forall c, extEqual p (a c) (b c)
  end.

```

Module Ackermann.primRec defines and export the notation “ $f =_x g$ ” for “`extEqual n f g`”<sup>1</sup>

*From MoreAck.PrimRecExamples*

**Compute** extEqual 2.

```

= λ a b : naryFunc 2,
  ∀ x x0 : nat, a x x0 = b x x0
: naryFunc 2 -> naryFunc 2 -> Prop

```

**Example** extEqual\_ex1: (Nat.mul: naryFunc 2) =<sub>x</sub> fun x y => y \* x + x - x.

**Proof.**

```

(Nat.mul : naryFunc 2) =x
(λ x y : nat, y * x + x - x)

```

```

intros x y; cbn.

```

<sup>1</sup>in parsing mode, the provided  $f$  should be explicitly typed as (naryFunc  $n$ ).

```
x, y: nat
-----
x * y = y * x + x - x
```

Getting rid of the term  $x-x$ , we generate two easy-to-solve subgoals.

```
rewrite <- Nat.add_sub_assoc, Nat.sub_diag.
```

```
x, y: nat
-----
x * y = y * x + 0
-----
x, y: nat
-----
x ≤ x
```

```
- ring.
- apply le_n.
```

**Qed.**

### 11.4.3 Boolean predicates

Like arithmetic functions, arbitrary boolean predicates may have an arbitrary number of arguments. The dependent type (`naryRel n`), defined in the same way as `naryFunc`, is the type of  $n$ -ary functions from `nat` into `bool`.

*From `Ackermann.extEqualNat`*

```
Fixpoint naryRel (n : nat) : Set :=
  match n with
  | 0 => bool
  | S n => nat -> naryRel n
  end.
```

```
Definition ltBool (a b : nat) : bool :=
  if le_lt_dec b a then false else true.
```

```
Definition leBool (a b : nat) : bool :=
  if le_lt_dec a b then true else false.
```

*From `Ackermann.extEqualNat`*

**Check** leBool : naryRel 2.

```
leBool : naryRel 2
       : naryRel 2
```

**Compute** leBool 4 5.

```
= true
: bool
```

**Compute** charFunction 2 leBool 4 5.

```
= 1
: nat
```

**Compute** charFunction 2 ltBool 7 7.

```
= 0
: nat
```

#### 11.4.4 A Data-type for Primitive Recursive Functions

O'Connor's formalization of primitive recursive functions takes the form of two mutually inductive dependent data types, each constructor of which is associated with one of these rules. These two types are `(PrimRec n)` (primitive recursive functions of  $n$  arguments), and `(PrimRecs n m)` ( $m$ -tuples of primitive recursive functions of  $n$  arguments).

**Remark 11.1** The `PrimRec` type family is indeed a kind of programming language for writing primitive recursive functions. The link to the mathematical notion of such functions will be established in Section 11.4.5 when we give a semantics which maps any term of type `(PrimRec n)` to a function of type `(naryFunc n)`.

*From Ackermann.primRec.*

```
Inductive PrimRec : nat -> Set :=
| succFunc : PrimRec 1
| zeroFunc : PrimRec 0
| projFunc : forall n m : nat, m < n -> PrimRec n
| composeFunc :
  forall (n m : nat) (g : PrimRecs n m) (h : PrimRec m),
    PrimRec n
| primRecFunc :
  forall (n : nat) (g : PrimRec n) (h : PrimRec (S (S n))),
    PrimRec (S n)
with PrimRecs : nat -> nat -> Set :=
| PRnil : forall n : nat, PrimRecs n 0
| PRcons : forall n m : nat, PrimRec n -> PrimRecs n m ->
  PrimRecs n (S m).
```

**Remark 11.2** Beware of the conventions used in the `primRec` library! The constructor `(projFunc n m)` is associated with the projection  $\pi_{n-m,n}$  and *not*  $\pi_{n,m}$ . For instance, the projection  $\pi_{2,5}$  defined by  $\pi_{2,5}(a,b,c,d,e) = b$  corresponds to the term `(projFunc 5 3 H)`, where `H` is a proof of  $3 < 5$ . This fact is reported in the comments of `primRec.v`. We presume that this convention makes it easier to define the evaluation function `(evalProjFunc n)` (see the next sub-section). Trying the other convention is left as an exercise.

In order to make the terms of type `Primrec n` more readable, we introduce some notations, mainly inspired by Coq's standard library's notations for vectors.

```
Module PRNotations.
Declare Scope pr_scope.
Delimit Scope pr_scope with pr.
```

## 11.4. ABSTRACT SYNTAX FOR PRIMITIVE RECURSIVE FUNCTIONS 207

```
Notation "h :: t" := (PRcons _ _ h t) (at level 60, right associativity)
  : pr_scope.
Notation "[ x ]" := (PRcons _ _ x (PRnil _)) : pr_scope.

Notation "[ x ; y ; .. ; z ]" :=
  (PRcons _ _ x (PRcons _ _ y .. (PRcons _ _ z (PRnil _)) ..)) : pr_scope.

Notation PRcomp f v := (composeFunc _ _ v f).

Notation PRrec f0 fS := (primRecFunc _ f0 fS).

(** Popular projections *)
Notation pil_1 := (projFunc 1 0 (le_n 1)).

Notation pil_2 := (projFunc 2 1 (le_n 2)).
Notation pi2_2 := (projFunc 2 0 (le_S 1 1 (le_n 1))).

Notation pil_3 := (projFunc 3 2 (le_n 3)).
Notation pi2_3 := (projFunc 3 1 (le_S 2 2 (le_n 2))).
Notation pi3_3 := (projFunc 3 0 (le_S 1 2 (le_S 1 1 (le_n 1)))).

End PRNotations.
```

### 11.4.4.1 Examples

Let us show how the functions described in 11.2.1 can be described by terms of type “PrimRec \_”.

*From MoreAck.primRecExamples.*

```
Module MoreExamples.

(** The unary constant function which returns 0 *)
Definition cst0 : PrimRec 1 := (PRcomp zeroFunc (PRnil _))%pr.

(** The unary constant function which returns i *)
Fixpoint cst (i: nat) : PrimRec 1 :=
  match i with
  0 => cst0
  | S j => (PRcomp succFunc [cst j])%pr
end.

Compute cst 7.
```

```

= PRcomp succFunc
  [PRcomp succFunc
    [PRcomp succFunc
      [PRcomp succFunc
        [PRcomp succFunc
          [PRcomp succFunc
            [PRcomp succFunc
              [PRcomp succFunc
                [PRcomp zeroFunc (PRnil 1)]]]]]]]]]]%pr
: PrimRec 1

```

```

(** Addition *)

```

```

Definition plus : PrimRec 2 :=
  (PRrec pi1_1 (PRcomp succFunc [pi2_3]))%pr.

```

```

(** Multiplication *)

```

```

Definition mult : PrimRec 2 :=
  PRrec cst0
  (PRcomp plus [pi2_3; pi3_3])%pr.

```

```

(** Factorial function *)

```

```

Definition fact : PrimRec 1 :=
  (PRrec
    (PRcomp succFunc [zeroFunc])
    (PRcomp mult [pi2_2; PRcomp succFunc [pi1_2]]))%pr.

```

```

End MoreExamples.

```

### 11.4.5 A little bit of semantics

Inhabitants of type  $(\text{PrimRec } n)$  are not Coq functions like `Nat.mul`, `Arith.Factorial.fact`, etc. but terms of an abstract syntax for the language of primitive recursive functions. The bridge between this language and the world of usual functions is an interpretation function  $(\text{evalPrimRec } n)$  of type  $\text{PrimRec } n \rightarrow \text{naryFunc } n$ . This function is defined by mutual recursion, together with the function  $(\text{evalPrimRecS } n \ m)$  of type  $(\text{PrimRecs } n \ m \rightarrow \text{Vector.t } (\text{naryFunc } n) \ m)$ .

Both functions are mutually defined through dependent pattern matching. We advise the readers who would feel uneasy with dependent types to consult Adam Chlipala's *cpdt* book [Chl11]. We invite the reader to look also at the helper functions in `Ackermann.primRec`, namely `evalConstFunc`, `evalProjFunc`, `evalComposeFunc`, and `evalPrimRecFunc`, etc.

```

Fixpoint evalPrimRec (n : nat) (f : PrimRec n) {struct f} :
  naryFunc n :=
  match f in (PrimRec n) return (naryFunc n) with
  | succFunc => S
  | zeroFunc => 0
  | projFunc n m pf => evalProjFunc n m pf
  | composeFunc n m l f =>
    evalComposeFunc n m (evalPrimRecs _ _ l) (evalPrimRec _ f)

```

```

| primRecFunc n g h =>
  evalPrimRecFunc n (evalPrimRec _ g) (evalPrimRec _ h)
end

with evalPrimRecs (n m : nat) (fs : PrimRecs n m) {struct fs} :
Vector.t (naryFunc n) m :=
  match fs in (PrimRecs n m) return (Vector.t (naryFunc n) m) with
  | PNil a => Vector.nil (naryFunc a)
  | PRcons a b g gs =>
    Vector.cons _ (evalPrimRec _ g) _ (evalPrimRecs _ _ gs)
end.

```

**Notation** `PReval f` := (evalPrimRec \_ f).

**Notation** `PRevalN fs` := (evalPrimRecs \_ \_ fs).

(\* [p] is a correct implementation of [f] in [PrimRec n] \*)

**Definition** `PRcorrect {n:nat}{p:PrimRec n}(f: naryFunc n) :=`  
`PReval p =x= f.`

#### 11.4.5.1 A few tests

The following examples show that the functions `evalPrimRec` and `evalPrimRecs` behave well w.r.t. Coq's reduction rules. They can also be considered as elementary tests of our definitions of `cst0`, `cst`, `plus`, `mult` and `fact`.

*From MoreAck.PrimRecExamples.*

**Compute** `PReval pi2_3 10 20 30.`

```

= 20
: nat

```

**Compute** `Vector.map (fun f => f 10 20 30) (PRevalN [pi2_3; pi1_3]%pr).`

```

= [20; 10]
: t nat 2

```

**Compute** `PReval cst0 42.`

```

= 0
: nat

```

**Compute** `PReval (cst 7) 19.`

```

= 7
: nat

```

**Compute** `PReval plus 9 4.`

```

= 13
: nat

```

**Compute** PReval mult 9 4.

```
= 36
: nat
```

**Compute** PReval fact 5.

```
= 120
: nat
```

### 11.4.5.2 Correctness proofs

It is now time to *prove* that our functions `cst0`, `cst`, `plus`, `mult` and `fact` are correct implementations in `PrimRec` of the mathematical functions we consider.

**Lemma** `cst0_correct` : PRcorrect `cst0` (fun \_ => 0).

**Proof.** `intros ?; reflexivity. Qed.`

**Lemma** `cst_correct` (k:nat) : PRcorrect (cst k) (fun \_ => k).

**Proof.**

```
induction k as [| k IHk]; simpl; intros p.
- reflexivity.
- cbn; now rewrite IHk.
```

**Qed.**

**Lemma** `plus_correct`: PRcorrect `plus` `Nat.add`.

**Proof.**

```
intros n; induction n as [| n IHn].
- intro; reflexivity.
- intro p; cbn in IHn |- *; now rewrite IHn.
```

**Qed.**

**Remark** `mult_eqn1` n p:

```
PReval mult (S n) p =
  PReval plus (PReval mult n p) p.
```

**Proof.** `reflexivity. Qed.`

**Lemma** `mult_correct`: PRcorrect `mult` `Nat.mul`.

**Proof.**

```
intro n; induction n as [| n IHn].
- intro p; reflexivity.
- intro p; rewrite mult_eqn1, (IHn p) , plus_correct. cbn. ring.
```

**Qed.**

**Lemma** `fact_correct` : PRcorrect `fact` `Coq.Arith.Factorial.fact`.

(\* ... \*)

## 11.5 Proving that a given Coq arithmetic function is primitive recursive

The example in the preceding section clearly shows that, in order to prove that a given arithmetic function (defined in Gallina as usual) is primitive recursive, trying to *type* a term of type `(PrimRec n)` is not a good method, since such terms may be too large, even for simple arithmetic functions. The method proposed in the library `Ackermann.primRec` is the following one:

1. Define a type corresponding to the statements of the form "the  $n$ -ary function  $f$  is primitive recursive".
2. Prove handy lemmas which may help to prove that a given function is primitive recursive. These lemmas can be considered as a way to build *silently* large terms of type `(PrimRec n)` in intermediate steps of the proof. More, we can associate *tactics* with these lemmas.

### 11.5.1 The predicate `isPR`

Let  $f$  be an arithmetic function of arity  $n$ . We say that  $f$  is primitive recursive if  $f$  is **extensionally** equal to the interpretation of some term of type `PrimRec n`.

From `Ackermann.primRec`.

```
Class isPR (n : nat) (f : naryFunc n) : Set :=
  is_pr : {p : PrimRec n | extEqual n (PReval p) f}.
```

```
Definition fun2PR {n:nat}(f: naryFunc n)
  {p: isPR _ f}: PrimRec n := proj1_sig p.
```

```
Class isPRrel (n : nat) (R : naryRel n) : Set :=
  is_pr_rel: isPR n (charFunction n R).
```

The library `primRec` contains a large catalogue of lemmas allowing to prove statements of the form `(isPR n f)`. We won't list all these lemmas here, but give a few examples of how they may be searched, then applied.

**Remark 11.3** In the library `primRec`, all these lemmas are opaque (registered with `Qed`). Thus they do not allow the user to look at the witness of a proof of a `isPR` statement. It may be useful to make transparent all the instances of `isPR` in the `Ackermann` and `goedel` libraries.

#### 11.5.1.1 Elementary proofs of `isPR` statements

Simple proofs of statements `(isPR n f)` may be just applications of the constructor `is_pr`, often thanks to the tactic call `exists x` where  $x$  is some (hopefully) correct term of type `(PrimRec n)`.

Let us show a few examples from `Ackermann.MoreAck.PrimRecExamples.v2`.

---

<sup>2</sup>Some of them are also in `Ackermann.primRec`.

```
#[export] Instance zeroIsPR : isPR 0 0.
```

```
Proof.
```

```
exists zeroFunc.
```

```
PReval zeroFunc =x= 0
```

```
cbn.
```

```
0 = 0
```

```
reflexivity.
```

```
Qed.
```

```
#[export] Instance succIsPR : isPR 1 S.
```

```
Proof.
```

```
exists succFunc; cbn; reflexivity.
```

```
Qed.
```

```
#[export] Instance addIsPR : isPR 2 Nat.add.
```

```
Proof. exists plus; intros n p; apply plus_correct. Qed.
```

Projections are proved primitive recursive, case by case (many examples in `Ackermann.primRec`). *Please notice again that the name of the projection follows the mathematical tradition, whilst the arguments of `projFunc` use another convention (cf remark 11.2 on page 206).*

```
#[export] Instance pi2_5IsPR : isPR 5 (fun a b c d e => b).
```

```
Proof.
```

```
assert (H: 3 < 5) by auto.
```

```
exists (projFunc 5 3 H).
```

```
cbn; reflexivity.
```

```
Qed.
```

Please note that the projection  $\pi_{1,1}$  is just the identity on `nat`, and is realized by `(projFunc 1 0)` (see Sect. 11.4.4.1 on page 207).

*From `Ackermann.primRec`.*

```
#[export] Instance idIsPR : isPR 1 (fun x : nat => x).
```

```
Proof.
```

```
exists pi1_1; cbn; reflexivity.
```

```
Qed.
```

### 11.5.1.2 The predecessor (total) function

The predecessor function is defined by the following equations:

$$\begin{aligned} \text{pred}(0) &= 0 \\ \text{pred}(1 + n) &= n = \pi_{1,2}(n, \text{pred}(n)) \end{aligned}$$

This definition is easy to translate into a term of type `PrimRec 1`.

**Definition** `xpred` := primRecFunc 0 zeroFunc pil\_2.

**Compute** evalPrimRec 1 xpred 10.

```
= 9
: nat
```

`#[export]` **Instance** `predIsPR` : isPR 1 Nat.pred.

**Proof.**

```
exists xpred; intro n; induction n; now cbn.
```

**Qed.**

### 11.5.1.3 Using function composition

Let us look at the proof that any constant  $n$  of type `nat` has type `(PR 0)` (lemma `const0_NIsPR` of `primRec`). We carry out a proof by induction on  $n$ , the base case of which is already proven. Now, let us assume  $n$  is `PR 0`, and call  $(x : \text{PrimRec } 0)$  its “realizer”. Thus we would like to compose this constant function with the unary successor function.

This is exactly the role of the function `(composeFunc 0 1)`. Here is a quite simple proof of `const0_NIsPR`.

*From MoreAck.PrimRecExamples.*

`#[export]` **Instance** `const0_NIsPR n` : isPR 0 n.

**Proof.**

```
induction n as [ | n [x Hx]].
```

```
isPR 0 0
```

```
n: nat
```

```
x: PrimRec 0
```

```
Hx: PReval x =x= n
```

```
isPR 0 n.+1
```

- apply zeroIsPR.
- exists (composeFunc \_\_ [x] succFunc)%pr; cbn in \*; intros;
 now rewrite Hx.

**Qed.**

### 11.5.1.4 Another proof that `Nat.add` is primitive recursive

We have already proven that `Nat.add` is primitive recursive. The following alternative proof, — more detailed —, shows how to search and apply lemmas from the Ackermann library.

Let us look for some lemma which could help to prove a given recursive arithmetic binary function is primitive recursive.

**Search** (isPR 2 (fun \_\_ => nat\_rec \_\_ \_\_)).

```

ind1ParamIsPR:
  ∀ f : nat → nat → nat → nat,
  isPR 3 f
  → ∀ g : nat → nat,
    isPR 1 g
    → isPR 2
      (λ a b : nat,
        nat_rec (λ _ : nat, nat) (g b)
          (λ x y : nat, f x y b) a)

```

Good! Let us express addition in terms of `nat_rec`.

```

Definition add' x y :=
  nat_rec (fun n : nat => nat)
    y
    (fun z t => S t)
    x.

```

```

Lemma add'_ok:
  extEqual 2 add' Nat.add.

```

**Proof.**

```

  intro x; induction x; cbn; auto.
  intro y; cbn; now rewrite <- (IHx y).

```

**Qed.**

The lemma `Ackermann.isPRextEqual` tells us that if a function  $g$  is extensionally equal to a primitive recursive function, then  $g$  is primitive recursive too.

**Check** `isPRextEqual`.

```

isPRextEqual
  : ∀ (n : nat) (f g : naryFunc n),
    isPR n f → f =x= g → isPR n g

```

Let us start our proof.

```

#[export] Instance addIsPR' : isPR 2 Nat.add.

```

**Proof.**

```

isPR 2 Nat.add

```

```

  apply isPRextEqual with add'.

```

```

isPR 2 add'

```

```

add' =x= Nat.add

```

```

-

```

```

isPR 2 add'

```

```

unfold add'; apply ind1ParamIsPR.

```

```
isPR 3 (λ _ y _ : nat, y.+1)
```

```
isPR 1 (λ b : nat, b)
```

We already proved that `S` is PR 1, but we need to consider a function of three arguments, which ignores its first and third arguments. Fortunately, the library `primRec` already contains lemmas adapted to this kind of situation.

```
+
```

```
isPR 3 (λ _ y _ : nat, y.+1)
```

```
Search (isPR 1 _ -> isPR 3 _).
```

```
filter001IsPR:
```

```
  ∀ g : nat → nat,  
  isPR 1 g → isPR 3 (λ _ _ c : nat, g c)
```

```
filter100IsPR:
```

```
  ∀ g : nat → nat,  
  isPR 1 g → isPR 3 (λ a _ _ : nat, g a)
```

```
filter010IsPR:
```

```
  ∀ g : nat → nat,  
  isPR 1 g → isPR 3 (λ _ b _ : nat, g b)
```

```
compose3_1IsPR:
```

```
  ∀ f : nat → nat → nat → nat,  
  isPR 3 f  
  → ∀ g : nat → nat,  
  isPR 1 g → isPR 3 (λ x y z : nat, g (f x y z))
```

```
isPR 3 (λ _ y _ : nat, y.+1)
```

```
  apply filter010IsPR, succIsPR.
```

Thus, our first subgoal is easily solved. The rest of the proof is just an application of already proven lemmas.

```
+ apply idIsPR.
```

```
- apply add'_ok.
```

```
Qed.
```

**Exercise 11.3** A few lemmas similar to `filter010IsPR`, also shown in the `primRec` library help the user to control the arity of functions. Thus, the reader may look at them, and invent h.er.is simple examples of application.

### 11.5.1.5 More examples

The following proof decomposes the `double` function as the composition of multiplication with the identity and the constant function which returns 2. *Note that the lemma `const1_NIsPR` considers this function as an unary function (unlike `const0_NIsPR`).*

**Definition** `double` (`n:nat`) := 2 \* n.

`#[export]` **Instance** `doubleIsPR` : isPR 1 double.

**Proof.**

`unfold double; apply compose1_2IsPR.`

```
isPR 1 (λ _ : nat, 2)
```

```
isPR 1 (λ x : nat, x)
```

```
isPR 2 Init.Nat.mul
```

- `apply const1_NIsPR.`
- `apply idIsPR.`
- `apply multIsPR.`

**Qed.**

**Exercise 11.4** Prove that the following functions are primitive recursive.

**Fixpoint** `exp n p` :=  
`match p with`  
`0 => 1`  
`| S m => exp n m * n`  
`end.`

**Fixpoint** `tower2 n` :=  
`match n with`  
`0 => 1`  
`| S p => exp 2 (tower2 p)`  
`end.`

**Hint:** You may have to look again at the lemmas of the library `Ackermann.primRec` if you meet some difficulty. You may start this exercise with the file `exercises/primrec/MorePRExamples.v`.

## 11.5.2 More advanced examples

### 11.5.2.1 The minimum of two natural numbers

Let  $a$  and  $b$  be two natural numbers. The *minimum* of  $a$  and  $b$  is  $a$  if  $a \leq b$ , otherwise  $b$ .

Thus, we propose the following definition:

```
Let min_alt (a b: nat) : nat :=  

  (charFunction 2 leBool a b) * a +  

  (charFunction 2 ltBool b a) * b.
```

Here is a sketch of proof that standard library's `min` is primitive recursive. The reader is kindly invited to fill the missing steps.

```

Lemma min_alt_correct : extEqual 2 min_alt Nat.min.
Proof.
(* ... *)
#[local] Instance minPR_PR : isPR 2 min_alt.
Proof.
(* ... *)
#[export] Instance minIsPR : isPR 2 Nat.min.
Proof.
  destruct minPR_PR as [f Hf].
  exists f; eapply extEqualTrans with (1:= Hf).
  apply min_alt_correct.
Qed.

```

**Exercise 11.5** Write a simple and readable proof that the Fibonacci function is primitive recursive.

```

Fixpoint fib (n:nat) : nat :=
  match n with
  | 0 => 1
  | 1 => 1
  | S ((S p) as q) => fib q + fib p
  end.

```

**Hint:** You may use as a helper the function which computes the pair  $(\text{fib}(n+1), \text{fib}(n))$ . Library `Ackermann.cPair` contains the definition of the encoding of  $\mathbb{N}^2$  into  $\mathbb{N}$ , and the proofs that the associated constructor and projections are primitive recursive.

Please find here some definitions and lemmas you may use in order to solve this exercise (non-exhaustive list).

```

Import LispAbbreviations.
Check cPair.

```

```

cPair
  : nat -> nat -> nat

```

```

Print car.

```

```

Notation car := cPairPil

```

```

Check car.

```

```

car
  : nat -> nat

```

```

Check cdr.

```

```

cdr
  : nat -> nat

```

```

Search cPair isPR.

```

```

cPairIsPR: isPR 2 cPair

```

```

Search car isPR.

```

```

cPairPilIsPR: isPR 1 car

```

**Search** car cdr cPair.

```
cPairProjections:
  forall a : nat, cPair (car a) (cdr a) = a
```

You may start this exercise with the file `exercises/primrec/FibonacciPR.v`.

See also the chapter 15 on Gödel's encodings.

### Exercise 11.6 (The integer square root)

1) Please consider the following specification of the function `boundedSearch` defined in `Ackermann.primRec`.

**Check** `boundedSearch`.

```
boundedSearch
  : naryRel 2 -> nat -> nat
```

**Search** `boundedSearch`.

```
boundSearchIsPR:
  forall P : naryRel 2,
  isPRrel 2 P -> isPR 1 (boundedSearch P)

boundedSearch1:
  forall (P : naryRel 2) (b x : nat),
  x < boundedSearch P b -> P b x = false

boundedSearch2:
  forall (P : naryRel 2) (b : nat),
  boundedSearch P b = b \ /
  P b (boundedSearch P b) = true
```

Prove the following lemmas.

**Lemma** `boundedSearch3` :

```
forall (P : naryRel 2) (b : nat), boundedSearch P b <= b.
```

**Lemma** `boundedSearch4` :

```
forall (P : naryRel 2) (b : nat),
  P b b = true ->
  P b (boundedSearch P b) = true.
```

2) Let us consider the following definition of the relation “ $r$  is the integer square root of  $n$ ”.

**Definition** `isqrt_spec`  $n r := r * r \leq n < r.+1 * r.+1$ .

Prove that the function which returns the integer square root of any natural number is primitive recursive (you may use the function `boundedSearch` for this purpose).

You may start this exercise with the file `exercises/primrec/isqrt.v`.

## 11.6 Proofs by induction over all primitive recursive functions

Let us consider the following theorem (see for instance [Pla13]).

There exists at least a total arithmetic function, *e.g.* the *Ackermann function*, which is not primitive recursive.

We can prove this theorem in three successive steps:

- Define Ackermann function in Gallina.
- Define and prove a property shared by any primitive recursive functions.
- Prove that Ackermann function does not satisfy this property.

We show how to adapt the classic proof (see for instance [Pla13]) to the constraints of Gallina. We hope this formal proof is a nice opportunity to explore the treatment of primitive recursive functions by R. O'Connor, and to play with dependent types.

### 11.6.1 Ackermann function

Ackermann function is traditionally defined as a function from  $\mathbb{N} \times \mathbb{N}$  into  $\mathbb{N}$ , through three equations:

$$\begin{aligned} A(0, n) &= n + 1 \\ A(m + 1, 0) &= A(m, 1) \\ A(m + 1, n + 1) &= A(m, A(m + 1, n)) \end{aligned}$$

Let us try to define this function in Coq (in curried form).

**Fail**

```
Fixpoint Ack (m n : nat) : nat :=
  match m, n with
  | 0, n => S n
  | m.+1, 0 => Ack m 1
  | m0.+1, p.+1 => Ack m0 (Ack m p)
  end.
```

The command has indeed failed with message:  
Cannot guess decreasing argument of `fix`.

A possible workaround is to make `m` be the decreasing argument, and define — within `m`'s scope — a local helper function which computes `(Ack m n)` for any `n`. This way, both functions `Ack` and `Ackm` have a (structurally) strictly decreasing argument.

**Module Alt.**

```
Fixpoint Ack (m n : nat) : nat :=
```

```

match m with
| 0 => n.+1
| p.+1 => let fix Ackm (n : nat) :=
            match n with
            | 0 => Ack p 1
            | S q => Ack p (Ackm q)
            end
          in Ackm n
end.

```

**Compute** Ack 3 2.

```

= 29
: nat

```

**End Alt.**

We preferred to define a variant which uses explicitly the functional `iterate`, where  $(\text{iterate } f\ n)$  is the  $n$ -th iteration of  $f$ <sup>3</sup>. It makes it possible to apply a few lemmas proved in `Prelude.Iterates`, for instance about the monotony of the  $n$ -th iterate of a given function.

*From `Prelude.Iterates`.*

```

Fixpoint iterate {A:Type}(f : A -> A) (n: nat)(x:A) :=
  match n with
  | 0 => x
  | S p => f (iterate f p x)
  end.

```

```

Lemma iterate_le_n_Sn (f: nat -> nat):
  (forall x, x <= f x) ->
  forall n x, iterate f n x <= iterate f (S n) x.

```

Thus, our definition of the Ackermann function is as follows:

*From `MoreAck.Ack`.*

```

Fixpoint Ack (m:nat) : nat -> nat :=
  match m with
  | 0 => S
  | n.+1 => fun k => iterate (Ack n) k.+1 1
  end.

```

**Compute** Ack 3 2.

```

= 29
: nat

```

**Exercise 11.7** The file `MoreAck.Ack` presents two other definitions of the Ackermann functions based on the lexicographic ordering on  $\mathbb{N} \times \mathbb{N}$ . Prove that the four functions are extensionally equal.

<sup>3</sup>Please do not mistake `iterate` (*i.e.* `Prelude.iterate`) for the monomorphic `primRec.iterate`, which does not share the same order of arguments.

### 11.6.1.1 First properties of the Ackermann function

The three first lemmas make us sure that our function `Ack` satisfies the “usual” equations.

**Lemma** `Ack_0` : `Ack 0 = S`.

**Proof** `refl_equal`.

**Lemma** `Ack_S_0 m` : `Ack m.+1 0 = Ack m 1`.

**Proof**. `reflexivity`. **Qed**.

**Lemma** `Ack_S_S` : `forall m p,`

`Ack m.+1 p.+1 = Ack m (Ack m.+1 p)`.

**Proof**. `reflexivity`. **Qed**.

The order of growth of the Ackermann function w.r.t. its first argument is illustrated by the following equalities.

**Lemma** `Ack_1_n n` : `Ack 1 n = n.+2`.

**Lemma** `Ack_2_n n` : `Ack 2 n = 2 * n + 3`.

**Lemma** `Ack_3_n n` : `Ack 3 n = exp2 n.+3 - 3`.

**Lemma** `Ack_4_n n` : `Ack 4 n = hyper_exp2 n.+3 - 3`.

**Remark 11.4** The statements above can be rewritten in a more uniform way:

For  $m \in 1..4$ ,  $\text{Ack } m \ n = f_m(n + 3) - 3$ , where

$$f_1(n) = n + 2$$

$$f_2(n) = n \times 2$$

$$f_3(n) = 2^n$$

$$f_4(n) = 2^{2^{\dots^2}} \quad (n \text{ levels})$$

An important property of the Ackermann function helps us to overcome the difficulty raised by nested recursion, by climbing up the hierarchy `Ack n _` ( $n \in \mathbb{N}$ ).

*From `MoreAck.Ack`.*

**Lemma** `nested_Ack_bound k m n` :

`Ack k (Ack m n) <= Ack (2 + max k m) n`.

Please note also that for any given  $n$ , the unary function `(Ack n)` is primitive recursive.

*From `MoreAck.AckNotPR`.*

`#[export] Instance Ackn_IsPR (n: nat) : isPR 1 (Ack n)`.

**Proof**.

`induction n`.

## 11.7 Ackermann function is not primitive recursive

In order to prove that `Ack` (considered as a function of two arguments) is not primitive recursive, the usual method consists in two steps:

1. Prove that for any primitive recursive function  $f : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$ , there exists some natural number  $n$  depending on  $f$ , such that, for any  $x$  and  $y$ ,  $f\ x\ y \leq \text{Ack}\ n\ (\max\ x\ y)$  (we say that  $f$  is “majorized” by `Ack`).
  
2. Show that `Ack` fails to satisfy this property.

First, we have to prove that any primitive function of two arguments is majorized by `Ack`. Let us look at the induction principles generated for the types `PrimRec n`.

If we look at the inductive definition of primitive recursive functions, page 206, it is obvious that a proof by induction on the construction of primitive recursive functions must consider functions of any arity.

*From `Ackermann.primRec`.*

```

Scheme PrimRec_PrimRecs_ind := Induction for PrimRec Sort Prop
  with PrimRecs_PrimRec_ind := Induction for PrimRecs Sort Prop.

Arguments PrimRec_PrimRecs_ind P P0 : rename.
Arguments PrimRecs_PrimRec_ind P P0 : rename.
Check PrimRec_PrimRecs_ind.

```

```

PrimRec_PrimRecs_ind
  : ∀ (P : ∀ n : nat, PrimRec n → Prop) (P0 : ∀ n
                                         n0 : nat,
                                         PrimRecs
                                         n n0
                                         → Prop),
    P 1 succFunc
  → P 0 zeroFunc
  → (∀ (n m : nat) (l : m < n),
      P n (projFunc n m l))
  → (∀ (n m : nat) (g : PrimRecs n m),
      P0 n m g
      → ∀ h : PrimRec m,
         P m h → P n (PRcomp h g))
  → (∀ (n : nat) (g : PrimRec n),
      P n g
      → ∀ h : PrimRec (S (S n)),
         P (S (S n)) h
         → P (S n) (PRrec g h))
  → (∀ n : nat, P0 n 0 (PRnil n))
  → (∀ (n m : nat) (p : PrimRec n),
      P n p
      → ∀ p0 : PrimRecs n m,
         P0 n m p0
         → P0 n (S m) (p :: p0)%pr)
  → ∀ (n : nat) (p : PrimRec n),
     P n p

```

Please note that, in order to prove a property shared by any primitive recursive function of, say, arity 2, this induction scheme leads you to consider an extension of the considered property to primitive recursive function of any arity.

Thus the lemma we will have to prove is the following one:

For any  $n$ , and any primitive recursive function  $f$  of arity  $n$ , there exists some natural number  $q$  such that the following inequality holds:

$$\forall x_1, \dots, x_n, f(x_1, \dots, x_n) \leq \text{Ack}(q, \max(x_1, \dots, x_n))$$

But dots don't belong to Gallina's syntax! So, we may use Coq's vectors for denoting arbitrary tuples.

First, we extend `max` to vectors of natural numbers (using the notations of module `VectorNotations` and some more definitions from `Prelude.MoreVectors`). So, `(t A n)` is the type of vectors of  $n$  elements of type  $A$ , and the constants `cons`, `nil`, `map`, etc., refer to vectors and not to lists. Likewise, the notation `x::v` is an abbreviation for `VectorDef.cons x _ v`.

```

Fixpoint max_v {n:nat} (v: Vector.t nat n) : nat :=
  match v with
  | nil => 0
  | cons x t => max x (max_v t)
  end.

```

```

Lemma max_v_2 : forall x y, max_v (x::y::nil) = max x y.

```

```

Lemma max_v_lub : forall n (v: t nat n) y,
  (forall (fun x => x <= y) v) ->
  max_v v <= y.

```

```

Lemma max_v_ge : forall n (v: t nat n) y,
  In y v -> y <= max_v v.

```

We have also to convert any application ( $f x_1 x_2 \dots x_n$ ) into an application of a function to a single argument: the vector of all the  $x_i$ s. This is already defined in Library Ackermann.primRec.

```

Fixpoint evalList (m : nat) (l : Vector.t nat m) {struct l} :
  naryFunc m -> nat :=
  match l in (Vector.t _ m) return (naryFunc m -> nat) with
  | Vector.nil => fun x : naryFunc 0 => x
  | Vector.cons a n l' =>
    fun x : naryFunc (S n) => evalList n l' (x a)
  end.

```

Indeed,  $(\text{evalList } m \ v \ f)$  is the application to the vector  $v$  of an uncurried version of  $f$ . In LibraryMoreAck.AckNotPR, we introduce a lighter notation.

```

Notation "'v_apply' f v" := (evalList _ v f) (at level 10, f at level 9).

```

```

Check [4].

```

```

Example Ex2 : forall (f: naryFunc 2) x y,
  v_apply f [x;y] = f x y.

```

```

Proof.

```

```

  intros; now cbn.

```

```

Qed.

```

```

Example Ex4 : forall (f: naryFunc 4) x y z t,
  v_apply f [x;y;z;t] = f x y z t.

```

```

Proof.

```

```

  intros; now cbn.

```

```

Qed.

```

We are now able to translate in Gallina the notion of “majorization”:

```

(** ** Comparing an n-ary and a binary functions **)

```

```

Definition majorized {n} (f: naryFunc n) (A: naryFunc 2) :=
  exists (q:nat),
  forall (v: t nat n), v_apply f v <= A q (max_v v).

```

```

Definition majorizedPR {n} (x: PrimRec n) A :=
  majorized (evalPrimRec n x) A.

```

```

(** For vectors of functions **)

```

```

Definition majorizedS {n m} (fs : Vector.t (naryFunc n) m)
  (A : naryFunc 2):=
  exists N, forall (v: t nat n),
    max_v (Vector.map (fun f => v_apply f v) fs) <= A N (max_v v).

```

```

Definition majorizedSPR {n m} (x : PrimRecs n m) :=
  majorizedS (evalPrimRecs _ _ x).

```

Now, it remains to prove that any primitive function is majorized by Ack. The three base cases are as follows:

```

Lemma majorSucc : majorizedPR succFunc Ack.

```

```

Lemma majorZero : majorizedPR zeroFunc Ack.

```

```

Lemma majorProjection (n m: nat)(H: m < n): majorizedPR (projFunc n m H) Ack.

```

The remaining cases are proved within the main mutual induction.

```

Lemma majorAnyPR: forall n (x: PrimRec n), majorizedPR x Ack.

```

**Proof.**

```

  intros n x; induction x using PrimRec_PrimRecs_ind with
    (P0 := fun n m y => majorizedSPR y Ack).
  - apply majorSucc.
  - apply majorZero.
  - apply majorProjection.

  - destruct IHx, IHx0; red; exists (2 + Nat.max x0 x1).

```

```

n, m: nat
g: PrimRecs n m
x: PrimRec m
x0: nat
H: forall v : t nat n,
  max_v
  (map (fun f : naryFunc n => v_apply f v)
  (PRevalN g)) <= Ack x0 (max_v v)
x1: nat
H0: forall v : t nat m,
  v_apply (PReval x) v <= Ack x1 (max_v v)
-----
forall v : t nat n,
v_apply (PReval (PRNotations.PRcomp x g)) v <=
Ack (2 + Nat.max x0 x1) (max_v v)

```

```

  - destruct IHx1 as [r Hg]; destruct IHx2 as [s Hh].

```

```

n: nat
x1: PrimRec n
x2: PrimRec (S (S n))
r: nat
Hg: forall v : t nat n,
  v_apply (PREval x1) v <= Ack r (max_v v)
s: nat
Hh: forall v : t nat (S (S n)),
  v_apply (PREval x2) v <= Ack s (max_v v)
-----
majorizedPR (PRNotations.PRrec x1 x2) Ack

```

The last two goals deal with vectors of functions.

```

n: nat
-----
majorizedSPR (PRnil n) Ack

```

```

n, m: nat
x: PrimRec n
p: PrimRecs n m
IHx: majorizedPR x Ack
IHx0: majorizedSPR p Ack
-----
majorizedSPR (PRcons n m x p) Ack

```

### 11.7.1 Looking for a contradiction

The following lemma is just a specialization of `majorAnyPR` to binary functions (forgetting vectors, coming back to usual notations).

**Lemma** `majorPR2` (`f`: naryFunc 2)(`Hf` : isPR 2 f)  
 : exists (n:nat), forall x y, f x y <= Ack n (max x y).

We prove also a strict version of this lemma, thanks to the following property (proved in `Library MoreAck.Ack`).

**Lemma** `Ack_strict_mono_l` : forall n m p, n < m ->  
 Ack n p.+1 < Ack m p.+1.

*From* `MoreAck.AckNotPR`.

**Lemma** `majorPR2_strict` (`f`: naryFunc 2)(`Hf` : isPR 2 f):  
 exists n:nat,  
 forall x y, 2 <= x -> 2 <= y -> f x y < Ack n (max x y).

If the Ackermann function were primitive recursive, then there would exist some natural number  $n$ , such that, for all  $x$  and  $y$ , the inequality  $\text{Ack } x y \leq \text{Ack } n (\max x y)$  holds. Thus, our impossibility proof is just a sequence of easy small steps.

**Remark 11.5** In the following snippet, some versions of *Alectryon*'s *Latex* generator print the *local definition* of  $x$  (as the maximum of 2 and  $m$ ) as a simple *declaration*  $x : \text{nat}$ . Thus the proof script is correct, but the three last sub-goals are not correctly displayed, since they do not show how the inequalities  $2 \leq x$  and  $m \leq x$  could be inferred by `lia`.

A correct goal display can be obtained with this fork.

**Section Impossibility\_Proof.**

**Context** (HACK : isPR 2 Ack).

**Lemma** Ack\_not\_PR : False.

**Proof.**

destruct (majorPR2\_strict Ack HACK) as [m Hm].

```
HACK: isPR 2 Ack
m: nat
Hm: forall x y : nat,
  2 <= x ->
  2 <= y -> Ack x y < Ack m (Init.Nat.max x y)
-----
False
```

set (x := Nat.max 2 m).

```
HACK: isPR 2 Ack
m: nat
Hm: forall x y : nat,
  2 <= x ->
  2 <= y -> Ack x y < Ack m (Init.Nat.max x y)
x: nat
-----
False
```

specialize (Hm x x); rewrite Nat.max\_idempotent in Hm.

```
HACK: isPR 2 Ack
m: nat
x: nat
Hm: 2 <= x -> 2 <= x -> Ack x x < Ack m x
-----
False
```

assert (H0: Ack m x <= Ack x x) by (apply Ack\_mono\_l; lia).

```
HACK: isPR 2 Ack
m: nat
x: nat
Hm: 2 <= x -> 2 <= x -> Ack x x < Ack m x
H0: Ack m x <= Ack x x
-----
False
```

lia.

**Qed.**

**End Impossibility\_Proof.**

**Remark 11.6** It is easy to prove that any unary function which dominates (fun

$n \Rightarrow \text{Ack } n \ n$ ) fails to be primitive recursive. To this end, we use an instance of `majorAnyPR` dealing with unary functions.

*From `MoreAck.AckNotPR`.*

```
Lemma majorPR1 (f: naryFunc 1)(Hf : isPR 1 f)
  : exists (n:nat), forall x, f x <= Ack n x.
```

Then, we write a short proof by contradiction, using a *diagonalized* version of Ackermann function.

**Section `dom_AckNotPR`.**

```
Variable f : nat -> nat.
Hypothesis Hf : dominates f (fun n => Ack n n).
```

```
Lemma dom_AckNotPR: isPR 1 f -> False.
```

**Proof.**

```
intros H; destruct Hf as [n Hn].
destruct (majorPR1 _ H) as [m Hm].
pose (x := Nat.max n m).
specialize (Hn x (Nat.le_max_l n m)); (* for 8.13.dev's lia *)
  cbn in Hn; specialize (Hm x).
  assert (Ack m x <= Ack x x) by (apply Ack_mono_l; subst; lia).
lia.
```

**Qed.**

**End `dom_AckNotPR`.**

**Remark 11.7** It may be interesting to compare the following statements:

- Ackermann function is not primitive recursive.
- For any  $n$ , the function `Ack n _` is primitive recursive (see 11.6.1.1 on page 221).

### 11.7.2 Related work

This proof is very close to the 1993 proof by Nora Szasz with the `Alf` proof assistant [Sza93]. This proof has also been adapted by Lawrence C. Paulson to Isabelle/HOL [PAU21].

## 11.8 The length of standard hydra battles

The module `Hydra_Theorems` contains a proof that the function which computes the length of standard hydra battles is not primitive recursive. More precisely, we consider, for a given hydra  $h = \iota(\alpha)$ , the length of a standard battle which starts with the replication factor  $k$  (see Sect 6.2.4.2 on page 129).

This proof is a little more complex than the preceding one.

### 11.8.1 Definitions

The function we consider is defined and proven correct in Module Hydra.Battle\_length.

**Definition** `l_std alpha k := (L_alpha (S k) - k)%nat.`

**Lemma** `l_std_ok : forall alpha : E0,  
alpha <> E0zero ->  
forall k : nat,  
1 <= k -> battle_length standard k (iota (cnf alpha))  
(l_std alpha k).`

### 11.8.2 Proof steps

Now, let us assume that the function `l_std` is primitive recursive.

*From Hydra.Hydra\_Theorems.*

**Section** `battle_length_notPR.`

**Context** `(H: forall alpha, isPR 1 (l_std alpha)).`

Let us consider the hydra represented by the ordinal  $\omega^\omega$ .

**Let** `alpha := E0_phi0 E0_omega.`

**Let** `h := iota (cnf alpha).`

In order to get rid of the subtraction in the definition of `l_std`, we work with a helper function.

**Let** `m k := L_alpha (S k).`

**Remark** `m_eqn : forall k, m k = (l_std alpha k + k)%nat.`

Under the hypothesis `H`, `m` is also primitive recursive.

`#[local] Instance mIsPR : isPR 1 m.`

#### 11.8.2.1 Comparison between $F$ and $H'$

In `Epsilon0.F_alpha`, we prove a relation between the  $F$  and  $H'$  functionals. For any  $\alpha$  and  $k > 0$ ,  $H'_{\omega^\alpha}(k) \geq F_\alpha(k)$ .

**Lemma** `H'_F alpha : forall n, F_alpha (S n) <= H'_ (E0_phi0 alpha) (S n).`

**Proof.**

`pattern alpha; apply well_founded_induction with E0lt.`

Our proof of this lemma is not trivial at all, it uses some properties of the Ketonen-Solovay's toolkit. We advise the reader to explore this proof, with the help of an IDE or software like Alectryon.

### 11.8.2.2 End of the proof

We finish the proof by comparing several fast growing functions.

*From Epsilon0.L\_alpha*

**Theorem H'\_L\_alpha :**  
`forall i:nat, (H'_alpha i <= L_alpha (S i))%nat.`

*From Epsilon0.F\_omega*

**Lemma F\_vs\_Ack n :** `2 <= n -> Ack n n <= F_E0_omega n.`

By transitivity, we get the inequality  $F_\omega(k+1) \leq m(k+1)$ , for any  $k$ .

**Remark m\_ge\_F\_omega k:** `F_E0_omega (S k) <= m (S k).`

We finish the proof by noting that the function  $m$  (composed with  $S$ ) dominates the Ackermann function, which leads to a contradiction.

**Remark m\_dominates\_Ack :**  
`dominates (fun n => S (m n)) (fun n => Ack.Ack n n).`

**Lemma SmNotPR :** `isPR 1 (fun n => S (m n)) -> False.`

**Theorem LNotPR :** `False.`

**Proof.**

```
apply SmNotPR, compose1_1IsPR.
- apply mIsPR.
- apply succIsPR.
```

**Qed.**

**End battle\_length\_notPR.**

**Check l\_std\_ok.**

```
l_std_ok
: forall alpha : E0,
  alpha <> E0zero ->
  forall k : nat,
  1 <= k ->
  battle_length standard k (iota (cnf alpha))
  (l_std alpha k)
```

**Check LNotPR.**

```
LNotPR
: (forall alpha : E0, isPR 1 (l_std alpha)) ->
  False
```

**Search L\_ F\_.**

```
m_ge_F_omega:
forall k : nat,
F_E0_omega (S k) <=
(fun k0 : nat => L_ (E0_phi0 E0_omega) (S k0)) (S k)
```

# Chapter 12

## First Order Logic (in construction)

### 12.1 Introduction

This chapter is devoted to the presentation of data structures for representing terms and first order formulas over a ranked alphabet, and the basic functions and predicates over these types, more precisely:

- Abstract syntax of terms and formulas over a ranked alphabet composed of function and relation symbols.
- Induction principles over terms and formulas.
- Definition and main properties of substitution of terms to variables.

Although all the following constructions come directly from Russel O'Connor's work [O'C05a], we introduced minor (mainly syntactic) changes to take into account recent changes in Coq (new constructions, tactics, notations, etc.).

### 12.2 Data types

#### 12.2.1 Languages

A *language* is a structure composed of relation and function symbols, each symbol is given an *arity* (number of arguments)<sup>1</sup>.

From Ackermann.fol

```
Record Language : Type := language
{ Relations : Set;
  Functions : Set;
  arityR : Relations -> nat;
  arityF : Functions -> nat}.
```

---

<sup>1</sup>As suggested by Russel O'Connor in [O'C05b], we consider two arity functions instead of a single function defined on the sum type Relations + Functions.

### 12.2.1.1 Example: $L$ , a toy language

In order to show a few simple examples of statements and proofs, we define a small language with very few symbols: two constant symbols:  $a$  and  $b$ , three function symbols  $f$ ,  $g$  and  $h$  (of respective arity 1, 1 and 2), three propositional symbols  $A$ ,  $B$  and  $C$ , two one-place predicates symbols  $P$  and  $Q$ , and a binary relational symbol  $R$ .

From MoreAck.FolExamples.

**Module Toy.**

**Inductive Rel:** Set := A\_ | B\_ | C\_ | P\_ | Q\_ | R\_.

**Inductive Fun :** Set := a\_ | b\_ | f\_ | g\_ | h\_.

**Definition arityR** (x : Rel): nat :=  
 match x with  
   P\_ | Q\_ => 1 | R\_ => 2 | \_ => 0  
 end.

**Definition arityF** (x : Fun): nat :=  
 match x with f\_ | g\_ => 1 | h\_ => 2 | \_ => 0 end.

**Definition L** := language Rel Fun arityR arityF.

**Remark 12.1** The constructors of types Rel and Fun are suffixed by an underscore, in order to reserve the names a, f, h, R, etc. to the functions which build terms and formulas (please look at Sect 12.2.2.1 and 12.2.4).

## 12.2.2 Terms

Given a language  $L$ , we define the type of *terms* and  *$n$ -tuples of terms* over  $L$ .

**Section First\_Order\_Logic.**

**Variable L :** Language.

**Inductive Term :** Set :=  
 | var : nat -> Term  
 | apply : forall f : Functions L, Terms (arityF L f) -> Term  
**with Terms :** nat -> Set :=  
 | Tnil : Terms 0  
 | Tcons : forall n : nat, Term -> Terms n -> Terms (S n).

**Remark 12.2** This representation of terms uses mutually inductive data-types instead of lists or vectors of terms. Please see also Remark 2.1 on page 25.

**Remark 12.3 (Variables)** In O'Connor's formalization of first-order logic, variables are just natural numbers, and the conversion from nat to Term L is the constructor (@var L). Although other choices may be considered : PHOAS, de Bruijn indices, etc, we still the data structures of [O'C05a], in order not to break long proof scripts which use this representation (please look at Section 2 of [O'C05b] for a related discussion).

### 12.2.2.1 Examples

Let us build a few Gallina terms over our toy language, respectively corresponding to the terms  $a$ ,  $f(a)$ ,  $h(f(a), a)$ , and  $h(f(v_0), g(v_1))$ .

First, in order to make terms on  $L$  more readable, we introduce a few abbreviations.

From `MoreAck.FolExamples`.

```

Notation a := (apply L a_ Tnil).
Notation b := (apply L b_ Tnil).
Notation f t := (apply L f_ (Tcons t Tnil)).
Notation g t := (apply L g_ (Tcons t Tnil)).
Notation h t1 t2 := (apply L h_ (Tcons t1 (Tcons t2 Tnil))).

```

**Example** `t0` : Term L := a.

**Example** `t1` : Term L := f t0.

**Example** `t2` : Term L := h t1 t0.

**Example** `t3` : Term L := h (f (var 0)) (g (var 1)).

The following “term” `t4` is not well formed, since the arity of  $h$  is not respected<sup>2</sup>.

**Fail Example** `t4` : Term L := h t0.

```

The command has indeed failed with message:
Abbreviation is not applied enough.

```

### 12.2.2.2 Other Languages

**To do 12.1** *Link to the chapter which presents LNT and LNN.*

## 12.2.3 First-order formulas

The type of first order formulas over  $L$  is defined in `Ackermann.fol` as an inductive data type, with a limited set of basic constructions: *term equalities*  $t_1 = t_2$ , *atomic propositions*  $R t_1 \dots t_n$ , where  $R$  is a relation symbol of arity  $n$ , *implications*  $A \rightarrow B$ , *negations*  $\sim A$ , and *universal quantifications*  $\forall i, A$ .

From `Ackermann.fol`

```

Inductive Formula : Set :=
| equal : Term -> Term -> Formula
| atomic : forall r : Relations L, Terms (arityR L r) -> Formula
| impH : Formula -> Formula -> Formula
| notH : Formula -> Formula
| forallH : nat -> Formula -> Formula.

```

<sup>2</sup>Strictly speaking, it's not a (well typed) term!

**Remark 12.4** In [O’C05a], no *constructors* of type (Formula L) are associated with disjunction, conjunction, logical equivalence and existential quantifier. These constructs are formalized through *definitions* in terms of `impH`, `notH` and `forallH`<sup>3</sup>.

**Definition** `orH (A B : Formula) := impH (notH A) B.`

**Definition** `andH (A B : Formula) := notH (orH (notH A) (notH B)).`

**Definition** `iffH (A B : Formula) := andH (impH A B) (impH B A).`

**Definition** `existH (x : nat) (A : Formula) := notH (forallH x (notH A)).`

**Definition** `ifThenElseH (A B C : Formula) :=  
andH (impH A B) (impH (notH A) C).`

This convention allows the user to reduce to 5 (instead of 10) the number of cases in “`match F with ...`” terms. On the other hand, some computation may expand a connective like  $\vee$  or  $\wedge$ , or an existential quantification into a “basic” formula (see Sect.12.3.1 on page 236).

## 12.2.4 Examples

Let us give a few examples of first-order formulas over  $L$ .

**F1**  $R a b$

**F2**  $\forall v_0 v_1, R v_0 v_1 \rightarrow R v_1 v_0$

**F3**  $\forall v_0, v_0 = a \vee \exists v_1, v_0 = f(v_1)$

**F4**  $(\forall v_1, v_0 = v_1) \vee \exists v_0 v_1, v_0 \neq v_1$

**F5**  $v_0 = a \vee v_0 = f(v_1)$

**F6**  $\forall v_0, \exists v_1, v_0 = f(v_1) \wedge v_0 \neq v_1$

Let us now define these formulas as terms of type (Formula L).

First, we define abbreviations for atomic formulas over  $L$ .

From `MoreAck.FolExamples`

**Notation** `A := (atomic L A_ Tnil).`

**Notation** `B := (atomic L B_ Tnil).`

**Notation** `C := (atomic L C_ Tnil).`

**Notation** `P t := (atomic L P_ (Tcons t Tnil)).`

**Notation** `Q t := (atomic L Q_ (Tcons t Tnil)).`

**Notation** `R t1 t2 := (@atomic L R_ (Tcons t1 (Tcons t2 Tnil))).`

**Example F1** : Formula L := `R a b.`

**Example F2** : Formula L :=  
`forallH 0 (forallH 1  
 (impH (R (var 0) (var 1)) (R (var 1) (var 0))))).`

**Example F3** : Formula L :=

<sup>3</sup>Please keep in mind that we are considering a classical logic.

```
forallH 0 (orH (equal (var 0) a)
              (existH 1 (equal (var 0) (f (var 1))))).
```

**Example F4:** Formula  $L :=$   
 $\text{orH} (\text{forallH } 1 (\text{equal} (\text{var } 0) (\text{var } 1)))$   
 $(\text{existH } 0 (\text{existH } 1 (\text{notH} (\text{equal} (\text{var } 0) (\text{var } 1))))).$

**Example F5:** Formula  $L := (\text{v}\#0 = a \ \backslash / \ \text{v}\#0 = f \ \text{v}\#1)\%fol.$

**Example F6:** Formula  $L := (\text{allH } 0, \text{exH } 1, \text{v}\#0 = f \ \text{v}\#1 \ \wedge \ \text{v}\#0 <> \text{v}\#1)\%fol.$

#### 12.2.4.1 Bound variables

In [O’C05a], there is no De Bruijn encoding of bound variables (see also [O’C05b]).

For instance, the term  $(\text{var } 0)$  occurs both freely and inside the scope of a quantifier in the formula F4 on the preceding page.

The following example shows two formulas which share the same structure, are logically equivalent, but are not Leibniz equal.

From `MoreAck.FolExamples`

```
Goal forallH 1 (equal (var 1) a) <> forallH 0 (equal (var 0) a).
  discriminate.
Qed.
```

**To do 12.2** *Link to the lemmas which attest the equivalence of these formulas (properties of substitution, logical equivalence).*

**Project 12.1** Define a [P]HOAS representation for FOL terms and formulas. Could we avoid to break some proof scripts?

## 12.3 A notation scope for first-order terms and formulas

We use Coq’s `Notation` features to print and parse terms and formulas in a more readable form. To this purpose, we build `fol_scope`, a notation scope where the main connectives and quantifiers get a syntax close to Coq’s. Additionally, a term of the form  $(\text{@var } \_ \ i)$  is just printed and parsed  $\text{v}\#i$ .

```
(** ** The [fol_scope] notation scope *)
```

```
Module FolNotations.
Declare Scope fol_scope.
Delimit Scope fol_scope with fol.

Infix "=" := (equal _): fol_scope.
Infix "\/" := (orH): fol_scope.
Infix "\&" := (andH): fol_scope.
Infix "->" := (impH): fol_scope.
```

**Notation** " $\sim A$ " := (@notH \_ A): fol\_scope.

**Notation** " $A \leftrightarrow B$ " := (@iffH \_ A B): fol\_scope.

**Notation** " $\forall\# i$ " := (var i) (at level 3, format " $\forall\# i$ ", i at level 0) : fol\_scope.

**Notation** " $\text{exH } x \dots y, p$ " := (existH x .. (existH y p) ..)

(x at level 0, y at level 0, at level 200, right associativity) : fol\_scope.

**Notation** " $\text{allH } x \dots y, p$ " := (forallH x .. (forallH y p) ..)

(x at level 0, y at level 0, at level 200, right associativity) : fol\_scope.

**Notation** " $t = u$ " := (@equal \_ t u): fol\_scope.

**Notation** " $t \leftrightarrow u$ " := ( $\sim t = u$ )%fol : fol\_scope.

The %fol delimiter allows the user to distinguish FOL connectives from their Coq equivalent. We discourage the reader from opening fol\_scope and similar scopes: nn\_scope, nt\_scope, which would make disappear the %fol suffix from the first-order formulas.

From Ackermann.fol

**Print** F1.

```
F1 = R a b
      : Formula L
```

**Print** F2.

```
F2 =
(allH 0 1, R v#0 v#1 -> R v#1 v#0)%fol
      : Formula L
```

**Print** F3.

```
F3 =
(allH 0, v#0 = a \\/ (exH 1, v#0 = f v#1))%fol
      : Formula L
```

### 12.3.1 The issue with derived constructions

The connectives and quantifiers  $\forall$ ,  $\wedge$ ,  $\exists$ , etc. may raise an issue when printing computed formulas. For instance, a formula like  $F \wedge G$  could be transformed into  $\sim(\sim F \vee \sim G)$ , and even into  $\sim(\sim\sim F \rightarrow \sim B)$ , which would cause serious problems of readability.

In such a case, we propose to print such a formula as  $F \wedge' G$ , to make it syntactically distinct but very similar to  $F \wedge G$ .

**Reserved Notation** " $x \ \backslash\prime \ y$ " (at level 85, right associativity).

**Reserved Notation** " $x \ \wedge' \ y$ " (at level 80, right associativity).

**Reserved Notation** " $x \ \langle\rightarrow' \ y$ " (at level 95, no associativity).

**Reserved Notation** " $x \ \langle\rightarrow' \prime \ y$ " (at level 95, no associativity).

**Notation** " $x \ \backslash\prime \ y$ " := ( $\sim x \rightarrow y$ )%fol : fol\_scope.

**Notation** " $x \ \wedge' \ y$ " := ( $\sim(\sim x \ \backslash\prime \ \sim y)$ )%fol : fol\_scope.

**Notation** " $x \ \langle\rightarrow' \ y$ " := (( $x \rightarrow y$ )  $\wedge$  ( $y \rightarrow x$ ))%fol: fol\_scope.

**Notation** `"x <-> y" := (~ (~ (x -> y) \/' ~ (y -> x)))%fol : fol_scope.`

**Notation** `exH' v A := (~ (forallH v (~ A)))%fol.`

**End FolNotations.**

The following examples show how the primed connectors and quantifiers behave with respect to convertibility and input/output.

**Section PrimedSymbols.**

**Compute** `(F3 /\ F1)%fol.`

```
= ((allH 0, v#0 = a \/' exH' 1 (v#0 = f v#1)) /\'
  R a b)%fol
: Formula L
```

**Goal** `(F3 /\ F1)%fol = (~(~ ~ F3 -> ~ F1))%fol.`

```
(F3 /\ F1)%fol = (F3 /\' F1)%fol
```

`reflexivity.`

**Qed.**

**Print** `F6.`

```
F6 =
(allH 0, exH 1, v#0 = f v#1 /\ v#0 <> v#1)%fol
: Formula L
```

**Compute** `F6.`

```
= (allH 0, exH' 1 (v#0 = f v#1 /\' v#0 <> v#1))%fol
: Formula L
```

**#[local] Unset Printing Notations.**

**Print** `F6.`

```
F6 =
forallH 0
  (existH 1
    (andH
      (equal (var 0)
        (apply L f_ (Tcons (var 1) Tnil)))
      (notH (equal (var 0) (var 1)))))
: Formula L
```

**Compute** `F6.`

```

= forallH 0
  (notH
    (forallH 1
      (notH
        (notH
          (impH
            (notH
              (notH
                (equal (var 0)
                  (apply L f_
                    (Tcons (var 1)
                      Tnil))))))
                (notH
                  (notH
                    (equal (var 0) (var 1))))))))))
      (notH
        (notH
          (equal (var 0) (var 1))))))
  : Formula L

```

End PrimedSymbols.

**Remark 12.5** In some situations (like in the proof of PrfEx4 12.5.1.7 on page 253) the user may be puzzled when a formula [s]he typed explicitly  $\sim A \rightarrow \sim B$  will be printed  $A \vee' \sim B$ . We will try to fix this issue.

## 12.4 Computing and reasoning on first-order formulas

### 12.4.1 Structural recursion on formulas

Structural induction/recursion principles are generated from Term, Terms and Formula's definition, for instance:

```

Scheme Term_Terms_rec_full := Induction for Term Sort Set
  with Terms_Term_rec_full := Induction for Terms Sort Set.
About Term_Terms_rec_full.

```

```

Term_Terms_rec_full :
forall (L : Language) (P : Term L -> Set)
  (P0 : forall n : nat, Terms L n -> Set),
(forall n : nat, P (v#n)%fol) ->
(forall (f0 : Functions L) (t : Terms L (arityF L f0)),
  P0 (arityF L f0) t -> P (apply L f0 t)) ->
P0 0 Tnil ->
(forall (n : nat) (t : Term L),
  P t ->
  forall t0 : Terms L n,
  P0 n t0 -> P0 (S n) (Tcons t t0)) ->
forall t : Term L, P t

Term_Terms_rec_full is not universe polymorphic
Arguments Term_Terms_rec_full L
  (P P0 f f)%function_scope f f%function_scope t
  (where some original arguments have been renamed)
Term_Terms_rec_full is transparent
Expands to: Constant
hydras.Ackermann.fol.Term_Terms_rec_full

```

**About** Formula\_rect.

```

Formula_rect :
forall (L : Language) (P : Formula L -> Type),
(forall t t0 : Term L, P (t = t0)%fol) ->
(forall (r : Relations L) (t : Terms L (arityR L r)),
  P (atomic L r t)) ->
(forall f1 : Formula L,
  P f1 ->
  forall f2 : Formula L, P f2 -> P (f1 -> f2)%fol) ->
(forall f2 : Formula L, P f2 -> P (~ f2)%fol) ->
(forall (n : nat) (f3 : Formula L),
  P f3 -> P (allH n, f3)%fol) ->
forall f4 : Formula L, P f4

Formula_rect is not universe polymorphic
Arguments Formula_rect L (P f f f f f)%function_scope
  f
Formula_rect is transparent
Expands to: Constant hydras.Ackermann.fol.Formula_rect

```

#### 12.4.1.1 Free variables

The functions `freeVarT` [resp. `freeVarTs`, and `freeVarF`] compute the multiset (as a list with possible repetitions) of the free occurrences of variables in a term [resp. a vector of terms, a formula].

```

Fixpoint freeVarT (s : fol.Term L) : list nat :=
  match s with
  | var v => v :: nil
  | apply f ts => freeVarTs (arityF L f) ts
  end
with freeVarTs (n : nat) (ss : fol.Terms L n) {struct ss} : list nat :=

```

```

match ss with
| Tnil => nil (A:=nat)
| Tcons m t ts => freeVarT t ++ freeVarTs m ts
end.

```

Concerning formulas, the treatment of binding is realized with the help of the List library function `remove`.

```

Fixpoint freeVarF (A : fol.Formula L) : list nat :=
  match A with
  | equal t s => freeVarT t ++ freeVarT s
  | atomic r ts => freeVarTs _ ts
  | impH A B => freeVarF A ++ freeVarF B
  | notH A => freeVarF A
  | forallH v A => remove eq_nat_dec v (freeVarF A)
  end.

```

```

Compute freeVarF (allH 0, v#0 = v#1)%fol.

```

```

= [1]
: list nat

```

```

Compute freeVarF (allH 0, v#0 = v#0)%fol.

```

```

= []
: list nat

```

```

Compute freeVarF (v#0 = v#1 \ / allH 0, v#0 = v#1)%fol.

```

```

= [0; 1; 1]
: list nat

```

**Remark 12.6** Note that `freeVarF` is defined by cases over the *basic* connectives. Formulas with contain `iffH` or `ifThenElseH` are expanded before the application of `freeVarF`, and the list returned by `freeVarF` may contain redundancies.

If we want to get the *set* of variables which occur freely in a formula  $F$ , we may use the function `List.nodup`.

```

Compute freeVarF (v#0 = v#1 <-> v#1 = v#0)%fol.

```

```

= [0; 1; 1; 0; 1; 0; 0; 1]
: list nat

```

```

Compute nodup Nat.eq_dec (freeVarF (v#0 = v#1 <-> v#1 = v#0)%fol).

```

```

= [0; 1]
: list nat

```

### 12.4.1.2 Closing a formula

Function `freeVarF` is used in the function `close`, which universally quantifies all the free variables of a formula.

From `Ackermann.folProp`

(\* added by PC \*)

```

Definition closed (a : fol.Formula L) :=
  forall v: nat, ~ In v (freeVarF a).

```

```

Fixpoint closeList (l : list nat)(a : fol.Formula L) :=
  match l with
  | nil => a
  | cons v l => f[  $\forall$  v, {closeList l a} ]f
  end.

```

```

Definition close (x : fol.Formula L) : fol.Formula L :=
  closeList (nodup eq_nat_dec (freeVarF x)) x.

```

From MoreAck.FolExamples

```

Compute close L (v#0 = a  $\wedge$  v#0 = f v#1)%fol.

```

```

= (allH 0 1, v#0 = a  $\wedge$  v#0 = f v#1)%fol
: Formula L

```

**Remark 12.7** The function `close` applies `freeVarF` and `List.nodup` in order to add a sequence of universal quantifications ( $\text{allH } i_1 \dots i_k$ ), in an order determined by the actual implementation of these functions. It may be interesting to check whether the proof of properties of `close` depend or not from this implementation.

## 12.4.2 Decidability of equality

Let  $L$  be a language, and let us assume that equality of function and relation symbols of  $L$  are decidable. Under this assumption, equality of terms and formulas over  $L$  is decidable too.

Because of dependent types, the proofs are quite long and technical. The reader may consult them in `Ackermann.fol`

**Section** `Formula_Decidability`.

```

Definition language_decidable :=
  ((forall x y : Functions L, {x = y} + {x <> y}) *
   (forall x y : Relations L, {x = y} + {x <> y}))%type.

```

```

Hypothesis language_eqdec : language_decidable.

```

```

Lemma term_eqdec : forall x y : Term, {x = y} + {x <> y}.

```

```

Lemma terms_eqdec n (x y : Terms n): {x = y} + {x <> y}.

```

```

Lemma formula_eqdec : forall x y : Formula, {x = y} + {x <> y}.

```

```

End Formula_Decidability.

```

**Remark 12.8** Please note that `term_dec`, `terms_dec` and `formula_dec` are *opaque*.

The function `formula_dec` is mainly used in `Ackermann.PA`, in order to check whether a given formula belongs to the axioms of Peano arithmetic.

**To do 12.3** *Look for the use of open (in codePA)*

### 12.4.3 Variables and substitutions

The substitution of a term to the free occurrences of a given variable  $v$  is at the heart of the implementation of universal quantifier elimination.

Since free and bound occurrences of a variable  $i$  are represented the same way, much care should be taken in programming the substitution of a term to a variable in order to avoid *variable capture*.

Substitution of a term  $t$  to all the occurrences of a variable  $x$  in a term or a vector of terms is easy to define as a pair of mutually structurally recursive functions.

```

Fixpoint substT (s : fol.Term L) (x : nat)
  (t : fol.Term L) {struct s} : fol.Term L :=
  match s with
  | var v =>
    match eq_nat_dec x v with
    | left _ => t
    | right _ => var v
    end
  | apply f ts => apply f (substTs _ ts x t)
  end
with substTs (n : nat) (ss : fol.Terms L n)
  (x : nat) (t : fol.Term L) {struct ss} : fol.Terms L n :=
  match ss in (fol.Terms _ n0) return (fol.Terms L n0) with
  | Tnil => Tnil
  | Tcons m s ts =>
    Tcons (substT s x t) (substTs m ts x t)
  end.

```

```

Compute substT (h v#1 (h (f v#1) (f v#2)))%fol 1 (h a b)%fol.

```

```

= h (h a b) (h (f (h a b)) (f (v#2)%fol))
: Term L

```

Concerning formulas, it could be tempting to define substitution the same way.

```

Module BadSubst.

```

```

Fixpoint substF L (F : Formula L) v (t: Term L) :=
  match F with
  | equal t1 t2 => equal (substT t1 v t) (substT t2 v t)
  | atomic r s => atomic L r (substTs s v t)
  | impH G H => impH (substF L G v t) (substF L H v t)
  | notH G => notH (substF L G v t)
  | forallH w G => if Nat.eq_dec w v then F else forallH w (substF L G v t)
  end.

```

```

End BadSubst.

```

Let us consider for instance the formula  $F = \forall v_1, \exists v_2, v_1 \neq f(v_2)$  (satisfiable if we take for instance  $f$  to be interpreted as the successor function on natural numbers).

If we eliminate the universal quantifier by substituting in the sub-formula  $F_1 = \exists v_2, v_1 \neq f(v_2)$  the free occurrences of  $v_1$  with  $f(v_2)$ , our naive implementation of `substF` returns the absurd proposition  $\exists v_2, f(v_2) \neq f(v_2)$ . We say that the free occurrence of  $v_2$  in the term  $f(v_2)$  has been *captured* by the binding  $\exists v_2, \dots$ .

### Section `BadExample`.

```
Let F := (allH 1, exH 2, v#1 <> f v#2)%fol.
Let F1: Formula L := (exH 2, v#1 <> f v#2)%fol.

Compute BadSubst.substF L F1 1 (f v#2)%fol.
```

```
= exH' 2 (f v#2 <> f v#2)
: Formula L
```

### End `BadExample`.

In this example, we could obtain a correct result, if

1. We consider a *fresh* variable, *i.e.* different from  $v_1$  and  $v_2$ , say for instance  $v_3$ ,
2. we substitute  $v_3$  to  $v_2$  in  $F_1$  which results in  $F_2 = \exists v_3, v_1 \neq f(v_3)$
3. we substitute the term  $f(v_2)$  to  $v_1$  in  $F_2$ , which gives us  $\exists v_3, f(v_2) \neq f(v_3)$ .

The notion of fresh variable is implemented through a function `newvar (l: list nat) : nat` which returns a number which doesn't belong to  $l$ .

But the following attempt fails, because the renaming of a variable in a sub-formula of a formula  $F$  is not structurally smaller than  $F$ .

```
Fail Fixpoint substF L (F : Formula L) v (t: Term L) :=
  match F with
  | equal t1 t2 => equal (substT L t1 v t) (substT L t2 v t)
  | atomic r s => atomic L r (substTs L (arityR L r) s v t)
  | impH G H => impH (substF L G v t) (substF L H v t)
  | notH G => notH (substF L G v t)
  | forallH w G => if Nat.eq_dec w v then F else
    let nv := newVar (w :: freeVarT L t ++ freeVarF L G)
    in let H := (substF L G w (var nv))
    in forallH nv (substF L H v t)
end.
```

```

The command has indeed failed with message:
In environment
substF : forall (L : Language) (F : Formula L)
  (v : ?T) (t : Term L), ?T0@{F:=F; f0:=F}
L : Language
F : Formula L
v : ?T
t : Term L
t1 : Term L
t2 : Term L
The term "L" has type "Language"
while it is expected to have type "Term ?L0".

```

Fortunately, Coq allows us to define functions by well-founded recursion, and in particular with the help of a *measure* mapping every formula to an already known well-founded type.

### 12.4.3.1 Depth of a formula

The function `depth` computes the *depth* of any formula, *i.e.* the height of the sub-tree made by erasing all nodes but those nodes labelled with `allH`, `impH` and `notH`.

```

Fixpoint depth (A : Formula) : nat :=
  match A with
  | equal _ _ => 0
  | atomic _ _ => 0
  | impH A B => S (Nat.max (depth A) (depth B))
  | notH A => S (depth A)
  | forallH _ A => S (depth A)
  end.

```

**Definition** `lt_depth (A B : Formula) : Prop := depth A < depth B`.

**Remark 12.9** The depth of a formula takes into account its abstract syntax tree *with respect to the base connective and quantifiers* :  $\rightarrow$ ,  $\sim$  and  $\forall$ . Formulas which contain  $\vee$ ,  $\wedge$ ,  $\exists$ , etc. are translated into basic formulas before the computation of their depth. In the example below, the conjunction is translated into a bigger term than the disjunction.

```

Goal lt_depth L (v#0 = v#1 \\/ exH 2, v#1 = f v#2)%fol
  (v#0 = v#1 /\ exH 2, v#1 = f v#2)%fol.
  red; simpl.

```

```

4 < 6

```

```

  auto with arith.

```

**Qed.**

### 12.4.3.2 Induction on depth

Lemma `fol.Formula_depth_rec` is the basic induction principle based on depth.

```

Formula_depth_rec :
forall (L : Language) (P : Formula L -> Set),
(forall a : Formula L,
 (forall b : Formula L, lt_depth L b a -> P b) -> P a) ->
forall a : Formula L, P a

Formula_depth_rec is not universe polymorphic
Arguments Formula_depth_rec L (P rec)%function_scope a
Formula_depth_rec is transparent
Expands to: Constant
hydras.Ackermann.fol.Formula_depth_rec

```

```

L: Language
P: Formula L -> Prop
a: Formula L
Ha: forall b : Formula L, lt_depth L b a -> P b
-----
P a

```

**To do 12.4** *Look for the principles which are really used in Ackermann or/and Goedel libraries, and comment them. Maybe skip the helpers (unused in other files)*

The library Ackermann.fol contains several derived induction principles, applied throughout Ackermann and Goedel projects.

Let us for instance have a look at `Formula_depth_ind2` which helps to prove a goal  $(P a)$  by generating five sub-goals.

<pre>L: Language P: Formula L -&gt; Prop a: Formula L forall t t0 : Term L, P (t = t0)%fol</pre>
<pre>L: Language P: Formula L -&gt; Prop a: Formula L forall (r : Relations L) (t : Terms L (arityR L r)), P (atomic L r t)</pre>
<pre>L: Language P: Formula L -&gt; Prop a: Formula L forall f : Formula L, P f -&gt; forall f0 : Formula L, P f0 -&gt; P (f -&gt; f0)%fol</pre>
<pre>L: Language P: Formula L -&gt; Prop a: Formula L forall f : Formula L, P f -&gt; P (~ f)%fol</pre>
<pre>L: Language P: Formula L -&gt; Prop a: Formula L forall (v : nat) (a : Formula L), (forall b : Formula L, lt_depth L b (allH v, a)%fol -&gt; P b) -&gt; P (allH v, a)%fol</pre>

- Goals 1 to 4 correspond to usual proofs by structural induction (without referring to `depth`).
- Goal 5 is associated with a universal quantification  $f = \forall v, a$ . In this case, we have to prove that  $P b$  holds for any formula  $b$  which has a depth strictly less than  $f$ . Such a  $b$  may for instance be the result of replacing the free occurrences of  $v$  in  $a$  with any term  $t$ .

**To do 12.5** *Make a link to an appropriate example.*

#### 12.4.4 A correct definition of `substF`

Substitution of a term to (free- occurrences of a variable in a formula is defined in a section of `Ackermann.folProp`. The definition itself takes 200 lines of Coq code, so we will only comment its structure.

Fortunately, the reader may skip a few complex definitions (often because of dependent pattern matching), whose purpose is twofold:

- Ensure that the substitution in a formula  $F$  of a term  $t$  to a variable  $x$  returns a formula of the same depth as  $F$ , which allows to define a function by well-founded recursion on depth.

**Definition** `substituteFormulaHelp` (`f` : `fol.Formula L`)  
 (`v` : `nat`) (`s` : `fol.Term L`) :  
`{y : fol.Formula L | depth L y = depth L f}`.

**Definition** `substF` (`f` : `fol.Formula L`) (`v` : `nat`) (`s` : `fol.Term L`) :  
`fol.Formula L := proj1_sig (substituteFormulaHelp f v s)`.

- Prove a few *equations* which will be used in further proofs.

**Lemma** `subFormulaEqual` :  
`forall (t1 t2 : fol.Term L) (v : nat) (s : fol.Term L),`  
`substF (t1 = t2)%fol v s =`  
`(substT t1 v s = substT t2 v s)%fol.`

**Proof.** `reflexivity. Qed.`

**Lemma** `subFormulaRelation` :  
`forall (r : Relations L) (ts : fol.Terms L (arityR L r))`  
`(v : nat) (s : fol.Term L),`  
`substF (atomic r ts) v s =`  
`atomic r (substTs (arityR L r) ts v s).`

**Proof.** `reflexivity. Qed.`

**Lemma** `subFormulaImp` :  
`forall (f1 f2 : fol.Formula L) (v : nat) (s : fol.Term L),`  
`substF (f1 -> f2)%fol v s =`  
`(substF f1 v s -> substF f2 v s)%fol.`

**Proof.**

`(* ... *)`

**Lemma** `subFormulaNot` :  
`forall (f : fol.Formula L) (v : nat) (s : fol.Term L),`  
`substF (~ f)%fol v s = (~ substF f v s)%fol.`

**Lemma** `subFormulaForall` :  
`forall (f : fol.Formula L) (x v : nat) (s : fol.Term L),`  
`let nv := newVar (v :: freeVarT s ++ freeVarF f) in`  
`substF (allH x, f)%fol v s =`  
`match eq_nat_dec x v with`  
`| left _ => forallH x f`  
`| right _ =>`  
`match In_dec eq_nat_dec x (freeVarT s) with`  
`| right _ => (allH x, substF f v s)%fol`  
`| left _ => (allH nv, substF (substF f x (v# nv) ) v s)%fol`  
`end`  
`end.`

- Similar equations are also proved for derived connectors and quantifiers, for instance:

```

Lemma subFormulaAnd :
  forall (f1 f2 : fol.Formula L) (v : nat) (s : fol.Term L),
    substF (f1 /\ f2)%fol v s =
      (substF f1 v s /\ substF f2 v s)%fol.

```

**Proof.**

```

  intros ? ? ? ?; unfold andH in |- *.
  rewrite subFormulaNot, subFormulaOr;
  now repeat rewrite subFormulaNot.

```

**Qed.**

```

Lemma subFormulaExist :
  forall (f : fol.Formula L) (x v : nat) (s : fol.Term L),
    let nv := newVar (v :: freeVarT s ++ freeVarF f) in
    substF (existH x f) v s =
      match eq_nat_dec x v with
      | left _ => existH x f
      | right _ =>
        match In_dec eq_nat_dec x (freeVarT s) with
        | right _ => existH x (substF f v s)
        | left _ =>
          existH nv (substF
            (substF f x (var nv)) v s)
        end
      end.

```

Let us look at a few examples. Despite the complexity of its definition, the function `substF` behaves well with respect with computations.

```

Let F : Formula L := (exH 2, v#1 <> f v#2)%fol.

```

```

Compute substF F 1 (f v#2)%fol.

```

```

= exH' 3 (f v#2 <> f v#3)
: Formula L

```

```

Compute substF (close L F -> F)%fol 1 (h v#2 v#3)%fol.

```

```

= ((allH 1, exH' 2 (v#1 <> f v#2)) ->
  exH' 4 (h v#2 v#3 <> f v#4))%fol
: Formula L

```

**Project 12.2** Is it possible to get a more readable definition of `substF` using the *Equations* plug-in [SM19] ?

### 12.4.5 Multiple substitutions

The function `subFormula` defined in `Ackermann.subAll` allows to substitute a term  $t_i$  to each free occurrence of the variable  $v_i$  in a formula  $F$ . The dependence of  $t - i$  from  $v_i$  is given through a function from `nat` to `(Term L)`.

**Check** `subAllFormula`.

```
subAllFormula
  : forall L : Language,
    Formula L -> (nat -> Term L) -> Formula L
```

```
Compute subAllFormula L
  (allH 2, P (h v#1 (h v#2 (h v#1 v#3))))%fol
  (fun x => let phi := fix phi (n: nat) :=
            match n with
            | 0 => a%fol
            | S p => (f (phi p))%fol
            end
            in phi x).
```

```
= (allH 4,
   P (h (f a) (h v#4 (h (f a) (f (f (f a))))))%fol
  : Formula L
```

## 12.5 Proofs

### 12.5.1 Proof trees

Proof trees in first-order logic are the inhabitants of the `Prf L` inductive type, defined in `Ackermann.folProof` and displayed in Figure 12.1 on the next page.

Please note that the following constructions are parametrized with an arbitrary language  $L$ , declared in the `ProofH` section.

**Section ProofH.**

**Variable L** : Language.

**Let Formula** := Formula L.

**Let Formulas** := Formulas L.

**Let System** := System L.

**Let Term** := Term L.

**Let Terms** := Terms L.

#### 12.5.1.1 Prf's type

The type `Prf l F` is the type of “proof trees of the formula  $F$ , where  $l$  is the list of assumptions used in the proof, enumerated left-to-right (*i.e.* the fringe of the proof tree)”.

We will also use (in the text) the notation  $l \vdash F$  for the type `Prf l F`.

In the rest of this section, we comment every one of `Prf`'s 14 constructors, and give simple examples of their application.

#### 12.5.1.2 Warning

We won't respect the order in which `Prf`'s constructors of type `folProof` are enumerated in `folProof.v` (see Figure 12.1). Instead, we preferred to present

```

Inductive Prf : Formulas -> Formula -> Set :=
| AXM : forall A : Formula, Prf [A] A
| MP :
  forall (Hyp1 Hyp2 : Formulas) (A B : Formula),
    Prf Hyp1 (A -> B)%fol -> Prf Hyp2 A -> Prf (Hyp1 ++ Hyp2) B
| GEN :
  forall (Hyp : Formulas) (A : Formula) (v : nat),
    ~ In v (freeVarListFormula L Hyp) -> Prf Hyp A ->
    Prf Hyp (allH v, A)%fol
| IMP1 : forall A B : Formula, Prf [] (A -> B -> A)%fol
| IMP2 :
  forall A B C : Formula,
    Prf [] ((A -> B -> C) -> (A -> B) -> A -> C)%fol
| CP :
  forall A B : Formula,
    Prf [] ((~ A -> ~ B) -> B -> A)%fol
| FA1 :
  forall (A : Formula) (v : nat) (t : Term),
    Prf [] ((allH v, A) -> substF A v t)%fol
| FA2 :
  forall (A : Formula) (v : nat),
    ~ In v (freeVarF A) -> Prf [] (A -> allH v, A)%fol
| FA3 :
  forall (A B : Formula) (v : nat),
    Prf []
      ((allH v, A -> B) -> (allH v, A) -> allH v, B)%fol
| EQ1 : Prf [] (v#0 = v#0)%fol
| EQ2 : Prf [] (v#0 = v#1 -> v#1 = v#0)%fol
| EQ3 : Prf [] (v#0 = v#1 -> v#1 = v#2 -> v#0 = v#2)%fol
| EQ4 : forall R : Relations L, Prf [] (AxmEq4 R)
| EQ5 : forall f : Functions L, Prf [] (AxmEq5 f).

```

Figure 12.1: Definition of the type Prf of proof trees

these constructors in an order inspired by a sequence of simple examples. On the other hand, we didn't change this order in `folProof.v`, in order not to break complex proofs by pattern-matching.

For more information on Hilbert proof system, you may consult [https://en.wikipedia.org/wiki/List\\_of\\_Hilbert\\_systems](https://en.wikipedia.org/wiki/List_of_Hilbert_systems).

### 12.5.1.3 Notation

#### 12.5.1.4 The axiom rule: AXM

Let  $A$  be a formula on  $L$ . The AXM rule builds a proof-tree of  $A$  which uses *exactly* the singleton list  $[A]$ .

From `MoreAck.FolExamples`

**Example PrfEx1:** `Prf L [(A -> B -> C)%fol] (A -> B -> C)%fol.`

**Proof.** `constructor. Qed.`

#### 12.5.1.5 Modus Ponens: MP

Let  $A$  and  $B$  be two formulas on  $L$ ,  $Axm1$  and  $Axm2$  two sequences of formulas. If we have two proof trees of respective types `Prf Axm1 A→B` and `Prf Axm2 B`, then we build a proof tree for  $B$  whose fringe is the concatenation of  $s_{AB}$  and  $s_A$ .

The following proof script is a quite naive application of AXM and MP.

**Lemma PrfEx2:** `Prf L [A -> B -> C; A; A -> B; A]%fol C.`

**Proof.**

```
change (Prf L ([A -> B -> C; A] ++ [A -> B; A])%fol C); eapply MP.
- change [(A -> B -> C)%fol; A] with ([A -> B -> C] ++ [A])%fol;
  eapply MP.
+ eapply AXM.
+ eapply AXM.
- change [(A -> B); A]%fol with ([A -> B] ++ [A])%fol; eapply MP.
+ eapply AXM.
+ eapply AXM.
```

**Qed.**

$$\frac{\frac{\frac{A \rightarrow B \rightarrow C}{A \rightarrow B \rightarrow C} \quad \frac{A}{A} \quad \frac{A \rightarrow B}{A \rightarrow B} \quad \frac{A}{A}}{B \rightarrow C} \quad \frac{A}{B}}{C}$$

Figure 12.2: The proof tree of `PrfEx2`

Figure 12.2 shows the tree-like structure of `PrfEx2`. The list of used hypotheses is the fringe of the tree. The unary nodes are applications of AXM and the binary nodes are associated with MP.

We can make our proof script shorter, using existential variables.

**Lemma MP' f g H1 H2 H:**  $H = H1 ++ H2 \rightarrow \text{Prf } L \text{ H1 } (f \rightarrow g)\%fol \rightarrow$   
 $\text{Prf } L \text{ H2 } f \rightarrow \text{Prf } L \text{ H } g.$

**Proof.**

intros; subst; eapply MP; eauto.

**Qed.**

(\* Cuts the current list of hypotheses as (G++H), then applies MP \*)

**Ltac cutMP G :=**  
 match goal with  
 |- Prf ?L ?H ?F => eapply MP' with (H1 := G);  
 [simpl; reflexivity | try apply AXM | try apply AXM ] end.

**Example PrfEx2':**  $\text{Prf } L [A \rightarrow B \rightarrow C; A; A \rightarrow B; A]\%fol \text{ C}.$

**Proof.**

cutMP [A -> B -> C; A]%fol.

---

Prf L [(A -> B -> C)%fol; A] (?f -> C)%fol

---

Prf L [(A -> B)%fol; A] ?f

-

---

Prf L [(A -> B -> C)%fol; A] (?f -> C)%fol

cutMP [A -> B -> C]%fol.

-

---

Prf L [(A -> B)%fol; A] B

cutMP [(A -> B)]%fol.

**Qed.**

### 12.5.1.6 Hilbert's axioms for implication: IMP1 and IMP2

#[local] **Arguments** MP {L Hyp1 Hyp2 A B} \_ \_.

**Example PrfEx3 :**  $\text{Prf } L [] (A \rightarrow A)\%fol.$

**Proof.**

pose (pf1 := IMP2 L A (A -> A)%fol A).

---

**pf1:** Prf L [] ((A -> (A -> A) -> A) -> (A -> A -> A) -> A -> A)%fol

---

Prf L [] (A -> A)%fol

pose (pf2 := IMP1 L A A).

---

**pf1:** Prf L [] ((A -> (A -> A) -> A) -> (A -> A -> A) -> A -> A)%fol

---

**pf2:** Prf L [] (A -> A -> A)%fol

---

Prf L [] (A -> A)%fol

pose (pf3 := IMP1 L A (A -> A)%fol).

```

pf1: Prf L [] ((A -> (A -> A) -> A) -> (A -> A -> A) -> A -> A)%fol
pf2: Prf L [] (A -> A -> A)%fol
pf3: Prf L [] (A -> (A -> A) -> A)%fol
-----
Prf L [] (A -> A)%fol

```

```
pose (pf4 := MP pf1 pf3).
```

```

pf1: Prf L [] ((A -> (A -> A) -> A) -> (A -> A -> A) -> A -> A)%fol
pf2: Prf L [] (A -> A -> A)%fol
pf3: Prf L [] (A -> (A -> A) -> A)%fol
pf4: Prf L ([] ++ []) ((A -> A -> A) -> A -> A)%fol
-----
Prf L [] (A -> A)%fol

```

```
exact (MP pf4 pf2).
```

**Qed.**

**Remark 12.10** One may think that this proof is quite clumsy. *Right.* But we must recall that Prf is a basic Hilbert-like proof system, which will make it easier to study. In the next chapter, we will consider a derived proof system where the *deduction theorem* will allow us to consider shorter and more natural proofs.

**Exercise 12.1 (\*\*)** Is it possible to build a term of type Prf L [A] B -> Prf L [] (A->B)%fol ?

### 12.5.1.7 The rule of contraposition: CP

The only rule about the notH connective is the *contraposition rule*.

The following script shows that CP entails the derived rule of *proof by contradiction*.

**Example PrfEx4** (A B: Formula L): Prf L [] (~B -> B -> A)%fol.

**Proof.**

```

assert (pf1 : Prf L nil (~B -> ~A -> ~B)%fol) by apply IMP1.
assert (pf2 : Prf L nil ((~A -> ~B) -> (B -> A))%fol) by apply CP.
pose (pf3 := IMP2 L (~B)%fol (~A -> ~B)%fol (B -> A)%fol).
assert (pf4: Prf L nil (~B -> (~A -> ~B) -> B -> A)%fol).
{ assert (pf5 : Prf L nil (((~A -> ~B) -> B -> A) -> ~B ->
      (~A -> ~B) -> B -> A)%fol)
  by eapply IMP1.
  apply(MP L _ _ _ _ pf5 pf2).
}
pose (pf6 := MP L _ _ _ _ pf3 pf4).
exact (MP L _ _ _ _ pf6 pf1).

```

**Defined.**

**Remark 12.11** Same remark as 12.10.

**Exercise 12.2** Replay this proof with pen and paper!

## 12.5.1.8 Rules about the universal quantifier: FA1, FA2 and FA3

**Example PrfEx5** : Prf L [] ((allH 1 2, R v#1 v#2) -> allH 2, R a v#2)%fol.

**Proof.**

```
change (allH 2, R a v#2)%fol with (substF (allH 2, R v#1 v#2)%fol 1 a).
eapply FA1.
```

**Qed.**

**Example PrfEx6** : Prf L [] (R v#1 v#1 -> allH 0, R v#1 v#1)%fol.

**Proof.**

```
apply FA2; simpl; intuition.
```

**Qed.**

**Example PrfContrex7** :

```
Prf L [] (R v#1 v#1 -> allH 1, R v#1 v#1)%fol.
```

**Proof.**

```
apply FA2; simpl.
```

```
~ (1 = 1 ∨ 1 = 1 ∨ False)
```

**Abort.**

**Example PrfEx8** : Prf L [] ((allH 0, P v#0 -> Q v#0) ->  
 (allH 0, P v#0) ->  
 (allH 0, Q v#0))%fol.

**Proof.** apply FA3. **Qed.**

## 12.5.1.9 Axioms for Equality: EQ1 to EQ3

The following proof applies FA1 in order to build a proof of Prf L [] (t=t)%fol for any  $t^4$ .

Please look also at Figure 12.3 on the facing page

**Lemma eq\_refl** (t:Term L): Prf L nil (t = t)%fol.

**Proof.**

```
assert (H: Prf L nil (allH 0, v#0 = v#0)%fol).
{
  apply GEN.
  - cbn; auto.
  - apply EQ1.
}
change (nil:(list (Formula L))) with (nil++nil: list(Formula L)).
eapply MP.
2: apply H.
apply (FA1 _ (v#0 = v#0)%fol 0 t).
```

**Defined.**

<sup>4</sup>Please note that the quantification on  $t$  is at the meta-level (Coq's level; not FOL).

$$\frac{\frac{\overline{(\forall v_0, v_0 = v_0) \rightarrow t = t} \quad FA1 \quad \frac{\overline{v_0 = v_0} \quad EQ1}{\forall v_0, v_0 = v_0} \quad GEN}}{t = t}}$$

Figure 12.3: Proof tree of (eq\_refl t)

### 12.5.1.10 Axioms schemes for Equality: EQ4 and EQ5

The constructors EQ4 and EQ5 build an infinite number of axioms, parameterized by a relation or function symbol.

The function AxmEq4 generates a formula with  $n$  pairs of free variables (with  $n$  the arity of the considered relation symbol).

**Compute** AxmEq4 L P\_.

```
= (v#0 = v#1 -> P v#0 <-> P v#1)%fol
: Formula L
```

**Example** PrfEx9: Prf L [] (v#0 = v#1 -> P v#0 <-> P v#1)%fol.

**Proof.**

```
apply (EQ4 L P_).
```

**Qed.**

**Compute** AxmEq4 L R\_.

```
= (v#2 = v#3 ->
   v#0 = v#1 -> R v#2 v#0 <-> R v#3 v#1)%fol
: Formula L
```

**Example** PrfEx10:

```
Prf L [] (v#2 = v#3 -> v#0 = v#1 -> R v#2 v#0 <-> R v#3 v#1)%fol.
```

**Proof.**

```
apply (EQ4 L R_).
```

**Qed.**

Please note that EQ4 uses a *sequence* of variables generated by AxmEq4. Any other sequence may cause EQ4 to fail.

**Example** PrfContrex9: Prf L [] (v#1 = v#0 -> P v#1 <-> P v#0)%fol.

**Proof.**

```
Prf L [] (v#1 = v#0 -> P v#1 <-> P v#0)%fol
```

```
Fail apply (EQ4 L P_).
```

```
The command has indeed failed with message:
Unable to unify "Prf L [] (AxmEq4 L P_)" with
"Prf L [] (v#1 = v#0 -> P v#1 <-> P v#0)%fol".
```

```
Prf L [] (v#1 = v#0 -> P v#1 <-> P v#0)%fol
```

**Abort.**

The following script shows that EQ5 is to function symbols what EQ4 is to relation symbols.

**Compute** AxmEq5 L h\_.

```
= (v#2 = v#3 ->
   v#0 = v#1 -> h v#2 v#0 = h v#3 v#1)%fol
: Formula L
```

**Example PrfEx11:**

```
Prf L [] (v#2 = v#3 -> v#0 = v#1 -> h v#2 v#0 = h v#3 v#1)%fol.
```

**Proof.**

```
apply (EQ5 L h_).
```

**Qed.**

## 12.6 Concluding remarks

The type `Prf` is composed of very simple rules. Nevertheless, the examples presented in the previous section seem to show that proving even simple theorems is not trivial at all.

Indeed, in the next chapter, we will consider a proof system `SysPrf`, based on `Prf`, the properties of which will allow us to prove theorems in a much simpler way.

# Chapter 13

## Natural Deduction (in construction)

### 13.1 Contexts as sets

Let us look again at the proof scripts shown in 12.5.1.5 on page 251 and 12.5.1.6 on page 252.

- The statement of `PrfEx2` contains a sequence of hypotheses with two occurrences of `A`. Moreover, the order in which the 4 hypotheses are listed is determined by the type of the constructor `MP` (please look at Figure 12.1 on page 250). It would be better to replace this precise list of hypotheses with “any list whose elements belong to the set  $\{A, A \rightarrow B; A \rightarrow B \rightarrow C\}$ ”.
- In the proof of `PrfEx3` the Coq user would certainly ask “How do we apply implication’s introduction rule?”.

The answer to both questions in [O’C05a] is the definition of a proof system, derived from `Prf`, which considers *sets of hypotheses* (called *systems* in [O’C05a]) instead of *list of hypotheses*, thus making abstraction of the repetition and order of appearance of hypotheses in the context.

The new system is simply defined as below (for a given language  $L$ )<sup>1</sup>.

```
Definition SysPrf (T : System) (f : Formula) :=  
  exists Hyp : Formulas,  
    (exists prf : Prf Hyp f,  
      (forall g : Formula, In g Hyp -> mem _ T g)).
```

In a few words, proving a statement `SysPrf _ T A` is proving the existence of a proof-tree of type `Prf Hyp A`, where `Hyp` is a list of hypotheses all elements of which belong to `T`.

---

<sup>1</sup>In some shown snippets, arguments like  $L$  may be or not be implicit (depending on the section they are extracted from). Please look at the Coq source.

### 13.1.0.1 Notations

In the text, we may use the abbreviation  $T \mid_{\mathcal{S}} A$  for  $(\text{SysPrf } L \ T \ A)$  and  $\mid_{\mathcal{S}} A$  for  $(\text{SysPrf } L \ \text{Empty\_set} \ A)$

We may also omit obvious braces in some set expressions:

- $T, U$  for  $T \cup U$ ,
- $A, B, C$  for  $\{A, B, C\}$ ,
- $T, A, \dots B$  for  $T \cup \{A \dots B\}$
- *etc.*

**Remark 13.1** The type `SysPrf` has sort `Prop`, which prevent us from extracting the underlying proof tree and its fringe from a proof of  $T \mid_{\mathcal{S}} f$ . We only know that such a proof exists, but cannot get it automatically through a `Coq` function.

**Project 13.1** It would be nice (*e.g.* for a better understanding of the proof of the deduction theorem) to be able to compute the proof-tree built by the proof of the deduction lemma. On a fresh branch of the project, please change the definition of `SysPrf` and fix the errors this change could cause in the rest of the files. *If ensuring compatibility with all the Goedel project is too long and/or difficult, you may just make the changes in separate modules with an “informative” SysPrf and limit the compatibility study to the contents of basic modules like Deduction, folLogic, etc. Perhaps you will have to consider other implementation of finite sets of formulas (e.g. lists).*

### 13.1.0.2 Example

In the following script, we use `PrfEx2` as a witness for proving a set-based version of the original proof term, namely  $A; A \rightarrow B; A \rightarrow B \rightarrow C \mid_{\mathcal{S}} C$ .

```
Example SysPrfEx2 : SysPrf L
      (fun x => List.In x [A; A->B; A -> B -> C]%fol)
      C.
```

**Proof.**

```
exists [A -> B -> C; A; A -> B; A]%fol, PrfEx2; unfold mem, In.
```

```
forall g : Formula L,
List.In g [(A -> B -> C)%fol; A; (A -> B)%fol; A] ->
List.In g [A; (A -> B)%fol; (A -> B -> C)%fol]
```

```
(* ... *)
```

**Exercise 13.1** Prove the following lemma (without the handy lemmas from `Ackermann.folLogic` and their corollaries!).

```
Lemma MPSys L (G : System L) (A B : Formula L) :
  SysPrf L G (A -> B)%fol -> SysPrf L G A -> SysPrf L G B.
```

### 13.1.1 Using properties of sets

The following three lemmas, from Ackermann.folLogic are direct consequences of SysPrf's definition.

**Lemma Axm T f:** mem \_ T f -> SysPrf T f.

**Proof.**

```
exists (f :: nil), (AXM L f).
intros g [ | []]; now subst.
```

**Qed.**

**Lemma sysExtend (T U : System) (f : Formula):**  
Included \_ T U -> SysPrf T f -> SysPrf U f.

**Proof.**

```
intros H [x [p H0]]; exists x, p.
intros g H1; apply H, H0, H1.
```

**Qed.**

**Lemma sysWeaken (T : System) (f g : Formula):**  
SysPrf T f -> SysPrf (Ensembles.Add T g) f.  
(\* ... \*)

The *rule of implication elimination* is derived from Prf's *modus ponens* MP. Since all elements of the fringe  $x$  [resp.  $x1$ ] of the proof tree  $px$  [resp.  $px1$ ] belong to  $T$ , so are the elements of the fringe  $x++x1$  of  $(MP \dots px \ px1)$ .

**Lemma impE (T : System) (f g : Formula):**  
SysPrf T (g -> f)%fol -> SysPrf T g -> SysPrf T f.

**Proof.**

```
intros [x [px Hx]] [x1 [px1 Hx1]].
set (A1 := MP L _ _ _ px px1); exists (x ++ x1), A1.
(* ... *)
```

## 13.2 The Deduction theorem

The deduction theorem (proved in Ackermann.Deduction) is a handy tool for proving a proposition  $f \rightarrow g$  by pushing the hypothesis  $f$  into the context (it corresponds roughly to the implication introduction rule in Coq).

**Theorem DeductionTheorem :**  
forall (T : System) (f g : Formula)  
(prf : SysPrf (Ensembles.Add \_ T g) f),  
SysPrf T (g -> f)%fol.

### 13.2.1 Sketch of proof

We advise the reader to replay this proof on h.er.is computer in order to better understand its structure, which we will only comment briefly.

Let us assume the hypothesis  $H: T, g \mid_S f$ , meaning that there exists some list  $F$  whose elements belong to  $T \cup \{g\}$ , and a proof-tree  $t$  of type  $F \vdash h$ .

The heart of the proof is an induction on  $t$  proving  $F \cap T \mid_{\mathbb{S}} (g \rightarrow f)$  <sup>2</sup>.

Please note that the case  $S = \text{nil}$  is common to many constructors of `Prf`, thus the proof script starts with a study of this particular case, simply applied 11 times in the rest of the proof.

### 13.3 Derived rules and natural deduction

The library `Ackermann.folLogic`, `Ackermann.folLogic2` and `Ackermann.folLogic3` contain many derived rules which allow the user to build proofs in a natural deduction style (with introduction and elimination rules).

We present here only a few examples of these rules, the reader may consult these libraries with `Search` or by looking at the `coqdoc` generated files. A meta-exercise would be to re-prove a few of these lemmas and/or build an example of application.

The *rule of implication introduction* is a trivial application of the deduction theorem.

```
Lemma impI (T : System) (f g : Formula):
  SysPrf (Ensembles.Add T g) f -> SysPrf T (g -> f)%fol.
Proof. intros ?; now apply (DeductionTheorem L). Qed.
```

The following lemma (corresponding to the `CP` constructor of type `Prf`) is proven with the help of `impE`, `impI`, `sysWeaken` and `CP`.

```
Lemma contradiction (T : System) (f g : Formula):
  SysPrf T f -> SysPrf T (~ f)%fol -> SysPrf T g.
Proof.
  intros H H0; eapply impE with f.
  - eapply impE with (~ g -> ~ f)%fol.
    + exists (nil (A:=Formula)).
      exists (CP L g f); contradiction.
    + apply impI; now apply sysWeaken.
  - assumption.
Qed.
```

We let the reader consult the proof of the following lemmas, or, much better, re-prove them as exercises.

```
Lemma nnE (T : System) (f : Formula): SysPrf T (~ ~ f)%fol -> SysPrf T f.
```

```
Lemma nnI (T : System) (f : Formula): SysPrf T f -> SysPrf T (~ ~ f)%fol.
```

```
Lemma cp1 (T : System) (f g : Formula) :
  SysPrf T (~ f -> ~ g)%fol -> SysPrf T (g -> f)%fol.
```

```
Lemma cp2 (T : System) (f g : Formula):
  SysPrf T (g -> f)%fol -> SysPrf T (~f -> ~g)%fol.
```

---

<sup>2</sup>Please forgive the implicit coercion from lists to sets!

### 13.3.1 Rules for derived connectives and quantifiers

Let us keep in mind that the derived connectives: `orH`, `andH`, etc. and the existential quantifier `existH` are defined in terms of `impH`, `notH`, `forallH`. By unfolding these definitions, we prove easily a few natural deduction rules for the derived symbols. For instance, the law of excluded middle for  $f$ :  $T \mid_{\mathcal{S}} \sim f \vee f$  for any  $T$ , is just an abbreviation of  $T \mid_{\mathcal{S}} \sim \sim f \rightarrow f$ .

**Lemma** `noMiddle` (`T` : System) (`f` : Formula): SysPrf T ( $\sim f \vee f$ )%fol.

**Proof.**

`unfold orH.` (*optional*)

**Unset Printing Notations.** (*optional*)

<code>T</code> : System
<code>f</code> : Formula
-----
SysPrf T (impH (notH (notH f)) f)

`apply impI, nnE, Axm; right; constructor.`

**Set Printing Notations.** (*optional* \*)

**Qed.**

**Remark 13.2** The lines marked “optional” are just here in order to temporarily deactivate the notation which print any formula of the form  $\sim A \rightarrow B$  as  $A \vee B$ . These three lines can be safely removed.

**Lemma** `orI1` (`T` : System) (`f g` : Formula): SysPrf T f -> SysPrf T (f  $\vee$  g)%fol.

**Proof.**

`intros H; apply impI; apply contradiction with f.`

<code>H</code> : SysPrf T f
-----
SysPrf (Ensembles.Add T ( $\sim f$ )%fol) f
<code>H</code> : SysPrf T f
-----
SysPrf (Ensembles.Add T ( $\sim f$ )%fol) ( $\sim f$ )%fol

(*... \**)

**Lemma** `orE` (`T` : System) (`f g h` : Formula):  
 SysPrf T (f  $\vee$  g)%fol ->  
 SysPrf T (f -> h)%fol -> SysPrf T (g -> h)%fol -> SysPrf T h.

**Lemma** `orSys` (`T` : System) (`f g h` : Formula):  
 SysPrf (Ensembles.Add T f) h -> SysPrf (Ensembles.Add T g) h ->  
 SysPrf (Ensembles.Add T (f  $\vee$  g)%fol) h.

**Lemma** `andI` (`T` : System) (`f g` : Formula):  
 SysPrf T f -> SysPrf T g -> SysPrf T (f  $\wedge$  g)%fol.

**Proof.**

`intros H H0; unfold andH;`

`apply orE with ( $\sim (\sim f \vee \sim g)$ )%fol ( $\sim f \vee \sim g$ )%fol.`

`- apply noMiddle.`

(*... \**)

Here are a few examples of looking for rules using Coq's Search command:

**Search** SysPrf (?A /\ ?B)%fol notH.

```
nImp:
  forall (L : Language) (T : System L)
    (f g : Formula L),
  SysPrf L T (f /\ ~ g)%fol ->
  SysPrf L T (~ (f -> g))%fol

nAnd:
  forall (L : Language) (T : System L)
    (f g : Formula L),
  SysPrf L T (~ f \/ ~ g)%fol ->
  SysPrf L T (~ (f /\ g))%fol

nOr:
  forall (L : Language) (T : System L)
    (f g : Formula L),
  SysPrf L T (~ f /\ ~ g)%fol ->
  SysPrf L T (~ (f \/ g))%fol
```

**Search** (SysPrf ?L ?T (?A /\ ?B)%fol -> SysPrf ?L ?T ?B).

```
andE2:
  forall (L : Language) (T : System L)
    (f g : Formula L),
  SysPrf L T (f /\ g)%fol -> SysPrf L T g
```

**Search** (SysPrf \_ \_ (~ ~ \_)%fol).

```
nnI:
  forall (L : Language) (T : System L) (f : Formula L),
  SysPrf L T f -> SysPrf L T (~ ~ f)%fol

nnE:
  forall (L : Language) (T : System L) (f : Formula L),
  SysPrf L T (~ ~ f)%fol -> SysPrf L T f
```

**Search** SysPrf (?a = ?b)%fol substF.

```
subWithEquals:
  forall (L : Language) (f : Formula L) (v : nat)
    (a b : Term L) (T : System L),
  SysPrf L T (a = b)%fol ->
  SysPrf L T (substF f v a -> substF f v b)%fol
```

**Search** SysPrf (exH ?v, \_)%fol (allH ?v, \_)%fol.

```

nExist:
  forall (L : Language) (T : System L) (f : Formula L)
    (v : nat),
  SysPrf L T (allH v, ~ f)%fol ->
  SysPrf L T (~ (exH v, f))%fol

nForall:
  forall (L : Language) (T : System L) (f : Formula L)
    (v : nat),
  SysPrf L T (exH v, ~ f)%fol ->
  SysPrf L T (~ (allH v, f))%fol

```

### 13.3.2 Example: proof of Peirce's law

For instance, let us prove Peirce's rule, *i.e.*  $\frac{}{\vdash ((A \rightarrow B) \rightarrow A) \rightarrow A}$  for any formulas  $A$  and  $B$ .

The proof in Coq is available in `MoreAck.FolExamples`.

**Section** `PeirceProof`.

**Arguments** `Add {U}`.

**Arguments** `Empty_set {U}`.

**Definition** `Peirce` : `Formula L := (((A -> B) -> A) -> A)%fol`.

**Lemma** `peirce` : `SysPrf L Empty_set Peirce`.

**Proof with** `auto with sets`.

Let us start with an implication introduction. The judgement to prove becomes  $(A \rightarrow B) \rightarrow A \mid \vdash A$ .

```

unfold Peirce; apply impI.

```

```

SysPrf L (Add Empty_set ((A -> B) -> A)%fol) A

```

Now, we may eliminate the disjunction in the instance  $\sim A \vee A$  of the law of excluded middle. The only non-trivial case is about  $\sim A$ .

Next goal is  $(A \rightarrow B) \rightarrow A \mid \vdash \sim A \rightarrow A$ . Please keep in mind that our current notation system interprets  $\sim A \rightarrow A$  as a disjunction!

```

eapply orE with (~A)%fol A%fol;
  [apply noMiddle | | apply impRef1].

```

```

SysPrf L (Add Empty_set ((A -> B) -> A)%fol)
  (A \/' A)%fol

```

The rest of the proof is composed of basic proof steps.

```

apply impI; eapply impE with (A -> B)%fol.

```

```

SysPrf L
  (Add (Add Empty_set ((A -> B) -> A)%fol) (~ A)%fol)
  ((A -> B) -> A)%fol

```

```

SysPrf L
  (Add (Add Empty_set ((A -> B) -> A)%fol) (~ A)%fol)
  (A -> B)%fol

```

- apply Axm ...
- apply impI; apply contradiction with A; apply Axm ...

**Qed.**

**End PeirceProof.**

**Exercise 13.2** Prove, using the rules described in Ackermann.folLogic, the famous *drinkers theorem*:

$$\exists x, (D(x) \implies \forall y, D(y))$$

where  $D$  (for “drinks”) is some predicate symbol of arity 1.

## Chapter 14

# Languages for Arithmetic (in construction)

**To do 14.1** *Make a chapter!*

Two languages built with the usual symbols of arithmetic are defined in `Ackermann.Languages`.

- The first language: `LNT` (*Language of Number Theory*) has just function symbols for  $+$ ,  $\times$ ,  $0$  and successor.
- The second language: `LNN` (*Language of Natural Numbers*) has the same function symbols as `LNT` plus one relation symbol for the strict inequality  $<$ : `LT` (less than).

### 14.0.0.1 Language of Number Theory (LNT)

First, we declare two alphabets.

```
Inductive LNTFunction : Set :=  
  | Plus_ : LNTFunction  
  | Times_ : LNTFunction  
  | Succ_ : LNTFunction  
  | Zero_ : LNTFunction.
```

```
Inductive LNTRelation : Set :=.
```

```
Definition LNTFunctionArity (x : LNTFunction) : nat :=  
  match x with  
  | Plus_ => 2  
  | Times_ => 2  
  | Succ_ => 1  
  | Zero_ => 0  
  end.
```

In a second time, we build `LNT` and `LNN` by filling `Language`'s arity field.

```
Definition LNTRelationR (x : LNTRelation) : nat :=
  match x with bot => LNTRelation_rec (fun _ => nat) bot end.
```

```
Definition LNT : Language := language LNTRelation LNTFunction LNTRelationR LNTFunctionArity.
```

**Remark 14.1** We depart a little from [O’C05a]’s notations, where the function and relation symbols are called `Plus`, `Mult`, `LT`, etc. In our version, these type constructors are called `Plus_`, `Mult_`, `LT_`, etc., while the names without final underscores are bound to term building functions (*e.g.* the function which takes two terms and builds the term representing their sum) (see Remark 12.1).

#### 14.0.0.2 Language of Natural Numbers (LNN)

LNN is an extension of LNT, by the addition of the `<` relation symbol.

```
Inductive LNNRelation : Set :=
  LT_ : LNNRelation.
```

```
Definition LNNArityR (x : LNNRelation) : nat :=
  match x with LT_ => 2 end.
```

```
Definition LNNArityF (f : LNTFunction) :=
  LNTFunctionArity f.
```

```
Definition LNN : Language := language LNNRelation LNTFunction
  LNNArityR LNNArityF.
```

#### 14.0.0.3 Examples

Let us show a few examples (from `MoreAck.FolExamples`).

```
Compute arityF LNT Plus_.
```

```
= 2
: nat
```

```
Compute arityF LNN Succ_.
```

```
= 1
: nat
```

```
Compute arityR LNN LT_.
```

```
= 2
: nat
```

```
Fail Compute arityF LNT LT_.
```

```
The command has indeed failed with message:
The reference LT was not found in the current
environment.
```

For instance the term  $v_1 + 0$ , where  $v_1$  is a variable, is represented by the following Gallina term of type `(fol.Term LNT)`.

```
(** v1 + 0 *)
Example t1_0: Term LNN :=
  apply LNN Plus_
    (Tcons (var 1)
      (Tcons (apply LNN Zero_ Tnil) Tnil )).
```

```
Definition Formula := Formula LNN.
Definition Formulas := Formulas LNN.
Definition System := System LNN.
Definition Sentence := Sentence LNN.
Definition Term := Term LNN.
Definition Terms := Terms LNN.
Definition SysPrf := SysPrf LNN.
```

```
#[local] Arguments apply _ _ _ : clear implicits.
#[local] Arguments atomic _ _ _ : clear implicits.
```

## 14.1 Notations for Formulas (experimental)

In order to get more readable terms and formulas, we can define a few notations in `MoreAck.FOL_notations` and `MoreAck.LNN`. Please note that these notation scopes are experimental: We are going to use them in examples and exercises before using them in large original proof scripts (in the `ordinals/Ackermann/` sub-directory).

We try to define notation scopes as close as possible to Coq's syntax for propositions.

Let us take for instance the following proposition (in math form):

$$\forall v_0, v_0 = 0 \vee \exists v_1, v_1 = 1 + v_0$$

Here is a definition, using directly the `goedel/Ackermann's` project syntax.

```
Definition f0 : Formula LNN :=
  forallH 0
    (orH
      (equal (var 0) Zero)
      (existH 1 (equal (var 0)
        (apply
          (Languages.Succ_ : Functions LNN)
          (Tcons (var 1) (@Tnil _)))))).
```

Note that, because of redefinitions, the disjunction `orH` can be expanded in terms of implication and negation (for instance when we use `Compute`).

**To do 14.2** *Present the general issue about evaluation, and our provisional solution.*

```
Print f0.
```

```
f0 =
(allH 0,
 v#0 = Zero \ /
 (exH 1,
  v#0 = apply (Succ_ : Functions LNN) (Tcons v#1 Tnil)))%fol
 : Formula LNN
```

**Compute** f0.

```
= (allH 0,
   v#0 = apply Zero_ Tnil \ /'
   exH' 1 (v#0 = S_ v#1))%fol
 : Formula LNN
```

**Goal** f0 = (allH 0, v#0 = Zero \ / exH 1, v#0 = Succ v#1)%fol.

---

```
f0 =
(allH 0, v#0 = Zero \ / (exH 1, v#0 = Succ v#1))%fol
```

reflexivity.

**Qed.**

# Chapter 15

## Gödel's Encoding (in construction)

### 15.1 Cantor pairing function

The library `Ackermann.cPair` defines and study Cantor's bijection from  $\mathbb{N} \times \mathbb{N}$  into  $\mathbb{N}$ . Indeed the `cPair` function used in this library is slightly different from the "usual" Cantor pairing function shown in a big part of the litterature , and Coq's standard library <sup>1</sup>. Since both versions are equivalent upto a swap of the rguments [a] and [b], we still use Russel O'Connors definitions and statements, mainly in order to not have to modify the order of sub-goals in long proofs.

#### 15.1.1 A helper function

The following function computes the sum of all natural numbers between 1 and  $n$ :  $\sum_{i=1}^n i$ .

```
Fixpoint sumToN (n : nat) : nat :=  
  match n with  
  | 0 => 0  
  | S p => S p + sumToN p  
  end.
```

```
Lemma sumToN1 n : n <= sumToN n.
```

```
Lemma sumToN1 n : n <= sumToN n.
```

The tools presented in Chapter 11 allow us to prove that `cPair` is primitive recursive.

```
#[export] Instance sumToNIsPR : isPR 1 sumToN.
```

```
Proof.
```

```
  unfold sumToN in |- *.  
  apply indIsPR with (f := fun x y : nat => S x + y).  
  apply compose2_2IsPR  
    with (f := fun x y : nat => S x)
```

---

<sup>1</sup>In <https://coq.inria.fr/distrib/current/stdLib/Coq.Arith.Cantor.html>

```

      (g := fun x y : nat => y)
      (h := plus).
- apply filter10IsPR, succIsPR.
- apply pi2_2IsPR.
- apply plusIsPR.
Qed.

```

### 15.1.2 Cantor's pairing function

In the Ackermann/Gödel projects, the Cantor pairing function is defined as below:

$$\text{cPair } a \ b = a + \sum_{i=1}^{i=a+b} i$$

**Definition** `cPair (a b : nat) := a + sumToN (a + b).`

**Compute** `cPair 4 0.`

```

= 14
: nat

```

Figure 15.1 shows a few values of `cPair a b`, where  $a$  is the line number and  $b$  the column number.

	0	1	2	3	4	...
0	0	1	3	6	10	...
1	2	4	7	11	...	
2	5	8	12	...		
3	9	13	...			
4	14	...				
...	...					

Figure 15.1: Cantor pairing function (first values)

**Remark 15.1** Compatibility with Standard lib's pairing function is stated by the equality `cPair a b = Cantor.to_nat (b, a)` for any  $a$  and  $b$ . Obviously, this swap and uncurrying of  $a$  and  $b$  doesn't change the fundamental properties of Cantor's pairing function (being a primitive recursive bijection, monotony properties).

#### 15.1.2.1 Main properties

In order to prove that the function `cPair` is primitive recursive, we express it as a composition of already proven primitive recursive functions.

**[export]** **Instance** `cPairIsPR : isPR 2 cPair.`

**Proof.**

```

unfold cPair; apply compose2_2IsPR
with

```

```

(f := fun x y : nat => x)
(g := fun x y : nat => sumToN (x + y))
(h := plus).
- apply pi1_2IsPR.
- apply compose2_1IsPR; [apply plusIsPR | apply sumToNIsPR].
- apply plusIsPR.
Qed.

```

cPair's injectivity is stated by two lemmas:

```

Lemma cPairInj1 a b c d : cPair a b = cPair c d -> a = c.
Lemma cPairInj2 a b c d : cPair a b = cPair c d -> b = d.

```

### 15.1.3 Projections

Let  $a$  be some natural number, we look for two natural numbers  $x$  and  $y$  such that  $\text{cPair } xy = a$ . More we want to prove that the function which compute  $x$  [resp.  $y$ ] out of  $a$  are primitive recursive.

Let us show on a small example how these projections are defined. Let's take for instance  $a = 11$ . Please look at the diagram of Fig. 15.1.

The following function looks for the anti-diagonal  $\{(x, y) | x + y = k\}$  the number 11 belongs to, and returns  $k$ .

```

Let searchXY (a : nat) :=
  boundedSearch (fun a y : nat => ltBool a (sumToN y.+1)) a.

```

In our example, 11 belongs to the anti-diagonal  $\{(x, y) | x + y = 4\}$ , which contains values from 10 to 14. Thus the line containing 11 is the line  $x = 11 - 10 = 1$ , and the column is  $y = 4 - 1 = 3$ . Finally, we get  $11 = \text{cPair } 13$ .

```

Definition cPairPi1 (a : nat) := a - sumToN (searchXY a).
Definition cPairPi2 (a : nat) := searchXY a - cPairPi1 a.

```

The following lemmas (still from Ackermann.cPair) prove the correctness of these projections.

```

Lemma cPairProjections a : cPair (cPairPi1 a) (cPairPi2 a) = a.
Lemma cPairProjections1 (a b : nat) : cPairPi1 (cPair a b) = a.
Lemma cPairProjections2 (a b : nat) : cPairPi2 (cPair a b) = b.
#[export] Instance cPairPi1IsPR : isPR 1 cPairPi1.
#[export] Instance cPairPi2IsPR : isPR 1 cPairPi2.

```

Finally, let us show a few inequalities.

```

Lemma cPairLe1 (a b : nat) : a <= cPair a b.
Lemma cPairLe1A (a : nat) : cPairPi1 a <= a.
Lemma cPairLe2 (a b : nat) : b <= cPair a b.
Lemma cPairLe2A (a : nat) : cPairPi2 a <= a.

Lemma cPairLe3 (a b c d : nat) : a <= b -> c <= d -> cPair a c <= cPair b d.
Lemma cPairLt1 (a b : nat) : a < cPair a (S b).
Lemma cPairLt2 (a b : nat) : b < cPair (S a) b.

```

### 15.1.4 List encoding

The encoding of a list of natural numbers is based on `cPair`, through a structural recursion.

```
Fixpoint codeList (l : list nat) : nat :=
  match l with
  | nil => 0
  | n :: l' => S (cPair n (codeList l'))
  end.
```

```
Compute codeList (3::1::nil).
```

```
Compute codeList (2::3::1::nil).
```

Let us look at the main step of the proof that `codeList` is injective.

```
Lemma codeListInj (l m : list nat): codeList l = codeList m -> l = m.
```

```
Proof.
```

```
(* ... *)
```

```
a: nat
l: list nat
Hrecl: forall m : list nat,
  codeList l = codeList m -> l = m
n: nat
l0: list nat
H: (cPair a (codeList l)).+1
  =
  (cPair n (codeList l0)).+1
-----
a :: l = n :: l0
```

Applying `cPair`'s injectivity and the induction hypothesis allows us to complete the proof.

#### 15.1.4.1 Encoding the *n*th function

The following function allows us to compute the *n*-th element of a list of natural numbers, *directly on the encoding of the considered list*.

In the current version of the Ackermann Library, this function is defined *via* an interactive proof. Its specification and correctness are proved by separate lemmas.

```
Definition codeNth (n m:nat) : nat :=
  let X := nat_rec (fun _ : nat => nat)
             m
             (fun _ Hrecn : nat => cPairPi2 (pred Hrecn)) n
  in cPairPi1 (pred X).
```

```
Lemma codeNthCorrect :
```

```
forall (n : nat) (l : list nat), codeNth n (codeList l) = nth n l 0.
```

```
#[export] Instance codeNthIsPR : isPR 2 codeNth.
```

**Exercise 15.1** Give a new equivalent definition of `codeNth`, without using tactics like `assert`. You may define an help function for this purpose, in which case, please specify what your helper computes.

**Exercise 15.2** Please consider the following definition:

```
From hydras.Ackermann Require Import primRec cPair.
From Coq Require Import Arith.
From Equations Require Import Equations.
```

```
Equations members (a:nat): list nat by wf a:=
members 0 := List.nil;
members (S z) := cPairPi1 z :: members (cPairPi2 z).
```

Prove that the functions `codeList` and `members` are inverse of each other.

### 15.1.5 Strong recursion

List encoding helps us to define primitive recursive functions where the computation of  $f\ n$  may depend of [part of] the values  $f\ 0, f\ 1, \dots, f\ (n - 1)$ .

A simple example is the `div2` function defined by

$$\begin{aligned} \text{div2 } 0 &= 0 \\ \text{div2 } 1 &= 1 \\ \text{div2 } (n + 2) &= S(\text{div2 } n) \end{aligned}$$

The trick is to define a helper  $h$  where  $h\ n\ a$  is the natural number which encodes the sequence  $\langle \text{div2}(n - 1), \text{div2}(n - 2), \dots, \text{div2 } 1, \text{div2 } 0 \rangle$ .

In Coq, the helper associated with `div2` is defined in `MoreAck.PrimRecExamples`.

```
Let fdiv2 : naryFunc 2 :=
  fun (n acc: nat) =>
    match n with
    | 0 | 1 => 0
    | _ => S (codeNth 1 acc)
    end.
```

The function `(evalStrongRec n h c)` computes  $(f\ c)$  if  $h$  is the helper associated with  $f$ . This function is defined in `Ackermann.cPair`.

```
Compute evalStrongRec _ fdiv2 0.
```

```
= 0
: nat
```

```
Compute evalStrongRec _ fdiv2 2.
```

```
= 1
: nat
```

```
Compute evalStrongRec _ fdiv2 3.
```

```
= 1
: nat
```

```
Compute evalStrongRec _ fdiv2 4.
```

```
= 2
: nat
```

Trying to compute the half of 5 this way resulted in unbearably long computation times ....

Definitions and related lemmas are quite tricky, but look easy to apply.

```
Definition evalStrongRecHelp (n : nat) (f : naryFunc n.+2) :
  naryFunc n.+1 :=
  evalPrimRecFunc n (evalComposeFunc n 0 (Vector.nil _) (codeList nil))
    (evalComposeFunc n.+2 2
      (Vector.cons _ f _
        (Vector.cons _ (evalProjFunc n.+2 n
          (Nat.lt_lt_succ_r _ _
            (Nat.lt_succ_diag_r _))) _
          (Vector.nil _)))
      (fun a b : nat => S (cPair a b))).
```

```
Definition evalStrongRec (n : nat) (f : naryFunc n.+2):
  naryFunc n.+1 :=
  evalComposeFunc n.+1 1
    (Vector.cons _
      (fun z : nat => evalStrongRecHelp n f z.+1) _ (Vector.nil _))
    (fun z : nat => cPairPil z.-1).
```

```
#[export] Instance
evalStrongRecIsPR (n : nat) (f : naryFunc n.+2):
  isPR _ f -> isPR _ (evalStrongRec n f).
```

```
Lemma computeEvalStrongRecHelp :
  forall (n : nat) (f : naryFunc n.+2) (c : nat),
  evalStrongRecHelp n f c.+1 =
  compose2 n (evalStrongRecHelp n f c)
    (fun a0 : nat =>
      evalComposeFunc n 2
        (Vector.cons (naryFunc n) (f c a0) 1
          (Vector.cons (naryFunc n) (evalConstFunc n a0) 0
            (Vector.nil (naryFunc n))))
        (fun a1 b0 : nat => S (cPair a1 b0))).
```

**Exercise 15.3** Prove formally that this implementation of `div2` is correct.

**Exercise 15.4** Define a function for computing the Fibonacci numbers by strong recursion.

## 15.2 First order logic and Gödel encoding

To do 15.1 *Add comments!*

**Section** `Check_Proof`.

**Generalizable All Variables.**

**Variable** `L` : Language.

**Context** `(cL: Lcode L cf cr).

**Variable** `codeArityF` : nat -> nat.

**Variable** `codeArityR` : nat -> nat.

**Context** (`codeArityFIsPR` : isPR 1 codeArityF).

**Hypothesis** `codeArityFIsCorrect1` :

forall `f` : Functions L, codeArityF (cf f) = S (arityF L f).

**Hypothesis** `codeArityFIsCorrect2` :

forall `n` : nat, codeArityF n <> 0 ->  
exists `f` : Functions L, cf f = n.

**Context** (`codeArityRIsPR` : isPR 1 codeArityR).

**Hypothesis**

`codeArityRIsCorrect1` :

forall `r` : Relations L, codeArityR (cr r) = S (arityR L r).

**Hypothesis**

`codeArityRIsCorrect2` :

forall `n` : nat, codeArityR n <> 0 ->  
exists `r` : Relations L, cr r = n.

**Hypothesis** `codeFInj` : forall `f g` : Functions L,

cf f = cf g -> f = g.

**Hypothesis** `codeRInj` :

forall `R S` : Relations L, cr R = cr S -> R = S.

**#[export] Instance** `checkPrfIsPR` : isPR 2 checkPrf.

**Lemma** `checkPrfCorrect1` (`l` : list Formula) (`f` : Formula) (`p` : Prf l f):

checkPrf (codeFormula f) (codePrf l f p)

= S (codeList (map codeFormula l)).

**Lemma** `checkPrfCorrect2` (`n m` : nat):

checkPrf n m <> 0 ->

exists `f` : Formula,

codeFormula f = n /\

(exists `l` : list Formula,

(exists `p` : Prf l f, codePrf l f p = m)).



## Chapter 16

# Every Primitive Recursive Function is representable



## Part III

# A few certified algorithms



# Chapter 17

## Smart computation of $x^n$

### 17.1 Introduction

Nothing looks simpler than writing a function for computing  $x^n$ . But on the contrary, this simple programming exercise allows us to address advanced programming techniques such as:

- monadic programming, and continuation passing style
- type classes, and generalized rewriting
- proof engineering, in particular proof reuse
- proof by reflection
- polymorphism and parametricity
- composition of correct programs, etc.

### 17.2 Some basic implementations

Let us start with a very naive way of computing the  $n$ -th power of  $x$ , where  $n$  is a natural number and  $x$  belongs to some type for which a multiplication and an identity element are defined.

*From Module `additions.FirstSteps`*

**Section Definitions.**

```
Variables (A : Type)
  (mult : A -> A -> A)
  (one : A).

#[local] Infix "*" := mult.
#[local] Notation "1" := one.

(** Naive (linear) implementation *)
```

```

Fixpoint power (x:A)(n:nat) : A :=
  match n with
  | 0%nat => 1
  | S p => x * x ^ p
  end
where "x ^ n" := (power x n).

```

An application of this function for computing  $x^n$  needs  $n$  multiplications. Despite this lack of efficiency, and thanks to its simplicity, we keep it as a specification for more efficient and complex exponentiation algorithms. A function will be considered a *correct* exponentiation function if we can prove it is extensionally equivalent to `power`.

### 17.2.1 A logarithmic exponentiation function

Using the following equations, we can easily define a polymorphic exponentiation whose application requires only a logarithmic number of multiplications.

$$x^1 = x \tag{17.1}$$

$$x^{2p} = (x^2)^p \tag{17.2}$$

$$x^{2p+1} = (x^2)^p \times x \tag{17.3}$$

$$x^1 \times a = x \times a \tag{17.4}$$

$$x^{2p} \times a = (x^2)^p \times a \tag{17.5}$$

$$x^{2p+1} \times a = (x^2)^p \times (a \times x) \tag{17.6}$$

In equalities 17.4 to 17.6, the variable  $a$  plays the role of an *accumulator* whose initial value (set by 17.3) is  $x$ . This accumulator helps us to get a tail-recursive implementation.

For instance, the computation of  $2^{14}$  can be decomposed as follows:

$$\begin{aligned}
 2^{14} &= 4^7 \\
 &= 16^3 \times 4 \\
 &= 256^1 \times (4 \times 16) \\
 &= 16384
 \end{aligned}$$

With the same notations as in Sect 17.2 on the preceding page, we can implement this algorithm in Gallina. The following definitions are still within the scope of the section open in 17.2 on the previous page.

*From Module additions.FirstSteps*

```

Fixpoint binary_power_mult (x a:A)(p:positive) : A
:=
  match p with
  | xH => a * x
  | x0 q => binary_power_mult (x * x) a q
  | xI q => binary_power_mult (x * x) (a * x) q
  end.

```

```

Fixpoint Pos_bpow (x:A)(p:positive) :=
  match p with
  | xH => x
  | xO q => Pos_bpow (x * x) q
  | xI q => binary_power_mult (x * x) x q
  end.

```

```

Definition N_bpow x (n:N) :=
  match n with
  | 0%N => 1
  | Npos p => Pos_bpow x p
  end.

```

Let us close the section `Definitions` and mark the argument `A` as implicit.

**End Definitions.**

**Arguments** N\_bpow {A}.

**Arguments** power {A}.

**Remark 17.1** Our function `Pos_bpow` can be considered as a tail recursive variant of the following function defined in `Coq.PArith.BinPosDef`.

```

Definition iter_op {A}(op:A->A->A) :=
  fix iter (p:positive)(a:A) : A :=
  match p with
  | 1 => a
  | p~0 => iter p (op a a)
  | p~1 => op a (iter p (op a a))
  end.

```

This scheme is used in `Coq.ZArith.Zpow_alt` in order to define a logarithmic exponentiation `Zpower_alt` on `Z` (notation :  $x^{^p}$ ).

**Remark** Note that closing the section `Definitions` makes us lose the handy notations `_ * _` and `one`. Fortunately, *operational type classes* will help us to define nice infix notations for polymorphic functions (Sect. 17.3.1 on page 288).

## 17.2.2 Examples of computation

It is now possible to test our functions with various interpretations of  $\times$  and 1:

**Compute** power Z.mul 1%Z 2%Z 10.

```

= 1024%Z
: Z

```

**Compute** N\_bpow Z.mul 1%Z 2%Z 10.

```
= 1024%Z
: Z
```

**Open Scope** string\_scope.

**Compute** power append "" "ab" 12.

```
= "abababababababababab"
: string
```

**Compute** N\_bpow append "" "ab" 12.

```
= "abababababababababab"
: string
```

### 17.2.3 Computing Fibonacci numbers

The sequence of Fibonacci numbers is defined by the following equations:

$$F_0 = 1 \quad (17.7)$$

$$F_1 = 1 \quad (17.8)$$

$$F_n = F_{n-1} + F_{n-2} \quad (n \geq 2) \quad (17.9)$$

In Coq, one can define this function by simple recursion.

*From Library additions.Fib2*

```
Fixpoint fib (n:nat) : N :=
  match n with
  | 0%nat | 1%nat => 1
  | (S ((S p) as q)) => fib p + fib q
  end.
```

**Compute** fib 20.

```
= 10946
: N
```

In [BC04a], several exercises <sup>1</sup> present ways to compute Fibonacci numbers, with the less number of recursive calls as possible. Please note that these optimizations and the formal proof of their correctness are *ad-hoc*, *i.e.*, exclusively written for the Fibonacci numbers. In contrast, the optimizations we present in this document apply, in their vast majority, *generic* techniques of efficient computation of powers in a monoid. This example of Fibonacci numbers has been developed with Yves Bertot, who wrote a first version with SSreflect/Mathcomp [MT18]. This original version is available on hydra-battles in the module additions.fib.

<sup>1</sup>Exercises 9.8 (page 270), 9.10 (page 271), 9.15 (page 276), 9.17 (page 284), and 15.8 (page 418).

### 17.2.3.1 Using 2x2 integer matrices

The following properties are well known. They are left as an exercise, since they are not part of our development.

**Exercise 17.1** 1. Prove in Coq the following equality (for any  $n \geq 2$ ).

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix} = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

2. Infer (still in Coq) the following equality (still for  $n \geq 2$ ).

$$\begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$$

3. Write a function using the previous equality for computing the  $n$ -th Fibonacci number, and prove its equivalence with `fib`.

### 17.2.3.2 Removing duplicate computations

Yves Bertot's optimization relies on the observation that all the powers of  $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$  have the form  $\begin{pmatrix} a+b & a \\ a & b \end{pmatrix}$  where  $a$  and  $b$  are natural numbers.

Thus, it is possible to remove duplicate data and computations by reflecting matrix multiplication and identity into  $\mathbb{N} \times \mathbb{N}$ .

If we pose  $\varphi(a, b) = \begin{pmatrix} a+b & a \\ a & b \end{pmatrix}$ , then  $\varphi(a, b) \times \varphi(c, d) = \varphi(ac + ad + bc, ac + bd)$ , and  $\varphi(0, 1) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ .

**Exercise 17.2** Prove formally these properties. *Please note that their proof is not needed in our development, they just help to understand the following optimization.*

So, let us define a binary operation, which makes  $\mathbb{N} \times \mathbb{N}$  a monoid (with  $(0, 1)$  as neutral element).

*From Library additions.Fib2*

*The Monoid type class is defined page 290.*

```

Definition mul2 (p q : N * N) :=
  match p, q with
  (a, b), (c, d) => (a*c + a*d + b*c, a*c + b*d)
  end.

#[ global ] Instance Mul2 : Monoid mul2 (0,1).

```

```

Monoid mul2 (0, 1)

```

The following lemma is a simplification of the equality of Exercise 17.1.

```

Lemma next_fib (n:nat) : mul2 (1,0) (fib (S n), fib n) =
  (fib (S (S n)), fib (S n)).

```

Let us consider a new definition of the Fibonacci function.

```
Definition fib_mul2 n := let (a,b) := power (M:=Mul2) (1,0) n
                        in (a+b).
```

```
Compute fib_mul2 20.
```

```
= 10946
: N
```

```
Lemma fib_mul2_OK_0 (n:nat) :
  power (M:=Mul2) (1,0) (S (S n)) =
  (fib (S n), fib n).
```

**Proof.**

```
  induction n.
  (* ... *)
```

```
Lemma fib_mul2_OK n : fib n = fib_mul2 n.
```

```
Time Compute fib_mul2 87.
```

```
= 1100087778366101931
: N
Finished transaction in 0.004 secs (0.004u,0.s) (successful)
```

Thus, any function able to compute more or less efficiently powers in a monoid will give an algorithm for computing Fibonacci numbers. Unlike the *ad-hoc* aforementioned proofs of [BC04a], the correctness of such an algorithm is a direct consequence of the correctness of the used powering function. Several examples will be presented in the rest of this document (in Section 17.4.6 on page 300).

**To do 17.1** *Document the files contributed by Yves*

- *additions/fib.v (to rename ?)*
- *additions/stub.ml (to keep inside theories/ or move to src/ ?)*
- *theories/additions/make\_fib\_tests.txt (to put in a Makefile?)*

## 17.2.4 Formal specification of an exponentiation function: a first attempt

Let us compare the functions `power` and `N_bpow`. The first one is obviously correct, since it is a straightforward translation of the mathematical definition. The second one is much more efficient, but it is not obvious that its 18-line long definition is bug-free. Thus, we must prove that the two functions are extensionally equal (taking into account conversions between `N` and `nat`).

More abstractly, we can define a predicate that characterizes any correct implementation of `power`, this “naive” function being a *specification* of any polymorphic exponentiation function.

First, we define a type for any such function.

```

Definition power_t := forall (A:Type)
  (mult : A -> A -> A)
  (one:A)
  (x:A)
  (n:N), A.

```

Then, we would say that a function `f:power_t` is a correct exponentiation function if it is extensionally equal to `power`.

**Module** Bad.

```

Definition correct_expt_function (f : power_t) : Prop :=
  forall A (mult : A -> A -> A) (one:A)
    (x:A) (n:N), power mult one x (N.to_nat n) =
      f A mult one x n.

```

Unfortunately, our definition of `correct_expt` is too general. It suffices to build an interpretation where the multiplication is not associative or `one` is not a neutral element to obtain different results through the two functions.

**Section** CounterExample.

```

Let mul (n p : nat) := n + 2 * p.
Let one := 0.

```

```

(** With our fake definition, [N_bpow] is not correct! *)

```

```

Remark mul_not_associative :
  exists n p q, mul n (mul p q) <> mul (mul n p) q.

```

**Proof.**

```

exists 1, 1, 1; discriminate.

```

**Qed.**

```

Remark one_not_neutral :
  exists n : nat, mul one n <> n.

```

**Proof.**

```

exists 1; discriminate.

```

**Qed.**

```

Lemma correct_exp_too_strong : ~ correct_expt_function (@N_bpow).

```

**Proof.**

```

intro H; specialize (H _ mul one 1 7%N).
discriminate H.

```

**Qed.**

**End** CounterExample.

**End** Bad.

So, we will have to improve our definition of correctness, by restricting the universal quantification to associative operations and neutral elements, *i.e.*, by considering *monoids*. An exponentiation function will be considered as correct if it returns always the same result as `power` in any monoid.

## 17.3 Representing monoids in Coq

In this section, we present a “minimal” algebraic framework in which exponentiation can be defined and efficiently implemented.

Exponentiation is built on multiplication, and many properties of this operation are derived from the associativity of multiplication. Furthermore, if we allow the exponent to be any natural number, including 0, then we need to consider a neutral element for multiplication.

The structure on which we define exponentiation is called a *monoid*. It is composed of a *carrier*  $A$ , an associative binary operation  $\times$  on  $A$ , and a neutral element  $\mathbf{1}$  for  $\times$ . The required properties of  $\times$  and  $\mathbf{1}$  are expressed by the following equations:

$$\forall x y z : A, x \times (y \times z) = (x \times y) \times z \quad (17.10)$$

$$\forall x : A, x \times \mathbf{1} = \mathbf{1} \times x = x \quad (17.11)$$

In Coq, we define the monoid structure in terms of *type classes* [SO08, SvdW11]. The tutorial on type classes [CS] gives more details on type classes and operational type classes, also illustrated with the monoid structure.

First, we define a class and a notation for representing multiplication operators, then we use these definitions for defining the `Monoid` type class.

### 17.3.1 A common notation for multiplication

*Operational type classes* [SvdW11] allow us to define a common notation for multiplication in any algebraic structure. First, we associate a class to the notion of *multiplication* on any type  $A$ .

From *Module additions/Monoid\_def.v*.

```
Class Mult_op (A:Type) := mult_op : A -> A -> A.
```

```
Print Mult_op.
```

```
Mult_op =
fun A : Type => A -> A -> A
  : Type -> Type

Arguments Mult_op A%type_scope
```

From the type theoretic point of view, the term  $(\text{Mult\_op } A)$  is  $\beta\delta$ -reducible to  $A \rightarrow A \rightarrow A$ , and if  $op$  has type  $(\text{Mult\_op } A)$ , then  $(@mult\_op A op)$  is convertible with  $op$ .

```
Goal forall A (op: Mult_op A), @mult_op A op = op.
reflexivity.
Qed.
```

We are now ready to define a new notation scope, in which the notation  $x * y$  will be interpreted as an application of the function `mult_op`.

```

Delimit Scope M_scope with M.
Infix "*" := mult_op : M_scope.
Open Scope M_scope.

```

Let us show two examples of use of the notation `scope M_scope`. Each example consists in declaring an instance of `Mult_op`, then type checking or evaluating a term of the form `x * y` in `M_scope`.

Note that, since the reserved notation `"_ * _"` is present in several scopes such as `nat_scope`, `Z_scope`, `N_scope`, etc., in addition to `M_scope`, the user should take care of which scopes are active — and with which precedence — in a Gallina term. In case of doubt, explicit scope delimiters should be used.

### 17.3.1.1 Multiplication on Peano numbers

Multiplication on type `nat`, called `Nat.mul` in Standard Library, has type `nat -> nat -> nat`, which is convertible with `Mult_op nat`. Thus the following definition is accepted:

```

Module Demo.

```

```

  #[local] Instance nat_mult_op : Mult_op nat := Nat.mul.

```

Inside `M_scope`, the expression `3 * 4` is correctly read as an application of `mult_op`. Nevertheless this term is convertible with `Nat.mul 3 4`, as shown by the interaction below.

*From Module additions.Monoid\_def*

```

Set Printing All.

```

```

Check 3 * 4.

```

```

@mult_op nat nat_mult_op (S (S (S 0)))
(S (S (S (S 0))))
: nat

```

```

Unset Printing All.

```

```

Compute 3 * 4.

```

```

= 12
: nat

```

```

End Demo.

```

### 17.3.1.2 String concatenation

We can use the notation `"_ * _"` for other types than numbers. In the following example, the expression `"abc" * "def"` is interpreted as `@mult_op string ?X "abc" "def"`, then the type class mechanism replaces the unknown `?X` with `string_op`.

*From Module additions.Monoid\_def*

```
#[ global ] Instance string_op : Mult_op string := append.
Open Scope string_scope.
```

```
Example ex_string : "ab" * "cde" = "abcde".
```

```
Proof. reflexivity. Qed.
```

### 17.3.1.3 Solving ambiguities

Let  $A$  be some type, and let us assume there are several instances of `Mult_op A`. For solving ambiguity issues, one can add a *precedence* to each instance declaration of `Mult_op A`. In any case, such ambiguity can be addressed by explicitly providing some arguments of `mult_op`. For instance, in Sect. 17.3.3.2 on the facing page, we consider various monoids on types `nat` and `N`.

## 17.3.2 The Monoid type class

We are now ready to give a definition of the `Monoid` class, using `*` as an infix operator in scope `%M` for the monoid multiplication.

The following class definition, from Module `additions.Monoid_def`, is parameterized with some type  $A$ , a multiplication (called `op` in the definition), and a neutral element `1` (called `one` in the definition).

```
Class Monoid {A:Type}(op : Mult_op A)(one : A) : Prop :=
{
  op_assoc : forall x y z, x * (y * z) = x * y * z;
  one_left : forall x, one * x = x;
  one_right : forall x, x * one = x
}.

```

## 17.3.3 Building instances of Monoid

Let  $A$  be some type,  $op$  an instance of `Mult_op A` and  $one: A$ . In order to build an instance of `(Monoid A op one)`, one has to provide proofs of “monoid axioms” `op_assoc`, `one_left` and `one_right`.

Let us show various instances, which will be used in further proofs and examples. Complete definitions and proofs are given in File `additions/Monoid_instances.v`.

### 17.3.3.1 Monoid on Z

The following monoid allows us to compute powers of integers of arbitrary size, using type `Z` from standard library:

```
#[ global ] Instance Z_mult_op : Mult_op Z := Z.mul.
```

```
#[ global ] Instance ZMult : Monoid Z_mult_op 1.
```

```
Proof.
  split.
```

```
forall x y z : Z, (x * (y * z))%M = (x * y * z)%M
forall x : Z, (1 * x)%M = x
forall x : Z, (x * 1)%M = x
```

```
all: unfold Z_mult_op, mult_op; intros; ring.
```

**Qed.**

### 17.3.3.2 Monoids on type nat and N

We define two monoids on type nat:

- The “natural” monoid  $(\mathbb{N}, \times, 1)$  :

```
#[ global ] Instance nat_mult_op : Mult_op nat | 5 := Nat.mul.
```

```
#[ global ] Instance Natmult : Monoid nat_mult_op 1%nat | 5.
```

**Proof.**

```
split; unfold nat_mult_op, mult_op; intros; ring.
```

**Qed.**

- The “additive” monoid  $(\mathbb{N}, +, 0)$ . This monoid will play an important role in correctness proofs of complex exponentiation algorithms. Its most important property is that the  $n$ -th power of 1 is equal to  $n$ . See Sect. 17.7.4 on page 313 for more details.

```
#[ global ] Instance nat_plus_op : Mult_op nat | 12 := Nat.add.
```

```
#[ global ] Instance Natplus : Monoid nat_plus_op 0%nat | 12.
```

**Proof.**

```
split; unfold nat_plus_op, mult_op; intros; ring.
```

**Qed.**

Similarly, instances `NMult` and `NPlus` are built for type `N`, and `PMult` for type `positive`.

### 17.3.3.3 Machine integers

Cyclic numeric types are good candidates for testing exponentiations with big exponents, since the size of data is bounded.

The type `int31` is defined in Module `Coq.Numbers.Cyclic.Int31.Int31` of Coq’s standard library. The tactic `ring` works with this type, and helps us to register an instance `Int31Mult` of class `Monoid int31_mult_op 1`.

```
#[ global ] Instance int63_mult_op : Mult_op int := mul.
```

```
#[ global ] Instance Int63mult : Monoid int63_mult_op 1.
```

**Proof.**

```
split; unfold int63_mult_op, mult_op; intros; ring.
```

**Qed.**

Beware that machine integers are not natural numbers!

**Module** `Bad`.

```

Fixpoint int63_from_nat (n:nat) :int :=
  match n with
  | 0 => 1
  | S p => 1 + int63_from_nat p
  end.

Coercion int63_from_nat : nat >-> int.

Fixpoint fact (n:nat) : int := match n with
  0 => 1
  | S p => n * fact p
  end.

Compute fact 160.

```

```

= 0
: int

```

**End** `Bad`.

### 17.3.4 Matrices on a semi-ring

Let  $(A, +, \times)$  be a semi-ring. We define a multiplicative monoid on the set of *e.g.*  $2 \times 2$ -square matrices over  $A$ . It suffices to define an instance of `Monoid` within the scope of a hypothesis of type `semi_ring_theory`.

**Section** `M2_def`.

```

Variables (A:Type)
  (zero one : A)
  (plus mult : A -> A -> A).

```

**Notation** `"0"` := zero.

**Notation** `"1"` := one.

**Notation** `"x + y"` := (plus x y).

**Notation** `"x * y"` := (mult x y).

**Variable** `rt` : semi\_ring\_theory zero one plus mult (@eq A).

**Add** Ring Aring : rt.

```

Structure M2 : Type := {c00 : A; c01 : A;
  c10 : A; c11 : A}.

```

**Definition** `Id2` : M2 := Build\_M2 1 0 0 1.

```

Definition M2_mult (m m':M2) : M2 :=
  Build_M2 (c00 m * c00 m' + c01 m * c10 m')
  (c00 m * c01 m' + c01 m * c11 m')

```

```

      (c10 m * c00 m' + c11 m * c10 m')
      (c10 m * c01 m' + c11 m * c11 m').
#[global] Instance M2_op : Mult_op M2 := M2_mult.

#[global] Instance M2_Monoid : Monoid M2_op Id2.
(* ... *)

```

### 17.3.5 Monoids and equivalence relations

In some contexts, the “axioms” of the `Monoid` class may be too restrictive. For instance, consider multiplication in  $\mathbb{Z}/m\mathbb{Z}$  where  $1 < m$ . Although it could be possible to compute with values of the dependent type  $\{n:\mathbb{N} \mid n < m\}$ , it looks simpler to compute with numbers of type `N` and consider the multiplication  $x \times y \bmod m$ .

It is easy to prove that this operation is associative, using library `NArith`. Unfortunately, the following proposition is false in general (left as an exercise).

$$\forall x : N, (1 * x) \bmod m = x$$

Thus, we define a more general class, parameterized by an equivalence relation `Aeq` on a type `A`, compatible with the multiplication `*`. The laws of associativity and neutral element are not expressed as Leibniz equalities but as equivalence statements:

First, let us define an operational type class for equivalence relations:

*From Module additions.Monoid\_def*

```

Class Equiv A := equiv : relation A.

Infix "==" := equiv (at level 70) : type_scope.

```

The definition of class `EMonoid` looks like `Monoid`’s definition, plus some constraints on `E_eq`.

Please look for instance at our tutorial on type classes and relations [CS] for understanding the use of type classes `Equivalence`, `Reflexive`, `Proper`, etc, in relation with tactics like `rewrite`, `reflexivity`, etc., in proofs which involve equivalence relations instead of equality.

```

Class EMonoid (A:Type)(E_op : Mult_op A)(E_one : A)
  (E_eq: Equiv A): Prop :=
{
  Eq_equiv :=> Equivalence equiv;
  Eop_proper : Proper (equiv ==> equiv ==> equiv) E_op;
  Eop_assoc : forall x y z, x * (y * z) == x * y * z;
  Eone_left : forall x, E_one * x == x;
  Eone_right : forall x, x * E_one == x
}.

```

#### 17.3.5.1 Coercion from Monoid to EMonoid

Every instance of class `Monoid` can be transformed into an instance of `EMonoid`, considering Leibniz’ equality `eq`. Thus, our definitions and theorems about

exponentiation will take place as much as possible within the more generic framework of `EMonoids`.

```
#[global] Instance eq_equiv {A} : Equiv A := eq.
```

```
#[global] Instance Monoid_EMonoid `(M:@Monoid A op one) :
  EMonoid op one eq_equiv.
```

**Proof.**

```
split; unfold eq_equiv, equiv in *.
- apply eq_equivalence.
- intros x y H z t H0; now subst.
- intros;now rewrite (op_assoc).
- intro; now rewrite one_left.
- intro;now rewrite one_right.
```

**Qed.**

**Remark 17.2** In the definition of `Monoid_EMonoid`, the free variables `A`, `op` and `one` are automatically generalized thanks to the *backquote* syntax (see the section about implicit generalization in the reference manual [Coq]).

Thanks to the following *coercion*, every instance of `Monoid` can now be considered as an instance of `EMonoid`. For more details, please look at the section *Implicit Coercions* of Coq’s reference manual [Coq].

```
Coercion Monoid_EMonoid : Monoid >-> EMonoid.
```

*From Module additions.Monoid\_instances*

```
Check NMult : EMonoid N.mul 1%N eq.
```

```
NMult : EMonoid N.mul 1 eq
      : EMonoid N.mul 1 eq
```

### 17.3.5.2 Example : Arithmetic modulo $m$

The following instance of `EMonoid` describes the set of integers modulo  $m$ , where  $m$  is any integer greater than or equal to 2. For simplicity’s sake, we represent such values using the `N` type, and consider “equivalence modulo  $m$ ” instead of equality.

**Section** Nmodulo.

```
Variable m : N.
```

```
Hypothesis m_gt_1 : 1 < m.
```

```
Definition mult_mod (x y : N) := (x * y) mod m.
```

```
Definition mod_eq (x y : N) := x mod m = y mod m.
```

```
Instance mod_equiv : Equiv N := mod_eq.
```

```
Instance mod_op : Mult_op N := mult_mod.
```

```
Instance mod_Equiv : Equivalence mod_equiv.
```

```

#[global] Instance mult_mod_proper :
  Proper (mod_equiv ==> mod_equiv ==> mod_equiv) mod_op.
#[local] Open Scope M_scope.

Lemma mult_mod_associative : forall x y z,
  x * (y * z) = x * y * z.
Lemma one_mod_neutral_l : forall x, 1 * x == x.
Lemma one_mod_neutral_r : forall x, x * 1 == x.
#[global] Instance Nmod_Monoid : EMonoid mod_op 1 mod_equiv.

```

End Nmodulo.

Section S256.

```

Let mod256 := mod_op 256.

#[local] Existing Instance mod256 | 1.

Compute (211 * 67)%M.

```

```

= 57
: N

```

Outside the section S256, the term  $(211 * 67)\%M$  is interpreted as a plain multiplication in type  $N$ :

End S256.

```

Compute (211 * 67)%M.

```

```

= 14137
: N

```

## 17.4 Computing powers in any EMonoid

The module `additions.Pow` defines two functions for exponentiation on any `EMonoid` on carrier  $A$ . They are essentially the same as in Sect. 17.2 on page 281. The main difference lies in the arguments of the functions, which now contain an instance  $M$  of class `EMonoid`. Thus, the arguments associated with the multiplication, the neutral element and the equivalence relation associated with  $M$  are left implicit.

### 17.4.1 The naive (linear) algorithm

The new version of the linear exponentiation function is as follows:

```

Fixpoint power `{M: @EMonoid A E_op E_one E_eq}
  (x:A)(n:nat) :=
  match n with 0%nat => E_one
             | S p => x * x ^ p

```

```

end
where "x ^ n" := (power x n) : M_scope.

```

The three following lemmas will be used by the `rewrite` tactic in further correctness proofs. Note that the first two lemmas are strong (*i.e.*, Leibniz) equalities, whilst `power_eq3` is only an equivalence statement, because its proof uses one of the `EMonoid` laws, namely `Eone_right`.

```

Lemma power_eq1 {M: @EMonoid A E_op E_one E_eq}(x:A) :
  x ^ 0 = E_one.
Proof. reflexivity. Qed.

```

```

Lemma power_eq2 {M: @EMonoid A E_op E_one E_eq}(x:A) (n:nat) :
  x ^ (S n) = x * x ^ n.
Proof. reflexivity. Qed.

```

```

Lemma power_eq3 {M: @EMonoid A E_op E_one E_eq}(x:A) :
  x ^ 1 == x.
Proof. cbn; rewrite Eone_right; reflexivity. Qed.

```

### 17.4.1.1 Examples of computation

In the following computations, we first show an exponentiation in  $\mathbb{Z}$ , then in the type of 31-bit machine integers.<sup>2</sup>

From Module `additions.Demo_power`

```

Open Scope M_scope.

```

```

Compute 22%Z ^ 20.

```

```

= 705429498686404044207947776
: Z

```

```

Import Uint63.

```

```

Search (Z -> int).

```

```

of_Z: Z -> int

```

```

Coercion of_Z : Z >-> int.

```

```

Compute 22%int63 ^ 50.

```

```

= 8855202767317237760%uint63
: int

```

## 17.4.2 The binary exponentiation algorithm

Please find below the implementation of binary exponentiation using type classes (to be compared with the version in 17.2.1 on page 282).

<sup>2</sup>`phi` and `phi_inv` are standard library's conversion functions between types `Z` and `int31`, used for making it possible to read and print values of type `int31`.

From Module additions.Pow

```

Fixpoint binary_power_mult `{M: @EMonoid A E_op E_one E_eq}
  (x a:A)(p:positive) : A
:=
  match p with
  | xH => a * x
  | x0 q => binary_power_mult (x * x) a q
  | xI q => binary_power_mult (x * x) (a * x) q
  end.
Fixpoint Pos_bpow `{M: @EMonoid A E_op E_one E_eq}
  (x:A)(p:positive) :=
  match p with
  | xH => x
  | x0 q => Pos_bpow (x * x) q
  | xI q => binary_power_mult (x * x) x q
  end.

```

It is easy to extend Pos\_bpow's domain to the type of all natural numbers:

From Module additions.Pow

```

Definition N_bpow `{M: @EMonoid A E_op E_one E_eq} x (n:N) :=
  match n with
  | 0%N => E_one
  | Npos p => Pos_bpow x p
  end.

```

```

Infix "^b" := N_bpow (at level 30, right associativity) : M_scope.

```

### 17.4.3 Refinement and correctness

We have got two functions for computing powers in any monoid. So, it is interesting to ask oneself whether this duplication is useful, and which would be the respective role of N\_bpow and power.

- The function power, although very inefficient, is a direct translation of the mathematical definition, as shown by lemmas power\_eq1 to power\_eq3. Moreover, its structural recursion over type nat allows simple proofs by induction over the exponent. Thus, we will consider power as a *specification* of any exponentiation algorithm.
- Functions N\_bpow and Pos\_bpow are more efficient, but less readable than power, and we cannot use these functions before having proved their correctness. In fact, the correctness of N\_bpow and Pos\_bpow will mean “being extensionally equivalent to power”. For instance N\_bpow's correctness is expressed by the following statement (in the context of an EMonoid on type A).

From Module additions.Pow

```

Lemma N_bpow_ok :
  forall n x, x ^b n == x ^ N.to_nat n.

```

The relationship between `power` and `N_bpow` can be considered as a kind of *refinement* as in the B-method [Abr96]. Note that the two representations of natural numbers and the function `N.to_nat` form a kind of *data refinement* [Abr10, CDM13].

#### 17.4.4 Proof of correctness of binary exponentiation w.r.t. the function `power`

Section `M_given` of Module `additions.Pow` is devoted to the proof of properties of the functions above. Note that properties of `power` refer to the *specification* of exponentiation, and can be applied for proving correctness of any implementation.

In this section, we consider an arbitrary instance `M` of class `EMonoid`.

**Section `M_given`.**

```
Variables (A:Type) (E_one:A) .
Context (E_op : Mult_op A) (E_eq : Equiv A)
        (M:EMonoid E_op E_one E_eq).
```

##### 17.4.4.1 Properties of exponentiation

We establish a few well-known properties of exponentiation, and define some basic tactics for simplifying proof search.

```
Ltac monoid_rw :=
  rewrite Eone_left ||
  rewrite Eone_right ||
  rewrite Eop_assoc.

Ltac monoid_simpl := repeat monoid_rw.
```

In order to make possible proof by rewriting on expressions which contain the exponentiation operator, we have to prove that, whenever  $x == y$ , the equality  $x^n == y^n$  holds for any exponent  $n$ . For this purpose, we use the `Proper` class of module `Coq.Classes.Morphisms`

```
#[global] Instance power_proper :
  Proper (equiv ==> eq ==> equiv) power.
```

In the following proofs, we note how notations, type classes and generalized rewriting can be used to write algebraic properties in a nice way.

```
Lemma power_of_plus :
  forall x n p, x ^ (n + p) == x ^ n * x ^ p.
Ltac power_simpl := repeat (monoid_rw || rewrite <- power_of_plus).
```

Please note that the following lemmas *do not require* the operation `*` to be commutative.

**Lemma** `power_commute x n p` :  
 $x^n * x^p == x^p * x^n$ .

**Proof.**  
`power_simpl; now rewrite (Nat.add_comm n p).`  
**Qed.**

**Lemma** `power_commute_with_x x n` :  
 $x * x^n == x^n * x$ .

**Proof.**  
`induction n; cbn.`  
`- now monoid_simpl.`  
`- rewrite IHn at 1; now monoid_simpl.`  
**Qed.**

**Lemma** `power_of_power x n p` :  
 $(x^n)^p == x^{(p * n)}$ .

**Proof.**  
`induction p; cbn.`  
`- reflexivity.`  
`- rewrite IHp; now power_simpl.`  
**Qed.**

The following two equalities are auxiliary lemmas for proving correctness of the binary exponentiation functions.

**Lemma** `sqr_eqn` : `forall x, x^2 == x * x`.

**Lemma** `power_of_square x n` :  $(x * x)^n == x^n * x^n$ .

**Proof.**  
`induction n; cbn; monoid_simpl.`  
`- reflexivity.`  
`- rewrite IHn; now factorize.`  
**Qed.**

### 17.4.5 Equivalence of the two exponentiation functions

Since `binary_power_mult` is defined by structural recursion on the exponent `p:positive`, its basic properties are proved by induction along `positive`'s constructors.

*From Module* `additions.Pow`

**Lemma** `binary_power_mult_ok` :  
`forall p a x, binary_power_mult x a p == a * x ^ Pos.to_nat p.`

**Proof.**  
`induction p as [q IHq | q IHq].`  
`(* ... *)`

**Lemma** `Pos_bpow_ok` :  
`forall p x, Pos_bpow x p == x ^ Pos.to_nat p.`

**Lemma** `Pos_bpow_ok_R` :  
`forall p x, p <> 0 ->`  
`Pos_bpow x (Pos.of_nat p) == x ^ p.`

```
Lemma N_bpow_ok :
  forall n x, x ^b n == x ^ N.to_nat n.
```

```
Lemma N_bpow_ok_R :
  forall n x, x ^b (N.of_nat n) == x ^ n.
```

#### 17.4.5.1 Remark

The preceding lemmas can be applied for deriving properties of the binary exponentiation functions:

```
Lemma N_bpow_commute : forall x n p,
  x ^b n * x ^b p ==
  x ^b p * x ^b n.
```

**Proof.**

```
intros x n p; repeat rewrite N_bpow_ok;
  apply power_commute.
```

**Qed.**

#### 17.4.6 Fibonacci, once again

We can use the function `Pos_bpow` for computing Fibonacci numbers (see Section 17.2.3.2 on page 285).

*From Module `additions.Fib2`*

```
Definition fib_pos n :=
  let (a,b) := Pos_bpow (M:= Mul2) (1,0) n in
  (a+b).
```

**Compute** `fib_pos xH`.

```
= 1
: N
```

**Compute** `fib_pos 10%positive`.

```
= 89
: N
```

**Time Compute** `fib_pos 153%positive`.

```
= 68330027629092351019822533679447
: N
Finished transaction in 0.012 secs (0.012u,0.s) (successful)
```

Fibonacci will come back in Sect. 17.9.11 on page 343.

### 17.5 Comparing exponentiation algorithms with respect to efficiency

It looks obvious that the binary exponentiation algorithm is more efficient than the naive one. Can we study *within Coq* the respective efficiency of both functions? Let us take a simple example with the exponent 17, in any `EMonoid`.



In the next section, we define a *data structure* for representing the computations that lead to the evaluation of some power  $x^n$ , where intermediary results are explicitly named for further use in the rest of the computation.

## 17.6 Addition chains

An *addition chain* (in short, a *chain*) [Bra39] is a representation of a sequence of intermediate steps that lead to the evaluation of  $x^n$ , under the assumption that each of these steps is a computation of a power  $x^i$ , with  $i < n$ .

In articles from the combinatorist community, *e.g.*, [Bra39, BB87], addition chains are represented as sequences of positive integers, each member of which is either 1 or the sum of two previous elements. For instance, the three following sequences are addition chains for the exponent 87:

$$c_{87} = (1, 2, 3, 6, 7, 10, 20, 40, 80, 87) \quad (17.12)$$

$$c'_{87} = (1, 2, 3, 4, 7, 8, 16, 23, 32, 64, 87) \quad (17.13)$$

$$c''_{87} = (1, 2, 4, 8, 16, 32, 64, 80, 84, 86, 87) \quad (17.14)$$

It is possible to associate to any addition chain a directed acyclic graph: whenever  $i = j + k$ , there is an arc from  $x^j$  to  $x^i$  and an arc from  $x^k$  to  $x^i$ . Figures 17.1 and 17.2 show the graphical representations of  $c_{87}$  and  $c'_{87}$ . Please note that some chains may be represented by various different dags (directed acyclic graphs). For instance, we can associate four different dags to the chain  $(1, 2, 3, 4, 6, 9, 13)$ .

Figure 17.1: Graphical representation of  $c_{87}$  (9 multiplications)

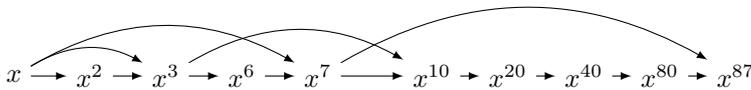
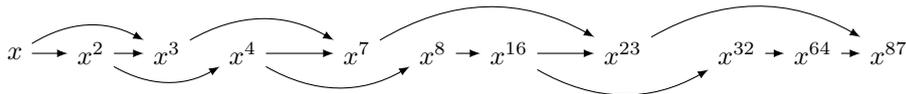


Figure 17.2: Graphical representation of  $c'_{87}$  (10 multiplications)



Let us assume that the efficiency of an exponentiation algorithm is proportional to the number of multiplications it requires. This assumption looks reasonable when the data size is bounded (for instance : machine integers, arithmetic modulo  $m$ , etc.). Let us define the *length* of a chain  $c$  as its number  $|c|$  of exponents (without counting the initial 1). This length is the number of multiplications needed for computing the  $x^i$ 's by applying the following algorithm:

For any item  $i$  of  $c$  (but the first one), there exists  $j$  and  $k$  in  $c$ , where  $i = j + k$ , and  $x^j$  and  $x^k$  are already computed.

Thus, compute  $x^i = x^j \times x^k$ .

In our little example, we have  $|c_{87}| = 9 < 10 = |c'_{87}|$ . In the rest of this chapter, we will try to focus on the following aspects:

- Define a representation of addition chains that allows to compute efficiently  $x^n$  in any monoid, for quite large exponents  $n$ ;
- Certify that our representation of chains is correct, *i.e.*, determines a computation of  $x^n$  for a given  $n$ ;
- Define and certify functions for automatically generating correct and shortest as possible chains.

In a previous work [BCHM95, BCS91, Cas04], addition chains were represented so as to allow efficient computations of powers and certification of a family of automatic chain generators. We present here a new implementation that takes into account some advances in the way we use Coq: generalized rewriting, type classes, parametricity, etc.

### 17.6.1 A type for addition chains

Let us recall that we want to represent some algorithms of the form described in section 17.5, but avoiding to represent intermediate results by **let-in** constructs. We describe below the main design choices we made:

- Continuation Passing Style (CPS) [Rey93] is a way to make explicit the control in the evaluation of an expression, in a purely functional way. For every intermediate computation step, the result is sent to a *continuation* that executes the further continuations. When the continuation is a lambda-abstraction, its bound variable gives a *name* to this result
- Like in Parametric Higher Order Abstract Syntax (PHOAS) [Chl08], the local variables associated to intermediate results are represented by variables of type  $A$ , where  $A$  is the underlying type of the considered monoid.

#### 17.6.1.1 Definition

Let  $A$  be some type; a *computation* on  $A$  is

- either a final step, returning some value of type  $A$
- or the multiplication of two values of type  $A$ , with a *continuation* that takes as argument the result of this multiplication, then starts a new computation.

In the following inductive type definition, the intended meaning of the construct `(Mult  $x$   $y$   $k$ )` is “*multiply  $x$  with  $y$ , then send the result of this multiplication to the continuation  $k$* ”.

From Module additions.Addition\_Chains

```
Inductive computation {A:Type} : Type :=
| Return (a : A)
| Mult (x y : A) (k : A -> computation).
```

### 17.6.1.2 Monadic notation

The following *monadic* notation makes terms of type `computation` look like expressions of a small programming language dedicated to sequences of multiplications. Please look at *CPDT* [Ch11] for more details on monadic notations in Coq.

```

Notation "z '<---' x 'times' y ';' e2 " :=
  (Mult x y (fun z => e2))
  (right associativity, at level 60).

```

The `computation` type family is able to express sharing of intermediate computations. For instance, the computation of  $2^7$  depicted in Figure 17.3 is described by the following term:

```

Example compl28 : computation :=
  x <--- 2 times 2;
  y <--- x times 2;
  z <--- y times y ;
  t <--- 2 times z ;
  Return t.

```

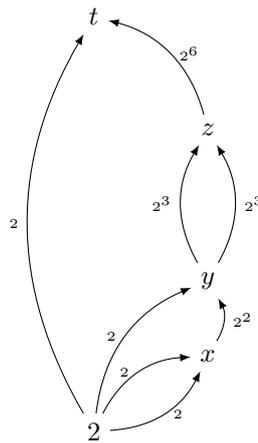


Figure 17.3: The dag associated to a computation of  $2^7$

### 17.6.1.3 Definition

Thanks to the `computation` type family, we can associate a type to the kind of computation schemes described in Figures 17.1 and 17.2.

We define *addition chains* (in short *chains*) as functions that map any type `A` and any value `a` of type `A` into a computation on `A`:

```

Definition chain := forall A:Type, A -> @computation A.

```

Thus, terms of type `chain` describe polymorphic exponentiation algorithms.

```

Example C87 : chain :=
  fun A (x : A) =>
    x2 <--- x times x ;
    x3 <--- x2 times x ;
    x6 <--- x3 times x3 ;
    x7 <--- x6 times x ;
    x10 <--- x7 times x3 ;
    x20 <--- x10 times x10 ;
    x40 <--- x20 times x20 ;
    x80 <--- x40 times x40 ;
    x87 <--- x80 times x7 ;
  Return x87.

```

Figure 17.4: A chain for raising  $x$  to its 87-th power

For instance, Fig 17.4 shows a definition of the chain of Figure 17.1, for the exponent 87. Note that, like in PHOAS, bound variables associated with the intermediary results are Coq variables of type  $A$ .

The structure of the definition of types `computation` and `chain` suggest that basic definitions over `chain` will have the following structure:

- A recursive function on type `computation A` (for a given type  $A$ )
- A main function on type `chain` that calls the previous one on any  $A:\text{Type}$ .

For instance, the following function computes the length of any chain, *i.e.*, the number of multiplications of the associated computation. Note that the function `chain_length` calls the auxiliary function `computation_length`, with the variable  $A$  instantiated to the singleton type `unit`.

Any other type in Coq would have fitted our needs, but `unit` and its unique inhabitant `tt` was the simplest solution.

```

Fixpoint computation_length {A} (a:A) (m : @computation A) : nat :=
  match m with
  | Mult _ _ k => S (computation_length a (k a))
  | _ => 0%nat
  end.

```

```

Definition chain_length (c:chain) := computation_length tt (c _ tt).

```

```

Compute chain_length C87.

```

```

= 9
: nat

```

## 17.6.2 Chains as a (small) programming language

The `chain` type can be considered as a tiny programming language dedicated to compute powers in any  $\text{EMonoid}$ . Thus, we have to define a semantics for this language. This semantics is defined in two parts:

- A structurally recursive function, — parameterized with an `EMonoid M` on a given type `A` —, that computes the value associated with any computation on `M`
- A polymorphic function that takes as arguments a chain `c`, a type `A`, an `EMonoid` on `A`, and a value `x:A`, then executes the computation `(c A x)`.

```

Fixpoint computation_execute {A:Type} (op: Mult_op A)
  (c : computation) :=
  match c with
  | Return x => x
  | Mult x y k => computation_execute op (k (x * y))
  end.

```

```

Definition computation_eval `{M:@EMonoid A E_op E_one E_eq}
  (c : computation) : A :=
  computation_execute E_op c.

```

```

Definition chain_execute (c:chain) {A} op (a:A) :=
  computation_execute op (c A a).

```

```

Definition chain_apply
  (c:chain) `{M:@EMonoid A E_op E_one E_eq} a : A :=
  computation_eval (c A a).

```

### Examples:

The following interactions show how to apply the chain `C87` for exponentiation within two different monoids:

**Time Compute** chain\_apply C87 3%Z.

```

= 323257909929174534292273980721360271853387%Z
: Z
Finished transaction in 0.006 secs (0.006u,0.s) (successful)

```

**Time Compute** chain\_apply (M:=M2N) C87 (Build\_M2 1 1 1 0)%N.

```

= {|
  c00 := 1100087778366101931%N;
  c01 := 679891637638612258%N;
  c10 := 679891637638612258%N;
  c11 := 420196140727489673%N
|}
: M2 N
Finished transaction in 0.005 secs (0.005u,0.s) (successful)

```

**Project 17.1** Study how to compile efficiently such data structures.

**Project 17.2** Define a function which returns the sequence of operations defined by a chain. For instance, the chain C87 of Figure 17.4 can be represented as a list containing terms of the form  $(i, \text{Add } j \ k)$  whenever the associated computation contains the operation  $x^i = x^j \times x^k$ .

**Compute** chain\_trace C87.

```
= [(1, Init); (2, Add 1 1); (3, Add 2 1);
   (6, Add 3 3); (7, Add 6 1); (10, Add 7 3);
   (20, Add 10 10); (40, Add 20 20);
   (80, Add 40 40); (87, Add 80 7)]
: list (positive * info)
```

**Note** A first solution (in `additions.Trace_exercise`) consists in the definition of a (non-associative) multiplication over a type of trace, and apply the function `chain_execute` as if it were computing a power of  $(1, \text{Init})$ .

### 17.6.2.1 Chain correctness and optimality

A chain is said to be *correct* with respect to a positive integer  $p$  if its execution in any monoid computes  $p$ -th powers.

```
Definition chain_correct_nat (n:nat) (c: chain) := n <> 0 /\
forall `(M:@EMonoid A E_op E_one E_eq) (x:A),
  chain_apply c x == x ^ n.
```

```
Definition chain_correct (p:positive) (c: chain) :=
  chain_correct_nat (Pos.to_nat p) c.
```

**Definition 17.1** A chain  $c$  is optimal for a given exponent  $p$  if its length is less than or equal to the length of any chain correct for  $p$ .

```
Definition optimal (p:positive) (c : chain) :=
  forall c', chain_correct p c' ->
    (chain_length c <= chain_length c')%nat.
```

## 17.7 Proving a chain's correctness

In this section, we present various ways of proving that a given chain is correct w.r.t. a given exponent. First, we just try to apply the definition in Section 17.6.2.1, but this method is very inefficient, even for small exponents. In a second step, we use more sophisticated techniques such as reflection and parametricity. Automatic generation of correct chains will be treated in Sect. 17.8 on page 318.

### 17.7.1 Proof by rewriting

Let us show how to prove the correctness of some chains, using the `EMonoid` laws shown in Sect. 17.3.5 on page 293.

```

Ltac slow_chain_correct_tac :=
  match goal with
  [ |- chain_correct ?p ?c ] =>
    let A := fresh "A" in
    let op := fresh "op" in
    let one := fresh "one" in
    let eqv := fresh "eqv" in
    let M := fresh "M" in
    let x := fresh "x"
    in split;[discriminate |
              unfold c, chain_apply, computation_eval; simpl;
              intros A op one eq M x; monoid_simpl M; reflexivity]
  end.

```

**Example C7\_ok** : chain\_correct 7 C7.

**Proof.**

slow\_chain\_correct\_tac.

**Qed.**

Unfortunately, this approach is terribly inefficient, even for quite small exponents:

```

Example C87_ok_slow : chain_correct 87 C87.
Proof.
Time slow_chain_correct_tac.

```

```

Finished transaction in 49.927 secs (49.445u,0.079s) (successful)

```

```

Qed.

```

In addition to this big computation time, this approach generates a huge proof term. Just try to execute the command “Print C87\_ok” to get a measure of its size. In order to understand this poor performance, let us consider an intermediate subgoal of the previous proof generated after a sequence of unfoldings and simplifications. This goal is presented in Figure 17.5 on the facing page.

This inefficiency certainly comes from the cost of setoid rewriting. At every application of an `EMonoid` law, the system must verify that the context of this rewriting is compatible with the equivalence relation associated with the current `EMonoid`. The rest of this chapter is devoted to the presentation of more efficient methods for proving chain correctness.

## 17.7.2 Correctness proofs by reflection

Instead of letting the tactic `rewrite` look for contexts in which setoid rewriting is possible, we propose to use (deterministic) computations for obtaining a “canonical” form for terms generated from a variable `x` by constructors associated with monoid multiplication and neutral element.

The reader will find general explanations about proofs by reflection in Coq, for instance in Chapter 16 of Coq’Art[BC04a] and the numerous examples (including the `ring` tactic) in Coq’s reference manual.



```

(** Binary trees of multiplications over A *)

Inductive Monoid_Exp (A:Type) : Type :=
  Mul_node (t t' : Monoid_Exp A) | One_node | A_node (a:A).

Arguments Mul_node {A} _ _ .
Arguments One_node {A} .
Arguments A_node {A} _ .

```

Thus, the main steps of a correctness proof of a given chain, *e.g.*, C87 will be the following ones:

1. generate a subgoal as in Fig. 17.5 on the preceding page,
2. express each term of the equivalence as the image of a term of type `Monoid_Exp A`,
3. normalize both terms and verify that their normal forms are equal.

The rest of this section is devoted to the definition of the normalization function on `Monoid_Exp A`, and the proofs of lemmas that link equivalence on type `A` and equality of normal forms of terms of type `Monoid_Exp A`.

### 17.7.2.2 Linearization function

The following functions help to transform any term of type `Monoid_Exp A` into a flat “normal form”.

```

(** Linearization functions *)

Fixpoint flatten_aux {A:Type}(t fin : Monoid_Exp A)
  : Monoid_Exp A :=
  match t with
  | Mul_node t t' => flatten_aux t (flatten_aux t' fin)
  | One_node => fin
  | x => Mul_node x fin
  end.

Fixpoint flatten {A:Type} (t: Monoid_Exp A)
  : Monoid_Exp A :=
  match t with
  | Mul_node t t' => flatten_aux t (flatten t')
  | One_node => One_node
  | X => Mul_node X One_node
  end.

Compute
  fun x y z t : nat =>
    flatten (Mul_node (Mul_node (A_node x) (A_node y))
      (Mul_node (A_node z) (A_node t))).

```

```

= fun x y z t : nat =>
  Mul_node (A_node x)
    (Mul_node (A_node y)
      (Mul_node (A_node z)
        (Mul_node (A_node t) One_node)))
: nat -> nat -> nat -> nat -> Monoid_Exp nat

```

### 17.7.2.3 Interpretation function

The function `eval` maps any term of type `Monoid_Exp A` into a term of type `A`.

```

Function eval {A:Type} {op one eqv}
  (M: @EMonoid A op one eqv)
  (t: Monoid_Exp A) : A :=
  match t with
  | Mul_node t1 t2 => (eval M t1 * eval M t2)%M
  | One_node => one
  | A_node a => a
end.

```

The following two lemmas relate the linearization function `flatten` with the interpretation function `eval`.

```

Lemma flatten_valid `(M: @EMonoid A op one eqv):
  forall t , eval M t == eval M (flatten t).

```

```

Lemma flatten_valid_2 `(M: @EMonoid A op one eqv):
  forall t t' , eval M (flatten t) == eval M (flatten t') ->
    eval M t == eval M t'.

```

### 17.7.2.4 Transforming a multiplication into a tree

Let us now build a tool for building terms of type `(Monoid_Exp A)` out of terms of type `A` containing multiplications of the form `(_ * _)%M` and the variable `one`. In fact, what we want to define is an inverse of the function `flatten`.

Since `mult_op` is not a constructor (see Sect. 17.3.1), the transformation of a product of type `A` into a term of type `Monoid_Exp A` is done with the help of a tactic:

```

Ltac model A op one v :=
  match v with
  | (?x * ?y)%M => let r1 := model A op one x
                  with r2 :=(model A op one y)
                  in constr:(@Mul_node A r1 r2)
  | one => constr:(@One_node A)
  | ?x => constr:(@A_node A x)
end.

```

For instance, the term `(x * x * x * (x * x * x) * x)` is transformed by `model` in the following term of type `Monoid_Exp A`

```
(eval M
  (Mul_node
    (Mul_node
      (Mul_node (Mul_node (A_node x) (A_node x)) (A_node x))
      (Mul_node (Mul_node (A_node x) (A_node x)) (A_node x)))
    (A_node x)))
```

### 17.7.3 Reflection tactic

The tactic `monoid_eq_A` converts a goal of the form  $(E\_eq\ X\ Y)$ , where  $X$  and  $Y$  are terms of type  $A$ , into  $(E\_eq\ (eval\ M\ (model\ X))\ (eval\ M\ (model\ Y)))$ . This last goal is intended to be solved thanks to the lemma `flatten_valid_2`.

```
Ltac monoid_eq_A A op one E_eq M :=
match goal with
| [ |- E_eq ?X ?Y ] =>
  let tX := model A op one X with
    tY := model A op one Y in
    (change (E_eq (eval M tX) (eval M tY)))
end.
```

#### 17.7.3.1 Main reflection tactic

The tactic `reflection_correct_tac` tries to prove a chain's correctness by a comparison of two terms of type `Monoid_Exp A`: one being obtained from the chain's definition, the other one by expansion of the naive exponentiation definition.

```
Ltac reflection_correct_tac :=
match goal with
| [ |- chain_correct ?n ?c ] =>
  split; [try discriminate |
    let A := fresh "A"
    in let op := fresh "op"
    in let one := fresh "one"
    in let E_eq := fresh "eq"
    in let M := fresh "M"
    in let x := fresh "x"
    in (try unfold c); unfold chain_apply;
      simpl; red; intros A op one E_eq M x;
      unfold computation_eval; simpl;
      monoid_eq_A A op one E_eq M;
      apply flatten_valid_2; try reflexivity
  ]
end.
```

#### 17.7.3.2 Example

The following dialogue clearly shows the efficiency gain over naive setoid rewriting.

**Example** C87\_ok : chain\_correct 87 C87.

**Proof.**

**Time** reflection\_correct\_tac.

```
Finished transaction in 0.033 secs (0.032u,0.s) (successful)
```

**Qed.**

This tactic is not adapted to much bigger exponents. In Module `Euclidean_Chains`, for instance, we tried to apply this tactic for proving the correctness of a chain associated with the exponent 45319. We had to interrupt the prover, which was trying to build a linear tree of  $2 \times 45319 + 1$  nodes! Indeed, using `reflection_correct_tac` is like doing a symbolic evaluation of an inefficient (linear) exponentiation algorithm.

In the next section, we present a solution that avoids doing such a lot of computations.

## 17.7.4 Chain correctness for —practically — free!

### 17.7.4.1 About parametricity

Let us now present another tactic for proving chain correctness, in the tradition of works on *parametricity* and its use for proving properties on programs. Strachey [Str00] explores the nature of *parametric polymorphism*: “*Polymorphic functions behave uniformly for all types*” then Reynolds [Rey83] formalizes this notion through binary relations. Wadler [Wad89], then Cohen *et al.* [CDM13] use this relation for deriving theorems about functions that operate on parametric polymorphic types.

Let us look again at the definitions of type family `computation` and the type `chain`:

```
Inductive computation {A:Type} : Type :=
| Return (a : A)
| Mult (x y : A) (k : A -> computation).
Definition chain := forall A:Type, A -> @computation A.
```

Let  $c$  be a closed term of type `chain`;  $c$  is of the form `fun (A:Type) (a:A) => ta`, where  $t_a$  is a term of type `@computation A`. Obviously, in every subterm of  $t_a$  of type `A`, the two first arguments of constructor `Mult` or the argument of `Return` are either `a` or a variable introduced as the formal argument of a continuation `k`. In effect, there is no other way to build terms of type `A` in the considered context.

Marc Lasson’s `paramcoq` plug-in (available as `opam` package `coq-paramcoq`) generates a family of binary relations definitions from `computation`’s definition.

Parametricity computation.

**Print** computation\_R.

```

Inductive
computation_R (A1 A2 : Type) (A_R : A1 -> A2 -> Type)
: computation -> computation -> Type :=
  computation_R_Return_R : forall (a1 : A1)
    (a2 : A2),
    A_R a1 a2 ->
    computation_R A1 A2 A_R
    (Return a1) (Return a2)
| computation_R_Mult_R : forall (x1 : A1) (x2 : A2),
  A_R x1 x2 ->
  forall (y1 : A1) (y2 : A2),
  A_R y1 y2 ->
  forall
    (k1 : A1 -> computation)
    (k2 : A2 -> computation),
  (forall (H : A1) (H0 : A2),
  A_R H H0 ->
  computation_R A1 A2 A_R
  (k1 H) (k2 H0)) ->
  computation_R A1 A2 A_R
  (z <--- x1 times y1; k1 z)
  (z <--- x2 times y2; k2 z).

Arguments computation_R (A1 A2)%type_scope
  A_R%function_scope _ _
Arguments computation_R_Return_R (A1 A2)%type_scope
  A_R%function_scope a1 a2 a_R
Arguments computation_R_Mult_R (A1 A2)%type_scope
  A_R%function_scope x1 x2 x_R y1 y2 y_R
  (k1 k2 k_R)%function_scope

```

Let  $A$  and  $B$  be two types, and  $R : A \rightarrow B \rightarrow \mathbf{Type}$  a relation. Two computations  $cA : @computation\ A$  and  $cB : @computation\ B$  are related *w.r.t.* `computation_R` if every pair of arguments of `Mult` and `Return` at the same position are related *w.r.t.*  $R$ .

#### 17.7.4.2 Definition

A chain  $c$  is *parametric* if it has the same behavior for any pair of types  $A$  and  $B$ , any relation  $R$  between  $A$  and  $B$  and any  $R$ -related pair of arguments  $a$  and  $b$ :

```

Definition parametric (c:chain) :=
  forall A B (R: A -> B -> Type) (a:A) (b:B),
  R a b -> computation_R A B R (c A a) (c B b).

```

#### 17.7.4.3 How to use these definitions?

Let us use parametricity for proving easily a given chain's correctness. In other words, let  $c$  be a chain and  $p$ :positive be a given exponent. Consider some

instance of `EMonoid` over a type  $A$ . We want to prove that the application of the chain  $c$  to any value  $a$  of type  $A$  returns the value  $a^p$ .

We first use Coq's computation facilities for "guessing" the exponent associated with any given chain. It suffices to instantiate "monoid multiplication" with addition on positive integers.

```
Definition the_exponent_nat (c:chain) : nat :=
  chain_apply c (M:=Natplus) 1%nat.
```

```
Definition the_exponent (c:chain) : positive :=
  chain_execute c Pos.add 1%positive.
```

```
Compute the_exponent C87.
```

```
= 87%positive
: positive
```

We show how to *prove* that a given chain  $c$ , applied to any  $a$ , really computes  $a^p$ , where  $p = \text{the\_exponent } c$ . Parametricity allows us to compare executions on any monoid  $M$  with executions on `NatPlus`. Let us consider the mathematical relation  $\{(x, n) \in M \times \mathbb{N} \mid 0 < n \wedge x = a^n\}$ .

```
Definition power_R (a:A) :=
  fun (x:A)(n:nat) => n <> 0 /\ x == a ^ n.
```

First, we prove the following lemma, that relates `computation_R` with the result of the executions of the corresponding computations:

```
Lemma power_R_is_a_refinement (a:A) :
  forall (gamma : @computation A)
    (gamma_nat : @computation nat),
  computation_R _ _ (power_R a) gamma gamma_nat ->
  power_R a (computation_eval gamma)
    (computation_eval (M:= Natplus) gamma_nat).
```

Thus, if  $c:\text{chain}$  is parametric, this refinement lemma allows us to prove a correctness result:

```
Lemma param_correctness_nat (c: chain) :
  parametric c ->
  chain_correct_nat (the_exponent_nat c) c.
```

A similar result can be proven with the exponent in `positive`. First we instantiate the parameter `R` of `computation_R`, with the relation that links the representations of natural numbers on respective types `nat` and `positive`. Then we use our lemmas for rewriting under the assumption that the considered chain is parametric. Please note how our approach is related with *data refinement* (see also [CDM13]). The reader may also consult a survey by D. Brown on the most important contributions to the notion of parametricity [Bro10].

```
Lemma exponent_pos2nat : forall c: chain,
  parametric c ->
  the_exponent_nat c = Pos.to_nat (the_exponent c).
```

```

Lemma exponent_pos_of_nat :
  forall c: chain, parametric c ->
    the_exponent c = Pos.of_nat (the_exponent_nat c).

```

```

Lemma param_correctness :
  forall (p:positive) (c:chain),
    p = the_exponent c -> parametric c ->
    chain_correct p c.

```

Lemma `param_correctness` suggests us a method for verifying that a given chain  $c$  is correct *w.r.t.* some positive exponent  $p$ :

1. Verify that  $c$  is parametric.
2. Verify that  $p$  is equal to `(the_exponent c)`.

#### 17.7.4.4 How to prove a chain's parametricity

Despite the apparent complexity of `computation_R`'s definition, it is very simple to prove that a given chain is parametric. The following tactics proceed as follows:

1. Given a chain  $c$ , consider two types  $A$  and  $B$ , and any relation  $R:A \rightarrow B \rightarrow \text{Prop}$ ,
2. Push into the context declarations of  $a:A$ ,  $b:B$  and an hypothesis assuming  $R\ a\ b$ .
3. Then the tactic crosses in parallel the terms  $(c\ A\ a)$  and  $(c\ B\ b)$  (of the same structure),
  - On a pair of terms of the form `Mult xA yA (fun zA => tA)` and `Mult xB yB (fun zB => tB)`, the tactic checks whether  $R\ xA\ xB$  and  $R\ yA\ yB$  are already assumed in the context, then pushes into the context the declaration of  $zA$  and  $zB$  and the hypothesis  $H_z: R\ zA\ zB$ , then crosses the terms  $tA$  and  $tB$
  - On a pair of terms of the form `(Return xA)` and `(Return xB)`, the tactic just checks whether  $(R\ xA\ xB)$  is assumed.

The tactic itself is simpler than its explanation.

```

Ltac parametric_tac :=
  match goal with [ |- parametric ?c ] =>
    red ; intros;
    repeat (right;[assumption | assumption | ]); left; assumption
  end.

```

**Example P87** : parametric C87.

**Proof.**

**Time** parametric\_tac.

```

Finished transaction in 0.005 secs (0.004u,0.s) (successful)

```

**Qed.**

#### 17.7.4.5 Proving a chain's correctness

Finally, for proving that a given chain  $c$  is correct with respect to an exponent  $p$ , it suffices to check that  $c$  is parametric, and to apply the lemma `param_correctness`. The reader will note how this computation-less method is much more efficient than our reflection tactic.

```
Ltac param_chain_correct :=
  match goal with
  [| - chain_correct ?p ?c ] =>
    let H := fresh "H" in
    assert (p = the_exponent c) by reflexivity;
    apply param_correctness;[trivial | parametric_tac]
  end.
```

**Lemma** `C87_ok'` : `chain_correct 87 C87`.

**Time** `param_chain_correct`.

```
Finished transaction in 0.004 secs (0.004u,0.s) (successful)
```

**Qed.**

#### 17.7.4.6 Remark

For the reasons exposed in Section 17.7.4.1 on page 313, it seems obvious that any well-written chain is parametric. Unfortunately, we cannot prove this property in Coq, for instance by induction on  $c$ , since `chain` is a product type and not an inductive type.

**Definition** `any_chain_parametric` : `Type` :=  
`forall c:chain, parametric c.`

**Goal** `any_chain_parametric`.

**Proof.**

```
intros c A B R a b ; induction (c A a) ; destruct (c B b).
```

```
c: chain
A, B: Type
R: A -> B -> Type
a: A
b: B
a0: A
a1: B
-----
R a b -> computation_R A B R (Return a0) (Return a1)
```

**Abort.**

Given this situation, we could admit (as an axiom) that any chain is parametric. Nevertheless, if a chain is under the form of a closed term, using `parametric_tac` is so efficient than we prefer to avoid a shameful introduction of an axiom in our development.

## 17.8 Certified chain generators

In this section, we are interested in the *correct by construction* paradigm. We just want to give a positive exponent to Coq and get a (hopefully) correct and efficient chain for this exponent.

We first define the notion of *chain generator*, then present a certified generator that simulates the binary exponentiation algorithm. Last, we present a better chain generator based on integer division.

### 17.8.1 Definitions

We call *chain generator* any function that takes as argument any positive integer and returns a chain. A generator  $g$  is *correct* if it returns a correct chain for any exponent.

**Definition** `chain_generator` := positive -> chain.

**Definition** `correct_generator` (gen : positive -> chain) :=  
forall p, chain\_correct p (gen p).

Correct generators can be used for computing powers on the fly, thanks to the following functions:

**Definition** `cpower_pos` (g : chain\_generator) p  
`{M:@EMonoid A E\_op E\_one E\_eq} a :=  
chain\_apply (g p) (M:=M) a.

**Definition** `cpower` (g : chain\_generator) n  
`{M:@EMonoid A E\_op E\_one E\_eq} a :=  
match n with 0%N => E\_one  
| Npos p => cpower\_pos g p a  
end.

Note also that the use of chain generators is independent from the techniques presented in Sect. 17.7: Designing an efficient and correct chain generator may be a long and hard task. On the other hand, once a generator is certified, we are assured of the correctness of all its outputs. Finally, we say that a generator  $g$  is *optimal* if it returns chains whose length are less than or equal to any chain returned by any correct generator:

**Definition** `optimal_generator` (g : positive -> chain ) :=  
forall p:positive, optimal p (g p).

### 17.8.2 The binary chain generator

Let us reinterpret the binary exponentiation algorithms in the framework of addition chains. Instead of directly computing  $x^n$  for some base  $x$  and exponent  $n$ , we build chains that describe the computations associated with the binary exponentiation method. Not surprisingly, this chain generation will be described in terms of recursive functions, once the underlying monoid is fixed.

As for the “classical” binary exponentiation algorithm, we define an auxiliary computation generator for the product of an accumulator  $a$  with an arbitrary power of some value  $x$ .

```

Fixpoint exp_scheme {A} p : A -> A -> @computation A :=
  match p with
  | xH => (fun a x => y <--- a times x ; Return y)
  | x0 q => (fun a x => x2 <--- x times x ; exp_scheme q a x2)
  | xI q => (fun a x => ax <--- a times x ;
            x2 <--- x times x ;
            exp_scheme q ax x2)
  end.

```

```

Fixpoint bin_pow_scheme {A} (p:positive) : A -> @computation A:=
  match p with | xH => fun x => Return x
              | xI q => fun x => x2 <--- x times x ; exp_scheme q x x2
              | x0 q => fun x => x2 <--- x times x ; bin_pow_scheme q x2
  end.

```

```

Definition binary_chain (p:positive) : chain :=
  fun A => bin_pow_scheme p.

```

**Compute** binary\_chain 87.

```

= fun (A : Type) (x : A) =>
  x0 <--- x times x;
  x1 <--- x times x0;
  x2 <--- x0 times x0;
  x3 <--- x1 times x2;
  x4 <--- x2 times x2;
  x5 <--- x4 times x4;
  x6 <--- x3 times x5;
  x7 <--- x5 times x5;
  x8 <--- x7 times x7;
  x9 <--- x6 times x8; Return x9
: chain

```

### 17.8.2.1 Proof of binary\_chain's correctness

Let us now prove that `binary_chain` always returns correct chains. First, due to the structure of this generator's definition, we study the properties of the auxiliary functions that operate *on a given monoid  $M$* .

**Section** `binary_power_proof`.

```

Variables (A: Type)
  (E_op : Mult_op A)
  (E_one : A)
  (E_eq: Equiv A).

```

**Context** (M : EMonoid E\_op E\_one E\_eq).

**Existing Instance** `Eop_proper`.

**Lemma** `exp_correct` : forall p a x,

```
computation_eval (axp_scheme p a x) ==
  a * x ^ (Pos.to_nat p).
```

**Proof.**

```
induction p.
(* ... *)
```

**Lemma binary\_correct :**

```
forall p x,
  computation_eval (bin_pow_scheme p (A:=A) x) ==
  x ^ (Pos.to_nat p).
```

**Proof.**

```
intros p ; induction p.
(* ... *)
```

**End binary\_power\_proof.**

**Lemma binary\_generator\_correct :** correct\_generator binary\_chain.

**Proof.**

```
red;unfold chain_correct; intros; unfold binary_chain, chain_apply;
split;[auto with chains] intros; apply binary_correct].
```

**Qed.**

### 17.8.2.2 The binary method is not optimal

It is easy to prove by contradiction that the binary method is not the most efficient for computing powers.

**Section non\_optimality\_proof.**

**Hypothesis binary\_opt :** optimal\_generator binary\_chain.

**Compute** chain\_length (binary\_chain 87).

```
= 10
: nat
```

**Compute** chain\_length C87.

```
= 9
: nat
```

**Lemma binary\_generator\_not\_optimal :** False.

**Proof.**

```
generalize (binary_opt _ _ C87_ok); compute; lia.
```

**Qed.**

**End non\_optimality\_proof.**

**Exercise 17.3** Prove that for any positive integer  $p$ , the length of any optimal chain for  $p$  is less than twice the number of digits of the binary representation of  $p$ .

## 17.9 Euclidean Chains

In this section, we present an efficient chain generator. The chains built by this generator are never longer than the chains built by the binary generator. Moreover, for an infinite number of exponents, the chains it builds are strictly shorter than the chain returned by `binary_chain`. Euclidean chains are based on the following idea:

For generating a chain that computes  $x^n$ , one may choose some natural number  $0 < p < n$ , and build a chain that computes first  $x^p$  **then** uses this value for computing  $x^n$ .

For instance, a computation of  $x^{42}$  can be decomposed into a computation of  $y = x^3$ , then a computation of  $y^{14}$ . The efficiency of the chain built with this methods depends heavily on the choice of  $p$ . See [BCHM95] for details.

Considering chain generators and their correctness, we may consider the dual of decomposition of exponents: we would like to write *composable* correct chain generators. For instance, we want to build some object that, “composed” with any correct chain for  $n$ , returns a correct chain for  $3n$ .

**17.9.0.0.1 Note:** All the Coq material described in this section is available on Module `additions/Euclidean_Chains.v`

### 17.9.1 Chains and continuations : f-chains

Please consider the following small example:

```
Example C3 : chain :=
  fun (A:Type) (x:A) =>
    x2 <--- x times x;
    x3 <--- x2 times x;
    Return x3.
```

The execution of this chain on some value  $x : A$  stops after computing  $x^3$ , because of the `Return` “statement”. However, we would like to compose the instructions of `C3` with a chain for another exponent  $n$ , in order to generate a chain for the exponent  $3 \times n$ .

The solution we present is based on functional programming and the concept of continuation.

#### 17.9.1.1 Type definition of f-chains

Let us consider *incomplete* or *open* chains. Such an object waits for another chain to resume a computation.

Figure 17.6 represents an f-chain associated with the exponent 3, as a dag with an input and one output the edges of which are depicted as thick arrows.

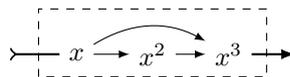


Figure 17.6: Graphical representation of `F3`

In other words, this kind of objects can be considered as *functions* from chains to chains. So, we called their type `Fchain`.

First, we define a type of *continuations*, *i.e.*, functions that wait for some value  $x$ , then build a computation for raising  $x$  to some given exponent. An `f-chain` is just a polymorphic function that combines a continuation and an element into a computation.

**Definition Fkont** `(A:Type) := A -> @computation A.`

**Definition Fchain** `:= forall A, Fkont A -> A -> @computation A.`

### 17.9.1.2 Examples

Let us define a chain for computing the cube of some  $x$ , then sending the result to a continuation  $k$ .

**Definition F3** `: Fchain :=`  
`fun A k (x:A) =>`  
`y <--- x times x ;`  
`z <--- y times x ;`  
`k z.`

Any `f-chain` can be converted into a chain by the help of the following function:

**Definition F2C** `(f : Fchain) : chain :=`  
`fun (A:Type) => f A Return .`

**Compute** `the_exponent (F2C F3).`

```
= 3
: positive
```

In the rest of this chapter, we will use two other `f-chains`, respectively associated with the exponents 1 and 2. Chains `F1`, `F2` and `F3` will form a basis to generate chains for many exponents by *composition of correct functions*.

**Definition F1** `: Fchain :=`  
`fun A k (x:A) => k x.`

**Definition F2** `: Fchain :=`  
`fun A k (x:A) =>`  
`y <--- x times x ;`  
`k y.`

### 17.9.1.3 F-chain application and composition

The following definition allows us to consider any value  $f$  of type `Fchain` as a function of type `chain → chain`.

**Definition Fapply** `(f : Fchain) (c: chain) : chain :=`  
`fun (A:Type) (x: A) => f A (c A) x.`

In a similar way, *composition* of f-chains is easily defined (see Figure 17.7).

**Definition** `Fcompose (f1 f2: Fchain) : Fchain :=`  
`fun A k x => f1 A (fun y => f2 A k y) x.`

**Lemma** `F1_neutral_l : forall f, Fcompose F1 f = f.`  
**Proof.** `reflexivity. Qed.`

**Lemma** `F1_neutral_r : forall f, Fcompose f F1 = f.`  
**Proof.** `reflexivity. Qed.`

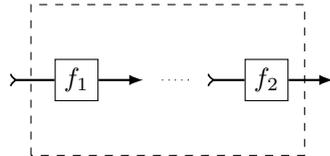


Figure 17.7: Composition of f-chains  $f_1$  and  $f_2$  (`Fcompose`)

### 17.9.1.4 Examples

The following examples show that the apparent complexity of the previous definition is counterbalanced with the simplicity of using `Fapply` and `Fcompose`.

**Example** `F9 := Fcompose F3 F3.`

**Compute** `F9.`

```
= fun (A : Type) (x : Fkont A) (x0 : A) =>
  x1 <--- x0 times x0;
  x2 <--- x1 times x0;
  x3 <--- x2 times x2; x4 <--- x3 times x2; x x4
: Fchain
```

**Remark** `F9_correct : chain_correct 9 (F2C F9).`  
`param_chain_correct.`  
**Qed.**

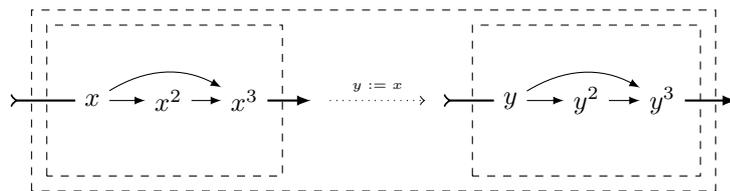


Figure 17.8: Composition of F-chains: `F9`

Using structural recursion and the operator `Fcompose`, we build a chain for any exponent of the form  $2^n$ :

```

Fixpoint Fexp2_of_nat (n:nat) : Fchain :=
  match n with 0 => F1
            | S p => Fcompose F2 (Fexp2_of_nat p)
  end.

```

```

Definition Fexp2 (p:positive) : Fchain :=
  Fexp2_of_nat (Pos.to_nat p).

```

**Compute** Fexp2 4.

```

= fun (A : Type) (x : Fkont A) (x0 : A) =>
  x1 <--- x0 times x0;
  x2 <--- x1 times x1;
  x3 <--- x2 times x2; x4 <--- x3 times x3; x x4
: Fchain

```

**Compute** the\_exponent (F2C (Fexp2 4)).

```

= 16
: positive

```

## 17.9.2 F-chain correctness

Let  $f$  be some term of type  $Fchain$ , and  $n:nat$ . We would like to say that  $f$  is correct *w.r.t.*  $n:nat$  if for any continuation  $k$  and  $a$ , the application of  $f$  to  $k$  and  $a$  computes  $k(a^n)$ .

**Module** Bad.

```

Definition Fchain_correct (n:nat) (fc : Fchain) :=
  forall A `(M : @EMonoid A op E_one E_equiv) k (a:A),
    computation_execute op (fc A k a) ==
    computation_execute op (k (a ^ n)).

```

Let us now try to prove that F3 is correct *w.r.t.* 3.

**Theorem** F3\_correct : Fchain\_correct 3 F3.

**Proof.**

```

  intros A op E_one E_equiv M k a ; cbn.
  monoid_simpl M.

```

```

A: Type
op: Mult_op A
E_one: A
E_equiv: Equiv A
M: EMonoid op E_one E_equiv
k: Fkont A
a: A
H: Proper (equiv ==> equiv ==> equiv) op
-----
computation_execute op (k (a * a * a)) ==
computation_execute op (k (a * (a * (a * E_one))))

```

**Abort.**

**End Bad.**

This failure is due to a lack of an assumption that the continuation  $k$  is *proper* with respect to the equivalence  $\text{equiv}$ . Thus, Coq is unable to infer from the equivalence  $(a * a * a) == (a * (a * (a * E\_one)))$  that  $(k (a * a * a))$  and  $(k (a * (a * (a * E\_one))))$  are equivalent computations.

### 17.9.2.1 Definition:

A continuation  $k : \text{Fkont } A$  is *proper* if, whenever  $x == y$  holds, the computations  $(k x)$  and  $(k y)$  are equivalent.

**Class Fkont\_proper**

```
(M : @EMonoid A op E_one E_equiv) (k : Fkont A) :=
  Fkont_proper_prf:
  Proper (equiv ==> computation_equiv op E_equiv) k.
```

We are now able to improve our definition of correctness, taking only proper continuations into account.

**Definition Fchain\_correct\_nat** (n:nat) (f : Fchain) :=

```
forall A `(M : @EMonoid A op E_one E_equiv) k
  (Hk : Fkont_proper M k)
  (a : A) ,
  computation_execute op (f A k a) ==
  computation_execute op (k (a ^ n)).
```

**Definition Fchain\_correct** (p:positive) (f : Fchain) :=

```
Fchain_correct_nat (Pos.to_nat p) f.
```

### 17.9.2.2 Examples

Let us show manual correctness proofs of some small f-chains:

**Lemma F1\_correct** : Fchain\_correct 1 F1.

**Proof.**

```
intros until M ; intros k Hk a ; unfold F1 ; simpl.
apply Hk ; monoid_simpl M ; reflexivity.
```

**Qed.**

While proving F3's correctness, we will have to apply the properness hypothesis on  $k$ :

**Lemma F3\_correct** : Fchain\_correct 3 F3.

**Proof.**

```
intros until M ; intros k Hk a ; simpl.
```

```

A: Type
op: Mult_op A
E_one: A
E_equiv: Equiv A
M: EMonoid op E_one E_equiv
k: Fkont A
Hk: Fkont_proper M k
a: A
-----
computation_execute op (k (a * a * a)) ==
computation_execute op (k (a * (a * (a * E_one))))

```

```
apply Hk.
```

```

A: Type
op: Mult_op A
E_one: A
E_equiv: Equiv A
M: EMonoid op E_one E_equiv
k: Fkont A
Hk: Fkont_proper M k
a: A
-----
a * a * a == a * (a * (a * E_one))

```

```
monoid_simpl M; reflexivity.
```

**Qed.**

Correctness of F2 is proved the same way:

**Lemma F2\_correct** : Fchain\_correct 2 F2.

**Proof.**

```
intros until M; intros k Hk a; simpl;
apply Hk; monoid_simpl M; reflexivity.
```

**Qed.**

### 17.9.2.3 Composition of correct f-chains: a first attempt

We are now looking for a way to generate correct chains for any positive number. It seems obvious that we could use `Fcompose` for building a correct f-chain for  $n \times p$  by composition of a correct f-chain for  $n$  and a correct f-chain for  $p$ . Let us try to certify this construction:

**Module Bad2.**

**Lemma Fcompose\_correct** :

```
forall f1 f2 n1 n2,
  Fchain_correct n1 f1 ->
  Fchain_correct n2 f2 ->
  Fchain_correct (n1 * n2) (Fcompose f1 f2).
```

**Proof.**

```
(* ... *)
```

```

intros x y Hxy; red.
Hk: Fkont_proper M k
a, x, y: A
Hxy: x == y
-----
computation_execute op (f2 A k x) ==
computation_execute op (f2 A k y)

```

No hypothesis guarantees us that the execution of `f2` respects the equivalence `x == y`.

**Abort.**

**End Bad2.**

Thus, we need to define also a notion of properness for `f`-chains. A first attempt would be :

**Module Bad3.**

```

Class Fchain_proper (fc : Fchain) := Fchain_proper_bad_prf :
  forall `(M : @EMonoid A op E_one E_equiv) k ,
    Fkont_proper M k ->
    forall x y, x == y ->
      @computation_equiv _ op E_equiv
        (fc A k x)
        (fc A k y).

```

This definition is powerful enough for proving that properness is preserved by composition:

```

#[ global ] Instance Fcompose_proper (f1 f2 : Fchain)
  (_ : Fchain_proper f1)
  (_ : Fchain_proper f2) :
  Fchain_proper (Fcompose f1 f2).

```

**Proof.**

```

intros until M; intros k Hk x y Hxy; unfold Fcompose; cbn.
apply (H _ _ _ M); auto.
intros u v Huv; apply (H0 _ _ _ M); auto.

```

**Qed.**

Nevertheless, we had to throw away this definition of properness: In further developments (Sect. 17.9.4 on page 329) we shall have to compare executions of the form `fc A  $k_x$  x` and `fc A  $k_y$  y` where `x == y` and  $k_x$  and  $k_y$  are “equivalent” but not *convertible* continuations.

**End Bad3.**

#### 17.9.2.4 A better definition of properness

The following generalization will allow us to consider continuations that are different (according to Leibniz equality) but lead to equivalent computations and results.

```

Definition Fkont_equiv `(M : @EMonoid A op E_one E_equiv)
(k k' : Fkont A ) :=
forall x y : A, x == y ->
  computation_equiv op E_equiv (k x) (k' y).

```

```

Class Fchain_proper (fc : Fchain) := Fchain_proper_prf :
forall `(M : @EMonoid A op E_one E_equiv) k k' ,
  Fkont_proper M k -> Fkont_proper M k' ->
  Fkont_equiv M k k' ->
forall x y, x == y ->
  @computation_equiv _ op E_equiv
    (fc A k x)
    (fc A k' y).

```

### 17.9.2.5 Examples

The definition above allows us to build simply several instances of the class `Fchain_proper`:

```

#[ global ] Instance F1_proper : Fchain_proper F1.
Proof.
  intros until M ; intros k k' Hk Hk' H a b H0; unfold F1; cbn;
  now apply H.
Qed.

#[ global ] Instance F2_proper : Fchain_proper F2.

#[ global ] Instance F3_proper : Fchain_proper F3.

```

## 17.9.3 Correctness of chain composition

The `Fcompose` operator respects chain correctness and properness.

```

Lemma Fcompose_correct :
forall fc1 fc2 n1 n2,
  Fchain_correct n1 fc1 ->
  Fchain_correct n2 fc2 ->
  Fchain_proper fc2 ->
  Fchain_correct (n1 * n2) (Fcompose fc1 fc2).

#[ global ] Instance Fcompose_proper (fc1 fc2 : Fchain)
  (_ : Fchain_proper fc1)
  (_ : Fchain_proper fc2) :
  Fchain_proper (Fcompose fc1 fc2).

```

Using chain composition, we get a correct and proper chain for any exponent of the form  $2^n$ .

```

Lemma Fexp2_correct (p:positive) :
  Fchain_correct (2 ^ p) (Fexp2 p).

```

```
#[ global ] Instance Fexp2_proper (p:positive) : Fchain_proper (Fexp2 p).
```

We are now able to build chains for any exponent of the form  $2^k \times 3^p$ , using `Fcompose`. Let us look at a simple example:

```
#[global] Hint Resolve F1_correct F1_proper
      F3_correct F3_proper Fcompose_correct Fcompose_proper
      Fexp2_correct Fexp2_proper : chains.
```

**Example F144:** `{f : Fchain | Fchain_correct 144 f /\ Fchain_proper f}`.

**Proof.**

```
change 144 with ( (3 * 3) * (2 ^ 4))%positive.
```

```
{f : Fchain
 | Fchain_correct (3 * 3 * 2 ^ 4) f /\ Fchain_proper f}
```

```
exists (Fcompose (Fcompose F3 F3) (Fexp2 4)); auto with chains.
```

**Defined.**

**Compute** `proj1_sig F144`.

```
= fun (A : Type) (x : Fkont A) (x0 : A) =>
  x1 <--- x0 times x0;
  x2 <--- x1 times x0;
  x3 <--- x2 times x2;
  x4 <--- x3 times x2;
  x5 <--- x4 times x4;
  x6 <--- x5 times x5;
  x7 <--- x6 times x6; x8 <--- x7 times x7; x x8
: Fchain
```

## 17.9.4 Building chains for two distinct exponents : k-chains

### 17.9.4.1 Introduction

Not every chain can be built efficiently with `Fcompose`. For instance, consider the exponent  $n = 23 = 3 + 2^4 + 2^2$ .

One may attempt to define a new operator for combining f-chains for  $n$  and  $p$  into an f-chain for  $n + p$ .

```

Definition Fplus (f1 f2 : Fchain) : Fchain :=
  fun A k x => f1 A
    (fun y =>
      f2 A
        (fun z => t <--- z times y; k t) x) x.

```

**Example** F23 := Fplus F3 (Fplus (Fexp2 4) (Fexp2 2)).

**Lemma** F23\_ok : chain\_correct 23 (F2C F23).

**Proof.**

param\_chain\_correct.

**Qed.**

Unfortunately, our construct is still very inefficient, since it results in duplication of computations, as shown by the normal form of F23.

**Compute** F23.

```

= fun (A : Type) (x : Fkont A) (x0 : A) =>
  x1 <--- x0 times x0;
  x2 <--- x1 times x0;
  x3 <--- x0 times x0;
  x4 <--- x3 times x3;
  x5 <--- x4 times x4;
  x6 <--- x5 times x5;
  x7 <--- x0 times x0;
  x8 <--- x7 times x7;
  x9 <--- x8 times x6;
  x10 <--- x9 times x2; x x10
: Fchain

```

We observe that the variables `x3` and `x7` are useless, since they will have the same value as `x1`. Likewise, computing `x8` (same value as `x4`) is a waste of time.

A better scheme for computing  $x^{23}$  would be the following one:

1. Compute  $x$ ,  $x^2$ ,  $x^3$ , **and**  $x^6 = (x^3)^2$ , then  $x^7$ ,
2. Compute  $x^{10} = x^7 \times x^3$ , then  $x^{20}$
3. Finally, return  $x^{23} = x^{20} \times x^3$

In fact, the first step of this sequence computes *two* values:  $x^7$  and  $x^3$ , that are re-used by the rest of the computation.

Like in some programming languages that allow “multiple values”, like Scheme and Common Lisp, we can express this feature in terms of continuations that accept two arguments. Thus, we extend our previous definitions to chains that return two different powers of their argument<sup>3</sup>.

<sup>3</sup>The name `Kchain` comes from previous versions of this development. It may be changed later.

(\*\* Continuations with two arguments \*\*)

**Definition** `Kkont A := A -> A -> @computation A.`

(\*\* CPS chain builders for two exponents \*\*)

**Definition** `Kchain := forall A, Kkont A -> A -> @computation A.`

### 17.9.4.2 Examples

The chain `k3_1` sends both values  $x$  and  $x^3$  to its continuation. Likewise, `k7_3` “returns”  $x^7$  and  $x^3$ .

**Example** `k3_1 : Kchain := fun A (k:Kkont A) (x:A) =>`  
`x2 <--- x times x ;`  
`x3 <--- x2 times x ;`  
`k x3 x.`

**Example** `k7_3 : Kchain := fun A (k:Kkont A) (x:A) =>`  
`x2 <--- x times x ;`  
`x3 <--- x2 times x ;`  
`x6 <--- x3 times x3 ;`  
`x7 <--- x6 times x ;`  
`k x7 x3.`

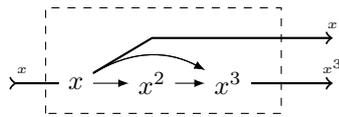


Figure 17.9: Graphical representation of `K3_1`

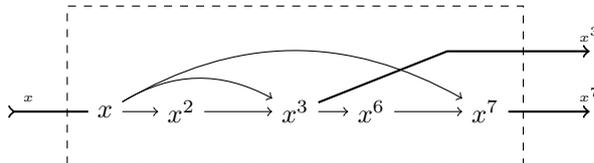


Figure 17.10: Graphical representation of `K7_3`

### 17.9.4.3 Definitions

First, we have to adapt to k-chains our definitions of correctness and properness.

**Definition** `Kkont_proper` (M : @EMonoid A op E_one E_equiv)`  
`(k : Kkont A) :=`  
`Proper (equiv ==> equiv ==> computation_equiv op E_equiv) k .`

**Definition** `Kkont_equiv` (M : @EMonoid A op E_one E_equiv)`

```

      (k k' : Kkont A ) :=
forall x y : A, x == y -> forall z t, z == t ->
  computation_equiv op E_equiv (k x z) (k' y t).

```

A k-chain is correct with respect to two exponents  $n$  and  $p$  if it computes  $x^n$  and  $x^p$  for any  $x$  in any monoid  $M$ .

```

Definition Kchain_correct_nat (n p : nat) (kc : Kchain) :=
forall (A : Type) (op : Mult_op A) (E_one : A) (E_equiv : Equiv A)
  (M : EMonoid op E_one E_equiv)
  (k : Kkont A),
Kkont_proper M k ->
forall (a : A) ,
  computation_execute op (kc A k a) ==
  computation_execute op (k (a ^ n) (a ^ p)).

```

```

Definition Kchain_correct (n p : positive) (kc : Kchain) :=
Kchain_correct_nat (Pos.to_nat n) (Pos.to_nat p) kc.

```

```

Class Kchain_proper (kc : Kchain) :=
Kchain_proper_prf :
forall `(M : @EMonoid A op E_one E_equiv) k k' x y ,
  Kkont_proper M k ->
  Kkont_proper M k' ->
  Kkont_equiv M k k' ->
  E_equiv x y ->
  computation_equiv op E_equiv (kc A k x) (kc A k' y).

```

#### 17.9.4.4 Example

For instance, let us prove that `k7_3` is proper and correct for the exponents 7 and 3.

```

#[ global ] Instance k7_3_proper : Kchain_proper k7_3.
Proof.
  intros until M; intros; red; unfold k7_3; cbn;
  add_op_proper M H3; apply H1; rewrite H2; reflexivity.
Qed.

```

```

Lemma k7_3_correct : Kchain_correct 7 3 k7_3.
Proof.
  intros until M; intros; red; unfold k7_3; simpl.
  apply H; monoid_simpl M; reflexivity.
Qed.

```

### 17.9.5 Systematic construction of correct f-chains and k-chains

We are now ready to define various operators on f- and k-chains, and prove these operators preserve correctness and properness. We will also show that

these operators allow to generate easily correct chains for any positive exponent. They will be used to generate chains for numbers of the form  $n = bq + r$  where  $0 \leq r < b$ , assuming the previous construction of correct chains for  $r$ ,  $b$  and  $q$ . For instance, Figure 17.11 shows how  $K7\_3$  is built as a composition of  $K3\_1$  and  $F2$ .

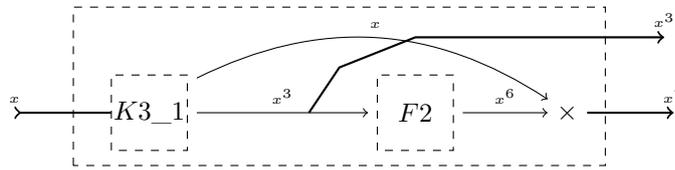


Figure 17.11: Decomposition of  $K7\_3$

### 17.9.5.1 Conversion from k-chains into f-chains

Any k-chain for  $n$  and  $p$  can be converted into an f-chain, just by applying it to a continuation that ignores its second argument.

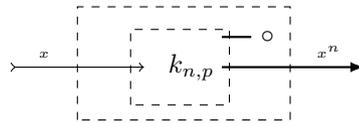


Figure 17.12: The K2F ( $knp$ ) construction

```
Definition K2F (knp : Kchain) : Fchain :=
  fun A (k:Fkont A) => knp A (fun y _ => k y).
```

```
Lemma K2F_correct :
  forall kc n p, Kchain_correct n p kc ->
    Fchain_correct n (K2F kc).
```

```
#[ global ] Instance K2F_proper (kc : Kchain)(_ : Kchain_proper kc) :
  Fchain_proper (K2F kc).
```

### 17.9.5.2 Construction associated with Euclidean division with a positive rest

Let  $n = bq + r$ , with  $0 < r < b$ . Then, for any  $x$ ,  $x^n = (x^b)^q \times x^r$ . Thus, we can compose an chain that computes  $x^b$  and  $x^r$  with a chain that raises any  $y$  to its  $q$ -th power for obtaining a chain that computes  $x^n$ .

```
Definition KFK (kbr : Kchain) (fq : Fchain) : Kchain :=
  fun A k a =>
    kbr A (fun xb xr =>
      fq A (fun y =>
        z <--- y times xr; k z xb) xb) a.
```

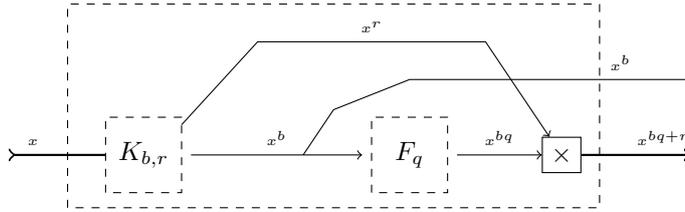


Figure 17.13: The KFK combinator

**Lemma KFK\_correct :**

```
forall (b q r : positive) (kbr : Kchain) (fq : Fchain),
  Kchain_correct b r kbr ->
  Fchain_correct q fq ->
  Kchain_proper kbr ->
  Fchain_proper fq ->
  Kchain_correct (b * q + r) b (KFK kbr fq).
```

**Check KFK\_proper.**

```
KFK_proper
: forall (kbr : Kchain) (fq : Fchain),
  Kchain_proper kbr ->
  Fchain_proper fq -> Kchain_proper (KFK kbr fq)
```

### 17.9.5.3 Ignoring the remainder

Let  $n = bq + r$ , with  $0 < r < b$ . The following construction computes  $x^r$  and  $x^b$ , then  $x^{bq}$ , and finally sends  $x^{bq+r}$  to the continuation, throwing away  $x^b$ .

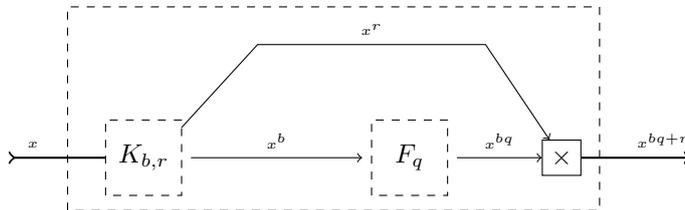


Figure 17.14: The KFF combinator

**Definition KFF** (kbr : Kchain) (fq : Fchain) : Fchain :=  
K2F (KFK kbr fq).

**Lemma KFF\_correct :**

```
forall (b q r : positive) (kbr : Kchain) (fq : Fchain),
  Kchain_correct b r kbr ->
  Fchain_correct q fq ->
  Kchain_proper kbr ->
  Fchain_proper fq ->
  Fchain_correct (b * q + r) (KFF kbr fq).
```

**Check** KFK\_proper.

```
KFK_proper
: forall (kbr : Kchain) (fq : Fchain),
  Kchain_proper kbr ->
  Fchain_proper fq -> Kchain_proper (KFK kbr fq)
```

#### 17.9.5.4 Conversion of an f-chain into a k-chain

The following conversion is useful when a chain generation algorithm needs to build a k-chain for exponents  $p$  and 1:

**Definition** FK ( $f : Fchain$ ) : Kchain :=  
 fun (A : Type) (k : Kkont A) (a : A) =>  
 f A (fun y => k y a) a.

Like our other combinators, FK respects chain correctness and properness.

#### 17.9.5.5 Computing $x^p$ and $x^{pq}$

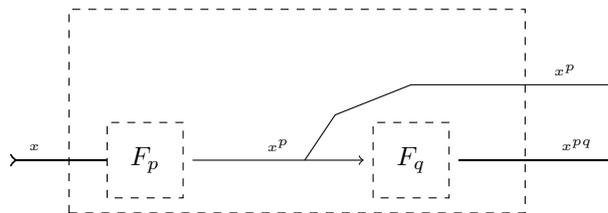


Figure 17.15: The FFK combinator

Our last combinator composes a chain for computing  $x^p$  with a chain for computing  $x^q$  to build a chain for computing  $x^p$  and  $x^{pq}$ .

**Definition** FFK ( $fp\ fq : Fchain$ ) : Kchain :=  
 fun A k a => fp A (fun xb => fq A (fun y => k y xb) xb) a.

**Lemma** FFK\_correct ( $p\ q : \text{positive}$ ) ( $fp\ fq : Fchain$ ):  
 Fchain\_correct p fp ->  
 Fchain\_correct q fq ->  
 Fchain\_proper fp ->  
 Fchain\_proper fq ->  
 Kchain\_correct (p \* q) p (FFK fp fq).

```
#[ global ] Instance FFK_proper
  (fp fq : Fchain)
  (_ : Fchain_proper fp)
  (_ : Fchain_proper fq)
: Kchain_proper (FFK fp fq).
```

### 17.9.5.6 A correct-by-construction chain

A simple example will show us how to build correct chains for any positive exponent, using the operators above.

```
#[global] Hint Resolve KFF_correct KFF_proper KFK_correct KFK_proper : chains.
```

```
Definition F87 :=
  let k7_3 := KFK k3_1 (Fexp2 1) in
  let k10_7 := KFK k7_3 F1 in
  KFF k10_7 (Fexp2 3).
```

```
Compute the_exponent (F2C F87).
```

```
= 87
: positive
```

```
Lemma OK87 : Fchain_correct 87 F87.
```

**Proof.**

```
unfold F87; change 87 with (10 * (2 ^ 3) + 7)%positive.
apply KFF_correct;auto with chains.
change 10 with (7 * 1 + 3);
  apply KFK_correct;auto with chains.
change 7 with (3 * 2 ^ 1 + 1)%positive;
  apply KFK_correct;auto with chains.
```

**Qed.**

Note that this method of construction still requires some interaction from the user. In the next section, we build a *function* that maps any positive number  $n$  into a correct and proper chain for  $n$ . Thus correct chain generation will be fully automated.

## 17.9.6 Automatic chain generation by Euclidean division

The goal of this section is to write a function `make_chain (p:positive): chain` that builds a correct chain for  $p$ , using the Euclidean method above. In other words, we want to get correct chains by computation. The correctness of the result of this computation should be asserted by a theorem:

```
Theorem make_chain_correct :
  forall p, chain_correct p (make_chain p).
```

In the previous section, we considered two different kinds of objects: f-chains, associated with a single exponent, and k-chains, associated with two exponents. We would expect that the function `make_chain` we want to define and certify is structured as a pair of mutually recursive functions. In Coq, various ways of building such functions are available:

- Structural [mutual] recursion with `Fixpoint`
- Using `Program Fixpoint`

- Using Function.

Since our construction is based on Euclidean division, we could not define our chain generator by structural recursion. For simplicity's sake, we chose to avoid dependent elimination and used `Function` with a decreasing measure.

For this purpose, we define a single data-type for associated with the generation of F- and K-chains.

We had two slight technical problems to consider:

- The generation of a k-chain for  $n$  and  $p$  is meaningful only if  $p < n$ . Thus, in order to avoid a clumsy dependent pattern-matching, we chose to represent a pair  $(n, p)$  where  $0 < p < n$  by a pair of positive numbers  $(p, d)$  where  $d = n - p$
- In order to avoid to deal explicitly with mutual recursion, we defined a type called `signature` for representing both forms of function calls. Thus, it is easy to define a decreasing measure on type `signature` for proving termination. Likewise, correctness and properness statements are also indexed by this type.

```
Inductive signature : Type :=
| gen_F (n:positive) (** Fchain for the exponent n *)
| gen_K (p d: positive) (** Kchain for the exponents p+d and p *).
```

The following dependently-typed functions will help us to specify formally any correct chain generator.

```
Definition signature_exponent (s:signature) : positive :=
  match s with
  | gen_F n => n
  | gen_K p d => p + d
end.
```

```
Definition kont_type (s: signature)(A:Type) : Type :=
  match s with
  | gen_F _ => Fkont A
  | gen_K _ _ => Kkont A
end.
```

```
Definition chain_type (s: signature) : Type :=
  match s with
  | gen_F _ => Fchain
  | gen_K _ _ => Kchain
end.
```

```
Definition correctness_statement (s: signature) :
  chain_type s -> Prop :=
  match s with
  | gen_F p => fun ch => Fchain_correct p ch
  | gen_K p d => fun ch => Kchain_correct (p + d) p ch
end.
```

```

Definition proper_statement (s: signature) :
chain_type s -> Prop :=
match s with
| gen_F _ => fun ch => Fchain_proper ch
| gen_K _ _ => fun ch => Kchain_proper ch
end.

```

```

(** ** Full correctness *)

```

```

Definition OK (s: signature)
:= fun c: chain_type s => correctness_statement s c /\
proper_statement s c.

```

### 17.9.7 Generation of chains using Euclidean Division

Assume we want to build automatically a correct f-chain for some positive integer  $n$ . If  $n$  equals to 1, 3, or  $2^p$  for some positive integer  $p$ , this task is immediate, thanks to the constants F1, F3 and Fexp2. Otherwise, like in [BCHM95], we decompose  $n$  into  $bq + r$ , where  $1 < b < n$ , and compose the recursively built chains for  $q$  and  $r$  on one side, and  $q$  on the other side.

The efficiency of this method depends on the choice of  $b$ . In [BCHM95], the function that maps  $n$  into  $b$  is called a *strategy*.

From additions.Strategies.

```

Class Strategy (gamma : positive -> positive):=
{
gamma_lt : forall p:positive, 3 < p -> gamma p < p;
gamma_gt : forall p:positive, 3 < p -> 1 < gamma p
}.

```

### 17.9.8 The dichotomic strategy

In this chapter, we concentrate on the so-called *dichotomic strategy*, defined as follows:

$$n \mapsto n \div 2^k \text{ where } k = \lfloor (\log_2 n)/2 \rfloor$$

Intuitively, it corresponds to splitting the binary representation of a positive integer into two halves. For instance, consider  $n = 87$  its binary representation is 1010111. The number  $\lfloor (\log_2 n)/2 \rfloor$  is equal to 3. Dividing  $n$  by  $2^3$  gives the decomposition  $n = 10 \times 2^3 + 7$ . Thus, a chain for  $n = 87$  can be built from a chain computing both  $x^7$  and  $x^{10}$ , and a chain that raises its argument to its 8-th power.

This strategy is defined in Module additions.Dichotomy.

```

Function dichotomic_aux (p:positive) {struct p} : positive :=
match p with
| 1%positive => xH
| 2%positive | 3%positive => 2

```

```
| x0 (x0 q) | x0 (xI q) | xI (x0 q) | xI (xI q) =>
                                x0 (dicho_aux q)
```

```
end.
```

```
Definition dicho (p:positive) : positive :=
  N2pos (N.div (Npos p) (Npos (dicho_aux p))).
```

```
Compute dicho 87.
```

```
= 10
: positive
```

```
Compute dicho 130.
```

```
= 8
: positive
```

```
Compute dicho 128.
```

```
= 8
: positive
```

```
#[ global ] Instance Dicho_strat : Strategy dicho.
```

### 17.9.9 Other strategies

For comparison's sake, we define two other strategies, much simpler but statically less efficient than the dichotomic strategy.

*From Module additions.BinaryStrat.*

```
Definition half (p:positive) :=
  match p with xH => xH
            | xI q | x0 q => q
end.
```

```
Definition two (p:positive) := 2%positive.
```

```
#[ global ] Instance Binary_strat : Strategy half.
```

```
Proof.
```

```
split; destruct p; unfold half; try lia.
```

```
Qed.
```

```
#[ global ] Instance Two_strat : Strategy two.
```

```
Proof.
```

```
split;unfold two; lia.
```

```
Qed.
```

Page 341, we compare the three strategies with respect to the length of the built chains.

### 17.9.10 Main chain generation function

We are now able to define a function that generates a correct chain for any signature. We use the `Recdef` module of Standard Library, with an appropriate *measure*.

**Section** `Gamma`.

**Variable** `gamma`: positive  $\rightarrow$  positive.

**Context** (`Hgamma` : Strategy `gamma`).

```
Definition signature_measure (s : signature) : nat :=
match s with
| gen_F n => 2 * Pos.to_nat n
| gen_K p d => 2 * Pos.to_nat (p + d) + 1
end.
```

The following function definition generates 9 proof obligations subgoals, for proving that the measure on signatures is strictly decreasing along the recursive calls. They are solved with the help of Standard Library's lemmas on arithmetic of positive numbers and Euclidean division.

```
Function chain_gen (s:signature) {measure signature_measure}
: chain_type s :=
match s return chain_type s with
| gen_F i =>
  if pos_eq_dec i 1 then F1 else
  if pos_eq_dec i 3
  then F3
  else
    match exact_log2 i with
    Some p => Fexp2 p
    | _ =>
      match N.pos_div_eucl i (Npos (gamma i))
      with
      | (q, 0%N) =>
        Fcompose (chain_gen (gen_F (gamma i)))
          (chain_gen (gen_F (N2pos q)))
      | (q, _r) => KFF (chain_gen
          (gen_K (N2pos _r)
            (gamma i - N2pos _r)))
          (chain_gen (gen_F (N2pos q)))
      end end
| gen_K p d =>
  if pos_eq_dec p 1 then FK (chain_gen (gen_F (1 + d)))
  else
    match N.pos_div_eucl (p + d) (Npos p) with
    | (q, 0%N) => FFK (chain_gen (gen_F p))
      (chain_gen (gen_F (N2pos q)))
    | (q, _r) => KFK (chain_gen (gen_K (N2pos _r)
      (p - N2pos _r)))
      (chain_gen (gen_F (N2pos q)))
```

```

    end
  end.
(* 9 Proof Obligations generated *)
Definition make_chain (n:positive) : chain :=
  F2C (chain_gen (gen_F n)).
Theorem make_chain_correct : forall p, chain_correct p (make_chain p).
Proof.
  intro p; destruct (chain_gen_OK (gen_F p)).
  unfold make_chain; apply F2C_correct; apply H.
Qed.

End Gamma.

Compute the_exponent (make_chain dicho 87).
= 87
: positive

```

### 17.9.10.1 A few tests

The following tests show various examples of chains for the same exponent, using different strategies. The dichotomic strategy seems clearly to be the winner (at least on this sample)<sup>4</sup>.

```
Compute chain_length (make_chain two 56789).
```

```
= 25%nat : nat
```

```
Compute chain_length (make_chain half 56789).
```

```
= 25%nat : nat
```

```
Compute chain_length (make_chain dicho 56789).
```

```
= 21%nat : nat
```

```
Compute chain_length (make_chain two 3456789).
```

```
= 33%nat : nat
```

```
Compute chain_length (make_chain half 3456789).
```

```
(= 33%nat : nat
```

```
Compute chain_length (make_chain dicho 3456789).
```

```
= 29%nat : nat
```

<sup>4</sup>For efficiency's sake, we commented out some (very) long computations. You may uncomment them freely in your own copy. For the same reason, we put a verbatim trace instead of Alectryon output

### 17.9.10.2 Correctness of the Euclidean chain generator

Recdef's functional induction tactic allows us to prove that every value returned by `(chain_gen s)` is correct w.r.t. `s` and proper. The proof obligations are solved thanks to the previous lemmas on the composition operators on chains: `Fcompose`, `KFK`, etc. Unfortunately, a lot of interaction is still needed for proving properties of Euclidean division and binary logarithm.

**Lemma** `chain_gen_OK` : forall s:signature,  
OK s (chain\_gen s).

**Proof.**

```
intro s; functional induction chain_gen s.
(* A lot of arithmetic sub-proofs ... *)
```

### 17.9.10.3 A last example

Let us compute  $67777^{6145319}$  with 32 bits integers:

```
Ltac compute_chain ch :=
  let X := fresh "x" in
  let Y := fresh "y" in
  let X := constr:ch in
  let Y := (eval vm_compute in X) in
  exact Y.

Let big_chain := ltac:(compute_chain (make_chain 6145319)).

Print big_chain.
```

```
big_chain =
fun (A : Type) (x : A) =>
x0 <--- x times x; x1 <--- x0 times x0;
x2 <--- x1 times x1; x3 <--- x2 times x1;
x4 <--- x3 times x3; x5 <--- x4 times x;
x6 <--- x5 times x5; x7 <--- x6 times x6;
x8 <--- x7 times x1; x9 <--- x8 times x5;
x10 <--- x9 times x8; x11 <--- x10 times x9;
x12 <--- x11 times x11; x13 <--- x12 times x11;
x14 <--- x13 times x10; x15 <--- x14 times x14;
x16 <--- x15 times x11; x17 <--- x16 times x16;
x18 <--- x17 times x17; x19 <--- x18 times x18;
x20 <--- x19 times x19; x21 <--- x20 times x20;
x22 <--- x21 times x21; x23 <--- x22 times x22;
x24 <--- x23 times x23; x25 <--- x24 times x24;
x26 <--- x25 times x25; x27 <--- x26 times x26;
x28 <--- x27 times x14; Return x28
  : forall A : Type, A -> computation
```

```
Time Compute Int31.phi
(chain_apply big_chain (snd (positive_to_int31 67777))).
```

```
= 2014111041%Z
   : Z
Finished transaction in 0.005 secs (0.005u,0.s) (successful)}
```

```
Compute chain_length big_chain.
```

```
= 29%nat
   : nat
```

### 17.9.11 Fibonacci, *le retour*

It is now possible to use Euclidean addition chains for computing Fibonacci numbers (see Sections 17.2.3.2 on page 285 and 17.4.6 on page 300).

The following function is parameterized by any strategy  $\gamma$ .

```
Definition fib_eucl gamma `{Hgamma: Strategy gamma} n :=
  let c := make_chain gamma n
  in let r := chain_apply c (M:=Mul2) (1,0) in
     fst r + snd r.
```

**Time Compute** fib\_eucl dicho 153.

```
= 68330027629092351019822533679447
   : N
Finished transaction in 0.014 secs (0.014u,0.s) (successful)
```

**Time Compute** fib\_eucl two 153.

```
= 68330027629092351019822533679447
   : N
Finished transaction in 0.011 secs (0.011u,0.s) (successful)
```

**Time Compute** fib\_eucl half 153.

```
= 68330027629092351019822533679447
   : N
Finished transaction in 0.01 secs (0.007u,0.003s) (successful)
```

## 17.10 Projects

### Project 17.3 (Optimality and relative efficiency)

1. Prove that the chain generated by `Fexp2` is optimal.
2. Prove that the length of any optimal chain for  $n$  is greater than or equal to  $\lfloor \log_2 n \rfloor$ .
3. Prove that, for any positive  $n$ , the length of any Euclidean chain generated by the dichotomic strategy is always less than or equal to the length of `binary_chain n`, and for an infinite number of positive integers  $n$ , the first chain is strictly shorter than the latter.

4. Prove that our implementation of the dichotomic strategy describes the same function as in the literature (for instance [BCHM95].) This is important if we want to follow the complexity analyses in this and similar articles.
5. Study how to *compile* a chain into imperative code, using a register allocation strategy (it may be useful to define *chain width* ).

**Remark:** The first two questions of the list above should involve a universal quantification on type *chain*. It may be necessary (but we're not sure) to consider some restriction on parametric chains.

### 17.10.1 A data structure for Euclidean chains

Figures 17.6 on page 321 to 17.15 on page 335 suggest that any computation following an Euclidean chain can be executed on a kind of abstract machine with a "register" and a stack, and only four operations:

- multiply the contents of the register by the top of the stack (and pop that stack),
- raising the contents of the register to its square,
- push the contents of the register into the stack,
- swapping the two elements at the top of the stack.

In Coq, we define the instructions as the four constructors of an inductive type.

From Module additions.AM

```
Inductive instr : Set :=
| MUL : instr
| SQR : instr
| PUSH : instr
| SWAP : instr.
```

```
Definition code := list instr.
```

**Section Semantics.**

```
Variable A : Type.
Variable mul : A -> A -> A.
Variable one : A.
```

```
Definition stack := list A.
Definition config := (A * list A)%type.
```

```
Fixpoint exec (c : code) (x:A) (s: stack) : option config :=
  match c, s with
  | nil, _ => Some (x,s)
  | MUL::c, y::s => exec c (mul x y) s
```

```

| SQR::c, s => exec c (mul x x) s
| PUSH::c, s => exec c x (x::s)
| SWAP::c, y::z::s => exec c x (z::y::s)
| _,_ => None
end.

Lemma exec_app :
  forall (c c' : code) x s ,
    exec (c ++ c') x s =
      match exec c x s with
      | None => None
      | Some (y,s') => exec c' y s'
      end.

(** Main well-formed chains *)
Definition F1 : code := nil.

(** raises x to its cube *)

Definition F3 := PUSH::SQR::MUL::nil.

(** chain for raising x to its (2 ^ q)th power *)

Fixpoint F2q_of_nat q := match q with
  | 0%nat => nil
  | S p => SQR:: F2q_of_nat p
  end.

Definition F2q (p:positive) :=
  F2q_of_nat (Pos.to_nat p).

(** for computing x^(pq+r) passing by x^p *)

Definition KFF (kpr mq:code) : code :=
  kpr++(mq++MUL::nil).

(** for computing x^p and x^(pq) *)

Definition FFK (mp mq: code) := mp ++ PUSH :: mq.

(** for computing x^p then x^(pq + r) *)

Definition KFK (kpr mq: code) :=
  kpr ++ PUSH::SWAP :: (mq ++ MUL :: nil).

Definition FK (fn: code) := PUSH::fn.

End Semantics.

Definition chain_apply c {A:Type}

```

```

      {op:A->A->A}{one:A}{equ: Equiv A}
      (M: EMonoid op one equ) x
:= exec _ op c x nil.

```

```

(** Example code for 7 via 3 *)

```

```

Example M7_3 := PUSH::PUSH::SQR::MUL::PUSH::SQR::SWAP::MUL::nil.

```

```

Compute chain_apply M7_3 Natplus 1%nat .

```

```

= Some (7%nat, 3%nat :: nil)
: option (config nat)

```

```

(** Example code for 31 via 7 *)

```

```

Example C31_7 := KFF M7_3 (F2q 2).

```

```

Compute chain_apply C31_7 Natplus 1%nat.

```

```

= Some (31%nat, nil)
: option (config nat)

```

For instance the chain of Fig. 17.4 on page 305 can be represented with the following code:

```

Compute chain_gen dichot (gen_F 87).

```

```

= PUSH
  :: PUSH
    :: SQR
      :: MUL
        :: PUSH
          :: SWAP
            :: SQR
              :: MUL
                :: PUSH
                  :: SWAP
                    :: MUL
                      :: SQR
                        :: SQR
                          :: SQR
                            :: MUL
                              :: nil
: code

```

In the library additions.AM, we define a chain generator for this data structure. Please note that many proof scripts are copied verbatim from `Euclidean_Chains` into `AM`. Removing such redundancies is left as a project.

**Project 17.4 (Some improvements)** 1. Improve automated proofs on types `positive` and `N`.

2. Compare `Program Fixpoint` and `Function` for writing `make_chain`. Consider measure *vs* well-founded relations, mutual recursion, possibility of using sigma-types, etc.

3. Chains are always associated with strictly positive exponents. Thus, many lemmas about chain correctness can be proved using semi-groups instead of monoids. Define type classes for semi-groups and use them whenever possible.



**Part IV**

**Appendices**



# Bibliography

- [Abr96] Jean-Raymond Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.
- [Abr10] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.
- [Bau] Andrej Bauer. The hydra game. <http://math.andrej.com/2008/02/02/the-hydra-game>.
- [Bau08] Andrej Bauer. The hydra game source code. <https://github.com/andrejbauer/hydra>, 2008.
- [BB87] Jean Berstel and Srećko Brlek. On the length of word chains. *Information Processing Letters*, 26(1):23–28, 1987. <http://www-igm.univ-mlv.fr/~berstel/Articles/1987WordChains.pdf>.
- [BBB<sup>+</sup>22] Jonas Bayer, Christoph Benzmüller, Kevin Buzzard, Marco David, Leslie Lamport, Yuri Matiyasevich, Lawrence Paulson, Dierk Schleicher, Benedikt Stock, and Efim Zelmanov. Mathematical proof between generations. <https://arxiv.org/abs/2207.04779>, 2022.
- [BC04a] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer, Berlin, Heidelberg, 2004. <https://www.labri.fr/perso/casteran/CoqArt/>.
- [BC04b] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*, chapter Foundations of Inductive Types. Springer, 2004. <https://www-sop.inria.fr/members/Yves.Bertot/coqart-chapter14.pdf>.
- [BCHM95] Srećko Brlek, Pierre Castéran, Laurent Habsieger, and Richard Mallette. On-line evaluation of powers using Euclid's algorithm. *RAIRO - Theoretical Informatics and Applications - Informatique Théorique et Applications*, 29(5):431–450, 1995. [http://www.numdam.org/item/ITA\\_1995\\_\\_29\\_5\\_431\\_0.pdf](http://www.numdam.org/item/ITA_1995__29_5_431_0.pdf).
- [BCS91] Srećko Brlek, Pierre Castéran, and Robert Strandh. On addition schemes. In *International Joint Conference on Theory and Practice of Software Development*, pages 379–393, Berlin, Heidelberg,

1991. Springer. [https://link.springer.com/content/pdf/10.1007%2F3540539816\\_77.pdf](https://link.springer.com/content/pdf/10.1007%2F3540539816_77.pdf).
- [BMR16] Pierre-Léo Bégay, Pascal Manoury, and Itsaka Rakotonirina. Une mesure ordinaire pour les preuves de terminaison en coq. In *Journées Francophones des Langages Applicatifs*, 2016. <https://hal.archives-ouvertes.fr/hal-01333597/document>.
- [BP01] Gilles Barthe and Olivier Pons. Type isomorphisms and proof reuse in dependent type theory. In *Proceedings of ETAPS 2001*, volume 2030, pages 57–71, 01 2001.
- [Bra39] Alfred Brauer. On addition chains. *Bulletin of the American Mathematical Society*, 45(10):736–739, 10 1939. <https://www.ams.org/journals/bull/1939-45-10/S0002-9904-1939-07068-7/S0002-9904-1939-07068-7.pdf>.
- [Bro10] Daniel Brown. Parametricity. <https://web.archive.org/web/20190628092255/http://www.ccs.neu.edu/home/matthias/369-s10/Transcript/parametricity.pdf>, 2010. Transcript of a lecture by Matthias Felleisen.
- [Bur75] William H. Burge. *Recursive programming techniques*. Addison-Wesley, 1975.
- [Can55] Georg Cantor. *Contributions to the Founding of the Theory of Transfinite Numbers*. Courier Corporation, 1955.
- [Cas04] Pierre Castéran. Additions. User Contributions to the Coq Proof Assistant, 2004. <https://github.com/coq-contribs/additions>.
- [Cas07] Pierre Castéran. Utilisation en Coq de l’opérateur de description. In *Actes des Journées Francophones des Langages Applicatifs*, pages 30–44, 2007. [http://jfla.inria.fr/2007/actes/PDF/03\\_casteran.pdf](http://jfla.inria.fr/2007/actes/PDF/03_casteran.pdf).
- [CC06] Pierre Castéran and Évelyne Contejean. On ordinal notations. User Contributions to the Coq Proof Assistant, 2006. <https://github.com/coq-contribs/cantor>.
- [CCK] The Coq community project. <https://github.com/coq-community/>.
- [CDM13] Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for free! In *Certified Programs and Proofs*, pages 147–162, Cham, 2013. Springer. <https://hal.inria.fr/hal-01113453>.
- [CDP<sup>+</sup>22] Pierre Castéran, Jérémy Damour, Karl Palmkog, Clément Pit-Claudel, and Théo Zimmermann. Hydras & Co.: Formalized mathematics in Coq for inspiration and entertainment. 2022. Proceedings of JFLA 2022.
- [Chl08] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *International Conference on Functional Programming*, pages 143–156, New York, NY, USA, 2008. ACM. <http://adam.chlipala.net/papers/PhoasICFP08/>.

- [Chl11] Adam Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2011. <http://adam.chlipala.net/cpdt/>.
- [CLKK07] Hubert Comon-Lundh, Claude Kirchner, and H el ene Kirchner, editors. *Rewriting, Computation and Proof: Essays Dedicated to Jean-Pierre Jouannaud on the Occasion of His 60th Birthday*. Springer, Berlin, Heidelberg, 2007. <https://link.springer.com/book/10.1007%2F978-3-540-73147-4>.
- [CN04] Pierre Cassou-Nogu es. *G odel*. Les Belles Lettres, 2004. In French.
- [Coq] Coq Development Team. The Coq Proof Assistant. <https://coq.inria.fr>.
- [CPC23] Shardul Chiplunkar and Cl ement Pit-Claudel. Diagrammatic notations for interactive theorem proving. In *4th International Workshop on Human Aspects of Types and Reasoning Assistants*, Cascais, Portugal, 2023. <https://infoscience.epfl.ch/record/305144>.
- [CPU<sup>+</sup>10]  velyne Contejean, Andrei Paskevich, Xavier Urbain, Pierre Courtieu, Olivier Pons, and Julien Forest. A3PAT, an approach for certified automated termination proofs. In *Workshop on Partial Evaluation and Program Manipulation*, pages 63–72, New York, NY, USA, 2010. Association for Computing Machinery. <https://hal.inria.fr/inria-00535655>.
- [CS] Pierre Cast eran and Matthieu Sozeau. A gentle Introduction to Type Classes and Relations in Coq. <https://www.labri.fr/perso/casteran/CoqArt/TypeClassesTut/typeclassestut.pdf>.
- [Der82] Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279–301, 1982. <https://www.sciencedirect.com/science/article/pii/0304397582900263/pdf>.
- [DM07] Nachum Dershowitz and Georg Moser. The hydra battle revisited. In *Rewriting, Computation and Proof: Essays Dedicated to Jean-Pierre Jouannaud on the Occasion of His 60th Birthday*, pages 1–27. Springer, Berlin, Heidelberg, 2007. <https://www.cs.tau.ac.il/~nachum/papers/LNCS/Hydra.pdf>.
- [Dow23] Gilles Dowek. Teaching G odel’s incompleteness theorems, 2023. <https://arxiv.org/pdf/2303.18099.pdf>.
- [G86] Kurt G odel. *Collected Works*. Oxford University Press, 1986.
- [GAA<sup>+</sup>13] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, Fran ois Garillot, St ephane Le Roux, Assia Mahboubi, Russell O’Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Th ery. A machine-checked proof of the odd order theorem. In *Interactive Theorem Proving*, pages 163–179, Berlin, Heidelberg, 2013. Springer. <https://hal.inria.fr/hal-00816699>.

- [Gal91] Jean H. Gallier. What's so special about Kruskal's theorem and the ordinal  $\Gamma_0$ ? A survey of some results in proof theory. *Annals of Pure and Applied Logic*, 53(3):199–260, 1991. <https://www.sciencedirect.com/science/article/pii/016800729190022E/pdf>.
- [GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993. <http://www.cs.ox.ac.uk/tom.melham/pub/Gordon-1993-ITH.html>.
- [Gon08] Georges Gonthier. Formal proof—the four-color theorem. *Notices of the American Mathematical Society*, 55(11):1382–1393, 2008. <http://www.ams.org/notices/200811/tx081101382p.pdf>.
- [Goo44] R. L. Goodstein. On the restricted ordinal theorem. *Journal of Symbolic Logic*, 9(2):33–41, 1944. <https://www.jstor.org/stable/2268019>.
- [GQS] José Grimm, Alban Quadrat, and Carlos Simpson. Gaia. <https://github.com/coq-community/gaia>. A Coq-community project.
- [Gri09a] José Grimm. Implementation of Bourbaki's Elements of Mathematics in Coq: Part one, theory of sets. Research Report RR-6999, INRIA, 2009. <https://hal.inria.fr/inria-00408143>.
- [Gri09b] José Grimm. Implementation of Bourbaki's Elements of Mathematics in Coq: Part two; ordered sets, cardinals, integers. Research Report RR-7150, INRIA, 2009. <https://hal.inria.fr/inria-00440786>.
- [Gri13] José Grimm. Implementation of three types of ordinals in Coq. Research Report RR-8407, INRIA, 2013. <https://hal.inria.fr/hal-00911710>.
- [Gri14] José Grimm. Fibonacci numbers and the Stern-Brocot tree in Coq. Research Report RR-8654, INRIA, 2014. <https://hal.inria.fr/hal-01093589>.
- [Gri16] José Grimm. Implementation of Bourbaki's Elements of Mathematics in Coq: Part three structures. Research Report RR-8997, INRIA, 2016. <https://hal.inria.fr/hal-01412037>.
- [HAB<sup>+</sup>17] Thomas Hales, Mark Adams, Gertrud Bauer, Tat Dat Dang, John Harrison, Le Truong Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Tat Thang Nguyen, Quang Truong Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason Rute, Alexey Solovyev, Thi Hoai An Ta, Nam Trung Tran, Thi Diep Trieu, Josef Urban, Ky Vu, and Roland Zumkeller. A formal proof of the Kepler conjecture. *Forum of Mathematics, Pi*, 5, 2017. <https://arxiv.org/abs/1501.02155>.
- [HBk] Hydra battles. <https://github.com/coq-community/hydra-battles>. A Coq-community project.

- [Hof99] Douglas R. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid*. Basic Books, 20 anv edition, February 1999.
- [Hue97] Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, September 1997. <https://www.cambridge.org/core/journals/journal-of-functional-programming/article/zipper/0C058890B8A9B588F26E6D68CF0CE204>.
- [KP82] Laurie Kirby and Jeff Paris. Accessible independence results for Peano arithmetic. *Bulletin of the London Mathematical Society*, 14(4):285–293, 1982. [https://faculty.baruch.cuny.edu/lkirby/accessible\\_independence\\_results.pdf](https://faculty.baruch.cuny.edu/lkirby/accessible_independence_results.pdf).
- [KS81] Jussi Ketonen and Robert Solovay. Rapidly growing Ramsey functions. *Annals of Mathematics*, 113(2):267–314, 1981. <http://www.jstor.org/stable/2006985>.
- [Mag03] Nicolas Magaud. Changing Data Representation within the Coq System. In *TPHOLs'2003*, volume 2758 of *LNCS*. Springer-Verlag, 2003.
- [MT18] Assia Mahboubi and Enrico Tassi. Mathematical Components. <https://doi.org/10.5281/zenodo.3999478>, 2018. With contributions by Yves Bertot and Georges Gonthier.
- [MV05] Panagiotis Manolios and Daron Vroon. Ordinal arithmetic: Algorithms and mechanization. *Journal of Automated Reasoning*, 34(4):387–423, May 2005. <http://www.ccs.neu.edu/home/pete/pub/ordinal-arithmetic-algs-mech.pdf>.
- [NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, Berlin, Heidelberg, 2002. <https://link.springer.com/book/10.1007%2F3-540-45949-9>.
- [O’C05a] Russel O’Connor. Goedel. <https://github.com/coq-community/goedel>, 2005. A Coq-community project.
- [O’C05b] Russell O’Connor. Essential incompleteness of arithmetic verified by Coq. In *International Conference on Theorem Proving in Higher Order Logics*, pages 245–260, Berlin, Heidelberg, 2005. Springer. <https://arxiv.org/abs/cs/0505034>.
- [P<sup>+</sup>] Benjamin Pierce et al. Software Foundations. <https://softwarefoundations.cis.upenn.edu>.
- [PAU21] LAWRENCE C. PAULSON. Ackermann’s function in iterative form: A proof assistant experiment. *The Bulletin of Symbolic Logic*, 27(4):426–435, 2021.
- [PC] Clément Pit-Claudel. Alectryon. <https://github.com/cpitclaudel/alectryon>.

- [PC20] Clément Pit-Claudel. Untangling mechanized proofs. In *International Conference on Software Language Engineering*, pages 155–174, New York, NY, USA, 2020. Association for Computing Machinery. <https://dl.acm.org/doi/pdf/10.1145/3426425.3426940>.
- [Pla13] PlanetMath. Ackermann function is not primitive recursive. <https://planetmath.org/ackermannfunctionisnotprimitiverecursive>, 2013.
- [Prö13] Hans Jürgen Prömel. Rapidly growing Ramsey functions. In *Ramsey Theory for Discrete Structures*, pages 97–103. Springer, Cham, 2013. [https://link.springer.com/chapter/10.1007/978-3-319-01315-2\\_8](https://link.springer.com/chapter/10.1007/978-3-319-01315-2_8).
- [Rey83] John C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523, Amsterdam, 1983. Elsevier.
- [Rey93] John C. Reynolds. The discoveries of continuations. *LISP and Symbolic Computation*, 6:233–247, 1993. <https://link.springer.com/content/pdf/10.1007/BF01019459.pdf>.
- [RPY<sup>+</sup>21] Talia Ringer, RanDair Porter, Nathaniel Yazdani, John Leo, and Dan Grossman. *Proof Repair across Type Equivalences*, page 112–127. Association for Computing Machinery, New York, NY, USA, 2021.
- [Sch] Daniel Schepler. Zorn’s lemma (the Topology project). <https://github.com/coq-community/topology>.
- [Sch77] Kurt Schütte. *Proof Theory*. Springer, 1977. <https://link.springer.com/book/10.1007%2F978-3-642-66473-1>.
- [Ser14] Ilya Sergey. Programs and Proofs: Mechanizing Mathematics with Dependent Types, 2014. <https://doi.org/10.5281/zenodo.4996238>.
- [Sim04a] Carlos Simpson. Category theory in ZFC. User Contributions to the Coq Proof Assistant, 2004. <https://github.com/coq-contribs/cats-in-zfc>.
- [Sim04b] Carlos Simpson. Set-theoretical mathematics in Coq, 2004. <https://arxiv.org/abs/math/0402336>.
- [Sla07] Will Sladek. The Termite and the Tower: Goodstein sequences and provability in PA. <https://www.uio.no/studier/emner/matnat/ifi/INF5170/v08/undervisningsmateriale/sladekgoodstein.pdf>, 2007.
- [SM19] Matthieu Sozeau and Cyprien Mangin. Equations reloaded: High-level dependently-typed functional programming and proving in Coq. *Proceedings of the ACM on Programming Languages*, 3(ICFP), July 2019. <https://hal.inria.fr/hal-01671777>.
- [Smu92] R.M. Smullyan. *Gödel’s Incompleteness Theorems*. Logic Guides Series. Oxford University Press, 1992.

- [SO08] Matthieu Sozeau and Nicolas Oury. First-class type classes. In *International Conference on Theorem Proving in Higher Order Logics*, pages 278–293, Berlin, Heidelberg, 2008. Springer. [https://sozeau.gitlabpages.inria.fr/www/research/publications/First-Class\\_Type\\_Classes.pdf](https://sozeau.gitlabpages.inria.fr/www/research/publications/First-Class_Type_Classes.pdf).
- [Str00] Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1-2):11–49, April 2000. <https://www.cs.cmu.edu/~crary/819-f09/Strachey67.pdf>.
- [SvdW11] Bas Spitters and Eelis van der Weegen. Type classes for mathematics in type theory. *Mathematical Structures in Computer Science*, 21(4):795–825, 2011. <https://arxiv.org/abs/1102.1323>.
- [Sza93] Nora Szasz. A machine checked proof that ackermann’s function is not primitive recursive. In *Papers Presented at the Second Annual Workshop on Logical Environments*, page 317–338, USA, 1993. Cambridge University Press.
- [Tel00] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, Cambridge, United Kingdom, second edition, 2000.
- [Wad89] Philip Wadler. Theorems for free! In *International Conference on Functional Programming Languages and Computer Architecture*, pages 347–359, New York, NY, USA, 1989. ACM. <https://homepages.inf.ed.ac.uk/wadler/papers/free/free.ps>.
- [Wai70] Stan Wainer. A classification of the ordinal recursive functions. *Archiv für mathematische Logik und Grundlagenforschung*, 13(3):136–153, Dec 1970. <https://link.springer.com/article/10.1007%2FBF01973619>.
- [WB87] Stan Wainer and Wilfried Buchholz. Provably computable functions and the fast growing hierarchy. In Stephen G. Simpson, editor, *Contemporary Mathematics*, volume 65, pages 179–198. American Mathematical Society, Providence, RI, USA, 1987. <http://nbn-resolving.de/urn/resolver.pl?urn=nbn:de:bvb:19-epub-3843-7>.



## Chapter 18

# Index and tables

**In progress** This index is currently under reorganization. We apologize for its incompleteness!

## Links to Gaia Library

Accessibility and paths inside  $\epsilon_0$ , 105,  
151

Canonical sequences, 104, 105, 149

Exponential of base  $\omega$ , 76

Finite ordinals, 76

Hessenberg sum, 97, 98, 147

Impossibility theorems, 111, 118

Introduction, 10, 19, 51, 75, 141

Limit and successor ordinals, 83

Ordinal notations, 91

Ordinal terms in Cantor normal form,  
80

Pretty printing Cantor normal forms,  
78

Rapidly growing functions, 138, 155

Strict order on ordinals below  $\epsilon_0$ , 79

Termination of all hydra battles, 99

The ordinal  $\omega$ , 76

Type of well formed ordinal terms  
below  $\epsilon_0$ , 82

## Coq, plug-ins and standard library

Commands

  Scheme, 222

Continuation Passing Style (CPS),  
  303, 330

Dependent pattern matching, 208

Dependent types, 204, 206

Dependently typed functions, 204,  
  223, 224, 337

Generalized rewriting, 293

Mutual induction, 225

Mutually inductive types, 29, 206

Parametric Higher-Order Abstract  
  Syntax (PHOAS), 303

Parametricity, 313

Plug-ins

  Equations, 62, 127, 129, 136

  Gaia, 141

  Paramcoq, 313

Proofs by reflection, 308

Type classes, 290, 293, 298, 325

  Equivalence relations, 293

  Operational type classes, 288

  Proper class, 298, 325

Unicity of equality proofs, 82

## Mathematical notions and algorithmics

- Abstract properties of arithmetic functions, 134
- Ackermann function, 62, 220
- Addition chains, 302
- Additive principal ordinals, 76
  
- Cantor normal form, 73
  
- Euclidean addition chains, 321
  
- Fibonacci numbers
  - Matrix exponentiation, 285
  
- Notations
  - Interval, 47
  
- Ordinal numbers, 51
  - Accessibility inside  $\epsilon_0$ , 105
  - Additive principal ordinals, 174
  - Canonical sequences, 102
  - Cantor normal form, 180
  - Critical ordinals, 179
  - Ketonen-Solovay machinery, 101
  - Large sets, 119
  - Minimal large sets, 119
  - Ordering functions, 170
  
- Primitive recursive functions, 203
  
- Rapidly growing functions, 136
  - Hardy Hierarchy, 129
  - Wainer Hierarchy, 136
  
- Transfinite induction, 87, 89, 97, 102, 104, 107, 109, 112, 113, 115, 122, 129, 134, 180

## Library hydras: Ordinals and hydra battles

- Abstract properties of arithmetic functions, 134
- Exercises, 27, 30, 33, 34, 45, 47, 57, 65, 69, 70, 84, 86, 87, 95, 105–107, 113, 129, 138, 148, 179, 181, 202, 203, 220, 272, 273
- Library Epsilon0
  - Functions
    - canon, 103
    - canonS, 103
    - F\_ (Wainer hierarchy), 136
    - H\_ (Hardy hierarchy (variant)), 129
    - L\_ (final step of a minimal path), 127
    - pp (pretty printing terms in Cantor normal form), 77
    - succ, 84
  - Notations
    - phi0 (exponential of base omega), 76
  - Predicates
    - mlarge (minimal large sequences), 120
    - path\_to, 106
  - Types
    - E0, 81
    - ppT1, 77
    - T1, 74
- Library Gamma0
  - Types
    - T2, 186
- Library Hydra
  - Predicates
    - round, 33
    - round\_n, 33
    - Termination, 44
  - Type classes
    - Battle, 34
    - Hvariant, 44
  - Types
    - Hydra, 25
    - Hydrae, 25
- Library OrdinalNotations
  - Type classes
    - ON, 52
    - ON\_correct, 69
    - ON\_Iso, 70
    - SubON, 68
  - Library Prelude
    - iterate, 38, 136, 137, 220
  - Library Schutte
    - Constants
      - zero, 166
    - Functions
      - phi0, 174
      - plus, 172
      - succ, 167
    - Predicates
      - AP (additive principal ordinals), 174
      - Closed, 175
      - Cr (critical ordinals), 179
      - is\_cnf\_of (to be a Cantor normal form of), 180
      - ordering\_function, 170
    - Type classes
      - WO (well order), 162
    - Types
      - Ord, 162
- Projects, 20, 25, 33, 69–71, 78, 91, 97, 98, 118, 181, 183, 185, 186, 188, 192, 193

## Library `hydras.Ackermann`: Primitive recursive functions, Gödel encoding

Ackermann function, 220

Exercises, 215–218, 264

Functions

`evalPrimRec`, 208

`evalPrimRecs`, 208

Predicates

`extEqual`, 204

`isPR`, 211

Projects, 235, 258

Types

`naryFunc`, 203

`PrimRec`, 206

`PrimRecs`, 206

## Library additions: Addition chains

Exercises, 285, 320

Projects, 306, 343

Type classes

EMonoid, 293

Monoid, 290

Types

chain (addition chains), 304

computation, 304