Surreal numbers in Coq

Mamane, L.E.

Award date:
2003

TECHNISCHE UNIVERSITEIT EINDHOVEN

Department of Mathematics and Computing Science

MASTER'S THESIS

Surreal Numbers in Coq

by
L.E. Mamane

Supervisor: dr. R.P. Nederpelt

Eindhoven, July 2003

Contents

Chapter 0

Introduction

0.0 Introduction

Preface

A Luxembourg citizen, I started my higher education in France, where I eventually entered the Computing and Modelling Science curriculum[0] of the "École Normale Supérieure de Lyon", one of France's top institutes for research and higher education. The education part focuses on producing future researchers, and thus tightly integrates the research and education parts. The diploma, called "magistère" is supposed to be ε more than a Master's degree: the students get both the Master's Degree (DEA) and the magistère, which is meant to be a mark of excellence, a token that the student was exposed to the research world during his curriculum, and has done more course work and research internships or scientific colloquiums than required for a Master's degree.

I spent one year as an exchange student at the Technische Universiteit Eindhoven, and decided I wanted to stay there to pass my Master's Degree. I'm still in the ENS Lyon's magistère curriculum.

The Subject

The subject of my Master's Thesis, of which you are reading the report, is "Surreal Numbers in Coq". More precisely, the object is to try to fully formalise a good chunk of the theory of Surreal Numbers, by staying as close as possible to the original (Conway's approach in [Con01]). The hidden agenda behind this is that in doing this, I would get a better idea of the current status of theorem provers technology, and hopefully identify areas where it can be improved, maybe even develop ideas for doing so. Surreal Numbers looked appropriate for this, because it is a profoundly set-theoretic piece of mathematics, thus a priori another approach than the type theory used in Coq. See section 4.0 for details.

I owe my discovery of surreal numbers, and the idea to formalise them in a theorem prover, to Freek Wiedijk.

[0]The exact official name is "Magistère d'Informatique et Modélisation". I'm translating "Informatique" (Informatica in Dutch) by "Computing Science". The reader may substitute "Computer Science" or "Computing and Computer Science" if he is so minded.

0.1 Introduction to Coq

Coq is a theorem prover, developed in the LogiCal project, a joint project of the INRIA (the French Institut Nationale pour la Recherche en Informatique et Automatique, i.e. National Institute for Research in Computing Science and Control, *the* French institute for research in Computing Science), the CNRS (Centre National de la Recherche Scientifique, i.e. National Center for Scientific Research, *the* French Institute for research in science) and the LRI laboratory (Laboratoire de Recherche en Informatique, i.e. Laboratory for Research in Computing Science, a joint laboratory by the University of Paris-South 11 and the CNRS, in Orsay).

It is based on type theory (typed lambda calculus). More precisely, the theory behind it is called Calculus of Inductive Constructions (CIC). For people familiar with the Barendregt cube, one can summarise the CIC as λ_C plus:

- An infinite sort hierarchy $\Box_i, i \in \mathbb{N}$. $\Box$ is called `Type` in Coq, and the subscript is hidden from the user.
- Inductive and Coinductive Definitions.

0.1.0 A very fast overview of type theory

I will not explain theory of typed λ-calculus here, but I will try to give you a rough idea of how it goes.

The central idea of type theory applied to theorem provers is that mathematical propositions are represented as types. An object (a term) of this type is a proof of the proposition. Types have a type, too, and to avoid too much confusion, types of types are called sorts.

Let's take the example of implication. If P and Q are propositions (propositional types), then $P \to Q$ is a (propositional) type. An inhabitant of this type (a term that is of this type), a proof of $P \to Q$ thus, is a function that takes a proof of P and returns a proof of Q. Now, what if P and Q are mathematical structures, like $\mathbb{N}$ and $\mathbb{R}$? Then $P \to Q$ is the type of the (total) functions of domain P and range Q. An inhabitant of this type is any function from $\mathbb{N}$ to $\mathbb{R}$. Implication and "type of function" are essentially the same thing.

Mathematical propositions are of sort $*_p$ (`Prop` in Coq) and "simple" mathematical structures are of sort $*_s$ (`Set` in Coq). For example, "$\forall n \in \mathbb{N}, \exists p \in N, n = 2 * p$" would be of sort `Prop`, but $\mathbb{N}$ is of sort `Set`. The central difference between `Prop` and `Set` is impredicativity: the consequence of a proposition P can not depend on the particular proof used to prove P, but the result of a function can depend on its input! Essentially, when using a proof, one must forget how exactly it was done, while one can use the structure of a non-proof object.

The sorts $*_p$ and $*_s$ are themselves of type $\Box_0$, and being of type $\Box_i$ implies being of type $\Box_j$ for $j \geq i$. I'm not explaining here what could force a mathematical structure to be of type $\Box_i$, and not $*_s$, it is beyond the scope of this introduction. To give you a very rough idea, things like $\mathbb{N}, \mathbb{Z}, \mathbb{N}^{\mathbb{Z}}, \mathbb{N} \times \mathbb{Z}, \ldots$ are of sort $*_s$, but if you start modelling Zermelo-Fraenkel set theory in Coq, your objects will lie in the $\Box_i$ hierarchy.

0.1.1 Coq syntax

Let's list the major syntactic constructions of Coq, so that you can read fragments of Coq code.

- . is the "end of phrase" marker.
- `Qed` is the command that, once a proof is finished, saves the lemma or theorem just proven for later use.
- Implication is `->`, and is primitive
- Falsity (i.e. $\bot$) is `False`
- Negation is `~` or `not`, and `~P` is by definition `P -> False`
- Inductive definitions are given by listing the *constructors* of the type, i.e. how a value of this type can be constructed.

  For example, $\mathbb{N}$ has two constructors:

  - 0, which doesn't take any argument. Saying that 0 is a constructor with no argument means "0 is a natural number".
  - s, the successor function, that takes exactly one natural number argument, i.e. for any natural number n, $s(n)$ is a different, new natural number.

  In Coq syntax:

```
Inductive nat : Set :=
| O : Nat
| S : Nat -> Nat.
```

  Another example, binary trees labelled at the nodes by natural numbers:

```
Inductive btree : Set :=
| Leaf : btree
| Node : nat -> btree -> btree -> btree.
```

- Universal quantification is primitive[1], $\forall p \in P, Q$ is `(p:P)Q`.
- $\lambda x : P.Q$ (the function that takes an x of type P and returns Q) is `[x:P]Q`.
- `[x:=P]Q` is Q with all free occurrences of x replaced by P.

[1] In fact, behind the curtains it is the same thing as implication.

0.1.2 Coq proofs

Proofs are constructed by backwards reasoning[2], by applying *tactics* to the goal. During the construction of a proof, the screen is separated into two areas, separated by a line of ='s:

1. In the lower area, the stack of goals.

   Initially, it contains only one element, the theorem one has set to prove.

2. In the upper area, the hypotheses under which the current goal (the top of the stack) is to be proven.

   Initially, this is empty.

   Together with all previously proven theorems and assumed axioms, the hypotheses form the *environment* or *context.*

A few examples of tactics:

- The `Intro` tactic applies to a goal of the form `(a:A)G`, adds `a:A` to the environment and leaves `G` as a goal. `Intros` is the plural of `Intro`. Because implication is (in type theory) a particular case of universal quantification, this applies to implications, too.

  Basically this tactic implements the principle "To prove $\forall a \in A, G$, it suffices to prove that under the hypothesis $a \in A$, G holds".

- The `Apply` tactic takes an already proven theorem (or an axiom), and tries to use this theorem on the goal.

- The `Inversion` tactic takes a hypothesis, generates all the necessary conditions that should hold for the hypothesis to hold, and adds these as hypotheses.

  For example, if "n is even" by the existence of the half of n ($even(n) \overset{\text{def}}{\leftrightarrow} \exists p \in \mathbb{N}, n = p + p$), then `Inversion` applied to the hypothesis "n is even" generates the two hypotheses $p \in \mathbb{N}$ and[3] $n = p + p$.

[2]This means that the goal (the proposition to prove) is reworked until it matches a hypothesis (or previously proven theorem). This in contrast with forward reasoning, where the hypotheses are reworked and combined until the goal is generated.

[3]Obviously, if the name p is already taken in the current context, a fresh name is picked

Chapter 1

Surreal Numbers

1.0 Introduction to Surreal Numbers

In this section, we try to give a gentle, intuitive introduction to the notion of surreal numbers and the main operations over it. A formal, "dry" presentation will be given in section 1.1.

1.0.0 Features

The notion of surreal numbers (sometimes called Conway Numbers), described by John Horton Conway in [Con01] and subsequently main hero of the novellette [Knu74] by Donald E. Knuth, is a notion of numbers that has all of the following nice properties:

- It covers the real numbers, and also all ordinals.
- In some sense, the surreal numbers fill the "holes" between the ordinals, like $\mathbb{R}$ fills the holes between the natural numbers.
- The surreal numbers fill the "holes" between the reals, a bit like $\mathbb{R}$ fills the holes between the rationals.

  For example, there are surreal numbers smaller than any strictly positive real number, yet strictly positive.
- The class of the surreal numbers is equipped with a *totally ordered field* structure.

Remark 1.0.1 *Already at this point, there is an implacable corollary we see: because the surreal numbers include all ordinals, the collection of all surreal numbers is not a set.*

Even better, the definition of the class of surreal numbers is achieved with only one set of inductive definitions which come out of the blue. This in contrast to the usual way of constructing $\mathbb{R}$:

- Construct $\mathbb{N}$, equip it with a monoid structure. One set of definitions (what is a natural number, define addition and multiplication of naturals).

- Construct $\mathbb{Z}$ on top of $\mathbb{N}$, equip it with a ring structure[0]. Another set of definitions (what is an integer, define addition and multiplication of integers).
- Construct $\mathbb{Q}$ on top of $\mathbb{Z}$, equip it with a field structure. Another set of definitions.
- Construct $\mathbb{R}$ on top of $\mathbb{Q}$, equip it with a field structure. Another set of definitions.

With the surreal numbers, one defines what a surreal number is, equips the class of surreal numbers with an addition and a multiplication, and one has everything: reals, ordinals, and beyond (hence the name *sur*real numbers).

It is interesting to note that the unpracticality of this tower of definitions has already been identified as an issue for effective[1] computer formalisations of the real numbers, like in [Udd80]. See also appendix A.

1.0.1 Definition of Surreal Numbers

The surreal numbers are defined inductively:

Definition 1.1 *A surreal number x is a pair of arbitrary sets of surreal numbers L_x and R_x (left (resp. right) of x), where no element of the left is greater than or equal to a right element, i.e.*

$$\forall l \in L_x, \neg\exists r \in R_x \text{ s.t. } l \geq r$$

The definition of $\geq$ is given in definition 1.6.

Definition 1.2 *Sometimes, "a left" (respectively right) will be used for "an element of the left" (respectively right).*

Notation 1.1 *This surreal number x is denoted $\{L_x|R_x\}$.*

Interpretation $x = \{L_x|R_x\}$ is to be interpreted as the simplest[2] surreal number that is strictly greater than all the left elements, yet strictly smaller than all the right elements:

$$\underline{\qquad L_x \qquad} \; x \; \underline{\qquad R_x \qquad}$$

This interpretation will be referred to as "the interpretation" throughout this document. The idea is similar to Dedekind cuts.

[0] Some construct $\mathbb{R}$ this way: $\mathbb{N} \to \mathbb{Q}^+ \to \mathbb{Q} \to \mathbb{R}$, rather than $\mathbb{N} \to \mathbb{Z} \to \mathbb{Q} \to \mathbb{R}$. My point is still valid: they still need four different structures, four different additions, four different multiplications, ...

[1] I mean formalisations that construct, respectively define objects that represent real numbers, in contrast with the approach of defining $\mathbb{R}$ without constructing reals, by axiomatisation, like the FTA project of the University of Nijmegen did.

[2] The exact meaning of this "simplest" is explained in section1.0.4, page 11

If we take this interpretation for granted, it is fairly simple to see we need the "no left greater than or equal to a right" condition: as x is supposed to be strictly between its lefts and rights, there better be room between the lefts and rights, which is not the case if a left is bigger than a right[3].

$$\frac{L_x}{x \qquad R_x}$$

Notation 1.2 *The class of all surreal numbers is called No.*

Notation 1.3 *If $x = \{L_x|R_x\}$ is a surreal number, we'll denote an arbitrary element of L_x (resp. R_x) by x_l (resp. x_r).*

So, for example "$\forall x_l$" will be a shortcut for "x is $\{L_x|R_x\}$, and $\forall x_l \in L_x$".

Notation 1.4 *In order to lighten notations,* e.g. $\{\{a,b,c,d\}\,|R_x \cup R_y\}$ *will be written as*

$$\{a,b,c,d|x_r,y_r\}$$

.

1.0.2 Examples

0 The only number we can construct without using any other number is $\{\emptyset|\emptyset\}$, i.e. an empty left and an empty right. We'll call this number 0 (zero), and according to the interpretation, it is the simplest number of all.

1 Now, we can construct:

- $\{\{0\}\,|\emptyset\}$, the simplest number greater than 0. Let's call it 1.
- $\{\emptyset|\,\{0\}\}$, the simplest number smaller than 0. Let's call it -1.
- $\{\{0\}\,|\,\{0\}\}$. This one doesn't fulfil the condition: a left (0) is greater than or equal to a right (0).

2 With the numbers we have just constructed, we now have:

- $\{\{0\}\,|\,\{1\}\}$, the simplest number between 0 and 1. That's $\frac{1}{2}$.
- $\{\{1\}\,|\emptyset\}$, the simplest number greater than 1. That's 2.
- $\{\{0,1\}\,|\emptyset\}$, the simplest number greater than 0 and 1. But being greater than 0 and greater than 1 is equivalent to being greater than 1. Thus, according to the interpretation, this number must be the same as $\{\{1\}\,|\emptyset\}$, i.e. 2.
- The same holds for $\{\emptyset|\,\{0,-1\}\}$ and $\{-1\}$, which are -2.

[3]Remember we aim for a *totally* ordered structure

1.0.3 Representatives and Numbers

We just made an important discovery: for the interpretation to hold, different constructions must stand for the same number, in the same way as in the classical construction of rationals as pairs of integers, $\frac{1}{2}$, $\frac{2}{4}$ and $\frac{15}{30}$ stand for the same rational. There is thus a difference between the particular *representative* we use and the *surreal number* it stands for.

Definition 1.3 (equal, identical) *Two representatives x and y are said to be* equal *if they stand for the same surreal number according to the interpretation, and this is written $x = y$.*

They are said to be identical *if they are the same as pair of sets, and this is written $x \equiv y$.*

Equality is not a primitive, but a *defined* relation. Identicality can also be expressed as "the smallest reflexive relation" or "the smallest equivalence relation".

For readers of [Con01], note that Conway writes "number" for "representative", and thus completely avoids the issue of "what exactly is a surreal number": it would be something like the equivalence class for equality of representatives. But to be able to speak about "equivalence classes", we need an equivalence relation, and a relation must operate on a set. As we saw surreal numbers don't (and can not, if we want them to fulfil the "features" bill of section 1.0.0) constitute a set. On the other hand, Conway's vocabulary hides compatibility problems: a function on surreal numbers is in fact defined on representatives, but is it well-defined on surreal numbers?

Following his lead, I'll merrily forget the difference between representatives and surreal numbers, but I'll keep the possibility to say "representative" for statements like "if x_1 and x_2 are representatives of the same surreal number, then $x_1 + y$ and $x_2 + y$ are representatives of the same surreal number".

Another distinction I will not be keeping is the distinction between the "usual" $\mathbb{N}, \mathbb{R}, \mathbb{Q}, \mathbb{Z}$, $On, \ldots$ (On is the class of all ordinals) and the corresponding subclasses of *No*. This is common in mathematics: structures are defined modulo isomorphism. The corresponding subclasses of *No* are isomorphic to $\mathbb{N}, \mathbb{R}, On, \ldots$; they are as good as any candidates for the title of "usual" $\mathbb{N}, \mathbb{R}, On, \ldots$

1.0.4 Birth date

It is now possible to explain the notion of "simplest" used in the interpretation in more detail.

The surreal number representative $\{\emptyset|\emptyset\}$ is constructed without referring to any other surreal number, and it is the only one constructed in this way. In some sense, it is the first to be born, on "day" zero (zero is here the ordinal zero): its birth date is zero, the smallest (first) ordinal.

$\{\{\{\emptyset|\emptyset\}\}|\emptyset\}$ (which by abuse of notation[4] I'll write $\{\{0\}|\emptyset\}$) is constructed using only representatives of birth date zero. It is thus born on the "next" day, day one.

[4] Abuse of notation because 0 is a surreal number, and not a representative. Generally, when putting numbers in a $\{|\}$ construction, the "most natural" representative is meant.

$\{\{\{\emptyset|\emptyset\}\}\,|\,\{\{\{\{\emptyset|\emptyset\}\}|\emptyset\}\}\}$ (which by abuse of notation I'll write $\{\{0\}\,|\,\{1\}\}$) is constructed using representatives of birth date zero and one. It is thus born on day two.

$\{\{0,1\}\,|\emptyset\}$ is born on day two, too: it is constructed from the same representatives as $\{\{0\}\,|\,\{1\}\}$. Other representatives born on day two: $\{\emptyset|\,\{-1\}\}$, $\{\emptyset|\,\{0,1\}\}$, $\{\{1\}\,|\emptyset\}$.

Definition 1.4 (birth date) *The birth date $\rho(x)$ of a representative x is the smallest ordinal strictly greater than the birth dates of its lefts and rights, i.e. the supremum of the successors of the birth dates of its lefts and rights:*

$$\rho\left(\{L|R\}\right) \stackrel{def}{=} \sup_{i \in L \cup R} \left(\rho(i)+1\right)$$

Remark 1.0.2 *The birth date establishes a pre-order on representatives (x less than y if and only if $\rho(x) \leq \rho(y)$), which can be restricted to a partial order that is well-founded ($x \preceq y \leftrightarrow \rho(x) < \rho(y) \vee x \equiv y$), and this order will generally be used to show well-foundedness of inductions and recursions.*

Definition 1.5 *The birth date of a surreal number is the smallest of the birth dates of its representatives.*

Interpretation A more precise expression of the interpretation is: the representative $\{L|R\}$ stands for the earliest born surreal number among the surreal numbers that are strictly greater than the elements of L, yet strictly smaller than the elements of R.

So, for example, $\{\emptyset|\,\{2\}\}$ stands for 0: 0 is smaller than two, and its $\{\emptyset|\emptyset\}$ representative is the earliest born of all.

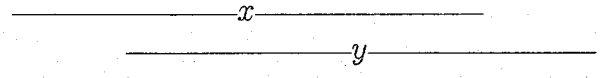
1.0.5 Order

I haven't yet defined the symbol $\leq$, although it is used in the very definition of a surreal number.

Definition 1.6 *$x \leq y$ if and only if no left of x is greater than or equal to y and no right of y is less than or equal to x, i.e.*

$$x \leq y \stackrel{def}{\leftrightarrow} (\forall x_l, \neg(x_l \geq y)) \wedge (\forall y_r, \neg(x \geq y_r))$$

in a picture:



As usual, $x \geq y \stackrel{\text{def}}{\leftrightarrow} y \leq x$, $x < y \stackrel{\text{def}}{\leftrightarrow} (x \leq y \wedge x \neq y)$ (note this is $\neq$, *not* $\not\equiv$) and $x > y \stackrel{\text{def}}{\leftrightarrow} y < x$.

This definition is consistent with the interpretation: let's suppose there is a left x_l of x that is greater than or equal to y, i.e. $x_l \geq y$. Then, as the interpretation says that $x_l < x$,

by transitivity of $\leq$[5], we have $y < x$, thus certainly not $x \leq y$. Having $(\forall x_l, \neg(x_l \geq y))$ as a necessary condition for $x \leq y$ is thus consistent with the interpretation. Similarly for $\neg(x \geq y_r)$.

Example

$0 \leq 1$

$$\begin{array}{ll} & \{\emptyset|\emptyset\} \leq \{\{\{\emptyset|\emptyset\}\}|\emptyset\} \\ \leftrightarrow & \quad \{\text{Def. } \leq\} \\ & (\forall 0_l \in \emptyset, \neg(0_l \geq 1)) \wedge (\forall 1_r \in \emptyset, \neg(0 \geq 1_r)) \\ \leftrightarrow & \quad \{\text{Universal quantification over empty domain}\} \\ & \top \end{array}$$

1.0.6 Equality

$\leq$ is a pre-order on representatives. Equality $(=)$ is defined to be the equivalence relation given by the pre-order:

$$x = y \overset{\text{def}}{\leftrightarrow} x \leq y \wedge y \leq x$$

1.0.7 Addition

We eventually want a field structure over the surreal numbers, we thus need an addition:

Definition 1.7 (addition)

$$x + y \overset{def}{=} \{x_l + y, x + y_l | x_r + y, x + y_r\}$$

Again, this definition is consistent with the interpretation: we want an ordered field structure, thus in particular compatibility of the order and addition, i.e.

$$\forall x, \forall z, \forall y, x \leq z \rightarrow x + y \leq z + y$$

In this definition, by putting $x_l + y$ in the left of $x + y$, by the interpretation, we force $x_l + y$ to be smaller than $x + y$, which is consistent with $x_l < x$ (instantiate z by x and x by x_l in the above formula and replace $\leq$ by $<$). The same holds for $x + y_l$, and we force $x_r + y > x + y$ and $x + y_r > x + y$.

Addition is commutative. This will be formally proved later, but follows easily from the definition.

[5] $\leq$ being an order, we want it to be transitive

Examples

Let's check that the names given to the numbers are consistent with what we expect from their properties:

$0 + x$

$$
\begin{array}{ll}
 & 0 + x \\
\equiv & \quad \{\text{Def. of } +\} \\
 & \{0_l + x, 0 + x_l | 0_r + x, 0 + x_r\} \\
\equiv & \quad \{\text{Induction: } 0 + x_l \equiv x_l \wedge 0 + x_r \equiv x_r\} \\
 & \{0_l + x, x_l | 0_r + x, x_r\} \\
\equiv & \quad \{0 \text{ has no left and no right}\} \\
 & \{x_l | x_r\} \\
\equiv & \\
 & x
\end{array}
$$

$1 + 1$

$$
\begin{array}{ll}
 & 1 + 1 \\
\equiv & \quad \{\text{Def. of 1 and } +\} \\
 & \{0 + 1, 1 + 0 | 1_r + 1, 1 + 1_r\} \\
\equiv & \quad \{1 \text{ has no right}\} \\
 & \{0 + 1, 1 + 0 | \} \\
\equiv & \quad \{0 + x \equiv x + 0 \equiv x\} \\
 & \{1 | \} \\
\equiv & \quad \{\text{Def. of 2}\} \\
 & 2
\end{array}
$$

$\frac{1}{2} + \frac{1}{2}$

$$
\begin{array}{ll}
 & \frac{1}{2} + \frac{1}{2} \\
\equiv & \quad \{\text{Def. of } \tfrac{1}{2} \text{ and } +\} \\
 & \{0 + \frac{1}{2}, \frac{1}{2} + 0 | 1 + \frac{1}{2}, \frac{1}{2} + 1\} \\
\equiv & \quad \{0 + x \equiv x, x + y \equiv y + x\} \\
 & \{\frac{1}{2} | 1 + \frac{1}{2}\} \\
\equiv & \quad \{\text{Def. of } \tfrac{1}{2}, 1 \text{ and } +\} \\
 & \{\frac{1}{2} | \{0 + \frac{1}{2}, 1 + 0 | 1_r + \frac{1}{2}, 1 + 1\}\} \\
\equiv & \quad \{1 \text{ has no right}\} \\
 & \{\frac{1}{2} | \{0 + \frac{1}{2}, 1 + 0 | 1 + 1\}\} \\
\equiv & \quad \{0 + x \equiv x + 0 \equiv x\} \\
 & \{\frac{1}{2} | \{\frac{1}{2}, 1 | 2\}\}
\end{array}
$$

So, $\frac{1}{2} + \frac{1}{2}$ is the simplest number between $\frac{1}{2}$ and $\{\frac{1}{2}, 1|2\}$, and $\{\frac{1}{2}, 1|2\}$ itself is between 1 and 2.

Let's enumerate the representatives by order of birth date:

0 0, the only surreal number with a representative of birth date 0 is not greater than $\frac{1}{2}$, and thus $\frac{1}{2} + \frac{1}{2}$ does not stand for 0.

1 There are only two representatives of birth date 1: $\{0|\}$, standing for 1, and $\{|0\}$, standing for -1.

- -1 is not greater than $\frac{1}{2}$, and thus $\frac{1}{2}+\frac{1}{2}$ does not stand for -1.
- 1 has the representative $\{0|\}$. One is (strictly) greater than $\frac{1}{2}$ (because 1 is a right of $\frac{1}{2}$), and strictly smaller than $\left\{\frac{1}{2}, 1|2\right\}$ (because 1 is a left of it). There is no other surreal number having a representative with birth date at most 1, thus $\left\{\frac{1}{2}|\left\{\frac{1}{2}, 1|2\right\}\right\}$ stands for 1.

So, $\frac{1}{2}+\frac{1}{2}=1$.

Note: this is not a formal proof, because I use the interpretation to get the result. To really formally check it, check that $\frac{1}{2}+\frac{1}{2}\leq 1$ and $1\leq\frac{1}{2}+\frac{1}{2}$ from the definition of $\leq$.

1.0.8 More examples

Let's see how to construct well-known subclasses of *No*

- For n an ordinal (and particularly a natural number), $\{n|\}$ is the successor of n, i.e. $n+1$.

  We have 0 to "seed" the construction, it is thus easy to see we have a copy of $\mathbb{N}$ in *No*.

- For $(n_i)_{i\in I}$ a family of ordinals, $\{n_i|\}$ is the limit ordinal of this family.

  We thus have a copy of all ordinals in *No*. The birth date of an ordinal is itself.

- We already constructed $\frac{1}{2}$, at birth date 2. Thus, at birth date 3, we get $\left\{0|\frac{1}{2}\right\}$, a representative of $\frac{1}{4}$, and $\left\{\frac{1}{2}|1\right\}$, a representative of $\frac{3}{4}$. You can check that $\frac{1}{4}+\frac{1}{4}=\frac{1}{2}$ and $\frac{3}{4}+\frac{1}{4}=1$.

  At birth date 4, we get (among others) $\left\{\frac{1}{4}|\frac{1}{2}\right\}$, a representative for $\frac{3}{8}$.

  In this manner, at natural number birth dates, we get all dyadic numbers between 0 and 1, i.e. numbers that admit a finite development in base 2, i.e. numbers of the form $\sum_{i\in\mathbb{N}^*} s_i\cdot 2^{-i}$, where $s_i\in\{0,1\}$ and the family $(s_i)_{i\in\mathbb{N}^*}$ is constant (either zero or one) except for a finite number of i's. $\mathbb{N}^*$ is my notation for $\mathbb{N}-\{0\}$, some call it $\mathbb{N}^0$, $\mathbb{N}_0$ or $\mathbb{N}_*$

  To generate the other dyadic numbers, add an integer number (element of $\mathbb{Z}$) to a dyadic number in $[0,1]$. We thus get all dyadic numbers (still at natural numbers birth dates), and because dyadic numbers are dense in $\mathbb{R}$, we get (at birth date ω)[6] a copy of $\mathbb{R}$ in *No*: to obtain a representative of a real number x, let L be a set of (representatives of) dyadic numbers smaller than x, such that one can get arbitrarily close to x (i.e. $\forall l\in L, l<x$ and $\forall\varepsilon\in\mathbb{R},\varepsilon>0\rightarrow\exists l\in L, 0<x-l<\varepsilon$) and R a set of dyadic numbers greater than x, such that one can get arbitrarily close to x (i.e. $\forall r\in R, r>x$ and $\forall\varepsilon\in\mathbb{R},\varepsilon>0\rightarrow\exists l\in L, 0<r-x<\varepsilon$). Then, $\{L|R\}$ is a surreal number representative for the surreal number x.

[6] $\omega=\sup\mathbb{N}$

Here is the definition of a function that will construct a representative of minimal birth date (at most ω, thus) of any real number between 0 and 1, by calculating its base 2 developments and putting the successive roundings/truncations of the base 2 development in the left or right of the representative it constructs:

```
let surreal_from_real = function
 | 0 => {|}
 | 1 => {0|}
 | x => (* Comment: x is neither 0, neither 1 *)
   let rec do_it lower_r lower_sr upper_r upper_sr {L|R} =
     let middle_r = ( / (+ lower_r upper_r) 2 ) and
         middle_sr = {lower_sr|upper_sr} in
     if x > middle then
       do_it middle_r middle_sr upper_r upper_sr
             {(add_to_set middle_sr L)|R}
     else if x < middle then
       do_it lower_r lower_sr middle_r middle_sr
             {L|(add_to_set middle_sr R)}
     else (* Comment: x = middle *)
       {L|R}
   in
   do_it 0 {|} 1 {0|} {|}
```

- Because we have every ordinal in *No*, we also have huge surreal numbers, like $\aleph_{3000}$. Because I told you they form a field structure, you expect very tiny surreal numbers, too. After all, if x is huge (but positive), $\frac{1}{x}$ should be tiny (but positive). Let's look at some of them.

  The first example is $\left\{\{0\} \mid \left\{\left(\frac{1}{2}\right)^n, n \in \mathbb{N}\right\}\right\}$. It is smaller than any strictly positive real, yet strictly positive. We'll call it ε, and qualify it by *first level infinitesimal.* Its birth date is ω (you might have noticed I was careful to use only dyadic reals to construct it). ε is $\frac{1}{\omega}$.

  We can then construct any polynomial in ε (with the multiplication I haven't shown you yet). But it goes beyond: what about $\{\{0\} \mid \{\varepsilon^n, n \in \mathbb{N}\}\}$? It is smaller than any positive polynomial of ε, yet positive. It is the *second level infinitesimal*, $\frac{1}{\aleph_1}$. Etc.

1.0.9 Multiplication

We eventually want a field structure over *No*, so we need a multiplication:

Definition 1.8 (Multiplication)

$$xy \stackrel{def}{=} \{x_l y + x y_l - x_l y_l, x_r y + x y_r - x_r y_r | x_l y + x y_r - x_l y_r, x_r y + x y_l - x_r y_l\}$$

This definition is consistent with the interpretation:

- For any surreal number χ, because $\chi_l < \chi$, we have $\chi - \chi_l > 0$. Because we eventually want an ordered field, we need to ensure that $(x - x_l)(y - y_l) > 0$ (compatibility of multiplication with the order). On the other hand:

$$\begin{array}{ll} & (x - x_l)(y - y_l) > 0 \\ \leftrightarrow & \quad \{\text{Distributivity}\} \\ & xy - x_l y - x y_l + x_l y_l > 0 \\ \leftrightarrow & \quad \{\text{Compatibility of } + \text{ with } >\} \\ & xy - x_l y - x y_l + x_l y_l + (x_l y + x y_l - x_l y_l) > 0 + (x_l y + x y_l - x_l y_l) \\ \leftrightarrow & \quad \{\text{calculus}\} \\ & xy > x_l y + x y_l - x_l y_l \end{array}$$

- Similarly, because $\chi_r > \chi$, $\chi - \chi_r < 0$, hence $(x - x_r)(y - y_r) > 0$, thus $xy > x_r y + x y_r - x_r y_r$
- On the other hand, $(x - x_l)(y - y_r) < 0$, the same calculation gives $xy < x_l y + x y_r - x_l y_r$. The same for $xy < x_r y + x y_l - x_r y_l$

1.1 Formal Definitions of Surreal Numbers

1.1.0 Surreal Numbers, order

Definition 1.9 (Surreal Number, order) *A surreal number $x = \{L_x | R_x\}$ is a pair of arbitrary sets of surreal numbers L_x and R_x, fulfilling the following condition:*

$$\forall l \in L_x, \neg\exists r \in R_x \text{ s.t. } l \geq r$$

where

$$\forall x, \forall y, x \leq y \overset{def}{\leftrightarrow} (\forall x_l, \neg(x_l \geq y)) \wedge (\forall y_r, \neg(x \geq y_r))$$

and

$$x \geq y \overset{def}{\leftrightarrow} y \leq x$$

Definition 1.10 (equal, identical)

$$x = y \overset{def}{\leftrightarrow} x \leq y \wedge y \leq x$$

$\equiv$ *is the smallest reflexive relation (Leibniz equality) on surreal numbers, and is called "identical".*

Definition 1.11 (birth date) *The birth date $\rho(x)$ of a surreal number x is the smallest ordinal strictly greater than the birth dates of its lefts and rights.*

$$\rho(\{L|R\}) \overset{def}{=} \min_{y \in L \cup R} \rho(y)$$

Definition 1.12 (earlier born)

$$x \preceq y \overset{def}{\leftrightarrow} \rho(x) < \rho(y) \vee x \equiv y$$

Definition 1.13 (strictly earlier born)

$$\begin{aligned} x \prec y &\overset{def}{\leftrightarrow} x \preceq y \wedge x \not\equiv y \\ &\leftrightarrow \rho(x) < \rho(y) \end{aligned}$$

Proposition 1.1.1 *$\prec$ is a well-founded strict order*

Definition 1.14 (structurally smaller) *$\lessdot$ is by definition the smallest relation on games such that:*

$$\forall L_x, \forall R_x, \forall x_l \in L_x, x_l \lessdot \{L_x | R_x\}$$
$$\forall L_x, \forall R_x, \forall x_r \in R_x, x_r \lessdot \{L_x | R_x\}$$
$\lessdot$ is transitive

In other words, it is the transitive closure of the relation "being a left or a right".

Proposition 1.1.2 *$\lessdot$ is a well-founded strict order.*

Proof: $x \lessdot y \rightarrow x \prec y$, and $\prec$ is well-founded.

1.1.1 Addition

Definition 1.15 (addition)

$$x + y \overset{def}{=} \{x_l + y, x + y_l | x_r + y, x + y_r\}$$

Definition 1.16 (Anti-number)

$$-x \overset{def}{=} \{-x_r | - x_l\}$$

1.1.2 Multiplication

Definition 1.17 (Multiplication)

$$xy \overset{def}{=} \{x_l y + x y_l - x_l y_l, x_r y + x y_r - x_r y_r | x_l y + x y_r - x_l y_r, x_r y + x y_l - x_r y_l\}$$

Having explained what surreal numbers are, we now turn to their formalisation in Coq.

Chapter 2

Surreal Numbers in Coq

In this chapter, we take a look at the model of surreal numbers I implemented in Coq, and discuss the difficulties in doing so. Some of these difficulties come from the type theory approach, others from Coq-the-implementation and others from unclarities in [Con01].

2.0 Games

2.0.0 The need for games

The definition of a surreal number (definition 1.9) makes use of the symbol "$\leq$" (the order on surreal numbers), and the definition of the latter depends on the structure of surreal numbers (the definition of any predicate or function depends on the structure of the class it is defined on). The question whether this mutual dependency is well-founded (in other words, whether this pair of intertwined definitions is valid, sound) is a real one.

The reader is probably familiar with such questions raised for functions defined by mutual recursion, and maybe for types (structures) defined by mutual induction. But here, the question is a "cross-border" one: the inductive type definition is mutually dependent on a recursive function definition. Hence the name, *inductive-recursive definition.*

The general question of validity of inductive-recursive definitions has been studied (see [Dyb00]), and the results implemented in proof checkers like Alf and its successor Alfa (see `http://www.math.chalmers.se/~hallgren/Alfa/`). But, for our purpose, the CIC doesn't allow inductive-recursive definitions. While [Con01] uses the inductive-recursive definition in its introduction, he doesn't dare let an inductive-recursive definition in the "precise treatment" (Preliminary Comments of Chapter 1 of [Con01]).

2.0.1 Definition of Games

Thus, a new structure is defined by omitting any reference to the order in the definition of a surreal number:

Definition 2.1 (Game) *A game g is a pair of arbitrary sets of games L_g and R_g.*

All definitions on surreal numbers (order, addition, ...) can then be applied to games by replacing "surreal number" by "game". Surreal numbers are defined as a subclass of the games:

Definition 2.2 (Surreal Number) *A surreal number is a game x fulfilling the following conditions:*

- *All lefts and rights of x are surreal numbers*
- *No left of x is greater than or equal to a right of x.*

2.0.2 Sets in Coq

The notion of game uses the notion of set. To define games, it is thus necessary to decide how sets will be modelled in type theory. There are two approaches to expressing a particular set:

1. As subset of some pre-existing set, which amounts to defining its characteristic function P:
$$X := \{x \in S, P(x)\}$$
2. Indexed by another set, which amounts to giving an index set I and a function f from the index set to the set one wishes to express:
$$X := \{f(x), x \in I\}$$

Intuitively, the first approach can not work for the very definition of games: this approach is a "decreasing" one. Only subsets of S can be given in this manner, P must be able to make a decision on any game (and in particular the very game it is used in), it thus cannot be used in the very construction of games. The second approach, on the other hand, works: f needs only to be able to construct structurally smaller games in order to construct new games.

More formally, some inductive definitions lead to contradictions. Any system must thus restrict what inductive definitions it accepts. The CIC does this by defining a notion of positivity, and requiring that every occurrence of a type G in the inductive definition of G be a positive occurrence. The first approach leads to a non-positive occurrence of the type `Game` in its definition. Our only option is thus the second approach.

2.0.3 Games in Coq

Games can be constructed only in one way, by giving two sets of games. A priori, the `Game` type will thus have one constructor, which we'll call `Game_cons`, for "Game constructor". We decided to use indexing to represent a set, so calling the index set `I`, it looks like:

```
Inductive Game : Type :=
 Game_cons : (I -> Game) -> (I -> Game) -> Game.
```

What exactly is I? Adding it as a parameter to our theory wouldn't be right, because then the size (in terms of cardinal) of the left and right sets of our games would be limited by the size of I. Therefore, it has to be a parameter of the constructor, so that it can change (and grow without upper bound) at each application of the constructor. This approach is inspired by [Wer97].

```
Inductive Game : Type :=
 Game_cons : (I:Type) (I -> Game) -> (I -> Game) -> Game.
```

Here, the left and right share the same index. While at first this doesn't look like a problem (if one is smaller than the other, having the indexing functions non-injective solves it), it in fact forces the left and right to be simultaneously empty or non-empty (I empty or non-empty).

One approach to address this would be to have indexing functions of type `I -> (Option Game)`, where `Option Game` is `Game` with one point, standing for "nothing", added, and axiomatising the properties of that "nothing" point, which brings added complexity, if not inconsistency.

A cleaner solution, and the one we will use, is simply to take separate indexes for the left and right:

```
Inductive Game : Type :=
 Game_cons : (LI,RI:Type) (LI -> Game) -> (RI -> Game) -> Game.
```

Type vs Set

The indexes in the constructor of type `Game` are of sort `Type`, "pushing" `Game` itself in the `Type` realm. Could we have them in `Set`? Well, no.

Explaining precisely why is out of scope of this document, but a somewhat technical explanation is that while this definition

```
Inductive Game : Set :=
 Game_cons : (LI,RI:Set) (LI -> Game) -> (RI -> Game) -> Game.
```

is valid and accepted by Coq, there is not much you can do with it. For example one cannot write the projection that from a game yields its left index. Accepting this projection would lead to an inconsistent system.

Utilities

We now define the projections extracting the various components of a Game. Extracting the indexes is quite straightforward, but the type inference engine needs some help for the functions extraction.

```
Definition GLeftIndex  := [G:Game] Cases G of (Game_cons A B f g) => A end.
Definition GRightIndex := [G:Game] Cases G of (Game_cons A B f g) => B end.
Definition GLeftFun  := [G:Game]  <[G:Game](GLeftIndex G)->Game>
                                   Cases G of (Game_cons A B f g) => f end.
Definition GRightFun := [G:Game] <[G:Game](GRightIndex G)->Game>
                                  Cases G of (Game_cons A B f g) => g end.
```

2.1 Order

We now model the order on games in Coq. This order is partial on games, but total on surreal numbers.

2.1.0 Definition in Coq

Fixpoint or Inductive?

The first choice that springs up is whether it should be considered as a function (introduced by `Fixpoint`) or as a (parametric) inductive type (introduced by `Inductive`). The design choice made here is to have it as an inductive type: it makes inductive proofs over or using the order much easier to write, and this is the main use of the order.

Definition

Defining the order in Coq is not quite as straightforward as putting its definition in Coq syntax. We construct it by trial and error, successive refinement.

The CIC doesn't accept the definition of the order as given in 1.0.5:

```
Inductive Glte : Game -> Game -> Prop :=
 Glte_cons : (xLI,xRI:Type)(xLf:xLI->Game)(xRf:xRI->Game)
             (yLI,yRI:Type)(yLf:yLI->Game)(yRf:yRI->Game)
             [x:=(Game_cons xLI xRI xLf xRf);
              y:=(Game_cons  yLI yRI yLf yRf)]
             ((l:xLI) [xl:=(xLf l)] ~(Glte xl y      )) ->
             ((r:yRI)               ~(Glte x  (yRf r))) ->
             (Glte x y)
.
```

`~(Glte xl y)` is (by definition of negation) `(Glte xl y) -> False`, and thus this occurrence of `Glte` is non-positive (this is the same issue as in the definition of games, page 19). The solution, inspired by [Ros01], is defining two predicates mutually inductive: "less than or equal" ($\leq$, `Glte`[0]) and "not greater than or equal" ($\lhd$, `NGgte`[1]), i.e.:

$$\begin{aligned} x \leq y &\leftrightarrow (\forall x_l, \neg(x_l \geq y)) \wedge (\forall y_r, \neg(x \geq y_r)) \\ &\overset{\text{def}}{\leftrightarrow} (\forall x_l, x_l \lhd y) \wedge (\forall y_r, x \lhd y_r) \end{aligned}$$

$$\begin{aligned} x \lhd y &\leftrightarrow \neg(y \leq x) \\ &\leftrightarrow \neg((\forall y_l, \neg(y_l \geq x)) \wedge (\forall x_r, \neg(y \geq x_r))) \\ &\overset{\text{def}}{\leftrightarrow} (\exists y_l, x \leq y_l) \vee (\exists x_r, x_r \leq y) \end{aligned}$$

In Coq syntax:

[0] Game less than or equal.

[1] Not Game greater than or equal.

```
Inductive Glte : Game -> Game -> Prop :=
 Glte_cons : (xLI,xRI:Type)(xLf:xLI->Game)(xRf:xRI->Game)
             (yLI,yRI:Type)(yLf:yLI->Game)(yRf:yRI->Game)
             [x:=(Game_cons xLI xRI xLf xRf);
              y:=(Game_cons  yLI yRI yLf yRf)]
             ((l:xLI) [xl:=(xLf l)](NGgte xl y)) ->
             ((r:yRI) (NGgte x (yRf r))) ->
             (Glte x y)
with NGgte : Game -> Game -> Prop :=
 NGgte_xr : (xLI,xRI:Type)(xLf:xLI->Game)(xRf:xRI->Game)(y:Game)
             [x:=(Game_cons xLI xRI xLf xRf)]
             (exT xRI [r:xRI] (Glte (xRf r) y)) ->
             (NGgte x y)
|NGgte_yl : (x:Game)(yLI,yRI:Type)(yLf:yLI->Game)(yRf:yRI->Game)
             [y:=(Game_cons  yLI yRI yLf yRf)]
             (exT yLI [l:yLI] (Glte x (yLf l))) ->
             (NGgte x y)
.
```

2.1.1 Glte and NGgte

$\leq$ and $\lhd$ are supposed to be linked by the formula:

$$x \leq y \leftrightarrow \neg(y \lhd x)$$

In this section, we prove this formally in Coq. This is the first proof we do, and I thus present it here in much more detail than the others; many approaches and problems that repeat for most proofs first show up here.

The equivalence is first decomposed into the two implications, back and forth.

Back

First, the $x \leq y \rightarrow \neg(y \lhd x)$ direction. We'll prove the contraposition, namely $y \lhd x \rightarrow \neg\,(x \leq y)$. This is done by induction, by first proving that the induction principle applies here, and then that the induction principle is a valid one.

The induction principle applies This is the following lemma:

$$\begin{aligned} \forall x, \forall y, \quad & (\forall y_l, y_l \lhd x \rightarrow \neg(x \leq y_l)) \rightarrow \\ & (\forall x_r, y \lhd x_r \rightarrow \neg(x_r \leq y)) \rightarrow \\ & (y \lhd x \rightarrow \neg\,(x \leq y)) \end{aligned}$$

In Coq syntax:

```
Theorem NGgteIsNGgte_Step :
        (xLI,xRI:Type)(xLf:xLI->Game)(xRf:xRI->Game)
```

```
(yLI,yRI:Type)(yLf:yLI->Game)(yRf:yRI->Game)
[x:=(Game_cons xLI xRI xLf xRf);
 y:=(Game_cons  yLI yRI yLf yRf)]
((yli:yLI)[yl:=(yLf yli)] (NGgte yl x) -> ~(Glte x yl)) ->
((xri:xRI)[xr:=(xRf xri)] (NGgte y xr) -> ~(Glte xr y)) ->
((NGgte x y) -> ~(Glte y x)).
```

Proof Initial situation:

```
============================
 (xLI,xRI:Type; xLf:(xLI->Game); xRf:(xRI->Game); yLI,yRI:Type;
  yLf:(yLI->Game); yRf:(yRI->Game))
  [x:=(Game_cons xLI xRI xLf xRf)]
   [y:=(Game_cons yLI yRI yLf yRf)]
    ((yli:yLI)[yl:=(yLf yli)](NGgte yl x)->~(Glte x yl))
    ->((xri:xRI)[xr:=(xRf xri)](NGgte y xr)->~(Glte xr y))
    ->(NGgte x y)
    ->~(Glte y x)
```

First, we do $\rightarrow_{intro}$, taking care to give the hypotheses reasonable names[2], and asking Coq to treat `~X` as `X -> False` and not an opaque definition[3]:

```
Intros until y; Intros IH0 IH1.
Unfold not; Intros NGxy Lyx.
```

```
xLI : Type
xRI : Type
xLf : xLI->Game
xRf : xRI->Game
yLI : Type
yRI : Type
yLf : yLI->Game
yRf : yRI->Game
x := (Game_cons xLI xRI xLf xRf) : Game
y := (Game_cons yLI yRI yLf yRf) : Game
IH0 : (yli:yLI)[yl:=(yLf yli)](NGgte yl x)->~(Glte x yl)
IH1 : (xri:xRI)[xr:=(xRf xri)](NGgte y xr)->~(Glte xr y)
NGxy : (NGgte x y)
Lyx : (Glte y x)
============================
 False
```

[2]IH stands for Induction Hypothesis throughout my Coq code.

[3]An opaque definition is a definition that should not automatically be replaced by its right hand side. If `not` is considered opaque, then `(not P)` will not automatically be replaced by `P->False`. But if `not` is considered *transparent* (not opaque), then `(not P)` is printed on screen as `(not P)`, but treated just like `P->False`

Starting now, I won't repeat the lines of the context that don't change or are irrelevant, or goals we aren't currently working on. Now, we deconstruct NGxy:

```
Simple Inversion NGxy.

  xLI0 : Type
  xRI0 : Type
  xLf0 : xLI0->Game
  xRf0 : xRI0->Game
  y0 : Game
  x0 := (Game_cons xLI0 xRI0 xLf0 xRf0) : Game
  H0 : x0==x
  H1 : y0==y
  ============================
   (EXT r:xRI0 | (Glte (xRf0 r) y0))->False

subgoal 2 is:
 (EXT l:yLI0 | (Glte x0 (yLf0 l)))->False
```

Coq has introduced synonyms for x and y, and used those synonyms in the new goals it has generated. But those goals follow from the previous hypotheses, which haven't changed and thus still use the names x and y. The first task at hand is thus to go back to those "canonical" names. First, replace y0 by y:

```
Rewrite H1.

  ============================
   (EXT r:xRI0 | (Glte (xRf0 r) y))->False
```

Then, we'd like to replace xRI0 by xRI and xRf0 by xRf. But we don't have a hypothesis that says we can do that, only that x, as a whole equals x0. We must thus generate the "sub-equalities", between the components of x and x0[4]:

```
Injection H0.

  ============================
   (existT Type [RI:Type]RI->Game xRI0 xRf0)
    ==(existT Type [RI:Type]RI->Game xRI xRf)
   ->(existT Type [LI:Type]LI->Game xLI0 xLf0)
      ==(existT Type [LI:Type]LI->Game xLI xLf)
   ->xRI0==xRI
   ->xLI0==xLI
   ->(EXT r:xRI0 | (Glte (xRf0 r) y))
   ->False
```

[4]Equality is here Coq equality, i.e. Leibniz equality of the representatives as modelled in Coq, which is more restrictive than being identical: Identical means the left and right are the same sets, but our model has many different representations of the same set.

We put these newly generated premises as hypotheses:

```
Intros xRf02xRf xLf02xLf xRI02xRI xLI02xLI.

  xRf02xRf : (existT Type [RI:Type]RI->Game xRI0 xRf0)
             ==(existT Type [RI:Type]RI->Game xRI xRf)
  xLf02xLf : (existT Type [LI:Type]LI->Game xLI0 xLf0)
             ==(existT Type [LI:Type]LI->Game xLI xLf)
  xRI02xRI : xRI0==xRI
  xLI02xLI : xLI0==xLI
  ============================
   (EXT r:xRI0 | (Glte (xRf0 r) y))->False
```

And use them to rewrite the goal:

```
Dependent Rewrite -> xRf02xRf.

  ============================
   (EXT r:(projT1 Type [RI:Type]RI->Game
             (existT Type [RI:Type]RI->Game xRI xRf)) |
        (Glte
          (projT2 Type [RI:Type]RI->Game
            (existT Type [RI:Type]RI->Game xRI xRf) r) y))
   ->False
```

Coq doesn't automatically reduce the term, so let's do it:

```
Cbv Beta Iota Delta -[y].

  ============================
   (EXT r:xRI | (Glte (xRf r) y))->False
```

Now, we can put the premise as hypothesis, and do the logic step "there is one x_r such that $x_r \leq y$, well let's call it `smallxr`" (again, Coq adds it as premises, that we immediately pull into the hypotheses):

```
Intro Ex_Lxry.
Elim Ex_Lxry.
Intros smallxr smallxr_is_small.

  Ex_Lxry : (EXT r:xRI | (Glte (xRf r) y))
  smallxr : xRI
  smallxr_is_small : (Glte (xRf smallxr) y)
  ============================
   False
```

We use our second induction hypothesis:

```
Unfold not in IH1; Apply IH1 with xri:=smallxr; Fold not in IH1.

  ============================
   (NGgte y (xRf smallxr))

subgoal 2 is:
 (Glte (xRf smallxr) y)
```

The first goal is a consequence of Lxy, so we deconstruct it, too, and the same story of yet another synonym for x and y happens:

```
Simple Inversion Lyx.
Rewrite H4; Rewrite H3.
Intros All_NGylx All_NGyxr.
Injection H4.
Intros yRf02xRf yLf02xLf yRI02xRI yLI02xLI.

  xLI1 : Type
  xRI1 : Type
  xLf1 : xLI1->Game
  xRf1 : xRI1->Game
  yLI0 : Type
  yRI0 : Type
  yLf0 : yLI0->Game
  yRf0 : yRI0->Game
  x1 := (Game_cons xLI1 xRI1 xLf1 xRf1) : Game
  y1 := (Game_cons yLI0 yRI0 yLf0 yRf0) : Game
  H3 : x1==y
  H4 : y1==x
  All_NGylx : (l:xLI1)[xl:=(xLf1 l)](NGgte xl x)
  All_NGyxr : (r:yRI0)(NGgte y (yRf0 r))
  yRf02xRf : (existT Type [RI:Type]RI->Game yRI0 yRf0)
             ==(existT Type [RI:Type]RI->Game xRI xRf)
  yLf02xLf : (existT Type [LI:Type]LI->Game yLI0 yLf0)
             ==(existT Type [LI:Type]LI->Game xLI xLf)
  yRI02xRI : yRI0==xRI
  yLI02xLI : yLI0==xLI
  ============================
   (NGgte y (xRf smallxr))
```

This is an instantiation of `All_NGyxr`, modulo the synonyms renaming complication. We'd like to replace `xRf` by `yRf0`, but this is impossible, because then `(yRf0 smallxr)` wouldn't be a well-typed term. We must simultaneously replace `xRf` by `yRf0` and `smallxr` by something of type `yRI0`. We achieve this by generalising over `smallxr`, i.e. by doing the logic step "if it is true for any `smallxr`, then it certainly is true for this particular `smallxr`".

```
Generalize smallxr.

  ============================
   (smallxr0:xRI)(NGgte y (xRf smallxr0))
```

Now, by simultaneously replacing `xRf` by `yRf0` and `xRI` by `yRI0`, we keep well-typedness of our terms:

```
Dependent Rewrite <- yRf02xRf.
Cbv Beta Iota Delta -[y].

  ============================
   (smallxr0:yRI0)(NGgte y (yRf0 smallxr0))
```

and the goal is now exactly[5] `All_NGyxr`. This fully resolves the current goal, and thus the next goal pops to the top of the stack, with its environment:

```
Exact All_NGyxr.

  smallxr : xRI
  smallxr_is_small : (Glte (xRf smallxr) y)
  ============================
   (Glte (xRf smallxr) y)
```

That one is easy, the goal is exactly `smallxr_is_small`. We still had a goal in the stack, the second one that was generated by deconstructing `NGxy`:

```
Exact smallxr_is_small.

  (...)
  H0 : x0==x
  H1 : y0==y
  ============================
   (EXT l:yLI0 | (Glte x0 (yLf0 l)))->False
```

Its proof is very similar to the one we did in detail here, therefore we will skip it.

The induction scheme is valid As announced 22, we now prove that the induction scheme used is valid. This is the following theorem:

$$\forall P,\ (\forall x, \forall y, (\forall y_l, P(y_l, x)) \to (\forall x_r, P(y, x_r)) \to P(x, y)) \to (\forall x, \forall y, P(x, y))$$

In Coq syntax[6]:

[5] modulo α-conversion, i.e. renaming of bound variables

[6] IS stands for Induction Scheme

```
Theorem IS2:  (P:Game->Game->Prop)
             ((x,y:Game)
              ((yli:(GLeftIndex y))[yl:=((GLeftFun y) yli)] (P yl x)) ->
              ((xri:(GRightIndex x))[xr:=((GRightFun x) xri)] (P y xr)) ->
              (P x y)) ->
             (x,y:Game) (P x y).
```

I have not found a better way to prove this than to program the (recursive) function that computes a proof of $P(x, y)$ by using the induction hypotheses directly. This by itself is non-trivial to do in Coq, as we'll see.

Let's first write this function in an ML-like syntax[7]. `IH`, the induction hypothesis is a function that takes as arguments:

1. x and y two games
2. A proof of $P(y_l, x)$ for all y_l, i.e. a function that takes an y_l and returns a proof of $P(y_l, x)$
3. A proof of $P(y, x_r)$ for all x_r, i.e. a function that takes an x_r and returns a proof of $P(y, x_r)$

and returns a proof of $P(x, y)$ Then, the proof construction function can be written as[8]:

```
let proof P IH =
 let rec pc x y = IH x y (fun yl -> (pc yl x)) (fun xr -> (pc y xr))
 in pc;;
```

This function is a proof of the validity of the induction scheme if and only if `pc` terminates on any input. To convince everyone that it does, here is a well-founded ordering $\overset{\Psi}{<}$ of tuples of games:

$$\begin{aligned} \Psi((x,y)) &\overset{\text{def}}{=} (\min(\rho(x),\rho(y)), \max(\rho(x),\rho(y))) \\ \overset{l}{<} &\overset{\text{def}}{=} \text{lexicographic order} \\ (x_1,y_1) \overset{\Psi}{<} (x_2,y_2) &\overset{\text{def}}{\leftrightarrow} \Psi((x_1,y_1)) \overset{l}{<} \Psi((x_2,y_2)) \end{aligned}$$

The recursive calls are strictly decreasing for $\overset{\Psi}{<}$, thus terminating.

By its very design, the CIC (being based on type theory) natively deals only with total functions[9]. It must thus restrict the recursive definitions it accepts to those defining a total function, or a subset thereof. The design choice made in the CIC is to have a simple, purely syntactical acceptance condition, which can be summarised and simplified as: There must be one argument, the i^{th} one, called *decreasing argument*, such that in each recursive call, the i^{th} parameter is structurally smaller than the i^{th} parameter in the left hand-side of the definition (the i^{th} definition-parameter). E.g., if f is defined by $f(x,y) = f(t_1,t_2) + f(t_3,t_4)$,

[7]ML is a strongly typed functional higher order programming language

[8]`pc` stands for **p**roof **c**onstructor

[9]There are various way to model partial functions.

and i is chosen to be 0 (recursion on the first argument, thus), then for this definition to be accepted, it is necessary that $t_1 \lessdot x$ and $t_3 \lessdot x$.

In the case of our naive definition of `pc`, this condition doesn't hold for any of the two arguments:

1. The first problem is that the parameters are swapped in the recursive calls; for any i, the i^{th} parameter is not comparable with the i^{th} definition-parameter. This can be worked around by defining two functions mutually inductive:

   ```
   let proof P IH =
    let rec pc1 x y = IH x y (fun yl -> (pc2 yl x)) (fun xr -> (pc2 y xr))
        and pc2 y x = IH y x (fun xl -> (pc1 xl y)) (fun yr -> (pc1 x yr))
    in pc1;;
   ```

   Let's call i_{pcX} the index of the decreasing argument of `pcX`. By choosing $i_{pc1} \in \{0, 1\}$ and $i_{pc2} = 1 - i_{pc1}$, we guarantee that parameters in recursive calls are compared only with the definition-parameter they are derived from in the expression of the condition.

2. The second problem is that none of the parameters is always structurally smaller than its corresponding definition-parameter: Both `x` and `y` appear as a whole in the right hand-side of the definition. If we were to choose $i_{pc1} = 0$, the first call to `pc2` (i.e. `(pc2 yl x)`) would not fulfil the condition, and if we choose $i_{pc1} = 1$, the second call to `pc2` (i.e. `pc2 y xr`) would not fulfil the condition.

   The solution to this, inspired from the classical example of the Ackerman function (whose definition brings the same difficulty) is decomposing the descent into the components of `x` and the components of `y`. The idea is to "swallow" the recursion on `y` (where `x` stays constant[10]) into a recursive function `auxpc`[11], whose recursion is embedded in the recursion of `pc`, i.e. we wish to replace the recursive call `pc yl x` by a call to `auxpc yl`:

   ```
   let proof P IH =
    let rec pc x y =
      let rec auxpc z = (* pc z x *)
        IH z x (fun xl -> (pc xl z)) (fun zr -> (pc x zr))
      in
      IH x y (fun yl -> (auxpc yl)) (fun xr -> (pc y xr))
    in pc;;
   ```

   The intention is to take $i_{pc} = i_{auxpc} = 1$. This nearly works: the recursive call `pc x zr` still breaks the condition. If the order of the parameters wasn't swapped, it would work, we could replace `pc x zr` by `auxpc zr`. Tackling the parameter swapping problem in the way suggested before doesn't work: `pc1` would have to call `auxpc2`, but `auxpc2` doesn't exist in the definition of `pc1`, it is out of scope.

[10] The choice of `x`, and not `y` is purely arbitrary

[11] `aux` for auxiliary

Thus, we must tackle the last problem is some other way. A way that actually works is unfolding the definition of pc in its recursive calls. In some sense, doing one recursive call "by hand", to swap the arguments again in place:

```
let proof P IH =
 let rec pc x y =
   IH x y
      (fun yl -> (IH yl x (fun xl  -> (pc xl yl))
                          (fun ylr -> (pc x  ylr))))
      (fun xr -> (IH y xr (fun xrl -> (pc xrl y))
                          (fun yr  -> (pc xr yr))))
 in pc;;
```

Combining this with the auxpc trick gives us the kind of definition that is easier to write than to read (because it is largely a mechanical, calculation process to write the definition, but seeing the calculation steps in the result is more difficult):

```
let proof P IH =
 let rec pc x y =
  let rec auxpc z = (* pc x z *)
    IH x z
       (fun zl -> (IH zl x (fun xl  -> (pc xl zl))
                           (fun zlr -> (auxpc zlr))))
       (fun xr -> (IH z xr (fun xrl -> (pc xrl z))
                           (fun zr  -> (pc xr zr))))
  in
    IH x y
       (fun yl -> (IH yl x (fun xl  -> (pc xl yl))
                           (fun ylr -> (auxpc ylr))))
       (fun xr -> (IH y xr (fun xrl -> (pc xrl y))
                           (fun yr  -> (pc xr yr))))
 in pc;;
```

This is accepted by the CIC:

(a) auxpc terminates because in its only recursive call, its only parameter is structurally smaller than its only definition-parameter.

(b) Once it is established that auxpc terminates, pc terminates too, because all its recursive calls have the first parameter (namely xl, xrl, xr) structurally smaller than its first definition-parameter (namely x), and all the function calls its definition contains are recursive calls and calls to auxpc, which terminates.

The last step left is putting this definition in Coq syntax:

```
Intros P IH.
Exact Fix pc { pc [x:Game]:(y:Game)(P x y) := [y:Game]
      [xLI:=(GLeftIndex x)][yRI:=(GRightIndex y)]
      [xRI:=(GRightIndex x)][yLI:=(GLeftIndex y)]
      let auxpc = Fix auxpc {auxpc [z:Game]:(P x z) :=
        [zRI:=(GRightIndex z)][zLI:=(GLeftIndex z)]
        (IH x z
          ([zli:zLI][zl:=((GLeftFun z) zli)]
           (IH zl x
               ([xli:xLI][xl:=((GLeftFun x) xli)] (pc xl zl))
               ([zlri:(GRightIndex zl)][zlr:=((GRightFun zl) zlri)]
                (auxpc zlr))))
          ([xri:xRI][xr:=((GRightFun x) xri)]
           (IH z xr
               ([xrli:(GLeftIndex xr)][xrl:=((GLeftFun xr) xrli)]
                (pc xrl z))
               ([zri:zRI][zr:=((GRightFun z) zri)](pc xr zr)))))

    }
    in
   (IH x y
     ([yli:yLI][yl:=((GLeftFun y) yli)]
      (IH yl x
          ([xli:xLI][xl:=((GLeftFun x) xli)] (pc xl yl))
          ([ylri:(GRightIndex yl)][ylr:=((GRightFun yl) ylri)]
           (auxpc ylr))))
     ([xri:xRI][xr:=((GRightFun x) xri)]
      (IH y xr
          ([xrli:(GLeftIndex xr)][xrl:=((GLeftFun xr) xrli)] (pc xrl y))
          ([yri:yRI][yr:=((GRightFun y) yri)](pc xr yr)))))
}
.
```

Remark 2.1.1 *This definition breaks the symmetry between x and y.*

Now, putting the pieces together. First, use the induction scheme:

```
Theorem NGgteIsNGgte :  (x,y:Game)
              (NGgte x y) -> ~(Glte y x).
Intros x y.
EApply IS2 with P:=[a,b:Game]((NGgte a b)->~(Glte b a)).

  x : Game
  y : Game
  ============================
```

```
  (x0,y0:Game)
   ((yli:(GLeftIndex y0))
     [yl:=(GLeftFun y0 yli)](NGgte yl x0)->~(Glte x0 yl))
   ->((xri:(GRightIndex x0))
       [xr:=(GRightFun x0 xri)](NGgte y0 xr)->~(Glte xr y0))
   ->(NGgte x0 y0)
   ->~(Glte y0 x0)
```

x has been renamed to x0, but this time only x0 appears in our context and goal, thus we can throw away x, and rename x0 to x, and the same for y.

```
Clear x y.
Intros x y.
```

```
  x : Game
  y : Game
  ============================
   ((yli:(GLeftIndex y))
     [yl:=(GLeftFun y yli)](NGgte yl x)->~(Glte x yl))
   ->((xri:(GRightIndex x))
       [xr:=(GRightFun x xri)](NGgte y xr)->~(Glte xr y))
   ->(NGgte x y)
   ->~(Glte y x)
```

This essentially is `NGgteIsNGgte_Step` (introduced on page 22), but technically, the latter doesn't apply to two games, but to 4 sets of games. We must thus "open up" the structure of x and y:

```
Case x; Case y.
Clear x y.
Intros yLI yRI yLf yRf xLI xRI xLf xRf.
Cbv Beta Iota Delta -[not].
```

```
  yLI : Type
  yRI : Type
  yLf : yLI->Game
  yRf : yRI->Game
  xLI : Type
  xRI : Type
  xLf : xLI->Game
  xRf : xRI->Game
  ============================
   ((yli:yLI)
     [yl:=(yLf yli)]
      (NGgte yl (Game_cons xLI xRI xLf xRf))
      ->~(Glte (Game_cons xLI xRI xLf xRf) yl))
```

```
->((xri:xRI)
    [xr:=(xRf xri)]
     (NGgte (Game_cons yLI yRI yLf yRf) xr)
     ->~(Glte xr (Game_cons yLI yRI yLf yRf)))
->(NGgte (Game_cons xLI xRI xLf xRf) (Game_cons yLI yRI yLf yRf))
->~(Glte (Game_cons yLI yRI yLf yRf) (Game_cons xLI xRI xLf xRf))
```

And we finish by using `NGgteIsNGgte_Step`:

```
Intros.
Apply NGgteIsNGgte_Step; Assumption.
```

This concludes the proof of $y \lhd x \to \neg\,(x \leq y)$.

Forth

The $\neg(y \lhd x) \to (x \leq y)$ direction uses the same induction scheme as the other direction above, but uses classical (non-constructive) reasoning. We thus add the `Classic` axiom, and start the proof by applying the induction scheme and decomposing x and y:

```
Axiom Classic: (P:Prop) ~~P->P.

Theorem NGgteNGgte : (x,y:Game) ~(NGgte x y) -> (Glte y x).
Intros x y.
EApply IS2 with P:=[a,b:Game](~(NGgte a b) -> (Glte b a)).
Clear x y.
Intros x y.
Case x.
Intros xLI xRI xLf xRf.
Case y.
Intros yLI yRI yLf yRf.
Clear x y.
LetTac x:=(Game_cons xLI xRI xLf xRf).
LetTac y:=(Game_cons yLI yRI yLf yRf).
Intros IHy IHx NNyx.

  xLI : Type
  xRI : Type
  xLf : xLI->Game
  xRf : xRI->Game
  yLI : Type
  yRI : Type
  yLf : yLI->Game
  yRf : yRI->Game
  x := (Game_cons xLI xRI xLf xRf) : Game
  y := (Game_cons yLI yRI yLf yRf) : Game
```

```
  IHy : (yli:(GLeftIndex y))
        [yl:=(GLeftFun y yli)]~(NGgte yl x)->(Glte x yl)
  IHx : (xri:(GRightIndex x))
        [xr:=(GRightFun x xri)]~(NGgte y xr)->(Glte xr y)
  NNyx : ~(NGgte x y)
  ============================
   (Glte y x)
```

The hypothesis NNyx is (constructively) equivalent to $(\forall x_r \in R_x, \neg(x_r \leq y)) \wedge (\forall y_l \in L_y, \neg(x \leq y_l))$. This has been proven in the lemma NNGgteToProp, which is not shown here. I'd like to use this form, so I introduce it with a cut (also known as Modus Ponens[12]), and decompose it into its clauses:

```
Cut ((r:(GRightIndex x)) ~(Glte (GRightFun x r) y)) /\
                         ((l:(GLeftIndex y)) ~(Glte x (GLeftFun y l))).
Intro NNyxProp.
Decompose [and] NNyxProp.
Rename H into NNyx_1.
Rename H0 into NNyx_2.

  xLI : Type
  xRI : Type
  xLf : xLI->Game
  xRf : xRI->Game
  yLI : Type
  yRI : Type
  yLf : yLI->Game
  yRf : yRI->Game
  x := (Game_cons xLI xRI xLf xRf) : Game
  y := (Game_cons yLI yRI yLf yRf) : Game
  IHy : (yli:(GLeftIndex y))
        [yl:=(GLeftFun y yli)]~(NGgte yl x)->(Glte x yl)
  IHx : (xri:(GRightIndex x))
        [xr:=(GRightFun x xri)]~(NGgte y xr)->(Glte xr y)
  NNyx : ~(NGgte x y)
  NNyxProp : ((r:(GRightIndex x))~(Glte (GRightFun x r) y))
             /\ ((l:(GLeftIndex y))~(Glte x (GLeftFun y l)))
  NNyx_1 : (r:(GRightIndex x))~(Glte (GRightFun x r) y)
  NNyx_2 : (l:(GLeftIndex y))~(Glte x (GLeftFun y l))
  ============================
   (Glte y x)

subgoal 2 is:
 ((r:(GRightIndex x))~(Glte (GRightFun x r) y))
```

[12]This is the step "to prove Q, it suffices to prove P and $P \rightarrow Q$"

```
 /\ ((l:(GLeftIndex y))~(Glte x (GLeftFun y l)))
```

There is only one way `(Glte y x)` can be true (constructed): By using its only constructor. The tactic `Exists` implements this principle: to prove `(Glte y x)`, it suffices to prove the premises of its only constructor.

```
Exists.
Fold x.
Intros yli yl.

  yli : yLI
  yl := (yLf yli) : Game
  ============================
   (NGgte yl x)
```

We now use `Classic`, so that we can use the hypothesis `NNyx_2`

```
Apply Classic.
Unfold not; Intro nnylx.
Unfold not in NNyx_2; Apply NNyx_2 with l:=yli.

  nnylx : (NGgte yl x)->False
  ============================
   (Glte x (GLeftFun y yli))
```

Using the induction hypothesis on `y`, we reduce the goal to `nnylx`, and this goal is fully taken care of.

```
Apply IHy.
Exact nnylx.

  ============================
   (r:xRI)(NGgte (Game_cons yLI yRI yLf yRf) (xRf r))

subgoal 2 is:
 ((r:(GRightIndex x))~(Glte (GRightFun x r) y))
 /\ ((l:(GLeftIndex y))~(Glte x (GLeftFun y l)))
```

The first subgoal is symmetrical to the one we just did, its proof goes by substituting y_l by x_r in the proof we just did.

```
Intros xri.
Apply Classic.
Unfold not; Intro nnyxr.
Unfold not in NNyx_1; Apply NNyx_1 with r:=xri.
Apply IHx.
Exact nnyxr.
```

```
  IHy : (yli:(GLeftIndex y))
        [yl:=(GLeftFun y yli)]~(NGgte yl x)->(Glte x yl)
  IHx : (xri:(GRightIndex x))
        [xr:=(GRightFun x xri)]~(NGgte y xr)->(Glte xr y)
  NNyx : ~(NGgte x y)
  ============================
   ((r:(GRightIndex x))~(Glte (GRightFun x r) y))
   /\ ((l:(GLeftIndex y))~(Glte x (GLeftFun y l)))
```

This leaves us with the proposition we introduced, which is taken care of by the lemma `NNGgteToProp`.

```
Apply NNGgteToProp.
Exact NNyx.
Qed.
```

This concludes the proof of $\neg(y \lhd x) \to (x \leq y)$.

2.1.2 Conway definition of the order

We now prove that our definition of `Glte` is equivalent to the definition given by Conway, page 4, chapter 0.

Back

`SplitInTail` is the lemma $(P \to Q) \to (P \to R) \to P \to (Q \wedge R)$, i.e. to prove $P \to (Q \wedge R)$, it suffices to prove $P \to Q$ and $P \to R$.

```
Theorem NGgte_Conway_Definition :
   (xLI,xRI:Type)(xLf:xLI->Game)(xRf:xRI->Game)
   (yLI,yRI:Type)(yLf:yLI->Game)(yRf:yRI->Game)
        [x:=(Game_cons xLI xRI xLf xRf);
         y:=(Game_cons  yLI yRI yLf yRf)]
        (Glte x y) ->
        ((l:xLI) ~(Glte y (xLf l))) /\ ((r:yRI) ~(Glte (yRf r) x)).
Proof.
Intros until y.
Apply SplitInTail; Intros.

  xLI : Type
  xRI : Type
  xLf : xLI->Game
  xRf : xRI->Game
  yLI : Type
  yRI : Type
```

```
  yLf : yLI->Game
  yRf : yRI->Game
  x := (Game_cons xLI xRI xLf xRf) : Game
  y := (Game_cons yLI yRI yLf yRf) : Game
  H : (Glte x y)
  l : xLI
  ============================
   ~(Glte y (xLf l))

subgoal 2 is:
 ~(Glte (yRf r) x)
```

Again, the two subgoals are symmetrical and I will show only the proof of the first one. We apply NGgteIsNGgte:

```
Apply NGgteIsNGgte.

  ============================
   (NGgte (xLf l) y)
```

This is derived from Glte x y. Again, we have the "new name for x and y" syndrome and we rewrite to go back to one unique name:

```
Simple Inversion H.
Dependent Rewrite -> H2.
Dependent Rewrite -> H3.
Inversion H2.
Compute in H1;Compute in H4; Compute in H5; Compute in H6.
Dependent Rewrite -> H5.
Cbv Beta Iota Zeta Delta -[x0 y0 x y].

  l : xLI
   ============================
   ((l:xLI)(NGgte (xLf l) y))
   ->((r:yRI0)(NGgte x (yRf0 r)))
   ->(NGgte (xLf l) y)
```

This is simple enough to be handled by the tactic Auto.

Forth

Again, Glte is proven by proving the premises of its only constructor.

```
Theorem NGgte_Conway_Definition2 :
     (xLI,xRI:Type)(xLf:xLI->Game)(xRf:xRI->Game)
     (yLI,yRI:Type)(yLf:yLI->Game)(yRf:yRI->Game)
        [x:=(Game_cons xLI xRI xLf xRf);y:=(Game_cons  yLI yRI yLf yRf)]
```

```
        ((l:xLI) ~(Glte y (xLf l))) -> ((r:yRI) ~(Glte (yRf r) x)) ->
        (Glte x y).
Proof.
Intros.
Unfold x y.
Exists.
Fold y. 2: Fold x.

  H : (l:xLI)~(Glte y (xLf l))
  H0 : (r:yRI)~(Glte (yRf r) x)
  ============================
   (l:xLI)[xl:=(xLf l)](NGgte xl y)

subgoal 2 is:
 (r:yRI)(NGgte x (yRf r))
```

Both subgoals go by applying `nGlteNGgte`, the contraposition of `NGgteNGgte` (not detailed here), which brings us back to a hypothesis.

```
Intros.
Apply nGlteNGgte.
Unfold xl.
Apply H.

Intros.
Apply nGlteNGgte.
Auto.
Qed.
```

This concludes the formal proof that our definition of the order is equivalent to the definition given by Conway.

2.1.3 `Glte` is a pre-order

A pre-order is a relation that is

1. reflexive
2. transitive

We will prove the two separately and sequentially.

Reflexivity

By induction on `x`

```
Theorem Glte_reflexive: (x:Game) (Glte x x).
Intros.
NewInduction x.
Rename g into Lf; Rename H into LIH.
Intros Rf RIH.
LetTac x:=(Game_cons LI RI Lf Rf).

  LI : Type
  RI : Type
  Lf : LI->Game
  LIH : (T:LI)(Glte (Lf T) (Lf T))
  Rf : RI->Game
  RIH : (l:RI)(Glte (Rf l) (Rf l))
  x := (Game_cons LI RI Lf Rf) : Game
  ============================
   (Glte x x)
```

`Glte` is proven by the arguments to its constructor

```
  ============================
   (l:LI)[xl:=(Lf l)](NGgte xl x)

subgoal 2 is:
 (r:RI)(NGgte x (Rf r))
```

So, we have to prove `NGgte xl x`. This is consistent with the interpretation: As $x_l < x$, certainly $\neg(x_l \geq x)$. At the Coq level, we prove `NGgte xl x` by its second constructor, that decomposes the right hand side of the inequality (here, `x`).

```
Intros xli xl.
Unfold x; Constructor 2.

  ============================
   (EXT l:LI | (Glte xl (Lf l)))
```

We must find a left of `x` that is greater or equal to `xl`; `xl` itself fills the bill.

```
Exists xli.

  xl := (Lf xli) : Game
  ============================
   (Glte xl (Lf xli))
```

and `xl` is smaller or equal to itself by the induction hypothesis.

```
Unfold xl; Apply LIH.
```

The remaining subgoal is symmetric to the one we just did, and we are finished proving reflexivity of `Glte`.

Transitivity

Constructiveness The proof of transitivity uses `nGlteNGgte` (which itself derives from the non-constructive NGgteNGgte), and thus is non-constructive. I haven't really looked into the question whether it can be done constructively. As transitivity is used nearly everywhere, nearly all other proofs (properties of addition, multiplication, ...) are tainted, but this is the only tainting point: If transitivity is re-done in a constructive manner, all the rest is constructive (except equivalence between `NGgte x y` and `~(Glte y x)`).

Proof Transitivity uses yet another induction scheme, which I called `Roll3Induction`:

```
Theorem Roll3Induction:
  (P,P1,P2:Game->Game->Prop)
  [F:=[x,y,z:Game](P x y)->(P y z)->(P x z)]
  ((x,z:Game) ((xli:(GLeftIndex  x))[xl:=(GLeftFun  x xli)](P1 xl z)) ->
              ((zri:(GRightIndex z))[zr:=(GRightFun z zri)](P2 x zr)) ->
              (P x z)) ->
  ((x,z:Game) (P x z)->
                ((xli:(GLeftIndex  x))[xl:=(GLeftFun  x xli)](P1 xl z)) /\
                ((zri:(GRightIndex z))[zr:=(GRightFun z zri)](P2 x zr))) ->
  ((x,y,z:Game) (P1 x y) -> (P y z) -> (F y z x) -> (P1 x z)) ->
  ((x,y,z:Game) (P2 y z) -> (P x y) -> (F z x y) -> (P2 x z)) ->
  ((x,y,z:Game) (F x y z)).
```

This scheme is specific to proving transitivity of P. The idea behind this induction scheme is that $P(x, y)$ can be "broken up" into (i.e. is equivalent to the conjunction of) $\forall x_l, P_1(x_l, z)$ and $\forall z_r, P_2(x, z_r)$ (that's the first two hypotheses), and to separately prove that a sort of "weak transitivity" holds for each component. This will be used with $\leq$, which by its very definition is constituted by two such parts. The `Roll3` in the name expresses that the "weak" transitivity for P_1 and P_2 for x, y, z assumes transitivity for P for a cyclic rotation of x, y, z: in a nutshell, transitivity of P for the cyclic rotations of x, y, z proves transitivity of P for x, y, z.

`Roll3Induction` is a special case of `Rol3Induction`:

```
Theorem Rol3Induction:
  (P:Game->Game->Game->Prop)
  ((x,y,z:Game)
   ((xli:(GLeftIndex  x))[xl:=(GLeftFun  x xli)](P y z xl)) ->
   ((zri:(GRightIndex z))[zr:=(GRightFun z zri)](P zr x y)) ->
   (P x y z)) ->
  ((x,y,z:Game) (P x y z)).
```

`Rol3Induction` can be seen as a generalisation to the 3-parameters case of `IS2`. The premises of the induction hypotheses in `IS2` are obtained by swapping the arguments of P, and

replacing one of the arguments by a child (a left or a right). To go to the 3-parameters case, "swapping" is replaced by "cyclicly rotating[13]".

Both `IS2` and `Rol3Induction` can be generalised into:

Conjecture 2.1.1 (Permuting inductions) *For $\lll$ a well-founded (strict) order, for n a natural number, P a predicate with n parameters, any induction scheme S of the following form:*

- *S is*
$$(\forall(x_0,\ldots,x_{n-1}),H) \to (\forall(x_0,\ldots,x_{n-1}),P(x_0,\ldots,x_{n-1}))$$
- *H is of the form*
$$P(t_{0,0},\ldots,t_{0,n-1}) \to \cdots \to P(t_{p,0},\ldots,t_{p,n-1}) \to P(x_0,\ldots,x_{n-1})$$
- *$\forall i < p$, σ_i is a permutation of $\{e \in \mathbb{N}, e < n\}$, and*
$$\forall j, t_{i,j} \lll x_{\sigma_i(j)} \vee t_{i,j} \equiv x_{\sigma_i(j)}$$
and
$$\forall i, \exists j, t_{i,j} \lll x_{\sigma_i(j)}$$

is valid.

Proof idea: I here call "birth date" an ordinal measurement of the universe X the x_i's live in, compatible with $\lll$. This means that $x \lll y \to \rho(x) < \rho(y)$. Order the $(x_0,\ldots,x_{n-1})$ tuples by the sorted tuple of the birth dates of its elements. Lexicographic order on this sorted birth dates tuple gives a well-founded order of X^n for which the induction scheme is decreasing.

This is called "conjecture" only because I haven't fully written out the proof. This conjecture is not expressible in the CIC, but most probably a tactic can be programmed to generate a CIC-proof of any instantiation of it (for a fixed n, a fixed p and fixed σ_i's, given a proof of well-foundedness of $\lll$ or for the case where $\lll$ is $<$). Every instantiation of this conjecture I used in my Coq proofs has been proven independently.

Proof of `Rol3Induction` in Coq The proof of `Rol3Induction` goes as the proof for `IS2`: by programming a proof constructor. For `IS2`, this was handled by a two-level use of the induction hypothesis (to put the arguments back in their "natural" order), because the permutations involved were of order 2. Here, the permutations involved are of order three, so induction hypothesis calls have to be embedded to the level three. Each auxiliary proof constructor has itself to be programmed with an auxiliary to the auxiliary. And because there are two different permutations involved, there are two mutually inductive proof constructors. The process of programming these proof constructors is quite mechanic, but gives rise to a rather large proof, in this case 67 lines long. You can see it in appendix B.

[13] Although it could be replaced by "permuting" or "swapping any number of times" (which amounts to the same)

Roll3Induction is a special case of Rol3Induction: In order to prove that Roll3Induction is a special case of Rol3Induction, we prove transitivity of P by Rol3Induction

```
Intros P P1 P2 F P12P PP12 p1 p2.
Intros x y z.
EApply Rol3Induction with P:=F.
Clear x y z.
Intros x y z.
Intros IH1 IH2.
```

```
  P : Game->Game->Prop
  P1 : Game->Game->Prop
  P2 : Game->Game->Prop
  F := [x,y,z:Game](P x y)->(P y z)->(P x z) : Game->Game->Game->Prop
  P12P : (x,z:Game)
         ((xli:(GLeftIndex x))[xl:=(GLeftFun x xli)](P1 xl z))
         ->((zri:(GRightIndex z))[zr:=(GRightFun z zri)](P2 x zr))
         ->(P x z)
  PP12 : (x,z:Game)
         (P x z)
         ->((xli:(GLeftIndex x))[xl:=(GLeftFun x xli)](P1 xl z))
           /\ ((zri:(GRightIndex z))[zr:=(GRightFun z zri)](P2 x zr))
  p1 : (x,y,z:Game)(P1 x y)->(P y z)->(F y z x)->(P1 x z)
  p2 : (x,y,z:Game)(P2 y z)->(P x y)->(F z x y)->(P2 x z)
  x : Game
  y : Game
  z : Game
  IH1 : (xli:(GLeftIndex x))[xl:=(GLeftFun x xli)](F y z xl)
  IH2 : (zri:(GRightIndex z))[zr:=(GRightFun z zri)](F zr x y)
  ============================
   (P x y)->(P y z)->(P x z)
```

We decompose (P x z) into its components

```
Intros xy yz.
Apply P12P.
```

```
  xy : (P x y)
  yz : (P y z)
  ============================
   (xli:(GLeftIndex x))[xl:=(GLeftFun x xli)](P1 xl z)

subgoal 2 is:
 (zri:(GRightIndex z))[zr:=(GRightFun z zri)](P2 x zr)
```

And use the "weak induction" to prove them

```
Intros xli xl.
EApply p1.

  xli : (GLeftIndex x)
  xl := (GLeftFun x xli) : Game
  ============================
   (P1 xl ?38)

subgoal 2 is:
 (P ?38 z)
subgoal 3 is:
 (F ?38 z xl)
```

We instantiate the meta-variable (hole) immediately

```
2: Apply yz.
```

And use the first induction hypothesis to prove transitivity for P on y, z, x_l

```
2:Apply IH1 with xli:=xli.
```

Now, only `P1 xl y` is left, and this derives (by `PP12`) from `P x y`, which we have as hypothesis `xy`.

```
Unfold xl.
Generalize xli.
EApply proj1.
Apply PP12.
Exact xy.
```

Treating the second component is very similar, and not shown here.

The induction scheme applies We now can use our new `Roll3Induction` induction scheme to prove transitivity of the order. We then decompose `x`, `y` and `z`.

```
Theorem Glte_transitive : (x,z,y:Game)
         (Glte x y) -> (Glte y z) -> (Glte x z).
Proof.
Intros x y z.
EApply Roll3Induction with P :=[x,y:Game] (Glte x y)
                           P1:=[x,y:Game] (NGgte x y)
                           P2:=[x,y:Game] (NGgte x y).

  x : Game
  y : Game
  z : Game
  ============================
```

```
   (x0,z0:Game)
    ((xli:(GLeftIndex x0))[xl:=(GLeftFun x0 xli)](NGgte xl z0))
    ->((zri:(GRightIndex z0))[zr:=(GRightFun z0 zri)](NGgte x0 zr))
    ->(Glte x0 z0)

subgoal 2 is:
 (x0,z0:Game)
  (Glte x0 z0)
  ->((xli:(GLeftIndex x0))[xl:=(GLeftFun x0 xli)](NGgte xl z0))
    /\ ((zri:(GRightIndex z0))[zr:=(GRightFun z0 zri)](NGgte x0 zr))
subgoal 3 is:
 (x0,y0,z0:Game)
  (NGgte x0 y0)
  ->(Glte y0 z0)
  ->((Glte y0 z0)->(Glte z0 x0)->(Glte y0 x0))
  ->(NGgte x0 z0)
subgoal 4 is:
 (x0,y0,z0:Game)
  (NGgte y0 z0)
  ->(Glte x0 y0)
  ->((Glte z0 x0)->(Glte x0 y0)->(Glte z0 y0))
  ->(NGgte x0 z0)
```

The first subgoal says that the subcomponents of `Glte` as per its definition ($\forall x_l, \neg(x_l \geq z)$ and $\forall z_r, \neg(x \geq z_r)$) imply `Glte`, and the second subgoal is the reciprocate (`Glte` implies its components). These subgoals are mostly Coq technicalities and term rewriting. The third and fourth subgoals are more interesting (mathematically less trivial), they form the weak induction hypotheses.

For the first subgoals, we remove the now unused `x`, `y` and `z`, so that we can use these names in place of the synonyms Coq introduced (`x0` and `z0`), and we decompose `x` and `z`.

```
Clear x y z.
Intros x z.
Case x.
Intros xLI xRI xLf xRf.
Case z.
Intros zLI zRI zLf zRf.
Clear x z.
Compute.

  xLI : Type
  xRI : Type
  xLf : xLI->Game
  xRf : xRI->Game
  zLI : Type
```

```
  zRI : Type
  zLf : zLI->Game
  zRf : zRI->Game
  ============================
   ((xli:xLI)(NGgte (xLf xli) (Game_cons zLI zRI zLf zRf)))
   ->((zri:zRI)(NGgte (Game_cons xLI xRI xLf xRf) (zRf zri)))
   ->(Glte (Game_cons xLI xRI xLf xRf) (Game_cons zLI zRI zLf zRf))
```

The `Compute` tactic tells Coq to normalise (simplify) the goal, e.g. replacing `(GLeftIndex (Game_cons xLI xRI xLf xRf))` by `xLI`. We pull the components of `Glte` up as hypotheses, and prove `Glte` by its only constructor, after which we have as goals only hypotheses.

```
Intros Hxl Hzr.
Exists.
Exact Hxl.
Exact Hzr.
```

The second subgoal now. Again, we remove the unused `x`, `y` and `z` and decompose `x` and `z`. We then pull `Glte x z` up as hypothesis.

```
Clear x y z.
Intros x z.
Case x.
Intros xLI xRI xLf xRf.
Case z.
Intros zLI zRI zLf zRf.
Clear x z.
Compute.
LetTac x:=(Game_cons xLI xRI xLf xRf).
LetTac z:=(Game_cons zLI zRI zLf zRf).
Intros xz.
```

```
  xLI : Type
  xRI : Type
  xLf : xLI->Game
  xRf : xRI->Game
  zLI : Type
  zRI : Type
  zLf : zLI->Game
  zRf : zRI->Game
  x := (Game_cons xLI xRI xLf xRf) : Game
  z := (Game_cons zLI zRI zLf zRf) : Game
  xz : (Glte x z)
  ============================
   ((xli:xLI)(NGgte (xLf xli) z)) /\ ((zri:zRI)(NGgte x (zRf zri)))
```

We have to decompose xz into its parts

```
Simple Inversion xz.
Compute.
Intros xl zr.

  x0 := (Game_cons xLI0 xRI0 xLf0 xRf0) : Game
  y := (Game_cons yLI yRI yLf yRf) : Game
  H1 : x0==x
  H2 : y==z
  xl : (l:xLI0)(NGgte (xLf0 l) (Game_cons yLI yRI yLf yRf))
  zr : (r:yRI)(NGgte (Game_cons xLI0 xRI0 xLf0 xRf0) (yRf r))
  ============================
   ((xli:xLI)(NGgte (xLf xli) (Game_cons zLI zRI zLf zRf)))
   /\ ((zri:zRI)(NGgte (Game_cons xLI xRI xLf xRf) (zRf zri)))
```

To get better readability, we fold a few definitions (replace the expanded version by the name)

```
Fold z x.
Fold y in xl.
Fold x0 in zr.

  xl : (l:xLI0)(NGgte (xLf0 l) y)
  zr : (r:yRI)(NGgte x0 (yRf r))
  ============================
   ((xli:xLI)(NGgte (xLf xli) z)) /\ ((zri:zRI)(NGgte x (zRf zri)))
```

We split the proof of the conjunction into the proof of its children.

```
Split.

  ============================
   (xli:xLI)(NGgte (xLf xli) z)

subgoal 2 is:
 (zri:zRI)(NGgte x (zRf zri))
```

Again new synonyms have been introduced for existing names, which we have to rewrite to.

```
Inversion H1.
Compute in H0; Compute in H3; Compute in H4; Compute in H5.
Dependent Rewrite <- H4; Cbv Beta Zeta Iota Delta -[y].

  H0 : xLI0==xLI
  H3 : xRI0==xRI
  H4 : (existT Type [LI:Type]LI->Game xLI0 xLf0)
```

```
      ==(existT Type [LI:Type]LI->Game xLI xLf)
  H5 : (existT Type [RI:Type]RI->Game xRI0 xRf0)
      ==(existT Type [RI:Type]RI->Game xRI xRf)
  ============================
   (xli:xLI0)(NGgte (xLf0 xli) y)
```

This goal is now exactly `xl`. The second part of the conjunction goes very similarly (not shown here).

This leaves us with the two weak inductions. We'll do only the proof of the first one (third subgoal above).

```
 x : Game
  y : Game
  z : Game
  ============================
   (x0,y0,z0:Game)
    (NGgte x0 y0)
    ->(Glte y0 z0)
    ->((Glte y0 z0)->(Glte z0 x0)->(Glte y0 x0))
    ->(NGgte x0 z0)
```

We get rid of unused names, decompose `x`, `y` and `z`.

```
Clear x y z.
Intros x y z.
Case x.
Intros xLI xRI xLf xRf.
Case y.
Intros yLI yRI yLf yRf.
Case z.
Intros zLI zRI zLf zRf.
Clear x y z.
Compute.
LetTac x:=(Game_cons xLI xRI xLf xRf).
LetTac y:=(Game_cons yLI yRI yLf yRf).
LetTac z:=(Game_cons zLI zRI zLf zRf).

  xLI : Type
  xRI : Type
  xLf : xLI->Game
  xRf : xRI->Game
  yLI : Type
  yRI : Type
  yLf : yLI->Game
  yRf : yRI->Game
  zLI : Type
```

```
  zRI : Type
  zLf : zLI->Game
  zRf : zRI->Game
  x := (Game_cons xLI xRI xLf xRf) : Game
  y := (Game_cons yLI yRI yLf yRf) : Game
  z := (Game_cons zLI zRI zLf zRf) : Game
  ============================
   (NGgte x y)
   ->(Glte y z)
   ->((Glte y z)->(Glte z x)->(Glte y x))
   ->(NGgte x z)
```

We pull the premises into the context as hypotheses, and replace `NGgte` by `~ Glte`

```
Intros xy yz Trans.
Apply nGlteNGgte.
```

```
  xy : (NGgte x y)
  yz : (Glte y z)
  Trans : (Glte y z)->(Glte z x)->(Glte y x)
  ============================
   ~(Glte z x)
```

We now unfold the definition of negation and pull `Glte` up as hypothesis.

```
Unfold not; Intro zx.
```

```
  zx : (Glte z x)
  ============================
   False
```

Absurdity is derived by proving `(Glte y x)` and its negation.

```
Absurd (Glte y x).
```

```
  ============================
   ~(Glte y x)

subgoal 2 is:
 (Glte y x)
```

We transform `~Glte` into `NGgte`, which brings us to an hypothesis.

```
Apply NGgteIsNGgte.
Assumption.
```

`(Glte y x)` is proven by the `Transitivity` hypothesis, the premises of `Trans` are present as hypotheses.

```
Apply Trans; Assumption.
```

This concludes the proof of transitivity of `Glte`.

Transitivity by Conway

It is interesting to compare the proof of transitivity in Coq and the one given by Conway in [Con01] chapter 1, theorem 1, page 16. A full reproduction is in figure 2.0. It seems quite

> THEOREM 1. *If $x \geqslant y$ and $y \geqslant z$, then $x \geqslant z$.*
>
> *Proof.* Since $x \geqslant y$, we cannot have $x^R \leqslant y$, and so by induction we cannot have $x^R \leqslant z$. Similarly we cannot have $x \leqslant z^L$, and so we must have $x \geqslant z$.

Figure 2.0: Proof of transitivity of the order in [Con01]

remarkable that such a short proof explodes to so long a proof in Coq. That is until you try to remove the "similarly" from Conway's proof and say exactly what induction is used. Let's try to gradually formalise (put all the details in) Conway's proof.

Level 1 Proof by induction. We assume $x \geq y$ and $y \geq z$, and we prove $x \geq z$ by proving $\neg(x_r \leq z)$ and $\neg(x \leq z_l)$.

- Proof of $\neg(x_r \leq z)$: we assume $(x_r \leq z)$ and derive a contradiction, namely $(x_r \leq y) \wedge \neg(x_r \leq y)$. As $x \geq y$, by definition of $\geq$, we have $\neg(x_r \leq y)$. On the other hand, since $x_r \leq z$ and $z \leq y$, by induction hypothesis we have $x_r \leq y$. Contradiction, thus the assumption $x_r \leq z$ cannot hold.
- Proof of $\neg(x \leq z_l)$: similar.

This level 1 of formality is just a more verbose and structured version of the proof by Conway. Now, let's remove the "similar".

Level 2 Proof by induction. We assume $x \geq y$ and $y \geq z$, and we prove $x \geq z$ by proving $\neg(x_r \leq z)$ and $\neg(x \leq z_l)$.

- Proof of $\neg(x_r \leq z)$: we assume $(x_r \leq z)$ and derive a contradiction, namely $(x_r \leq y) \wedge \neg(x_r \leq y)$. As $x \geq y$, by definition of $\geq$, we have $\neg(x_r \leq y)$. On the other hand, since $x_r \leq z$ and $z \leq y$, by induction hypothesis we have $x_r \leq y$. Contradiction, thus the assumption $x_r \leq z$ cannot hold.
- Proof of $\neg(x \leq z_l)$: we assume $(x \leq z_l)$ and derive a contradiction, namely $(y \leq z_l) \wedge \neg(y \leq z_l)$. As $y \geq z$, by definition of $\geq$, we have $\neg(y \leq z_l)$. On the other hand, since $x \leq z_l$ and $y \leq x$, by induction hypothesis we have $y \leq z_l$. Contradiction, thus the assumption $x \leq z_l$ cannot hold.

In our proof, we have names like z_l, x_r that crop up without being introduced by any hypothesis nor quantification. Let's fix this.

Level 3 Proof by induction. We assume $x \geq y$ and $y \geq z$, and we prove $x \geq z$ by proving $\forall x_r \in R_x, \neg(x_r \leq z)$ and $\forall z_l \in L_z, \neg(x \leq z_l)$.

- Proof of $\forall x_r \in R_z, \neg(x_r \leq z)$: we assume $\exists x_r \in R_x, (x_r \leq z)$ (the negation of our goal) and derive a contradiction, namely $\exists x_r \in R_x, (x_r \leq y) \wedge \neg(x_r \leq y)$. Let $x_r \in R_x$ such that $x_r \leq z$. As $x \geq y$, by definition of $\geq$, we have $\forall x_{r_0} \in R_x, \neg(x_{r_0} \leq y)$ and in particular $\neg(x_r \leq y)$. On the other hand, since $x_r \leq z$ and $z \leq y$, by induction hypothesis we have $x_r \leq y$. Contradiction, thus the assumption $\exists x_r \in R_x, x_r \leq z$ cannot hold.

- Proof of $\forall z_l \in L_z, \neg(x \leq z_l)$: we assume $\exists z_l \in L_z, (x \leq z_l)$ and derive a contradiction, namely $\exists z_l \in L_z, (y \leq z_l) \wedge \neg(y \leq z_l)$. Let $z_l \in L_z$ such that $x \leq z_l$. As $y \geq z$, by definition of $\geq$, we have $\forall z_{l_0} \in L_z, \neg(y \leq z_{l_0})$ and in particular $y \leq z_l$. On the other hand, since $x \leq z_l$ and $y \leq x$, by induction hypothesis we have $y \leq z_l$. Contradiction, thus the assumption $\exists z_l \in L_z, x \leq z_l$ cannot hold.

Now, we are trying to prove a property of $\geq$, but we are reasoning on the symbol $\leq$ when we use an induction hypothesis. So what is that? Mutual induction between $\leq$ and $\geq$? Hmm... Maybe we can use the link between $\geq$ and $\leq$ to use only $\geq$.

Level 4 Proof by induction. We assume $x \geq y$ and $y \geq z$, and we prove $x \geq z$ by proving $\forall x_r \in R_x, \neg(z \geq x_r)$ and $\forall z_l \in L_z, \neg(z_l \geq x)$.

- Proof of $\forall x_r \in R_z, \neg(z \geq x_r)$: we assume $\exists x_r \in R_x, (z \geq x_r)$ (the negation of our goal) and derive a contradiction, namely $\exists x_r \in R_x, (y \geq x_r) \wedge \neg(y \geq x_r)$. Let $x_r \in R_x$ such that $z \geq x_r$. As $x \geq y$, by definition of $\geq$, we have $\forall x_{r_0} \in R_x, \neg(y \geq x_{r_0})$ and in particular $\neg(y \geq x_r)$. On the other hand, since $z \geq x_r$ and $y \geq z$, by induction hypothesis we have $y \geq x_r$. Contradiction, thus the assumption $\exists x_r \in R_x, z \geq x_r$ cannot hold.

- Proof of $\forall z_l \in L_z, \neg(z_l \geq x)$: we assume $\exists z_l \in L_z, (z_l \geq x)$ and derive a contradiction, namely $\exists z_l \in L_z, (z_l \geq y) \wedge \neg(z_l \geq y)$. Let $z_l \in L_z$ such that $z_l \geq x$. As $y \geq z$, by definition of $\geq$, we have $\forall z_{l_0} \in L_z, \neg(z_{l_0} \geq y)$ and in particular $z_l \geq y$. On the other hand, since $z_l \geq x$ and $x \geq y$, by induction hypothesis we have $z_l \geq y$. Contradiction, thus the assumption $\exists z_l \in L_z, z_l \geq x$ cannot hold.

Now, let's fill in what induction hypotheses we are using.

Level 5 Proof by induction. We assume $x \geq y$ and $y \geq z$, and we prove $x \geq z$ by proving $\forall x_r \in R_x, \neg(z \geq x_r)$ and $\forall z_l \in L_z, \neg(z_l \geq x)$.

- Proof of $\forall x_r \in R_z, \neg(z \geq x_r)$: we assume $\exists x_r \in R_x, (z \geq x_r)$ (the negation of our goal) and derive a contradiction, namely $\exists x_r \in R_x, (y \geq x_r) \wedge \neg(y \geq x_r)$. Let $x_r \in R_x$ such that $z \geq x_r$. As $x \geq y$, by definition of $\geq$, we have $\forall x_{r_0} \in R_x, \neg(y \geq x_{r_0})$ and in particular $\neg(y \geq x_r)$. On the other hand, since $z \geq x_r$ and $y \geq z$, by the induction hypothesis $\forall x_{r_0} \in R_x, y \geq z \rightarrow z \geq x_{r_0} \rightarrow y \geq x_{r_0}$ we have $y \geq x_r$. Contradiction, thus the assumption $\exists x_r \in R_x, z \geq x_r$ cannot hold.

- Proof of $\forall z_l \in L_z, \neg(z_l \geq x)$: we assume $\exists z_l \in L_z, (z_l \geq x)$ and derive a contradiction, namely $\exists z_l \in L_z, (z_l \geq y) \wedge \neg(z_l \geq y)$. Let $z_l \in L_z$ such that $z_l \geq x$. As $y \geq z$, by definition of $\geq$, we have $\forall z_{l_0} \in L_z, \neg(z_{l_0} \geq y)$ and in particular $z_l \geq y$. On the other hand, since $z_l \geq x$ and $x \geq y$, by the induction hypothesis $\forall z_{l_0} \in L_z, z_{l_0} \geq x \rightarrow x \geq y \rightarrow z_{l_0} \geq y$ we have $z_l \geq y$. Contradiction, thus the assumption $\exists z_l \in L_z, z_l \geq x$ cannot hold.

But we cannot parachute induction hypotheses like that in the middle of a proof! We must say up front what induction we do, this gives us induction hypotheses, and we use them in the proof. The induction hypotheses used have the references to x, y, z or their lefts and rights all shuffled up. This looks like the kind of inductions managed by conjecture 2.1.1, but Conway has given exactly zero explanation on what induction he is using.

More exactly, in chapter 0, page 5 he writes (emphasis mine):

> In general, when we wish to establish a proposition $P(x)$ for all numbers x, we will prove it inductively by deducing $P(x)$ from the truth of all the propositions $P(x^L)$ and $P(x^R)$. (...) When proving propositions $P(x, y)$ involving two variables we may use double induction, deducing $P(x, y)$ from the truth of all propositions of the form $P(x^L, y)$, $P(x^R, y)$, $P(x, y^L)$, $P(x, y^R)$ (and, if necessary, $P(x^L, y^L)$, $P(x^L, y^R)$, $P(x^R, y^R)$, $P(x^R, y^L)$). *Such multiple inductions can be justified in the usual way in terms of repeated single inductions.*

and in in the appendix to part zero, page 64, he writes:

> In this appendix, we informally discuss the formalisation of our theory, with particular regard to the nature of the inductions involved.
>
> (...) We suspect that many readers would have felt happier had we described all our inductive arguments in terms of birthdays. (...) The feeling that this sort of treatment adds to the precision of an inductive argument is much too common (...) the notion of birthday is completely irrelevant (...) *all that is needed to justify the induction is the principle*:
>
> "If P is some proposition that holds for x whenever it holds for all x_l and x_r, then P holds universally".
>
> (...) The general induction principle above has for its counterpart in the Zermelo-Fraenkel set theory ZF the so-called axiom of restriction, or foundation, which can be stated in the form:
>
> "If P is some proposition that holds for a set x whenever it holds for all members of x, then P holds for every set."

This approach indeed is much more aesthetic than birth dates based induction, because it doesn't pre-suppose that the ordinals are already constructed, which would defeat the nice feature of surreal numbers to have ordinals and reals in the same nice, simple definition. That idea is essentially captured by our $\lessdot$ well-founded order.

To get back to our main concern, he (informally) claims that all his inductions are justified by this principle, and thus in particular are inductions on a *single* variable (possibly repeated or nested). Yet, this property here clearly doesn't use nested induction. Using something as

general as conjecture 2.1.1 is thus not formalising his proof, but making a *different* proof. So, let's backtrack to when we introduced this shuffling of variables, i.e. when we removed occurrences of $\leq$, and try to go the "mutual induction between $\leq$ and $\geq$ route".

Level 5bis We prove transitivity of $\leq$ and transitivity of $\geq$ by mutual induction. We'll write only the proof for $\geq$ for now, and if it turns out to be OK, then we'll write the proof for $\leq$, or use a symmetry argument or some such.

We assume $x \geq y$ and $y \geq z$, and we prove $x \geq z$ by proving $\forall x_r \in R_x, \neg(x_r \leq z)$ and $\forall z_l \in L_z, \neg(z \leq z_l)$.

- Proof of $\forall x_r \in R_z, \neg(x_r \leq z)$: we assume $\exists x_r \in R_x, (x_r \leq z)$ (the negation of our goal) and derive a contradiction, namely $\exists x_r \in R_x, (x_r \leq y) \wedge \neg(x_r \leq y)$. Let $x_r \in R_x$ such that $x_r \leq z$. As $x \geq y$, by definition of $\geq$, we have $\forall x_{r_0} \in R_x, \neg(x_{r_0} \leq y)$ and in particular $\neg(x_r \leq y)$. On the other hand, since $x_r \leq z$ and $z \leq y$, by the induction hypothesis $\forall x_{r_0} \in R_x, x_{r_0} \leq z \to z \leq y \to x_{r_0} \leq y$ we have $x_r \leq y$. Contradiction, thus the assumption $\exists x_r \in R_x, x_r \leq z$ cannot hold.

- Proof of $\forall z_l \in L_z, \neg(x \leq z_l)$: we assume $\exists z_l \in L_z, (x \leq z_l)$ and derive a contradiction, namely $\exists z_l \in L_z, (y \leq z_l) \wedge \neg(y \leq z_l)$. Let $z_l \in L_z$ such that $x \leq z_l$. As $y \geq z$, by definition of $\geq$, we have $\forall z_{l_0} \in L_z, \neg(y \leq z_{l_0})$ and in particular $y \leq z_l$. On the other hand, since $x \leq z_l$ and $y \leq x$, by the induction hypothesis $\forall z_{l_0} \in L_z, y \leq x \to x \leq z_{l_0} \to y \leq z_{l_0}$ we have $y \leq z_l$. Contradiction, thus the assumption $\exists z_l \in L_z, x \leq z_l$ cannot hold.

The induction hypotheses used now at least have the right of x (respectively the left of z) in the right place: induction on x gives us the induction hypothesis $\forall x_r \in R_x, \forall (x_0, x_1) \in G^2, x_r \leq x_0 \to x_0 \leq x_1 \to x_r \leq x_1$ (and the same with a left of x, but we don't use it), which indeed is a generalisation of $\forall x_{r_0} \in R_x, x_{r_0} \leq z \to z \leq y \to x_{r_0} \leq y$, and induction on z gives us $\forall z_l \in L_z, \forall (x_0, x_1) \in G^2, x_0 \leq x_1 \to x_1 \leq z_l \to x_0 \leq z_l$, which indeed is a generalisation of $\forall z_l \in L_z, y \leq z \to z \leq z_l \to y \leq z_l$.

Aha, but how can we mix induction on x and on z?

Let P be defined by $P(x, y, z) \overset{\text{def}}{\leftrightarrow} x \leq y \to y \leq z \to z \leq z$. We can do first induction on x, then induction on z, i.e. prove $\forall y, \forall z, P(x, y, z)$ by induction on x (generating the IH $\forall x_r \in R_x, \forall (x_0, x_1) \in G^2, P(x_r, x_0, x_1)$) and prove $\forall y, P(x, y, z)$ for one *fixed* x by induction on z, generating the IH $\forall z_l \in L_z, \forall y \in G, P(x, y, z_l)$. But the latter is not a generalisation of what we need.

So we need to somehow do induction on x and z at the same time. Maybe because we go *right* for x but *left* for z, this will be valid? First, I dare you to justify this with the induction principle of Conway, second, the answer is "no, this won't be valid", and we prove that now.

We suppose that the induction scheme (all non-detailed quantifications are over G)

$\forall P,$
$(\forall x, \forall z, (\forall x_r \in R_x, \forall y, \forall z, P(x_r, y, z)) \to (\forall z_l \in L_z, \forall x, \forall y, P(x, y, z_l)) \to \forall y, P(x, y, z)) \to$
$(\forall x, \forall y, \forall z, P(x, y, z))$

is valid and derive a contradiction.

The scheme being valid is equivalent to the relation $\lll$, defined by[14]

$\lll$ is the smallest relation such that

$$\forall x, \forall x_r \in R_x, \forall y_1, \forall y_2, \forall z_1, \forall z_2, (x_r, y_1, z_1) \lll (x, y_2, z_2)$$
$$\forall z, \forall z_l \in L_z, \forall y_1, \forall y_2, \forall x_1, \forall x_2, (x_1, y_1, z_l) \lll (x_2, y_2, z)$$
$\lll$ is transitive

being a well-founded strict order. But it is neither well-founded, neither a strict order: let $t_1 \ggg t_2 \overset{\text{def}}{\leftrightarrow} t_2 \lll t_1$, let x, y, z be games. Then $(x, y, z) \ggg (x_r, y, \{z|\}) \ggg (x, y, \{z|\}_l) \equiv (x, y, z)$ and by transitivity $(x, y, z) \lll (x, y, z)$.

This concludes the proof that induction on x and z at the same time is not valid, and thus a call to an instance of conjecture 2.1.1 looks unavoidable.

Conclusion I hope I now have convinced you that the proof of Conway was written in a misleadingly simple way, that was sorely missing an additional explanation, which Coq rightly forces me to provide in the form of the `Roll3Induction` theorem.

2.2 Equivalence relations

Having fully defined the order permits us to define equality. We also define identicality.

2.2.0 Equality

Definition of equality is straightforward, it is simply being smaller and greater than:

```
Definition Geq:Game->Game->Prop := [x,y:Game](Glte x y) /\ (Glte y x).
```

We now prove that equality is an equivalence relation: symmetric, transitive, reflexive.

Symmetry

Symmetry essentially derives from commutativity of conjunction.

```
Theorem Geq_symmetric: (x,y:Game) (Geq x y) -> (Geq y x).
Proof.
Intros.

  x : Game
  y : Game
  H : (Geq x y)
  ============================
   (Geq y x)
```

[14]Less formally, it is the relation "$t_1 \lll t_2$ if and only if $P(t_1)$ is used in the proof of $P(t_2)$ by the scheme", where t_1 and t_2 are game triples.

All we have to do is decompose H into `Glte x y` and `Glte y x`, split the goal into `Glte y x` and `Glte x y` and notice all goals are hypotheses[15]:

```
Inversion H; Split; Assumption.
Qed.
```

Transitivity

Derives from transitivity of `Glte`.

```
Theorem Geq_transitive: (x,y,z:Game)
             (Geq x y) -> (Geq y z) -> (Geq x z).
Proof.
Intros.
Inversion H; Inversion H0.

  x : Game
  y : Game
  z : Game
  H : (Geq x y)
  H0 : (Geq y z)
  H1 : (Glte x y)
  H2 : (Glte y x)
  H3 : (Glte y z)
  H4 : (Glte z y)
  ============================
   (Geq x z)
```

We split the goal into the two inequalities, use transitivity of the order, and all generated goals are assumptions.

```
Split; Apply Glte_transitive with y:=y; Assumption.
Qed.
```

Reflexivity

Derives from reflexivity of `Glte`: both sides of the conjunction are true by reflexivity of `Glte`.

```
Theorem Geq_reflexive: (x:Game) (Geq x x).
Proof.
Intros.
Split; Apply Glte_reflexive.
Qed.
```

[15]The semi-colon ";" means "use the following tactic on all goals generated by the previous tactic"

2.2.1 Identicality

Our model of sets in our definition of Games has many different representations for the same set. E.g. $\{0,1\}$ can be any of:

- Index set natural numbers and index function $\lambda n.(n \text{ modulo } 2)$.
- Index set natural numbers and index function $\lambda n.1 - (n \text{ modulo } 2)$.
- Index set natural numbers and index function $\lambda n.\text{if } n < 18 \text{ then } 0 \text{ else } 18$.
- Index set having two elements a and b and index function f defined by $f(a) = 0$ and $f(b) = 1$.
- Index set having two elements a and b and index function f defined by $f(a) = 1$ and $f(b) = 0$.
- Index set having three elements a, b and c and index function f defined by $f(a) = f(b) = 1$ and $f(c) = 0$.
- Index set having three elements a, b and c and index function f defined by $f(a) = f(c) = 1$ and $f(b) = 0$.

So, Conway-style identicality is really a proper (non-trivial) equivalence relation on our games. x and y are identical if and only if their right (respectively left) sets contain the same elements modulo identicality, i.e.:

$$\forall x_l \in L_x, \exists y_l \in L_y, x_l \equiv y_l$$
$$\forall x_r \in R_x, \exists y_r \in R_y, x_r \equiv y_r$$
$$\forall y_l \in L_y, \exists x_l \in L_x, y_l \equiv x_l$$
$$\forall y_r \in R_y, \exists x_r \in R_x, y_r \equiv x_r$$

In Coq, existential quantification over something of sort `Type` is `exT D P`, where `D` is the quantification domain, and `P` is of type `D -> Prop`, i.e. for every `d` of type `D`, `(P d)` is a proposition. So, the definition of identicality in Coq syntax is:

```
Inductive Gidentical:Game->Game->Prop :=
  Gidentical_cons : (x,y:Game)
    ((xli:(GLeftIndex  x)) (exT (GLeftIndex  y) [yli:(GLeftIndex  y)]
                (Gidentical (GLeftFun  x xli) (GLeftFun  y yli))))  ->
    ((xri:(GRightIndex x)) (exT (GRightIndex y) [yri:(GRightIndex y)]
                (Gidentical (GRightFun x xri) (GRightFun y yri))))  ->
    ((yli:(GLeftIndex  y)) (exT (GLeftIndex  x) [xli:(GLeftIndex  x)]
                (Gidentical (GLeftFun  y yli) (GLeftFun  x xli))))  ->
    ((yri:(GRightIndex y)) (exT (GRightIndex x) [xri:(GRightIndex x)]
                (Gidentical (GRightFun y yri) (GRightFun x xri))))  ->
    (Gidentical x y).
```

Reflexivity and symmetry of `Gidentical` have been formally proven in Coq, but are not detailed here. Reflexivity ($\forall x, x \equiv x$) is proven by induction on x, and symmetry is very like symmetry of `Geq`, essentially commutativity of conjunction.

Transitivity ($\forall(x, y, z), x \equiv y \to y \equiv z \to x \equiv z$) is more interesting: in order to prove $x \equiv z$, one needs to prove $z_l \equiv x_l$, and there is no hope a non-permuting induction will help there: the roles of x and z are permuted. There are a few ways to address this:

- Use a permuting induction. Relatively lot of work: one must express the right permuting induction scheme, prove it correct and prove it applies to the transitivity of `Gidentical`.
- Change the definition of `Gidentical` not to swap its arguments around, i.e. to:

```
Inductive Gidentical:Game->Game->Prop :=
  Gidentical_cons : (x,y:Game)
    ((xli:(GLeftIndex  x)) (exT (GLeftIndex  y) [yli:(GLeftIndex  y)]
                 (Gidentical (GLeftFun  x xli) (GLeftFun  y yli))))  ->
    ((xri:(GRightIndex x)) (exT (GRightIndex y) [yri:(GRightIndex y)]
                 (Gidentical (GRightFun x xri) (GRightFun y yri))))  ->
    ((yli:(GLeftIndex  y)) (exT (GLeftIndex  x) [xli:(GLeftIndex  x)]
                 (Gidentical (GLeftFun  x xli) (GLeftFun  y yli))))  ->
    ((yri:(GRightIndex y)) (exT (GRightIndex x) [xri:(GRightIndex x)]
                 (Gidentical (GRightFun x xri) (GRightFun y yri))))  ->
    (Gidentical x y).
```

  This helps tremendously in the proof of transitivity, but it breaks the symmetry of the definition and thus complicates reflexivity and symmetry somewhat (not much).

- Notice we already have proven symmetry and use it to transform the goal `Glte zl xl` into `Glte xl zl`. This is the solution adopted.

Except for that issue, transitivity is straightforward by induction on x.

2.3 Numbers

Having defined the order, we now can define the predicate "Is a number":

```
Inductive IsNumber : Game->Prop :=
  IsNumber_cons : (x:Game)
                  ((xli:(GLeftIndex  x)) (IsNumber (GLeftFun  x xli))) ->
                  ((xri:(GRightIndex x)) (IsNumber (GRightFun x xri))) ->
                  ((xli:(GLeftIndex x))(xri:(GRightIndex x))
                           (NGgte (GLeftFun x xli) (GRightFun x xri))) ->
                  (IsNumber x)
```

Chapter 3

Ordered Field Structure

We now have defined a model of surreal numbers in Coq, and defined the order on it, but not yet any operation. We will first define addition, then multiplication.

3.0 Definition of Addition

The definition of addition:

$$x + y \stackrel{\text{def}}{=} \{x_l + y, x + y_l | x_r + y, x + y_r\}$$

brings up a problem already discussed page 27 for the proof constructor of the IS2 induction scheme: Neither x, nor y is a decreasing argument in this recursive definition. The same solution, "swallowing" the recursion on y into an auxiliary function is used:

```
Fixpoint GPlus [x:Game] : Game->Game := [y:Game]
 Cases x of
   (Game_cons xLI xRI xLf xRf) =>
   Cases y of
     (Game_cons yLI yRI yLf yRf) =>
      let GPlusAux =
      Fix GPlusAux { GPlusAux [z:Game] : Game :=
      (* (GPlusAux) == (GPlus x) *)
       Cases z of
         (Game_cons zLI zRI zLf zRf) =>
         (Game_cons
           (sumT xLI zLI) (sumT xRI zRI)
           (map_on_sumT xLI zLI Game
             ([xli:xLI] (GPlus (xLf xli) z))
             ([zli:zLI] (GPlusAux (zLf zli)))
           )
           (map_on_sumT xRI zRI Game
             ([xri:xRI] (GPlus (xRf xri) z))
             ([zri:zRI] (GPlusAux (zRf zri)))
```

```
          )
        )
      end
    }
     in
    (Game_cons
      (sumT xLI yLI) (sumT xRI yRI)
      (map_on_sumT xLI yLI Game
        ([xli:xLI] (GPlus (xLf xli) y))
        ([yli:yLI] (GPlusAux (yLf yli)))
      )
      (map_on_sumT xRI yRI Game
        ([xri:xRI] (GPlus (xRf xri) y))
        ([yri:yRI] (GPlusAux (yRf yri)))
      )
    )
  end
end
.
```

sumT is the disjoint union of types of sort Type (hence the T) and map_on_sumT constructs a function on a sum type from functions over the components of the sum. Strangely enough, these are not defined in the Coq base library, and I thus defined them myself:

```
Inductive sumT [A,B:Type] : Type :=
   inlT : A -> (sumT A B)
 | inrT : B -> (sumT A B).

Definition map_on_sumT := [A,B,C:Type] [fa:A->C;fb:B->C][x:(sumT A B)]
 Cases x of
   (inlT e) => (fa e)
   |(inrT e) => (fb e)
 end.
```

3.1 Anti-Game

The definition of the anti-game (symmetric with respect to addition) is just a question of putting it in Coq syntax:

```
Fixpoint AntiGame [g:Game] : Game :=
 Cases g of
   (Game_cons LI RI Lf Rf) => (Game_cons RI LI ([i:RI] (AntiGame (Rf i)))
                                               ([i:LI] (AntiGame (Lf i))) )
 end.
```

3.2 $(G,+)$ is a commutative group

We now prove that this addition provides G with a commutative group structure.

3.2.0 Neutral

Definition of 0

A group has a neutral element, traditionally called *zero* and denoted 0 for additive groups. Our zero is $\{|\}$. We thus need an empty type to index its left and right. Falsity is by its very definition empty (there is no proof of it), so we'll use it. We use Coq's proof construction tactics to help us contruct the corresponding indexing function:

```
Lemma EmptyGameSet : False -> Game.
Proof.
Intro.
Contradiction.
Defined.
```

And now we can define zero:

```
Definition Zero := (Game_cons False False EmptyGameSet EmptyGameSet).
```

0 is the neutral

We now prove that `Zero` is indeed the neutral, by first proving

$$\forall x \in G, x \leq x + 0$$

and then

$$\forall x \in G, x + 0 \leq x$$

We do only the first one here, both go by induction on `x`:

```
Lemma GltexGPlusxZero: (x:Game)(Glte x (GPlus x Zero)).
Proof.
Intros.
NewInduction x.
Rename RI into xRI.
Rename LI into xLI.
Rename g into xLf.
Rename H into xLIH.
Intros xRf xRIH.
LetTac x:=(Game_cons xLI xRI xLf xRf).

  xLI : Type
  xRI : Type
  xLf : xLI->Game
```

```
  xLIH : (T:xLI)(Glte (xLf T) (GPlus (xLf T) Zero))
  xRf : xRI->Game
  xRIH : (l:xRI)(Glte (xRf l) (GPlus (xRf l) Zero))
  x := (Game_cons xLI xRI xLf xRf) : Game
  ============================
   (Glte x (GPlus x Zero))
```

We now need to compute `GPlus x Zero`, at least to the point where we get a term (an expression) whose root (when the term is seen as a tree) is `Game_cons`, because the only means we have to prove an inequality is by using its constructor, and its constructor, as it is defined, can only be used to prove things of the form `(Glte (Game_cons ...) (Game_cons ...))`[0]. Coq can do this automatically, but it gives a rather large term (9 pages if I would include it here in full).

```
Unfold x Zero GPlus map_on_sumT; Cbv Beta Iota.
```

```
  ============================
   (Glte (Game_cons xLI xRI xLf xRf)
     (Game_cons (sumT xLI False) (sumT xRI False)
       [x0:(sumT xLI False)]
      ...))
```

Now we can use `Glte`'s only constructor. The resulting goals are again manually shortened, mainly be removing the in-line definitions of `GPlus` and `GPlusAux`, that are pasted in the term.

```
Exists.
```

```
  ============================
   (l:xLI)
    [xl:=(xLf l)]
     (NGgte xl
       (Game_cons (sumT xLI False) (sumT xRI False)
         ...))

subgoal 2 is:
 (r:(sumT xRI False))
  (NGgte (Game_cons xLI xRI xLf xRf)
    Cases r of
      (inlT e) =>
       (Fix GPlus {...} (xRf e)
         (Game_cons False False EmptyGameSet EmptyGameSet))
    | (inrT e) =>
       (Fix GPlusAux
```

[0]See section 4.1.4

```
        {...} (EmptyGameSet e))
    end)
```

Intros, and then we prove NGgte by going in the left of its righ hand-side , which is done be the second constructor.

```
Intros xli xl.
Constructor 2.
```

```
  xli : xLI
  xl := (xLf xli) : Game
  ============================
   (EXT l:(sumT xLI False) |
        (Glte xl
          Cases l of
            (inlT e) =>
             (Fix GPlus {...} (xLf e)
               (Game_cons False False EmptyGameSet EmptyGameSet))
          | (inrT e) =>
             (Fix GPlusAux {...} (EmptyGameSet e))
          end))
```

So, we must find an l such that:

- either l is a left of x and $x_l \leq l + 0$
- or l is a left of 0 and $x_l \leq x + l$

  One cannot see in this shortened view that this GPlusAux is GPlus x, but it can be seen in the definition I replaced by "...".

Well, we'd better take a left of x, because Zero has no left. So, what left of x added to 0 is greater or equal than x_l? By induction, x_l itself fills the bill.

```
Exists (inlT xLI False xli).
Exact (xLIH xli).
```

This leaves us to deal with the second goal we generated:

```
 (r:(sumT xRI False))
  (NGgte (Game_cons xLI xRI xLf xRf)
    Cases r of
      (inlT e) =>
       (Fix GPlus {...} (xRf e)
         (Game_cons False False EmptyGameSet EmptyGameSet))
    | (inrT e) =>
       (Fix GPlusAux
         {...} (EmptyGameSet e))
    end)
```

We have an r that is either a right of x or a right of Zero. We separate the two cases, and the second one is absurd.

```
Intro r; Case r.
2: Intros; Contradiction.
```

```
  r : (sumT xRI False)
  ============================
   (x0:xRI)
    (NGgte (Game_cons xLI xRI xLf xRf)
      (Fix GPlus {} (xRf x0)
        (Game_cons False False EmptyGameSet EmptyGameSet)))
```

We prove NGgte by digging into its first argument, this is the first constructor.

```
  xri : xRI
  ============================
   (EXT r0:xRI |
        (Glte (xRf r0)
          (Fix GPlus {} (xRf xri)
            (Game_cons False False EmptyGameSet EmptyGameSet))))
```

We must find a right of x such that it is less than or equal to $x_r + 0$. Again, by induction, x_r itself fills the bill.

```
Exists xri.
Exact (xRIH xri).
Qed.
```

By commutativity of addition (which we prove later in this report, but before in the Coq code), we obtain

$$\forall x \in G, x \leq 0 + x$$
$$\forall x \in G, 0 + x \leq x$$

3.2.1 Commutativity

We prove

$$\forall (x, y) \in G^2, (x + y) \leq (y + x)$$

and then

$$\forall (x, y) \in G^2, (x + y) = (y + x)$$

follows easily.

The proof goes by induction on x and then y.

```
Theorem GPlusCommutative_one: (x,y:Game)(Glte (GPlus x y) (GPlus y x)).
Intros x.
NewInduction x.
```

```
Rename RI into xRI.
Rename LI into xLI.
Rename g into xLf.
Rename H into xLIH.
Intros xRf xRIH.
LetTac x:=(Game_cons xLI xRI xLf xRf).
Intros y.
NewInduction y.
Rename RI into yRI.
Rename LI into yLI.
Rename g into yLf.
Rename H into yLIH.
Intros yRf yRIH.
LetTac y:=(Game_cons yLI yRI yLf yRf).

  xLI : Type
  xRI : Type
  xLf : xLI->Game
  xLIH : (T:xLI; y:Game)(Glte (GPlus (xLf T) y) (GPlus y (xLf T)))
  xRf : xRI->Game
  xRIH : (l:xRI; y:Game)(Glte (GPlus (xRf l) y) (GPlus y (xRf l)))
  x := (Game_cons xLI xRI xLf xRf) : Game
  yLI : Type
  yRI : Type
  yLf : yLI->Game
  yLIH : (T:yLI)(Glte (GPlus x (yLf T)) (GPlus (yLf T) x))
  yRf : yRI->Game
  yRIH : (l:yRI)(Glte (GPlus x (yRf l)) (GPlus (yRf l) x))
  y := (Game_cons yLI yRI yLf yRf) : Game
  ============================
   (Glte (GPlus x y) (GPlus y x))
```

We must compute `GPlus x y` and `GPlus y x` to have `Game_cons` as root, and then we can use `Glte`'s only constructor.

```
Unfold x y; Cbv Beta Iota Delta [GPlus]. Exists.

  ============================
   (l:(sumT xLI yLI))
    [xl:=(map_on_sumT xLI yLI Game
           [xli:xLI]
            (Fix GPlus {...} (xLf xli) (Game_cons yLI yRI yLf yRf))
           [yli:yLI]
            (Fix GPlusAux {...} (yLf yli)) l)]
     (NGgte xl
```

```
      (Game_cons (sumT yLI xLI) (sumT yRI xRI)
        (map_on_sumT yLI xLI Game
          [xli:yLI]
           (Fix GPlus {...} (yLf xli) (Game_cons xLI xRI xLf xRf))
          [yli:xLI]
           (Fix GPlusAux {...} (xLf yli)))
        (map_on_sumT yRI xRI Game
          [xri:yRI]
           (Fix GPlus {...} (yRf xri) (Game_cons xLI xRI xLf xRf))
          [yri:xRI]
           (Fix GPlusAux {...} (xRf yri)))))
```

```
subgoal 2 is:
...
```

1 is a left index of x or a left index of y, we thus call it xyli, and do case distinction on it. This gives us a left of x (repectively y), which we call xl (respectively yl). We then use NGgte's second constructor, and prove the existential quantification with xl (respectively yl).

```
Intro xyli.
Case xyli; [Intros xli xl | Intros yli yl]; Constructor 2;
           [ Exists (inrT yLI xLI xli) | Exists (inlT yLI xLI yli)];
           Compute.
```

```
  xyli : (sumT xLI yLI)
  xli : xLI
  xl := (map_on_sumT xLI yLI Game
          [xli:xLI]
           (Fix GPlus {...} (xLf xli) (Game_cons yLI yRI yLf yRf))
          [yli:yLI]
           (Fix GPlusAux {...} (yLf yli)) (inlT xLI yLI xli)) : Game
  ============================
   (Glte
     (Fix GPlus {...} (xLf xli) (Game_cons yLI yRI yLf yRf))
     (Fix GPlusAux {...} (xLf xli)))
```

We now replace the GPlusAux with GPlus y.

```
Apply Glte_transitive with y:=(GPlus y (xLf xli)).
```

```
  ============================
   (Glte
     (Fix GPlus {...} (xLf xli) (Game_cons yLI yRI yLf yRf))
     (GPlus y (xLf xli)))
```

```
subgoal 2 is:
 (Glte (GPlus y (xLf xli))
   (Fix GPlusAux {...} (xLf xli)))
```

The first goal is an instatiation of the xLIH induction hypothesis, and the second one is equality[1] of GPlus y and the GPlusAux in the term, which is the GPlusxGPlusAux lemma (I'm not showing the proof of this lemma here).

```
2:EApply proj1; Apply GPlusxGPlusAux with x:=y.
Apply xLIH.

  ============================
   (Glte
     (Fix GPlusAux {...} (yLf yli))
     (Fix GPlus {} (yLf yli) (Game_cons xLI xRI xLf xRf)))

subgoal 2 is:
 (r:(sumT yRI xRI))
  (NGgte
    (Game_cons (sumT xLI yLI) (sumT xRI yRI)
      (map_on_sumT xLI yLI Game
        [xli:xLI]
         (Fix GPlus {...} (xLf xli) (Game_cons yLI yRI yLf yRf))
        [yli:yLI]
         (Fix GPlusAux {...} (yLf yli)))
      (map_on_sumT xRI yRI Game
        [xri:xRI]
         (Fix GPlus {...} (xRf xri) (Game_cons yLI yRI yLf yRf))
        [yri:yRI]
         (Fix GPlusAux {...} (yRf yri))))
    (map_on_sumT yRI xRI Game
      [xri:yRI]
       (Fix GPlus {...} (yRf xri) (Game_cons xLI xRI xLf xRf))
      [yri:xRI]
       (Fix GPlusAux {...} (xRf yri)) r))
```

The remaining goals are similar, I'm not detailing their proof here. We have proven inequality of (GPlus x y) and (GPlus y x), equality follows easily thanks to the symmetry of the expression, by using the inequality lemma on both sides of the conjuction equality is defined by,

```
Theorem GPlusCommutative: (x,y:Game)(Geq (GPlus x y) (GPlus y x)).
Proof.
Split; Apply GPlusCommutative_one.
Qed.
```

[1] precisely inequality, but they are equal

3.2.2 Associativity

As for commutativity, we first prove $\forall(x,y,z), x+(y+z) \leq (x+y)+z$, and the other inequality derives from the first, here with the help of commutativity.

The first inequality goes by induction on the variables, by order of appearance (x, y, z).

```
Theorem GPlusAssociative_one: (x,y,z:Game)
           (Glte (GPlus x (GPlus y z)) (GPlus (GPlus x y) z)).
Intros x.
NewInduction x.
Rename RI into xRI.
Rename LI into xLI.
Rename g into xLf.
Rename H into xLIH.
Intros xRf xRIH.
LetTac x:=(Game_cons xLI xRI xLf xRf).
Intros y.
NewInduction y.
Rename RI into yRI.
Rename LI into yLI.
Rename g into yLf.
Rename H into yLIH.
Intros yRf yRIH.
LetTac y:=(Game_cons yLI yRI yLf yRf).
Intros z.
NewInduction z.
Rename RI into zRI.
Rename LI into zLI.
Rename g into zLf.
Rename H into zLIH.
Intros zRf zRIH.
LetTac z:=(Game_cons zLI zRI zLf zRf).
```

```
  xLI : Type
  xRI : Type
  xLf : xLI->Game
  xLIH : (T:xLI; y,z:Game)
         (Glte (GPlus (xLf T) (GPlus y z)) (GPlus (GPlus (xLf T) y) z))
  xRf : xRI->Game
  xRIH : (l:xRI; y,z:Game)
         (Glte (GPlus (xRf l) (GPlus y z)) (GPlus (GPlus (xRf l) y) z))
  x := (Game_cons xLI xRI xLf xRf) : Game
  yLI : Type
  yRI : Type
  yLf : yLI->Game
```

```
yLIH : (T:yLI; z:Game)
        (Glte (GPlus x (GPlus (yLf T) z)) (GPlus (GPlus x (yLf T)) z))
yRf : yRI->Game
yRIH : (l:yRI; z:Game)
        (Glte (GPlus x (GPlus (yRf l) z)) (GPlus (GPlus x (yRf l)) z))
y := (Game_cons yLI yRI yLf yRf) : Game
zLI : Type
zRI : Type
zLf : zLI->Game
zLIH : (T:zLI)
        (Glte (GPlus x (GPlus y (zLf T))) (GPlus (GPlus x y) (zLf T)))
zRf : zRI->Game
zRIH : (l:zRI)
        (Glte (GPlus x (GPlus y (zRf l))) (GPlus (GPlus x y) (zRf l)))
z := (Game_cons zLI zRI zLf zRf) : Game
============================
 (Glte (GPlus x (GPlus y z)) (GPlus (GPlus x y) z))
```

We reduce the goal, to have something of the form `(Glte (Game_cons ...) (Game_cons ...))` so that we can use `Glte`'s only constructor. We then immediately do case distinction on the left of $x + (y + z)$ (respectively right of $(x + y) + z$) this has introduced (which for the lack of a better name, we call `i`), which is either $x_l + (y + z)$ (respectively $(x + y)_r + z$) or $x + (y + z)_l$ (respectively $(x + y) + z_r$).

```
Compute.
Exists; Fold x y z; Intro i; Case i.
```

```
i : (sumT xLI (sumT yLI zLI))
============================
 (x0:xLI)
  [xl:=(Fix GPlus {...} (xLf x0)
         (Game_cons (sumT yLI zLI) (sumT yRI zRI)
           [x:(sumT yLI zLI)]
            Cases x of
              (inlT e) =>
               (Fix GPlus {...} (yLf e) z)
            | (inrT e) =>
               (Fix GPlusAux {...} (zLf e))
            end
           [x:(sumT yRI zRI)]
            Cases x of
              (inlT e) =>
               (Fix GPlus {...} (yRf e) z)
            | (inrT e) =>
               (Fix GPlusAux {...} (zRf e))
```

```
          end))]
    (NGgte xl
      (Game_cons (sumT (sumT xLI yLI) zLI) (sumT (sumT xRI yRI) zRI)
        [x1:(sumT (sumT xLI yLI) zLI)]
         Cases x1 of
           (inlT e) =>
            (Fix GPlus {...}
              Cases e of
                (inlT e0) =>
                 (Fix GPlus {...} (xLf e0) y)
              | (inrT e0) =>
                 (Fix GPlusAux {...} (yLf e0))
              end z)
         | (inrT e) =>
            (Fix GPlusAux {...} (zLf e))
         end
        [x1:(sumT (sumT xRI yRI) zRI)]
         Cases x1 of
           (inlT e) =>
            (Fix GPlus {...}
              Cases e of
                (inlT e0) =>
                 (Fix GPlus {...} (xRf e0) y)
              | (inrT e0) =>
                 (Fix GPlusAux {...} (yRf e0))
              end z)
         | (inrT e) =>
            (Fix GPlusAux {...} (zRf e))
         end))
(... 3 other goals not shown ...)
```

Careful inspection of the first goal reveals that it is (what Coq calls `xl` is the s below):

$$\forall x_l \in L_x, \text{let } s := x_l + (y + z) \text{ in } s \triangleleft (x + y) + z$$

We use `NGgte`'s second constructor, after giving reasonable names to the quantification variable and the definition (s, which we call `PxlPyz` in the Coq code[2]).

```
Intros xli PxlPyz.
Constructor 2.
```

The goal is now

$$\exists l \in L_{(x+y)+z}, x_l + (y + z) \leq l$$

[2]because in polish notation, it is $+\ x_l\ +\ y\ z$. Replacing $+$ by `P`, for "Plus", you get the name. This convention will be used again.

l must be either a $(x+y)_l+z$ or a $(x+y)+z_l$. We choose l to be of the form $(x+y)_l+z$, more precisely $(x_l+y)+z$, so that the resulting goal is proven by the left induction hypothesis on x (xLIH).

```
Exists (inlT (sumT xLI yLI) zLI (inlT xLI yLI xli)).
```

```
  ...
  xLIH : (T:xLI; y,z:Game)
         (Glte (GPlus (xLf T) (GPlus y z)) (GPlus (GPlus (xLf T) y) z))
  ...
  ============================
   (Glte PxlPyz
     (Fix GPlus {...} (Fix GPlus {...} (xLf xli) y) z))
```

We now finish this goal off with xLIH. This brings to the top of the stack the case where i is an $x+(y+z)_l$. This again calls for a case distinction (it is either an $x+(y_l+z)$ or an $x+(y+z_l)$), and on the two cases we do the same as before, i.e. second constructor of NGgte, and instantiate the existential quantification with the right thing.

```
Apply xLIH with T:=xli y:=y z:=z.
```

```
Intro s.
Case s; [Intros yli | Intros zli]; [Intros PxPylz | Intros PxPyzl ];
Constructor 2; [Exists (inlT (sumT xLI yLI) zLI (inrT xLI yLI yli)) |
                Exists (inrT (sumT xLI yLI) zLI zli)].
```

```
  i : (sumT xLI (sumT yLI zLI))
  s : (sumT yLI zLI)
  yli : yLI
  PxPylz := (Fix GPlusAux {...}
              (Fix GPlus {...} (yLf yli) z)) : Game
  ============================
   (Glte PxPylz
     (Fix GPlus {...}
       (Fix GPlusAux {...} (yLf yli)) z))

subgoal 2 is:
 (Glte PxPyzl
   (Fix GPlusAux {...} (zLf zli)))
```

We cannot directly use the induction hypotheses, because they give us a property of GPlus, and the goals contain GPlusAux's. So, we must replace them by GPlus's. We insert the terms we want by transitivity of Glte.

```
Apply Glte_transitive with y:=(GPlus (GPlus x (yLf yli)) z).
```

```
Apply Glte_transitive with y:=(GPlus x (GPlus (yLf yli) z)).

  ============================
   (Glte PxPylz (GPlus x (GPlus (yLf yli) z)))

subgoal 2 is:
 (Glte (GPlus x (GPlus (yLf yli) z)) (GPlus (GPlus x (yLf yli)) z))

subgoal 3 is:
 (Glte (GPlus (GPlus x (yLf yli)) z)
   (Fix GPlus {...} (Fix GPlusAux {...} (yLf yli)) z))
subgoal 4 is:
 (Glte PxPyzl
   (Fix GPlusAux {...} (zLf zli)))
```

The first goal is proven by `GPlusxGPlusAux`. `proj2` is the lemma `(P,Q:Prop) (P /\ Q) -> Q`. We need it because `GPlusxGPlusAux` proves equality, i.e. a conjunction of inequalities, and our goals is just one of these inequalities. The second goal is an instantiation of the left y-induction hypothesis.

```
EApply proj2.
Apply GPlusxGPlusAux with x:=x y:=(GPlus (yLf yli) z).

  ============================
   (Glte (GPlus (GPlus x (yLf yli)) z)
     (Fix GPlus {...}
       (Fix GPlusAux {...} (yLf yli)) z))
```

Here, the GPlusAux to compare with a `GPlus x` is embedded in the left of a `GPlus`. We thus need to use compatibility of addition and the order at the left. This is lemma `GPlusGlteLeft`, see section 3.3.1.

```
Apply GPlusGlteLeft.

  ============================
   (Glte (GPlus x (yLf yli))
     (Fix GPlusAux {} (yLf yli)))
```

And now is the same `projN`/`GPlusxGPlusAux` pair.

```
EApply proj1.
Apply GPlusxGPlusAux with x:=x y:=(yLf yli).
```

The other goals are similar, and not detailed here. We finished the proof of $x + (y + z) \leq (x + y) + z$

We now move on the equality of $x + (y + z)$ and $(x + y) + z$. One direction we just did.

```
Theorem GPlusAssociative: (x,y,z:Game)
        (Geq (GPlus x (GPlus y z)) (GPlus (GPlus x y) z)).
Intros.
Split.
Apply GPlusAssociative_one.

  x : Game
  y : Game
  z : Game
  ============================
   (Glte (GPlus (GPlus x y) z) (GPlus x (GPlus y z)))
```

The other direction will come with the help of commutativity.

```
EApply Glte_transitive.
Apply GPlusCommutative_one.

  ============================
   (Glte (GPlus z (GPlus x y)) (GPlus x (GPlus y z)))

EApply Glte_transitive.
Apply GPlusAssociative_one.

  ============================
   (Glte (GPlus (GPlus z x) y) (GPlus x (GPlus y z)))

EApply Glte_transitive.
Apply GPlusCommutative_one.

  ============================
   (Glte (GPlus y (GPlus z x)) (GPlus x (GPlus y z)))

EApply Glte_transitive.
Apply GPlusAssociative_one.

  ============================
   (Glte (GPlus (GPlus y z) x) (GPlus x (GPlus y z)))

EApply Glte_transitive.
Apply GPlusCommutative_one.

  ============================
   (Glte (GPlus x (GPlus y z)) (GPlus x (GPlus y z)))
```

And we finish off with reflexivity of `Glte`.

```
Apply Glte_reflexive.
Qed.
```

This concludes the proof of the associativity of addition.

3.2.3 Symmetric

We now prove that `AntiGame x` is the symmetric of `x` for addition, i.e. `Geq (GPlus x (AntiGame x)) Zero`. Again, we first prove inequality, and the other inequality follow with the help of other properties.

$x + (-x) \leq 0$

By induction on x. We use `Glte`'s only constructor.

```
Theorem xPlusAntixZero_one : (x:Game)(Glte (GPlus x (AntiGame x)) Zero).
Proof.
Intros x.
NewInduction x.
Rename RI into xRI.
Rename LI into xLI.
Rename g into xLf.
Rename H into xLIH.
Intros xRf xRIH.
LetTac x:=(Game_cons xLI xRI xLf xRf).
Compute; Exists; Fold Zero.

  xLI : Type
  xRI : Type
  xLf : xLI->Game
  xLIH : (T:xLI)(Glte (GPlus (xLf T) (AntiGame (xLf T))) Zero)
  xRf : xRI->Game
  xRIH : (l:xRI)(Glte (GPlus (xRf l) (AntiGame (xRf l))) Zero)
  x := (Game_cons xLI xRI xLf xRf) : Game
  ============================
   (l:(sumT xLI xRI))
    [xl:=Cases l of
           (inlT e) =>
            (Fix GPlus {...} (xLf e)
              (Game_cons xRI xLI
                [i:xRI]
                 (Fix AntiGame {...} (xRf i))
                [i:xLI]
                 (Fix AntiGame {...} (xLf i))))
         | (inrT e) =>
```

```
            (Fix GPlusAux {...}
              (Fix AntiGame {...} (xRf e)))
          end]
     (NGgte xl
       (Game_cons False False [H:False]<Game>Cases H of end
         [H:False]<Game>Cases H of end))

subgoal 2 is:
 (r:False)
  (NGgte
    (Game_cons (sumT xLI xRI) (sumT xRI xLI)
      ...
    ))
```

The second goal assumes a right r of `Zero`, and is easily taken care of (`Zero` has no right).

```
2: Intro; Contradiction.
```

We give `1` the more reasonable name `xi` (because it is either an `xli` or an `xri`), and do case distinction on it.

```
Intro xi; Case xi; [Intro xli | Intro xri].

  xi : (sumT xLI xRI)
  xli : xLI
  ============================
   [xl:=(Fix GPlus {...} (xLf xli)
          (Game_cons xRI xLI
            [i:xRI]
             (Fix AntiGame {...} (xRf i))
            [i:xLI]
             (Fix AntiGame {...} (xLf i))))]
    (NGgte xl
      (Game_cons False False [H:False]<Game>Cases H of end
        [H:False]<Game>Cases H of end))

subgoal 2 is:
 [xl:=(Fix GPlusAux {...}
        (Fix AntiGame {...} (xRf xri)))]
  (NGgte xl
    (Game_cons False False [H:False]<Game>Cases H of end
      [H:False]<Game>Cases H of end))
```

The first goal is essentially

$$x_l + (-x) \lhd 0$$

Here is the proof of this, in mathematical notation. After, we will try to do it in Coq and see why it fails.

x_l is a game, thus there is (L, R) a couple of sets of games, such that $x_l = \{L|R\}$, and we now have to prove

$$\{L|R\} + \{-R_x| - L_x\} \lhd 0$$

By definition of $+$, this is

$$\{\ldots | \{r + (-x), r \in R_x\} \cup \{x_l + (-l), l \in L_x\}\} \lhd 0$$

Using the second constructor of $\lhd$ is hopeless, because 0 has no left. We thus use the first constructor of $\lhd$, and the left to prove is:

$$\exists y \in \{r + (-x), r \in R_x\} \cup \{x_l + (-l), l \in L_x\}, y \leq 0$$

We choose y in $\{x_l + (-l), l \in L_x\}$ and $l := x_l$, i.e. $y := x_l + (-x_l)$.

$$x_l + (-x_l) \leq 0$$

This is true by induction hypothesis `xLIH`.

Back to Coq, now. To use the first constructor of `NGgte`, its first argument must be of the form `Game_cons ....` We thus need to reduce `GPlus (xLf xli) ....` To do this, we need to replace `(xLf xli)` by something of the form `Game_cons ...`, i.e. give the left and right index and function of `(xLf xli)` a name. Let's accept the names Coq chooses (`LI`, `RI`, `g`, `g0`), and make use of the first constructor of `NGgte`.

```
Case (xLf xli).
Constructor 1.
```

```
  LI : Type
  RI : Type
  g : LI->Game
  g0 : RI->Game
  ============================
   (EXT r:(sumT RI xLI) |
        (Glte
          Cases r of
            (inlT e) =>
             (Fix GPlus {...} (g0 e)
               (Game_cons xRI xLI
                 [i:xRI]
                  (Fix AntiGame {...} (xRf i))
                 [i:xLI]
                  (Fix AntiGame {...} (xLf i))))
          | (inrT e) =>
             (Fix GPlusAux {...}
               (Fix AntiGame {...} (xLf e)))
```

```
        end
        (Game_cons False False [H:False]<Game>Cases H of end
          [H:False]<Game>Cases H of end)))
```

We instantiate the r with `xli`.

```
Exists (inrT RI xLI xli).
```

```
  ============================
   (Glte
     (Fix GPlusAux {...}
       (Fix AntiGame {...} (xLf xli)))
     (Game_cons False False [H:False]<Game>Cases H of end
       [H:False]<Game>Cases H of end))
```

We replace `GPlusAux` by the corresponding `GPlus ?`:

```
EApply Glte_transitive.
EApply proj2; Apply GPlusxGPlusAux with x:=(Game_cons LI RI g g0).
```

```
  ============================
   (Glte
     (GPlus (Game_cons LI RI g g0)
       (Fix AntiGame {...} (xLf xli)))
     (Game_cons False False [H:False]<Game>Cases H of end
       [H:False]<Game>Cases H of end))
```

To improve human readability, we introduce the folded version of this term:

```
Cut (Glte (GPlus (Game_cons LI RI g g0) (AntiGame (xLf xli)))
          Zero).
Intro P; Exact P.
```

```
  xLIH : (T:xLI)(Glte (GPlus (xLf T) (AntiGame (xLf T))) Zero)
  ...
  ============================
   (Glte (GPlus (Game_cons LI RI g g0) (AntiGame (xLf xli))) Zero)
```

We'd like to use `xLIH`, after all, what we have is exactly `xLIH` with `T` instantiated by `xli`. Well, that is because we remember that `(Game_cons LI RI g g0)` is just the expanded form of `(xLf xli)`. The whole problem is there: Coq doesn't remember. It has *thrown* that information *away*. Presented like that, it looks like an implementation flaw in Coq, but if we temporarily forget this example, the following reasoning might look reasonable:

> As we replace every occurrence of `(xLf xli)` in the goal by its expansion, adding the hypothesis `(xLf xli) == (Game_cons LI RI g g0)` is not necessary, as there is no occurrence of `(xLf xli)` left.

The problem here arises because at the time we do the Case, the goal contains a subterm that is more generic than (xLf xli), and can be instantiated to (xLf xli) only later.

So, what can we do about this? We could have introduced (xLIH xli) by a cut in the goal before doing the Case. But then, we would end up with

```
xLIHxli : (Glte (GPlus (Game_cons LI RI g g0)
                       (AntiGame (Game_cons LI RI g g0)))
                Zero)
============================
 (Glte (GPlus (Game_cons LI RI g g0) (AntiGame (xLf xli))) Zero)
```

Doesn't work either, because the goal still has an (xLf xli), and now the hypothesis has *none*. So, we'd need to "protect" one of the (xLf xli)'s of being replaced by the expansion by the Case. Coq indeed has a notion of *transparent* and *opaque* constants, where transparent constants get replaced, and opaque constants don't. We can indeed get into the situation:

```
xl := (xLf xli)
============================
 (Glte (GPlus (Game_cons (xLf xli)) (AntiGame xl)) Zero) ->
 (NGgte (GPlus (xLf xli)
   (Game_cons False False [H:False]<Game>Cases H of end
     [H:False]<Game>Cases H of end)))
```

If only we could now declare xl to be opaque, everything would go well. Unfortunately, this opaque vs. transparent separation exists only for section-level constants, not constants defined in hypotheses. So, it looks like the only way out is taking the goal in a separate lemma, in a separate section. Let's call this section SillyWorkAround_0 (in case we need other sections like this).

```
Section SillyWorkAround_0.
Variable xLI : Type.
Variable xRI : Type.
Variable xLf : xLI->Game.
Variable xRf : xRI->Game.
Variable xli : xLI.
Variable xri:xRI.
Definition SillyWorkAround_0_xl := (xLf xli) : Game.
Definition SillyWorkAround_0_xr := (xRf xri) : Game.
Lemma SillyWorkAround_0_L0 :
     (Glte (GPlus (xLf xli) (AntiGame SillyWorkAround_0_xl)) Zero)
   ->(NGgte
       (Fix GPlus {...} (xLf xli)
         (Game_cons xRI xLI
           [i:xRI]
            (Fix AntiGame {...} (xRf i))
           [i:xLI]
```

```
            (Fix AntiGame {...} (xLf i)))) Zero).
Proof.
```

First thing we do is declare SillyWorkAround_0_xl opaque, and then, bingo! The Case leaves SillyWorkAround_0_xl alone.

```
Opaque SillyWorkAround_0_xl.
Case (xLf xli).

  xLI : Type
  xRI : Type
  xLf : xLI->Game
  xRf : xRI->Game
  xli : xLI
  xri : xRI
  ============================
   (LI,RI:Type; g:(LI->Game); g0:(RI->Game))
    (Glte
      (GPlus (Game_cons LI RI g g0) (AntiGame SillyWorkAround_0_xl))
      Zero)
    ->(NGgte
        (Game_cons (sumT LI xRI) (sumT RI xLI)
          [x1:(sumT LI xRI)]
           Cases x1 of
             (inlT e) =>
              (Fix GPlus {...} (g e)
                (Game_cons xRI xLI
                  [i:xRI]
                   (Fix AntiGame {...} (xRf i))
                  [i:xLI]
                   (Fix AntiGame {...} (xLf i))))
           | (inrT e) =>
              (Fix GPlusAux {...}
                (Fix AntiGame {...} (xRf e)))
           end
          [x1:(sumT RI xLI)]
           Cases x1 of
             (inlT e) =>
              (Fix GPlus {...} (g0 e)
                (Game_cons xRI xLI
                  [i:xRI]
                   (Fix AntiGame {...} (xRf i))
                  [i:xLI]
                   (Fix AntiGame {...} (xLf i))))
           | (inrT e) =>
```

```
            (Fix GPlusAux {...}
              (Fix AntiGame {...} (xLf e)))
          end) Zero)
```

The rest then goes as outlined above.

```
Intros.
Constructor 1.
Exists (inrT RI xLI xli).
Compute.
Apply Glte_transitive with y:=(GPlus (Game_cons LI RI g g0)
                                     (AntiGame (xLf xli))).
EApply proj2; Apply GPlusxGPlusAux with x:=(Game_cons LI RI g g0)
                                        y:=(AntiGame (xLf xli)).
Exact H.
Qed.
```

And we can use that `SillyWorkaround_0_L0`, and the similar `SillyWorkaround_0_L1` in the proof of `xPlusAntixZero_one`:

```
  ============================
   [xl:=...]
    (NGgte xl
      (Game_cons False False [H:False]<Game>Cases H of end
        [H:False]<Game>Cases H of end))

Apply SillyWorkAround_0_L0; Apply (xLIH xli).
Apply SillyWorkAround_0_L0; Apply (xLIH xli).
Apply SillyWorkAround_0_L1 with
                 xLI:=xLI xRI:=xRI xLf:=xLf xRf:=xRf xri:=xri;
Apply (xRIH xri).
Qed.
```

$x + (-x) = 0$

$\leq$ is the previous lemma

```
Theorem xPlusAntixZero : (x:Game)(Geq (GPlus x (AntiGame x)) Zero).
Intros; Split.
Apply xPlusAntixZero_one.

  x : Game
  ============================
   (Glte Zero (GPlus x (AntiGame x)))
```

And $\geq$ come with the help of `AntiGlte`, which expresses the relationship between `Glte` and `AntiGame`, i.e. `(x,y:Game)(Glte (AntiGame y) (AntiGame x))->(Glte x y)` (this lemma is not detailed in this report).

```
Apply AntiGlte.

  ============================
   (Glte (AntiGame (GPlus x (AntiGame x))) (AntiGame Zero))
```

We now use `AntiGPlus` (not detailed here, it is $-(x+y) = (-x) + (-y)$) to trickle the `AntiGame` into the arguments of the `GPlus`.

```
EApply Glte_transitive.
EApply proj1; Apply AntiGPlus.

  ============================
   (Glte (GPlus (AntiGame x) (AntiGame (AntiGame x))) (AntiGame Zero))
```

`AntiGame Zero` is `Zero`

```
EApply Glte_transitive with y:=Zero.

  ============================
   (Glte (GPlus (AntiGame x) (AntiGame (AntiGame x))) Zero)

subgoal 2 is:
 (Glte Zero (AntiGame Zero))
```

This is exactly the previous lemma.

```
Apply xPlusAntixZero_one.

  ============================
   (Glte Zero (AntiGame Zero))
```

And this goes nearly automatic.

```
Compute.
Exists; Intro; Contradiction.
Qed.
```

3.3 $(G, +)$ is an ordered group

We now tackle compatibility of addition and symmetrisation (`AntiGame`) with `Glte`.

3.3.0 `Glte` and `AntiGame`

The interaction between `Glte` and `AntiGame` is used in various forms in the other proofs:

$$\forall (x,y), x \leq y \to (-y) \leq (-x)$$
$$\forall (x,y), x = y \to (-y) = (-x)$$
$$\forall (x,y), (-y) \leq (-x) \to x \leq y$$

We will detail only the first one here, the others follow with the help of idempotence of `GlteAnti` (which is not detailed in this report either. Is done by induction, no particular difficulties.).

We prove

$$\forall(x,y), x \leq y \to (-y) \leq (-x)$$

and

$$\forall(x,y), x \lhd y \to (-y) \lhd (-x)$$

by mutual induction. More precisely, we prove

$$\forall(x,y), (x \leq y \to (-y) \leq (-x)) \wedge (y \lhd x \to (-x) \lhd (-y))$$

with the induction scheme `IS2Reverse`, very similar to `IS2`, which we already used.

```
Theorem IS2Reverse:  (P:Game->Game->Prop)
             ((x,y:Game)
              ((xli:(GLeftIndex x))[xl:=((GLeftFun x) xli)] (P y xl)) ->
              ((yri:(GRightIndex y))[yr:=((GRightFun y) yri)] (P yr x)) ->
              (P x y)) ->
             (x,y:Game) (P x y).
Proof.
...
Qed.

Lemma GlteNGgteAnti:(x,y:Game)
 ((Glte x y) -> (Glte (AntiGame y) (AntiGame x))) /\
 ((NGgte y x) -> (NGgte (AntiGame x) (AntiGame y))).
Intros x y.
EApply IS2Reverse with
 P:=[a,b:Game]((Glte a b) -> (Glte (AntiGame b) (AntiGame a)))
            /\ ((NGgte b a) -> (NGgte (AntiGame a) (AntiGame b))).
Clear x y.
Intros x y.
Case x.
Intros xLI xRI xLf xRf.
Case y.
Intros yLI yRI yLf yRf.
Clear x y.
LetTac x:=(Game_cons xLI xRI xLf xRf).
LetTac y:=(Game_cons yLI yRI yLf yRf).
Intros IHxl IHyr.
Split.

  IHxl : (xli:(GLeftIndex x))
          [xl:=(GLeftFun x xli)]
```

```
          ((Glte y xl)->(Glte (AntiGame xl) (AntiGame y)))
          /\ ((NGgte xl y)->(NGgte (AntiGame y) (AntiGame xl)))
  IHyr : (yri:(GRightIndex y))
         [yr:=(GRightFun y yri)]
          ((Glte yr x)->(Glte (AntiGame x) (AntiGame yr)))
          /\ ((NGgte x yr)->(NGgte (AntiGame yr) (AntiGame x)))
  ============================
   (Glte x y)->(Glte (AntiGame y) (AntiGame x))

subgoal 2 is:
 (NGgte y x)->(NGgte (AntiGame x) (AntiGame y))
```

Intro, and Glte has only one constructor.

```
Intro xy.
Unfold x y.
Compute.
Exists; Fold AntiGame.

  xy : (Glte x y)
  ============================
   (l:yRI)
    [xl:=(AntiGame (yRf l))]
     (NGgte xl
       (Game_cons xRI xLI [i:xRI](AntiGame (xRf i))
         [i:xLI](AntiGame (xLf i))))

subgoal 2 is:
 (r:xLI)
  (NGgte
    (Game_cons yRI yLI [i:yRI](AntiGame (yRf i))
      [i:yLI](AntiGame (yLf i))) (AntiGame (xLf r)))
```

The two goals correspond to the left case and right case. Very similar, we will do only the left case. We need to use the hypothesis xy, we deconstruct it.

```
Simple Inversion xy.
Intros Nxly Nxyr.

  xLI0 : Type
  xRI0 : Type
  xLf0 : xLI0->Game
  xRf0 : xRI0->Game
  yLI0 : Type
  yRI0 : Type
  yLf0 : yLI0->Game
```

```
  yRf0 : yRI0->Game
  x0 := (Game_cons xLI0 xRI0 xLf0 xRf0) : Game
  y0 := (Game_cons yLI0 yRI0 yLf0 yRf0) : Game
  H1 : x0==x
  H2 : y0==y
  Nxly : (l:xLI0)[xl:=(xLf0 l)](NGgte xl y0)
  Nxyr : (r:yRI0)(NGgte x0 (yRf0 r))
  ============================
   (l:yRI)
    [xl:=(AntiGame (yRf l))]
     (NGgte xl
       (Game_cons xRI xLI [i:xRI](AntiGame (xRf i))
         [i:xLI](AntiGame (xLf i))))
```

We introduce the hypothesis we need to use with a `Cut` (it is a part of `IHyr`), and we merge the goal's `l` with the `yri` we introduced.

```
Cut (yri:yRI)
(NGgte (Game_cons xLI xRI xLf xRf) (yRf yri) )
->          (NGgte (AntiGame (yRf yri))
              (Game_cons xRI xLI [i:xRI](AntiGame (xRf i))
                [i:xLI](AntiGame (xLf i))) )
.
Apply Merge_quanteurs with
T:= yRI
P:=[yri:yRI](NGgte (Game_cons xLI xRI xLf xRf) (yRf yri))
     ->(NGgte (AntiGame (yRf yri))
         (Game_cons xRI xLI [i:xRI](AntiGame (xRf i))
           [i:xLI](AntiGame (xLf i))))
Q:=[l:yRI]
      [xl:=(AntiGame (yRf l))]
       (NGgte xl
         (Game_cons xRI xLI [i:xRI](AntiGame (xRf i))
           [i:xLI](AntiGame (xLf i))))
.
Fold x.

  ============================
   (t:yRI)
    ((NGgte x (yRf t))
     ->(NGgte (AntiGame (yRf t))
         (Game_cons xRI xLI [i:xRI](AntiGame (xRf i))
           [i:xLI](AntiGame (xLf i)))))
    ->[xl:=(AntiGame (yRf t))]
       (NGgte xl
```

```
        (Game_cons xRI xLI [i:xRI](AntiGame (xRf i))
          [i:xLI](AntiGame (xLf i))))

subgoal 2 is:
 (yri:yRI)
  (NGgte (Game_cons xLI xRI xLf xRf) (yRf yri))
  ->(NGgte (AntiGame (yRf yri))
      (Game_cons xRI xLI [i:xRI](AntiGame (xRf i))
        [i:xLI](AntiGame (xLf i))))
```

And here is why we pulled the hypothesis back down into the goal: we have to rewrite it to match the new names for x and y the Inversion has introduced.

```
Injection H2.
Intros yRf0yRf yLf0yLf yRI0yRI yLI0yLI.
Dependent Rewrite <- yRf0yRf.
Compute; Fold x AntiGame.
Intros yri IHyrGlte.

  yRf0yRf : (existT Type [RI:Type]RI->Game yRI0 yRf0)
            ==(existT Type [RI:Type]RI->Game yRI yRf)
  yLf0yLf : (existT Type [LI:Type]LI->Game yLI0 yLf0)
            ==(existT Type [LI:Type]LI->Game yLI yLf)
  yRI0yRI : yRI0==yRI
  yLI0yLI : yLI0==yLI
  yri : yRI0
  IHyrGlte : (NGgte (Game_cons xLI0 xRI0 xLf0 xRf0) (yRf0 yri))
             ->(NGgte (AntiGame (yRf0 yri))
                 (Game_cons xRI xLI [i:xRI](AntiGame (xRf i))
                   [i:xLI](AntiGame (xLf i))))
  ============================
   (NGgte (AntiGame (yRf0 yri))
     (Game_cons xRI xLI [i:xRI](AntiGame (xRf i))
       [i:xLI](AntiGame (xLf i))))
```

We use the induction hypothesis we introduced with the Cut, and the goal is the Nxyr hypothesis.

```
Apply IHyrGlte.
Apply Nxyr.

  Nxly : (l:xLI0)[xl:=(xLf0 l)](NGgte xl y0)
  Nxyr : (r:yRI0)(NGgte x0 (yRf0 r))
  ============================
   (yri:yRI)
    (NGgte x (yRf yri))
```

```
    ->(NGgte (AntiGame (yRf yri))
        (Game_cons xRI xLI [i:xRI](AntiGame (xRf i))
          [i:xLI](AntiGame (xLf i))))
```

This leaves us with the task to prove that the proposition we introduced with the `Cut` is true. As mentioned before, it is part of `IHyr`.

```
Intros.
EApply MP.
EApply proj2.
Apply IHyr.
Assumption.
```

Left to prove: the right case of `Glte`, and `(NGgte y x)->(NGgte (AntiGame x) (AntiGame y))`. Similar, not detailed here.

No is stable by `AntiGame`

This permits us to prove that if x is a number, then so is $-x$. By induction on x.

```
Lemma Number_AntiGame_closed:
          (x:Game) (IsNumber x) -> (IsNumber (AntiGame x)).
Intros.
NewInduction x.

Intros.
Exists.

  LI : Type
  RI : Type
  g : LI->Game
  H0 : (T:LI)(IsNumber (g T))->(IsNumber (AntiGame (g T)))
  H : RI->Game
  H1 : (l:RI)(IsNumber (H l))->(IsNumber (AntiGame (H l)))
  H2 : (IsNumber (Game_cons LI RI g H))
  ============================
   (xli:(GLeftIndex (AntiGame (Game_cons LI RI g H))))
    (IsNumber (GLeftFun (AntiGame (Game_cons LI RI g H)) xli))

subgoal 2 is:
 (xri:(GRightIndex (AntiGame (Game_cons LI RI g H))))
  (IsNumber (GRightFun (AntiGame (Game_cons LI RI g H)) xri))
subgoal 3 is:
 (xli:(GLeftIndex (AntiGame (Game_cons LI RI g H)));
  xri:(GRightIndex (AntiGame (Game_cons LI RI g H))))
  (NGgte (GLeftFun (AntiGame (Game_cons LI RI g H)) xli)
```

```
    (GRightFun (AntiGame (Game_cons LI RI g H)) xri))
```

```
Intros.
Compute in xli.
```

```
  xli : RI
  ============================
   (IsNumber (GLeftFun (AntiGame (Game_cons LI RI g H)) xli))
```

We reduce the goal

```
Cbv Beta Iota Zeta Delta [GLeftFun AntiGame].
Fold AntiGame.
```

```
  ============================
   (IsNumber (AntiGame (H xli)))
```

The symmetric of a left of x is a number. By induction this is true if the left of x is a number.

```
Apply H1.
```

```
  ============================
   (IsNumber (H xli))
```

And this derives from "x is a number".

```
Inversion H2.
Compute in H3 H4.
Cbv Beta Iota Zeta Delta in H5.
```

```
  x : Game
  H6 : x==(Game_cons LI RI g H)
  H3 : (xli:LI)(IsNumber (g xli))
  H4 : (xri:RI)(IsNumber (H xri))
  H5 : (xli:LI; xri:RI)(NGgte (g xli) (H xri))
  ============================
   (IsNumber (H xli))
```

```
Apply H4.
```

Next goal: the symmetric of a right of x is a number, similar, not detailed here. Last goal: no left is greater than or equal to a right.

```
Intros.
Compute in xri xli.
```

```
  LI : Type
```

```
  RI : Type
  g : LI->Game
  H0 : (T:LI)(IsNumber (g T))->(IsNumber (AntiGame (g T)))
  H : RI->Game
  H1 : (l:RI)(IsNumber (H l))->(IsNumber (AntiGame (H l)))
  H2 : (IsNumber (Game_cons LI RI g H))
  xli : RI
  xri : LI
  ============================
   (NGgte (GLeftFun (AntiGame (Game_cons LI RI g H)) xli)
     (GRightFun (AntiGame (Game_cons LI RI g H)) xri))
```

We rewrite the goal to make it more human-readable. This, too, derives from "x is a number".

```
Cbv Beta Iota Zeta Delta -[AntiGame].
Inversion H2.
Compute in H3 H4.
Cbv Beta Iota Zeta Delta in H5.
Compute; Fold AntiGame.
```

```
  x : Game
  H6 : x==(Game_cons LI RI g H)
  H3 : (xli:LI)(IsNumber (g xli))
  H4 : (xri:RI)(IsNumber (H xri))
  H5 : (xli:LI; xri:RI)(NGgte (g xli) (H xri))
  ============================
   (NGgte (AntiGame (H xli)) (AntiGame (g xri)))
```

We use the relationship between `NGgte` and `AntiGame`. While NGgteAnti is not shown per se in this report, it is a part of `GlteNGgteAnti`.

```
Apply NGgteAnti.
```

```
  ============================
   (NGgte (g xri) (H xli))
```

And we finish off

```
Apply H5.
Qed.
```

3.3.1 Glte and GPlus

Compatibility between `Glte` and `GPlus`. It goes similarly to compatibility of `Glte` and `AntiGame`:

1. Prove `((Glte x y) -> (Glte (GPlus x z) (GPlus y z)))` and `((NGgte x y) -> (NGgte (GPlus x z) (GPlus y z)))` by mutual induction. This time it is usual induction: on `x`, then on `y`, then on `z`. This is `GPlusOrder`.

```
Theorem GPlusOrder: (x,y,z:Game)
   ((Glte x y) -> (Glte (GPlus x z) (GPlus y z))) /\
   ((NGgte x y) -> (NGgte (GPlus x z) (GPlus y z))).
```

2. From there, derive the corollaries that are actually used in other proofs.

```
Theorem GPlusGlteLeft: (x,y,z:Game)
  (Glte x y) -> (Glte (GPlus x z) (GPlus y z)).
Theorem GPlusGlteRight: (x,y,z:Game)
  (Glte x y) -> (Glte (GPlus z x) (GPlus z y)).
Theorem GPlusGlteBoth: (x1,x2,y1,y2:Game)
  (Glte x1 x2) -> (Glte y1 y2) -> (Glte (GPlus x1 y1) (GPlus x2 y2)).
```

3.4 Multiplication

Definition of multiplication, and proof of most of its properties, don't bring really new issues. Mainly, the previous issues grow more important: the terms, instead of being a few thousands of lines long, are several dozens, or even hundreds of thousands lines, the memory consumption grows beyond the capacity of a current desktop PC, the tower of recursive calls and auxiliaries in the definition is even higher, etc.

The lack of an efficient way to reason modulo associativity and commutativity of addition is felt, leading to tedious proofs: see section 4.1.3 for details.

We have not done everything contained in "*No* is a totally ordered field". Most notably, we don't define division, and it looks complex (see section 3.5). Associativity of multiplication lacks, too. We restrict ourselves to a point-wise summary of what has been done formally in Coq; for details the reader is referred to the Coq source in appendix B:

- Definition of multiplication (`GMult`). This takes 52 lines!
- Zero is absorbent
- One is neutral
- Equality between `GMultAux` and `GMult x`, where `GMultAux` is the auxiliary function to have Coq accept the recursion, like for addition.
- Commutativity of multiplication
- $(-x) \cdot y = -(x \cdot y) = x \cdot (-y)$

- Distributivity, assuming compatibility of multiplication with equality (which has not been proven).

All these proofs go by quite straightforward (simple) induction, or simple case analysis, but are quite long: there are many, many similar cases to treat, especially distributivity (1200 lines).

Distributivity, however, raises a new issue: in the proof, one encounters terms like `(GMult (GPlusAux {...} t2) t3)`, which one needs to replace by `(GMult (GPlus t1 t2) t3)`. In a decision intended as a simplification, I used exclusively games equality, not identicality, nor Coq equality, i.e. I have the theorem `(Geq (GPlusAux {...} y) (GPlus x y))`, but not `(GPlusAux {...} y) == (GPlus x y)`. This means that our proof of distributivity depends on compatibility of multiplication with equality, which is true only for numbers, restricting our proof of distributivity to numbers only, for no good reason.

3.5 Division

Inversion (and thus indirectly division) is defined in [Con01] as follows. Let $x := \{\{0\} \cup L_x | R_x\}$ be a representative of a positive surreal number such that $\forall x_l \in L_x, x_l > 0$. Every positive surreal number has such a representative and $\forall x_r \in R_x, x_r > 0$ is automatic. The inverse of x, which we call y is:

$$\left\{0, \frac{1+(x_r-x)y_l}{x_r}, \frac{1+(x_l-x)y_r}{x_l} \middle| \frac{1+(x_l-x)y_l}{x_l}, \frac{1+(x_r-x)y_r}{x_r}\right\}$$

where x_l is an element of L_x, and occurrences of y_r or y_l are to be understood as: the first left of y is 0. Every left (respectively right) of y itself generates new lefts and rights of y, which in turn generate new lefts and rights.

Implementing this in Coq looks delicate, but doable: every left or right of y is constructed after a finite amount of such steps. Construction of the lefts and rights (options) of y can be described in the form of a tree, where nodes are labelled with options of y and edges by options of x. The root is zero. A node labelled by an y_l has a left son for every x_r (respectively a right son for every x_l). The corresponding edge is labelled with x_r (respectively x_l), and the son is labelled with the left (respectively right) of y constructed from y_l and x_r (respectively x_l). Similarly for a node labelled with an y_r. The choice of a left (respectively right) of y is then reduced to the choice of a path trough this tree that ends on a left (respectively right) son. The natural index set for the left of y is then something like $\mathcal{S}(L_x \boxplus R_x) \times L_x$, where $\boxplus$ is the disjoint union and $\mathcal{S}(X)$ is the set of strings (finite sequences) of X, plus one point (for zero). This leaves the problem of restricting ourselves to paths that finish with a left son, but that can be addressed.

This is one notch above multiplication in complexity. With a good dose of courage and concentration, the definition of inversion can be done. Division then follows easily. Proving properties of division looks like a scary job: properties of multiplication already toe the limit of feasibility, I'm afraid properties of division cross it.

3.6 Summary

We have proven the additive group structure on G and parts of the field structure on *No*: We have done in Coq some theorems about multiplication and nothing about division. For this, we needed induction and recursion schemes that Coq didn't recognise automatically as valid. Multiplication toes the limit of feasibility with respect to complexity and it is possible that division crosses it.

Chapter 4

Summary, Results and Conclusions

4.0 Introduction

The initial goal of this Master's thesis was to get a good idea of the current state of proof assistant technology, by subjecting one of its main representatives to a "torture test". The choice of the proof assistant settled to Coq, for various reasons: I had past experience with it, I have access to people experienced in Coq that could help me with Coq technical problems I might have and Coq fulfils my criteria for "acceptable mathematical/scientific tool", namely:

1. its general design is open and documented for anyone to study, understand and improve;
2. its detailed implementation is open, free for anyone to study, understand and improve;
3. its use, study and improvement is free of financial or political burden: anybody is free to use it, study it, understand it and improve it, irrespective of academic status, past achievements, ethnic origin, nationality or other status or affiliation.

The long term goal of proof assistant development is that one day, computer-checked mathematics will be the norm, not only because it heightens the confidence in correctness of the developments and results, but also because it will actually be easier to use a proof assistant rather than do everything by hand. In this view, a proof assistant has two somewhat contradictory roles:

help The proof assistant *assists* you, makes things easier.

- It does big, boring but simple case analysis autonomously, it takes care of the details of the proof, helping you to focus on the interesting stuff.
- It helps you to structure your results, to keep track of the dependencies between them.
- It helps you to know where you are, by keeping an up-to-date overview of your work. This includes keeping track of what you have already proven, permitting you to browse trough it effectively, maybe even warning you the lemma you are now starting is already proven.

supervisor The proof assistant makes your life miserable by insisting that what you do be correct, in some sense makes things harder.

- It constantly checks validity of what you do, forcing you to spell out any assumption you make. This component is partly a "help" component, too, because by spotting errors early, it prevents you from constructing a whole monument on shaky bases and it saves you the embarrassment of having an undergraduate student tell you twenty years after that your proof is incomplete.
- By keeping track of proof obligations, it ensures you don't forget any. It permits you to postpone a particular assumption to later, but doesn't let you get away with not eventually proving these assumptions.

For this to happen, mathematicians must be able to reason as they do on paper, without thinking too much to the encoding in how the proof checker works. The question is:

> Through how many hoops does a current system make you jump? How far away from your view of mathematics are you, even if your approach to mathematics is the "opposite" of the one used by the proof assistant?

This concern of testing the proof assistant on mathematics stemming from a different approach than the one behind it leads, in the case of Coq, which is based on type theory, to use a very set-centric notion as torture test: Type Theory and Set Theory are competing approaches to foundations of mathematics. So, how does Coq, a type theory based tool, deal with a very set-centric notion?

Once the proof assistant had been chosen, a good torture test had to be found. The FTA project of the university of Nijmegen (`http://www.cs.kun.nl/gi/projects/fta/`) had already done in Coq the full proof of a significant mathematical theorem, namely the fundamental theorem of algebra[0]. This includes defining the notion of (real and) complex numbers. This a good torture test, but real and complex numbers, as well as the fundamental theorem of algebra are very widely studied and known notions, respectively theorem. As a result, there are many different known ways to define, respectively prove, them, and the participants of the FTA project had quite some freedom in choosing the approach to take. This is not representative of the situation one would like the proof assistants to be used in: if one keeps waiting until a theorem has a dozen different proofs before running one through a proof assistant, computer mathematics will always be decades behind "paper mathematics", and proof assistants will never be assistants, only post-work checkers. What if as a complement to (or first filter before) peer review, a conference or journal wanted to require a fully formalised proof, in a proof assistant? Is Coq ready to fulfil this role? In this role, it is not acceptable to require the mathematician to re-develop his theory in a way that is compatible with the way the proof assistant works: he has a theory, he wants to run *his* theory through the proof assistant, not something else equivalent to it. He has a proof of a particular theorem, he wants to run *this* proof through the proof assistant, not another proof that fits better in the framework.

[0]Every non-constant polynomial over the complex numbers has a root.

In this perspective, surreal numbers, a notion of numbers which encompasses both the real numbers and the ordinals in a totally ordered field structure, presented by J.H. Conway in [Con01], fit in quite well:

1. The description of the essence of it is small enough that I could understand the big picture in a short time (on the scale of the time allotted to the Master's Thesis).

2. It is a "deep" mathematical construct. The very definition of what a surreal number is uses deep induction (a surreal number is made of arbitrary sets of surreal numbers) and makes use of a defined relation (the order: no left greater than a right), the operations use complicated recursions and the proofs use complicated inductions.

3. Surreal numbers are a very set-centric notion, an approach opposite to the one of Coq, thus.

4. The initial author didn't pay attention to constructiveness of the operations, constructs and reasonings. How much trouble does this cause in Coq?

   Even better: the natural foundations of mathematics for surreal numbers isn't quite Zermelo-Fraenkel set theory either, but rather a theory of sets with two "element of" relations. Let's call them bisets. A biset would have left elements and right elements, and a surreal number would be a biset of surreal numbers such that no left element is greater than a right element. The "pair of sets" construction already is an encoding in ZF set theory of "The Idea" of surreal numbers.

5. The interesting equality on surreal numbers is not Leibniz equality, but a weaker equivalence relation. How does Coq deal with this situation?

Thus, I set to the task of defining surreal numbers and the field structure on them in Coq, trying to stay as close as possible to Conway's presentation and keeping track of how much, where and why I have to do things differently from Conway, for example by following the approach of later works on the subject. For aesthetic reasons, I didn't want to be non-constructive without need, either. I defined and proved the additive group structure on games, multiplication, commutativity of multiplication and distributivity of multiplication (which are not detailed in this report), the latter with the (unproven) assumption that multiplication is a morphism for equality, which is true only for numbers (not all games).

This practical work has given me some experience on the limits of Coq. I also made some design mistakes in the encoding of the concepts in the CIC, and in the way to approach the proofs. These teachings form the personal results of my Master's Thesis. More general, scientific results are exposed in section 4.2.

4.1 Results

We now take a look in some detail at the points where things did go easily, and points where it didn't go as easily as in an ideal world, the rough spots I encountered, that could maybe be

improved, either by extending Coq-the-implementation, the CIC or by a different approach in the design of the encoding of the concepts in the CIC.

We will first examine how Coq passed the test points we specifically set up, namely dealing with sets, non-constructive mathematics, complex recursion and induction schemes and a defined equality. Then we will take a look at difficulties that cropped up during the work: We have to deal with explosively huge terms and two seemingly different points, but linked by the fact they both are infringements of the compulsive librarian principle, namely "Every scrap of information is precious, never throw any away".

4.1.0 Sets

Coq doesn't have a primitive notion of "set", but the standard library contains a module "Sets". Alas, these use the descending, characteristic function approach and thus are not adequate for surreal numbers-like (set-inductive) constructions. It looks relevant to add the ascending, index type and function, approach to the standard library

On a more basic level, the index type and function encoding works well enough. It makes things slightly more complicated (two objects instead of one), but the complication is similar to the one introduced by Conway's encoding of bisets in set theory, and can thus be called reasonable. The two ways to approach sets, their advantages and their limitations can reasonably be packed in a "set-theoretic mathematician's bag of tricks to use Coq", as part of the skill "being able to use Coq". I haven't compiled this package, but this kind of documentation would be essential to the widespread use of a proof assistant. It can not really be done by the set-theoretic community itself, it would be best done by someone familiar with both approaches, as a hand extended by the type theory community to its brother. The skill of being able to use Coq (or another specific proof assistant) is not directly mathematics-related, but it is more or less on the same level as the skill "be able to use TEX or LATEX", including for example knowing how to put something in bold.

4.1.1 Constructiveness

Coq's constructiveness introduced essentially one difficulty: the definition of the order is split up (in a mutual induction) between $\leq$ and $\lhd$. Each time one wants to use an hypothesis (or prove a goal) of the form $t_1 \leq t_2$, $\lhd$ rears its head. This is not a major practical problem: Once equivalence between $\neg \leq$ and $\lhd$ is proven, one can replace occurrences of one by the other. The equivalence between both is non-constructive, though. But this two-piece definition sure looks artificial and weird to the classical mathematician. He probably won't take as good news to be forced by Coq to take such detours.

It is also one of the points where I had to refer to a later work on the subject, namely [Ros01], although the notion of $\lhd$ is present in [Con01], in the "Games" part of the book.

4.1.2 Recursive Functions and Induction Schemes

Recursive definitions of functions and induction schemes are very tightly coupled: induction is at the type level what recursion is at the object level, proving an induction scheme correct

is proving that the dual function (proof constructor) terminates.

Coq wasn't really brilliant here: a lot of work had to be done by hand, in the construction of these fixpoint towers (embedded auxiliary functions). While this embedded recursions trick could also be classified as included in the skill of being able to use Coq, the problem grows too big to get rid of it in this way. When defining a function needs three auxiliary functions, which each need three auxiliaries to the auxiliary, which each need three auxiliaries themselves, the expression becomes too complicated to expect the user to handle it.

Coq, seen as a programming language, is quite impractical: it forces you to construct `FixPoint` towers, with auxiliary functions, etc. One would very much like to be able to give an ML-like definition, possibly annotated.

Dually, in this surreal numbers formalisation, we had to use induction schemes that Coq wouldn't generate for us, like it does for simple ones. We'd like them to be automatically generated or very simple to prove.

There are two directions that can be taken to address this issue:

1. Add tools that transform a "humanly reasonable" recursive definition into a definition of (extensionally) the same function accepted by the CIC, respectively automatically prove a reasonable induction scheme.
2. Extend the CIC to accept more recursive definitions primitively.

Transformation Tools

In fact, Coq used to have such a facility, `Recursive Definition`, that would take an ML-like definition and would try to turn it into a `FixPoint` tree. Obviously, as algorithm termination is not decidable, it didn't always succeed. Why has it been removed?

In complement to this facility, one would like also to be able to provide a proof of termination, to help a construct like `Recursive Definition` when it fails to do its work automatically, for example by providing a suitable well-founded order. The construct could then generate the proof goals "the recursive definition is decreasing according to this order", and use the proofs to construct a CIC-acceptable definition of the function.

On the induction schemes side, all the induction schemes used here are instances of a more generic class (conjecture 2.1.1), and it strongly looks like encoding their correctness in the CIC is long and boring, but very systematic and programmable. Just the kind of things computers are so good at!

Extending the CIC

On the recursive definitions side, all this complication basically comes from the fact that type theory cannot handle partial functions. It thus seems reasonable to think that if we could have partial functions (algorithms that are not known a priori to terminate) as objects, then we could define our total functions as partial functions, and prove them total (terminating) with the whole Coq machinery available. Venanzio Capretta and Ana Bove did a substantial step in this direction ([BC02], [BC01], [Bov02]) by giving a way to represent partial functions in type theory. Basically, they require an additional argument in the

function, that serves as proof of termination (in the form of a trace of computation) for the particular instances of the arguments of the function. One can then later prove the function total by constructing this additional argument for arbitrary arguments. Alas, this can not be used in the scheme outlined before (define a total function as partial, and prove it total later), because constructing that additional argument is exactly as complex as computing the function in the first place, it requires the same kind of recursion schemes.

On the inductions side, here is an idea that might be worth to explore further: The following slight extension of Conway's Induction principle (see page 51 of this report) covers the permuting inductions used in this work[1]:

> If P is some proposition that holds for $x_0, \ldots, x_n$ whenever it holds for some number of expressions of the form
>
> $$x_{\sigma(1)}, \ldots, x_{\sigma(i-1)}, x_{\sigma(i)_l}, x_{\sigma(i+1)}, \ldots, x_{\sigma(n)}$$
>
> or
>
> $$x_{\sigma(1)}, \ldots, x_{\sigma(i-1)}, x_{\sigma(i)_r}, x_{\sigma(i+1)}, \ldots, x_{\sigma(n)}$$
>
> where σ is a permutation and σ and i can be different for every expression, then P holds universally.

We could take inspiration from the above to slightly expand CIC's acceptance condition for recursive definitions (which includes the notion of "structurally smaller"), to make intricate recursion schemes (like the proof constructors of permuting induction schemes, and thus permuting induction schemes) acceptable in the CIC in their natural definition form.

The idea is to abandon the focus on one, and only one recursion variable, and accept recursion on several variables under conditions similar to the ones of conjecture 2.1.1:

- In every recursive call, all the recursion variable, or a term structurally smaller, occur.
- Maybe we need the condition that all recursion variables (or a structurally smaller term) appear in recursion slots (i.e. at the position where there was a recursion variable in the left hand-side of the definition), and/or the reciprocate: at every recursion slot, there is a recursion variable or a term structurally smaller.
- In every recursive call, at least one recursion variable does not occur (but a subterm thereof). Maybe with the additional condition "in a recursion slot".

Maybe this can be implemented without loosing the focus on one and only one recursion variable by packing the recursion variables in a tuple, and extending the notion of "structurally smaller than" ($\lessdot$) so that $(x_5, x_2, x_{1l}, x_3, x_4) \lessdot (x_1, x_2, x_3, x_4, x_5)$. It would be nice if this tuple-packing would be hidden from the user.

[1] In order to simplify the principle, I have limited us to the cases where in each induction hypothesis, exactly one induction variable "goes down". The case where more than one induction variable goes down is a natural extension.

Note that this extended acceptance condition (respectively structurally smaller than) looks much more difficult to programmatically decide[2] than Coq's: one would have to try every permutation of the recursion variables. A quick solution is requiring or permitting the user to explicitly give the right permutations. This heuristic should also help: in most cases, all but one (respectively, all but a few) recursive variables will be present in full (not through a subterm) in the recursive calls. So, first take those out and do the hard work (finding what term is smaller than what variable) only on the remaining. Here is another trail to follow: a subterm of `x` must have a (direct or indirect) reference to `x`: it is something like `(GLeftFun x)`, or if there is a case distinction on `x`, it contains a reference to the names introduced by this case distinction, ... So maybe we can fish out the permutation (in linear time in the number of permutation variables) from those references. But what about terms that contain references to *both* x and y (two recursion variables)? So maybe this approach won't help in the worst case, but will bring the common/average case down to linear time, like for bucket sort.

On the other hand, this extended acceptance condition is not harder to *check*, it would not cripple the system with regards to the de Bruijn criterion[3]. Indeed, the type checker doesn't have to find the right permutation, it suffices that it checks that the term type-checks with the permutation the user or the rest of the system gives.

4.1.3 Defined equality

Game equality is a defined relation, which is strictly weaker than identicality, which in turn is strictly weaker than Coq equality of inhabitants of the type `Game`. Coq has tools that help to reason modulo Coq equality. It would be extremely helpful if all these tools would be extended (or if other, similar, tools existed) to handle more general equivalence relations.

Currently, Coq has exactly one tool to help reason modulo an equivalence relation: The `Setoid` module. Alas, two points limit its usefulness:

1. It is well hidden in the Coq manual. Although it integrates with the usual `Rewrite` tactic, the documentation of `Rewrite` has no pointer to `Setoid`, and neither the table of contents, nor the index of the manual contain "equivalence relation" or "morphism", and "equality" leads only to Coq equality.

2. `Setoid` is not very integrated with the rest of Coq. Most other specialised tools work with Coq equality exclusively. For example, the sets of the standard library and the `Ring` tactic. Speaking about `Ring`, one wonders why Coq has a tactic specialised in reasoning modulo (semi-)ring properties, but not for groups.

4.1.4 Term rewriting

The reader is assumed to be somewhat familiar with the vocabulary of term rewriting.

One of the main practical problems encountered during the proof developments is that the subgoals generated grow huge, particularly when multiplication gets involved, to the

[2] As in "its worst case complexity is strictly greater than the one of"

[3] Have a small, simple type checker, so that confidence in correctness of its implementation can be high.

point where the memory consumption of Coq grows beyond the typical capacity of a current desktop computer (2^{28} to 2^{29} bytes). These large terms do not only take up a lot of resources (Coq memory consumption, time the front-end takes to parse Coq's answer[4], etc), but they are difficult for the human user to read, to the point where it becomes more convenient to do the calculations Coq performs by hand on paper, keeping the terms more "collapsed", and blindly give instructions to Coq while discarding its output. In some sense, this is re-implementing Coq in your brain. While the standard dodge to this kind of issues applies (wait a few years, you'll have machines big and fast enough), it is somewhat unsatisfying: the explosion looks exponential, what will happen with more complex structures? More important: human brains don't scale up. If Coq generates unnecessarily huge terms, the human user still has to read them, even if all computer parts involved are fast enough to handle the load well.

The explosion essentially comes from a conjunction of factors:

- Our inductive definitions, most notably `Glte`, are defined from the components of games, not from games themselves. For the specific example of `Glte`, it is defined like this[5]:

```
Inductive Glte : Game -> Game -> Prop :=
 Glte_cons : (xLI,xRI:Type)(xLf:xLI->Game)(xRf:xRI->Game)
             (yLI,yRI:Type)(yLf:yLI->Game)(yRf:yRI->Game)
             ((l:xLI) [xl:=(xLf l)]
                      (NGgte xl (Game_cons yLI yRI yLf yRf))) ->
             ((r:yRI) (NGgte (Game_cons xLI xRI xLf xRf) (yRf r))) ->
             (Glte (Game_cons xLI xRI xLf xRf)
                   (Game_cons  yLI yRI yLf yRf))
with NGgte ...
```

  and not like this:

```
Inductive Glte : Game -> Game -> Prop :=
 Glte_cons : (x:Game)(y:Game)
             ((l:(GLeftIndex  x)) [xl:=(GLeftFun x l)](NGgte xl y)) ->
             ((r:(GRightIndex y)) (NGgte x (GRightFun y r))) ->
             (Glte x y)
with NGgte ...
```

  The difference is subtle, but real. With the first definition, when the goal is something like `(Glte t1 t2)`, one can make the step "the proof of `(Glte t1 t2)` can be constructed with `Glte_cons`, thus proving the prerequisites of `Glte_cons` is enough", which we did numerous times with the `Exists` tactic, if and *only if* `t1` and `t2` match

[4]While the Coq memory is cheap (in the limit of what the processor can address), this parsing time is very expensive because is it human time, time the human waits.

[5]When compared to the definition given in section 2.1.0 on page 21, I have expanded the syntactic shortcuts `x` and `y` the former contains to make the structure more clear.

`(Game_cons LI RI Lf Rf)`. So, for example if `t1` is something like `(GPlus x y)`, it is not possible, `t1` must be converted until its root is `Game_cons`. Coq then generates subgoals containing the `LI`, `Lf`, ...

With the second definition, on the other hand, it is always possible: all that is needed is that `t1` and `t2` are recognised to be of type `Game`, which is already the case if Coq lets you have the term `(Glte t1 t2)`. Coq then generates subgoals using `(GLeftIndex t1)`, `(GLeftFun t1)`, ...

- The only conversion tactics Coq has are *normalisation* tactics. When the normal form is huge, but what is needed is only partial reduction, the alternative is to do the reduction steps needed by hand and feed the result to Coq, e.g. with the `Change` tactic. This is not satisfying: when one uses a proof assistant, it is in the hope it will do that kind of work for you.

  The only reduction that can really be controlled very granularly is δ-reduction, i.e. unfolding of definitions. This in turn can be used as a coarse control of the other normalisations: for example, in `(GPlus t1 t2)`, if the definition of `GPlus` is not unfolded, β-reduction will not occur there.

It is unclear at this point exactly how much the change of strategy in the inductive definitions would help by itself: to do something out of `(GLeftIndex (GPlus t1 t2))` (for example case analysis), one must reduce `(GPlus t1 t2)` to `Game_cons ...` anyway, so it might only postpone the problem. At the very least, it will help by limiting the explosion to one subgoal at a time instead of all at the same time.

There are some other factors, that while not a primary cause of the problem, make it worse or whose absence would be a partial solution:

- Expansion[6] is difficult/impossible. If Coq were able to recognise `GPlus` in the huge normal forms it computes, then the user could avoid being faced with this explosion: he could do something like `Compute; Fold GPlus.` (or maybe instead of `Fold`, a more powerful tactic called for example "Recognize"). Coq would then internally generate a big intermediate term (and take up a lot of memory), but never show it to the user. This expansion shouldn't be total, either, or it would just cancel the preceding `Compute`, so this somewhat joins the previous point: we need a *controlled* calculus.

- The conversion tactics apply only to the whole goal. There is no easy way to reduce only a subterm, short of giving this subterm `t1` a name `N`, doing δ-expansion for this name, using the conversion tactics on the hypothesis `N := t1`. This is exactly the kind of things that should be done under the hood.

Even addressing only these "worsening factors" would help tremendously.

4.1.5 /dev/null is not a good destination

In a move that was intended as a simplification step, I decided not to constantly jump between the three equalities on Conway Games, namely equality, identicality and Coq equality.

[6] Reduction backwards

The thought was "Do everything with equality, it simplifies the picture". By taking this too far, I did not only make my life more complicated in later proofs, but even made some things impossible to prove, because I had thrown away information about identicality or Coq-equality of terms, replacing them by the weaker game equality: weaker hypotheses, more difficult to prove results. For a specific example, let's take the `GPlus/GPlusAux` couple and distributivity. I encounter terms like `(GMult (GPlusAux y) z)`. Because I only have the theorem `Geq (GPlusAux y) (GPlus x y)`, and not `(GPlusAux y) == (GPlus x y)` (which, by the way, is true), I'm forced to assume compatibility of multiplication with equality, which is not true for general games, only numbers, thereby restricting my proof of distributivity to numbers.

Coq also throws information away, as shown by the example of `SillyWorkAround_0` on page 75: There is no way to do case analysis in Coq while keeping the link between the thing one is doing case analysis on and the new names introduced by Coq. For example, in this situation:

```
x: nat
P: nat -> Prop
===============
(P x)
```

there is no way to get to

```
x: nat
P: nat -> Prop
n: nat
H: x == (S n)
===============
(P (S n))

subgoal 2 is:
 (P 0)
```

Instead, one gets the same, but without the hypothesis `H`.

Because every occurrence of `x` is replaced by `(S n)`, one might think that the new hypothesis is useless. But a problem arises when the goal contains terms more generic than x, that are intended to be instantiated by `x` later in the proof, like happens in the proof of $x + (-x) \leq 0$.

4.1.6 Tainting

There is also no easy way to track what theorem depends on what. This would be particularly useful for axioms, i.e. what theorem is tainted by what axiom: the system would tell you which ones of your proofs are non-constructive (directly or indirectly depends on axiom `classic`), would give a better framework for speculative mathematics (if the Riemann hypothesis holds, then this too) and help the mathematician during theory development, by

helping him to track the assumptions he made to explore a track and whose proof he has postponed to later.

4.1.7 Formalising "similar"

Mathematical communication often uses "similar" and "similarly", as in "we have done one case, the others are similar". This is not (only) laziness from the writer, it enhances readability by peers, by showing only the most relevant bits of information. "More of the same" is not relevant. It is even considered inelegant to paste the same treatment of different cases over and over again. Yet, sometimes this similarity hides real problems, difficulties or issues, as we have seen with the example of transitivity of the order (section 2.1.3 on page 49).

While, in the context of a Coq-like proof assistant, treating similar cases is rather efficiently covered for the author by kill/yank (copy/paste) and textual search/replace, the result is unsatisfying: the resulting proof bears no trace that all 18 cases are essentially similar. The reader would have to read them all and deduce this himself. Even in the presence of a comment of the author stating that the 18 cases are similar, the reader wishing to test this assertion has to read all 18 cases in all their details. The exact link between the different cases is hidden, or at the very least not checked by the proof assistant. Additionally, should the author wish to change his proof, the kill/yank and search/replace work is to be done again.

There would thus be a good use for means to express this widely used "similar", in just enough details to make it unambiguous and checkable that the different cases are indeed similar. For example, if the first and second case are the same with x and z swapped, Coq could have a command `MergeGoals 1 2 with x:=z z:=x`.

But this is not enough. For example, in our formalisation of surreal numbers, typical inductions have a left case and a right case. Sometimes, these are exactly the same by replacing "left" by "right" in the proof, but this is not fully covered by term substitution, so some way to express this should be found. A step in this direction would be permitting the user to express similarity or duality between notions or lemmas, and later use this duality. In our example of lefts and rights, a left is dual to a right, what the left does with $\leq$, the right does with $\geq$. So if there were some way to express this duality, to give it the name `GLeftRightDual` and use it to express similarity between goals, to say that the second goal is the dual through this duality of the first one, it would be a big progress.

Sometimes, the subgoals generated from different goals are similar, but don't show up in the same order, so there should also be a way to express this reordering. A simple example: the two goals are $P \wedge Q$ and $R \wedge T$ and P is similar to T and Q to R.

A way to implement this in a Coq-like proof assistant would be to permit the user to get from the system the proof term of a previous goal, express transformations to be applied to it (things like substitution, mirroring/switching of parts of the term, etc.) and use the result as proof of the next goal. The user would then express the relationship between the similar goals instead of repeating the same argument over and over again. That is exactly what a human reader wants to read and it is much more pleasant than large search/replace operations for the author to write. The proof script could look like:

```
(* Two subgoals *)
```

```
(* I name the (not yet done) proof of the current goal "Case1" *)
NameProof Case1.
(* Here, I prove this goal *)
...
(* The current goal is now the second case *)
UseProof Case1 with x:=z GLeftRightDual AutoReorderSubgoals.
Qed.
```

This can be summed up by the following general approach: The proof is a mathematical object, just like any other. The proof assistant should have powerful tools to manipulate these objects.

This need is already partially fulfilled by the lemma mechanism. If the similar expressions are instances of a more general property that is both expressible in the language of the proof assistant and provable, then proving this more general property in a lemma and using this lemma repetitively is a good solution, provided the proof of the general lemma doesn't amount to a case analysis where exactly the same similar expressions show up again as separate goals. For example, let's suppose we have a way to express the notion of *option* of a surreal number, i.e. "a left or a right". I denote O_x the set of all options of x, i.e. $O_x := L_x \cup R_x$. Let's suppose that for some predicate P, we are faced with the two goals $\forall x_l \in L_x, P(x_l, x)$ and $\forall x_r \in R_x, P(x_r, x)$ (or maybe their instantiation with a particular x_l and x_r).If the proof of $\forall x_o \in O_x, P(x_o, x)$ can go only by case analysis on whether x_o is a left or a right of x, nothing is won by the more general lemma $\forall x_o \in O_x, P(x_o, x)$, compared to proving $\forall x_l \in L_x, P(x_l, x)$ and $\forall x_r \in R_x, P(x_r, x)$ separately.

4.2 Conclusions

My final conclusions fall into two categories: General findings and more specific remarks to and principles for proof assistant designers.

4.2.0 General findings

General findings:

1. Formalising a notion like surreal numbers in Coq is more complicated, takes more time than it should. After all, it shouldn't take much more time than neatly and completely writing it down in a book or an article. It was doable (meaning the CIC is formally powerful enough to deal with notions like surreal numbers), but too complicated.

2. Yet, it is relevant to do it. Even when the mathematician behind the theory is gifted, it still catches errors, or at the very least omissions, like the example of transitivity of the order has shown us, in section 2.1.3 on page 49.

4.2.1 Design principles

3. The documentation and standard library should be fairly complete and aimed towards the user, of various mathematical approaches. Documentation should include sections specifically for mathematicians of various traditions, listing how to obtain the concepts they are used to, if it cannot reasonably be done in the standard library. Give a way to browse through the standard library efficiently, with user-oriented descriptions of how to use the concepts defined in the library. Simply pretty-printing the definitions is not enough. A feature or a tool that isn't adequately documented, in an adequate place in the documentation, is a feature or tool no one besides you knows about.

4. Don't systematically throw away information. While it is essential that the system can do so (or at least hide) at the user's *explicit* request, so that the user isn't swamped in non-relevant information, never have the proof assistant throw away a particular information automatically. Even if you think this information is not useful. As we have seen with the example section 4.1.5, there will be a border case where the information will be needed, or at least, even if not formally needed, useful.

5. The proof assistant should track dependencies of results, with per-lemma (respectively theorem or axiom) granularity. Have a special case "show me only the axioms this result depends on".

6. Deal with similarity of different results, respectively proof obligations (goals). Let user express duality or similarity between notions and results and use this duality, respectively similarity, to derive a proof of similar results from the proof of a previous result.

7. Let the user in control of the calculus included in your proof assistant. Don't force him to go to a kind of normal form.

8. Deal with a defined equality (*any* equivalence relation) gracefully. Let the user hop easily from one equality to the other.

4.2.2 Closing

In my goal of formalising surreal numbers in Coq, I have partially succeeded. The essence is done, Coq is formally powerful enough to deal with it, but it took far too much time, preventing total completion, i.e. Coq is practically not convenient and efficient enough to do it. In my goal of putting Coq through a torture test, I'm rather successful: I now know much more about the state of proof assistant technology than before starting, I have identified areas where it can be improved and developed some ideas on how to improve it. But I failed to make use of the full potential of Coq, of all its tools, simply because they were not adequately documented. I made design errors in my formalisation that most probably could have been avoided if the documentation contained a list of "good style" recommendations.

Acknowledgements

I am deeply grateful to Rob Nederpelt for his insightful input on drafts of this report and continuous encouragements during the whole project.

Roel Bloo and Francien Dechesne helped me throughout this project. During scheduled and unscheduled meetings, they listened patiently to descriptions of the difficulties I encountered, providing suggestions. I thank them for this.

I thank Michel Reniers for joining the evaluation committee.

Additionally to the very idea of the subject, Freek Wiedijk provided occasions for me to take a step back from my work, organise my thoughts and summarise what I had done, by asking to be kept informed of my progress. He also provided valuable directions in early steps of the work.

Appendix A

Sign sequences

Surreal numbers can be defined as sign sequences (sequences of ordinal length of elements of $\{+, -\}$), rather than with "bisets", like [Con01] does. Sign sequences are introduced in chapter 3, on page 30 of [Con01]. At first glance, this looks similar to de Bruijn's way of constructing $\mathbb{R}$ without the rationals ([dB75]), formalised in AUTOMATH by J.T. Udding ([Udd80]).

I really haven't explored the sign-sequence version of surreal numbers, but the basic construction looks more "computery", more algorithmic. Because of my hidden agenda[0], I chose to follow Conway's biset approach. It would be interesting to formalise this approach (we would need a notion of ordinals first) in Coq and compare results.

[0]See sections 0.0 and 4.0.

Appendix B

Coq code

Here is the list of all notions defined and results proved in Coq, with the proofs themselves stripped away. The full Coq proof script is available from the author and on `http://www.mamane.lu/`.

```
Section Conway_Games.

Inductive Game : Type :=
 Game_cons : (LI,RI:Type) (LI -> Game) -> (RI -> Game) -> Game.

Inductive AdHocprod : Type := AdHocpair : (A:Type)(A -> Game) -> AdHocprod.

Definition GLeft := [G:Game] Cases G of (Game_cons A B f g) => (AdHocpair A f) end.
Definition GRight := [G:Game] Cases G of (Game_cons A B f g) => (AdHocpair B g) end.
Definition GLeftIndex := [G:Game] Cases G of (Game_cons A B f g) => A end.
Definition GRightIndex := [G:Game] Cases G of (Game_cons A B f g) => B end.
Definition GLeftFun := [G:Game]  <[G:Game](GLeftIndex G)->Game>Cases G of (Game_cons A B f g) => f end.
Definition GRightFun := [G:Game] <[G:Game](GRightIndex G)->Game>Cases G of (Game_cons A B f g) => g end.

Theorem AlternateInduction:
        (P,Q:Game->Game->Prop)(P1,P2,Q1,Q2:Game->Game->Prop)
        ((x,z:Game) (P x z) -> ((xli:(GLeftIndex  x))[xl:=((GLeftFun  x) xli)](P1 xl z)) /\
                               ((zri:(GRightIndex z))[zr:=((GRightFun z) zri)](P2 x zr)) ) ->
        ((x,z:Game) ((xli:(GLeftIndex  x))[xl:=((GLeftFun  x) xli)](P1 xl z)) /\
                    ((zri:(GRightIndex z))[zr:=((GRightFun z) zri)](P2 x zr)) -> (P x z)) ->
        ((x,z:Game) (Q x z) -> ((zli:(GLeftIndex  z))[zl:=((GLeftFun  z) zli)](Q1 x zl)) /\
                               ((xri:(GRightIndex x))[xr:=((GRightFun x) xri)](Q2 xr z)) ) ->
        ((x,z:Game) ((zli:(GLeftIndex  z))[zl:=((GLeftFun  z) zli)](Q1 x zl)) /\
                    ((xri:(GRightIndex x))[xr:=((GRightFun x) xri)](Q2 xr z)) -> (Q x z)) ->
        ((x,z:Game)(xli:(GLeftIndex  x))[xl:=((GLeftFun  x) xli)] (Q xl z)->(P1 xl z)) ->
        ((x,z:Game)(zri:(GRightIndex z))[zr:=((GRightFun z) zri)] (Q x zr)->(P2 x zr)) ->
        ((x,z:Game)(zli:(GLeftIndex  z))[zl:=((GLeftFun  z) zli)] (P x zl)->(Q1 x zl)) ->
        ((x,z:Game)(xri:(GRightIndex x))[xr:=((GRightFun x) xri)] (P xr z)->(Q2 xr z)) ->
        (x,z:Game)((P x z) /\ (Q x z)).

Theorem Rol3Induction: (P:Game->Game->Game->Prop)
                         ((x,y,z:Game)((xli:(GLeftIndex  x))[xl:=(GLeftFun  x xli)](P y z xl)) ->
                       ((zri:(GRightIndex z))[zr:=(GRightFun z zri)](P zr x y)) ->
                                    (P x y z)) ->
                         ((x,y,z:Game) (P x y z)).


Theorem Roll3Induction: (P,P1,P2:Game->Game->Prop)[F:=[x,y,z:Game](P x y)->(P y z)->(P x z)]
                      ((x,z:Game) ((xli:(GLeftIndex  x))[xl:=(GLeftFun  x xli)](P1 xl z)) ->
                                   ((zri:(GRightIndex z))[zr:=(GRightFun z zri)](P2 x zr)) ->
                                   (P x z)) ->
                       ((x,z:Game) (P x z)-> ((xli:(GLeftIndex  x))[xl:=(GLeftFun  x xli)](P1 xl z)) /\
                                   ((zri:(GRightIndex z))[zr:=(GRightFun z zri)](P2 x zr))) ->
```

```
                    ((x,y,z:Game) (P1 x y) -> (P y z) -> (F y z x) -> (P1 x z)) ->
                    ((x,y,z:Game) (P2 y z) -> (P x y) -> (F z x y) -> (P2 x z)) ->
      ((x,y,z:Game) (F x y z)).

Lemma EmptyGameSet : False -> Game.
Proof.
Intro.
Contradiction.
Defined.

Inductive OnePointIndex:Set := OnePoint:OnePointIndex.

Inductive sumT [A,B:Type] : Type :=
   inlT : A -> (sumT A B)
 | inrT : B -> (sumT A B).

Definition map_on_sumT := [A,B,C:Type] [fa:A->C;fb:B->C][x:(sumT A B)]
 Cases x of
   (inlT e) => (fa e)
   |(inrT e) => (fb e)
 end.

(* The following two are a kind of "for all":
   They calculate wether
   PR (resp. PL) is true on their third argument
   PL (resp. PR) is true on the left (resp. right) elements of it
   etc, alternating
 *)

Fixpoint RecurseAlternateLeft[PL:(Game->Prop);PR:(Game->Prop);g2:Game] : Prop :=
  <Prop> Case g2 of
    [LI,RI:Type;Lf:(LI->Game);Rf:(RI->Game)]
     ((l:LI) (RecurseAlternateRight PL PR (Lf l)))/\(PR g2)
   end
with RecurseAlternateRight[PL:(Game->Prop);PR:(Game->Prop);g2:Game] : Prop :=
  <Prop>Case g2 of
    [LI,RI:Type;Lf:(LI->Game);Rf:(RI->Game)]
     ((r:RI) (RecurseAlternateLeft PL PR (Rf r)))/\(PL g2)
   end
.

Theorem RecurseAlternateLeftThusLevel1: (PL,PR:Game->Prop)(g:Game) (RecurseAlternateLeft PL PR g) -> (PR g).

Definition Zero := (Game_cons False False EmptyGameSet EmptyGameSet).
Definition One := (Game_cons OnePointIndex False ([x:OnePointIndex] Zero) EmptyGameSet).

Inductive Glte : Game -> Game -> Prop :=
 Glte_cons : (xLI,xRI:Type)(xLf:xLI->Game)(xRf:xRI->Game)(yLI,yRI:Type)(yLf:yLI->Game)(yRf:yRI->Game)
             [x:=(Game_cons xLI xRI xLf xRf);y:=(Game_cons  yLI yRI yLf yRf)]
     ((l:xLI) [xl:=(xLf l)](NGgte xl y)) -> ((r:yRI) (NGgte x (yRf r))) ->
             (Glte x y)
with NGgte : Game -> Game -> Prop :=
 NGgte_xr : (xLI,xRI:Type)(xLf:xLI->Game)(xRf:xRI->Game)(y:Game)
             [x:=(Game_cons xLI xRI xLf xRf)]
     (exT xRI [r:xRI] (Glte (xRf r) y)) ->
             (NGgte x y)
|NGgte_yl : (x:Game)(yLI,yRI:Type)(yLf:yLI->Game)(yRf:yRI->Game)
             [y:=(Game_cons  yLI yRI yLf yRf)]
     (exT yLI [l:yLI] (Glte x (yLf l))) ->
             (NGgte x y)
.

Fixpoint AntiGame [g:Game] : Game :=
 Cases g of
   (Game_cons LI RI Lf Rf) => (Game_cons RI LI ([i:RI] (AntiGame (Rf i))) ([i:LI] (AntiGame (Lf i))) )
 end.
```

```
Fixpoint GPlus [x:Game] : Game->Game := [y:Game]
 Cases x of
   (Game_cons xLI xRI xLf xRf) =>
   Cases y of
     (Game_cons yLI yRI yLf yRf) =>
      let GPlusAux =
      Fix GPlusAux { GPlusAux [z:Game] : Game :=
      (* (GPlusAux) == (GPlus x) *)
       Cases z of
 (Game_cons zLI zRI zLf zRf) =>
 (Game_cons
   (sumT xLI zLI) (sumT xRI zRI)
   (map_on_sumT xLI zLI Game
     ([xli:xLI] (GPlus (xLf xli) z))
     ([zli:zLI] (GPlusAux (zLf zli)))
   )
   (map_on_sumT xRI zRI Game
     ([xri:xRI] (GPlus (xRf xri) z))
     ([zri:zRI] (GPlusAux (zRf zri)))
   )
 )
       end
     }
      in
     (Game_cons
       (sumT xLI yLI) (sumT xRI yRI)
       (map_on_sumT xLI yLI Game
 ([xli:xLI] (GPlus (xLf xli) y))
 ([yli:yLI] (GPlusAux (yLf yli)))
       )
       (map_on_sumT xRI yRI Game
 ([xri:xRI] (GPlus (xRf xri) y))
 ([yri:yRI] (GPlusAux (yRf yri)))
       )
     )
   end
 end
.

Theorem IS:  (P:Game->Game->Prop)
             ((x,y:Game)
              ((yli:(GLeftIndex y))[yl:=((GLeftFun y) yli)] (P x yl)) ->
      ((xri:(GRightIndex x))[xr:=((GRightFun x) xri)] (P xr y)) ->
      (P x y)) ->
     (x,y:Game) (P x y).

Theorem IS2:  (P:Game->Game->Prop)
             ((x,y:Game)
              ((yli:(GLeftIndex y))[yl:=((GLeftFun y) yli)] (P yl x)) ->
      ((xri:(GRightIndex x))[xr:=((GRightFun x) xri)] (P y xr)) ->
      (P x y)) ->
     (x,y:Game) (P x y).

(* This one is IS2, swapping x and y in the premises, but not in the conclusion *)

Theorem IS2Reverse:  (P:Game->Game->Prop)
             ((x,y:Game)
              ((xli:(GLeftIndex x))[xl:=((GLeftFun x) xli)] (P y xl)) ->
      ((yri:(GRightIndex y))[yr:=((GRightFun y) yri)] (P yr x)) ->
      (P x y)) ->
     (x,y:Game) (P x y).

Theorem NGgteIsNGgte_Step :
             (xLI,xRI:Type)(xLf:xLI->Game)(xRf:xRI->Game)(yLI,yRI:Type)(yLf:yLI->Game)(yRf:yRI->Game)
             [x:=(Game_cons xLI xRI xLf xRf);y:=(Game_cons  yLI yRI yLf yRf)]
             ((yli:yLI)[yl:=(yLf yli)] (NGgte yl x) -> ~(Glte x yl)) ->
     ((xri:xRI)[xr:=(xRf xri)] (NGgte y xr) -> ~(Glte xr y)) ->
```

```
     ((NGgte x y) -> ~(Glte y x)).

Theorem NGgteIsNGgte :  (x,y:Game)
             (NGgte x y) -> ~(Glte y x).

Theorem NGgteGgteAbsurd : (x,y:Game)
             (NGgte x y) -> (Glte y x) -> False.

Theorem NGgteToProp: (x,y:Game) (NGgte x y) ->
                     (exT (GRightIndex x) [r:(GRightIndex x)] (Glte (GRightFun x r) y)) \/
                     (exT (GLeftIndex  y) [l:(GLeftIndex  y)] (Glte x (GLeftFun y l))).

Theorem NNGgteToProp: (x,y:Game) ~(NGgte x y) ->
                      ((r:(GRightIndex x)) ~(Glte (GRightFun x r) y)) /\
                      ((l:(GLeftIndex y)) ~(Glte x (GLeftFun y l))).

Axiom Classic: (P:Prop) ~~P->P.

Theorem NGgteNGgte : (x,y:Game) ~(NGgte x y) -> (Glte y x).

Theorem nGlteNGgte : (x,y:Game) ~(Glte x y) -> (NGgte y x).

Theorem SomeName:(xLI,xRI:Type)(xLf:xLI->Game)(xRf:xRI->Game)(yLI,yRI:Type)(yLf:yLI->Game)(yRf:yRI->Game)
            [x:=(Game_cons xLI xRI xLf xRf);y:=(Game_cons  yLI yRI yLf yRf)]
    (xri:xRI) (Glte (xRf xri) y) -> (Glte y x) -> False.

Theorem Glte_transitive : (x,z,y:Game) (Glte x y) -> (Glte y z) -> (Glte x z).

Theorem Glte_reflexive: (x:Game) (Glte x x).

(* Glte is a pre-order *)

Definition Geq:Game->Game->Prop := [x,y:Game](Glte x y) /\ (Glte y x).

Theorem Geq_symmetric: (x,y:Game) (Geq x y) -> (Geq y x).

Theorem Geq_transitive: (x,y,z:Game) (Geq x y) -> (Geq y z) -> (Geq x z).

Theorem Geq_reflexive: (x:Game) (Geq x x).

Inductive Gidentical:Game->Game->Prop :=
  Gidentical_cons : (x,y:Game)
               ((xli:(GLeftIndex  x))
                (exT (GLeftIndex  y)
                     [yli:(GLeftIndex  y)](Gidentical (GLeftFun  x xli) (GLeftFun  y yli))))  ->
               ((xri:(GRightIndex x))
                (exT (GRightIndex y)
                     [yri:(GRightIndex y)](Gidentical (GRightFun x xri) (GRightFun y yri))))  ->
               ((yli:(GLeftIndex  y))
                (exT (GLeftIndex  x)
                     [xli:(GLeftIndex  x)](Gidentical (GLeftFun  y yli) (GLeftFun  x xli))))  ->
               ((yri:(GRightIndex y))
                (exT (GRightIndex x)
                     [xri:(GRightIndex x)](Gidentical (GRightFun y yri) (GRightFun x xri))))  ->
 (Gidentical x y).

Theorem Gidentical_reflexive: (x:Game) (Gidentical x x).

Theorem Gidentical_commutative: (x,y:Game) (Gidentical y x) -> (Gidentical x y).

Theorem Gidentical_transitive: (x,y,z:Game) (Gidentical x y)->(Gidentical y z)->(Gidentical x z).

Theorem NGgte_Conway_Definition :
        (xLI,xRI:Type)(xLf:xLI->Game)(xRf:xRI->Game)(yLI,yRI:Type)(yLf:yLI->Game)(yRf:yRI->Game)
        [x:=(Game_cons xLI xRI xLf xRf);y:=(Game_cons  yLI yRI yLf yRf)]
        (Glte x y) -> ((l:xLI) ~(Glte y (xLf l))) /\ ((r:yRI) ~(Glte (yRf r) x)).
```

```
Theorem NGgte_Conway_Definition2 :
        (xLI,xRI:Type)(xLf:xLI->Game)(xRf:xRI->Game)(yLI,yRI:Type)(yLf:yLI->Game)(yRf:yRI->Game)
        [x:=(Game_cons xLI xRI xLf xRf);y:=(Game_cons  yLI yRI yLf yRf)]
        ((l:xLI) ~(Glte y (xLf l))) -> ((r:yRI) ~(Glte (yRf r) x)) -> (Glte x y).

Inductive IsNumber : Game->Prop :=
  IsNumber_cons : (x:Game) ((xli:(GLeftIndex  x))  (IsNumber (GLeftFun x xli))) ->
                           ((xri:(GRightIndex x)) (IsNumber (GRightFun x xri))) ->
                           ((xli:(GLeftIndex x))(xri:(GRightIndex x))
                            (NGgte (GLeftFun x xli) (GRightFun x xri))) ->
                           (IsNumber x)
.

Lemma ZeroIsNumber: (IsNumber Zero).

Lemma OneIsNumber: (IsNumber One).

Lemma AntiAnti:(x:Game) (Geq (AntiGame (AntiGame x)) x).

Lemma GlteNGgteAnti:(x,y:Game) ((Glte  x y) -> (Glte  (AntiGame y) (AntiGame x))) /\
                               ((NGgte y x) -> (NGgte (AntiGame x) (AntiGame y))).

Lemma GlteAnti:(x,y:Game) (Glte x y) -> (Glte (AntiGame y) (AntiGame x)).

Theorem GeqAnti: (x,y:Game)(Geq x y)->(Geq (AntiGame y) (AntiGame x)).

Theorem AntiGlte: (x,y:Game)(Glte (AntiGame y) (AntiGame x)) -> (Glte x y).

Lemma NGgteAnti:(x,y:Game) (NGgte x y) -> (NGgte (AntiGame y) (AntiGame x)).

Lemma Number_AntiGame_closed: (x:Game) (IsNumber x) -> (IsNumber (AntiGame x)).

Theorem GPlusxGPlusAux: (x,y:Game)
      [xLI:=(GLeftIndex x);xRI:=(GRightIndex x);xLf:=(GLeftFun x);xRf:=(GRightFun x)]
      let GPlusAux =
      Fix GPlusAux { GPlusAux [z:Game] : Game :=
      (* (GPlusAux) == (GPlus x) *)
       Cases z of
 (Game_cons zLI zRI zLf zRf) =>
 (Game_cons
   (sumT xLI zLI) (sumT xRI zRI)
   (map_on_sumT xLI zLI Game
     ([xli:xLI] (GPlus (xLf xli) z))
     ([zli:zLI] (GPlusAux (zLf zli)))
   )
   (map_on_sumT xRI zRI Game
     ([xri:xRI] (GPlus (xRf xri) z))
     ([zri:zRI] (GPlusAux (zRf zri)))
   )
 )
       end
     }
      in
(Geq (GPlus x y) (GPlusAux y)).

Definition GPlusAux := [x:Game]
      [xLI:=(GLeftIndex x);xRI:=(GRightIndex x);xLf:=(GLeftFun x);xRf:=(GRightFun x)]
      Fix GPlusAux { GPlusAux [z:Game] : Game :=
      (* (GPlusAux) == (GPlus x) *)
       Cases z of
 (Game_cons zLI zRI zLf zRf) =>
 (Game_cons
   (sumT xLI zLI) (sumT xRI zRI)
   (map_on_sumT xLI zLI Game
     ([xli:xLI] (GPlus (xLf xli) z))
     ([zli:zLI] (GPlusAux (zLf zli)))
   )
```

```
   (map_on_sumT xRI zRI Game
     ([xri:xRI] (GPlus (xRf xri) z))
     ([zri:zRI] (GPlusAux (zRf zri)))
   )
 )
       end
     }.

Theorem GPlusCommutative_one: (x,y:Game)(Glte (GPlus x y) (GPlus y x)).

Theorem GPlusCommutative: (x,y:Game)(Geq (GPlus x y) (GPlus y x)).

Theorem GPlusOrder: (x,y,z:Game) ((Glte  x y) -> (Glte  (GPlus x z) (GPlus y z))) /\
                                 ((NGgte x y) -> (NGgte (GPlus x z) (GPlus y z))).

Theorem GPlusGlteLeft: (x,y,z:Game) (Glte x y) -> (Glte (GPlus x z) (GPlus y z)).

Theorem GPlusGlteRight: (x,y,z:Game) (Glte x y) -> (Glte (GPlus z x) (GPlus z y)).

Theorem GPlusGlteBoth: (x1,x2,y1,y2:Game) (Glte x1 x2) -> (Glte y1 y2) -> (Glte (GPlus x1 y1) (GPlus x2 y2)).

Theorem GPlusAssociative_one: (x,y,z:Game) (Glte (GPlus x (GPlus y z)) (GPlus (GPlus x y) z)).

Theorem GPlusAssociative: (x,y,z:Game) (Geq (GPlus x (GPlus y z)) (GPlus (GPlus x y) z)).

Theorem AntiGPlus_one : (x,y:Game)(Glte (AntiGame (GPlus x y)) (GPlus (AntiGame x) (AntiGame y))).

Theorem AntiGPlus_two : (x,y:Game)(Glte (GPlus (AntiGame x) (AntiGame y)) (AntiGame (GPlus x y))).

Theorem AntiGPlus : (x,y:Game)(Geq (AntiGame (GPlus x y)) (GPlus (AntiGame x) (AntiGame y))).

Section SillyWorkAround_0.
Variable xLI : Type.
Variable xRI : Type.
Variable xLf : xLI->Game.
Variable xRf : xRI->Game.
Variable xli : xLI.
Variable xri:xRI.
Definition SillyWorkAround_0_xl := (xLf xli) : Game.
Definition SillyWorkAround_0_xr := (xRf xri) : Game.
Lemma SillyWorkAround_0_L0 :  (Glte (GPlus (xLf xli) (AntiGame SillyWorkAround_0_xl)) Zero)
   ->(NGgte
       (Fix GPlus
          {GPlus [x0:Game] : Game->Game :=
             [y:Game]
              Cases x0 of
                (Game_cons xLI0 xRI0 xLf0 xRf0) =>
                 Cases y of
                   (Game_cons yLI yRI yLf yRf) =>
                    (Game_cons (sumT xLI0 yLI) (sumT xRI0 yRI)
                      [x1:(sumT xLI0 yLI)]
                       Cases x1 of
                         (inlT e) => (GPlus (xLf0 e) y)
                       | (inrT e) =>
                          (Fix GPlusAux
                             {GPlusAux [z:Game] : Game :=
                                Cases z of
                                  (Game_cons zLI zRI zLf zRf) =>
                                   (Game_cons (sumT xLI0 zLI)
                                     (sumT xRI0 zRI)
                                     [x2:(sumT xLI0 zLI)]
                                      Cases x2 of
                                        (inlT e0) =>
                                         (GPlus (xLf0 e0) z)
                                      | (inrT e0) =>
                                         (GPlusAux (zLf e0))
                                      end
```

```
                                [x2:(sumT xRI0 zRI)]
                                 Cases x2 of
                                   (inlT e0) =>
                                    (GPlus (xRf0 e0) z)
                                 | (inrT e0) =>
                                    (GPlusAux (zRf e0))
                                 end)
                            end} (yLf e))
                     end
                    [x1:(sumT xRI0 yRI)]
                     Cases x1 of
                       (inlT e) => (GPlus (xRf0 e) y)
                     | (inrT e) =>
                        (Fix GPlusAux
                           {GPlusAux [z:Game] : Game :=
                              Cases z of
                                (Game_cons zLI zRI zLf zRf) =>
                                 (Game_cons (sumT xLI0 zLI)
                                   (sumT xRI0 zRI)
                                   [x2:(sumT xLI0 zLI)]
                                    Cases x2 of
                                      (inlT e0) =>
                                       (GPlus (xLf0 e0) z)
                                    | (inrT e0) =>
                                       (GPlusAux (zLf e0))
                                    end
                                   [x2:(sumT xRI0 zRI)]
                                    Cases x2 of
                                      (inlT e0) =>
                                       (GPlus (xRf0 e0) z)
                                    | (inrT e0) =>
                                       (GPlusAux (zRf e0))
                                    end)
                            end} (yRf e))
                  end)
             end
          end} (xLf xli)
       (Game_cons xRI xLI
         [i:xRI]
          (Fix AntiGame
             {AntiGame [g:Game] : Game :=
                Cases g of
                  (Game_cons LI RI Lf Rf) =>
                   (Game_cons RI LI [i0:RI](AntiGame (Rf i0))
                     [i0:LI](AntiGame (Lf i0)))
             end} (xRf i))
         [i:xLI]
          (Fix AntiGame
             {AntiGame [g:Game] : Game :=
                Cases g of
                  (Game_cons LI RI Lf Rf) =>
                   (Game_cons RI LI [i0:RI](AntiGame (Rf i0))
                     [i0:LI](AntiGame (Lf i0)))
             end} (xLf i)))) Zero).

Lemma SillyWorkAround_0_L1 :  (Glte (GPlus SillyWorkAround_0_xr (AntiGame (xRf xri))) Zero) ->

 [xPlusAntixr:=(Fix GPlusAux
        {GPlusAux [z:Game] : Game :=
           Cases z of
             (Game_cons zLI zRI zLf zRf) =>
              (Game_cons (sumT xLI zLI) (sumT xRI zRI)
                [x:(sumT xLI zLI)]
                 Cases x of
                   (inlT e) =>
                    (Fix GPlus
                       {GPlus [x0:Game] : Game->Game :=
```

```
[y:Game]
 Cases x0 of
   (Game_cons xLI xRI xLf xRf) =>
    Cases y of
      (Game_cons yLI yRI yLf yRf) =>
       (Game_cons (sumT xLI yLI)
         (sumT xRI yRI)
         [x1:(sumT xLI yLI)]
          Cases x1 of
            (inlT e0) => (GPlus (xLf e0) y)
          | (inrT e0) =>
             (Fix GPlusAux0
                {GPlusAux0 [z0:Game]
                   : Game :=
                   Cases z0 of
                     (Game_cons zLI0 zRI0
                       zLf0 zRf0) =>
                      (Game_cons
                        (sumT xLI zLI0)
                        (sumT xRI zRI0)
                        [x2:(sumT xLI zLI0)]
                         Cases x2 of
                           (inlT e1) =>
                            (GPlus (
                             xLf e1) z0)
                         | (inrT e1) =>
                            (GPlusAux0
                              (zLf0 e1))
                         end
                        [x2:(sumT xRI zRI0)]
                         Cases x2 of
                           (inlT e1) =>
                            (GPlus (
                             xRf e1) z0)
                         | (inrT e1) =>
                            (GPlusAux0
                              (zRf0 e1))
                         end)
                  end} (yLf e0))
          end
         [x1:(sumT xRI yRI)]
          Cases x1 of
            (inlT e0) => (GPlus (xRf e0) y)
          | (inrT e0) =>
             (Fix GPlusAux0
                {GPlusAux0 [z0:Game]
                   : Game :=
                   Cases z0 of
                     (Game_cons zLI0 zRI0
                       zLf0 zRf0) =>
                      (Game_cons
                        (sumT xLI zLI0)
                        (sumT xRI zRI0)
                        [x2:(sumT xLI zLI0)]
                         Cases x2 of
                           (inlT e1) =>
                            (GPlus (
                             xLf e1) z0)
                         | (inrT e1) =>
                            (GPlusAux0
                              (zLf0 e1))
                         end
                        [x2:(sumT xRI zRI0)]
                         Cases x2 of
                           (inlT e1) =>
                            (GPlus (
                             xRf e1) z0)
```

```
                                    | (inrT e1) =>
                                       (GPlusAux0
                                         (zRf0 e1))
                                    end)
                                end} (yRf e0))
                        end)
                    end
                end} (xLf e) z)
 | (inrT e) => (GPlusAux (zLf e))
 end
[x:(sumT xRI zRI)]
 Cases x of
   (inlT e) =>
    (Fix GPlus
       {GPlus [x0:Game] : Game->Game :=
          [y:Game]
           Cases x0 of
             (Game_cons xLI xRI xLf xRf) =>
              Cases y of
                (Game_cons yLI yRI yLf yRf) =>
                 (Game_cons (sumT xLI yLI)
                   (sumT xRI yRI)
                   [x1:(sumT xLI yLI)]
                    Cases x1 of
                      (inlT e0) => (GPlus (xLf e0) y)
                    | (inrT e0) =>
                       (Fix GPlusAux0
                          {GPlusAux0 [z0:Game]
                             : Game :=
                             Cases z0 of
                               (Game_cons zLI0 zRI0
                                  zLf0 zRf0) =>
                                (Game_cons
                                  (sumT xLI zLI0)
                                  (sumT xRI zRI0)
                                  [x2:(sumT xLI zLI0)]
                                   Cases x2 of
                                     (inlT e1) =>
                                      (GPlus (
                                       xLf e1) z0)
                                   | (inrT e1) =>
                                      (GPlusAux0
                                        (zLf0 e1))
                                   end
                                  [x2:(sumT xRI zRI0)]
                                   Cases x2 of
                                     (inlT e1) =>
                                      (GPlus (
                                       xRf e1) z0)
                                   | (inrT e1) =>
                                      (GPlusAux0
                                        (zRf0 e1))
                                   end)
                          end} (yLf e0))
                    end
                   [x1:(sumT xRI yRI)]
                    Cases x1 of
                      (inlT e0) => (GPlus (xRf e0) y)
                    | (inrT e0) =>
                       (Fix GPlusAux0
                          {GPlusAux0 [z0:Game]
                             : Game :=
                             Cases z0 of
                               (Game_cons zLI0 zRI0
                                  zLf0 zRf0) =>
                                (Game_cons
                                  (sumT xLI zLI0)
```

```
                                                    (sumT xRI zRI0)
                                                    [x2:(sumT xLI zLI0)]
                                                     Cases x2 of
                                                       (inlT e1) =>
                                                        (GPlus (
                                                         xLf e1) z0)
                                                     | (inrT e1) =>
                                                        (GPlusAux0
                                                          (zLf0 e1))
                                                     end
                                                    [x2:(sumT xRI zRI0)]
                                                     Cases x2 of
                                                       (inlT e1) =>
                                                        (GPlus (
                                                         xRf e1) z0)
                                                     | (inrT e1) =>
                                                        (GPlusAux0
                                                          (zRf0 e1))
                                                     end)
                                                end} (yRf e0))
                                        end)
                                    end
                                end} (xRf e) z)
                        | (inrT e) => (GPlusAux (zRf e))
                        end)
                end}
            (Fix AntiGame
               {AntiGame [g:Game] : Game :=
                  Cases g of
                    (Game_cons LI RI Lf Rf) =>
                     (Game_cons RI LI [i:RI](AntiGame (Rf i))
                       [i:LI](AntiGame (Lf i)))
                  end} (xRf xri))))](NGgte xPlusAntixr Zero).
End SillyWorkAround_0.

Theorem xPlusAntixZero_one : (x:Game)(Glte (GPlus x (AntiGame x)) Zero).

Theorem xPlusAntixZero : (x:Game)(Geq (GPlus x (AntiGame x)) Zero).

Lemma GltexGPlusxZero: (x:Game)(Glte x (GPlus x Zero)).

Lemma GltexGPlusZerox: (x:Game)(Glte x (GPlus Zero x)).

Lemma GlteGPlusxZerox: (x:Game)(Glte (GPlus x Zero) x).

Lemma GlteGPlusZeroxx: (x:Game)(Glte (GPlus Zero x) x).

(***** We now have "(No, +) is a commutative group" *****)

Fixpoint GMult [x:Game] : Game->Game := [y:Game]
 Cases x of
   (Game_cons xLI xRI xLf xRf) =>
  Cases y of
     (Game_cons yLI yRI yLf yRf) =>

      let GMultAux =
      Fix GMultAux { GMultAux [z:Game] : Game :=
      (* (GPlusAux) == (GPlus x) *)
       Cases z of
         (Game_cons zLI zRI zLf zRf) =>
 (Game_cons
   (sumT (prodT xLI zLI) (prodT xRI zRI))
   (sumT (prodT xLI zRI) (prodT xRI zLI))
   (map_on_sumT (prodT xLI zLI) (prodT xRI zRI) Game
             ([lli:(prodT xLI zLI)] let (xli,zli)=lli in [xl:=(xLf xli)][zl:=(zLf zli)]
               (GPlus (GPlus (GMult xl z) (GMultAux zl)) (AntiGame (GMult xl zl))))
             ([rri:(prodT xRI zRI)] let (xri,zri)=rri in [xr:=(xRf xri)][zr:=(zRf zri)]
```

```
            (GPlus (GPlus (GMult xr z) (GMultAux zr)) (AntiGame (GMult xr zr))))
  )
  (map_on_sumT (prodT xLI zRI) (prodT xRI zLI) Game
           ([lri:(prodT xLI zRI)] let (xli,zri)=lri in [xl:=(xLf xli)][zr:=(zRf zri)]
             (GPlus (GPlus (GMult xl z) (GMultAux zr)) (AntiGame (GMult xl zr))))
           ([rli:(prodT xRI zLI)] let (xri,zli)=rli in [xr:=(xRf xri)][zl:=(zLf zli)]
             (GPlus (GPlus (GMult xr z) (GMultAux zl)) (AntiGame (GMult xr zl))))
  )
)
      end
    }
     in
    (Game_cons
      (sumT (prodT xLI yLI) (prodT xRI yRI))
      (sumT (prodT xLI yRI) (prodT xRI yLI))
      (map_on_sumT (prodT xLI yLI) (prodT xRI yRI) Game
        ([lli:(prodT xLI yLI)] let (xli,yli)=lli in
                               [xl:=(xLf xli)][yl:=(yLf yli)]
                               (GPlus (GPlus (GMult xl y) (GMultAux yl)) (AntiGame (GMult xl yl))))
        ([rri:(prodT xRI yRI)] let (xri,yri)=rri in
                               [xr:=(xRf xri)][yr:=(yRf yri)]
                               (GPlus (GPlus (GMult xr y) (GMultAux yr)) (AntiGame (GMult xr yr))))
      )
      (map_on_sumT (prodT xLI yRI) (prodT xRI yLI) Game
        ([lri:(prodT xLI yRI)] let (xli,yri)=lri in
                               [xl:=(xLf xli)][yr:=(yRf yri)]
                               (GPlus (GPlus (GMult xl y) (GMultAux yr)) (AntiGame (GMult xl yr))))
        ([rli:(prodT xRI yLI)] let (xri,yli)=rli in
                               [xr:=(xRf xri)][yl:=(yLf yli)]
                               (GPlus (GPlus (GMult xr y) (GMultAux yl)) (AntiGame (GMult xr yl))))
      )
    )
  end
end
.

Definition GMultAux := [x:Game][xLI:=(GLeftIndex x)][xRI:=(GRightIndex x)]
                       [xLf:=(GLeftFun x)][xRf:=(GRightFun x)]
     Fix GMultAux { GMultAux [z:Game] : Game :=
     (* (GPlusAux) == (GPlus x) *)
      Cases z of
        (Game_cons zLI zRI zLf zRf) =>
(Game_cons
  (sumT (prodT xLI zLI) (prodT xRI zRI))
  (sumT (prodT xLI zRI) (prodT xRI zLI))
  (map_on_sumT (prodT xLI zLI) (prodT xRI zRI) Game
           ([lli:(prodT xLI zLI)] let (xli,zli)=lli in [xl:=(xLf xli)][zl:=(zLf zli)]
             (GPlus (GPlus (GMult xl z) (GMultAux zl)) (AntiGame (GMult xl zl))))
           ([rri:(prodT xRI zRI)] let (xri,zri)=rri in [xr:=(xRf xri)][zr:=(zRf zri)]
             (GPlus (GPlus (GMult xr z) (GMultAux zr)) (AntiGame (GMult xr zr))))
  )
  (map_on_sumT (prodT xLI zRI) (prodT xRI zLI) Game
           ([lri:(prodT xLI zRI)] let (xli,zri)=lri in [xl:=(xLf xli)][zr:=(zRf zri)]
             (GPlus (GPlus (GMult xl z) (GMultAux zr)) (AntiGame (GMult xl zr))))
           ([rli:(prodT xRI zLI)] let (xri,zli)=rli in [xr:=(xRf xri)][zl:=(zLf zli)]
             (GPlus (GPlus (GMult xr z) (GMultAux zl)) (AntiGame (GMult xr zl))))
  )
)
      end
    }
.

Theorem GlteGMultxZeroZero : (x:Game)(Glte (GMult x Zero) Zero).

Theorem GlteZeroGMultxZero : (x:Game)(Glte Zero (GMult x Zero)).

Theorem GlteGMultZeroxZero : (x:Game)(Glte (GMult Zero x) Zero).
```

```
Theorem GlteZeroGMultZerox : (x:Game)(Glte Zero (GMult Zero x)).

Lemma GMult_OneOne : (Geq (GMult One One) One).

Lemma GlteZeroAntiZero: (Glte Zero (AntiGame Zero)).

Lemma GlteAntiZeroZero: (Glte (AntiGame Zero) Zero).

Lemma GlteGMultGMultAux: (x,y:Game)(Glte (GMult x y) (GMultAux x y)).

Lemma GlteGMultAuxGMult: (x,y:Game)(Glte (GMultAux x y) (GMult x y)).

Theorem GltexGMultOnex : (x:Game)(Glte x (GMult One x)).

Theorem GlteGMultOnexx : (x:Game)(Glte (GMult One x) x).

Theorem GMultCommutative: (x,y:Game)(Geq (GMult x y) (GMult y x)).

Theorem GeqGMultAntixyAntiGMultxy: (x,y:Game) (Geq (GMult (AntiGame x) y) (AntiGame (GMult x y))).

Theorem GeqGMultxAntiyAntiGMultxy: (x,y:Game) (Geq (GMult x (AntiGame y)) (AntiGame (GMult x y))).

Axiom GMultGeqLeft: (x1,x2,y:Game) (Geq x1 x2) -> (Geq (GMult x1 y) (GMult x2 y)).

Hint GPlusComm : Surreal := Resolve GPlusCommutative_one.
Hint GPlusAssoc1 : Surreal := Resolve GPlusAssociative_one.
Hint Gltexx : Surreal := Resolve Glte_reflexive.
Hint GPlusGlteRight : Surreal := Resolve GPlusGlteRight.
Hint GPlusGlteLeft : Surreal := Resolve GPlusGlteLeft.
Hint GPlusGlteBoth : Surreal := Resolve GPlusGlteBoth.
Hint GlteTrans : Surreal := Resolve Glte_transitive.
Hint GlteGMultGMultAux : Surreal := Resolve GlteGMultGMultAux.
Hint AntiGPlus_one : Surreal := Resolve AntiGPlus_one.
Hint xPlusAntixZero_one : Surreal := Resolve xPlusAntixZero_one.
Hint GlteGPlusxZerox: Surreal := Resolve GlteGPlusxZerox.

Theorem GMultDistrib: (x,y,z:Game) (Geq (GMult (GPlus x y) z) (GPlus (GMult x z) (GMult y z))).

End Conway_Games.
```

Bibliography

[BC01] Ana Bove and Venanzio Capretta. Nested general recursion and partiality in type theory. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics: 14^{th} International Conference, TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, pages 121–135. Springer-Verlag, 2001.

[BC02] Ana Bove and Venanzio Capretta. Modelling general recursion in type theory. Under consideration for publication in Math. Struct. in Comp. Science., June 2002.

[Bov02] Ana Bove. *General Recursion in Type Theory.* Thesis for the degree of doctor of philosophy, Department of Computing Science - Chalmers University of Technology and Göteborg University, SE-412 96 Göteborg, Sweden, November 2002.

[Cap00] Venanzio Capretta. Recursive families of inductive types. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 73–89. Springer-Verlag, 2000.

[Con01] John H. Conway. *On Numbers and Games.* A K Peters, Ltd, second edition, 2001. First Edition: 1976.

[dB75] N.G. de Bruijn. Introducing the reals as a totally ordered additive group without using the rationals. Memorandum 1975-13, Eindhoven University of Technology, PO Box 513, Eindhoven, The Netherlands, November 1975.

[Dyb00] Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2), June 2000.

[Knu74] Donald E. Knuth. *Surreal numbers: how two ex-students turned on to pure mathematics and found total happiness.* Addison-Wesley, 1974.

[Lur98] Jacob Lurie. The effective contents of surreal algebra. *Journal of Symbolic Logic*, 63, June 1998.

[PMW99] Christine Paulin-Mohring and Benjamin Werner. Calcul des constructions inductives. Course, January 1999. DEA Sémantique, Preuves et Programmation.

[PMWBH02] Christine Paulin-Mohring, Benjamin Werner, Bruno Barras, and Hugo Herbelin. Calcul des constructions inductives. Course, 2001-2002. DEA Sémantique, Preuves et Programmation.

[Pro] The Coq Development Team LogiCal Project. *The Coq Proof Assistant - Reference Manual.* INRIA, Domaine de Voluceau, Rocquencourt - B.P. 105, F-78153 Le Chesnay Cedex - France.

[Ros01] Frank Rosemeier. On conway-numbers and generalized real numbers. In Ulrich Berger, Horst Oswald, and Peter Schuster, editors, *Reuniting the Antipodes.* Kluwer Academic Publishers, 2001.

[Rosar] Frank Rosemeier. A constructive approach to conway's theory of games and numbers. In *Seminarberichte aus dem Fachbereich Mathematik.* Fernuniversität Hagen, to appear.

[Tøn01] Claus Tøndering. Surreal numbers - an introduction. HTTP, December 2001. Version 1.2.

[Udd80] J.T. Udding. A theory of real numbers and its presentation in automath. Master's thesis, Eindhoven University of Technology, P.O. Box 13, NL-5600MB Eindhoven, The Netherlands, February 1980. under supervision of dr. N.G. de Bruijn.

[Wer97] Benjamin Werner. Sets in types, types in sets. In T. Ito and M. Abadi, editors, *TACS'97*, LNCS, page 1281. Springer-Verlag, 1997.