Università di Firenze, Università di Perugia, INdAM consorziate nel CIAFM

DOTTORATO DI RICERCA
IN MATEMATICA, INFORMATICA, STATISTICA

CURRICULUM IN MATEMATICA
CICLO XXX

Sede amministrativa Università degli Studi di Firenze
Coordinatore Prof. Graziano Gentili

Defining, calculating and reasoning in Higher-Order Logic: Complex and Hypercomplex Analysis and applications

Settore Scientifico Disciplinare MAT/03 - INF/01

Dottorando:
Andrea Gabrielli

Tutore
Prof. Marco Maggesi

Coordinatore
Prof. Graziano Gentili

Anni 2014/2017

Acknowledgements

Looking back at my PhD from the finishing line, I realize that I couldn't face this long journey alone.

First of all, I would like to acknowledge the efforts and input of my supervisor, Prof. Marco Maggesi, who were of great help during my research. I am very grateful to him for the knowledge, the professionalism and the enthusiasm with which he taught me precious lessons that go far beyond the technical ones.

Moreover, my thanks go to my family and my girlfriend Federica, who supported me in these three years, in particular in the last two, which has been full of important changes.

Abstract

The occasional ambiguity of traditional mathematical notations, and the increasing complexity of proofs, has led to the situation that the use of a proof-verification system is desirable if not, in some situations, unavoidable. Moreover, by merging calculation techniques and proofs, theorem provers also allow a deep interconnection of the three basic mathematical activities, that is, *defining*, *calculating* and *reasoning*.

In this thesis we explore such activities from different points of view, dealing with complex and hypercomplex analysis and computability theory using the HOL Light theorem prover.

More precisely, the work is divided into four parts, each independent from the others. In the first part (I) we report on a formal development of quaternions and their algebraic structure, and we discuss automatic and certified procedures to perform calculations on them.

The second part (II) is dedicated to investigate the formalization of possible applications of our framework about quaternions. They are interesting theories on their own and, at the same time, a test for our work. In particular, we formalize basics definitions and theorems about two of the most recent and stimulating theories based on quaternions, that is, *Slice regular quaternionic functions* and *Pythagorean-Hodograph curves*. Slice regular functions extend, in a suitable way, the notion of complex holomorphic function to the quaternionic case whereas, PH-curves are a class of polynomial functions with many theoretical properties and several significant computational advantages in many fields like Computer-Aided Design (CAD), digital motion control, path planning, robotics applications and animation. The main points of the work presented in the first two parts has been published in proceedings of the 8th International Conference on Interactive Theorem Proving in Brasilia 2017 [Gabrielli and Maggesi, 2017].

In part three and four (III - IV) we consider computability as a theory on its own. In particular, we focus on two radically different models of computation (but equally important), namely *Turing Machines* and *quantum computing*. We give the basic definitions and we develop two certified systems to simulate computations in such models. Moreover, by implementing the concepts of Turing machines and quantum circuits in HOL Light, we explore these different approaches formalizing some simple different problems they can solve.

Contents

Preface

Why should mathematics be formalized in a proof-verification system?
In order to give an answer, we can explore the evolution of mathematics from the beginning of its creation process. Barendregt, in his beautiful work [Barendregt, 2013], describes it as follows. At the starting point, the human beings created the subjects of arithmetic and geometry by looking around and abstracting from the nature. Subsequently, many theories above these two, in order to solve basic questions, was created. Every mathematical contribution can be divided into three basic activities: *defining*, *calculating* and *reasoning* (Fig. 1). They started in this order indeed, as example, before doing arithmetic we must have (*define*) numbers (in order to perform operations on them) then we can develop methods to compute (*calculating*) complex expressions and, finally, we can prove (*reasoning*) the correctness of calculations and constructions. However, gradually over time, this schema became increasingly ephemeral and such basic activities became more and more intertwined.

Over the centuries, the focus has shifted periodically on one of the sides of the Barendregt's triangle. In the Egyptian-Chinese-Babylonian tradition emphasis was put on calculation (they could solve linear and quadratic equations correctly but they didn't have a structured notion of proof) whereas in Greek tradition the emphasis was on proofs (they could prove properties that could not be shown by mere computation alone as, for example, the irrationality of $\sqrt{2}$).

Moreover, at the dawn of the modern calculus, Newton and Leibniz used different approaches. The first had a proving style, he wanted to convince others of the correctness of what he did using geometrical proofs to arrive at his conclusions. The second had a style focused on computation, his method worked so well, from a computational point of view, that it did not matter if its foundation (the infinitesimals) wasn't completely clear. Only two hundred years later, by the work of Cauchy and Weierstrass, the computational and proving styles of doing calculus were unified.

Now, it is clear that a proof-verification system (a theorem prover) allows a full integration of defining, calculating and reasoning unifying the previous different approaches. In an interactive theorem prover, the user defines concepts, constructs calculation procedures and provides proofs that show their correctness. During the process, the machine checks that definitions are well-formed and that the proofs and computations are correct. The result of a computation is not only a mere manipulation of symbols (as in case of a Computer Algebra Systems that

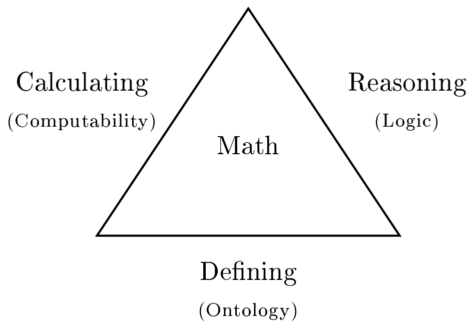


Figure 1: Barendregt's triangle of mathematical activities.

performs calculation from a syntactical point of view), but it is a theorem that states that, given the base assumptions and following the underlying logic, such computation is correct. These considerations partially answer the initial question, but there is something else.

During the history, mathematical proofs have become increasingly complex, so much that, in some cases, their correctness can no longer be controlled by humans. In these cases, a verification system that is able to check the correctness of a proof, following a few simple rules that humans can easily control, is needed if not essential. Two very famous examples are the proofs of the Kepler's Conjecture by Hales [Hales et al., 2017] and of the Four-Color Theorem by Gonthier [Gonthier, 2005] that have been formalized, respectively, in a combination of the HOL Light and Isabelle theorem provers and in the Coq proof assistant.

Furthermore, over the centuries, mathematical theories (from simplest to more advanced and complex ones) have often been able to model part of reality and have applications. The power of mathematics in giving quantitative, or even more qualitative, description of reality has led to the fact that, nowadays, every science needs mathematics. The latter, on the one hand provides them the right language to formulate statements, on the other hand, it guarantees the logical rigour and correctness of results.

Since this pervasive role played by mathematics, it would be convenient that mathematical notations are always clear and unambiguous. Unfortunately, sometimes it is not the case. For example, as observed by Spivak in his book *Calculus on Manifolds* [Spivak, 1965], the symbols used in traditional mathematical notation of differential geometry have ambiguous meanings, which depend on context, and often even change within a given context. Contrary to what one expects, it is very easy to learn to well manipulate symbols, often getting the right answer with fallacious reasoning or without real understanding. This implies that, in many cases, problems related with learning disciplines like physics, as instance, or similar, depends on the difficulties to understand the underlying mathematical language. Indeed, a student must simultaneously learn the mathematical language used and the content that is expressed in that language.

However, also this time, a theorem prover can be a possible instrument to address, at least partially, this problem.

When we express mathematical notions in a formal language, we are forced to define them in an unambiguous and computationally effective way. When the computer interprets our formulas, we quickly realize if they are correct or not. In case that a formula is not clear, it will not be interpretable by the computer. This process necessarily constrains us to improve our understanding.

Moreover, the correctness of all the calculations rules and algorithms used to manipulate mathematical objects is automatically certified by the system. Once formalized as a procedure, a mathematical idea becomes a certified tool that can be used directly to compute (prove) further results.

In our work, we explore *defining*, *calculating* and *reasoning* from different points of view by using the HOL Light theorem prover (authored and maintained by Harrison), that is, a descendant of the original HOL (Higher-Order Logic) system written by Gordon in 1980s.

We specify mathematical objects in the HOL Light formal language (*defining*), we provide procedures to manipulate them and we prove formal theorems (*calculating* and *reasoning*). It is obvious that, since the correctness of every computation is guaranteed by the system, reasoning and calculating are strictly interconnected.

Next, we explore computability as theory on its own. We implement in HOL Light two mathematical models of computation (based on different paradigms) and we provide certified formal methods to perform computations in these models. Such methods don't produce mere manipulations of symbols, they produce theorems that prove the correctness of the calculations done.

More precisely, in this thesis we give a formalization in HOL Light of a basic, but significant, part of *complex and hypercomplex analysis* and some of their applications (parts I and II), and of the abstract models of computation given by *Turing machines* and *quantum computing* (parts III and IV). The main points of the work presented in the first two parts has been published in proceedings of the 8th International Conference on Interactive Theorem Proving in Brasilia 2017 [Gabrielli and Maggesi, 2017].

As regards to hypercomplex analysis, we focus particularly on quaternions. They are a well-known and elegant mathematical structure which lies at the intersection of algebra, analysis and geometry. They have a wide range of theoretical and practical applications from mathematics and physics to CAD, computer animations, robotics, signal processing and avionics. Arguably, a computer formalization of quaternions can be useful, or even essential, for further developments in pure mathematics or for a wide class of applications in formal methods. For these reasons, we develop a basic formal theory about them in our library. Moreover, two recently developed mathematical theories which are based on quaternions are considered, we give the formal definition and some basic theorems about *Slice regular quaternionic functions* and *Pythagorean-Hodograph curves*. Slice regular functions extend, in a suitable way, the notion of complex holomorphic function to the quaternionic case whereas, PH-curves are a class of polynomial functions with many theoretical properties and several significant computational advantages.

As regards to computation models, Turing machines are one of the referring models of our idea of computation (in the classical sense). A physical realization of the abstract Turing machine is a mechanical computing device (with a potentially infinite memory) that obeys the laws of classical physics. However, due to the constant decrease of the dimensions of the calculation devices, we must consider an alternative because, at the microscopic level, classical physics fails and we have to use quantum mechanics. For this reason, but not only, quantum computing is born as an alternative paradigm (to the classical one) based on the principles of quantum mechanics. Furthermore, quantum computing is a connection point between computability and complex numbers because the underlying mathematical theory on which it is based (like quantum mechanics) is complex linear algebra.

So, by implementing the concepts of Turing machines and of quantum circuits in HOL Light, we explore these different approaches to computation formalizing some simple problems they can solve. Moreover, as results of our work, we develop two certified calculation systems about Turing machines and quantum circuits.

As a final observation, even if during our work we did not find any significant error in the underlying informal theories, we realized that, in the light of the formalization developed here, sometimes informal statements or proofs needed considerable rewriting efforts to be rendered in a theorem prover.

For example, in the case of *Slice regular functions*, the implicit isomorphism between the set of complex numbers and appropriate subsets of quaternions can simplify informal statements and proofs but, in our formal context, it has to be always explicit. Moreover, our formal definition (equivalent to the informal one) of *regularity* clarify and simplify the representation in HOL Light of such class of functions.

Similarly, tedious and obvious steps (for example simple properties that can be be proved easily by induction) in informal proofs about *Turing machines* are often omitted but, again, they have to be written explicitly to produce formal theorems.

Part I

FORMALIZING QUATERNION ALGEBRA - THE CORE LIBRARY

Introduction

In the first part of this thesis we present all the background, already available in the HOL Light standard library, that has been developed by several authors before the beginning of our work. In the first chapter (1), after giving a very brief overview of the basic notions (types, terms, theorems etc.), and notations, of the HOL Light theorem prover, we focus on all the formal mathematical structures which are needed to deal with quaternions. More precisely, we recall the HOL Light formalization of:

- univariate and multivariate analysis:
    - cartesian products and vector analysis,
    - limits,
    - series,
    - continuity and differentiability,
- automatic procedures to perform calculations over real numbers.

In the next chapters (2 - 3) we show in details the library about quaternions. The highlights are:

- basic definitions of the algebra of quaternions, that is:
    - definition of quaternions,
    - definition of the arithmetic operations over quaternions,
- conversions to compute with quaternions,
- properties of the arithmetic operations,
- analytic and geometrical results about quaternions, that is:
    - limits, continuity and derivatives of the arithmetic operations,
    - space isometries via quaternions.

So, the goal of this part is twofold. On the one hand, we recall quickly all the HOL Light formal theories which are needed to understand and develop the library about quaternions, on the other hand, we give an exhaustive explanation of the library itself.

Outline of the code

The code about quaternions presented in this part of the thesis is already available in the standard HOL Light distribution in the directory hol-light/Quaternions, at the link

https://github.com/jrh13/hol-light/tree/master/Quaternions

and it is organized as follows.

- **misc.hl** -Miscellanea

- **quaternion.hl** - Basic definitions about quaternions
- **qcalc.hl** - Computing with literal quaternions
- **qnormalizer.hl** - Normalization of quaternionic polynomials
- **qanal.hl** - Elementary quaternionic analysis
- **qderivative.hl** - Derivative of quaternionic functions (in particular arithmetic operations)
- **qisom.hl** - Space isometries via quaternions (embedding of real numbers in the set of quaternions and the geometry of quaternions).

Moreover, the code can be loaded by the command `needs "Qauternions/make.ml";;`.

Chapter 1

HOL Light background

This chapter is dedicated to give a brief introduction of the basic concepts of Higher-Order Logic and the most common pattern usage of the HOL Light theorem prover, as far as it is useful for the material presented in this work. Even if this thesis is not the place to present them extensively, we propose a quick summary to recall the highlights and to fix terminology and notations. Hopefully, this will make able also the readers that are unfamiliar with the HOL Light theorem prover to read and understand the code presented in this work.

1.1 Basic HOL Light notation

- **Basic type system:** the types most frequently used are the following:
  - `` `:num` `` for natural numbers.
  - `` `:real` `` for real numbers. Natural numbers are embedded in the real field by the operator `` `&` ``, for example `` `&1` `` and `` `&0` `` are the unit and zero of reals.
  - `` `:complex` `` for complex numbers. Real numbers are embedded in the complex field by the operator `` `Cx` ``, for example `` `Cx(&1)` `` is the unit of complex numbers.
  - `` `:bool` `` for booleans, `` `T:bool` `` and `` `F:bool` `` represent true and false respectively.
  - `` `:(A)list` `` for lists of elements of type `` `:A` ``.
  - `` `:A->B` `` for functions from the type `` `:A` `` to the type `` `:B` ``. An element `` `f:A->B` `` is such that `` `f a` `` has type `` `:B` ``, for every element `` `a:A` ``.
- **Terms:** Ocaml objects of type :*term*. They are purely symbolic mathematical expressions or logical assertions and are represented between backquotes. Every term has a well-defined type, for example `` `0 = 1` `` is a term and its type is `` `:bool` ``. We recall that HOL Light uses simple polymorphic types.
- **Theorems:** Ocaml objects of type :*thm*. They are formulas that have been proved using the accepted methods of proof starting from a set of assumptions. They are denoted following the standard notation $p_1, \ldots, p_n \vdash p$ that means that the boolean p is provable starting from hypothesis $p_1, \ldots, p_n$. Moreover, a name can be associate to every formal theorem to identify and use it. For example, the writing

  ```
  THEOREM
    |- p
  ```

  means that the boolean `` `p:bool` `` has been proved, with an empty set of assumptions, and the name associated to the theorem is `THEOREM`.

- **Functions:** a function $x \mapsto t$ is denoted, in the lambda calculus notation, as $(\lambda x.\ t)$ that is, formally, the HOL term `` `(\x. t)` ``. For example, the function $f\colon \mathbb{R} \to \mathbb{R}$, such that $f(x) = x + 1$, is formally represented by `` `(\x. x + &1):real->real` ``. Moreover, the evaluation (and so the representation) of functions $f\colon A \times B \to C$ with two arguments is translated in this context by using the currying technique, as an alternative to the more traditional paired approach. In fact, we observe that, given an element $a \in A$, the function $g(x) = f_a(x) = f(a, x)\colon B \to C$ is a function with only one argument $x \in B$. So, the type of the formal counterpart of f is `` `:A->(B->C)` `` that, since the convention to associate on the right, is written simply `` `:A->B->C` ``. The same representation is extended for functions with an arbitrary number of arguments.

- **Logical operators:** `~`, `/\`, `\/`, `==>`, `<=>` represent logical negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), implication ($\rightarrow$) and biconditional ($\leftrightarrow$) respectively.

- **Quantifiers:** the universal and the existential quantifiers $\forall$, $\exists$ are denoted by `` `!` `` and `` `?` `` respectively.

- **Hilbert choice operator:** it is defined by `` `@:(A->bool)->A` `` that, given a predicate `` `P:A->bool` ``, returns the element `` `a:A` `` when `` `P a` `` is true and an unknown element of type `` `:A` `` otherwise. The function `` `\x. (@a. P a)` `` is total but, in practice, we can prove non-trivial properties for its image only in the case that `` `P` `` is satisfied by at least one element.

- **Definitions:** the simplest instruction that allows us to define new constant is the command `new_definition`. Typing

  ```
  let thm = new_definition
     `const_name = form`
  ```

  we obtain essentially two results:

  1. the system expands with a new constant named `` `const_name` ``,
  2. the system produces a new theorem

     ```
     |- const_name = form
     ```

     that defines, by the formulas `` `form` ``, the new constant.

  In order to avoid possible inconsistencies, the command `` `new_definition` `` has some limitations on what it can defines. However, much more general forms of definition, using recursion and sophisticated pattern-matching, can be given by using the commands `` `new_recursive_definition` `` and `` `define` `` in a similar way.

1.2 HOL Light internal representation of natural numbers

Many formal objects in this work will be developed coherently with the HOL Light internal representation of natural numbers. For this reason, we quickly show the latter in details.
HOL Light represents numerals with a binary encoding with the following four constants:

1. `` `_0:num` `` that encodes the number $0 \in \mathbb{N}$,
2. `` `BIT0:num->num` `` that for every natural number n returns the double $2n$,
3. `` `BIT1:num->num` `` that for every natural number n returns the successor of the double $2n + 1$,
4. `` `NUMERAL:num->num` `` that is the identity function, it is defined only for technical reasons.

Therefore, every numeral is represented by a term of the form `‘NUMERAL (f (_0))‘`, where `‘f:num->num‘` represents a function that is an arbitrary composition of `‘BIT0‘` and `‘BIT1‘`. Since `‘NUMERAL‘` is the identity function, we can avoid to consider it in our informal reasoning thinking a numerals simply as an iteration of `‘BIT0‘` and `‘BIT1‘`.

Intuitively, having in mind the binary representation of a natural number as a string of zeros and ones, the functions `‘BIT0‘` and `‘BIT1‘` replace every occurrence, of zero and one respectively, in such a string. For example, the binary representation of 5 is 101 so, replacing every zero with `‘BIT0‘` and every one with `‘BIT1‘`, we obtain the HOL Light internal representation of the term `‘5‘`, i.e.,

```
‘NUMERAL (BIT1 (BIT0 (BIT1 _0))‘
```

that, as said before, can be rewritten as `‘BIT1 (BIT0 (BIT1 _0)‘`.

The binary encoding makes very easy to recognize easily if a number is even or odd. In fact, `‘n:num‘` is even if and only if it is of the form `‘BIT0 k‘` and, conversely, it is odd if and only if it is of the form `‘BIT1 k‘`, for some `‘k:num‘` (obviously, `‘_0‘` is even because `‘_0 = BIT0 _0‘`). This is a feature that comes handy in several situations as, for example, in our formal representation of vectors presented in chapter 9 (section 9.3).

1.3 Vector analysis in HOL Light

Given a set A, that is the universe of some type `‘:A‘`, it is convenient to have a corresponding type for the Cartesian product A^n. If n is fixed we can easily repeat the product type `‘(#)‘` n-times to construct the type `‘:A#A#...#A‘` of the cartesian product A^n.

However, if n is large this becomes unwieldy and it does not extend to the case where n is a variable, because HOL's type theory doesn't support dependent types. More precisely, a type may only be parametrized by another type and not by a term. Therefore, HOL provides an indexed Cartesian product constructor written as an infix `‘(^)‘`, as in `‘:A^N‘`. The second argument of `‘(^)‘` is also a type and it is used to specify the size of the universe set of that type that represents the parameter n.

The type `‘:A^N‘` is designed for finite Cartesian products, when the (universe of) type `‘:N‘` is finite, the type `‘:A^N‘` is isomorphic to the function space `‘:N->A‘`, otherwise it just collapses to being equivalent to `‘:A‘`. The size of `‘:N‘` is indicated with `‘dimindex(:N)‘`. For example, the type `‘:1‘`, that has only one element, has size equal to one so it holds that `‘dimindex(:1) = 1‘`. Moreover, if a type `‘:X‘` is infinite then, by the definition of `‘dimindex‘`, its size collapses to one, that is, it holds that `‘dimindex(:X) = 1‘`.

The operator `‘($):A^N->num->A‘` is the indexing operator so `‘x$i:A‘` ($x_i \in A$) denotes the i-th component of the vector `‘x:A^N‘` ($x \in A^n$).

Since `‘:A^N‘` is isomorphic to the type `‘:N->A‘` (when `‘:N‘` is finite), HOL Light provides a general constructor `‘lambda‘` such that, given a function `‘f:N->A‘`, returns the vector `‘lambda i. f‘` with the property that `‘(lambda i. f)$i = f i‘` for every natural number `‘i:num‘` between one and the size of the type `‘:N‘`.

However, a handy notation is desirable for denoting vectors by enumerating their elements. So, HOL Light provides another general notion `‘vector:(A)list->A^N‘` that, for every indexing type `‘:N‘` and list `‘l:(A)list‘`, returns the element `‘vector l:A^N‘`.

Notice that the HOL type system cannot infer that `‘length l = dimindex(:N)‘` from the element `‘vector l:A^N‘`. For example, given a list `‘l:(A)list‘` with length three, we can declare and create any element `‘vector l:A^N‘` even if `‘dimindex(:N)‘` is different from three. However, in the following we will never use this counterintuitive construction.

In principle, every concrete indexing finite type can be constructed, by hand, by iterating the type constructor `finite_sum` starting from `‘:1‘`. In fact, given two types `‘:A‘` and `‘:B‘`, it holds that `‘dimindex(:(A,B)finite_sum) = dimindex(:A) + dimindex(:B)‘`. If `‘:A‘` and `‘:B‘` are finite types it is obvious whereas, if `‘:A‘` (`‘:B‘`) is infinite, the finite sum collapses, by definition, to `‘:(1,B)finite_sum‘` (`‘:(A,1)finite_sum‘`) and the previous property still holds since the properties of `‘dimindex‘`. For example, the type `‘:3‘`, with three elements, can be defined as `` `:((1,1)finite_sum),1)finite_sum` `` and it can be proved that its size is three, that is, the formal theorem `|- dimindex(:3) = 3`. HOL Light standard library provides type constants, already defined, only for finite types up to size four, they are `` `:1` ``, `` `:2` ``, `` `:3` `` and `` `:4` ``. An example of a theorem about their size is the following, in the case of the type `` `:2` ``.

```
DIMINDEX_2
  |- dimindex(:2) = 2
```

Moreover, an automatic procedure to compute directly components of a concrete vector `` `vector l:A^N` ``, that is, terms of the form `` `vector l $ i` ``, is not defined. HOL Light provides *ad hoc* theorems to rewrite components of a concrete vector only for up to 4-dimensional real vector spaces. For example, in case of a 2-dimensional vector `` `vector[w;x]:real^2` ``, we have the following theorem.

```
VECTOR_2
  |- vector [w; x]$1 = w /\
     vector [w; x]$2 = x
```

Similar theorems are given for `` `:real^1` ``, `` `:real^3` `` and `` `:real^4` ``. However, dealing with concrete real vector spaces with any other dimension greater than four, we have to do, each time, some extra work. Firstly, we have to define the relative finite indexing type (and prove a theorem about its size). Secondly, we have to prove a theorem, in the same way of `VECTOR_2`, to be able to rewrite components of a concrete vector expressed, by enumerating its elements, in the form `` `vector l` ``. In chapter 9 (section 9.3) we refine this approach providing a concrete representation of finite types that allows us both to define them with an uniform mechanism and to prove, automatically, theorems about their size.

However, with the formal instruments provided by HOL Light, we can profitably work with general real vector spaces $\mathbb{R}^n$ using the related type `` `:real^N` ``. In fact, the system allows us to define general structures and operators as, for example, the addition of vectors `` `(+):real^N->real^N->real^N` ``, that don't depend directly on a particular dimension. In the HOL Light standard library we can found an exhaustive theory of real vector spaces.

Finally, note that, by construction, the types `` `:real` `` and `` `:real^1` `` are isomorphic (that is, there exists a bijection) but they are not the same. For this reason, the functions `` `lift:real->real^1 `` and `` `drop:real^1->real` ``, that connect them, are defined. The following theorem shows the interaction between `` `lift` `` and `` `drop` ``.

```
LIFT_DROP
  |- (!x. lift (drop x) = x) /\ (!x. drop (lift x) = x)
```

The use of `` `:real^1` `` or `` `:real` `` depends on the context, the usual HOL conventions and the local usefulness (for example a curve is represented in HOL Light as a function of type `` `:real^1->real^N` `` instead of `` `:real->real^N` ``).

1.4 Multivariate analysis in HOL Light

1.4.1 Limits

Classically, when we work with limits, we have in mind many different definitions of limit. For example, given a function $f\colon A \to B$ or a sequence $a_n \in A$, the symbols

$$\lim_{x\to x_0} f(x), \qquad \lim_{x\to x_0^+} f(x), \qquad \lim_{x\to +\infty} f(x), \qquad \lim_{n\to +\infty} a_n \tag{1.4.1}$$

are definend differently and have different meanings depending on the topology structure of A and B. Moreover, for example, the notation

$$\lim_{x\to x_0} f(x) = l$$

can be ambiguous or even misleading. The latter can have two possible interpretations:

- the limit exists and is equal to l, considering lim as a ternary relation where the arguments are, in this case, the limit considered ($x \to x_0$), the given function ($f\colon A \to B$) and the value of the limit ($l \in B$),
- knowing that the limit exists its value is l, considering lim as an operator, i.e. a functional (higher-order function), that takes two arguments, the given function ($f\colon A \to B$) and the limit considered ($x \to x_0$), and returns the value of the limit $l \in B$.

Mathematicians often consider lim in the second way, i.e. as an operator, implicitly implying that it is well defined only in case that the function considered admits the limit required.

However, the natural way to formalize the concept of limit, in a HOL formal theory, is to consider it as a predicate, so to codify the existence and the value of the limit at the same time. Thus, the highlights of the HOL Light standard implementation of the notion of limit are essentially two.

1. Limit is introduced as a ternary predicate and not as an operator. Contrarily, the existence of the limit should be explicitly expressed separately and, since functions are total in HOL, a value should be assigned also to functions that don't admit the limit considered.
2. HOL Light has a general notion that let us represent a variety of limits in a compositional way. This is achieved using the mathematical notion of net (whose discussion is far from the aim of this work for reasons of time and space) that is implemented by a polymorphic dedicated type `` `:(A)net` ``. The elements of such type specify, each time, what is the limit considered over the type `` `:A` ``. So, in order to represent a specific limit we have only to istantiate the generic element `` `net:(A)net` `` in the general definition of limit.

Obviously, since derivatives and series are defined as limits, all these considerations can be repeated for them, so they are formalized in HOL light following the same style of limits.

Concretely, the ternary predicates `` `((f --> l) net)` `` and `` `((f ---> l) net)` `` are used to say that the function f admits limit and its value is l, with respect to the limit defined by `` `net:(A)net` ``, in the vector- or real-valued case respectively. In this setting, limits of sequences are represented considering functions `` `a:num->real` `` or `` `a:num->real^N` ``.
The most common limits are

$$\lim_{n\to+\infty} a_n = l \tag{1.4.2}$$

for sequences $\{a_n\}_{n\in\mathbb{N}}$ and

$$\lim_{x\to x_0} f(x) = l \tag{1.4.3}$$

for functions $f : \mathbb{R}^m \to \mathbb{R}^n$ so we briefly report their representations.

Limits of sequences. The limit (1.4.2) is formalized using the net

`` `sequentially:(num)net` ``

by the term `` `(a --> l) sequentially` ``. The following theorem allows us to use its usual definition, where `` `dist:real^N#real^N->real` `` represents the standard distance over `` `:real^N` ``.

```
LIM_SEQUENTIALLY
  |- !s l. (s --> l) sequentially <=>
           (!e. &0 < e ==> (?N. !n. N <= n ==> dist (s n,l) < e))
```

Limits of functions. The limit (1.4.3) is formalized using the operator

`` at:real^M->(real^M)net` ``

that, given the limit point $x_0 \in \mathbb{R}^m$, returns the net `` `at x:(real^M)net` `` (representing the limit $x \to x_0$). So, the resulting formal representation is the term `` `(f --> l) (at x)` ``. We have the usual definition of (1.4.3) in the following theorem.

```
LIM_AT
  |- !f l a. (f --> l) (at a) <=>
             (!e. &0 < e
                  ==> (?d. &0 < d /\
                           (!x. &0 < dist (x,a) /\ dist (x,a) < d
                                ==> dist (f x,l) < e)))
```

In case of functions `‘f:real->real^N‘`, we have to use `‘atreal:real->(real)net‘` to represent the limit at a certain point `‘x:real‘`. In HOL Light, this distinction is necessary because `‘:real‘` is different from all types `‘:real^N‘`. As above, the corresponding theorem is the following.

```
LIM_ATREAL
  |- !f l a. (f --> l) (atreal a) <=>
             (!e. &0 < e
                  ==> (?d. &0 < d /\
                           (!x. &0 < abs (x - a) /\ abs (x - a) < d
                                ==> dist (f x,l) < e))).
```

Moreover, by using the Hilbert choice operator `‘@‘`, two functions are defined to allow to use the limit as an operator in fact, given a function and a net, they return the appropriate limit, if it exists, in the vector- and real-valued case respectively.

```
lim
  |- !f net. lim net f = (@l. (f --> l) net)

reallim
  |- !f net. reallim net f = (@l. (f ---> l) net)
```

The operator `‘within‘`. In many cases, we want to consider limits with some restrictions. For example, when we consider subsequencies, or left- and right-hand limits of real-valued functions, we want to evaluate the limit within a specific set. For this purpose, HOL provides the infix operator

`‘within :(A)net->(A->bool)->(A)net‘`

that, given a net `‘n:(A)net‘` and a subset `‘s:A->bool‘` of the universe of the type `‘:A‘`, returns the net `‘n within s :(A)net‘`. Intuitively, the latter is the net `‘n:(A)net‘` where we consider only elements of `‘s:A->bool‘`. In this way, we can formalize, for example, the limit

$$\lim_{n\to+\infty} a_{2n} = \lim_{n\in\{n\in\mathbb{N}\ \mid\ \exists m\ s.t.\ n=2m\}\to+\infty} a_n$$

of the even terms of a sequence. It can be done by using the net

`‘sequentially within EVEN :(num)net‘`

where `‘EVEN:num->bool‘` represents the set of even numbers.

Eventually. When we talk about limits, we usually say that a predicate holds from a certain point on approaching the limit point. In HOL Light we can do this with the general notion

`‘eventually :(A->bool)->(A)net->bool‘.`

For every predicate `‘P:A->bool‘` and every net `‘net:(A)net‘`, the latter returns the boolean

`‘eventually P net :bool‘`

which is true if and only if `‘P‘` holds from a certain point on *approaching* the limit point of the limit defined by `‘net‘`.

Very common cases are that of the nets `‘sequentially‘` and `‘at x‘` where `‘eventually‘` works as we expect as shown by the following theorems.

```
EVENTUALLY_SEQUENTIALLY
  |- !p. eventually p sequentially <=> (?N. !n. N <= n ==> p n)

EVENTUALLY_AT
  |- !a p. eventually p (at a) <=>
          (?d. &0 < d /\ (!x. &0 < dist (x,a) /\ dist (x,a) < d ==> p x))

EVENTUALLY_ATREAL
  |- !a p. eventually p (atreal a) <=>
          (?d. &0 < d /\ (!x. &0 < abs (x - a) /\ abs (x - a) < d ==> p x))
```

Trivial limit. Not every net `‘n:(A)net‘` defines a limit that makes sense. For example, given a finite set `‘k:num->bool‘` we can construct the net

`‘sequentially within k :(num)net‘`

but it is *trivial* because we are considering the limit to infinity within a finite set. Therefore, HOL Light provides a predicate

`‘trivial_limit :(A)net->bool‘`

to characterize the set of trivial limits. Clearly, if the predicate `‘trivial_limit net‘` holds for a net `‘net:(A)net‘`, then it implies that:

- `‘(f --> l) net‘` holds for every function `‘f:real^M->real^N‘` and limit `‘l:real^N‘`,
- `‘eventually P net‘` holds for every predicate `‘P:A->bool‘`.

For the most common nets `‘sequentially‘`, `‘at x‘` and `‘atreal x‘`, it is proved, in the following theorems, that they are not trivial limits.

```
TRIVIAL_LIMIT_SEQUENTIALLY
  |- ~trivial_limit sequentially

TRIVIAL_LIMIT_AT
  |- !a. ~trivial_limit (at a)

TRIVIAL_LIMIT_ATREAL
  |- !a. ~trivial_limit (atreal a).
```

1.4.2 Series

In order to talk about real-valued or vector-valued series, depending on the type `‘:real‘` or `‘:real^N‘` that we consider, we have to use different operators. Using the HOL Light functions

`‘sum:(A->bool)->(A->real)->real‘`

`‘vsum:(A->bool)->(A->real^N)->real^N‘`

we can represent sums of the form $\sum_{n\in S} a_n$ with the terms `‘sum k a‘` and `‘vsum k a‘` for the real or vector case respectively. If $S \subseteq \mathbb{N}$ is an infinite set, we can consider the notion of convergence of the series above. As always, it is implemented in HOL Light by two predicates, involving `‘sum‘` and `‘vsum‘` respectively, that make sense only in case that `‘:A‘` is the type of natural numbers `‘:num‘`.

Given a vector-valued sequence $f\colon \mathbb{N} \to \mathbb{R}^n$, a vector $l \in \mathbb{R}^n$ and a subset of natural numbers $S \subseteq \mathbb{N}$, the predicate `‘(f sums l) s‘` is used to say, formally, that the series

$$\sum_{n\in S} f(n)$$

is convergent and its value is l. In case of a real-valued sequence $f\colon \mathbb{N} \to \mathbb{R}$, we must use the predicate `(f real_sums l) s` to express the same concept.

Definitions are given in the usual way in fact, the following theorems state that, as one can expect, a function $f : \mathbb{N} \to \mathbb{R}^n$ ($f : \mathbb{N} \to \mathbb{R}$) has sum $l \in \mathbb{R}^n$ ($l \in \mathbb{R}$), over $S \subseteq \mathbb{N}$, if and only if $\sum\limits_{n \in S \cap \{0..n\}} f(n)$ tends to l for $n \to +\infty$.

```
sums
  |- !s f l. (f sums l) s <=>
             ((\n. vsum (s INTER (0..n)) f) --> l) sequentially

real_sums
  |- !s f l. (f real_sums l) s <=>
             ((\n. sum (s INTER (0..n)) f) ---> l) sequentially
```

More concisely, we can use `real_summable` or `summable` defined by

```
real_summable
  |- !f s. real_summable s f <=> (?l. (f real_sums l) s)

summable
  |- !f s. summable s f <=> (?l. (f sums l) s)
```

to express only the convergence of a series without considering the right value of the sum. As for limits, two functions are defined to formalize the infinite sum, if it is defined, for vector- and real-valued series respectively

```
infsum
  |- !f s. infsum s f = (@l. (f sums l) s)

real_infsum
  |- !f s. real_infsum s f = (@l. (f real_sums l) s)
```

1.4.3 Continuity and Differentiability

Given a function $f : \mathbb{R}^n \to \mathbb{R}^m$, we can express in the HOL Light formalism the continuity of f, in a point of its domain $x_0 \in D_f \subseteq \mathbb{R}^n$, by using the following predicate.

`f continuous at x0`

Furthermore, the (Fréchet) derivative of f in x_0, if it exists, is the linear function from $\mathbb{R}^n$ to $\mathbb{R}^m$ that "best" approximates the variation of f in a neighborhood of x_0, i.e,

$$f(x) - f(x_0) \approx \mathrm{D}f_{x_0}(x - x_0)$$

and it is denoted by $\mathrm{D}f_{x_0}$ or $\frac{\mathrm{d}}{\mathrm{d}x} f(x)|_{x_0}$. In HOL Light, the ternary predicate

`(f has_derivative f') (at x0)`

is used to assert that f is differentiable at x_0 and $f' = \mathrm{D}f_{x_0}$. We can also generalize these notions considering a generic net `net:(real^N)net` instead of `at x0`. For functions $f : \mathbb{R} \to \mathbb{R}$ the above predicates are replaced by

`f real_continuous atreal x0`

`(f has_real_derivative f') (atreal x0)`

where now the real number f' is the real derivative of f. In these cases, the linear function that best approximates the variation of f in a neighborhood of x_0 is the multiplication by f'.

We can use `differentiable` or `real_differentiable` to express the differentiability of a vector- or real-valued function respectively. They are defined by the following theorems.

```
differentiable
  |- !f net. f differentiable net <=>
             (?f'. (f has_derivative f') net)

real_differentiable
  |- !f net. f real_differentiable net <=>
             (?f'. (f has_real_derivative f') net)
```

Given a curve $f : \mathbb{R} \to \mathbb{R}^n$, it can be expressed by its components $f_i : \mathbb{R} \to \mathbb{R}$ with $i \in \{0...n\}$ as $f(x) = (f_1(x), ..., f_n(x)) \in \mathbb{R}^n$. The tangent vector to $f(x)$ in $x_0 \in \mathbb{R}$, if it exists, is

$$f'(x_0) = (f_1'(x_0), ..., f_n'(x_0)) \in \mathbb{R}^n$$

and it is represented in HOL Light by following predicate.

```
`(f has_vector_derivative f') (at x0)`
```

Since complex numbers are implemented in HOL Light as element of $\mathbb{R}^2$, we can consider complex holomorphic functions as functions $f : \mathbb{R}^2 \to \mathbb{R}^2$ such that their (Fréchet) derivative is $\mathbb{C}$-linear. From this point of view, such functions can be described using `` `has_derivative` `` but, considering the relevance of this class of functions, they have a dedicated predicate. This is

```
`(f has_complex_derivative f') (at z0)`
```

that means that the function $f : \mathbb{C} \to \mathbb{C}$ is holomorphic in z_0 and its complex derivative is the complex number f'. Once again, we can use the following functions to represent the different kinds of derivatives as operators instead of relations.

```
frechet_derivative
  |- !f net. frechet_derivative f net =
             (@f'. (f has_derivative f') net)

real_derivative
  |- !f x. real_derivative f x =
           (@f'. (f has_real_derivative f') (atreal x))

vector_derivative
  |- !f net. vector_derivative f net =
             (@f'. (f has_vector_derivative f') net)

complex_derivative
  |- !f x. complex_derivative f x =
           (@f'. (f has_complex_derivative f') (at x))
```

1.5 Conversions in HOL Light

1.5.1 Basic conversions and conversionals

In a proof we often want to show that one term is equal to another using a systematic process of transformation, perhaps passing through several intermediate stages. In order to do this easily, efficiently and mechanically as much as possible, HOL Light provides a systematic framework for conversions.

A conversion is simply an inference rule of type :*term* $\to$ *thm* that, given a term t, always returns (assuming it doesn't fail) an equational theorem of the form $\vdash t = t'$ that is, it proves that the term t that was given is equal to some other term, possibly the same as the original. For example, the theorem `|- 2 + 2 = 4` can be produced automatically, from the term `` `2 + 2` ``, by a conversion.

For each type of numbers (for example `‘:num‘` or `‘:real‘` or `‘:complex‘`) there is a whole family of conversions performing ‘evaluation’ of expressions involving arithmetic operations, one for each arithmetic operator.

Another kind of very useful conversions are those which perform rewrites that originate in the Paulson’s paper [Paulson, 1983]. The basic is `‘REWR_CONV‘` that takes an equational theorem $\vdash s = t$ and produces a conversion that, when applied to a term s' which s can be matched to, returns the corresponding theorem $\vdash s' = t'$ or fails otherwise. Since the biconditional in HOL is equality of booleans, such a conversion can also be used for theorems of the form $\vdash p \Leftrightarrow q$.

Equally important are functions, called conversionals, that take one or more conversions and return a new conversion. The most important are:

- `THENC` takes two conversions and returns the new conversion resulting from the consequential application of them; it is used as an infix operator,
- `ORELSEC` takes two conversions, tries to apply the first and, if it fails, tries to apply the second,
- `TRY_CONV` tries a conversion but happily return a reflexive theorem if it fails,
- `RAND_CONV` applies a conversion to the rand of a combination (e.g. x in $f(x)$),
- `RATOR_CONV` applies a conversion to the rator of a combination (e.g. f in $f(x)$),
- `LAND_CONV` applies a conversion to the left-hand argument of a binary operator,
- `BINDER_CONV` applies a conversion to the body of a binder (for example a quantifier),
- `BINOP_CONV` applies a conversion to both arguments of a binary operator,
- `DEPTH_CONV` applies a conversion at every level of a term,
- `TOP_DEPTH_CONV` applies a conversion as long as possible all over the term.

Combining `REWR_CONV` with the functions above, HOL provides more sophisticated conversions to perform rewrites. The most general is

```
GEN_REWRITE_CONV conv1 [th1;...;thn]
```

that takes a conversional and a list of equational theorems. It is basically as

```
conv1(REWR_CONV th1 ORELSEC ... ORELSEC thn).
```

A significant generalization is `REWRITE_CONV` that do rewrites as long as possible all over the term. Essentially it has the same basic strategy as

```
GEN_REWRITE_CONV TOP_DEPTH_CONV
```

except that a suite of standard rewrite rules are always included in the rewrites in addition to the theorems list supplied to it.

1.5.2 Arithmetic conversion for the type `‘:real‘`

As said before, for every type of numbers, we have a specific conversion for every relational and arithmetic operator. For example, to compute terms of the form `‘x + y‘`, were `‘x‘` and `‘y‘` are concrete rational real numbers (e.g. `‘&1 / &2‘`), we can use the conversion `REAL_RAT_ADD_CONV` as in the following example.

```
REAL_RAT_ADD_CONV ‘&2 / &5 + &1‘;;
val it = |- &2 / &5 + &1 = &7 / &5
```

We underline the fact that REAL_RAT_ADD_CONV works only for this particular class of terms. For example, it fails if the real numbers ‘x‘, ‘y‘ are variables (or expressions of real numbers) or if the operator that has to be computed is not the addition. We show an example.

```
REAL_RAT_ADD_CONV ‘&2 / &5 * &1‘;;
Exception: Failure "dest_binop".
```

For each other arithmetic and relational operator over real numbers (subtraction, negation, multiplication, power etc.), the corresponding conversion is defined. Such conversions are:

- REAL_RAT_LE_CONV and REAL_RAT_LT_CONV to compute terms of the form ‘x < y‘ ($x < y$) and ‘x <= y‘ ($x \leq y$) respectively,
- REAL_RAT_EQ_CONV to compute terms of the form ‘x = y‘ ($x = y$),
- REAL_RAT_NEG_CONV to compute terms of the form ‘-- x‘ ($-x$),
- REAL_RAT_ABS_CONV to compute terms of the form ‘abs x‘ ($|x|$),
- REAL_RAT_INV_CONV to compute terms of the form ‘ inv x‘ ($\frac{1}{x}$),
- REAL_RAT_SUB_CONV to compute terms of the form ‘ x - y‘ ($x - y$),
- REAL_RAT_MUL_CONV to compute terms of the form ‘x * y‘ (xy),
- REAL_RAT_DIV_CONV to compute terms of the form ‘x / y‘ ($\frac{x}{y}$),
- REAL_RAT_POW_CONV to compute terms of the form ‘x pow y‘ (x^y).

As before, the real numbers ‘x‘ and ‘y‘ must be concrete rational real numbers, otherwise the conversions fail. As examples, we show some simple cases.

```
REAL_RAT_LE_CONV ‘&2 / &5 < &1‘;;
val it = |- &2 / &5 < &1 <=> T

REAL_RAT_MUL_CONV ‘&2 / &5 + &1‘;;
val it = |- &2 / &5 * &1 / &3 = &2 / &15
```

Moreover, the conversion REAL_RAT_RED_CONV is able to apply the appropriate conversion depending on the form of the term that must be computed. Finally, the conversion

```
REAL_RAT_REDUCE_CONV = DEPTH_CONV REAL_RAT_RED_CONV
```

can be used to perform calculations, repeatedly, on every level of a term (that is, on every subterm) as shown by the following example.

```
REAL_RAT_REDUCE_CONV ‘&1 / &4 + &5 * (&7 / &3 pow 2)‘;;
val it = |- &1 / &4 + &5 * &7 / &3 pow 2 = &149 / &36
```

Chapter 2

Quaternion algebra in HOL Light

Quaternions are an elegant mathematical structure which lies at the intersection of algebra, analysis and geometry. They have a wide range of theoretical and practical applications from mathematics and physics to Computer-Aided Design, computer animations, robotics, signal processing and avionics. Arguably, a computer formalization of quaternions can be useful, or even essential, for further developments in pure mathematics or for a wide class of applications in formal methods.

For example, quaternions are very useful to describe spatial rotations. A formalization of basics about them has been developed by Affeldt and Cohen [Affeldt and Cohen, 2017] in order to deal with the mathematics of rigid body transformations in the Coq proof assistant.

Our task is to contribute to the development of the HOL Light library that formalizes the basics of the theory of quaternions (started by M. Maggesi at the University of Florence) that we found at the beginning of our work. Such implementation is presented in details in this chapter and, in the second part of this thesis, it will be needed as a formal support theory when we will deal with our development of two of the most recent and interesting theories based on quaternions, that is, *Slice regular functions* [Gentili et al., 2013] and *Pythagorean-Hodograph curves* [Farouki, 2009].

In the first section 2.1, we give an informal summary about the highlights of the theory of quaternions. More precisely, we recall, firstly, the definition of quaternions and, secondly, the definitions and the main properties of the arithmetic operations.

Next, in section 2.2 we show how the set of quaternions can be coded in HOL Light by the dedicated type `` `:quat` `` and its associated basic operations like constructors and destructors.

Finally, in the last section 2.3 we show how the type `` `:quat` `` is equipped with the non-commutative field structure by defining formally the algebraic operations.

2.1 Quaternion algebra

We begin the chapter with an informal introduction to the theory of quaternions. More precisely, we recall notations and arithmetic definitions. In this section, we limit ourselves of the basic algebraic structure but, further theoretical developments will be exposed in sections 3.4 and 3.5, where we discuss analytic and geometric results. A modern account of the theory of quaternions can be found in [Conway and Smith, 2003].

Quaternions are "four-dimensional numbers" of the form

$$q = q_0 + q_1\mathbf{i} + q_2\mathbf{j} + q_3\mathbf{k} \tag{2.1.1}$$

where $q_i \in \mathbb{R}$ for $i = 0 \ldots 3$ are called the real components of the quaternion q, and the *basis elements* 1, $\mathbf{i}$, $\mathbf{j}$, $\mathbf{k}$ satisfy the relations

$$\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1 \tag{2.1.2}$$

where 1 is the usual real unit (its product with $\mathbf{i}$, $\mathbf{j}$, $\mathbf{k}$ leaves them unchanged). Furthermore, the real number $q_0 \in \mathbb{R}$ is called the *real part* of q and is denoted by $\mathrm{Re}(q)$, whereas the 3-dimensional vector $(q_1, q_2, q_3) \in \mathbb{R}^3$ is called the *imaginary part* of q and is denoted by $\mathrm{Im}(q)$. The set of quaternions is denoted by $\mathbb{H}$ and by the above construction it's clear that $\mathbb{H} = \mathbb{R} \oplus \mathbb{R}^3 \simeq \mathbb{R}^4$, thus $\mathbb{R}$ and $\mathbb{R}^3$ are proper subsets of $\mathbb{H}$.

As in the complex case ("two-dimensional numbers" over $\mathbb{R}$) the main operations defined over $\mathbb{H}$ are addition, product (Hamilton product), conjugation, norm, negation and inverse. Let be $q = q_0 + q_1\mathbf{i} + q_2\mathbf{j} + q_3\mathbf{k}$ and $p = p_0 + p_1\mathbf{i} + p_2\mathbf{j} + p_3\mathbf{k}$ two elements of $\mathbb{H}$, we have:

- **addition:**
$$q + p = (q_0 + p_0) + (q_1 + p_1)\mathbf{i} + (q_2 + p_2)\mathbf{j} + (q_3 + p_3)\mathbf{k} \tag{2.1.3}$$
defined componetwise with neutral element the usual zero 0 of reals,

- **product:**
$$\begin{aligned} qp = {} & (q_0p_0 - q_1p_1 - q_2p_2 - q_3p_3) + \\ & (q_0p_1 + q_1p_0 + q_2p_3 - q_3p_2)\mathbf{i} + \\ & (q_0p_2 - q_1p_3 + q_2p_0 + q_3p_1)\mathbf{j} + \\ & (q_0p_3 + q_1p_2 - q_2p_1 + q_3p_0)\mathbf{k} \end{aligned} \tag{2.1.4}$$
determined by the the products of the *basis elements* and the distributive law with nuetral element the usual unit 1 of reals,

- **conjugation:**
$$\bar{q} = q_0 - q_1\mathbf{i} - q_2\mathbf{j} - q_3\mathbf{k} \tag{2.1.5}$$

- **norm:**
$$\|q\| = \sqrt{q\bar{q}} = \sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2} \tag{2.1.6}$$

- **negation:**
$$-q = -q_0 - q_1\mathbf{i} - q_2\mathbf{j} - q_3\mathbf{k} \tag{2.1.7}$$
since $q + (-q) = (q_0 - q_0) + (q_1 - q_1)\mathbf{i} + (q_2 - q_2)\mathbf{j} + (q_3 - q_3)\mathbf{k} = 0$,

- **inverse:** if $q \neq 0$ then
$$q^{-1} = \frac{\bar{q}}{|q|^2} \tag{2.1.8}$$
since $qq^{-1} = q\frac{\bar{q}}{\|q\|^2} = \frac{q\bar{q}}{\|q\|^2} = \frac{\|q\|^2}{\|q\|^2} = 1$.

From these definitions, and the properties of real addition and product, we can easily check the main properties of the quaternionic operations:

- addition is associative and commutative,
- product is associative but not commutative, in general $qp \neq pq$ as in the case of
$$\mathbf{ij} = \mathbf{k} \neq -\mathbf{k} = \mathbf{ji}$$
(commutativity holds only in special cases as, for example, when at least one of q and p is an element of $\mathbb{R}$ or when they both belong to a substet of $\mathbb{H}$ that is isomorphic to $\mathbb{C}$),
- product is left- and right-distributive over addition,
- conjugation is an involution because $\overline{(\bar{q})} = q$ and, moreover, it holds that $q = \bar{q}$ if and only if q is an element of $\mathbb{R}$,
- $\|q\| \in \mathbb{R}_{\geq 0}$ and it holds that $\|q\| = 0$ if and only if $q = 0$,
- the elements $-q$ and q^{-1} are unique and such that $q = -q$ is equivalent to $q = 0$ and, for every $q \neq 0$, $q = q^{-1}$ is equivalent to $q = 1$.

2.2 Formalization of quaternions

Complex numbers, together with a consistent part of the theory of holomorphic functions, are implemented in HOL Light by Harrison [Harrison, 2007] following a specific style. More precisely, they are formally considered as elements of $\mathbb{R}^2$ and the dedicated type `` `:complex` `` is defined as an alternative name of the type `` `:real^2` ``. The implementation of quaternions closely follows this style. Therefore, $\mathbb{H}$ is identified with $\mathbb{R}^4$ and the type `` `:real^4` `` is used to represent it formally. As in the complex case, this choice is the most convenient because in `` `:real^4` `` (as in `` `:real^2` ``) it is already defined the appropriate topology and normed vector space structure. Again, the type `` `:quat` `` is defined as an alternative name of the type `` `:real^4` ``.

2.2.1 The type `` `:real^4` ``

As we discussed in chapter 1 (section 1.3), the finite type `` `:4` `` is already defined and it is proved, in the next theorem, that its size is four.

```
DIMINDEX_4
 |- dimindex (:4) = 4
```

Moreover, HOL Light standard library provides *ad hoc* theorems that prove handy rewrites, about components, equality and universal and existential quantification, for 4-dimensional real vectors written in the explicit enumerated form `` `vector[w;x;y;z]:real^4` ``. Let be `` `P:real^4->bool` `` a generic predicate about 4-dimensional real vectors, such theorems are the following.

- VECTOR_4
```
   |- vector [w; x; y; z]$1 = w /\
      vector [w; x; y; z]$2 = x /\
      vector [w; x; y; z]$3 = y /\
      vector [w; x; y; z]$4 = z
```
- VECTOR_EQ_4
```
   |- !u v. u = v <=> u$1 = v$1 /\ u$2 = v$2 /\ u$3 = v$3 /\ u$4 = v$4
```
- FORALL_VECTOR_4
```
   |- (!v. P v) <=> (!w x y z. P (vector [w; x; y; z]))
```
- EXISTS_VECTOR_4
```
   |- (?v. P v) <=> (?w x y z. P (vector [w; x; y; z]))
```

The usual structure of normed vector space is already defined over the type `` `:real^4` ``. Therefore, we don't have to re-define operations like addition, negation, subtraction, multiplication by a scalar and norm. As usual, in this formal setting, negation and subtraction are different operations (negation is unary whereas subtraction is binary) and they need different symbols to be denoted. The HOL Light standard formalism use `` `(--) :real^4->real^4` `` for negation and `` `(-) :real^4->real^4->real^4` `` for subtraction.

2.2.2 The type `` `:quat` ``

Even if `` `:quat` `` is only an alternative name for the type `` `:real^4` `` that is, quaternions are formally 4-real vectors, a set of constants for constructing and destructing quaternions is defined in order to setup a suitable *abstraction barrier*. Following the informal theory, every time it's possible, the same constant names already defined for complex numbers are used. For example `` `Re` `` for the real part of a quaternion, `` `cnj` `` for the conjugate operator, `` `ii` `` for the imaginary unit **i** and so on. The same holds for the operation symbols, for example `` `(+):complex->complex` `` and `` `(+):quat->quat` `` are formally different operators but the same constant symbol is overloaded to represent both of them. The whole set of such constants is summarized in table 2.1 when we report, for every constant, the name, the type and a brief informal mathematical description.

Table 2.1: Basic notations for the `:quat` datatype

Constant name	Type	Description
`Hx`	`:real->quat`	Embedding $\mathbb{R} \to \mathbb{H}$
`ii`, `jj`, `kk`	`:quat`	Imaginary units $\mathbf{i}, \mathbf{j}, \mathbf{k}$
`quat`	`:real#real#real#real->quat`	Generic constructor
`Hv`	`:real^3->quat`	Embedding $\mathbb{R}^3 \to \mathbb{H}$
`Re`	`:quat->real`	Real component
`Im1`, `Im2`, `Im3`	`:quat->real`	Imaginary components
`HIm`	`:quat->real^3`	Imaginary part
`cnj`	`:quat->quat`	Conjugation
`real`	`:quat->bool`	Whether a quaternion is real

The general constructor `quat` is defined using the constructor `vector` as follows.

```
let quat = new_definition
  `quat(x,y,z,w) = vector[x;y;z;w]:quat`;;
```

The real components `Re`, `Im1`, `Im2`, `Im3` (of type `:quat->real`) are defined, as one expects, by using the usual indexing operator `($):real^4->num->real` for vectors.

```
let QUAT_RE_DEF  = new_definition  `Re(x:quat) = x$1`;;
let QUAT_IM1_DEF = new_definition `Im1(x:quat) = x$2`;;
let QUAT_IM2_DEF = new_definition `Im2(x:quat) = x$3`;;
let QUAT_IM3_DEF = new_definition `Im3(x:quat) = x$4`;;
```

With these definitions, and the theorems of the previous paragraph about `:real^4`, the following useful results are proved:

- QUAT_COMPONENTS
  ```
  |- Re (quat (x,y,z,w)) = x /\
     Im1 (quat (x,y,z,w)) = y /\
     Im2 (quat (x,y,z,w)) = z /\
     Im3 (quat (x,y,z,w)) = w
  ```
  describes the obvious interaction of `Re`, `Im1`, `Im2` and `Im3` with `quat`,

- QUAT_EQ
  ```
  |- !p q. p = q <=> Re p = Re q /\ Im1 p = Im1 q /\
                     Im2 p = Im2 q /\ Im3 p = Im3 q
  ```
  states that two quaternions are equal if and only if they are equal componentwise,

- FORALL_QUAT
  ```
  |- (!q. P q) <=> (!x y z w. P (quat (x,y,z,w)))
  ```
  states that a predicate `P:quat->bool` is true for every quaternion if and only if it holds for every 4-tupla of real numbers,

- EXISTS_QUAT
  ```
  |- (?q. P q) <=> (?x y z w. P (quat (x,y,z,w)))
  ```
  states that a predicate `P:quat->bool` is true for some quaternion if and only if it holds for some 4-tupla of real numbers.

In this context, the *basis - elements* **i**, **j**, **k** are respectively the vectors

$$\begin{pmatrix}0\\1\\0\\0\end{pmatrix},\begin{pmatrix}0\\0\\1\\0\end{pmatrix},\begin{pmatrix}0\\0\\0\\1\end{pmatrix}$$

so the related formal definition are given.

```
let quat_ii = new_definition `ii = quat(&0,&1,&0,&0)`;;
let quat_jj = new_definition `jj = quat(&0,&0,&1,&0)`;;
let quat_kk = new_definition `kk = quat(&0,&0,&0,&1)`;;
```

The standard representation of a quaternion (2.1.1) is formalized in the following theorems.

```
QUAT_EXPAND
  |- !q. q = Hx (Re q) + ii * Hx (Im1 q) + jj * Hx (Im2 q) + kk * Hx (Im3 q)

QUAT_TRAD
  |- !x y z w. quat (x,y,z,w) = Hx x + ii * Hx y + jj * Hx z + kk * Hx w
```

Note that, with respect to the traditional representation of a quaternion (2.1.1), the real components $q_i \in \mathbb{R}$ with $i = 0 \dots 3$, represented formally by `Re q`, `Im1 q`, `Im2 q`, `Im3 q` (everyone of type `:real`), must be injected in `:quat` with the function `Hx:real->quat` to make the type agree in the multiplication with the basis vectors.

In fact, from an informal point of view, a real number can be considered also a quaternion, i.e. $\mathbb{R}$ can be considered as a proper subset of $\mathbb{H}$. Unfortunately, HOL Light is a polymorphic system with simple types and doesn't have subtypes. It means that an element `x:real` can't be of type `:quat` at the same time. Formally, a quaternion that is a real number is represented in this formalism by `quat(x,&0,&0,&0)` but, a special notation to clearly identify and use them is more convenient. For these reasons, as it happens in the case of complex numbers, real numbers are injected inside quaternions using the dedicated function `Hx:real->quat` defined as follows.

```
let HX_DEF = new_definition
 `Hx(a) = quat(a,&0,&0,&0)`;;
```

The following theorems show that the function `Hx` is an injection and allow us to rewrite the components of a quaternion that is a "real number" respectively.

```
HX_INJ
  |- !x y. (Hx(x) = Hx(y)) <=> (x = y)

HX_COMPONENTS
  |- (!a. Re (Hx a) = a) /\
     (!a. Im1 (Hx a) = &0) /\
     (!a. Im2 (Hx a) = &0) /\
     (!a. Im3 (Hx a) = &0)
```

Thus, the zero and the unit of quaternions can be represented respectively by `Hx(&0)` and `Hx(&1)` instead of `quat(&0,&0,&0,&0)` and `quat(&1,&0,&0,&0)`.

Obviously, a quaternion is a real number if and only its imaginary projections are zero, so the set of "real numbers" is identified by the predicate `real:quat->bool`[1] defined as follows.

```
let quat_real = new_definition
 `real(q:quat) <=> Im1 q = &0 /\ Im2 q = &0 /\ Im3 q = &0`;;
```

[1]Every set S of elements of type `:A` is represented in HOL Light as a predicate of type `:A->bool` that is true for every elements $s \in S$ and is false otherwise.

The following theorem shows that `real:quat->bool` is, in fact, the image of the universe `:real` through the function `Hx`.

```
QUAT_REAL_EXISTS
 |- !q. real q <=> (?x. q = Hx x)
```

The embedding $\mathbb{R}^3 \to \mathbb{H}$ and the projection $\mathbb{H} \to \mathbb{R}^3$ are formalized in the same way with the constants `Hv:real^3->real^4` and `HIm:real^4->real^3` defined as usual.

```
let HV = new_definition
  `Hv(x:real^3) = quat(&0,x$1,x$2,x$3)`;;

let HIM_DEF = new_definition
  `HIm(q:quat) : real^3 = vector[q$2;q$3;q$4]`;;
```

The rules to rewrite their components are explicit in the following theorems.

```
HIM_COMPONENT
  |- (!q. HIm q$1 = Im1 q) /\
     (!q. HIm q$2 = Im2 q) /\
     (!q. HIm q$3 = Im3 q)

HV_COMPONENTS
  |- (!x. Re (Hv x) = &0) /\
     (!x. Im1 (Hv x) = x$1) /\
     (!x. Im2 (Hv x) = x$2) /\
     (!x. Im3 (Hv x) = x$3)
```

The obvious interaction between `Hx`, `HIm` and `Hv` is described below where `vec 0` represents, for every type `:real^N`, the zero vector $(0, \dots, 0) \in \mathbb{R}^n$.

```
HV_HIM
 |- !q. Hv (HIm q) = quat (&0,Im1 q,Im2 q,Im3 q)

HIM_HV
 |- !x. HIm (Hv x) = x

HIM_HX
 |- !a. HIm (Hx a) = vec 0
```

2.3 Formalizing quaternionic operations

As mentioned in the previous subsection, over every type `:real^N`, the normed vector space structure is already defined. Thus, addition, negation, subtraction and norm

```
`(+) :real^N->real^N->real^N`
`(--) :real^N->real^N`
`(-) :real^N->real^N->real^N`
`norm :real^N->real`
```

are defined. Where they are istantiated over `:quat` (i.e. `:real^4`), they act compatibly with definitions (2.1.3), (2.1.7) and (2.1.6). This implies that such operations don't need to be redefined. However, to be consistent with the formalism introduced, they are expressed using the projections `Re`, `Im1`, `Im2` and `Im3` through the following theorems.

```
quat_add
  |- p + q = quat (Re p + Re q,Im1 p + Im1 q, Im2 p + Im2 q,Im3 p + Im3 q)
```

```
quat_neg
  |- --q = quat (--Re q,--Im1 q,--Im2 q,--Im3 q)

quat_norm
  |- norm q = sqrt (Re q pow 2 + Im1 q pow 2 + Im2 q pow 2 + Im3 q pow 2)
```

Moreover, the linear behaviour of the projections over addition, negation and subtraction is formally proved. Here we report, as example, the formal theorem in the case of addition, the others are similar.

```
QUAT_ADD_COMPONENTS
  |- (!x y. Re (x + y) = Re x + Re y) /\
     (!x y. Im1 (x + y) = Im1 x + Im1 y) /\
     (!x y. Im2 (x + y) = Im2 x + Im2 y) /\
     (!x y. Im3 (x + y) = Im3 x + Im3 y)
```

Finally, the fact that $q = 0$ if and only if $|q|^2 = 0$, for all $q \in \mathbb{H}$, is formalized in the theorem

```
QUAT_EQ_0
  |- !q. q = Hx (&0) <=> Re q pow 2 + Im1 q pow 2 +
                         Im2 q pow 2 + Im3 q pow 2 = &0
```

where the squared norm is written explicitly.

As regards to the product, the situation is very different because it is not a concept definable over all types `:real^N`. Since it is specific for the type `:quat`, no more general definition can be istantiated and it must be defined from scratch following definition (2.1.4). The resulting formalization is the following.

```
let quat_mul = new_definition
`p * q =
 quat(Re p * Re q - Im1 p * Im1 q - Im2 p * Im2 q - Im3 p * Im3 q,
      Re p * Im1 q + Im1 p *  Re q + Im2 p * Im3 q - Im3 p * Im2 q,
      Re p * Im2 q - Im1 p * Im3 q + Im2 p *  Re q + Im3 p * Im1 q,
      Re p * Im3 q + Im1 p * Im2 q - Im2 p * Im1 q + Im3 p *  Re q)`;;
```

From the latter, the power `pow :quat->num->quat` can be inductively defined .

```
let quat_pow = define
  `(q pow 0 = Hx(&1)) /\
   (!n. q pow (SUC n) = q * q pow n)`;;
```

It's the same for conjugate and inverse, the respective formal definitions, compatibly with the informal ones (2.1.5) and (2.1.8), are the following.

```
let quat_cnj = new_definition
  `cnj(q:quat) = quat(Re(q),--Im1(q),--Im2(q),--Im3(q))`;;

let quat_inv = new_definition
  `inv q =
   quat(Re q  / (Re q pow 2 + Im1 q pow 2 + Im2 q pow 2 + Im3 q pow 2),
       --(Im1 q) / (Re q pow 2 + Im1 q pow 2 + Im2 q pow 2 + Im3 q pow 2),
       --(Im2 q) / (Re q pow 2 + Im1 q pow 2 + Im2 q pow 2 + Im3 q pow 2),
       --(Im3 q) / (Re q pow 2 + Im1 q pow 2 + Im2 q pow 2 + Im3 q pow 2))`;;
```

Note that, the definition of the inverse is equivalent to the informal one (2.1.8) where the norm and the conjugate are made explicit. A theorem that follows closely the standard notation is proved.

```
QUAT_INV_CNJ
  |- !q. inv q = inv (Hx (norm q pow 2)) * cnj q
```

At this point, only the basic algebraic definitions are implemented. In the next chapter we will show how to automate calculations about quaternions by using specific conversions. Thanks to these, many of the main properties of arithmetic operations (e.g. associativity, commutativity etc.), as well as more advanced analytic and geometric results, are proved.

Chapter 3

Computing with quaternions

In this chapter we present a set of theorems and conversions to perform calculations with quaternions and quaternionic functions from an algebraic and analytic point of view. In the spirit of a deep interconnection of *defining*, *calculating* and *reasoning*, that is one of the guidelines of this work, a simple certified and automatic calculation procedure that allows to prove a set of basic arithmetic theorems is presented. Then, such theorems will be used to develop more advanced conversions and to prove more complex theorems.

More precisely, in section 3.1 we present a very elementary tactic used to prove the main properties of the algebraic operations, whereas sections 3.2 and 3.3 are dedicated to more advanced conversions: one for arithmetic calculations with rational quaternions and one for normalization of quaternionic polynomials.

After, these tools are used to prove results about quaternions from an analytic and geometric point of view. More precisely, in section 3.4 we show how limits and derivatives of quaternionic functions are formally computed. The last section 3.5 is dedicated to an investigation of the geometric nature of quaternions and a brief formal overview on the link between quaternions and spatial orthogonal transformations is given.

3.1 Proving basic quaternionic algebraic identities

3.1.1 A simple decision procedure for quaternionic algebraic identities

After the formalization of quaternions and their basic algebraic operations, a procedure to automate, as much as possible, easy and tedious calculations is needed. A very crude tactic to prove simple algebraic equivalences over `:quat`, and its related rule, are defined.

```
let SIMPLE_QUAT_ARITH_TAC =
  REWRITE_TAC[QUAT_EQ; QUAT_COMPONENTS; HX_DEF;
              quat_add; quat_neg; quat_sub; quat_mul;
              quat_inv] THEN
  CONV_TAC REAL_FIELD;;

let SIMPLE_QUAT_ARITH tm = prove(tm,SIMPLE_QUAT_ARITH_TAC);;
```

However, more advanced conversions to automate calculations will be presented in sections 3.2 and 3.3 when we deal with polynomial normalization.

The basic idea behind SIMPLE_QUAT_ARITH is very simple. Everything is rewritten componentwise in order to reduce the goal over `:quat` into four subgoals over `:real`. Then, such subgoals can be proved by the decision procedure REAL_FIELD that can automatically prove statements over `:real` which are true due to the field structure of $\mathbb{R}$. Now, we show an example for more details.

Example. If we have the goal over `` `:quat` ``

```
val it : goalstack = 1 subgoal (1 total)

`!q. ~(q = Hx (&0)) ==> q * (inv q + H(&1)) = Hx (&1) + q`
```

we can do the first rewrite with

```
REWRITE_TAC[QUAT_EQ_0]
```

to obtain

```
val it : goalstack = 1 subgoal (1 total)

`!q. ~(Re q pow 2 + Im1 q pow 2 + Im2 q pow 2 + Im3 q pow 2 = &0)
     ==> q * (inv q + Hx (&1)) = Hx (&1) + q`
```

and then we can use `SIMPLE_QUATH_ARITH_TAC` to automate the computation. The first part of the tactic

```
REWRITE_TAC[QUAT_EQ; QUAT_COMPONENTS; HX_DEF;
            quat_add; quat_neg; quat_sub; quat_mul;
            quat_inv]
```

reduces the goal in the conjunction of four subgoals over `` `:real` `` as follows

```
val it : goalstack = 1 subgoal (1 total)

`!q. ~(Re q pow 2 + Im1 q pow 2 + Im2 q pow 2 + Im3 q pow 2 = &0)
     ==> Re q * (Re q / (Re q pow 2 + Im1 q pow 2 +
                        Im2 q pow 2 + Im3 q pow 2) + &1) -
         Im1 q * (--Im1 q / (Re q pow 2 + Im1 q pow 2 +
                            Im2 q pow 2 + Im3 q pow 2) + &0) -
         Im2 q * (--Im2 q / (Re q pow 2 + Im1 q pow 2 +
                            Im2 q pow 2 + Im3 q pow 2) + &0) -
         Im3 q * (--Im3 q / (Re q pow 2 + Im1 q pow 2 +
                            Im2 q pow 2 + Im3 q pow 2) + &0) =
         &1 + Re q /\
         Re q * (--Im1 q / (Re q pow 2 + Im1 q pow 2 +
                           Im2 q pow 2 + Im3 q pow 2) + &0) +
         Im1 q * (Re q / (Re q pow 2 + Im1 q pow 2 +
                         Im2 q pow 2 + Im3 q pow 2) + &1) +
         Im2 q * (--Im3 q / (Re q pow 2 + Im1 q pow 2 +
                            Im2 q pow 2 + Im3 q pow 2) + &0) -
         Im3 q * (--Im2 q / (Re q pow 2 + Im1 q pow 2 +
                            Im2 q pow 2 + Im3 q pow 2) + &0) =
         &0 + Im1 q /\
         Re q * (--Im2 q / (Re q pow 2 + Im1 q pow 2 +
                           Im2 q pow 2 + Im3 q pow 2) + &0) -
         Im1 q * (--Im3 q / (Re q pow 2 + Im1 q pow 2 +
                            Im2 q pow 2 + Im3 q pow 2) + &0) +
         Im2 q * (Re q / (Re q pow 2 + Im1 q pow 2 +
                         Im2 q pow 2 + Im3 q pow 2) + &1) +
         Im3 q * (--Im1 q / (Re q pow 2 + Im1 q pow 2 +
                            Im2 q pow 2 + Im3 q pow 2) + &0) =
         &0 + Im2 q /\
         Re q * (--Im3 q / (Re q pow 2 + Im1 q pow 2 +
                           Im2 q pow 2 + Im3 q pow 2) + &0) +
         Im1 q * (--Im2 q / (Re q pow 2 + Im1 q pow 2 +
                             Im2 q pow 2 + Im3 q pow 2) + &0) -
         Im2 q * (--Im1 q / (Re q pow 2 + Im1 q pow 2 +
                             Im2 q pow 2 + Im3 q pow 2) + &0) +
         Im3 q * (Re q / (Re q pow 2 + Im1 q pow 2 +
                          Im2 q pow 2 + Im3 q pow 2) + &1) =
         &0 + Im3 q`
```

and the second part (CONV_TAC REAL_FIELD) can prove them automatically because they are true due to the field structure of $\mathbb{R}$. Much more concisely, we can use the code

```
let THEOREM = prove
 (`!q. ~(q = Hx (&0)) ==> q * (inv q + Hx (&1)) = Hx (&1) + q`,
  REWRITE_TAC[QUAT_EQ_0] THEN SIMPLE_QUAT_ARITH_TAC);;
```

to produce the same theorem in one shot.

```
THEOREM
  |- !q. ~(q = Hx (&0)) ==> q * (inv q + Hx (&1)) = Hx (&1) + q.
```

3.1.2 Proving basic arithmetical properties

This approach, although very crude, can prove directly nearly 60 basic identities, as, for example, many of the basic properties of algebraic operations like associativity or commutativity. Moreover, it is also occasionally useful to prove *ad hoc* identities in the middle of more complex proofs. In this way, a small library with the essential algebraic results, which are needed for building more complex procedures and theorems, is quickly developed.

The following list contains the formal theorems about the main properties of arithmetic operations, it can be used as basic guidelines for the interested users that start to use the HOL Light library about quaternions.

```
QUAT_ADD_ASSOC |- !x y z. x + y + z = (x + y) + z

QUAT_ADD_SYM |- !x y. x + y = y + x

QUAT_ADD_LID |- !x. Hx (&0) + x = x

QUAT_ADD_RID |- !x. x + Hx (&0) = x.

QUAT_MUL_ASSOC |- !x y z. x * y * z = (x * y) * z

QUAT_MUL_LID |- !x. Hx (&1) * x = x

QUAT_MUL_RID |- !x. x * Hx (&1) = x

QUAT_MUL_HX_SYM |- !q a. q * Hx a = Hx a * q

QUAT_ADD_LDISTRIB |- !x y z. x * (y + z) = x * y + x * z

QUAT_ADD_RDISTRIB |- !x y z. (y + z) * x = y * x + z * x

QUAT_SUB_LDISTRIB |- !x y z. x * (y - z) = x * y - x * z

QUAT_SUB_RDISTRIB |- !x y z. (x - y) * z = x * z - y * z.

QUAT_NORM_MUL |- !p q. norm(p * q) = norm(p) * norm(q)
```

```
QUAT_NEG_0 |- --Hx (&0) = Hx (&0))

QUAT_NEG_NEG |- !x. -- --x = x)

QUAT_SUB_REFL |- !x. x - x = Hx (&0)

QUAT_SUB_RZERO |- !x. x - Hx (&0) = x

QUAT_SUB_LZERO |- !x. Hx (&0) - x = --x

QUAT_CNJ_CNJ |- !q. cnj (cnj q) = q

QUAT_CNJ_INJ |- !p q. cnj p = cnj q <=> p = q

QUAT_REAL_CNJ |- !q. real q <=> cnj q = q

CNJ_HX |- !x. cnj (Hx x) = Hx x.

QUAT_INV_1 |- inv (Hx (&1)) = Hx (&1)

QUAT_INV_INV |- !q. inv (inv q) = q

QUAT_MUL_LINV |- !q. ~(q = Hx (&0)) ==> inv q * q = Hx (&1)

QUAT_MUL_RINV |- !q. ~(q = Hx (&0)) ==> q * inv q = Hx (&1).

QUAT_POW_ADD |- !x m n. x pow (m + n) = x pow m * x pow n

QUAT_POW_POW |- !x m n. x pow m pow n = x pow (m * n)

QUAT_POW_UNITS_2 |- ii pow 2 = --Hx (&1) /\
                    jj pow 2 = --Hx (&1) /\
                    kk pow 2 = --Hx (&1)

QUAT_NORM_POW |- !q n. norm(q pow n) = norm(q) pow n

QUAT_REAL_ADD |- !p q. real p /\ real q ==> real (p + q)

QUAT_REAL_NEG |- !q. real q ==> real (--q)

QUAT_REAL_SUB |- !p q. real p /\ real q ==> real (p - q)

QUAT_REAL_MUL |- !p q. real p /\ real q ==> real (p * q)

QUAT_REAL_POW |- !q n. real q ==> real (q pow n)

QUAT_REAL_INV |- !q. real q ==> real (inv q)

Hx_ADD |- !x y. Hx(x + y) = Hx(x) + Hx(y)

HX_NEG |- !x. Hx(--x) = --(Hx(x))

HX_SUB |- !x y. Hx(x - y) = Hx(x) - Hx(y)

HX_MUL |- !x y. Hx(x * y) = Hx(x) * Hx(y)
```

```
HX_POW |- !x n. Hx(x pow n) = Hx(x) pow n
```

```
HX_INV |- !x. Hx(inv x) = inv(Hx x).
```

Moreover, in addition to these simple, but significant, theorems, we proved a property for the inverse function that looks “strange”. In fact, contrarily to what one expects by the informal theory, it holds that the inverse of zero is zero itself as shown by the following HOL theorem.

```
QUAT_INV_0 |- inv (Hx (&0)) = Hx (&0)
```

This happens because the constant `‘inv:quat->quat‘` (as `‘inv:complex->complex‘`) inherits its properties from the constant `‘inv:real->real‘`. It is known that, in the theory of real numbers, the inverse of zero is not defined. However, since HOL Light admits only total functions, also the element `‘inv (&0)‘` must be formally defined. In the HOL Light construction of real numbers, the latter is proved to be equal to `‘&0‘`

```
REAL_INV_0 |- inv (&0) = &0
```

and thus, as we have seen before, the same is proved for quaternions .

Now, even if `SIMPLE_QUAT_ARITH_TAC` enables us to prove basic quaternionic algebraic identities, more advanced automatic procedures are needed to perform calculations involving rational quaternions and polynomial normalization. We present them in details in the next two sections.

3.2 Arithmetic conversion for rational quaternions

In the HOL Light standard library, a conversion to compute with concrete literal quaternions with rational components, i.e. terms of the form `‘quat(x,y,z,w)‘` where `‘x‘`,`‘y‘`,`‘z‘`,`‘w‘` are concrete rational real numbers (e.g. `‘&2 / &3‘`), is available. It is constructed by composing different conversions, one for every arithmetic operator. The name convention used is `RAT_QUAT_op_CONV` (with `op = ADD, MUL, NEG,` etc.) to represent each of them.

Working with concrete literal quaternions, conversions for real numbers can be used on each component of them. In order to do that, the conversion has to get down firstly, in the rand of the operator `‘quat‘` and secondly, in both sides of the constructor for pairs `‘(,)‘` recursively (`‘(x,y,z,w)‘` is an iteration of the binary operator `‘(,)‘`). Hence, the following conversional, that applies a conversion to each component of a quaternion of the form `‘quat(x,y,z,w‘)`, is defined as follows.

```
let QUAT_COMPONENTS_CONV conv =
  RAND_CONV (LAND_CONV conv THENC
             RAND_CONV (LAND_CONV conv THENC
                        RAND_CONV (BINOP_CONV conv)));;
```

Subsequently, conversions to prove automatically statements about equality and all arithmetic operators are given. In case of equality the theorem

```
qth_eq
  |- !x1 x2 x3 x4 y1 y2 y3 y4.
         quat (x1,x2,x3,x4) = quat (y1,y2,y3,y4) <=>
         x1 = y1 /\ x2 = y2 /\ x3 = y3 /\ x4 = y4
```

is used to convert an equality between quaternions into four equalities between real numbers. At this point, the conversion `REAL_RAT_EQ_CONV` is applied to each of them. Since the output of rewriting `‘qth_eq‘` is a conjunction, the process can be optimized by rewriting at every step one of the following theorems.

```
th1 |- F /\ p <=> F

th2 |- T /\ p <=> p
```

In this way, when the first equality is examined there are two possibilities:

- `‘x1 = y1‘` is false so the conversion returns false without considering the next components,
- `‘x1 = y1‘` is true so the process is repeated for the second equality and so on.

Formally, the above reasoning is implemented with `RAT_QUAT_EQ_CONV` that is defined essentially as follows.

```
let RAT_QUAT_EQ_CONV : conv =
  let c1,c2 = REWR_CONV th1, REWR_CONV th2 in
  REWR_CONV qth_eq THENC LAND_CONV REAL_RAT_EQ_CONV THENC
  (c1 ORELSEC
   (c2 THENC LAND_CONV REAL_RAT_EQ_CONV THENC
    (c1 ORELSEC
     (c2 THENC LAND_CONV REAL_RAT_EQ_CONV THENC
      (c1 ORELSEC
       (c2 THENC
        REAL_RAT_EQ_CONV))))));;
```

Obviously, the latter works only with terms in the right form, that is, it fails if the term is not an equality between quaternions or if the quaternions aren't in explicit form. Here there are some examples.

```
RAT_QUAT_EQ_CONV ‘quat(&1, &2, &3, &34) = quat(&1, &2, &3, &34)‘;;
val it = |- quat (&1,&2,&3,&34) = quat (&1,&2,&3,&34) <=> T

RAT_QUAT_EQ_CONV ‘quat(&1, &2, &3, &34) = quat(&1, &2, &3, &4)‘;;
val it = |- quat (&1,&2,&3,&34) = quat (&1,&2,&3,&4) <=> F

RAT_QUAT_EQ_CONV ‘quat(&1, &2, &3, &34) + quat(&1, &2, &3, &4)‘;;
Exception: Failure "term_pmatch".

RAT_QUAT_EQ_CONV ‘(x:quat) = (y:quat)‘;;
Exception: Failure "dest_comb: not a combination".
```

As in the real case, the corresponding conversions for all the arithmetic operators are defined. Now, the basic strategy is to rewrite the output of an operation in explicit form and then use, when it's possible, the conversion `‘QUAT_COMPONENTS_CONV REAL_RAT_op_CONV‘` (for `op = ADD, MUL, NEG`, etc) that applies the right conversion for real numbers componentwise. For example, in the case of addition, the related conversion is defined as follows.

```
let RAT_QUAT_ADD_CONV : conv =
  let qth_add = prove
   (‘!x1 x2 x3 x4 y1 y2 y3 y4.
       quat(x1,x2,x3,x4) + quat(y1,y2,y3,y4) =
       quat(x1+y1,x2+y2,x3+y3,x4+y4)‘,
    REWRITE_TAC[quat_add; QUAT_EQ; QUAT_COMPONENTS]) in
  REWR_CONV qth_add THENC
  QUAT_COMPONENTS_CONV REAL_RAT_ADD_CONV;;
```

The same construction works also for negation and substraction so, `RAT_QUAT_NEG_CONV` and `RAT_QUAT_SUB_CONV` can be defined in the same way. Unfortunately, it fails with multiplication and inverse. This happens because the real components of a product or an inverse doesn't involve only real product or real inverse. In these cases, the conversion REAL_RAT_REDUCE_CONV, instead of REAL_RAT_MUL_CONV or REAL_RAT_INV_CONV, has to be used since it can computes every expression involving real rational numbers. With this tweak, the conversion RAT_QUAT_MUL_CONV is defined as follows.

```
let RAT_QUAT_MUL_CONV : conv =
  let qth_mul = prove
   (`!x1 x2 x3 x4 y1 y2 y3 y4.
       quat(x1,x2,x3,x4) * quat(y1,y2,y3,y4) =
       quat(x1 * y1 - x2 * y2 - x3 * y3 - x4 * y4,
            x1 * y2 + x2 * y1 + x3 * y4 - x4 * y3,
            x1 * y3 - x2 * y4 + x3 * y1 + x4 * y2,
            x1 * y4 + x2 * y3 - x3 * y2 + x4 * y1)`,
    REWRITE_TAC[quat_mul; QUAT_EQ; QUAT_COMPONENTS]) in
  REWR_CONV qth_mul THENC
  QUAT_COMPONENTS_CONV REAL_RAT_REDUCE_CONV;;
```

Similarly, also the conversion RAT_QUAT_INV_CONV is defined.

In case of powers, things are bit more complicated. If the exponent `n:num` of the power `quat(x,y,z,w) pow n` is large, the multiplication rule of quaternions can't be rewritten n-times. However, we can use the well known technique *exponentiation by squaring* in order to simplify the computation. The strategy on which it is based is to exploit the binary representation of natural numbers. The fundamental observation is that the computation of a power, of the form q^n, can involve two cases:

- $n = 2m$ is even then, setting $p = q^m$, it holds that $q^n = q^{2m} = p * p$,
- $n = 2m + 1$ is odd then, setting $p = q^m$ and $y = p * p$, it holds that $q^n = q * y$.

Therefore, following this style, the related conversion for powers `RAT_QUAT_POW_CONV` is defined.

Finally, the following global conversion is defined by using the procedure REWRITES_CONV which implements a net rewriting strategy.

```
let RAT_QUAT_RED_CONV =
  let gconv_net = itlist (uncurry net_of_conv)
    [`quat x = quat y`,RAT_QUAT_EQ_CONV;
     `quat x + quat y`,RAT_QUAT_ADD_CONV;
     `quat x - quat y`,RAT_QUAT_SUB_CONV;
     `quat x * quat y`,RAT_QUAT_MUL_CONV;
     `inv (quat x)`,RAT_QUAT_INV_CONV;
     `quat x pow n`,RAT_QUAT_POW_CONV;
     `cnj(quat x)`,RAT_QUAT_CNJ_CONV;
     `-- (quat x)`,RAT_QUAT_NEG_CONV]
    (basic_net()) in
  REWRITES_CONV gconv_net;;
```

The latter has to be applied all over the term, so it is defined the final conversion

```
let RAT_QUAT_REDUCE_CONV = DEPTH_CONV RAT_QUAT_RED_CONV;;
```

that is able to compute every expression involving rational quaternions.
We show some examples.

```
RAT_QUAT_REDUCE_CONV
`cnj (quat (&3 / &5,&1,&4 / &3,&0)) * inv (quat (&7 / &3,&0,&0,&0)) pow 3 -
 quat (-- &1,&0,&0,&0)`;;
val it =
|- cnj (quat (&3 / &5,&1,&4 / &3,&0)) * inv (quat (&7 / &3,&0,&0,&0)) pow 3 -
   quat (-- &1,&0,&0,&0) =
   quat (&1796 / &1715,-- &27 / &343,-- &36 / &343,&0)

RAT_QUAT_REDUCE_CONV
`(quat (&3 / &2, &2,&6 / &7,&1) + quat (&0,-- &2,-- &6 / &7,-- &1)) *
 cnj (quat (&2,&0,&0,&0) pow 3)`;;
val it =
|- (quat (&3 / &2,&2,&6 / &7,&1) + quat (&0,-- &2,-- &6 / &7,-- &1)) *
   cnj (quat (&2,&0,&0,&0) pow 3) =
   quat (&12,&0,&0,&0)
```

Moreover, in order to work with quaternions written in traditional form $x+y\mathbf{i}+z\mathbf{j}+w\mathbf{k}$, that is formally `Hx x + ii * Hx y + jj * Hx z + kk * Hx w`, also the following conversions are defined.

```
let QUAT_TRAD_CONV : conv =
 REWR_CONV QUAT_TRAD THENC
   GEN_REWRITE_CONV DEPTH_CONV [QUAT_MUL_RID; QUAT_MUL_RZERO;
                                QUAT_ADD_LID; QUAT_ADD_RID]) tm);;

let RATIONAL_QUAT_CONV : conv =
  GEN_REWRITE_CONV DEPTH_CONV [HX_DEF; quat_ii;
                               quat_jj; quat_kk] THENC
  RAT_QUAT_REDUCE_CONV THENC QUAT_TRAD_CONV;;
```

The first rewrites quaternions of the form `quat(x,y,z,w)` in the traditional form and simplifies the components that are zero. The second, given an arithmetic expression with quaternions in traditional form, acts essentially in three steps:

1. converts the input term in another involving only the constructor `quat`, rewriting the definition of `Hx` and of the *basis-elements* `ii`, `jj`, `kk`,

2. applies the arithmetic conversion with RAT_QUAT_REDUCE_CONV,

3. converts the output in traditional form using QUAT_TRAD_CONV.

We show an example.

```
RATIONAL_QUAT_CONV
`(Hx(&1) + Hx(&2) * ii - jj) * cnj (Hx(&4) + Hx(&3) * kk)`;;
val it =
  |- (Hx(&1) + Hx(&2) * ii - jj) * cnj (Hx(&4) + Hx(&3) * kk) =
     Hx(&4) + Hx(&11) * ii + Hx(&2) * jj - Hx(&3) * k
```

3.3 Quaternionic polynomial normalization

HOL Light provides a general procedure for polynomial normalization, which unfortunately works only for commutative rings, so it can't be used in the quaternionic case. Hence, a new approach must be developed.

The goal of this section is to explain in details the construction of a specific conversion to perform calculations with quaternionic polynomials. Specifically, such a procedure will transform any algebraic quaternionic expression into an sum of terms

$$t_1 + \cdots + t_n$$

where:

1. each term t_i is a normalized product of a numeric coefficient and a quaternionic monomial;
2. the terms are listed according to a lexicographic order.

In principle, such a procedure can be generalized to work with arbitrary (non-commutative) rings (by abstracting over the constant and the characterizing theorems of an arbitrary ring), but, at the present stage, it is hardwired to the specific case of quaternions. The procedure is naive, in the sense that it does not use advanced optimization techniques (such as caching of subterms), however, it has been proven of practical usefulness in several occasions in our work.

Firstly, the simplest entities like monomials, terms and the rules to multiply them are considered. Monomials are products of literal powers as, for example, q^5p^2w while terms are monomials with numeric coefficient as $3q^5p^2w$.

Secondly, the methods that permit to compute addition and multiplication between ordered sums of terms are shown.

3.3.1 Syntax for binary and unary operators on quaternions

First of all, some OCaml functions (acting over HOL terms) to recognize the syntactic form of a given quaternionic expression are defined. The function `is_quat_real` returns true only if it is applied to a HOL term of the form `‘Hx x‘`. In the same way, the functions `is_quat_op` (one for every binary arithmetic operator `op = add, mul, pow`) return true if they are applied to a term of the form `‘q op p‘` and false otherwise. Here, we report some examples.

```
is_quat_real‘Hx x‘;;
val it : bool = true

is_quat_pow ‘q:quat pow 3‘;;
val it : bool = true

is_quat_mul ‘Hx x + q‘;;
val it : bool = false

is_quat_add ‘x + y:real‘;;
val it : bool = false
```

Note that, in the fourth case, the function `is_quat_add` returns false because the term `‘x + y‘`, even if is an addition, is not of type `‘:quat‘`.

Contextually, the functions `dest_quat_op`, that destruct additions, products and powers in their fundamental components as addends, factors and base and exponent respectively, are defined. Take a look to the following examples.

```
dest_quat_mul ‘q:quat * t‘;;
val it : term * term = (‘q‘, ‘t‘)

dest_quat_pow ‘q:quat pow 4‘;;
val it : term * term = (‘q‘, ‘4‘)

dest_quat_add ‘q:quat + t‘;;
val it : term * term = (‘q‘, ‘t‘)
```

3.3.2 Monomials and terms

The conversion `POW_MUL_FUSE_CONV` rewrites the usual *exponent rules* for multiplication of powers with the same base.

```
POW_MUL_FUSE_CONV ‘x:quat pow 3 * x pow 2‘;;
val it = |- x pow 3 * x pow 2 = x pow 5

POW_MUL_FUSE_CONV ‘x:quat * x‘;;
val it = |- x * x = x pow 2
```

Then, the case of a multiplication between a variable, with power, and a monomial is considered. The strategy consists to associate to the left and then to try to apply `POW_MUL_FUSE_CONV` on the left side. For example, let be `‘x pow 3 * (x pow 2 * y)‘` the term to be computed. Associating on the left, the output term `‘(x pow 3 * x pow 2) * y‘` is obtained. Then, the conversion `POW_MUL_FUSE_CONV` can be applied on the left side that is, `‘(x pow 3 * x pow 2)‘`.

Moreover, note that this strategy is successful only if the first variable of the monomial is equal to that which is multiplied. For example, it produces no effects in case of the product `‘x pow 3 * (y * x pow 2)‘`. In fact, since the lack of commutativity for the quaternionic product, the last term can't be rewritten as `‘x pow 3 * (x pow 2 * y)‘`. So, associating on the left, the term `‘(x pow 3 * y) * x pow 2‘` is obtained and `POW_MUL_FUSE_CONV` leaves it unchanged. The conversion `POW_MONOMIAL_MUL_FUSE_CONV` implements such a strategy.

```
POW_MONOMIAL_MUL_FUSE_CONV ‘x pow 3 * (x pow 2 * y)‘;;
val it = |- x pow 3 * x pow 2 * y = x pow 5 * y

POW_MONOMIAL_MUL_FUSE_CONV ‘x pow 3 * (y * x pow 2)‘;;
val it = |- x pow 3 * y * x pow 2 = x pow 3 * y * x pow 2
```

The conversion `MONOMIAL_MUL_CONV` performs multiplication between two monomials. Given two monomials `‘m1‘` and `‘m2‘`, the strategy used to compute `‘m1 * m2‘` is to control if the last conversion `POW_MONOMIAL_FUSE_CONV` can be applied to the last variable on the right of `‘m1‘` and the monomial `‘m2‘`. Otherwise, nothing has to be done. So, `MONOMIAL_MUL_CONV` acts as follows.

```
MONOMIAL_MUL_CONV ‘(x:quat pow 2 * y pow 3) * (y pow 3 * w)‘;;
val it = |- (x pow 2 * y pow 3) * y pow 3 * w = x pow 2 * y pow 6 * w

MONOMIAL_MUL_CONV ‘(x:quat pow 2 * y pow 3) * (z pow 3 * w)‘;;
val it = |- (x pow 2 * y pow 3) * z pow 3 * w =
              x pow 2 * y pow 3 * z pow 3 * w
```

Let's take a step forward considering multiplication between terms, instead of only monomials. Things are now more complicated because, working with numerical coefficients `‘Hx a‘`, calculations over them must be performed. The following theorems

```
th1 |- !a b x y. (Hx a * x) * Hx b * y = Hx (a * b) * x * y
th2 |- !a b x. (Hx a * x) * Hx b = Hx (a * b) * x
th3 |- !a b x. Hx a * Hx b * x = Hx (a * b) * x
th4 |- !a x y. (Hx a * x) * y = Hx a * x * y
th5 |- !a x y. x * Hx a * y = Hx a * x * y
```

are used to operate with numerical coefficients.

Given two terms `‘Hx a1 * m1‘` and `‘Hx a2 * m2‘` (with `‘m1‘` and `‘m2‘` monomials), the conversion `TERMS_MUL_CONV` acts on `‘(Hx a1 * m1) * (Hx a2 * m2)‘` essentially in two steps:

1. it tries to apply arithmetic simplifications about multiplication by `‘Hx(&0)‘` or `‘Hx(&1)‘`,

2. if the product to compute is of the form `‘Hx a * Hx b‘` (i.e. if `‘m1‘` and `‘m2‘` are equal to unit), it rewrites the theorem `HX_MUL`. Otherwise, it tries to rewrites, one by one, the previous theorems `th1`, `th2`, `th3`, `th4`, `th5`. After, it applies, when it is possible, the conversions `REAL_POLY_CONV` on the rand of `‘Hx‘` and `MONOMIAL_MUL_CONV` to the multiplication of the related monomial `‘m1 * m2‘`.

In practice, we observe the following behaviour.

```
TERM_MUL_CONV `(Hx (&2) * x pow 2 * y pow 3) * (Hx (&0))`;;
val it = |- (Hx (&2) * x pow 2 * y pow 3) * Hx (&0) = Hx (&0)

TERM_MUL_CONV `(Hx (&2) * x pow 2 * y pow 3) * (Hx (&3) * y pow 2 * w)`;;
val it =
  |- (Hx (&2) * x pow 2 * y pow 3) * Hx (&3) * y pow 2 * w =
     Hx (&6) * x pow 2 * y pow 5 * w
```

Then, another common operation in normalization of polynomials is considered: the sum of similar terms `Hx a * m + Hx b * m`. It is well know that, in case of two terms with the same literal part, they can be added simply adding their numerical coefficients. As before, in order to do this, the following theorems are proved

```
th1 |- !x y q. Hx x * q + Hx y * q = Hx (x + y) * q
th2 |- !x q. Hx x * q + q = Hx (x + &1) * q
th3 |- !x q. q + Hx x * q = Hx (&1 + x) * q
```

and then the conversion `TERMS_FUSE_ADD_CONV` is defined. The latter acts in three steps:

1. it tries to apply arithmetic simplifications about addition by `Hx(&0)`,
2. if the addition to compute is of the form `Hx a + Hx b` it rewrites `HX_ADD`. Otherwise, it tries to rewrites, one by one, the theorems `th1 th2 th3`. Then, it applies, when it is possible, the conversion `REAL_POLY_CONV` to the rand of `Hx`,
3. it tries to apply arithmetic simplifications about the left-multiplication by `Hx(&0)` or `Hx(&1)`, since the addition of the numerical coefficients can produce `Hx(&1)` or `Hx(&0)`.

Take a look to the following examples.

```
TERMS_FUSE_ADD_CONV `(Hx(&2) * x pow 2 * w) + (Hx(&3) * x pow 2 * w)`;;
val it =
  |- Hx (&2) * x pow 2 * w + Hx (&3) * x pow 2 * w = Hx (&5) * x pow 2 * w

TERMS_FUSE_ADD_CONV `(Hx(&2) * x pow 2 * w) + ((Hx(-- &2)) * x pow 2 * w)`;;
val it =
  |- Hx (&2) * x pow 2 * w + Hx (-- &2) * x pow 2 * w = Hx (&0)

TERMS_FUSE_ADD_CONV `(Hx(&2) * x pow 2 * w) + (Hx(-- &1) * x pow 2 * w)`;;
val it =
  |- Hx (&2) * x pow 2 * w + Hx (-- &1) * x pow 2 * w = x pow 2 * w

TERMS_FUSE_ADD_CONV `(Hx(&2) * x pow 2 * y) + (Hx(-- &1) * x pow 2 * w)`;;
Exception: Failure "safe_inserta".
```

Notice that, when the terms aren't similar like in the fourth example, the conversion fails and returns an error instead of leaves the expression unchanged.

3.3.3 Monomials and terms comparison

In order to manipulate and normalize polynomials, it is important to be able to compare and order terms. For this purpose, the OCaml function `compare: A -> A -> int` is used. Given an order relation $\leq$ over the universe A (of the type `:A`) and two elements $a, b \in A$, the integer number $compare(a, b)$ is:

- -1 if $a < b$,

- 0 if $a = b$,
- 1 if $b < a$.

The compare function acts over integer numbers and strings according to the standard order and the lexicographical order, respectively, as shown by the following examples.

```
compare 3 4;;
val it : int = -1

compare "x" "z";;
val it : int = 1

compare "x" "x";;
val it : int = 0
```

Now, some functions to compare literal powers, monomials and terms are defined using the *compare* function. The strategy, in case of literal powers `‘x pow n‘` and `‘y pow m‘`, is to compare firstly the bases `‘x‘`,`‘y‘` according to the lexicographical order and then, only in case that they are equal, compare the exponents `‘n‘` and `‘m‘` according to the standard order of natural numbers. Formally, the following function is defined to do that.

```
 let pow_compare tm1 tm2 =
    let b1,e1 = dest_literal_quat_pow tm1
    and b2,e2 = dest_literal_quat_pow tm2 in
    let c = compare b1 b2 in
    if c <> 0 then c else compare e1 e2;;
```

In such definition, `dest_literal_quat_pow` is the function that, given a power `‘q pow n‘` returns the couple `(‘q‘, n)`. We show some examples.

```
dest_literal_quat_pow ‘x pow 5‘;;
val it : term * int = (‘x‘, 5)

pow_compare ‘x pow 3‘ ‘y pow 1‘;;
val it : int = -1

pow_compare ‘x pow 3‘ ‘x pow 1‘;;
val it : int = 1

pow_compare ‘x pow 3‘ ‘x pow 3‘;;
val it : int = 0
```

As regards to monomials, the function `mon_compare` is given and it is defined by cases. Let `‘m1‘` and `‘m1‘` be two monomials, then:

- if both `‘m1‘` and `‘m2‘` are of the form `‘p1 * n1‘` and `‘p2 * n2‘` with `‘p1‘`, `‘p2‘` literal powers and `‘n1‘`, `‘n2‘` monomials respectively, then it returns the value of

  ```
   pow_compare ‘p1‘ ‘p2‘
  ```

  if it is not zero, otherwise `mon_compare` is applied recursively to `‘n1‘` and `‘n2‘`,
- if exactly one of `‘m1‘` and `‘m2‘` is of the form `‘p * n‘` it returns 1 or -1 in case of `‘m1‘` or `‘m2‘` respectively,
- if neither `‘m1‘` or `‘m2‘` are of the previous form then the function `pow_compare` is applied to `‘m1‘` and `‘m2‘`.

The formal recursive definition of `mon_compare` is the following and we report some examples.

```
let rec mon_compare m1 m2 =
    match is_quat_mul m1, is_quat_mul m2 with
    | true,true ->
        let c = pow_compare (lhand m1) (lhand m2) in
        if c <> 0 then c else
        mon_compare (rand m1) (rand m2)
    | true,false -> 1
    | false,true -> -1
    | false,false -> pow_compare m1 m2;;

mon_compare `x pow 3 * y pow 4` `z pow 2 * w`;;
val it : int = -1

mon_compare `x pow 3 * y pow 4` `z pow 2`;;
val it : int = 1
```

Since comparison between two terms, `t1` and `t2`, is essentially a comparison between their literal parts, it can be easily obtained from comparison between monomials distinguishing the following cases:

- if both `t1` and `t2` are of the form `Hx x * m1` and `Hx y * m2`, with `m1`, `m2` monomials, then it returns mon_compare `m1` `m2`,
- if exactly one of `t1` and `t2` has no literal part (i.e. is of the form `Hx x`) then it returns -1 or 1 if `t1 = Hx x` or `t2 = Hx x` respectively,
- if both `t1` or `t2` have no literal part then they are considered equal so it returns 0.

The formal function that compares terms is term_compare and it is defined as follows.

```
let term_compare t1 t2 =
    match is_quat_real t1, is_quat_real t2 with
    | true, false -> -1
    | false, true -> 1
    | true, true -> 0
    | false,false -> mon_compare (mon_of_term t1) (mon_of_term t2);;
```

Here, mon_of_term is an auxiliary function that, given a term `t1` of the form `Hx x * m1`, returns its literal part (i.e. the monomial) `m1`. We show some examples.

```
mon_of_term `Hx(&5) * x pow 2 * y pow 3`;;
val it : term = `x pow 2 * y pow 3`

term_compare `x pow 2` `x pow 3`;;
val it : int = -1

term_compare `x pow 2 * y pow 2` `x pow 2 * y pow 3`;;
val it : int = -1
```

3.3.4 Term insertion, addition, multiplication and power of polynomials

In the following, we will work with polynomials encoded in a normalized form as an ordered (with respect to the lexicographic order defined by term_compare) sum of terms

$$t_1 + t_2 + ... + t_n$$

with the convention that the addition is associated to the right $t_1 + (t_2 + (\dots))$. Informally, t_1 and $t_2 + ... + t_n$ can be thought as the *head* and the *tail* of p respectively. Note that the head is always well definend whereas tail can be empty. In this case, the polynomial p is reduced to its head, that is, the single term t_1. So, from now on, every time we will write `‘p + q‘`, we have in mind a polynomial with *head* `‘p‘` and *tail* `‘q‘`.

Term insertion. At this point, the recursive conversion `INSERT_TERM` that, given a normalized and ordered sum of terms, inserts another term in the right position, according to the order, is defined. Let `‘q + r‘` be an ordered and normalized polynomial, with leading term `‘q‘`, and `‘p‘` a new term. Then, the conversion `INSERT_TERM p (q + r)` acts essentially by cases:

1. if `‘p‘` is `‘Hx(&0)‘` then it makes the appropriate rewrites,
2. it computes `term_compare ‘p‘ ‘q‘` and this calculation produces three subcases:
   - 2.1. `term_compare ‘p‘ ‘q‘ = 0` means that `‘p‘` and `‘q‘` have the same literal part so they can be added using `TERMS_FUSE_ADD_CONV`,
   - 2.2. `term_compare ‘p‘ ‘q‘ > 0` means that `‘p‘` is "greater" than `‘q‘` so the theorem

     ```
     insert_term_th
        |- !p q r. p + (q + r) = q + (p + r)
     ```

     can be used to swap them and, subsequently, `INSERT_TERM` is applied recursively on `‘p‘` and `‘r‘` considering `‘r‘` as the sum of its head and tail,
   - 2.3. `term_compare ‘p‘ ‘q‘ < 0` means that `‘p‘` is "less" than `‘q‘` (hence than each terms of `‘r‘` since `‘q + r‘` is ordered) so there is nothing to do.

In practice, we have the following examples.

```
INSERT_TERM ‘x pow 3‘ ‘x pow 2 + x pow 4‘;;
val it = |- x pow 3 + x pow 2 + x pow 4 = x pow 2 + x pow 3 + x pow 4

INSERT_TERM ‘y:quat‘ ‘x:quat pow 2‘;;
val it = |- y + x pow 2 = x pow 2 + y
```

Polynomial addition. Let `‘p + q‘` and `‘r + s‘` be two polynomials. The conversion `PRE_ADD_CONV` that computes, orders and normalizes the addition `‘(p + q) + (r + s)‘` is provided. Also this time, such a conversion is defined by a case analysis:

1. if `‘q‘` or `‘s‘` are empty then `INSERT_TERM ‘p‘ ‘r + s‘`, or `INSERT_TERM ‘r‘ ‘p + q‘` respectively, are returned,
2. if both `‘q‘` or `‘s‘` are not empty then there are three possible subcases:
   - 2.1 if `term_compare ‘p‘ ‘r‘ = 0` then the theorem

     ```
     th_add1 =
       |- !p q r. (p + q) + (r + s) = (p + r) + (q + s)
     ```

     is rewritten to join togheter `‘p‘` and `‘r‘`.

     Then, `TERM_FUSE_ADD_CONV` is applied to `‘p + r‘` and `PRE_ADD_CONV` is applied recursively to `‘q + s‘`, considering `‘q‘` and `‘s‘` as sums of their heads and tails respectively,
   - 2.2 if `term_compare ‘p‘ ‘r‘ < 0` the theorem

     ```
     th_add2
       |- !p q r. (p + q) + (r + s) = p + (q + (r + s))
     ```

     is rewritten to isolate the smaller term `‘p‘` and then `‘PRE_ADD_CONV‘` is applied recursively to `‘(q + (r + s))‘`, considering `‘q‘` as sum of its head and tail respectively, 2.3 if `term_compare ‘p‘ ‘r‘ > 0` the theorem

```
th_add3
  |- !p q r. (p + q) + (r + s) = r + ((p + q) + s)
```

is rewritten to isolate the smaller term `‘r‘` and then `PRE_ADD_CONV` is applied recursively to `‘(p + q) + s‘`, considering `‘s‘` as sum of its head and tail.

We show an example.

```
PRE_ADD_CONV
‘(x pow 3 + Hx (&4) * x pow 5) + (x pow 2 + Hx(-- &5) * x pow 5)‘;;
val it =
  |- (x pow 3 + Hx (&4) * x pow 5) + x pow 2 + Hx (-- &5) * x pow 5 =
     x pow 2 + x pow 3 + Hx (-- &1) * x pow 5
```

Polynomial multiplication. Multiplication of polynomials is slightly more difficult than addition since it is not enough to add similar terms and then reorder them. The computation must be performed using the distributive law and the multiplication of terms.

In order to do this, two conversions `LDISTRIB` and `RDISTRIB` are provided. Given as input the product of two polynomials of the form `‘(p + r) * (q + s)‘`, they rewrite recursively the left- and right-distributive law obtaining `‘(p + r) * q + (p + r) * s‘` and `p * (q + s) + r * (q + s)‘` respectively. The latter two conversions do the appropriate simplifications, using `TERM_MUL_CONV`, in cases that `‘r‘` or `‘s‘` is the empty tail. Here, there are some examples.

```
LDISTRIB_CONV ‘(x pow 2 + x pow 3) * (x pow 5 + x pow 6)‘;;
val it =
  |- (x pow 2 + x pow 3) * (x pow 5 + x pow 6) =
     (x pow 2 + x pow 3) * x pow 5 + (x pow 2 + x pow 3) * x pow 6

RDISTRIB_CONV ‘(x pow 2 + x pow 3) * (x pow 5 + x pow 6)‘;;
val it =
  |- (x pow 2 + x pow 3) * (x pow 5 + x pow 6) =
     x pow 2 * (x pow 5 + x pow 6) + x pow 3 * (x pow 5 + x pow 6)

LDISTRIB_CONV ‘(x pow 2) * (x pow 5 + x pow 6)‘;;
val it =
  |- x pow 2 * (x pow 5 + x pow 6) = x pow 7 + x pow 8
```

At this point, the main conversion `PRE_MUL_CONV`, that computes `‘(p + r) * (q + s)‘`, is defined by another case analysis:

1. if `‘r‘` or `‘s‘` is an empty tail then `LDISTRIB_CONV`, or `RDISTRIB_CONV` respectively, is applied,
2. if both `‘r‘` and `‘s‘` are not the empty tail the theorem

   ```
   th_mul
     |- !p q r s. (p + q) * (r + s) = p * r + (p * s + q * r) + q * s
   ```

   is rewritten to divide the difficulties. In fact, remembering that `‘p‘` and `‘r‘` are terms whereas `‘q‘` and `‘s‘` are sums of terms, it is possible to apply:

   - `TERM_MUL_CONV` to `‘p * r‘`,
   - `LDISTRIB_CONV` to `‘p * s‘` and `RDISTRIB_CONV` to `‘q * r‘`, then, `PRE_ADD_CONV` is applied to the result,
   - `PRE_MUL_CONV` recursively to `‘q * s‘` and then `PRE_ADD_CONV` to the result.

Note that, even if `‘p * r‘` is certainly the head of the polynomial product, the terms produced by the multiplication have to be added with `PRE_ADD_CONV` to normalize the result. We show an example.

```
PRE_MUL_CONV
‘(Hx (&8) * x pow 2 + Hx(-- &2) *  x pow 3) * (Hx(&4) * x pow 4 + x pow 5)‘;;
val it =
  |- (Hx (&8) * x pow 2 + Hx (-- &2) * x pow 3) *
     (Hx (&4) * x pow 4 + x pow 5) =
     Hx (&32) * x pow 6 + Hx (-- &2) * x pow 8
```

Polynomial powers. As said in the previous section (3.2), powers can't be computed naively by iteration because it would be too expensive. In fact, if the exponent `‘n:num‘` of the power `‘x pow n‘` is large, the multiplication rule of quaternions can't be rewritten n-times. Again, we use the *exponentiation by squaring* technique to perform calculations by proving the following formal theorems

```
QUAT_POW_DENUMERAL
  |- x pow NUMERAL n = x pow n

QUAT_POW_0
  |- !x. x pow _0 = Hx (&1)

QUAT_POW_BIT0
  |- !x n. x pow BIT0 n = (x * x) pow n

QUAT_POW_BIT1
  |- !x n. x pow BIT1 n = x * (x * x) pow n
```

where we use the internal HOL Light binary representation of natural numbers explained in chapter 1 (section 1.2).

Therefore, given a power `‘q pow n‘` to be computed, the conversion `PRE_POW_CONV` acts essentially in two steps:

1. it rewrites `QUAT_POW_DENUMERAL`,
2. it tries to rewrite one of `QUAT_POW_0`, `QUAT_POW_BIT0` or `QUAT_POW_BIT1` then applies `PRE_MUL_CONV` and itself, recursively, on the appropriate subterms.

For example, computing `‘q pow 5‘`, the action of `PRE_POW_CONV` is described step by step:

- it rewrites `QUAT_POW_DENUMERAL` and `QUAT_POW_BIT1` obtaining the term `‘q * ((q * q) pow 4)‘`,
- it computes `‘q * q‘` with `PRE_MUL_CONV` and `‘(q * q) pow 4‘` with `PRE_POW_CONV` recursively,
- finally, it finishes the computation using again `PRE_MUL_CONV` on the whole term `‘q * (q * q) pow 4‘`.

We give a concrete example.

```
PRE_POW_CONV ‘(p + Hx(&2)) pow 5‘;;
val it =
  |- (p + Hx (&2)) pow 5 =
     p pow 5 +
     Hx (&32) +
     Hx (&80) * p +
     Hx (&80) * p pow 2 +
     Hx (&40) * p pow 3 +
     Hx (&10) * p pow 4
```

3.3.5 Glue all together and the main call

All the previous conversions, one for each basic operation, are glued in the following recursive definition.

```
let rec POLY_CONV tm =
    if is_quat_add tm then ADD_CONV tm else
    if is_quat_mul tm then MUL_CONV tm else
    if is_quat_pow tm then POW_CONV tm else
    ALL_CONV tm
  and ADD_CONV tm = (BINOP_CONV POLY_CONV THENC PRE_ADD_CONV) tm
  and MUL_CONV tm = (BINOP_CONV POLY_CONV THENC PRE_MUL_CONV) tm
  and POW_CONV tm = (LAND_CONV POLY_CONV THENC
                     RAND_CONV NUM_NORMALIZE_CONV THENC
                     TRY_CONV PRE_POW_CONV) tm;;
```

As we can see by the explicit definition, a structural analysis, on the term `‘tm‘` that must be computed, is made. It divides three cases:

1. if `‘tm‘` is an addition `‘tm1 + tm2‘` then, firstly, `POLY_CONV` is applied recursively to `‘tm1‘` and `‘tm2‘`, secondly, `PRE_ADD_CONV` is applied to `‘tm‘`,
2. if `‘tm‘` is a multiplication `‘tm1 * tm2‘` then, firstly, `POLY_CONV` is applied recursively to `‘tm1‘` and `‘tm2‘`, secondly, `PRE_MUL_CONV` is applied to `‘tm‘`,
3. if `‘tm‘` is a power `‘tm1 pow tm2‘` then, firstly, `POLY_CONV` is applied recursively to `‘tm1‘`, secondly, `PRE_POW_CONV` is applied to `‘tm‘`.

Now, the main conversion `QUAT_POLY_CONV` is defined.

```
let QUAT_POLY_CONV : conv =
  QUAT_DESUGAR_CONV THENC POLY_CONV;;
```

The conversion `QUAT_DESUGAR_CONV` performs an initial normalization rewriting some easy identities proved in the following theorem.

```
QUAT_DESUGAR_CLAUSES
  |- (!x. Hx (&0) + x = x) /\
     (!x. --x = Hx (-- &1) * x) /\
     (!x y. x - y = x + Hx (-- &1) * y) /\
     (!x. Hx (&1) * x = x) /\
     (!x. x * Hx (&1) = x) /\
     (!x. Hx (&0) * x = Hx (&0)) /\
     (!x. x * Hx (&0) = Hx (&0)) /\
     (!x. x pow 0 = Hx (&1)) /\
     (!x y. cnj (x + y) = cnj x + cnj y) /\
     (!x y. cnj (x * y) = cnj y * cnj x) /\
     (!x. cnj (inv x) = inv (cnj x)) /\
     (!x n. cnj (x pow n) = cnj x pow n) /\
     (!a. cnj (Hx a) = Hx a) /\
     cnj ii = Hx (-- &1) * ii /\
     cnj jj = Hx (-- &1) * jj /\
     cnj kk = Hx (-- &1) * kk
```

As for example we can compute the term

```
‘q - Hx(&5)) pow 3 + (q - Hx(&2)) * (q + Hx(&2))‘
```

with `QUAT_POLY_CONV`. In fact, the command

```
QUAT_POLY_CONV`(q - Hx(&5)) pow 3 + (q - Hx(&2)) * q`;;
```

produces automatically the following theorem.

```
val it  =
  |- (q - Hx (&5)) pow 3 + (q - Hx (&2)) * q =
     Hx (-- &125) + Hx (&73) * q + Hx (-- &14) * q pow 2 + q pow 3
```

A simple conversion to prove equality between quaternionic polynomials is derived from QUAT_POLY_CONV. The idea is basic: in case of a polynomial equality `p = q`, both `p` and `q` are normalized with `QUAT_POLY_CONV`and then they are compared. Such a conversion and the corresponding rule are define as follows.

```
let QUAT_EQ_CONV : conv =
  BINOP_CONV QUAT_POLY_CONV THENC REWR_CONV REFL_CLAUSE;;

let QUAT_POLY tm = prove(tm, REPEAT GEN_TAC THEN CONV_TAC QUAT_EQ_CONV);;
```

We show an example.

```
QUAT_POLY`(p + Hx (&3)) pow 2 - (p + Hx (&3)) * (p - Hx (&3)) =
          Hx (&18) + Hx (&6) * p`;;
  val it =
  |- (p + Hx (&3)) pow 2 - (p + Hx (&3)) * (p - Hx (&3)) =
     Hx (&18) + Hx (&6) * p
```

Finally, the conversions presented in the last two sections (3.2 and 3.3) enable us to automate a very large number of long and tedious calculations about quaternions (see for instance the examples about PH-curves shown in Part II of this thesis). They has been very useful in the formalization of the analytic and geometric results that we will show in the next two sections.

3.4 Elementary quaternion analysis

Passing from algebra to analysis, a series of technical theorems about the analytical behaviour of the algebraic operations are proved. A systematic formalization of such results, from the point of view of limits, continuity and derivatives, is conducted and it brought to the formalization of more than fifty theorems overall. Some of them are indeed trivial, some are less immediate and forced to dive into a technical $\epsilon\delta$-reasoning.

Limits and continuity. A set of composition theorems that link the notion of limit and continuity with the algebraic quaternionic operations are proved.

It is well known that for every pair of functions $f, g : \mathbb{R}^n \to \mathbb{R}^m$ the limit operator (hence also continuity) distributes over addition, subtraction and negation. The HOL Light Multivariate library already provides some theorems that formalize these properties. For example we report the following theorems about addition.

```
LIM_ADD
  |- !net f g l m. (f --> l) net /\ (g --> m) net
                   ==> ((\x. f x + g x) --> l + m) net

CONTINUOUS_ADD
  |- !f g net. f continuous net /\ g continuous net
               ==> (\x. f x + g x) continuous net
```

The same results are proved for the other quaternionic operations like product, power and inverse as shown by the following formal theorems:

- **Product:**

```
LIM_QUAT_MUL
  |- !net f g l m. (f --> l) net /\ (g --> m) net
                   ==> ((\x. f x * g x) --> l * m) net

CONTINUOUS_QUAT_MUL
  |- !net f g. f continuous net /\ g continuous net
               ==> (\x. f x * g x) continuous net
```

- **Power:**

```
LIM_QUAT_POW
  |- !net f l n. (f --> l) net
                 ==> ((\x. f x pow n) --> l pow n) net

CONTINUOUS_QUAT_POW
  |- !net f n. f continuous net
               ==> (\x. f x pow n) continuous net
```

- **Inverse:**

```
LIM_QUAT_INV
  |- !net f l. (f --> l) net /\ ~(l = Hx (&0))
               ==> ((\x. inv (f x)) --> inv l) net

CONTINUOUS_QUAT_INV
  |- !net f. f continuous net /\ ~(f (netlimit net) = Hx (&0))
             ==> (\x. inv (f x)) continuous net
```

Differentiability. Next, the differential structure is considered and the derivative of the basic quaternionic operations is computed. Notice that, if f is a quaternionic valued function, the derivative $\mathrm{D}f_{q_0}(x)$ is a quaternion (natural identification of the tangent space $T_{f(q_0)}\mathbb{H} \simeq \mathbb{H}$). As in the case of limits, for addition, subtraction and negation the situation is easy. Such theorems are special instance of more general ones about functions $f : \mathbb{R}^n \to \mathbb{R}^m$ that are already proved in the Multivariate library of HOL Light.

```
HAS_DERIVATIVE_ADD
  |- !f f' g g' net.
       (f has_derivative f') net /\ (g has_derivative g') net
       ==> ((\x. f x + g x) has_derivative (\h. f' h + g' h)) net

HAS_DERIVATIVE_SUB
  |- !f f' g g' net.
       (f has_derivative f') net /\ (g has_derivative g') net
       ==> ((\x. f x - g x) has_derivative (\h. f' h - g' h)) net

HAS_DERIVATIVE_NEG
  |- !f f' net.
       (f has_derivative f') net
       ==> ((\x. --f x) has_derivative (\h. --f' h)) net
```

As regards the product, given two differentiable quaternionic functions $f(q)$ and $g(q)$, the derivative of their product in $q_0 \in \mathbb{H}$ is

$$\frac{\mathrm{d}\big(f(q)g(q)\big)}{\mathrm{d}q}|_{q_0}(x) = f(q_0)\,\mathrm{D}g_{q_0}(x) + \mathrm{D}f_{q_0}(x)g(q_0).$$

In our formalism, the previous formula becomes the following theorem.

```
QUAT_HAS_DERIVATIVE_MUL
  |- !f f' g g' q.
       (f has_derivative f') (at q) /\ (g has_derivative g') (at q)
       ==> ((\x. f x * g x) has_derivative
            (\x. f q * g' x + f' x * g q)) (at q)
```

A direct consequence is the following formula for the power

$$\frac{\mathrm{d}q^n}{\mathrm{d}q}|_{q_0}(x) = \sum_{i=1}^{n} q_0^{n-i} x q_0^{i-1},$$

that is the HOL theorem

```
QUAT_HAS_DERIVATIVE_POW
  |- !q0 n. ((\q. q pow n) has_derivative
             (\h. vsum (1..n) (\i. q0 pow (n - i) * h * q0 pow (i - 1))))
            (at q0)
```

which is easily proved by induction using the derivative of the product. Note the difference from the usual formula of derivative of a power due to the lack of commutativity for the quaternionic product.

Finally, a straightforward but important observation is the next. Let $R_p\colon \mathbb{H} \to \mathbb{H}$ be the right multiplication by the quaternion p, i.e., $R_p(x) = xp$. Since R_p is $\mathbb{R}$-linear, it holds that $\mathrm{D}R_p = R_p$, that is, formally, the following theorem.

```
|- !net p. ((\q. q * p) has_derivative (\q. q * p)) net
```

3.5 The geometry of quaternions

As regards to geometry, it is very useful to consider a number of different possible decompositions for a quaternion q, as briefly sketched in the following schema (here $\mathbb{I} = \mathbb{R}^3$ can be interpreted, depending on the context, as the imaginary part of $\mathbb{H}$ or the 3-dimensional space):

$$\begin{aligned}
q &= \underbrace{a_0}_{\mathrm{Re}\,q} + \underbrace{a_1\,\mathbf{i} + a_2\,\mathbf{j} + a_3\,\mathbf{k}}_{\mathrm{Im}\,q} && \in \mathbb{H} = \mathbb{R} \oplus \mathbb{I} \\
&= \underbrace{a_0}_{\text{scalar}} + \underbrace{a_1\,\mathbf{i} + a_2\,\mathbf{j} + a_3\,\mathbf{k}}_{\text{3d-vector}} && \in \mathbb{R}^4 = \mathbb{R} \oplus \mathbb{R}^3 \\
&= \underbrace{a_0 + a_1\,\mathbf{i}}_{z\in\mathbb{C}} + \underbrace{(a_3 + a_4\,\mathbf{i})}_{w\in\mathbb{C}}\,\mathbf{j} && \in \mathbb{H} \simeq \mathbb{C} \oplus \mathbb{C}
\end{aligned}$$

Regarding a quaternion $a_0 + \mathbf{a}$ as an element of $\mathbb{R} \oplus \mathbb{R}^3$, the projections

$$a_0 \in \mathbb{R}, \qquad \mathbf{a} = a_1\mathbf{i} + a_2\mathbf{j} + a_3\mathbf{k} \in \mathbb{R}^3$$

are called *pure scalar* and *pure vector* respectively. Since HOL Light doesn't have subtypes $a \in \mathbb{R}$ and $\mathbf{a} \in \mathbb{R}^3$ can't be considered, from a formal point of view, as elements of $\mathbb{H}$ at the same time as mathematicians usually do. Hence, in our formalism, a *pure scalar* is represented by `` `Hx a` `` (with `` `a:real` ``) and a *pure vector* by `` `Hv a` `` (with `` `a:real^3` ``). A unit vector is a pure vector $\mathbf{u}$ such that $\|\mathbf{u}\| = 1$ and by direct computation can be proved that $u^2 = -1$ if and only if $u = \mathbf{u}$ is a pure unit vector, for every $u \in \mathbb{H}$.

It easy to show, again by direct computation, that the quaternionic product encodes both the scalar and vector products of $\mathbb{R}^3$ in fact, let $a_0 + \mathbf{a}$ and $b_0 + \mathbf{b}$ be two quaternions, we have that

$$(a_0 + \mathbf{a})(b_0 + \mathbf{b}) = (a_0 b_0 - \mathbf{a} \cdot \mathbf{b}) + (a_0\mathbf{b} + b_0\mathbf{a} + \mathbf{a} \times \mathbf{b}). \tag{3.5.1}$$

If we consider pure vectors, that is, vectors such that $a_0 = b_0 = 0$, the previous formula becomes

$$\mathbf{ab} = (-\mathbf{a} \cdot \mathbf{b}) + (\mathbf{a} \times \mathbf{b}) \tag{3.5.2}$$

that is, the HOL Light following formal theorem.

```
MUL_HV_EQ_CROSS_DOT
  |- !a b. Hv a * Hv b = Hv (a cross b) - Hx (a dot b)
```

Moreover, quaternions can be used to encode orthogonal transformations. For $q \neq 0$, the conjugation map is defined as

$$c_q : \mathbb{H} \longrightarrow \mathbb{H}$$
$$c_q(x) := q^{-1}\, x\, q$$

and we have

$$c_{q_1} \circ c_{q_2} = c_{q_1 q_2}, \qquad c_q^{-1} = c_{q^{-1}}.$$

One important special case is when q unitary, i.e., $\|q\| = 1$ for which we have $q^{-1} = \bar{q}$ (the conjugate) and thus

$$c_q(x) = \bar{q}\, x\, q.$$

It holds the following proposition

Proposition 3.5.1. *If $q^2 = -1$ then $-c_q \colon \mathbb{R}^3 \to \mathbb{R}^3$ is the reflection w.r.t. the orthogonal space of the quaternion q, that is $q^\perp = \{w \in \mathbb{H} \mid \langle q, w\rangle = 0\}$ where $\langle , \rangle$ is the Euclidean inner product of $\mathbb{R}^4$.*

that has a corresponding statement proved in HOL Light.

```
REFLECT_ALONG_EQ_QUAT_CONJUGATION
  |- !v. ~(v = vec 0)
         ==> reflect_along v = \x. --HIm (inv (Hv v) * Hv x * Hv v)
```

The Cartan-Dieudonné theorem asserts that any orthogonal transformation $f \colon \mathbb{R}^n \longrightarrow \mathbb{R}^n$ is the composition of at most n reflections. Using this and the previous proposition, we get the following result.

Proposition 3.5.2. *Any orthogonal transformation $f : \mathbb{R}^3 \longrightarrow \mathbb{R}^3$ is of the form*

$$f = c_q \quad \text{or} \quad f = -c_q, \qquad \|q\| = 1.$$

The corresponding formalization is the following.

```
ORTHOGONAL_TRANSFORMATION_AS_QUAT_CONJUGATION
  |- !f. orthogonal_transformation f
         ==> (?q. norm q = &1 /\
                  ((!x. f x = HIm (inv q * Hv x * q)) \/
                   (!x. f x = --HIm (inv q * Hv x * q))))
```

Part II

APPLICATIONS OF QUATERNIONS

Introduction

The main goal of this part of the work is to explain in details the formalization of the basics of two of the most recent theories which are based on quaternions. More precisely, we give the definition and some basic theorems about *Slice regular functions* [Gentili and Struppa, 2006] (Chapters 4 and 5) and *Pythagorean-Hodograph curves* [Farouki, 2009] (Chapter 6).

Slice regular functions play a central role in quaternionic analysis because they generalize, in a very suitable way, the concept of complex holomorphic functions to the quaternionic case.

As regards to PH-curves, they are polynomial curves that present many theoretical properties an practical advantages from a computational point of view. Therefore, they are very useful in Computer-Aided Design (also known as CAD), digital motion control, path planning, robotics applications and animation.

Since the relevance of such theories, their formalizations are interesting on their own but, at the same time, they are good tests for the formal framework about quaternions presented in the first part of this thesis.

Outline of the code

The source code of this part of the work is reachable at the url

https://bitbucket.org/gabra/phdthesis/src/master/Application%20of%20Quaternions/

and it is divided in three subdirectories.

- **Complex** - Prerequisites about real and complex analysis which are needed for our purposes but that are not strictly related to quaternions (Included in HOL Light on Mon. 10th April 2017).
  - **limsup.hl** - Limit superior, limit inferior.
  - **root_test.hl** - Root test for series.
  - **cauchy_hadamard.hl** - Cauchy-Hadamard radius of convergence.
- **Slice_Regular** - Gentili and Struppa's definition of slice regular functions and proof of the existence of series expansion centred in the origin.
  - **misc.hl** - Miscellanea.
  - **slice.hl** - Slice regular functions.
  - **analytic.hl** - Power series expansion of slice regular functions centred in the origin.
  - **cauchy_hadamard.hl** - Copy of the previous file in the directory Complex (here it serves to made the code about slice regular functions independent, that is, loadable without loading the directory Complex.)
- **PH_Curve** - Definition of Farouki's Pythagorean-Hodograph curves.
  - **ph_curves.hl** - Quaternionic representation of spacial PH-curves, cubic and quintic PH Hermite interpolation curves.

Chapter 4

Slice regular functions

4.1 Historical overview about regularity for quaternionic functions

Complex holomorphic functions have a central role in mathematics, so, given the deep link and the evident analogy between complex numbers and quaternions, it is natural to seek for a theory of *quaternionic holomorphic functions*. Unfortunately, a more careful investigation shows that the situation is less simple than expected. Naive attempts to generalize the complex case to the quaternionic case fail, because they lead to conditions which are either too strong or too weak and do not produce *interesting* classes of functions. For instance, asking for a function $f : \mathbb{H} \to \mathbb{H}$ to be *quaternion-differentiable*, i.e., imposing that, for $h \in \mathbb{H}$,

$$\lim_{h \to 0} h^{-1}(f(q+h) - f(q))$$

exists for all $q \in \mathbb{H}$[1], implies that f is an affine function of the form $f(q) = a + qb$ for some $a, b \in \mathbb{H}$. This result definitively shows that a naive approach is inadequate to replicate the richness of the theory of holomorphic functions of one complex variable.

Fueter, in the 1920s, proposed a definition of *regular* quaternionic function generalizing the Cauchy-Riemann operator. He defined a function to be *regular* if it solves the equation

$$\frac{\partial f}{\partial \bar{q}} = \frac{1}{4}\left(\frac{\partial}{\partial q_0} + \mathbf{i}\frac{\partial}{\partial q_1} + \mathbf{j}\frac{\partial}{\partial q_2} + \mathbf{k}\frac{\partial}{\partial q_3}\right) f \equiv 0.$$

It turns out that the operator $\frac{\partial f}{\partial \bar{q}}$ is a very good analog of the Cauchy-Riemann operator in the sense that functions, which are solutions of the above equation, enjoy many of the key properties of the holomorphic functions. A full theory of such functions has been extensively studied and developed also in several variables. Although Fueter's regular functions have significant applications to physics and engineering, they present also some undesirables aspects. For example, the identity function and the polynomials $P(q) = a_0 + a_1 q + \ldots + a_n q^n$, $a_i \in \mathbb{H}$ are not Fueter-regular. Even if a more detailed discussion on this subject is far beyond the goal of this work, what we have said so far is enough to motivate the search of an alternative definition of *regularity*. To the interested reader we recommend Sudbery's excellent survey [Sudbery, 1979].

A different definition of regularity in the quaternionic context was given by Cullen in the 1960s [Cullen, 1965], considering the solution of the equation

$$\left(\frac{\partial}{\partial q_0} + \frac{\operatorname{Im}(q)}{r}\frac{\partial}{\partial r}\right) f \equiv 0$$

[1]Remember that, since the lack of commutativity both right and left quotient can be defined inducing different definitions, even if specular, of differentiability.

where $\mathrm{Im}(q) = q_1\mathbf{i} + q_2\mathbf{j} + q_3\mathbf{k}$ and $r = \|\mathrm{Im}(q)\| = \sqrt{q_1^2 + q_2^2 + q_3^2}$.

Gentili and Struppa recently restated and developed Cullen's definition and, in their seminal paper in 2006 [Gentili and Struppa, 2006], they introduced the definition of *slice regular* function. Moreover, they proved that such functions can be expansed in series in an appropriate open ball centred in the origin. Slice regular functions are now a stimulating and active subject of research for several mathematicians world-wide. A comprehensive introduction on the foundations of this new theory can be found in the book of Gentili, Stoppato and Struppa [Gentili et al., 2013].

In this chapter, we will use the framework about quaternions presented in the previous part to formalize in HOL Light the basics of the theory of slice regular functions, essentially the main results of the already cited seminal paper of Gentili and Struppa.

4.2 Imaginary unit ball and Cullen slices

A real 2-dimensional subspace $L \subset \mathbb{H}$ containing the real line is called a *slice* (or *Cullen slice*) of $\mathbb{H}$. The key fact is that the quaternionic product becomes commutative when it is restricted on a slice, that is, if p, q are in the same slice L, then $pq = qp$. In other words, each slice L can be seen as a copy of the complex field $\mathbb{C}$. More precisely, for every quaternionic imaginary unit (i.e., a unitary imaginary quaternion) $I \in \mathbb{S} = \{q \in \mathbb{H} \mid q^2 = -1\}$, the set

$$L_I = \mathrm{Span}\{1, I\} = \mathbb{R} \oplus \mathbb{R}I$$

is a slice and the injection $j_I \colon \mathbb{C} \to L_I \subset \mathbb{H}$, defined by

$$j_I \colon x + y\mathbf{i} \mapsto x + yI,$$

is a field homomorphism. It is easy to show that, for every quaternion q, there exists $I \in \mathbb{S}$ such that $q \in L_I$. If $q \in \mathbb{R}$ it is trivial because, by definition, it holds that $\mathbb{R} \subset L_I$ for all $I \in \mathbb{S}$. Otherwise, if $q \notin \mathbb{R}$, and thus $\|\mathrm{Im}\, q\| \neq 0$, the unitary quaternion $\frac{\mathrm{Im}(q)}{\|\mathrm{Im}(q)\|} \in \mathbb{S}$ does the job so, we have that $\mathbb{H} = \bigcup_{I \in \mathbb{S}} L_I$.
The formal counterparts of L_I and j_I, that we defined in HOL Light, are the following.

```
let cullen_slice = new_definition
  `cullen_slice (i:quat) = span{Hx(&1),i}`;;

let cullen_inc = new_definition
  `cullen_inc i z = Hx(Re z) + Hx(Im z) * i`;;
```

Notice that, since HOL Light admits only total functions, the latter two are defined also when `i:quat` is real. In this case, the set `cullen_slice i` is not a *Cullen slice* since it coincides with the real line. We will use the function `cullen_slice` always under the appropriate assumption that $I \in \mathbb{S}$ that is, formally, `i pow 2 = --Hx(&1)`.

4.2.1 Imaginary unit ball $\mathbb{S} \subset \mathbb{H}$

Now, we focus the attention on the properties of $\mathbb{S}$ and, more precisely, we study orthogonal imaginary units.

The set $\mathbb{S}$ is defined through the quaternionic product as the set of the elements whose square is equal to -1. Alternatively, it can be characterized, from a more geometrical point of view, trough the real component and the norm. In fact, for every quaternion $I \in \mathbb{H}$, it holds that it has square equal to minus one if and only if it is unitary (i.e. has norm equal to one) and its real part is equal to zero, that is,

$$I^2 = -1 \qquad \text{if and only if} \qquad \mathrm{Re}(I) = 0 \quad \text{and} \quad \|I\| = 1. \tag{4.2.1}$$

This implies that $\mathbb{S} = \{I \in \mathbb{H} \mid \mathrm{Re}(I) = 0 \text{ and } \|I\| = 1\}$ thus, it is isomorphic to the usual 2-sphere in $\mathbb{R}^3$. The equivalence (4.2.1) becomes the following HOL theorem.

```
QUAT_IMG_UNIT_IFF_IMG_SPHERE
  |- !i. i pow 2 = --Hx(&1) <=> norm i = &1 /\ Re i = &0
```

Another useful property of imaginary units is that, given $I, J \in \mathbb{S}$ such that $I \perp J$ (i.e. $\langle I, J\rangle = 0$), their product IJ is still an imaginary unit orthogonal to both I and J. Since the facts that all imaginary units are naturally orthogonal to $1 \in \mathbb{H}$, and orthogonal vectors are independent, we have that, for every $I, J \in \mathbb{S}$ such that $I \perp J$, the set

$$\{1,\ I,\ J,\ IJ\}$$

is an orthogonal basis of $\mathbb{H} \simeq \mathbb{R}^4$. In HOL Light, we have the following theorems where the predicates `‘orthogonal:real^N->real^N->bool‘` refers to the canonical dot product in $\mathbb{R}^n$ ($\mathbb{R}^4$ in the specific case of quaternions).

```
QUAT_IMG_UNIT_ORTHOGONAL_PRODUCT_UNIT
  |- !i j. i pow 2 = --Hx(&1) /\ j pow 2 = --Hx(&1) /\ orthogonal i j
          ==> (i * j) pow 2 = --Hx(&1)

QUAT_IMG_UNIT_ORTHOGONAL_PRODUCT_ORTHOGONAL
  |- !i j. i pow 2 = --Hx(&1) /\ j pow 2 = --Hx(&1)
          ==> orthogonal i (i * j) /\ orthogonal j (i * j)

QUAT_ORTHONORMAL_BASIS_ORTHOGONAL
  |- !i j. i pow 2 = --Hx(&1) /\ j pow 2 = --Hx(&1) /\ orthogonal i j
          ==> pairwise orthogonal {Hx(&1), i, j, i * j}‘,

QUAT_ORTHONORMAL_BASIS_INDEPENDENT
  |- !i j. i pow 2 = --Hx(&1) /\ j pow 2 = --Hx(&1) /\ orthogonal i j
          ==> independent {Hx (&1), i, j, i * j}‘,

QUAT_IM_UNIT_ORTHOGAL_BASIS
  |- !i j. i pow 2 = --Hx(&1) /\ j pow 2 = --Hx(&1) /\ orthogonal i j
          ==> span {Hx (&1), i, j, i * j} = (:real^4)
```

4.2.2 Cullen slices

As said before, a *Cullen slice* is a real 2-dimensional subspace of $\mathbb{H}$ containing the real line (more precisely it is enough to require that it contains 1). Since every $I \in \mathbb{S}$ is orthogonal (hence independent) to 1 we have that L_I is a *Cullen slice* for all $I \in \mathbb{S}$. Moreover, it turns out that for every slice L, there exists $I \in \mathbb{S}$ such that $L = L_I$. Such an imaginary unit is not unique since for instance $L_I = L_{-I}$. The first assertion is proved observing that, for every *Cullen slice* L, there exists $q \in L$ such that $q \notin \mathbb{R}$ since $\dim L = 2$, by definition, and $\dim \mathbb{R} = 1$. Since $q \notin \mathbb{R}$ we have that $\|\operatorname{Im} q\| \neq 0$. So, let be $I = \frac{\operatorname{Im}(q)}{\|\operatorname{Im}(q)\|}$ then, by equivalence (4.2.1), it is an element of $\mathbb{S}$ and it holds that $q = \operatorname{Re}(q) + \|\operatorname{Im}(q)\|\, I \in L_I$. Therefore, L_I is a 2-dimensional subspace of $\mathbb{H}$ containing the real line such that $L_I \cap (L - \mathbb{R}) \neq \emptyset$, this implies that $L = L_I$. The second statement is proved easily from the definition of L_I by direct computation. All these simple results are formalized in the following theorems.

```
SUBSPACE_CULLEN_SLICE
  |- !i. subspace (cullen_slice i)

DIM_CULLEN_SLICE
  |- !i. i pow 2 = --Hx(&1) ==> dim (cullen_slice i) = 2

REAL_IN_CULLEN_SLICE
  |- !i q. real q ==> q IN cullen_slice i
```

```
CULLEN_SLICE_NEG
  |- !i. cullen_slice i = cullen_slice (--i)


CULLEN_SLICE_SPAN
  |- !l. subspace l /\ dim l = 2 /\ Hx(&1) IN l
         ==> (?i. i pow 2 = --Hx(&1) /\ l = cullen_slice i)
```

With similar arguments, it is shown that, for every quaternion $q \in \mathbb{H}$, there exists a *Cullen slice* L_I such that $q \in L_I$. Such a slice is unique if $q \notin \mathbb{R}$, otherwise it holds that $q \in L_I$ for every $I \in \mathbb{S}$. In any case, there exist $I \in \mathbb{S}$ and unique $x, y \in \mathbb{R}$ such that $q = x + yI \in L_I$. Note that, the real numbers x, y are unique depending on the choice of I (since $L_I = L_{-I}$) in fact, they are uniquely determined only if we fix I or $-I$.
Formally, we prove the following theorems.

```
QUAT_IN_SLICE
  |- !q. ?i. i pow 2 = --Hx(&1) /\ q IN cullen_slice i

CULLEN_DECOMPOSITION
  |- !q. ?j x y. j pow 2 = --Hx(&1) /\ q = Hx(x) + Hx(y) * j

CULLEN_SLICE_DECOMPOSITION
  |- !l. subspace l /\ dim l = 2 /\ Hx(&1) IN l
         ==> (?i. i pow 2 = --Hx(&1) /\
                  (!q. q IN l
                       ==> (?x y. q = Hx x + Hx y * i /\
                                  (!x' y'. q = Hx x' + Hx y' * i
                                           ==> x' = x /\ y' = y))))
```

As mentioned in the introduction, the key properties of the *Cullen slices* is that if we restrict the quaternionic product to a single slice L, it becomes commutative, inducing the complex structure on L. In fact, the following formal statement is proved.

```
QUAT_MUL_SYM
  |- !l. subspace l /\ dim l = 2 /\ Hx(&1) IN l /\ p IN l /\ q IN l
         ==> p * q = q * p
```

4.2.3 Embedding of the complex plane into a Cullen slice

An easy but important observation is that every Cullen slices L_I is a subfield of $\mathbb{H}$ which is naturally isomorphic to $\mathbb{C}$ trough the field isomorphism j_I, denoted `cullen_inc i` in HOL Light.

So, let be $I \in \mathbb{S}$ an imaginary unit, the main properties of $j_I : \mathbb{C} \to L_I$ are the following.

- It is injective and surjective that is, for all $z_1, z_2 \in \mathbb{C}$ it holds that

$$j_I(z) = j_I(z') \Leftrightarrow z = z'$$

  and, for all $q \in L_I$, there exists $z \in \mathbb{C}$ such that

$$q = j_I(z)$$

  hence, we have that $j_I(\mathbb{C}) = L_I$. In HOL Light we have the following three theorems.

```
IM_UNIT_CULLEN_INC_INJ
  |- !i z z'. i pow 2 = --Hx (&1)
              ==> (cullen_inc i z = cullen_inc i z' <=> z = z')
```

```
CULLEN_INC_SURJ
  |- !i q. q IN span{Hx(&1),i} ==> ?z. cullen_inc i z = q

IMAGE_CULLEN_INC
  |- !i. IMAGE (cullen_inc i) (:real^2) = cullen_slice i
```

- It maps the real line to itself, that is, $j_I(z) \in \mathbb{R}$ if and only if $z \in \mathbb{R}$. More precisely, it is the identity map over $\mathbb{R}$, that is, $j_I(x) = x$ for all $x \in \mathbb{R}$. In HOL Light, they are the following theorems.

```
REAL_CULLEN_INC_ALT
  |- !z i. i pow 2 = --Hx(&1)
           ==> (real (cullen_inc i z) <=> real z)

CULLEN_INC_REAL_ID
  |- !i x. cullen_inc i (Cx x) = Hx x
```

- It is a linear map, that is, for all $z, z' \in \mathbb{C}$ and $a, b \in \mathbb{R}$

$$j_I(az + bz') = aj_I(z) + bj_I(z').$$

In HOL Light, it becomes the formal theorem

```
LINEAR_CULLEN_INC
  |- !i. linear (cullen_inc i)
```

where the constant ‘`linear`‘ is defined as one expects by the following theorem.

```
linear
  |- !f. linear f <=> (!x y. f (x + y) = f x + f y) /\
                      (!c x. f (c % x) = c % f x)
```

Here, the function ‘`%:real->real^N->real^N`‘ denotes the multiplication of a vector by a scalar.

- It respects the product, in fact, for all $z, z' \in \mathbb{C}$ it holds that

$$j_I(zz') = j_I(z)j_I(z')$$

and consequently $j_I(z^n) = j_I(z)^n$ for every $n \in \mathbb{N}$. The following HOL Light theorems formalize these properties.

```
CULLEN_INC_MUL
  |- !i x y. i pow 2 = -- Hx (&1)
             ==> cullen_inc i (x * y) = cullen_inc i x * cullen_inc i y

CULLEN_INC_POW
  |- !i z n. i pow 2 = -- Hx(&1)
             ==> cullen_inc i (z pow n) = (cullen_inc i z) pow n
```

The last important property that we formalize, which will be very useful in the following, is about the decomposition of the ball $B_{\mathbb{H}}(0, R) \subseteq \mathbb{H}$ (with center $0 \in \mathbb{H}$ and radius $R > 0$) by the isomorphism j_I. Let be $z = x + y\mathbf{i} \in \mathbb{C}$ a complex number and $I \in \mathbb{S}$ then, it is clearly true that

$$\|z\| = \left\|j_I(z)\right\| \tag{4.2.2}$$

since $j_I(z) = x + yI$ and $\|I\| = 1$. From this, let be $R \in \mathbb{R}_{\geq 0}$ and $I \in \mathbb{S}$, we have immediately that $j_I(B_{\mathbb{C}}(0, R)) \subseteq B_{\mathbb{H}}(0, R) \cap L_I$. Conversely, let be $q \in B_{\mathbb{H}}(0, R) \cap L_I$ then, since j_I is surjective, $q = j_I(z)$ for some $z \in \mathbb{C}$. From equation (4.2.2), we deduce that

$$\|z\| = \|j_I(z)\| = \|q\| < R$$

hence $z \in B_{\mathbb{C}}(0, R)$. This means that $B_{\mathbb{H}}(0, R) \cap L_I \subseteq j_I(B_{\mathbb{C}}(0, R))$. Therefore, we have definitively proved that, for every $R \in \mathbb{R}_{\geq 0}$ and $I \in \mathbb{S}$,

$$B_{\mathbb{H}}(0, R) \cap L_I = j_I(B_{\mathbb{C}}(0, R)) \tag{4.2.3}$$

that is, the following HOL Light formal theorem.

```
CULLEN_INC_BALL_IN_CULLEN_SLICE
  |- !r i. i pow 2 = --Hx(&1)
          ==> ball(Hx(&0), r) INTER (cullen_slice i) =
              IMAGE (cullen_inc i) (ball(Cx(&0),r))
```

Finally, we have that, for all $R \in \mathbb{R}_{\geq 0}$,

$$B_{\mathbb{H}}(0, R) = \bigcup_{I \in \mathbb{S}} j_I(B_{\mathbb{C}}(0, R)) \tag{4.2.4}$$

with $j_I(B_{\mathbb{C}}(0, R)) \cap j_J(B_{\mathbb{C}}(0, R)) = (-R, R) \subseteq \mathbb{R}$ for every $J \in \mathbb{S}$ such that $J \neq \pm I$.

4.3 The definition of slice regular functions

Now, we can introduce the definition of Gentili and Struppa of regular functions [Gentili et al., 2013].

Definition 4.3.1 (Slice regular function). *Given a domain (i.e., an open, connected set) $\Omega \in \mathbb{H}$ a function $f\colon \Omega \to \mathbb{H}$ is* slice regular *if it is holomorphic (in the complex sense) on each slice, that is, the restricted function $f_I\colon \Omega \cap L_I \to \mathbb{H}$ has continuous partial derivatives and satisfies the condition*

$$\bar{\partial}_I f(x + yI) = \frac{1}{2}\left(\frac{\partial}{\partial x} + I\frac{\partial}{\partial y}\right) f_I(x + yI) = 0 \tag{4.3.1}$$

for each $q = x + yI$ in $\Omega \cap L_I$, for every imaginary unit $I \in \mathbb{S}$. In that case, we define the I-derivative *of f at q to be the quaternion*

$$\partial_I f(x + yI) = \frac{1}{2}\left(\frac{\partial}{\partial x} - I\frac{\partial}{\partial y}\right) f_I(x + yI). \tag{4.3.2}$$

The slice derivative *of f is the function $f' = \partial_c f\colon \Omega \to \mathbb{H}$ defined by $\partial_I f$ on $\Omega \cap L_I$, for all $I \in \mathbb{S}$.*

Note that the definition of *slice derivative* is well posed because it is applied only to regular functions. In fact, when $q = x + yI \notin \mathbb{R}$, the slice L_I containing q is unique so the *slice derivative* of f at q is uniquely determined and it coincides with the *I-derivative* $\partial_I f$ related to L_I. Conversely, if $q \in \mathbb{R}$, we have that it belongs to every slice L_I (for all $I \in \mathbb{S}$) thus, we can compute the *I-derivative* of f at q using different slices (i.e. different imaginary units) and, a priori, there is no reason why the values which one obtains should coincide. However, differentiability (and thus regularity since, by definition, regularity implies differentiability) constraints the *slice derivative* to be uniquely determined also in the real case, not depending on the slice that we use to compute it. For these reasons, in the following, we will use *I-derivative* and *slice derivative* indifferently when we talk about regular functions.

Our first goal is to code the previous definition in our formalism. One problem is the notation for partial derivatives, which is notorious for being occasionally opaque and potentially misleading. When it has to be rendered in a formal language, its translation might be tricky or at least cumbersome. This is essentially due to the fact that it is a convention that induces us to use the same name for different functions, depending on the name of the arguments.[2]

We decided that the best way to avoid potential problems in our development was to systematically replace partial derivatives with (Fréchet) derivatives. This leads to an alternative, and equivalent, definition of slice regular function which could be interesting in its own right.

The basic idea is the following. A complex function f is holomorphic in z_0 precisely when its derivative $\mathrm{D}f_{z_0}$ is $\mathbb{C}$-linear. Hence, by analogy, a quaternionic function should be slice regular if its derivative is $\mathbb{H}$-linear on slices in a suitable sense. This is indeed the case: consider $f : \Omega \to \mathbb{H}$ as before and a quaternion $q_0 \in \Omega$. Let L be a slice containing q_0 and denote by $f_{|L}$ the restriction of f to $\Omega \cap L$ then we have

Proposition 4.3.1. *The function f is slice regular in q_0 if and only if the derivative of $f_{|L}$ is right-$\mathbb{H}$-linear, that is, there exists a quaternion c such that*

$$\mathrm{D}f_{|L\,q_0}(p) = pc. \tag{4.3.3}$$

for every L that contains q_0. In that case, c is the slice derivative *$f'(q_0)$.*

Proof. Note that the hypothesis of regularity in one verse, and the existence of the derivative (as right-$\mathbb{H}$-linear function) in the other, ensure that, in both directions, f is differentiable. So, the *slice derivative* is well defined in both cases, also for $q \in \mathbb{R}$, and not depends on the slice used to compute it.

Let L be a slice such that $q_0 \in L$, then there exist two imaginary units $I, J \in \mathbb{S}$ such that $L = L_I$ and $I \perp J$. Therefore, we have that $\{1, I, J, IJ\}$ is an orthogonal basis of $\mathbb{H}$ so, the restricted function to the slice $L = L_I$ can be written as

$$f_{|L}(x + yI) = f_I(x + yI) = u(x, y) + v(x, y)I + r(x, y)J + s(x, y)IJ \tag{4.3.4}$$

and thus, omitting the dependence on x and y, the partial derivatives are

$$\frac{\partial f_I}{\partial x} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial x}I + \frac{\partial r}{\partial x}J + \frac{\partial s}{\partial x}IJ \tag{4.3.5}$$

$$I\frac{\partial f_I}{\partial y} = -\frac{\partial v}{\partial y} + \frac{\partial u}{\partial y}I - \frac{\partial s}{\partial y}J + \frac{\partial r}{\partial y}IJ. \tag{4.3.6}$$

It is easy to check, by direct computation, that f_I satisfies condition (4.3.1) if and only if the functions $F(x + yI) = u(x, y) + v(x, y)I$ and $G(x + yI) = r(x, y) + s(x, y)I$ are complex holomorphic, that is, satisfy the Cauchy-Riemann equations:

$$\frac{\partial u}{\partial x} = \frac{\partial v}{\partial y} \qquad \frac{\partial v}{\partial x} = -\frac{\partial u}{\partial y} \qquad \frac{\partial r}{\partial x} = \frac{\partial s}{\partial y} \qquad \frac{\partial s}{\partial x} = -\frac{\partial r}{\partial y} \tag{4.3.7}$$

On the other hand we have that the derivative of f_I, at a point $q_0 = x + yI \in L_I$, is expressed by the Jacobian as

$$\mathrm{D}f_{I\,q_0} = \begin{pmatrix} \frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} & 0 & 0 \\ \frac{\partial v}{\partial x} & \frac{\partial v}{\partial y} & 0 & 0 \\ \frac{\partial r}{\partial x} & \frac{\partial r}{\partial y} & 0 & 0 \\ \frac{\partial s}{\partial x} & \frac{\partial s}{\partial y} & 0 & 0 \end{pmatrix} \tag{4.3.8}$$

[2]Spivak, in his book *Calculus on manifolds* ([Spivak, 1965], p.65), notices that if $f(u, v)$ is a function and $u = g(x, y)$ and $v = h(x, y)$, then the chain rule is often written

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial u}\frac{\partial u}{\partial x} + \frac{\partial f}{\partial v}\frac{\partial v}{\partial x},$$

where f denotes two different functions on the left- and right-hand of the equation.

so, given a quaternion $p = p_0 + p_1 I \in L_I$, we have that

$$\begin{aligned}\mathrm{D}f_{I\,q_0}(p) = \left(\frac{\partial u}{\partial x}p_0 + \frac{\partial u}{\partial y}p_1\right) + \left(\frac{\partial v}{\partial x}p_0 + \frac{\partial v}{\partial y}p_1\right) I + \\ \left(\frac{\partial r}{\partial x}p_0 + \frac{\partial r}{\partial y}p_1\right) J + \left(\frac{\partial s}{\partial x}p_0 + \frac{\partial s}{\partial y}p_1\right) IJ\end{aligned} \tag{4.3.9}$$

Now, since the right multiplication of p by a quaternion $c = c_0 + c_1 I + c_2 J + c_3 IJ$ is computed by the formula

$$pc = (p_0c_0 - p_1c_1) + (c_1p_0 + p_1c_0)I + (c_2p_0 - p_1c_3)J + (p_0c_3 + p_1c_2)IJ \tag{4.3.10}$$

we have that the derivative of f_I at q_0 (4.3.9) is right-$\mathbb{H}$-linear if and only if

$$\begin{aligned}\frac{\partial u}{\partial x} = c_0 = \frac{\partial v}{\partial y} \qquad & -\frac{\partial u}{\partial y} = c_1 = \frac{\partial v}{\partial x} \\ \frac{\partial r}{\partial x} = c_2 = \frac{\partial s}{\partial y} \qquad & -\frac{\partial r}{\partial y} = c_3 = \frac{\partial s}{\partial x}\end{aligned} \tag{4.3.11}$$

that is, if and only if conditions (4.3.7) hold. In this case, the quaternion

$$c = \frac{\partial f_I}{\partial x} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial x}I + \frac{\partial r}{\partial x}J + \frac{\partial s}{\partial x}IJ$$

is the *slice derivative* $f'(q_0)$ and it easy to check that

$$c = f'(q_0) = \frac{1}{2}\left(\frac{\partial}{\partial x} - I\frac{\partial}{\partial y}\right) f_I(x + yI)$$

completing the proof. □

Proposition 4.3.1 shows that a more efficient, for our purposes, way to render these notions in a formal language is to use derivatives and vector subspaces, instead of partial derivatives and imaginary units, for primitive definitions. Therefore, we take the alternative formulation given by Proposition 4.3.1 as the definition of slice regular function in our development. The resulting formalization in HOL Light is the following.

```
let has_slice_derivative = new_definition
  `!f (f':quat) net.
      (f has_slice_derivative f') net <=>
      (!l. subspace l /\ dim l = 2 /\ Hx(&1) IN l /\ netlimit net IN l
           ==> (f has_derivative (\q. q * f')) (net within l))`;;
```

In the case of the more familiar net `at` it becomes the next theorem.

```
HAS_SLICE_DERIVATIVE_AT
  |- !f f' q0. (f has_slice_derivative f') (at q0) <=>
               (!l. subspace l /\ dim l = 2 /\ Hx(&1) IN l /\ q0 IN l
                    ==> (f has_derivative (\q. q * f')) (at q0 within l))
```

Notice that the predicate `has_slice_derivative`, as `has_derivative`, formalizes at the same time the notion of slice regular function and the notion of slice derivative. The domain Ω does not appear in the definition because functions in HOL are total and, in any case, the notion of slice derivative is local. For the same reason, we can't consider the restricted function $f_{|L}$, so we use the net operator `within` to represent its derivative $\mathrm{D}f_{|L}$. The use of HOL nets makes our formalization slightly more general than the informal definition of Proposition 4.3.1.

A special case is when the point q_0 that we are considering is real. In fact, if the function f is differentiable in $q_0 \in \mathbb{R}$, we have that it is regular if and only if the derivative of the whole function $\mathrm{D}f_{q_0}$ in q_0 is right-$\mathbb{H}$-linear. It depends essentially on the fact that a real number belongs to every slice L so, if the function is differentiable in q_0, it doesn't matter what slice we use to compute the derivative because it has to give the same result on everyone of them. Practically, we have that if f is differentiable in $q_0 \in \mathbb{R}$, then the condition

$$\mathrm{D}f_{|L\,q_0}(p) = pc \quad \text{for all } L \text{ such that } q_0 \in L$$

is equivalent to $\mathrm{D}f_{q_0}(p) = pc$. In HOL Light, this means that in the real case, considering a differentiable function, we can forget the restriction to a specific slice given by the net`within`. The formal theorem is the following.

```
HAS_SLICE_DERIVATIVE_AT_REAL
  |- !f f' q0. real q0 /\ f differentiable (at q0)
               ==> ((f has_slice_derivative f') (at q0) <=>
                    (f has_derivative (\q. q * f')) (at q0))
```

Then, we can prove formally Proposition 4.3.1 in the non-real case (i.e. with $q \in L \setminus \mathbb{R}$). Proving this theorem, which seems a mere change of notation, is a non-negligeable effort which requires a formal proof that spans more than 200 lines of code.

```
HAS_SLICE_DERIVATIVE
  |- !f f' i x y.
        i pow 2 = -- Hx(&1) /\ ~(y = &0) /\
        f differentiable at (Hx x + Hx y * i)
        ==> ((f has_slice_derivative f') (at (Hx x + Hx y * i)) <=>
             (?fx fy.
               ((\a. f(Hx(drop a) + Hx y*i)) has_vector_derivative fx)
                 (at(lift x)) /\
               ((\b. f(Hx x + Hx(drop b)*i)) has_vector_derivative fy)
                 (at(lift y)) /\
               fx + i * fy = Hx(&0) /\ f' = fx /\ f' = --(i * fy)))
```

Alternatively, we can characterize regular functions using imaginary units. Again, since we can't consider restricted functions, we have to represent f_I, for all $I \in \mathbb{S}$, as $f \circ j_I \colon \mathbb{C} \mapsto \mathbb{H}$. So, given a function `f:quat->quat`, we study the composition

`(f o cullen_inc i):complex->quat`

for every imaginary units `i:quat`. Note that, from a formal point of view, we always have to explicit the function `cullen_inc i` because the identification of L_I with $\mathbb{C}$ is only an abuse of notation that can't be done formally. This implies that L_I has to be represented as $j_I(\mathbb{C})$ and, for all $q = j_I(z) \in L_I$, the restricted function $f_I(q)$ is $f(j_I(z))$.

The isomorphism j_I is a linear function so its derivative, in every point $z \in \mathbb{C}$, is j_I itself, that is, $\mathrm{D}j_{I\,z} = j_I$. With this property and the chain rule we can compute the derivative of $f \circ j_I$ in a point $z_0 \in \mathbb{C}$ as

$$\mathrm{D}(f \circ j_I)_{z_0}(z) = (\mathrm{D}f_{j_I(z_0)} \circ \mathrm{D}j_{I\,z_0})(z) = \mathrm{D}f_{j_I(z_0)}(j_I(z)). \tag{4.3.12}$$

From this point of view, and following Proposition 4.3.1, we can prove that a function is *slice regular* and has *slice derivative* $c \in \mathbb{H}$, in a point $q_0 \in \mathbb{H}$, if and only if, for every $I \in \mathbb{S}$, it holds that

$$\mathrm{D}(f \circ j_I)_{j_I^{-1}(q_0)}(z) = \mathrm{D}f_{q_0}(j_I(z)) = j_I(z)c. \tag{4.3.13}$$

In HOL Light, we have the following formal theorem.

```
HAS_SLICE_DERIVATIVE_IFF_HAS_DERIVATIVE_CULLEN_INC
  |- !f f' q0. ((f has_slice_derivative f') (at q0) <=>
               (!i z0. i pow 2 = -- Hx(&1) /\ q0 = cullen_inc i z0
                       ==> (f o cullen_inc i has_derivative
                            (\z. cullen_inc i z * f')) (at z0)))`
```

Note that, we reformulated the statement in a way that avoids the use of the inverse j_I^{-1} writing explicitly the condition $q_0 = j_I(z_0)$. Indeed, since j_I is an isomorphism between the sets $\mathbb{C}$ and L_I but not an isomorphism of their ambient types, we prefer to don't make use of its inverse.

An useful reformulation, that allows us to compute the *slice derivative* on a fixed slice, is the following.

```
CULLEN_SLICE_DERIVATIVE_IFF_SLICE_DERIVATIVE
  |- !i f f' z0.
        i pow 2 = --Hx(&1)
        ==> ((f has_slice_derivative f')
             (at (cullen_inc i z0) within cullen_slice i) <=>
             (f o cullen_inc i has_derivative (\z. cullen_inc i z * f'))
             (at z0))
```

The further restriction to the slice `cullen_slice i` serves to fix the slice that we are considering for the calculation. Notice that, in the non-real case, it is not necessary. In fact, removing this constrain, the theorem is still true in the non-real case because, if $z_0 \in \mathbb{C} \setminus \mathbb{R}$, then $j_I(z_0) \in \mathbb{H} \setminus \mathbb{R}$ and L_I is the unique slice that contains $j_I(z_0)$. Thus, only L_I can be used to compute the derivative and the *slice derivative* of f in $q_0 = j_I(z_0)$.

However, in the real case things are different. If $z_0 \in \mathbb{R}$, then $j_I(z_0) \in \mathbb{R}$, so $j_I(z_0) \in L_J$, for all $J \in \mathbb{S}$. In one verse things are already trivial, in fact, if there exists $c \in \mathbb{H}$ such that

$$\mathrm{D}f_{J\, j_I(z_0)}(p) = pc \qquad \text{for all } J \in \mathbb{S} \tag{4.3.14}$$

then we have that

$$\mathrm{D}f_{I\, j_I(z_0)}(p) = pc. \tag{4.3.15}$$

Conversely, we have already observed that, without the hypothesis that f is differentiable, a priori, there is no reason why that equation (4.3.14) implies equation (4.3.15), so, in this case, it is important to restrict to the slice L_I.

Following the HOL Light style for multivariate analysis, we define the *slice derivative* w.r.t. a Cullen slice L_I using the Hilbert choice operator as

```
let slice_derivative = new_definition
  `slice_derivative i f q =
   @f'. (f has_slice_derivative f') (at q within cullen_slice i)`;;
```

where, again, the further restriction to L_I is necessary to be able to manage also the case in which q is real.

Moreover, it is easy to check that, for every $I \in \mathbb{S}$, the operators $\bar{\partial}_I$ and ∂_I defined by 4.3.1 and 4.3.2 commute, so we have that the *slice derivative* of every slice regular function is again a regular function in the same domain. This allows us to iterate the derivation process defining, recursively over $\mathbb{N}$, the n-th *slice derivative* $f^{(n)}$ for every $n \in \mathbb{N}$. Formally, we have the following definition.

```
let higher_slice_derivative =
 new_recursive_definition num_RECURSION
  `(!i f. higher_slice_derivative i 0 f = f) /\
   (!i f n. higher_slice_derivative i (SUC n) f =
          slice_derivative i (higher_slice_derivative i n f))`;;
```

Now, we define a predicate to express regularity over a subset of $\mathbb{H}$. For technical reasons, specifically to make the types agree, and to avoid a problematic use of j_I^{-1}, we have chosen to define the formal predicate `slice_regular_on` as follows.

```
let slice_regular_on = new_definition
  `(f slice_regular_on s) i <=>
   (!z. z IN s ==> (?f'. (f o cullen_inc i has_derivative
                          (\z. cullen_inc i z * f'))  (at z)))`;;
```

This definition has the merit to make explicit the dependence on the chosen imaginary unit I and the complex (instead of quaternionic) domain $S \subseteq \mathbb{C}$ that identifies the related domain $j_I(S) \subseteq L_I$. Letting the unit I varying on $\mathbb{S}$, we get the already introduced notion of regularity on a quaternionic domain of the form $\bigcup_{I \in \mathbb{S}} j_I(S)$. A classical example of such a domain is the ball centred in the origin with radius R that is, $B_{\mathbb{H}}(0, R) = \bigcup_{I \in \mathbb{S}} j_I(B_{\mathbb{C}}(0, R))$.

Finally, we prove the consistency of the function `slice_derivative` with the predicate `has_slice_derivative` in the following theorem.

```
HAS_SLICE_DERIVATIVE_SLICE_DERIVATIVE
 |- ! f f' z i. i pow 2 = --Hx(&1)
               ==> (f has_slice_derivative f')
                   (at (cullen_inc i z) within cullen_slice i)
                   ==> slice_derivative i f (cullen_inc i z) = f'
```

4.4 Slice derivative of algebraic expressions

After the definition of *slice regular* function, we provide a series of lemmas that allow us to compute the slice derivative of algebraic expressions. In particular, constant functions and the powers q^n are slice regular and, if $f(q)$ and $g(q)$ are slice regular functions and c is a quaternion, then $f(q) \pm g(q)$, $-f(q)$ and $f(q)c$ are slice regular. It follows that *right* polynomials (i.e., polynomials with coefficients on the right)

$$c_0 + qc_1 + q^2c_2 + \cdots + q^n c_n$$

are all slice regular functions. Most of these results are easy consequences of those discussed in Section 3.4. However, we should stress that the product of two slice regular functions $f(q)g(q)$, including left multiplication by a constant $cf(q)$ and arbitrary polynomials of the form

$$c_0 + c_{1,1}q + c_{2,0}qc_{2,1}qc_{2,2} + c_{3,0}qc_{3,1}qc_{3,2}qc_{3,3} + \cdots,$$

are not slice regular in general. The formal results are the following.

```
HAS_SLICE_DERIVATIVE_CONST
 |- !p q. ((\q. p) has_slice_derivative Hx(&0)) (at q)

HAS_SLICE_DERIVATIVE_ADD
 |- !net f g. (f has_slice_derivative f') net /\
              (g has_slice_derivative g') net
              ==> ((\q. f q + g q) has_slice_derivative f' + g') net

HAS_SLICE_DERIVATIVE_NEG
 |- !net f. (f has_slice_derivative f') net
            ==> ((\q. --f q) has_slice_derivative --f') net`,

let HAS_SLICE_DERIVATIVE_SUB
 |- !net f g. (f has_slice_derivative f') net /\
              (g has_slice_derivative g') net
              ==> ((\q. f q - g q) has_slice_derivative f' - g') net

HAS_SLICE_DERIVATIVE_RMUL
 |- !net p. ((\q. q * p) has_slice_derivative p) net
```

```
RLINEAR_HAS_SLICE_DERIVATIVE
 |- !net f f' p. (f has_slice_derivative f') net
                 ==> ((\q. f q * p) has_slice_derivative f' * p) net

HAS_SLICE_DERIVATIVE_POW
 |- !q0 n.
    ((\q. q pow n) has_slice_derivative Hx(&n) * q0 pow (n - 1)) (at q0)
```

Note that, in the last theorem, we have the usual formula for the derivative of a power as opposed to section 3.4. This depends on the fact that the quaternionic product is commutative if it is restricted to a Cullen slice.

4.5 Splitting Lemma

As we have already seen in the proof of proposition 4.3.1, there exists a deep link between slice regular functions and complex holomorphic functions. More precisely, the following *splitting lemma* links them explicitly and, at the same time, is a fundamental tool to prove several subsequent results. Given two imaginary units I and J, orthogonal to one other, we know from section 4.2 that the set $\{1, I, J, IJ\}$ is an orthonormal basis of $\mathbb{H}$. This implies that every quaternion can be *split*, in an unique way, into a sum

$$\begin{aligned} q &= a + bI + cJ + dIJ \\ &= a + bI + (c + dI)J \\ &= z + wJ \end{aligned} \tag{4.5.1}$$

with $z = a + bI, w = c + dI \in L_I$. Now, given a function $f\colon \Omega \to \mathbb{H}$, we can obviously split its restriction f_I as

$$f_I(z) = F(z) + G(z)J \tag{4.5.2}$$

with $F, G\colon \Omega \cap L_I \to L_I$. Then, the following lemma holds [Gentili and Struppa, 2006].

Lemma 4.1 (Splitting Lemma)**.** *The function f is slice regular at $q_0 \in L_I$ if and only if the functions F and G are holomorphic at q_0.*

Notice that, in the above statement, the two functions F, G are 'complex holomorphic' with respect to the implicit identification $\mathbb{C} \simeq L_I$ given by j_I. Unfortunately, from a formal point of view, this identification must be made explicit. So, using functions $F, G\colon \mathbb{C} \to \mathbb{C}$, equation (4.5.2) must be rewritten as

$$f(j_I(z)) = j_I(F(z)) + j_I(G(z))J. \tag{4.5.3}$$

Such functions can be computed explicitly using the change basis operator from $\{1, \mathbf{i}, \mathbf{j}, \mathbf{k}\}$ to $\{1, I, J, IJ\}$ combined with the projections on the complex component of a quaternion. In practice we have that

$$f_I(z) = f(j_I(z)) = j_I(Q^1_{IJ}(f(j_I(z)))) + j_I(Q^2_{IJ}(f(j_I(z))))J \tag{4.5.4}$$

with $Q^1_{IJ} = P_1 \circ M_{IJ}\colon \mathbb{H} \to \mathbb{C}$ and $Q^2_{IJ} = P_2 \circ M_{IJ}\colon \mathbb{H} \to \mathbb{C}$ where:

- M_{IJ} is the linear operator over $\mathbb{H}$ such that $M_{IJ}(1) = 1$, $M_{IJ}(\mathbf{i}) = I$, $M_{IJ}(\mathbf{j}) = J$ and $M_{IJ}(\mathbf{k}) = IJ$,
- $P_1, P_2\colon \mathbb{H} \to \mathbb{C}$ are the projections on the complex components of a quaternion, that is, given $q = z_1 + z_2\mathbf{k} \in \mathbb{H} \simeq \mathbb{C} \oplus \mathbb{C}$ we have

$$P_1(q) = z_1,\ P_2(q) = z_2 \in \mathbb{C}.$$

Essentially, Q^1_{IJ} and Q^2_{IJ} rewrite a quaternion q in the base $\{1, I, J, IJ\}$ as

$$q = x_1 + y_1 I + x_2 J + y_2 IJ$$

and then returns the complex numbers $z_1 = x_1 + y_1\mathbf{i}$ and $z_2 = x_2 + y_2\mathbf{i}$ respectively.

Since in HOL Light the set of quaternions $\mathbb{H}$ is represented as $\mathbb{R}^4$ (and $\mathbb{C}$ as $\mathbb{R}^2$), we have that M_{IJ} is the multiplication by the matrix $(1\ I\ J\ IJ)$ and, given $q = (q_0, q_1, q_2, q_3) \in \mathbb{H}$, it holds that

$$P_1(q) = \begin{pmatrix} q_0 \\ q_1 \end{pmatrix},\ P_2(q) = \begin{pmatrix} q_2 \\ q_3 \end{pmatrix} \in \mathbb{C}.$$

From this more geometrical point of view, the functions Q^1_{IJ} and Q^2_{IJ} rewrite the vector q in the basis $\{1, I, J, IJ\}$ and then return the first and the last two components respectively.

Obviously M_{IJ}, P_1 and P_2 are linear functions so, also Q^1_{IJ} and Q^2_{IJ} are linear. The formal counterpart of M_{IJ}, P_1 and P_2, respectively, are the following.

- The matrix associated to M_{IJ} is formally a vector of quaternions so, an element of type `‘:quat^4‘` (i.e. `‘:real^4^4‘`). It is formalized by the constant `‘slice_matrix‘` defined as follows.

```
let slice_matrix = new_definition
  ‘slice_matrix i j:quat^4 = vector[Hx(&1); i; j; i * j]‘;;
```

- In order to formalize the projections P_1 and P_2 we define a more general constant that, given a function $f\colon \mathbb{N} \to \mathbb{N}$ and a vector $v \in \mathbb{R}^m$, returns the vector $w \in \mathbb{R}^n$ such that ,

$$w_i = \begin{cases} v_{f(i)}, & \text{if } 1 \le f(i) \le n \\ 0, & \text{if } f(i) = 0 \text{ or } f(i) > n \end{cases} \tag{4.5.5}$$

for all $i \in \{1, \dots n\}$. The formal definition is the following.

```
let reindex = new_definition
  ‘reindex f (v:real^M):real^N =
   lambda i. if 1 <= f i /\ f i <= dimindex(:M) then v$f i else &0‘;;
```

The reindex function can be used to define two general functions

```
‘vecfst:real^M->real^N‘
‘vecsnd:real^M->real^N‘
```

that, given a m-vector (a vector in $\mathbb{R}^m$), return a n-vector by the following rules:

  - if $m = n$, they are the identity function,
  - if $m < n$, then `‘vecfst‘` fills the tail (last $n - m$ elements) of the vector with zeros whereas `‘vecsnd‘` fills the head (first $n - m$ elements) of the vector with zeros,
  - if $m > n$, then `‘vecfst‘` drops the last $m - n$ components of the vector whereas `‘vecsnd‘` drops the first $m - n$ components of the vector.

In our context, the formal functions `‘vecfst‘` and `‘vecsnd‘`, instantiated with type `‘:quat->complex‘` (that is `‘:real^4->real^2‘`), work as P_1 and P_2 in fact, the following formal theorems are proved.

```
QUAT_VECFST_COMPLEX
 |- !q. vecfst q = complex(Re q, Im1 q)

QUAT_VECSND_COMPLEX
 |- !q. vecsnd q = complex(Im2 q, Im3 q)
```

Therefore, the functions $F, G\colon \mathbb{C} \to \mathbb{C}$ in equation (4.5.2) can be written explicitly, using equation (4.5.4), as

$$F = Q^1_{IJ} \circ f \circ j_I \qquad G = Q^2_{IJ} \circ f \circ j_I \tag{4.5.6}$$

and they are formalized by the constants `quat_split1` and `quat_split2` defined as follows.

```
let quat_split1 = new_definition
    `quat_split1 i j (f:quat->quat) z:complex =
     vecfst (slice_matrix i j ** f (cullen_inc i z))`;;

let quat_split2 = new_definition
    `quat_split2 i j (f:quat->quat) z:complex =
     vecsnd (slice_matrix i j ** f (cullen_inc i z))`;;
```

Now, we are able to prove formally the Splitting Lemma.

Proof. (Splitting Lemma). Let be $q_0 = j_I(z_0) \in L_I$. Suppose that F, G are complex holomorphic functions at z_0. Thus, the derivative of F and G at z_0 is the multiplication for a complex number $F'(z_0), G'(z_0) \in \mathbb{C}$ respectively, that is,

$$\mathrm{D}F_{z_0}(z) = zF'(z_0) \qquad \mathrm{D}G_{z_0}(z) = zG'(z_0).$$

Then, using equation (4.5.3) and reminding that the right multiplication for a quaternion and j_I are linear functions[3], we compute the derivative of f_I at $j_I(z_0)$. By the above observations and the chain rule we have that, for every $q = j_I(z) \in L_I$, it holds that

$$\begin{aligned}\mathrm{D}f_{I\,q_0}(q) = \mathrm{D}(f \circ j_I)_{z_0}(z) &= j_I(\mathrm{D}F_{z_0}(z)) + j_I(\mathrm{D}G_{z_0}(z))J \\ &= j_I(zF'(z_0)) + j_I(zG'(z_0))J \\ &= j_I(z)(j_I(F'(z_0)) + j_I(G'(z_0))J)\end{aligned} \tag{4.5.7}$$

where the last equality depends on the algebraic properties of j_I. This proves that, for the proposition 4.3.1, f is regular in $q_0 = j_I(z_0) \in L_I$ and the *slice derivative* of f at q_0 is the quaternion $j_I(F'(z_0)) + j_I(G'(z_0))J$.

Conversely, suppose that f is regular at q_0. Then we have that, for proposition 4.3.1, the derivative of f_I at q_0, for all $q = j_I(z) \in L_I$, is of the form

$$\mathrm{D}f_{I\,q_0}(q) = \mathrm{D}(f \circ j_I)_{z_0}(z) = \mathrm{D}f_{j_I(z_0)}(j_I(z)) = j_I(z)c \tag{4.5.8}$$

where the quaternion c is the slice derivative. It holds that c can be uniquely splitted as $c = q_1 + q_2 J$ with $q_1, q_2 \in L_I$. This implies that there exist two complex numbers $z_1, z_2 \in \mathbb{C}$ such that

$$c = j_I(z_1) + j_I(z_2)J.$$

Again, by equation (4.5.6) and the chain rule, we compute the derivative of F

$$\begin{aligned}\mathrm{D}F_{z_0}(z) &= \mathrm{D}(Q^1_{IJ} \circ f \circ j_I)_{z_0}(z) = Q^1_{IJ}(\mathrm{D}f_{j_I(z_0)}(j_I(z)) \\ &= Q^1_{IJ}(j_I(z)c) = Q^1_{IJ}(j_I(z)(j_I(z_1) + j_I(z_2)J) \\ &= Q^1_{IJ}(j_I(zz_1) + j_I(zz_2)J) = zz_1\end{aligned} \tag{4.5.9}$$

and, analogously we obtain that $\mathrm{D}G_{z_0}(z) = zz_2$. Therefore, F, G are differentiable at z_0 and their derivatives are $\mathbb{C}$-linear. This implies that they are complex holomorphic and their complex derivatives are the complex components of the slice derivative of f_I at q_0. This concludes the proof. $\square$

We formalize the previous proof in HOL Light and we obtain the formal version of the Splitting Lemma in its existential version

[3]For every linear function h we have that $\mathrm{D}h_{q_0} = h$ for all q_0 in the domain of h.

```
QUAT_SPLITTING_LEMMA
 |- !f s i j.
     open s /\
     i pow 2 = --Hx (&1) /\
     j pow 2 = --Hx (&1) /\
     orthogonal i j
     ==> (?g h.
          (!z. f (cullen_inc i z) = cullen_inc i (g z) +
                                    cullen_inc i (h z) * j) /\
               (!g' h' z.
                 z IN s
                 ==> ((g has_complex_derivative g') (at z) /\
                      (h has_complex_derivative h') (at z) <=>
                      (f o cullen_inc i has_derivative
                       (\z. cullen_inc i z *
                            (cullen_inc i g' +
                             cullen_inc i h' * j))) (at z))) /\
                     (g holomorphic_on s /\ h holomorphic_on s
                      <=> (f slice_regular_on s) i))
```

and in its explicit form.

```
EXPLICIT_QUAT_SPLITTING_LEMMA
 |- !f s i j.
     open s /\
     i pow 2 = --Hx (&1) /\
     j pow 2 = --Hx (&1) /\
     orthogonal i j
     ==> (!z. f (cullen_inc i z) =  cullen_inc i (quat_split1 i j f z) +
                                    cullen_inc i (quat_split2 i j f z) * j) /\
         (!g' h' z.
           z IN s
           ==> ((quat_split1 i j f has_complex_derivative g') (at z) /\
                (quat_split2 i j f has_complex_derivative h') (at z) <=>
                (f o cullen_inc i has_derivative
                 (\z. cullen_inc i z * (cullen_inc i g' +
                                        cullen_inc i h' * j))) (at z))) /\
               (quat_split1 i j f holomorphic_on s /\
                quat_split2 i j f holomorphic_on s
                <=> (f slice_regular_on s) i)
```

As we have seen in the proof of the Splitting Lemma, also the slice derivative of f can be split as $f'(q_0) = j_I(F'(z_0)) + j_I(G'(z_0))J$ using the complex derivative of F and G. The process can be iterated to obtain the split of the higher slice derivatives of f using higher complex derivatives of F and G. In fact, it holds that

$$f^{(n)}(q_0) = j_I(F^{(n)}(z_0)) + j_I(G^{(n)}(z_0))J \tag{4.5.10}$$

for every $n \in \mathbb{N}$.

Equation (4.5.10) will be central in the proof of the existence of the series expansion of regular functions in a neighborhood of the origin. The corresponding HOL Light formal theorems, about the split of *slice derivatives* and *higher slice derivatives*, are the following.

```
SLICE_DERIVATIVE_SPLITTING
 |- !i j f s.
    open s /\ i pow 2 = --Hx(&1) /\ j pow 2 = --Hx(&1) /\
    orthogonal i j /\ (f slice_regular_on s) i
    ==> (!z. z IN s
             ==> slice_derivative i f (cullen_inc i z) =
                 cullen_inc i (complex_derivative (quat_split1 i j f) z) +
                 cullen_inc i (complex_derivative (quat_split2 i j f) z) * j)

HIGHER_SLICE_DERIVATIVE_SPLIT
 |- !i j f s.
    open s /\ i pow 2 = --Hx(&1) /\ j pow 2 = --Hx(&1) /\
    orthogonal i j /\ (f slice_regular_on s) i
    ==> (!n z. z IN s
               ==> higher_slice_derivative i n f (cullen_inc i z) =
                   cullen_inc i (higher_complex_derivative n
                                 (quat_split1 i j f) z) +
                   cullen_inc i (higher_complex_derivative n
                                 (quat_split2 i j f) z) * j)`,
```

Chapter 5

Power expansion of slice regular functions

In this chapter we approach power series expansion of slice regular functions at the origin, which is one of the corner stone for the development of the whole theory. First of all, we prove formally that, with the Gentili and Struppa's definition of regularity, right power series are regular functions. Secondly, we formalize the key fact that any regular function has a series expansion in an appropriate open ball centred in the origin. With these results, we completely formalize the Gentili and Struppa's seminal paper [Gentili and Struppa, 2006].
Therefore, the goal of this chapter is the formalization of the following two theorems.

Theorem 5.1 (Abel's Theorem for quaternionic power series)**.** *The quaternionic power series*

$$\sum_{n \in \mathbb{N}} q^n a_n \tag{5.0.1}$$

is absolutely convergent in the ball $B = B_{\mathbb{H}}\big(0, 1/\limsup_{n \to +\infty} \sqrt[n]{|a_n|}\big)$ *and uniformly convergent on any compact subset of* B*. Moreover, its sum defines a slice regular function on* B*.*

Theorem 5.2 (Series expansion of regular functions)**.** *Any regular function* $f \colon B_{\mathbb{H}}(0, R) \to \mathbb{H}$ *has a series expansion of the form*

$$f(q) = \sum_{n \in \mathbb{N}} q^n \frac{1}{n!} f^{(n)}(0). \tag{5.0.2}$$

Unfortunately, while the HOL Light library has a rather complete support for sequences and series in general, at the beginning of our work it was still lacking the proof of various theorems that were important prerequisites for our task. Hence, we undertake a systematic formalization of the missing theory, including

1. the definition of limit superior and inferior and their basic properties;

2. the root test for series;

3. the Cauchy-Hadamard formula for the radius of convergence.

All these preliminaries, that we present in the next sections, have been recently included in the HOL Light standard library.[1]

[1] Commit on Apr 10, 2017, HOL Light GitHub repository.

5.1 Limit superior and Limit inferior formal theory

Classically the limit superior and inferior, of a sequence of real numbers, are defined as

$$\begin{aligned} \limsup_{n\to\infty} a_n &= \lim_{n\to\infty} (\sup_{m\geq n} a_m) \\ \liminf_{n\to\infty} a_n &= \lim_{n\to\infty} (\inf_{m\geq n} a_m) \end{aligned} \tag{5.1.1}$$

or equivalently as

$$\begin{aligned} \limsup_{n\to\infty} a_n &= (\inf_{n\geq 0} \sup_{m\geq n} a_m) \\ \liminf_{n\to\infty} a_n &= (\sup_{n\geq 0} \inf_{m\geq n} a_m) \end{aligned} \tag{5.1.2}$$

and they are, obviously, closely linked to standard limit. In our formalization we choose to follow definitions (5.1.2), so we develop a little framework about the sup and inf of a set.

5.1.1 The predicates `has_sup` and `has_inf`

HOL Light provides the functions `‘sup‘` and `‘inf‘`, of type `‘:(real->bool)->real‘`, such that, given a subset $S \subseteq \mathbb{R}$, allow us to deal formally with $\sup S$ and $\inf S$. As always, functions in HOL are total, so `‘sup‘` and `‘inf‘` have to be defined also for those sets that do not admit infimum or supremum in $\mathbb{R}$. The formal definitions for `‘sup‘` and `‘inf‘` are given, using the Hilbert choice operator, in the next theorems.

```
sup
  |- !s. sup s =
         (@a. (!x. x IN s ==> x <= a) /\
              (!b. (!x. x IN s ==> x <= b) ==> a <= b))
inf
  |- !s. inf s =
         (@a. (!x. x IN s ==> a <= x) /\
              (!b. (!x. x IN s ==> b <= x) ==> b <= a))
```

In case of non-empty bounded sets, we have the following usual theorems.

```
SUP
  |- !s. ~(s = {}) /\ (?b. !x. x IN s ==> x <= b)
         ==> (!x. x IN s ==> x <= sup s) /\
             (!b. (!x. x IN s ==> x <= b) ==> sup s <= b)

INF
  |- !s. ~(s = {}) /\ (?b. !x. x IN s ==> b <= x)
         ==> (!x. x IN s ==> inf s <= x) /\
             (!b. (!x. x IN s ==> b <= x) ==> b <= inf s).
```

With this tools, following definitions (5.1.1), we could define a predicate

`‘has_limsup:(num->real)->real->(num)net->bool‘`

as follows.

```
let has_limsup = new_definition
‘((a:num->real) has_limsup l) sequentially within k <=>
 (?b. eventually (\n. a n <= b) sequentially within k) /\
 ((\n. sup {a k | k >= n}) ---> l) sequentially within k‘;;
```

However, this definition has two main problems:

1. it is specific to sequences, while we would like to have a general definition as for other kind of limits,
2. it uses the function ‘sup‘ that, as we already noticed, has a subtle semantics because it is defined also for unbounded sets.

In order to have a compositional general notion of limit superior (not specific to sequences), we will use the HOL nets in its formal definition. Moreover, to address the second issue we define two predicates (instead of functions)

```
‘has_sup:(real->bool)->real->bool‘

‘has_inf:(real->bool)->real->bool‘
```

that, given a subset ‘s:real->bool‘ of reals, code at the same time the existence and the value of the supremum (infimum), because their semantic is clearer than that of ‘sup‘ and ‘inf‘.

It is true that a set ‘s:real->bool‘ has supremum (infimum) ‘b:real‘ if and only if each of its upper (lower) bound is grater (less) or equal then ‘b‘. Therefore, formal definitions of ‘has_sup‘ and ‘has_inf‘ are the following.

```
let has_sup = new_definition
  ‘s has_sup b <=> (!c. (!x. x IN s ==> x <= c) <=> b <= c)‘;;

let has_inf = new_definition
  ‘s has_inf b <=> (!c. (!x. x IN s ==> c <= x) <=> c <= b)‘;;
```

Now, we can prove that, in case of non-empty bounded sets, our definitions are coherent with the functions ‘sup‘ and ‘inf‘. The HOL theorems are the following.

```
HAS_SUP_SUP
  |- !s l. s has_sup l <=>
          ~(s = {}) /\ (?b. !x. x IN s ==> x <= b) /\ sup s = l

HAS_INF_INF
  |- !s l. s has_inf l <=>
          ~(s = {}) /\ (?b. !x. x IN s ==> b <= x) /\ inf s = l
```

Alternatively, we can characterize the existence of the supremum (infimum) as follows. A subset of real numbers $S \subset \mathbb{R}$ has supremum (infimum), and its value is $l \in \mathbb{R}$, if and only if the following properties hold:

1. S is not empty,
2. l is an upper (lower) bound of S, that is, $x \leq l$ ($x \geq l$) for all $x \in S$,
3. for every real number $c < l$ ($c > l$) we have that there exists an element $x \in S$ such that $c < x$ ($c > x$).

The resulting formal theorems are the following.

```
HAS_SUP
  |- !s l. s has_sup l <=>
          ~(s = {}) /\
          (!x. x IN s ==> x <= l) /\
          (!c. c < l ==> (?x. x IN s /\ c < x))

HAS_INF
  |- !s l. s has_inf l <=>
          ~(s = {}) /\
          (!x. x IN s ==> l <= x) /\
          (!c. l < c ==> (?x. x IN s /\ x < c))
```

5.1.2 The predicates `has_limsup` and `has_liminf`

Now, following definitions (5.1.2) we can formalize the lim sup with the predicate

`has_limsup:(A->real)->real->(A)net->bool`

polymorphic on the domain type `:A` by the following definition.

```
let has_limsup = new_definition
  `(f:A->real has_limsup l) net <=>
   trivial_limit net \/ {b | eventually (\x. f x <= b) net} has_inf l`;;
```

The latter is more general than definition (5.1.2) because allows us to encode not only the limit superior of sequences of real numbers, but also of real-valued functions at a specific point. It depends on the type `:A` considered and on the element `net:(A)net` that specifies the limit. If the element `net:(A)net` is a trivial limit then `(f has_limsup l) net` is trivially true else, it holds when the set of real numbers such that are upper bound for `f`, from a certain point onwards over the net, has infimum `l`.
Dually, we define the predicate `has_liminf`, with the same type and in the same way of `has_limsup`, by the following definition.

```
let has_liminf = new_definition
  `(f:A->real has_liminf l) net <=>
   trivial_limit net \/ {b | eventually (\x. b <= f x ) net} has_sup l`;;
```

The notion of limit superior (and, of course, of limit inferior) is weaker than that of limit. If we have in mind a sequence of real numbers, it holds that $\lim_{n\to+\infty} a_n = l \in \mathbb{R}$ implies $\limsup_{n\to+\infty} a_n = l$ (the same holds for lim inf). This is clear because, if for every measure of closeness $\varepsilon > 0$ there exists $N \in \mathbb{N}$ such that $a_n \in (l-\varepsilon, l+\varepsilon)$ for every $n \geq N$, then it will be the same for the sequence $b_n = \sup_{m\geq n} a_m$.

In our formalization, this is a general property and it doesn't depend on the specific kind of function or limit that we are considering, so we can prove the useful following theorem.

```
REALLIM_IMP_HAS_LIMSUP
  |- !net f l. (f ---> l) net ==> (f has_limsup l) net
```

From the definition of `has_limsup` and the previous theorem `HAS_INF`, we can prove some "boundedness theorems" for those functions that admits limit superior, and then we can use them to characterize `has_limsup`.

```
HAS_LIMSUP_EVENTUALLY_UBOUND
  |- !net f l b. ~trivial_limit net /\ (f has_limsup l) net /\ l < b
                ==> eventually (\x. f x < b) net

HAS_LIMSUP_NOT_UBOUND
  |- !net f l c. ~trivial_limit net /\ (f has_limsup l) net /\ c < l
                ==> ~eventually (\x. f x <= c) net

HAS_LIMSUP
  |- !net f l. (f has_limsup l) net <=>
              trivial_limit net \/
              ((!c. l < c ==> eventually (\x. f x <= c) net) /\
               (!c. c < l ==> ~eventually (\x. f x <= c) net))
```

The last theorem `HAS_LIMSUP` states that a function $f : A \to \mathbb{R}$ admits limit superior $l \in \mathbb{R}$, in the case of a non-trivial limit, if and only if the following properties hold:

- every $c \in \mathbb{R}$, such that $l < c$, is upper bound for f from a certain point onwards,

- for every $c \in \mathbb{R}$ such that $c < l$, the set of the elements of $\operatorname{Im} f$ that is bounded superiorly by c is finite.

Another good property of limit superior is that it respects the order of $\mathbb{R}$. Let be $f, g : A \to \mathbb{R}$ two functions and $l, m \in \mathbb{R}$ their respective limits superior with respect to a non-trivial limit. If it holds that eventually $f(x) \leq g(x)$ then $l \leq m$. This is shown in the next formal theorem.

```
HAS_LIMSUP_LE
 |- !net f g l m. (f has_limsup l) net /\
                  (g has_limsup m) net /\
                  ~trivial_limit net /\
                  eventually (\x. f x <= g x) net
                  ==> l <= m
```

Finally, in the specific case of sequences a_n, we show formally, with the next HOL theorem, that our definition implies definition (5.1.1).

```
HAS_LIMSUP_SEQUENTIALLY_IMP_REALLIM_SUP
 |- !f l. (f has_limsup l) sequentially
          ==> ((\n. sup {f m | m >= n}) ---> l) sequentially
```

Moreover, if we add a boundedness condition over a_n to ensure the good definition of the supremum for every set $\{a_k \mid k \geq n\}$, we get the equivalence.

```
HAS_LIMSUP_SEQUENTIALLY_REALLIM_SUP
 |- !a l. (a has_limsup l) sequentially <=>
          (?b. !n. a n <= b) /\
          ((\n. sup {a k | k >= n}) ---> l) sequentially
```

Furthermore, we can show that, if we consider non-negative functions, i.e. such that $f(x) \geq 0$ for every $x \in D_f$, the limit superior, if it exists in the case of a non trivial limit, is necessary non-negative too.

```
HAS_LIMSUP_SEQUENTIALLY_WITHIN_LBOUND_ZERO
 |- !f b k. (f has_limsup b) (sequentially within k) /\
            (!x. &0 <= f x) /\ ~FINITE k
            ==> &0 <= b
```

Dually, we proved every theorems presented in this subsection also for `` `has_liminf` `` but we omit to show them because they aren't in the focus of this work. However, now we have the right tools to formalize the root test and finally, prove theorem 5.1.

5.1.3 Root test

The classic Cauchy's root test is a criterion to control the convergence or divergence of a series.

Theorem 5.3 (Root test.). *Let be* $\sum\limits_{n \in \mathbb{N}} a_n$ *a series and*

$$L = \limsup_{n \to +\infty} \sqrt[n]{|a_n|}$$

its limit superior (possibly infinity), then

1. *if* $L < 1$ *the series is absolutely convergent (and so convergent),*
2. *if* $L > 1$ *the series is divergent,*
3. *if* $L = 1$ *the series may be divergent, conditionally convergent, or absolutely convergent.*

Our goal is to develop a tool to check in HOL Light the convergence of a series (in particular we will be interested in power series) using the root test. In order to do that, it's enough to formalize only the sufficient condition, the one that implies the convergence, given by point (1) of theorem 5.3. Moreover, we decide to consider every subset $K \subseteq \mathbb{N}$ as the set of the indexes of a_n, in order to have the most general theorem. In this way, we are able to prove the convergence of a series also when we consider only terms with particular set of indexes (for example a_{2n} or a_{2n+1}). If the set $K \subseteq \mathbb{N}$ is finite, then the convergence of $\sum_{n \in K} a_n$ is trivial for every sequence a_n, so the following formal theorems are proved.

```
SUMMABLE_FINITE
  |- !k a. FINITE k ==> summable k a

REAL_SUMMABLE_FINITE
  |- !k a. FINITE k ==> real_summable k a
```

In the HOL Light standard library, the theorem that allows to infer convergence from absolute convergence is available only for real-valued series, so we prove the analogous for vector-valued series.

```
SERIES_ABSCONV_IMP_CONV
  |- !k a. real_summable k (\n. norm (a n)) ==> summable k a
```

Now, we can prove the part of the root test that ensure the convergence of a real-valued series.

```
REAL_SERIES_ROOT_TEST
  |- !a b k. (!n. n IN k ==> &0 <= a n) /\
             b < &1 /\
             ((\n. root n (a n)) has_limsup b) (sequentially within k)
             ==> real_summable k a
```

The formal proof is divided into two cases. First, when $K \subseteq \mathbb{N}$ is finite, it is trivial and we can prove it directly as consequence of the previous theorem `REAL_SUMMABLE_FINITE`. Second case closely follows the informal proof using the boundedness theorems for `'has_limsup'` and the theorems, available in the HOL Light standard library, about the direct comparison test and the convergence of the geometric series.

Finally, from the root test for real-valued series and the fact that absolute convergence implies convergence, we can easily obtain the root test also for vector-valued series.

```
SERIES_ROOT_TEST
  |- !a b k.
       ((\n. root n (norm (a n))) has_limsup b) (sequentially within k) /\
       b < &1
       ==> summable k a
```

5.1.4 Uniform convergence of functions series

In the last subsection we have developed some formal tools to check if a series is convergent. Now, we want to be able to check the uniform convergence of a series. In order to do that, we can use the following Weiestrass M-test.

Theorem 5.4 (Weiestrass M-test.)**.** *Let be $\{f_n\}$ a sequence of real- or vector-valued functions defined on a set E and $\{M_n\}$ a sequence of real numbers such that:*

1. *for all $n \in \mathbb{N}$ and $x \in E$, it holds that $|f_n(x)| \leq M_n$,*
2. *$\sum_{n \in \mathbb{N}} M_n$ is convergent,*

then $\sum_{n \in \mathbb{N}} f_n$ is uniformly convergent on E.

In the HOL Light standard library, the following theorem, that formalizes the Weistrass M-test, is already available.

```
SERIES_COMPARISON_UNIFORM
  |- !f g P s.
      (?l. (lift o g sums l) s) /\
      (?N. !n x. N <= n /\ n IN s /\ P x ==> norm (f x n) <= g n)
      ==> (?l. !e. &0 < e
                   ==> (?N. !n x.
                              N <= n /\ P x
                              ==> dist (vsum (s INTER (0..n)) (f x), l x) < e))
```

However, in such theorem the uniform convergence is written explicitly, so it is very difficult to read. Therefore, we define the formal predicates

```
`uniformly_convergent_on:(A->B->real^N)->(B->bool)->(A)net->bool`

`sums_uniformly_on:(num->B->real^N)->(B->bool)->(num->bool)->bool`
```

to express the uniform convergence of a sequence, or of a series of functions respectively, in order to improve readability. The formal definitions are the following.

```
let uniformly_convergent_on = new_definition
  `((f:A->B->real^N) uniformly_convergent_on (s:B->bool)) net <=>
   ?l:A->real^N.
    !e. &0 < e
        ==> eventually (\n:B. !x. x IN s ==> dist (f n x,l x) < e) net`;;

let sums_uniformly_on = new_definition
 `((f:num->B->real^N) sums_uniformly_on (s:B->bool)) k <=>
  ((\n x. vsum (k INTER (0..n)) (\i. f i x))
   uniformly_convergent_on s) (sequentially)`;;
```

Note that the predicate `` `uniformly_convergent_on` ``, that express the uniform convergence of a functions sequence, is polymorphic over two types `` `:A` `` (the type of the indexes) and `` `:B` `` (the type of the domain of the functions) and can be used with a generic `` `net:(A)net` ``. Contrarily, the notion of uniform convergence of a functions series makes sense only in case of type `` `:num` `` and net `` `sequentially` ``.

We can check that `` `sums_uniformly_on` `` is well defined in fact, if we rewrite the definition of `` `uniformly_convergent_on` `` and the theorem `` `EVENTUALLY_SEQUENTIALLY` `` in its formal definition, then we obtain the theorem

```
  |- !k f s.
      (f sums_uniformly_on s) k <=>
      (?l.
        !e. &0 < e
            ==> (?N.
                   !n. N <= n
                       ==> (!x.
                             x IN s
                             ==> dist (vsum (k INTER (0..n)) (\i. f i x),l x)
                                 < e)))
```

that is the usual definition of uniform convergence of a functions series (logically equivalent to that of the theorem `` `SERIES_COMPARISON_UNIFORM` ``).

Using these new definitions, and the theorem `SERIES_COMPARISON_UNIFORM`, we formalized the M-test, in a more readable form, in the next HOL theorem.

```
WEIESTRASS_SERIES_COMPARISON_UNIFORM
  |- !f M s k. (!x. x IN s ==> (!n. norm (f n x) <= M n)) /\
              (?l. (M real_sums l) k)
              ==> (f sums_uniformly_on s) k
```

5.2 The Abel theorem

Finally, we have everything we need to formalize theorem 5.1. The corresponding formalization is split into several theorems.

5.2.1 Convergence of a power series

First of all, as regards to the convergence we have three statements, one for each kind of convergence (pointwise, absolute and uniform). In fact, let be $\sum\limits_{n\in\mathbb{N}} q^n a_n$ a quaternionic power series and $b = \limsup\limits_{n\to+\infty} \sqrt[n]{|a_n|}$, then by the root test and Weiestrass M-test the following formal theorems are proved.

- **Absolute convergence.** It holds in the whole open ball $B(0, \frac{1}{b}) \subseteq \mathbb{H}$ and not only in its compact subsets. The resulting formalization is the following.

```
QUAT_ABSCONV_POWER_SERIES
  |- !q a k b.
      ((\n. root n (norm (a n))) has_limsup b) (sequentially within k) /\
      b * norm q < &1
      ==> real_summable k (\n. norm (q pow n * a n))
```

- **Simple convergence.** It follows easily from absolute convergence and has the same set of convergence. The corresponding HOL theorem is the next.

```
QUAT_CONV_POWER_SERIES
  |- !q a k b.
      ((\n. root n (norm (a n))) has_limsup b) (sequentially within k) /\
      b * norm q < &1
      ==> summable k (\n. q pow n * a n)
```

- **Uniform convergence.** It holds only in compact subsets of the open ball $B(0, \frac{1}{b}) \subseteq \mathbb{H}$. The following formal theorem is the related formalization.

```
QUAT_UNIFORM_CONV_POWER_SERIES
  |- !a b s k.
      ((\n. root n (norm (a n))) has_limsup b) (sequentially within k) /\
      compact s /\
      (!q. q IN s ==> b * norm q < &1)
      ==> ((\i q. q pow i * a i) sums_uniformly_on s) k
```

Note that the hypothesis `b * norm q < &1` allows a correct representation of the domain of convergence also in the case of infinite radius (case $b = 0$). Note also that we use the net `sequentially within k`, instead of `sequentially`, to encode the limit superior. The reason why we make such a choice is to have a more general formal theorems that consider also power series that have coefficients with indexes belonging to a specific subset of natural numbers. For example, let be $\{a_n\}_{n\in\mathbb{N}}$ a sequence in $\mathbb{H}$ and $K \subseteq \mathbb{N}$ a subset of natural numbers. In case that K is infinite, let be

$$\limsup_{n\in K\to+\infty} \sqrt[n]{|a_n|} \tag{5.2.1}$$

the limit obtained considering only terms a_m such that $m \in K$. We can consider the power series

$$\sum_{n \in K} q^n a_n \tag{5.2.2}$$

and we can prove, with similar arguments to those of the standard proof of theorem 5.1, that it is convergent in:

1. the whole $\mathbb{H}$ if K is finite or if K is infinite and $\limsup\limits_{n \in K \to +\infty} \sqrt[n]{|a_n|} = 0$,
2. the open ball $B(0, \frac{1}{b}) \subseteq \mathbb{H}$ if K is infinite and $\limsup\limits_{n \in K \to +\infty} \sqrt[n]{|a_n|} = b$.

In HOL Light, we always have to make explicit the set `‘k:num->bool‘` on which we want to consider power series, in fact, the convergence of (5.2.2) is expressed by the predicate `‘summable k (\n. q pow n * a n)‘`.

If we use the net `‘sequentially‘` to encode the limit superior we are making an assumption that is unnecessarily too strong since, as shown in (1.) and (2.), the weaker condition on such limit superior along K (5.2.1) is enough to infer the convergence of the power series. Limit superior (5.2.1) is naturally encoded, in the HOL Light formalism, trough the net `‘sequentially within k‘` and, in case that `‘k = (:num)‘`, we have exactly the same situation of theorem 5.1.

5.2.2 Slice derivative of a power series

In order to prove that the power series (5.0.1) defines a regular functions, we prove that its slice derivative is its formal derivative

$$\sum_{n \in \mathbb{N}^+} q^{n-1} n a_n. \tag{5.2.3}$$

Before doing this, we have to prove that power series (5.2.3) has exactly the same radious of convergence of the original power series (5.0.1).

It follows essentially from the observation that

$$\limsup_{n \to +\infty} \sqrt[n]{|n a_n|} = \limsup_{n \to +\infty} \sqrt[n]{|a_n|} \tag{5.2.4}$$

since the formal derivative (5.2.3) is also a power series with coefficients $b_n = n a_n$. At first glance, we could think that equation (5.2.4) is a consequence of a more general conservation property of limit superior under multiplication, that is,

$$\begin{cases} \limsup\limits_{n \to +\infty} a_n = l \\ \limsup\limits_{n \to +\infty} b_n = m \end{cases} \quad \text{imply that} \quad \limsup_{n \to +\infty} (a_n \cdot b_n) = l \cdot m \tag{5.2.5}$$

but, we realize quickly that it is false since we can give a counterexample. Let be $a_n = (-1)^n$ and $b_n = (-1)^{n+1}$ we have trivially that

$$\limsup_{n \to +\infty} a_n = \limsup_{n \to +\infty} b_n = 1$$

but the sequence $c_n = a_n \cdot b_n = (-1)^{2n+1}$ is the constant sequence -1 that has limit superior equal to -1 that is different from the product of the limits superior of a_n and b_n.

The right assumptions to be taken, in order to prove the implication (5.2.5), are that a_n and b_n are non-negative sequences (or more weakly non-negative from a certain point onward) and that l or m is the limit (instead of the limit superior), if it exists, of a_n or b_n. Thus, we have that, given two eventually non-negative sequence a_n and b_n, it holds that

$$\begin{cases} \lim\limits_{n \to +\infty} a_n = l \\ \limsup\limits_{n \to +\infty} b_n = m \end{cases} \quad \text{imply that} \quad \limsup_{n \to +\infty} (a_n \cdot b_n) = l \cdot m \tag{5.2.6}$$

and

$$\begin{cases} \limsup\limits_{n\to+\infty} a_n = l \\ \lim\limits_{n\to+\infty} b_n = m \end{cases} \quad \text{imply that} \quad \limsup_{n\to+\infty}(a_n \cdot b_n) = l \cdot m. \tag{5.2.7}$$

Using the latter implications (5.2.6) and (5.2.7), together with the following observations that

$$\sqrt[n]{|na_n|} = \sqrt[n]{|n|} \cdot \sqrt[n]{|a_n|} \qquad \lim_{n\to+\infty} \sqrt[n]{|n|} = 1$$

we can easily prove equality (5.2.4). In our formal setting, these results are proved in the following theorems.

```
HAS_LIMSUP_MUL_REALLIM_RIGHT
  |- `!net a b l m. (a has_limsup l) net /\ (b ---> m) net /\
                    eventually (\x:A. &0 <= a x) net /\
                    eventually (\x:A. &0 <= b x) net
                    ==> ((\x. a x * b x) has_limsup l * m) net`

HAS_LIMSUP_MUL_REALLIM_LEFT
  |- !net a b l m. (a ---> l) net /\ (b has_limsup m) net /\
                   eventually (\x. &0 <= a x) net /\
                   eventually (\x. &0 <= b x) net
                   ==> ((\x. a x * b x) has_limsup l * m) net

REALLIM_ROOT_REFL
  |- ((\n. root n (&n)) ---> &1) sequentially
```

Note that, the theorems about the limit superior of a product is true for every general net and not only for `sequentially`.

Now, we are able to prove again the same formal theorems about convergence of the previous paragraph, also for the formal derivative power series (5.2.3).

- **Absolute convergence.**

```
QUAT_ABSCONV_POWER_SERIES_DERIVATIVE
  |- !q a k b.
      ((\n. root n (norm (a n))) has_limsup b (sequentially within k) /\
      b * norm q < &1
      ==> real_summable k (\n. norm (q pow (n - 1) * Hx (&n) * a n))
```

- **Simple convergence.**

```
QUAT_CONV_POWER_SERIES_DERIVATIVE
  |- !q a k b.
      ((\n. root n (norm (a n))) has_limsup b (sequentially within k) /\
      b * norm q < &1
      ==> summable k (\n. q pow (n - 1) * Hx (&n) * a n)
```

- **Uniform convergence.**

```
QUAT_UNIFORM_CONV_POWER_SERIES_DERIVATIVE
  |- !a b s k.
      ((\n. root n (norm (a n))) has_limsup b (sequentially within k) /\
      compact s /\
      (!q. q IN s ==> b * norm q < &1)
      ==> ((\i q. q pow (i - 1) * Hx (&i) * a i) sums_uniformly_on s) k
```

Finally, from the previous results, and the fact that derivative distributes over uniformly convergent series, we prove formally, in the next theorem, that right quaternionic power series are slice regular functions on any compact subsets of their domain of convergence.

```
QUAT_HAS_SLICE_DERIVATIVE_POWER_SERIES_COMPACT
|- !a b k q0 s.
    ((\n. root n (norm (a n))) has_limsup b (sequentially within k) /\
    compact s /\ s SUBSET {q | b * norm q < &1} /\
    ~(s = {}) /\
    q0 IN s
    ==> ((\q. infsum k (\n. q pow n * a n)) has_slice_derivative
         infsum k (\n. q0 pow (n - 1) * Hx (&n) * a n))
        (at q0)
```

This completes the formalization of theorem 5.1.

5.3 Power series expansion of slice regular function

The existence of the series expansion of a regular function is an easy consequence of the *Splitting Lemma* and the theory of complex holomorphic functions. In the following we show a guideline of the proof of theorem 5.2.

Proof. (Series expansion of regular functions). Let be $R > 0$, $q \in B = B_{\mathbb{H}}(0, R)$ and $f \colon B \to \mathbb{H}$ a regular function. Then, there exist $I, J \in \mathbb{S}$ such that $q \in L_I$ and $I \perp J$. Note that $q = j_I(z)$ for some $z \in B_{\mathbb{C}}(0, R)$.

For the *Splitting Lemma*, there exist two complex holomorphic functions $F, G \colon B_{\mathbb{C}}(0, R) \to \mathbb{C}$ such that

$$f(q) = f(j_I(z)) = j_I(F(z)) + j_I(G(z))J.$$

From the theory of complex holomorphic functions, we have that F, G are analytic in $B_{\mathbb{C}}(0, R)$, so they can be written as power series in the following way.

$$F(z) = \sum_{n\in\mathbb{N}} z^n \frac{1}{n!} F^{(n)}(0) \qquad G(z) = \sum_{n\in\mathbb{N}} z^n \frac{1}{n!} G^{(n)}(0)$$

The last two equalities imply that

$$f(q) = f(j_I(z)) = j_I(\sum_{n\in\mathbb{N}} z^n \frac{1}{n!} F^{(n)}(0)) + j_I(\sum_{n\in\mathbb{N}} z^n \frac{1}{n!} G^{(n)}(0))J$$

and, by the algebraic properties of j_I[2], this becomes

$$f(q) = f(j_I(z)) = \sum_{n\in\mathbb{N}} j_I(z)^n \frac{1}{n!} (j_I(F^{(n)}(0)) + j_I(G^{(n)}(0))J).$$

Then, from equation (4.5.10) it holds that

$$f^{(n)}(0) = f^{(n)}(j_I(0)) = j_I(F^{(n)}(0)) + j_I(G^{(n)}(0))J$$

so we obtain the desired equality

$$f(q) = f(j_I(z)) = \sum_{n\in\mathbb{N}} j_I(z)^n \frac{1}{n!} f^{(n)}(0) = \sum_{n\in\mathbb{N}} q^n \frac{1}{n!} f^{(n)}(0)$$

completing the proof. □

[2] j_I is the identity function on the real line hence $j_I(\frac{1}{n!}) = \frac{1}{n!}$ and $j_I(0) = 0$

The corresponding HOL Light formal theorem is the following.

```
SLICE_REGULAR_SERIES_EXPANTION
 |- !r q f.
     &0 < r /\ q IN ball(Hx(&0), r) /\
     (!i. (f slice_regular_on ball(Cx(&0), r)) i)
     ==> ?z i. i pow 2 = --Hx(&1) /\
               q = cullen_inc i z /\
               f q = infsum (:num)
                      (\n. (cullen_inc i z) pow n * inv (Hx (&(FACT n))) *
                           higher_slice_derivative i n f (Hx(&0)) )
```

As mentioned in section 4.3, because of the type of '`slice_regular_on`', the hypothesis about regularity of f in $B_{\mathbb{H}}(0, R)$ is represented by the equivalent assertion that f is regular in $j_I(B_{\mathbb{C}}(0, R))$ for every $I \in \mathbb{S}$ in fact, it holds that $\bigcup_{I \in \mathbb{S}} j_I(B_{\mathbb{C}}(0, R)) = B_{\mathbb{H}}(0, R)$.

Chapter 6

Pythagorean-Hodograph curves

6.1 PH-curves and the first-order Hermite interpolation problem

The *hodograph* of a parametric curve $\mathbf{r}(t)$ in $\mathbb{R}^n$ is just its derivative $\mathbf{r}'(t)$, regarded as a parametric curve in its own right. A parametric polynomial curve $\mathbf{r}(t)$ is said to be a *Pythagorean-Hodograph* (PH) curve if it satisfies the *Pythagorean condition*, i.e., there exists a polynomial $\sigma(t)$ such that

$$\left\|\mathbf{r}'(t)\right\|^2 = x_1'^2(t) + \cdots + x_n'^2(t) = \sigma^2(t) \tag{6.1.1}$$

that is, the parametric speed $\left\|\mathbf{r}'(t)\right\|$ is polynomial. In general, for a polynomial curve $\mathbf{r}(t)$, the irrational nature of $\left\|\mathbf{r}'(t)\right\|$ has unfortunate computational implications:

- arc length must be computed approximately by numerical quadrature,
- unit tangent $\mathbf{t}$, normal $\mathbf{n}$, curvature k, etc, are not rational functions of t,
- offset curve, i.e. of the form $\mathbf{r}_d(t) = \mathbf{r}(t) + d\mathbf{n}(t)$, at distance d must be approximated,
- approximate real-time CNC (Computer Numerical Control) interpolator algorithms, for motion along $\mathbf{r}(t)$ with given speed (feedrate) $V = \frac{ds}{dt}$, are required.

However, in the case of a PH-curve $\mathbf{r}(t)$, we achieve some advantages:

- rational offset curves $\mathbf{r}_d(t) = \mathbf{r}(t) + d\mathbf{n}(t)$,
- polynomial arc-length function $s(t) = \int_0^t \left\|\mathbf{r}'(\tau)\right\| d\tau$ that permits an exact arch-length computation,
- closed-form evaluation of energy integral,
- real–time CNC interpolators, rotation-minimizing frames.

For these reasons, Pythagorean-Hodograph curves, introduced by Farouki and Sakkalis in 1990, are used for computer-aided design (CAD), digital motion control, path planning, robotics applications and animation. Farouki's book [Farouki, 2009] offers a fairly complete and self-contained exposition of this theory.

Therefore, since the practical relevance of PH-curve cited above, their computer formalization, as for quaternions, can be useful, or even essential, for a wide class of applications in formal methods. In particular, we are interested about one basic problem, with many obvious practical applications, that is whether there exists a PH-curve with prescribed conditions on its endpoints.

Problem 6.1 (First-order Hermite Interpolation Problem). *Given the initial and final point $\{\mathbf{P}_i, \mathbf{P}_f\}$ and derivatives $\{\mathbf{d}_i, \mathbf{d}_f\}$, find a PH interpolation for this data set.*

It turns out that planar ($\mathbf{r}(t) \in \mathbb{R}^2$) and spatial ($\mathbf{r}(t) \in \mathbb{R}^3$) Pythagorean–Hodograph curves are characterized by different approaches since Pythagorean polynomial triples and quadruples involve disparate algebraic structures. A convenient algebraic model for planar PH-curves is based on the properties of the complex numbers while, spatial PH-curves can be succinctly and profitably described by quaternions [Farouki, 2009]. In this work, we deal with spatial PH-curves as a natural application of our formalization of quaternionic algebra.

We recall that, in the HOL Light style, a curve $\mathbf{r}\colon \mathbb{R} \to \mathbb{R}^n$ is represented by an element `` `r:real^1->real^N` ``. Therefore, with this specification in mind, the formal definition of a generic PH-curve is straightforward

```
let pythagorean_hodograph = new_definition
 `pythagorean_hodograph r <=>
  vector_polynomial_function r /\
  real_polynomial_function (\t. norm (vector_derivative r (at t)))`;;
```

where the predicate `` `real_polynomial_function` `` is the HOL Light formal characterization of real polynomial functions, while `` `vector_polynomial_function r` `` means that the curve $\mathbf{r}(t) \in \mathbb{R}^n$ is polynomial componentwise.

In the following, we will focus on spatial PH-curves, we will characterize them using quaternions and, finally, we will certify the solutions (of lowest degree) of the problem 6.1.

However, before doing this, we have to prove a series of formal theorems, about *Bernstein polynomials* and *real polynomial functions*, that we need to deal with PH-curves expressed as *Bézier curves*, that is, the canonical style in which curves are represented in this field of research. The main motivation is that curves, expressed in this style, can be easily displayed and graphically manipulated as we show in the next section.

6.2 Bézier curves and Bernstein basis polynomials

6.2.1 Bézier Curves

A Bézier curve of degree n, with control points (or Bézier points) $\mathbf{P}_0, \mathbf{P}_1, \ldots, \mathbf{P}_n \in \mathbb{R}^m$, is a parametric polynomial curve defined by the formula

$$\mathbf{B}(t) = \sum_{k=0}^{n} b_k^n(t)\mathbf{P}_k \qquad t \in [0,1] \tag{6.2.1}$$

where $b_k^n(t) = \binom{n}{k}(1-t)^{n-k}t^k$ are the Bernstein basis polynomials.

The polygon formed by connecting the Bézier points with lines, starting with $\mathbf{P}_0$ and finishing with $\mathbf{P}_n$, is called the Bézier polygon (or control polygon). The convex hull of the Bézier polygon contains the Bézier curve. For example, for $n = 3$ we obtain the cubic

$$\mathbf{B}(t) = \mathbf{P}_0(1-t)^3 + 3\mathbf{P}_1 t(1-t)^2 + 3\mathbf{P}_2 t^2(1-t) + \mathbf{P}_3 t^3 \qquad t \in [0,1] \tag{6.2.2}$$

as shown in figure 6.1.

The main properties of a Bézier curve are:

- the curve begins at $\mathbf{P}_0$ and ends at $\mathbf{P}_n$, this is the so-called endpoint interpolation property,
- the curve is a straight line if and only if all the control points are collinear,
- the start and the end of the curve is tangent to the first and last section of the Bézier polygon, respectively,

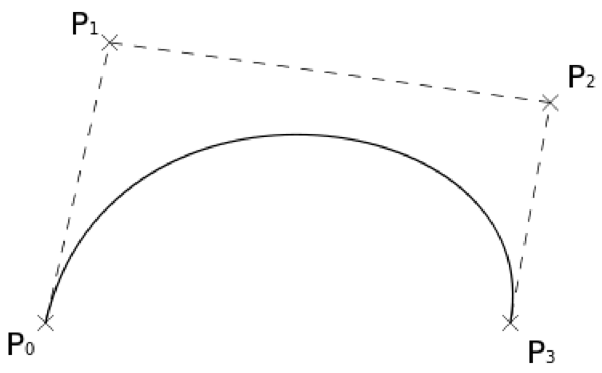


Figure 6.1: Cubic Bézier curve with control points $\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3 \in \mathbb{R}^2$.

- there is a numerically stable method, the de Casteljau's algorithm, to evaluate Bézier curves.

Bézier curves are frequently used in computer graphics and related fields as, for example, animation and robotics, to model smooth curves. Since the curve is completely contained in the convex hull of its control points, they can be graphically displayed and used to manipulate the curve intuitively. Geometric transformations such as translation, homothety and rotation can be applied to the curve by applying the respective transformations to its control points.

Bézier curves are also used in the solution of the problem 6.1 founded by Farouki, Giannelli et al. in [Farouki et al., 2008]. In this context, such a problem is reduced to find the appropriate control points such that the Bézier curves related on them is a PH-curve. For example, if we want a cubic interpolant, we have to find, if they exist, four points $\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3$ such that the Bézier cubic (6.2.2) is PH and interpolates the initial points $\{\mathbf{P}_i, \mathbf{P}_f\}$ and derivatives $\{\mathbf{d}_i, \mathbf{d}_f\}$.

6.2.2 Bernstein polynomial basis

The algebraic and analytic properties of Bézier curves depend on the properties of the Bernstein polynomial basis. As said before, given $n \in \mathbb{N}$ and $i \leq n$, the Bernstein basis polynomials (or simply Bernstein basis) on $[0, 1]$ are defined by

$$b_k^n(t) = \binom{n}{k}(1-t)^{n-k}t^k \tag{6.2.3}$$

and a polynomial of degree at most n in the Bernstein form is written as

$$P(t) = \sum_{k=0}^{n} c_k b_k^n(t)$$

with $c_k \in \mathbb{R}$.
However, the Bernstein basis can be calculated inductively on k and n. More precisely, from definition (6.2.3) it follows that, for all k,$n \in \mathbb{N}$, the following properties hold.

$$\begin{aligned} b_0^0(t) &\equiv 1 \\ b_k^0(t) &\equiv 0 \\ b_0^{n+1} &= (1-t)^{n+1} \\ b_{k+1}^{n+1}(t) &= tb_k^n(t) + (1-t)b_{k+1}^n(t) \end{aligned} \tag{6.2.4}$$

The HOL Light standard library provides definition (6.2.3) with the following theorem.

```
bernstein
  |- !x n k.
       bernstein n k x = &(binom (n,k)) * x pow k * (&1 - x) pow (n - k)
```

Therefore, we prove formally equations (6.2.4).

```
BERNSTEIN_RECUR
  |- (!t. bernstein 0 0 t = &1) /\
     (!n t. bernstein (SUC n) 0 t = (&1 - t) pow (SUC n)) /\
     (!k t. bernstein 0 (SUC k) t = &0) /\
     (!n k t. bernstein (SUC n) (SUC k) t =
              t * bernstein n k t + (&1 - t) * bernstein n (SUC k) t)
```

In our formal setting, the inductive representation given by equations (6.2.4) is often more useful then that given by definition (6.2.3) because the inductive structure simplifies the proof of some easier properties. First of all, we compute $b_k^n(t)$ in $t = 0$ and $t = 1$ (it will be useful to evaluate a Bézier curve in its endpoints) giving the following theorems.

```
BERNSTEIN_0
  |- !k n. bernstein n k (&0) = if k = 0 then &1 else &0

BERNSTEIN_1 = prove
  |- !k n. bernstein n k (&1) = if k = n then &1 else &0
```

Secondly, we prove that $b_k^n(t) \equiv 0$ if $n < k$.

```
BERNSTEIN_EQ_ZERO
  |- !n k t. n < k ==> bernstein n k t = &0
```

Also the derivative of the Bernstein basis polynomials has an inductive representation. From equations (6.2.4) we can easily check, by direct computation, that, for all k,$n \in \mathbb{N}$ the following properties hold.

$$\begin{aligned}
&\frac{d}{dt}b_0^0(t) \equiv 0\\
&\frac{d}{dt}b_k^0(t) \equiv 0\\
&\frac{d}{dt}b_0^{n+1}(t) = -(n+1)b_0^n(t)\\
&\frac{d}{dt}b_{k+1}^{n+1}(t) = (n+1)(b_k^n(t) - b_{k+1}^n(t))
\end{aligned} \tag{6.2.5}$$

The corresponding formalization is the next HOL Light theorem.

```
HAS_REAL_DERIVATIVE_BERNSTEIN
  |- (!t. (bernstein 0 0 has_real_derivative &0) (atreal t)) /\
     (!t n. (bernstein (SUC n) 0 has_real_derivative
             (-- &(SUC n) * bernstein n 0 t))
            (atreal t))/\
     (!t k. (bernstein 0 (SUC k) has_real_derivative &0) (atreal t)) /\
     (!t k n. (bernstein (SUC n) (SUC k) has_real_derivative
               (&(SUC n) * (bernstein n k t - bernstein n (SUC k) t)))
             (atreal t))
```

From the latter theorem `HAS_REAL_DERIVATIVE_BERNSTEIN`, the same result about the function ‘`real_derivative`‘ easily follows.

```
BERNSTEIN_REAL_DERIVATIVE
  |- (!t. real_derivative (bernstein 0 0) t = &0) /\
     (!t n. real_derivative (bernstein (SUC n) 0) t =
            (-- &(SUC n) * bernstein n 0 t))/\
     (!t k. real_derivative (bernstein 0 (SUC k)) t = &0) /\
     (!t k n. real_derivative (bernstein (SUC n) (SUC k)) t =
              &(SUC n) * (bernstein n k t - bernstein n (SUC k) t)
```

In the following sections we will certify the solution of the first-order Hermite interpolation problem by a PH cubic or quintic (Farouki, Giannelli et al. [Farouki et al., 2008]), where all the curve will be expressed in the Bernstein form.

But, before this, we prove in the next section, some missing theorems about polynomial functions and their derivatives.

6.3 Formal polynomial functions and derivatives

HOL Light provides three constants to deal with formal real polynomial functions.

- The constant ‘`polynomial_function:(real->real)->bool`‘ characterizes real polynomial functions of one variable ($f\colon \mathbb{R} \to \mathbb{R}$), that is, functions defined as $f(x) = \sum_{i=0}^{n} a_i x^i$.
- The constant ‘`real_polynomial_function:(real^N->real)->bool`‘ characterizes polynomial functions of several variables ($f\colon \mathbb{R}^n \to \mathbb{R}$) and it is defined inductively by the following rules:
  - $x \to x_i$ is polynomial, for all $i \in \{1, \dots, n\}$,
  - $x \to c$ is polynomial, for all $c \in \mathbb{R}$,
  - if f and g are polynomial functions then $x \to f(x) + g(x)$ is polynomial
  - if f and g are polynomial functions then $x \to f(x)g(x)$ is polynomial.
- The constant ‘`vector_polynomial_function:(real^1->real^N)->bool`‘ is used to represent formally polynomial curves, that is, functions $t \mapsto \mathbf{r}(t) = (r_1(t), \dots, r_n(t)) \in \mathbb{R}^n$ that are polynomial componentwise (i.e. $t \mapsto r_i(t)$ is polynomial, for all $i \in \{1 \dots n\}$).

It’s clear that ‘`real_polynomial_function`‘ is more general then ‘`polynomial_function`‘ because allows to consider polynomial functions in several variables. However, when we consider functions $f\colon \mathbb{R} \to \mathbb{R}$, they are equivalent up to the use of ‘`lift`‘ or ‘`drop`‘ to makes the types agree. The formal HOL Light statement is the following.

```
REAL_POLYNOMIAL_FUNCTION_IFF_POLYNOMIAL_FUNCTION_1
  |- !f. real_polynomial_function f <=>
         polynomial_function (f o lift)‘,
```

In the following, we will often prove, formally, that the Bézier curve (6.2.1) is a polynomial function, for every n-upla of control points $\mathbf{P}_0, \dots, \mathbf{P}_n \in \mathbb{R}^m$. In order to do that, we have to prove formally both various support theorems about vector polynomial functions, as, for instance,

```
VECTOR_POLYNOMIAL_FUNCTION_ADD
  |- !f g. vector_polynomial_function f /\
           vector_polynomial_function g
           ==> vector_polynomial_function (\x. f x + g x));
```

and that Bernstein basis polynomials are actually polynomial functions.

```
POLYNOMIAL_FUNCTION_BERNSTEIN
  |- !n k. polynomial_function (\t. bernstein n k t)
```

Moreover, we prove a theorem that makes coherent the languages in case of a quaternion polynomial $A(t)$ (a function $A\colon \mathbb{R} \to \mathbb{H}$ such that $A_i\colon \mathbb{R} \to \mathbb{R}$ is polynomial, for all $i = 0 \ldots 3$) since the real functions $t \to A_i(t)$, for $i = 0 \ldots 3$, can be expressed by the quaternionic projections.

```
REAL_POLYNOMIAL_FUNCTIONS_QUAT_COMPONENTS
  |- !A. vector_polynomial_function A <=>
         real_polynomial_function (Re o A) /\
         real_polynomial_function (Im1 o A) /\
         real_polynomial_function (Im2 o A) /\
         real_polynomial_function (Im3 o A)
```

As regards derivative of a polynomial function, it is well know that it is also polynomial. More precisely, we have that $f\colon \mathbb{R} \to \mathbb{R}$ is a polynomial function if and only if its derivative is a polynomial function. This simple theorem was missing in the HOL Light standard library, so we have proved it.

```
HAS_REAL_DERIVATIVE_POLYNOMIAL_FUNCTION_IFF
  |- !f f'. (!t. (f has_real_derivative f' t) (atreal t))
            ==> (polynomial_function f' <=> polynomial_function f)
```

Moreover, an analogous result is formalized in case of a curve $\mathbf{r}\colon \mathbb{R} \to \mathbb{R}^n$.

```
HAS_VECTOR_DERIVATIVE_VECTOR_POLYNOMIAL_FUNCTION_IFF
  |- !r r'. (!t. (r has_vector_derivative r' t) (at t))
           ==> (vector_polynomial_function r' <=>
                vector_polynomial_function r)
```

The formal theorems above are necessary to prove, in HOL Light, the quaternionic representation of a PH-curve that we will present in the next section.

6.4 Quaternionic representation of spatial PH-curves

Following the style and the notations used by Farouki in his book [Farouki, 2009], we recall that, regarding $\mathbf{i}$, $\mathbf{j}$, $\mathbf{k}$ as unit vectors in spatial Cartesian coordinates, we may consider a quaternion $A = a_0 + a_1\mathbf{i} + a_2\mathbf{j} + a_3\mathbf{k}$ as comprising "scalar" and "vector" parts $a_0 = scal(A)$ and $\mathbf{a} = a_1\mathbf{i} + a_2\mathbf{j} + a_3\mathbf{k} = vect(A)$. In this setting, we can use different notations as

$$A = scal(A) + vect(A) = a_0 + \mathbf{a} = (a_0, \mathbf{a}). \tag{6.4.1}$$

All real numbers, and three-dimensional vectors, are subsumed as "pure scalar" and "pure vector" quaternions of the form $(a_0, \mathbf{0})$ and $(0, \mathbf{a})$ respectively. For brevity, we denote such quaternions by simply a_0 and $\mathbf{a}$.

Also the sum and product can be rewritten in these notations as follows.

$$\begin{aligned} A + B &= (a_0 + b_0, \mathbf{a} + \mathbf{b}) \\ AB &= (ab - \mathbf{a} \cdot \mathbf{b}, a\mathbf{b} + b\mathbf{a} + \mathbf{a} \times \mathbf{b}) \end{aligned} \tag{6.4.2}$$

In our formalization, the scalar and the vector part of a quaternion are `Re a` and `HIm a` while, "pure scalar" and "pure vector" quaternions are represented by `Hx a0` and `Hv a` respectively. Therefore, equations 6.4.1 and 6.4.2 are formalized in the following HOL theorems.

```
QUAT_HX_HIM_SPLIT
  |- !q. q = Hx (Re q) + Hv (HIm q)

QUAT_ADD_ALT
  |- !p q. p + q = Hx(Re p + Re q) + Hv(HIm p + HIm q)
```

```
QUAT_MUL_ALT
  |- !p q. p * q = Hx (Re p * Re q - HIm p dot HIm q) +
           Hv (Re p % HIm q + Re q % HIm p + HIm p cross HIm q)
```

If $\|A\| = 1$ then A is called a unit quaternion. Moreover, every unit quaternion is necessarily of the form $A = (\cos \frac{1}{2}\theta, \sin \frac{1}{2}\theta\mathbf{u})$, for some angle θ and unit pure vector $\mathbf{u} \in \mathbb{R}^3$. We recall also that a "pure vector" quaternion $\mathbf{u}$ is unitary (i.e. $\|\mathbf{u}\| = 1$) if and only if its square is equal to minus 1, that is, $\mathbf{u}^2 = -1$. Therefore the set of unitary "pure vector" coincides with the set $\mathbb{S} = \{q \in \mathbb{H} \mid q^2 = -1\}$.

It is well known that Pythagorean quadruples of relatively primes polynomials can be characterized by the following theorem [Farouki, 2009].

Theorem 6.1 (Polynomial Pythagorean quadruples)**.** *Let be $a(t), b(t), c(t), d(t)$ four relatively prime real polynomials, then they satisfy the Pythagorean condition*

$$a(t)^2 + b(t)^2 + c(t)^2 = d(t)^2 \tag{6.4.3}$$

if an only if they are expressible in terms of other real polynomials $u(t), v(t), p(t), q(t)$ in the form

$$\begin{aligned} a(t) &= u(t)^2 + v(t)^2 - p(t)^2 - q(t)^2 \\ b(t) &= 2[u(t)q(t) + v(t)p(t))] \\ c(t) &= 2[v(t)q(t) - u(t)p(t)] \\ d(t) &= u(t)^2 + v(t)^2 + p(t)^2 + q(t)^2. \end{aligned} \tag{6.4.4}$$

This implies that $\mathbf{r}(t) = (x(t), y(t), z(t)) \in \mathbb{R}^3$ is a PH-curve if and only if there exist four real polynomials such that

$$\begin{aligned} x'(t) &= u(t)^2 + v(t)^2 - p(t)^2 - q(t)^2 \\ y'(t) &= 2[u(t)q(t) + v(t)p(t))] \\ z'(t) &= 2[v(t)q(t) - u(t)p(t)]. \end{aligned} \tag{6.4.5}$$

In these cases, the norm of $\mathbf{r}'(t)$ is the polynomial $\sigma(t) = u(t)^2 + v(t)^2 + p(t)^2 + q(t)^2$.

It turns out [Farouki, 2009] that this is equivalent to the requirement that the hodograph $\mathbf{r}'(t)$ can be expressed as a quaternion product of the form

$$\begin{aligned} \mathbf{r}'(t) = A(t)\mathbf{i}\bar{A}(t) =& (u(t)^2 + v(t)^2 - p(t)^2 - q(t)^2)\mathbf{i} + \\ & 2[u(t)q(t) + v(t)p(t))]\mathbf{j} + \\ & 2[v(t)q(t) - u(t)p(t)]\mathbf{k} \end{aligned} \tag{6.4.6}$$

where $A(t) = u(t) + v(t)\mathbf{i} + p(t)\mathbf{j} + q(t)\mathbf{k}$ is a quaternion polynomial (i.e. a function $A\colon \mathbb{R} \to \mathbb{H}$ such that $A_i\colon \mathbb{R} \to \mathbb{R}$ is polynomial, for all $i = 0 \ldots 3$) and $\bar{A}(t) = u(t) - v(t)\mathbf{i} - p(t)\mathbf{j} - q(t)\mathbf{k}$ is its conjugate. Equation (6.4.6) can be checked by direct computation and, moreover, it holds that the choice of $\mathbf{i}$ as *reference vector* is merely conventional. It is in fact a general properties that a quaternionic product of the form $A\mathbf{u}\bar{A}$ represents a "pure vector" for every $A \in \mathbb{H}$ and $\mathbf{u} \in \mathbb{S}$. The latter property can be reformulated stating that, for every $\mathbf{u} \in \mathbb{H}$ such that $\mathbf{u}^2 = -1$, the real part $\mathrm{Re}(A\mathbf{u}\bar{A})$ is equal to zero. This is proved formally by direct computation, producing the following HOL theorem.

```
QUAT_RE_ROTATION
  |- !A u. u pow 2 = -- Hx(&1)
           ==> Re (A * u * cnj A) = &0',
```

Finally, we have the following proposition.

Proposition 6.4.1 (PH-curves)**.** *A curve $\mathbf{r}(t)$ is PH if and only if its hodograph can be expressed on the form*

$$\mathbf{r}'(t) = A(t)\mathbf{u}\bar{A}(t) \tag{6.4.7}$$

for some quaternion polynomial $A(t)$ and some unitary "pure vector" $\mathbf{u} \in \mathbb{S}$.

The function $A(t)$ is polynomial, so the same holds for $A(t)\mathbf{u}\bar{A}(t)$. Moreover, their components are elements of a Pythagorean quadruples, hence $\|\mathbf{r}'(t)\| = \|A(t)\mathbf{u}\bar{A}(t)\|$ is a real polynomial function. Finally, since $\mathbf{r}'(t) = A(t)\mathbf{u}\bar{A}(t)$ is a vector polynomial function, it is obvious that also $\mathbf{r}(t)$ is a vector polynomial function hence, the curve $\mathbf{r}(t)$ is PH. This considerations are formalized in the following formal statement.

```
QUAT_PH_CURVE
  |- !r A u. u pow 2 = -- Hx(&1) /\
             vector_polynomial_function A /\
             (!t. (r has_vector_derivative (A t * u * cnj (A t))) (at t))
             ==> pythagorean_hodograph r
```

The latter theorem formalizes only one verse of proposition 6.4.1 but, it provides a sufficient condition to show that a given spatial curve is PH. The other implication is proved using the properties of polynomials as syntactic objects (for example polynomial division with quotient and remainder) but such a theory is still lacking in HOL Light standard library and its development was out of the goal of this work. However, it could be interesting for further developments because, it would allow us to prove formally both the two implications of proposition 6.4.1 in order to use quaternions to characterize completely spatial PH-curves.

In this setting, the first-order Hermite interpolation problem can be reduced to find a quaternion polynomial, in the Bernstein form $A(t) = \sum_{k=0}^{n} A_k b_k^n(t)$ with $A_k \in \mathbb{H}$, such that the curve $\mathbf{r}(t)$ obtained by integrating (6.4.7) satisfies the following conditions

$$\begin{aligned}
\mathbf{r}(0) &= \mathbf{P}_0 \\
\mathbf{r}(1) &= \mathbf{P}_1 \\
\mathbf{r}'(0) &= A(0)\mathbf{u}\bar{A}(0) = \mathbf{d}_0 \\
\mathbf{r}'(1) &= A(1)\mathbf{u}\bar{A}(1) = \mathbf{d}_1.
\end{aligned} \tag{6.4.8}$$

From proposition 6.4.7, we have immediately that the degree of a PH-curve is odd and it depends on the degree of $A(t)$. More precisely, if $deg(A(t)) = n$, then the degree of $\mathbf{r}(t)$ is $m = 2n+1$. This implies that PH-curves of lowest degree, that solve the first-order Hermite interpolation problem, if they exist, are cubic or quintic. They are related on the choice of $A(t)$ of the form (linear)

$$A(t) = A_0(1-t) + A_1 t \tag{6.4.9}$$

or of the form (quadratic)

$$A(t) = A_0(1-t)^2 + A_1 2t(1-t) + A_2 t^2 \tag{6.4.10}$$

for the cubic and quintic case respectively.

6.5 PH cubic and PH quintic interpolants

6.5.1 Solutions of the equation $A\mathbf{u}\bar{A} = \mathbf{d}$

From equations (6.4.8), we have that the problem of Hermite interpolation by spatial PH-curves requires the quaternionic solutions of equations of the form

$$A\mathbf{u}\bar{A} = \mathbf{d} \tag{6.5.1}$$

where $\mathbf{u} \in \mathbb{S}$ and $\mathbf{d} = d_x\mathbf{i} + d_y\mathbf{j} + d_z\mathbf{k}$ is a given vector. This equation defines a mapping of the unit vector $\mathbf{u}$ to a general vector $\mathbf{d} \in \mathbb{R}^3$, through a spatial rotation and a scaling by the factor $\|\mathbf{d}\| = \|A\|^2$.

The quaternionic solutions of (6.5.1) comprise a one-parameter family [Farouki et al., 2002], which can be conveniently described in terms of the unit vectors

$$\delta = \frac{\mathbf{d}}{\|\mathbf{d}\|} \qquad \mathbf{n} = \frac{\mathbf{u}+\delta}{\|\mathbf{u}+\delta\|} \tag{6.5.2}$$

as

$$A = \sqrt{\|\mathbf{d}\|}\mathbf{n}(\cos\phi + \sin\phi\mathbf{u}) \tag{6.5.3}$$

where ϕ is a free angular variable. We remark that solution (6.5.3) is well defined only in the case that $\mathbf{d}$ is not aligned with $-\mathbf{u}$. The free angular variable ϕ is due to the fact that

$$Q\mathbf{u}\bar{Q} = \mathbf{u} \tag{6.5.4}$$

for every quaternion $Q = \cos\phi + \sin\phi\mathbf{u}$, with $\phi \in \mathbb{R}$. In general, the map $\mathbf{u} \mapsto Q\mathbf{u}\bar{Q}$ defines the spatial rotation of angle ϕ and rotation axis the span of $\mathbf{u}$. The formalization of equation (6.5.4) is the following.

```
QUAT_IMG_UNIT_SIN_COS
  |- !i t. i pow 2 = --Hx (&1)
          ==> (Hx (cos t) + Hx (sin t) * i) * i *
              cnj (Hx (cos t) + Hx (sin t) * i) = i
```

Now, we formally certify that quaternions of the form (6.5.3) are solution of equation (6.5.1). First of all, we prove that $A = \mathbf{n}(\cos\phi + \sin\phi\mathbf{u})$ is solution of equation (6.5.1) for every $\mathbf{d} \in \mathbb{S}$ such that it is different form $-\mathbf{u}$. From this, we easily check that equation (6.5.3) defines a solution of (6.5.1) since, defining $\mathbf{d}_0 = \frac{\mathbf{d}}{\|\mathbf{d}\|} \in \mathbb{S}$ and $A_0 = \mathbf{n}(\cos\phi + \sin\phi\mathbf{u})$, we have that

$$A\mathbf{u}\bar{A} = \sqrt{\|\mathbf{d}\|}A_0\mathbf{u}\bar{A}_0\sqrt{\|\mathbf{d}\|} = \sqrt{\|\mathbf{d}\|}^2\mathbf{d}_0 = \mathbf{d}.$$

The corresponding formal theorems are the following.

```
QUAT_UNIT_ROTATION_QUAT_SOLUTIONS
  |- !u d A t.
       u pow 2 = --Hx(&1) /\
       d pow 2 = --Hx(&1) /\ ~(u = -- d) /\
       A = inv (norm (u + d)) % (u + d) * (Hx(cos t) + Hx(sin t) * u)
       ==> A * u * (cnj A) = d`,

QUAT_ROTATION_QUAT_SOLUTIONS
  !- !u d A t.
       u pow 2 = -- Hx(&1) /\ (!a. ~(u = Hx(a) * d)) /\
       ~(d = Hx(&0)) /\ Re d = &0 /\
        A = Hx (sqrt (norm d)) * (inv (norm (u + (inv (norm d) % d))) %
            (u + inv (norm d) % d) ) * (Hx(cos t) + Hx(sin t)* u)
        ==> A * u * (cnj A) = d
```

We recall that, since in our formalization quaternions are element of $\mathbb{R}^4$, we have that, given $a \in \mathbb{R}$ and $q = q_0+q_1\mathbf{i}+q_2\mathbf{j}+q_3\mathbf{k} \in \mathbb{H}$, the quaternionic product aq is equivalent to the multiplication of the vector $q = (q_0, q_1, q_2, q_3)$ by the scalar a, that is, $a(q_0, q_1, q_2, q_3) = (aq_0, aq_1, aq_2, aq_3)$. Formally, the writings `Hx a * q` (quaternionic product) and `a % q` (multiplication of a vector by a scalar) have the same meaning, that is, they can be converted into one another just by rewriting.

6.5.2 Existence of PH cubic interpolant

Given four control points $\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3$, the related cubic Bézier curve is defined as

$$\mathbf{r}(t) = b_0^3(t)\mathbf{P}_0 + b_1^3(t)\mathbf{P}_1 + b_2^3(t)\mathbf{P}_2 + b_3^3(t)\mathbf{P}_3 \tag{6.5.5}$$

and by equations (6.2.5) its hodograph is

$$\mathbf{r}'(t) = 3b_0^2(t)(\mathbf{P}_1 - \mathbf{P}_0) + 3b_1^2(t)(\mathbf{P}_2 - \mathbf{P}_1) + b_2^2(t)(\mathbf{P}_3 - \mathbf{P}_2). \tag{6.5.6}$$

Given initial conditions

$$\mathbf{r}(0) = \mathbf{P}_i, \qquad \mathbf{r}(1) = \mathbf{P}_f, \qquad \mathbf{r}'(0) = \mathbf{d}_i, \qquad \mathbf{r}'(1) = \mathbf{d}_f,$$

since by (6.5.5) and (6.5.6) it holds that

$$\begin{aligned} \mathbf{r}(0) &= \mathbf{P}_0 = \mathbf{P}_i & \mathbf{r}'(0) &= 3(\mathbf{P}_1 - \mathbf{P}_0) = \mathbf{d}_i \\ \mathbf{r}(1) &= \mathbf{P}_3 = \mathbf{P}_f & \mathbf{r}'(1) &= 3(\mathbf{P}_3 - \mathbf{P}_2) = \mathbf{d}_f \end{aligned}$$

we have that the control points must be expressed as follows:

- $\mathbf{P}_0 = \mathbf{P}_i$
- $\mathbf{P}_1 = \mathbf{P}_i + \frac{1}{3}\mathbf{d}_i$
- $\mathbf{P}_2 = \mathbf{P}_f - \frac{1}{3}\mathbf{d}_f$
- $\mathbf{P}_3 = \mathbf{P}_f$.

Thus, the "ordinary" cubic interpolant of the dataset $\{\mathbf{P}_i, \mathbf{P}_f, \mathbf{d}_i, \mathbf{d}_f\}$ is

$$\mathbf{r}(t) = b_0^3(t)\mathbf{P}_i + b_1^3(t)(\mathbf{P}_i + \frac{1}{3}\mathbf{d}_i) + b_2^3(t)(\mathbf{P}_f - \frac{1}{3}\mathbf{d}_f) + b_3^3(t)\mathbf{P}_f \tag{6.5.7}$$

and its hodograph is

$$\mathbf{r}'(t) = b_0^2(t)\mathbf{d}_i + b_1^2(t)\mathbf{w} + b_2^2(t)\mathbf{d}_f \tag{6.5.8}$$

with $\mathbf{w} = 3(\mathbf{P}_f - \mathbf{P}_i) - (\mathbf{d}_i + \mathbf{d}_f)$.

Now, the question is: *under which conditions on the initial dataset the curve defined by* (6.5.7) *is a PH-curve?*

From equation (6.4.7), the "ordinary" cubic interpolant $\mathbf{r}(t)$ is PH if and only if exists a quaternion polynomial $A(t) = A_0(1-t) + A_1 t$ such that

$$\mathbf{r}'(t) = A(t)\mathbf{u}\bar{A}(t) = A_0\mathbf{u}\bar{A}_0 b_0^2(t) + \frac{1}{2}(A_0\mathbf{u}\bar{A}_1 + A_1\mathbf{u}\bar{A}_0)b_1^2(t) + A_1\mathbf{u}\bar{A}_1 b_2^2(t) \tag{6.5.9}$$

for some $\mathbf{u} \in \mathbb{S}$.

Assuming (without loss of generality) that $\mathbf{u} = \delta_i = \frac{\mathbf{d}_i}{\|\mathbf{d}_i\|}$, the above hodograph agrees with that in equation (6.5.8) if

$$\begin{aligned} &A_0\delta_i\bar{A}_0 = \mathbf{d}_i \\ &A_1\delta_i\bar{A}_1 = \mathbf{d}_f \\ &A_0\delta_i\bar{A}_1 + A_1\delta_i\bar{A}_0 = 2\mathbf{w} \end{aligned} \tag{6.5.10}$$

It turns out [Farouki et al., 2008] that conditions (6.5.10) are satisfiable if and only if the initial data set satisfies some particular constraints, as shown in the next proposition.

Proposition 6.5.1. *The cubic Hermite interpolant to the data points* $\{\mathbf{P}_i, \mathbf{P}_f\}$ *and derivatives* $\{\mathbf{d}_i, \mathbf{d}_f\}$ *is a PH-curve if and only if*

$$\mathbf{w}\cdot(\delta_i - \delta_f) = 0, \qquad \left(\mathbf{w}\cdot\frac{\delta_i + \delta_f}{\|\delta_i + \delta_f\|}\right)^2 + \frac{(\mathbf{w}\cdot\mathbf{z})^2}{\|z\|^4} = \|\mathbf{d}_i\|\|\mathbf{d}_f\| \tag{6.5.11}$$

with δ_i, $\mathbf{w}$ *defined as above and* $\delta_f = \frac{\mathbf{d}_f}{\|\mathbf{d}_f\|}$, $\mathbf{z} = \frac{\delta_i \times \delta_f}{\|\delta_i \times \delta_f\|}$.

We certify formally that the ordinary cubic interpolant, with a dataset that satisfy conditions (6.5.11), is effectively a PH-curve. The formal proof is about 500 lines of code and involves essentially algebraic properties of quaternions and spatial vectors also that solutions of equation (6.5.1). However, many parts of it can be automated thanks to the automatic procedures as the conversion `QUAT_POLY_CONV` and the rule `QUAT_POLY` presented in section 3.3. The HOL Light formal statement that we proved is the following.

```
PH_CUBIC_ITERPOLANT_EXISTS
  |- !Pf Pi di df.
     let w = Hx(&3) * (Pf - Pi) - (di + df) in
     let n = \v. Hx(inv(norm v)) * v in
     let z = Hx(inv (norm (n di + n df))) *
             Hv (HIm (n di) cross HIm (n df)) in
     let r = (\t. bernstein 3 0 (drop t) % Pi +
              bernstein 3 1 (drop t) % (Pi + Hx(&1 / &3) * di) +
              bernstein 3 2 (drop t) % (Pf - Hx(&1 / &3) * df) +
              bernstein 3 3 (drop t) % Pf) in
     Re Pf = &0 /\ Re Pi = &0 /\ Re di = &0 /\ Re df = &0 /\
     ~(Hx(&0) = di) /\ ~(Hx(&0) = df) /\
     (!a. ~(n di = Hx a * df)) ==>
     pathstart r = Pi /\
     pathfinish r = Pf /\
     pathstart (\t. vector_derivative r (at t)) = di /\
     pathfinish (\t. vector_derivative r (at t)) = df /\
     (w dot (n di - n df) = &0 /\
     (w dot (n (n di + n df))) pow 2 + inv(norm z) pow 4 * (w dot z) pow 2 =
      norm di * norm df
      ==> pythagorean_hodograph r)
```

In the above statement, '`pathstart r`' and '`pathfinish r`' represent $\mathbf{r}(0)$ and $\mathbf{r}(1)$ respectively. Note also that we decide to represent formally every "pure vector" quaternion as an element of type '`:quat`' such that its real part is zero instead of an element of the form '`Hv v`'. This choice allows us to avoid element of type '`:real^3`' so to perform all calculations inside the type '`:quat`'.

6.5.3 PH quintic Hermite interpolants

We have seen in the previous subsection that the lack of an appropriate number of degrees of freedom, on the choices of the coefficients of $A(t)$, doesn't permit to find a PH cubic interpolant for every initial data set. To increase the degrees of freedom, Farouki et al., seek for the existence of quintic interpolants using a quadratic quaternion polynomial

$$A(t) = A_0(1-t)^2 + 2A_1 t(1-t) + A_2 t^2 \tag{6.5.12}$$

instead of a linear polynomial. It turns out [Farouki et al., 2008] that for quintic interpolants things are very different. In fact, a PH quintic interpolant can be found for every initial data points $\{\mathbf{P}_i, \mathbf{P}_f\}$, and derivatives $\{\mathbf{d}_i, \mathbf{d}_f\}$, by the right choice of its control points depending on the coefficients of the polynomial (6.5.12). Actually, there is a two-parameter family of such interpolants and the algebraic expression of $\mathbf{r}(t)$ is substantially more complex with respect to the case of cubics. Following strictly the work of Farouki, Giannelli et al., we certify these results about PH quintics.

The use of $A(t)$ in the form (6.5.12) to define the hodograph $\mathbf{r}'(t) = A(t)\mathbf{u}\bar{A}(t)$ produces, by integration, the associated PH-curve

$$\mathbf{r}(t) = \sum_{i=0}^{5} b_i^5(t)\mathbf{P}_i$$

with control points

- $\mathbf{P}_1 = \mathbf{P}_0 + \frac{1}{5}A_0\mathbf{u}\bar{A}_0$
- $\mathbf{P}_2 = \mathbf{P}_1 + \frac{1}{10}(A_0\mathbf{u}\bar{A}_1 + A_1\mathbf{u}\bar{A}_0)$
- $\mathbf{P}_3 = \mathbf{P}_2 + \frac{1}{30}(A_0\mathbf{u}\bar{A}_2 + A_1\mathbf{u}\bar{A}_1 + A_2\mathbf{u}\bar{A}_0)$

- $\mathbf{P}_4 = \mathbf{P}_3 + \frac{1}{10}(A_1\mathbf{u}\bar{A}_2 + A_2\mathbf{u}\bar{A}_1)$
- $\mathbf{P}_5 = \mathbf{P}_4 + \frac{1}{5}A_2\mathbf{u}\bar{A}_2$.

Now, interpolation of the end-derivatives yields the equations

$$\begin{aligned} \mathbf{r}'(0) &= A_0\mathbf{u}\bar{A}_0 = \mathbf{d}_i \\ \mathbf{r}'(0) &= A_2\mathbf{u}\bar{A}_2 = \mathbf{d}_f \end{aligned} \tag{6.5.13}$$

for A_0 and A_1 that are of the form (6.5.1). Thus, they give solutions

$$\begin{aligned} A_0 &= \sqrt{\|\mathbf{d}_i\|}\mathbf{n}_i(\cos\phi_0 + \sin\phi_0\mathbf{u}) \\ A_2 &= \sqrt{\|\mathbf{d}_f\|}\mathbf{n}_f(\cos\phi_2 + \sin\phi_2\mathbf{u}) \end{aligned} \tag{6.5.14}$$

with ϕ_0, ϕ_2 free angular parameters and $\mathbf{n}_i$, $\mathbf{n}_f$ defined as in equations (6.5.2). Moreover, interpolation of the end points $\mathbf{r}(0) = \mathbf{P}_0 = \mathbf{P}_i$ and $\mathbf{r}(1) = \mathbf{P}_f$ gives the condition

$$\begin{aligned} \int_0^1 \mathbf{r}'(t) = \int_0^1 A(t)\mathbf{u}\bar{A}(t) = \mathbf{P}_f - \mathbf{P}_i =& \frac{1}{5}A_0\mathbf{u}\bar{A}_0 + \frac{1}{10}(A_0\mathbf{u}\bar{A}_1 + A_1\mathbf{u}\bar{A}_0) \\ &+ \frac{1}{30}(A_0\mathbf{u}\bar{A}_2 + A_1\mathbf{u}\bar{A}_1 + A_2\mathbf{u}\bar{A}_0) \\ &+ \frac{1}{10}(A_1\mathbf{u}\bar{A}_2 + A_2\mathbf{u}\bar{A}_1) + \frac{1}{5}A_2\mathbf{u}\bar{A}_2 \end{aligned}$$

that can be reduced to

$$B\mathbf{u}\bar{B} = \mathbf{d} \tag{6.5.15}$$

with

$$B = 3A_0 + 4A_1 + 3A_2 \qquad \mathbf{d} = c + 5(A_0\mathbf{u}\bar{A}_2 + A_2\mathbf{u}\bar{A}_0)$$

where $c = 120(\mathbf{P}_f - \mathbf{P}_i) - 15(\mathbf{d}_i + \mathbf{d}_f)$, by using conditions (6.5.13).
The latter equation is again of the form (6.5.1) so, its general solution is

$$B = \sqrt{\|\mathbf{d}\|}\mathbf{n}(\cos\phi_1 + \sin\phi_1\mathbf{u}) \tag{6.5.16}$$

where ϕ_1 is another free angular parameter and $\mathbf{n}$ is defined as in equations (6.5.2). Finally, we can compute $A_1 = \frac{1}{4}B - \frac{3}{4}(A_0 + A_2)$.

Note that A_1 depends on ϕ_0, ϕ_2 as well as ϕ_1. However, one can show [Farouki et al., 2002] that the hodograph $\mathbf{r}'(t) = A(t)\mathbf{u}\bar{A}(t)$ depends only on the differences of ϕ_0,ϕ_1,ϕ_2. Thus, without loss of generality, we can take $\phi_1 = 0$ and regard the PH quintic interpolant as dependent on just the two angular parameters defined by

$$\alpha = \frac{1}{2}(\phi_0 + \phi_2) \qquad \beta = \phi_2 - \phi_0.$$

Finally, the three quaternions A_0,A_1,A_2, that define a PH quintic Hermite interpolant, can be expressed in terms of α and β as

- $A_0 = \sqrt{\|\mathbf{d}_i\|}\mathbf{n}_i(\cos(\alpha - \frac{1}{2}\beta) + \sin(\alpha - \frac{1}{2}\beta)\mathbf{u})$
- $A_2 = \sqrt{\|\mathbf{d}_f\|}\mathbf{n}_f(\cos(\alpha + \frac{1}{2}\beta) + \sin(\alpha + \frac{1}{2}\beta)\mathbf{u})$
- $A_1 = \frac{1}{4}\sqrt{\|\mathbf{d}\|}\mathbf{n} - \frac{3}{4}(A_0 + A_2)$.

We certify that the curve, obtained by integrating the hodograph $\mathbf{r}'(t) = A(t)\mathbf{u}\bar{A}(t)$, with $A(t)$ defined (as in equation (6.5.12)) by the quaternionic coefficients A_0,A_1,A_2 defined as above, is effectively a PH-curve interpolating the initial data points $\{\mathbf{P}_i, \mathbf{P}_f\}$ and derivatives $\{\mathbf{d}_i, \mathbf{d}_f\}$. The formal theorem is the following.

```
PH_QUINTIC_INTERPOLANT
  |- !Pi Pf di df u p q.
      let w = \v. Hx(inv(norm v)) * v in
      let e = \a. Hx(cos a) + Hx(sin a) * u in
      let a = (p + q) / &2 in
      let b = q - p in
      let ni = w(u + w di) in
      let nf = w(u + w df) in
      let A0 = Hx(sqrt(norm di)) * ni * e (a - b / &2) in
      let A2 = Hx(sqrt(norm df)) * nf * e (a + b / &2) in
      let c = Hx(&120) * (Pf - Pi) - Hx(&15) * (di + df) in
      let d = c + Hx(&5) * (A2 * u * cnj A0 + A0 * u * cnj A2) in
      let n = w (u + w d) in
      let A1 = if d = Hx(&0) then -- Hx(&3 / &4) * (A0 + A2)
               else Hx(&1 / &4 * sqrt(norm d)) * n -
                    Hx(&3 / &4) * (A0 + A2)  in
      let P0 = Pi in
      let P1 = P0 + Hx(&1 / &5) * (A0 * u * cnj A0) in
      let P2 = P1 + Hx(&1 / &10) * (A0 * u * cnj A1 +
                                    A1 * u * cnj A0) in
      let P3 = P2 + Hx(&1 / &30) * (A0 * u * cnj A2 +
                                     Hx(&4) * A1 * u * cnj A1 +
                                     A2 * u * cnj A0) in
      let P4 = P3 + Hx(&1 / &10) * (A1 * u * cnj A2 +
                                    A2 * u * cnj A1) in
      let P5 = P4 + Hx(&1 / &5) * (A2 * u * cnj A2) in
      let r = \t. bernstein 5 0 (drop t) % P0 +
                  bernstein 5 1 (drop t) % P1 +
                  bernstein 5 2 (drop t) % P2 +
                  bernstein 5 3 (drop t) % P3 +
                  bernstein 5 4 (drop t) % P4 +
                  bernstein 5 5 (drop t) % P5 in
      Re Pi = &0 /\ Re Pf = &0 /\ Re di = &0 /\ Re df = &0 /\
      u pow 2 = -- Hx(&1) /\ ~(di = Hx(&0)) /\ ~(df = Hx(&0)) /\
      (!c. ~(u = Hx (c) * di)) /\ (!c. ~(u = Hx(c) * df)) /\
      (!a. ~(u = Hx a * d))
      ==>
      pythagorean_hodograph r /\
      pathstart r = Pi /\
      pathfinish r = Pf /\
      pathstart (\t. vector_derivative r (at t)) = di /\
      pathfinish (\t. vector_derivative r (at t)) = df
```

Again, the formal proof is about 500 lines of code and involves essentially algebraic manipulations of quaternions that can be automated, at least partly, using the rule `QUAT_POLY` and the conversion `QUAT_POLY_CONV`.

Conclusions

In conclusion we presented two applications: a formalization of quaternionic analysis, with focus on the theory of slice regular functions, and the computer verified solutions to the Hermite interpolation problem for cubic and quintic PH-curves.

Along the way, we provided a few extensions of the HOL Light library about multivariate and complex analysis, comprising limit superior and inferior, root test for series, Cauchy-Hadamard formula for the radius of convergence and some basic theorems about derivatives.

Overall, our contribution takes about 10,000 lines of code and consists in about 600 theorems, of which more than 350 have been included in the HOL Light standard library.

This work is certainly open to a wide range of possible improvements and extensions. The most obvious line of developement would be to formalize further mathematical results about quaternions; there is an endless list of potential interesting candidates within reach from the present state of art.

As regards to the core formalization of quaternions, only basic procedures for algebraic simplification are provided. They were somehow sufficient for automating several computations occurring in our applications, but it surely would be interesting to implement more powerful decision procedures. Some of them would probably involve advanced techniques from non-commutative algebra. Moreover, having more automation would be very useful to simplify formal proofs about slice regular functions and PH-curves.

Bibliography

[Affeldt and Cohen, 2017] Affeldt, R. and Cohen, C. (2017). Formal foundations of 3d geometry to model robot manipulators. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, pages 30–42, New York, NY, USA. ACM.

[Barendregt, 2013] Barendregt, H. (2013). *Mathematics, Computer Science and Logic - A Never Ending Story*, chapter Foundations of Mathematics from the Perspective of Computer Verification, pages 1–49. In [Paule, 2013].

[Ciolli et al., 2011] Ciolli, G., Gentili, G., and Maggesi, M. (2011). A certified proof of the cartan fixed point theorems. *Journal of Automated Reasoning*, 47(3):319–336.

[Conway and Smith, 2003] Conway, J. H. and Smith, D. A. (2003). *On quaternions and octonions : their geometry, arithmetic, and symmetry.* Natick, Mass. : AK Peters, c2003.

[Cullen, 1965] Cullen, C. G. (1965). An integral theorem for analytic intrinsic functions on quaternions. *Duke Math. J.*, 32(1):139–148.

[Farouki, 2009] Farouki, R. (2009). *Pythagorean-Hodograph Curves: Algebra and Geometry Inseparable.* Geometry and Computing. Springer Berlin Heidelberg.

[Farouki et al., 2002] Farouki, R. T., al Kandari, M., and Sakkalis, T. (2002). Hermite interpolation by rotation-invariant spatial pythagorean-hodograph curves. *Advances in Computational Mathematics*, 17(4):369 – 383.

[Farouki et al., 2008] Farouki, R. T., Giannelli, C., Manni, C., and Sestini, A. (2008). Identification of spatial ph quintic hermite interpolants with near-optimal shape measures. *Computer Aided Geometric Design*, 25(4):274 – 297.

[Fuchs and Théry, 2014] Fuchs, L. and Théry, L. (2014). Implementing geometric algebra products with binary trees. *Advances in Applied Clifford Algebras*, 24(2):589–611.

[Gabrielli and Maggesi, 2017] Gabrielli, A. and Maggesi, M. (2017). Formalizing basic quaternionic analysis. In Ayala-Rincón, M. and Muñoz, C., editors, *Interactive Theorem Proving*, volume 10499 of *Lecture Notes in Computer Science*, pages 225–240. Springer, Cham.

[Gentili et al., 2013] Gentili, G., Stoppato, C., and Struppa, D. (2013). *Regular Functions of a Quaternionic Variable.* Springer Monographs in Mathematics. Springer Berlin Heidelberg.

[Gentili and Struppa, 2006] Gentili, G. and Struppa, D. C. (2006). A new approach to cullen-regular functions of a quaternionic variable. *Comptes Rendus Mathematique*, 342(10):741 – 744.

[Gonthier, 2005] Gonthier, G. (2005). A computer-checked proof of the four colour theorem.

[Hales, 2005] Hales, T. (2005). A proof of the kepler conjecture. *Ann. Math.*, 162:1065–1185.

[Hales et al., 2017] Hales, T., A., M., Bauer, G., Dang, T. D., Harrison, J., H., L. T., Kaliszyk, C., Magron, V., MCLaughlin, S., Nguyen, T. T., and et al. (2017). A formal proof of the kepler conjecture. *Forum of Mathematics, Pi*, 5.

[Harrison, 2005] Harrison, J. (2005). A HOL theory of Euclidean space. In Hurd, J. and Melham, T., editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005*, volume 3603 of *Lecture Notes in Computer Science*, pages 114–129, Oxford, UK. Springer-Verlag.

[Harrison, 2007] Harrison, J. (2007). Formalizing basic complex analysis. In Matuszewski, R. and Zalewska, A., editors, *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*, volume 10(23) of *Studies in Logic, Grammar and Rhetoric*, pages 151–165. University of Białystok.

[Ma et al., 2016] Ma, S., Shi, Z., Shao, Z., Guan, Y., Li, L., and Li, Y. (2016). Higher-order logic formalization of conformal geometric algebra and its application in verifying a robotic manipulation algorithm. *Advances in Applied Clifford Algebras*, 26(4):1305–1330.

[Paule, 2013] Paule, P., editor (2013). *Mathematics, Computer Science and Logic - A Never Ending Story.* Springer International Publishing.

[Paulson, 1983] Paulson, L. (1983). A higher-order implementation of rewriting.

[Spivak, 1965] Spivak, M. (1965). *Calculus on Manifolds: A Modern Approach to Classical Theorems of Advanced Calculus.* Advanced book program. Avalon Publishing.

[Sudbery, 1979] Sudbery, A. (1979). Quaternionic analysis. *Mathematical Proceedings of the Cambridge Philosophical Society*, 85(02):199–225.

Part III

FORMALIZING BASIC COMPUTABILITY THEORY - THE TURING MACHINE

Introduction

In this part we formalize the basics of a very classical presentation of the theory of Turing machines in the HOL Light theorem prover. Moreover, we prove some basic results about computability. The formalization includes:

- the very definition of Turing machine, execution and halt;
- a few examples of simple Turing machines, the most significant of which is the machine that compute the successor function;
- two HOL conversions to run (i.e., to simulate a certified execution of) a Turing machine (one for the nondeterministic case and another specialised for the deterministic one);
- the definition of Turing computable function (only for the special case of univariate functions);
- the proof that the zero function and the successor function are Turing computable;
- the proof that the composition of computable functions is computable.

At the present stage, the work presented in the next chapters, far from being a complete formalisation of the theory, is the outcome of my first experience of using a theorem prover and has been written as part of an assignment for one of my PhD exams. It is more a proof-of-concept or an extended exercise than a full featured framework.

However, it turns out to be very useful for a deeper understanding of computability theory because it made me aware of how also simple things (e.g. the Turing computability of the successor function) require a lot of work to be proved formally.

For definitions and theorems we follow the textbook [Mundici, 2013] by Mundici, more precisely, the first two chapters.

Outline of the code

The source code of this part of the thesis is reachable at url

https://bitbucket.org/gabra/phdthesis/src/master/Turing%20Machines/

and is written in collaboration with M. Maggesi.
It is composed by the following files:

- **misc.hl** - miscellanea,
- **turing.hl** - definition of Turing machine (including the notions of instructions, configurations, computation step etc.) and simulation of its execution,
- **numtape.hl** - various theorems specific for numeric tapes, that is, tapes that represent a single natural number in unary notation,
- **computability.hl** - definition of Turing computability and verification that the zero function, the identity function and the successor function are Turing computable.

- **composition.hl** - theorem about preservation of Turing computability under composition,
- **examples.hl** - simulation of the execution of some easy Turing machines,
- **run.hl** - definition of a conversion that computes automatically the certified (the output of this process is a theorem) execution of a Turing machine initialized on a concrete configuration,
- **fixprinter.hl** - printer for terms.
- **compute.hl** - Conversion `COMPUT_CONV` by M. Maggesi (University of Florence).

Chapter 7

Turing Machines in HOL Light

This chapter is dedicated to the formalization of one of the most classical topics in Computer Science: Turing Machines and Computability Theory. Our work is directed both to the general theory and to the implementation of a mechanism for a certified execution of a machine.

We are not the first who dealt with the mechanization of basic computability theory and Turing Machines. For example, Norrish [Norrish, 2011] has formalized, in the HOL4 system, results like the Recursion Theorem and the existence of a universal machine using two computational models: the recursive functions and the λ-calculus. He also proved that such models have equivalent computational power.

Moreover, as regarding Turing Machines, Asperti and Ricciotti [Asperti and Ricciotti, 2012] describe a complete formalisation of Turing machines in the *Matita* theorem prover, including a universal Turing machine, but they do not formalize the undecidability of the halting problem since their main focus is complexity, rather than computability theory. However, their setup is very different from ours, since the foundation of *Matita* is based on intuitionistic dependent type theory.

Similarly, Ciaffaglione [Ciaffaglione, 2016] uses the Coq proof assistant to formalize Turing Machines and their operational semantics by using corecursion and coinduction. His approach allows both to certify the correctness of concrete Turing Machines and, as an immediate application, to prove the undecidability of the halting problem. However, also this time, his setup is very different from ours because of the significant differences between the HOL Light theorem prover and the Coq proof assistant foundation.

Again, Xu, Zhang and Urban [Xu et al., 2013] have given a formalization of Turing machines in the Isabelle/HOL theorem prover. This time, the underlying type theory of the theorem prover used is the same of HOL Light but, the main difference from our formalization is that they use the approach by Boolos et. al [Boolos et. al., 2007] as informal guideline whereas we use the textbook by Mundici [Mundici, 2013].

Generally speaking, the textbook of Mundici -and our realization- is much closer to the original Turing's presentation than the implementation given by Xu, Zhang and Urban. More precisely, they use some particular representations and conventions that made the formalization easier so, their basic definitions are slightly different from ours. We give some examples. Turing machines are thought of as having a head, "gliding" over a potentially infinite tape. Both of us and Xu, Zhang and Urban, consider tapes with cells being either blank or occupied but, the effective representation of such tapes are different. In fact, following Mundici's book, we represent tapes as functions $f\colon \mathbb{N} \to A$ from natural numbers to the alphabet (set of symbols) A whereas Xu, Zhang and Urban represent tapes as pairs of list (l, r) where l stands for the tape on the left-hand side of the head and r for the tape on the right-hand side. Furthermore, they accept the *Nop* (do-nothing) operator but we don't consider this possibility. Again, in their formalization, instructions of a Turing machine are pairs consisting of an action and a state of the machine (specific conventions allow us to interpret them in the usual way that is, as quintuples) whereas, in our formal setting, instructions are explicitly *quintuples*. These are only simple examples but they give a good idea of how the same concepts can be formalized in many different way in order to make the formalization easier.

7.1 Basic definitions

In this section we present the main implementation choices regarding the definition of Turing machines.

7.1.1 Turing Machines

A Turing Machine is presented classically as a triple $M = (A, S, I)$ where:

- A is a finite set of symbols called the alphabet whose elements $a_0 = \Box, a_1, \ldots, a_n$ are called symbols,
- S is a subset of natural numbers, called the set of *states* of M, and between its elements there is one called *initial state*,
- I is a finite set of quintuples of the form

  $$\boxed{s_q\ a_i\ a_j\ \Rightarrow\ s_p} \quad \text{or} \quad \boxed{s_q\ a_i\ a_j\ \Leftarrow\ s_p}$$

  where $a_i, a_j \in A$ and $s_q, s_p \in S$; every quintuple of I is called *instruction* and I is *deterministic* if it doesn't contain two instructions with the same beginning $s_q a_i$.

In our implementation, the sets A and S are formalized as types. Hence, several of our type constructions will be polymorphic on two parameters ':A' and ':S', which will be often implicit, since they are automatically inferred by the HOL Light type checking mechanism.

We define a new type `:(A,S)instr` for the instructions, polymorphic in A and S. HOL Light does not provide a mechanism for defining record types, thus we manually introduce the type `:(A,S)instr` as an inductive type, with only one constructor `Instr`, by the following definition.

```
let instr_INDUCT,instr_RECUR = define_type
  "instr = Instr S A A bool S ";;
```

We derive the various projections from the recursive. principle

```
INSTR_PROJS
  |- init (Instr s u v m t) = s /\
     read (Instr s u v m t) = u /\
     write (Instr s u v m t) = v /\
     move (Instr s u v m t) = m /\
     final (Instr s u v m t) = t
```

Here, `s:S` is the initial state of the instruction, `u:A` and `v:A` are symbols, `m:bool` represents the move to do and `t:S` is the final state.

Therefore, in our formalism, a Turing machine is specified by providing its own set of instructions I, i.e. a predicate `:(A,S)instr->bool`[1].

The triple (A, S, I), although it is not made explicit, is uniquely determined by the types and the elements in the set of instructions I, thus it will be frequently denoted formally by `m:(A,S)instr->bool`, as mnemonic for "machine".

We remark that, in principle, the above types allows us to consider Turing machines with an infinite set of instructions but, both in the theorems and in the worked examples, we will restrict ourselves only to the usual finite case.

One other possible implementation would be to employ lists of instructions, which are necessarily finite, instead of sets. However, beside the fact that one may want to explore the theoretical setting of infinite sets of instructions, lists carry information on order and repetitions which are not meaningful in this context.

[1]HOL Light does not have a dedicated type for sets, they are represented as predicates.

7.1.2 Configurations

A configuration is a triple $C = (s, x, f)$ where $s \in S$ is a state, $x \in \mathbb{N}$ is a natural number and $f : \mathbb{N} \longrightarrow A$ is a function that represents the tape. We will assume that $f(y) = \Box$ except for a finite number of exceptions. It would be possible to code in HOL Light the set of configurations with the type `` `:S#num#(num->A)` `` [2]. However, for convenience and clarity, we decide to establish a dedicated type `` `:(A,S)conf` `` (also polymorphic in A and S) isomorphic to the previous Cartesian product

```
let conf_INDUCT,conf_RECUR = define_type
   "conf = Conf S num (num->B)";;
```

with projections

```
CONF_PROJS
  |- status (Conf s x f) = s /\
     cursor (Conf s x f) = x /\
     tape (Conf s x f) = f
```

where `` `s:S` `` is a state, `` `x:num` `` the cursor and `` `f:num->B` `` the tape. In principle, our definitions allow maximum flexibility, thanks to polymorphism of HOL Light, on the choice of the type of states `` `:S` `` and alphabet `` `:A` ``. However, in our development we didn't exploit this generality since we will use natural numbers `` `:num` `` for the states and, following Mundici, the alphabet $alph = \{|, \Box, M\}$ for the symbols.

We define a new finite type

```
let alph_INDUCT,alph_RECUR = define_type
  "alph = One | Blank | Mark";;
```

which has three constructors, `` `One` ``, `` `Blank` `` and `` `Mark` ``, that represent respectively the symbols $|$, $\Box$ and M. Therefore, from our formal point of view, Turing machines and configurations are represented simply by terms of type `` `:(alph,num)instr->bool` `` and `` `:(alph num)conf` `` respectively. In the following, we will write simply `` `:conf` `` and `` `:instr->bool` `` to indicate the types `` `:(alph,num)conf` `` and `` `:(alph,num)instr->bool` `` respectively, but in the attached code it is always specified.

We think that the decision to define the types in a polymorphic way needs more careful reconsideration. In retrospect, it might have been easier not to use polymorphism, which was superfluous for what we formalized, fixing once and for all the types of the alphabet and machine states. Conversely, it might have been useful to make our definition polymorphic to the set of indexes for the cursor of the tape. In this way, the definition would include, for example, the case of Turing machines whose tape is indexed by $\mathbb{Z}$ instead of $\mathbb{N}$ (taking `` `:int` `` instead of `` `:num` `` as the type of indexes). Since this is a preliminary exercise, we have maintained the definitions initially chosen without further thoughts but, for a future development of this code, these implementation choices should definitely be revised more carefully.

7.2 Execution of Turing Machines in HOL Light

Our next task is to specify the execution of a Turing machine in HOL. Subsequently, we will show how the execution can be simulated in concrete examples inside HOL Light.

An instruction $i \in I$ is called ***relevant*** for a configuration $C = (s, x, f)$, if it is of the form

$$\boxed{s\ f(x)\ a_j\ \Rightarrow\ t} \qquad \text{or} \qquad \boxed{s\ f(x)\ a_j\ \Leftarrow\ t}.$$

Assuming $C = (s, x, f)$ to be a configuration, we have two possibilities:

1. no instruction of M is relevant for C, then M halts and the configuration C is called ***final***,

[2]The symbol # represents in HOL Light the Cartesian product between types.

2. there exists at least one (exactly one in the deterministic case) quintuple i, of M, relevant for $C = (s, x, f)$; then i will be of the form $\boxed{s\ f(x)\ a_j\ \Rightarrow\ t}$ or $\boxed{s\ f(x)\ a_j\ \Leftarrow\ t}$ and, starting from C, it yields the following ***next configuration*** $C' = (x', s', f')$ where:
   - $s' = t$;
   - $x' = x + 1$ if i is of the form $\boxed{s\ f(x)\ a_j\ \Rightarrow\ t}$ or $x' = x - 1$ if i is of the form $\boxed{s\ f(x)\ a_j\ \Leftarrow\ t}$;
   - $f' = f$ everywhere except in the box x where $f'(x) = a_j$.

     In a non deterministic case, we obtain a computation tree with a branch for every instruction relevant for C.

Let's define a ***computation step*** of M as a couple of configurations (C, C') where C' is the next configuration of C. The forthcoming subsection illustrate in details the implementation of these concepts.

7.2.1 Relevant instructions

We start with the formal predicate `‘RELEVANT:conf->instr->bool‘` that selects relevant instructions for a given configuration. This is defined as an inductive predicate with just one deduction rule by the command

```
let RELEVANT_RULES,RELEVANT_INDUCT,RELEVANT_CASES =
    new_inductive_definition ‘!s x f v m t.
    RELEVANT (Conf s x f) (Instr s (f x) v m t)‘;;
```

that produces three theorems:

- RELEVANT_RULES
  ```
  |- !s x f v m t. RELEVANT (Conf s x f) (Instr s (f x) v m t)
  ```
  that defines the rules of `‘RELEVANT‘`,
- RELEVANT_INDUCT
  ```
  |- !RELEVANT'.
      (!s x f v m t. RELEVANT' (Conf s x f) (Instr s (f x) v m t))
      ==> (!a0 a1. RELEVANT a0 a1 ==> RELEVANT' a0 a1)
  ```
  that ensures that `‘RELEVANT‘` is the smallest predicate that satisfies the previous rules,
- RELEVANT_CASES
  ```
  |- !a0 a1. RELEVANT a0 a1 <=>
             (?s x f v m t. a0 = Conf s x f /\ a1 = Instr s (f x) v m t)
  ```
  that establishes in which cases `‘RELEVANT‘` is true.

Using these theorems, we can immediately prove a theorem that shows `‘RELEVANT‘` in a more familiar way, in which we find the informal definition. Such a theorem

```
RELEVANT
  |- !s x f s' u v m t. RELEVANT (Conf s x f) (Instr s' u v m t) <=>
                        s = s' /\ u = f x
```

shows that instruction `‘Instr s' u v m t‘` is relevant for a configuration `‘Conf s x f‘` if and only if `‘s = s'‘` and `‘u = f x‘` as prescribed by informal theory.

7.2.2 Next configurations and computation step

Now, we focus on the formalization of next configuration and computation step. In order to define the next configuration we need a function that “update” the tape f and a function that “move” the cursor x when a relevant instruction acts.

UPDATE. We define formally the update function as

```
let UPDATE = new_definition
   `UPDATE f (x:A) (v:B) = (\y. if y = x then v else f y)`;;
```

that prints the symbol `v:B` in the box of abscissa `x:A` on the tape `f`. The following theorem shows the characterizing property of `UPDATE`, that is, the updated tape differs from the older only in position `x:A` where it presents the symbol `v:B`.

```
UPDATE_CLAUSES
   |- (!f x v. UPDATE f x v x = v) /\
      (!f x v y. ~(y = x) ==> UPDATE f x v y = f y)
```

NEXT. Using the update function `UPDATE`, we can define the function `NEXT` that computes the next configuration. The formal definition is the following.

```
let NEXT = new_definition
  `NEXT c i = Conf (final i) (MOVE (move i) (cursor c))
                  (UPDATE (tape c) (cursor c) (write i))`;;
```

We observe that the function `NEXT:conf->instr->conf` always produces the next configuration, also when the instruction `i` is not relevant for the configuration `c`. Furthermore, the auxiliary function `MOVE:bool-> num-> num` updates the position of the cursor of `c` by incrementing its value if the boolean `(move i)` is true or decrementing it otherwise. The following theorem defines the constant `MOVE`.

```
MOVE
  |- MOVE T = SUC /\ MOVE F = PRE
```

Note that, in this implementation, moving the cursor on the left when it sits at the origin, results in no motion at all, since `PRE` is the *truncated* predecessor on natural numbers. Following classical theory presented in the book of Mundici, we should ensure to not have left-hand instructions that act where the cursor is in the origin, because this can create problems with functions of arity greater than 1. In our case, we will work always with function of arity 1 and with well-formed Turing machines that doesn't have left-hand instructions acting when the cursor is in abscissa zero.

Now, we consider the definition of computation step that is formalized by our HOL constant `STEP`.

STEP. We can combine the predicate `RELEVANT` and the function `NEXT` to obtain a definition of *computation step* by the predicate `STEP`. Such a predicate, `STEP m c c'`, holds when `c'` is the next configuration of `c` according to a relevant instruction `i`. The formal definition is the following.

```
let STEP_RULES,STEP_INDUCT,STEP_CASES =
  new_inductive_definition
  `!i c. i IN m /\ RELEVANT c i ==> STEP m c (NEXT c i)`;;
```

Since every finite set can be represented in HOL Light as iteration of `INSERT` starting from the empty set `{}`,[3] for our purposes it is useful to prove the following theorem

```
STEP
  |- (!c c. ~STEP {} c c') /\
     (!i m c c'. STEP (i INSERT m) c c' <=>
                 RELEVANT c i /\ c' = NEXT c i \/ STEP m c c')
```

[3]For example, the set $\{1, 2, 3\}$ is represented in HOL Light as `1 INSERT (2 INSERT (3 INSERT {}))`.

in order to have an operative tool (in the case of finite set of instructions) to calculate `STEP` recursively on `INSERT`. This theorem reduces the computation to a case analysis on the set of instructions formalizing two basic ideas. First of all, we cannot go from a configuration `c` to `c'` if the set of instructions is empty (`{}`). Secondly, if the set of instructions is of the form `i INSERT m`, we can realize a computation step only in two cases:

- `i` is relevant for `c` and `c'` is the next configuration of `c` with respect to `i`,
- the computation step can be realized using an instruction of the set of instructions `m` that we obtain by eliminating `i` from the original set.

This approach works both in the case of a deterministic and non deterministic Turing machine.

Finally, we show how `STEP` works on subsets or unions of set of instructions. Let be `m` a formal Turing machine and suppose that it can performs a computation step from a configuration `c` to a configuration `c'`. Then, it is obvious that the same holds for every machine `m'` that is a superset of `m`. Two different formal reformulations of this properties are given by the following theorems.

```
STEP_MONO
  |- !m m' c c'. m SUBSET m' /\ STEP m c c' ==> STEP m' c c'

STEP_UNION
  |- !m m' c c'. STEP m c c' ==> STEP (m UNION m') c c'
```

7.2.3 Execution of a Turing machine

In this paragraph we define the predicate `EXEC` that implements the notion of execution of a machine. Let `m:instr->bool` be a machine and let `c`, `c'` be two configurations, then `EXEC m c c'` holds if and only if we can make one or more computation steps from `c` to `c'`, using only the instructions of the machine `m`. The formal definition is the following

```
let EXEC = new_definition
  `EXEC m = RTC (STEP m)`;;
```

where `RTC` is the transitive-reflexive closure of a binary relation.

As for the predicate `STEP`, we show some basic proprieties of `EXEC`:

- `EXEC m c c'` holds when the configurations `c` and `c'` are the same,

  ```
  EXEC_REFL
    |- !m c c. EXEC m c c
  ```

- if the set of instructions is empty then `EXEC {} c c'` holds if and only if the configurations are the same,

  ```
  EXEC_EMPTY
    |- !c c'. EXEC {} c c' <=> c = c'
  ```

- `EXEC m c c'` holds if and only if the configurations are equal or if an intermediate configuration `b` exists such that:
  - a computation step from `c` to `b` can be realized,
  - the machine execution brings from `b` to `c'`.

  ```
  EXEC_STEP_LEFT
    |- !m c c'. EXEC m c c' <=>
                c = c' \/ (?b. STEP m c b /\ EXEC m b c').
  ```

An other easy, but very important, propriety of `EXEC` is the transitivity over configurations, namely:

```
EXEC_TRANS
  |- !m c1 c2 c3. EXEC m c1 c2 /\ EXEC m c2 c3 ==> EXEC m c1 c3
```

that ensures that if we can execute the machine from `c1` to `c2` and from `c2` to `c3`, then we can execute it from `c1` to `c3`.

Now, we need an evaluation strategy to simulate in HOL Light the execution of a Turing machine. We define it with the following constant `RUN`.

```
let RUN_DEF = new_definition
  `!m a c c'. RUN m a c c' <=>
             c = c' \/ (?b. STEP a c b /\ EXEC m b c')`;;
```

The relation `RUN` takes as arguments two sets of instructions `m` and `a`, both of type `:(alph,num)instr->bool`. The first is the whole machine (i.e. the whole set of instructions of the machine), whereas the second contains the instructions, that has yet to be processed, in order to find the ones that are relevant for the current configuration.

The intended usage is to take `a` to be equal to `m` in the initial invocation and, in that case, the computation invariant is that `a` will be a subset of `m` in its calculation.

Therefore, `RUN m a c c'` holds if and only if the configurations `c` and `c'` are equal or there exists a configuration `b`, reachable with a single computation step from `c` using only the instructions in `a`, such that we can execute the machine from it to `c'`.

A very important property is the equivalence between `EXEC` and `RUN`, when `a = m`, proved in the following formal theorem.

```
EXEC_EQ_RUN
  |- !m c c'. EXEC m c c' <=> RUN m m c c'
```

The above equivalence follows from the following result (the other implication is obvious).

```
RUN_IMP_EXEC
  |- !m a c c'. a SUBSET m /\ RUN m a c c' ==> EXEC m c c'.
```

Note the specific hypothesis that `a` is a subset of `m`.

Then, we need a theorem that, in case of a finite set of instructions `m`[4], allows us to calculate, in a recursive way, the predicate `RUN` over the quintuples of the machine. The following theorem proved in HOL Light

```
RUN_CLAUSES
  |- (!m c c'. RUN m {} c c' <=> c = c') /\
     (!i m a c c'. RUN m (i INSERT a) c c' <=>
                   (RELEVANT c i /\ RUN m m (NEXT c i) c') \/ RUN m a c c')
```

describes, by case analysis on `a'`, when `RUN m a' c c'` holds:

- when `a' = {}` it holds if and only if the configurations `c` and `c'` are equal,
- when `a' = (i INSERT a)` it holds if and only if one of the next properties hold:
    1. `i` is relevant for `c` and we can execute the machine `m`, using all its instructions, from the next configuration of `c` to `c'`,
    2. we can execute the machine, from `c` to `c'`, with only the instructions of `a` (i.e. the set `a'` without the instruction `i`).

[4]We recall that every finite set in HOL Light is represented as iteration of INSERT

In practice, during the simulation of the execution of a machine, the recursive rewriting of the previous theorems implements the following informal idea. If the instruction `‘i‘` is relevant for the current configuration `‘c‘`, then we compute the next configuration `‘NEXT c i‘` and we restart the research of the relevant instruction for the latter, if it exists, on the whole set of instructions of the machine `‘m‘`. Contrarily, if the instruction `‘i‘` is not relevant for `‘c‘`, then we discard it and we continue the research of the relevant instruction for `‘c‘` on the remaining instructions of `‘m‘`.

Therefore, our strategy to prove a statement of the form `‘EXEC m c c'‘`, where `‘m‘`, `‘c‘` and `‘c'‘` are a concrete Turing machine and configurations, will be in two steps:

1. put the statement in the equivalent form `‘RUN m m c c'‘` thanks to the previous theorem `‘EXEC_EQ_RUN‘`;

2. compute recursively `‘RUN‘` by using the theorem `‘RUN_CLAUSES‘` which performs the step-by-step/instruction-by-instruction calculation.

7.2.4 Execution of finite and deterministic Turing machines

We note that the evaluation strategy presented in the last section can be used to simulate in HOL Light the execution of any Turing machine, including non deterministic machines and machines that are defined by an infinite set of instructions.

However, in case of deterministic Turing machines, our strategy can't capture the benefits, from a computational point of view, of determinism. In fact, in these cases, at each step of the computation there is an unique relevant instruction (i.e., the computation tree is uniquely determined by the initial configuration) so, once it is found, there is no need to look further in the instruction set.

For this reason, we decide to define an alternative evaluation strategy, equivalent to the previous for a deterministic machine, that allow us to obtain, in this case, a reduction of the computational costs during the simulation of the execution.

First of all, we define when a Turing machine is deterministic

```
let DETERMINISTIC = new_definition
  ‘DETERMINISTIC m <=> (!i j. i IN m /\
                              j IN m /\
                              init i = init j /\
                              read i = read j
                              ==> i = j)‘;;
```

and then we prove a few basic results which allow us to easily prove when a given (finite) machine is deterministic:

```
DETERMINISTIC_EMPTY
  |- DETERMINISTIC {}

DETERMINISTIC_INSERT
  |- !m a. DETERMINISTIC (a INSERT m) <=>
           DETERMINISTIC m /\
           (!y. y IN m /\ ~(y = a)
                ==> ~(init a = init y) \/ ~(read a = read y))
```

Since we have in mind to simulate the execution of finite and deterministic Turing machines, we decided, in this special case, to code them as lists (of their instructions), instead of as sets, since their finite and ordered structure. From now on, any Turing machine to which we will refer in this subsection will be an element of type `‘m:(instr)list‘` and the corresponding finite set of instructions will be given by `‘set_of_list m:instr->bool‘`.

So, we define an alternative notion of computational step and execution for deterministic Turing machines as follows.

```
let FSTEP = new_recursive_definition list_RECURSION
  `(!c. FSTEP [] c = NONE) /\
   (!i m c. FSTEP (CONS i m) c =
            if RELEVANT c i then SOME (NEXT c i) else FSTEP m c)`;;

let FEXEC_RULES,FEXEC_INDUCT,FEXEC_CASE = new_inductive_definition
  `(!m c. FEXEC m c c) /\
   (!m c b c'. FSTEP m c = SOME b /\ FEXEC m b c'
               ==> FEXEC m c c')`;;
```

Note that `FSTEP` is now a function (the prefix F can be taken as mnemonic of *function*) instead of a predicate (`STEP`) as in the non-deterministic case.

In the first place, we prove the relation between `STEP`, `EXEC` and `FSTEP`,`FEXEC` and we obtain good rules of calculation for `FEXEC` with the theorem FEXEC_CLAUSES.

- FSTEP_IMP_STEP
  ```
    |- !m c c'. FSTEP m c = SOME c' ==> STEP (set_of_list m) c c'
  ```
  The "deterministic" computation step always implies the "not deterministic" computation step. If we work with a deterministic Turing machine we have an if and only if.

  ```
  FSTEP_EQ_STEP
    |- !m c c'. DETERMINISTIC (set_of_list m)
                ==> (FSTEP m c = SOME c' <=>
                     STEP (set_of_list m) c c')
  ```

- The same holds for `FEXEC`.

  ```
  FEXEC_IMP_EXEC
    |- !m c c'. FEXEC m c c' ==> EXEC (set_of_list m) c c'
  ```

  ```
  FEXEC_EQ_EXEC
    |- !m c c'. DETERMINISTIC (set_of_list m)
                ==> (FEXEC m c c' <=>
                     EXEC (set_of_list m) c c')
  ```

- Finally, the calculation rules for `FEXEC` are the following.

  ```
  FEXEC_CLAUSES
    |- !m c c'. FEXEC m c c' <=>
                c = c' \/ FROM_OPTION (\b. FEXEC m b c') F (FSTEP m c)
  ```

In each of the examples that we will show in the next section, given a Turing machine through the finite set of his quintuples (i.e. an element of type `:(alph,num)instr->bool`), we used the deterministic strategy in the following way:

- we converted the set of the quintuples of the machine into a list (obviously it can be done if and only if the set of quintuples is finite),
- using FEXEC_IMP_EXEC we prove `EXEC m c c'` by proving `FEXEC m c c'` through recursive rewritings of the theorem FEXEC_CLAUSES.

All the examples of simulation, of which we report in section 8.3 only some cases, are fully presented in the online code in the file examples.hl.

7.2.5 Final configuration and Halt

Now we have to implement the "final configuration" and the concept of "halt". We formally define them in the following way.

```
let FINAL = new_definition
  `FINAL m c <=> ~(?i. i IN m /\ RELEVANT c i)`;;

let HALT = new_definition
  `HALT m c c' <=> EXEC m c c' /\ FINAL m c'`;;
```

A configuration `` `c` `` is final for a machine `` `m` `` if and only if every instructions of the machine is not relevant for it, whereas a machine `` `m` `` halts in a configuration `` `c'` ``, starting from a configuration `` `c` ``, if and only if we can execute the machine from `` `c` `` to `` `c'` `` and `` `c'` `` is a final configuration for the machine.

For practical reasons of calculation, we proved the theorem

```
FINAL_CLAUSES
  |- (!c. FINAL {} c) /\
     (!i m c. FINAL (i INSERT m) c <=> ~RELEVANT c i /\ FINAL m c)
```

so to have a recursive technique of calculation over the set of instructions of a machine.

7.3 Tape

As already said, in our formalization the tape of a machine is simply a function over natural numbers. However, to simulate the behaviour of a given machine is appropriate to provide a representation which is easily treatable by a computational point of view. Therefore, thinking of the tape as to a conductive type (specifically to an infinite sequence), we develop a generic library for the representation of infinite sequences.

Again, definitions are made in a polymorphic way although, as before, from a practical point of view we always use the types `` `:alph` `` and `` `:num` ``. We define:

-
  ```
  let CONST = new_definition
      `CONST (a:A) (x:B) = a`;;
  ```

  so `` `CONST Blank` `` represents a blank tape,

-
  ```
  let INS = new_recursive_definition num_RECURSION
    `(!a:A f. INS a f 0 = a) /\
     (!a f x. INS a f (SUC x) = f x)`;;
  ```

  so the function `` `INS a f` `` represents the tape obtained from `` `f` `` by shifting its elements to the right by one and placing a new element `` `a:A` `` at its origin.

We can now write every numeric tape as an iteration of `` `INS` `` on the base function `` `CONST` ``.

We recall that a "numeric tape" is a blank tape where, starting from the abscissa $x = 1$, is printed a natural number in asticolar notation. A natural number n is expressed in asticolar notation by $n + 1$ bars (0 must be represented by one bar). For example □ || □...□ that is, the numeric tape where is printed the number 1, is expressed in its normal form by the HOL term `` `INS Blank (INS One (INS One (CONST Blank)))` ``. This writing allows us to easily compare if two tapes are the same or not.

Since during the execution of a machine the tape have to be "updated" that is, the function `` `UPDATE` `` acts over it, we had to proved some theorems that described the behaviour of `` `UPDATE` `` acting on `` `INS` `` or `` `COST` ``, so to have good rules of calculation. The theorems that regulate the interaction of `` `UPDATE` `` with `` `INS` `` and `` `CONST` `` are the following.

```
UPDATE_CONST
   |- (!a n. UPDATE (CONST a) n a = CONST a) /\
      (!a b. UPDATE (CONST a) 0 b = INS b (CONST a)) /\
      (!a b n. UPDATE (CONST a) (SUC n) b = INS a (UPDATE (CONST a) n b))

UPDATE_INS
   |- (!f a b. UPDATE (INS a f) 0 b = INS b f) /\
      (!f a b n. UPDATE (INS a f) (SUC n) b = INS a (UPDATE f n b))
```

Now, we are able, through successive rewrites, to update the old tape into the new tape that will be in normal form too.

As you note, the previous theorems assume the Peano's representation of natural numbers through the constant `0:num` and the successor function `SUC:num->num`. This means that these theorems can be used to manipulate formal terms of the form, for example,

```
`UPDATE f (SUC (SUC (SUC 0))) One`
```

where we want to update the tape `f` printing `One` in the abscissa `(SUC (SUC (SUC 0)))`, that is, the abscissa $x = 3$.

However, in concrete situations, we write natural numbers by digits (instead of iterations of the successor function), so, in the previous example, we need to be able to compute the term in the form `UPDATE f 3 One`.

We recall that, as explained in chapter 1 (section 1.2), HOL Light uses a binary encoding of natural numbers so, even if definitions of `INS` and `UPDATE` are more natural using Peano's representation, it is convenient to prove again theorems about them in their binary version. In this way, we obtain that they take account of the HOL Light internal representation of numerals. Such theorems are the following.

```
INS_BINARY
  |- (!a f n. INS a f (NUMERAL n) = INS a f n) /\
     (!a f n. INS a f (BIT1 n) = f (BIT0 n)) /\
     (!a f. INS a f _0 = a) /\
     (!a f n. INS a f (BIT0 n) =
     (if n = _0 then a else f (BIT1 (PRE n))))


UPDATE_BINARY
  |- (!f b n. UPDATE f (NUMERAL n) b = UPDATE f n b) /\
     (!a b. UPDATE (CONST a) _0 b = INS b (CONST a)) /\
     (!a b n. UPDATE (CONST a) (BIT1 n) b =
              INS a (UPDATE (CONST a) (BIT0 n) b)) /\
     (!a b n. UPDATE (CONST a) (BIT0 n) b =
              (if n = _0 then INS b (CONST a)
               else INS a (UPDATE (CONST a) (PRE (BIT0 n)) b))) /\
     (!f a b. UPDATE (INS a f) _0 b = INS b f) /\
     (!f a b n. UPDATE (INS a f) (BIT1 n) b =
                INS a (UPDATE f (BIT0 n) b)) /\
     (!f a b n. UPDATE (INS a f) (BIT0 n) b =
                (if n = _0 then INS b f
                 else INS a (UPDATE f (PRE (BIT0 n)) b)))
```

Numeric tape. Now, defining the function

```
let TAPE_OF_NUM = new_definition
  `TAPE_OF_NUM n = INS Blank (ITER n (INS One)(INS One (CONST Blank)))`;;
```

we can quickly create a numeric tape. The function `ITER n f` is the n-th iteration

$$f^n = \underbrace{f \circ f \circ \cdots \circ f}_{n \text{ times}}$$

of a function f. So, given a natural number $n \in \mathbb{N}$, the HOL term `TAPE_OF_NUM n` represents the informal tape

$$\square \ \underbrace{|| \cdots |}_{n+1 \text{ times}} \ \square \ldots$$

where the number n is printed, in asticular notation ($n + 1$ bars), starting from the abscissa $x = 1$.

Finally, we prove a trivial property of `TAPE_OF_NUM n` that allows us to evaluate it in a specific abscissa `x:num`.

```
TAPE_OF_NUM_EVAL
  |- !n x. TAPE_OF_NUM n x = (if 0 < x /\ x <= n + 1 then One else Blank)
```

In the next chapter we formalize the concept of Turing-computability, we prove that some simple functions are Turing-computable and, at the end, we simulate the execution of the related machines.

Chapter 8

Turing computability

In this chapter we develop a first attempt to the formalization in HOL Light of the basics of computability theory. At the present stage, our code is at the level of proof-of-concept to show the feasibility of our approach to encoding the theory in Higher-Order Logic. The main restriction of our formalization is that it is, for now, limited to consider computable functions on natural numbers of arity 1.

In such restricted setting, we show some examples of computable functions and a simple theoretical result. More precisely, we show that computability is preserved under composition.

8.1 Definition and examples

A function $f\colon \mathbb{N} \to \mathbb{N}$ is said to be Turing computable if and only if a Turing machine M exists such that, when it is initialized in the configuration $(s_i, 1, n)$, it halts in the configuration $(s_f, 1, f(n))$, for every input $n \in \mathbb{N}$. Here s_i and s_f are the initial and final states of the machine M respectively. Therefore, the formal definition is the following

```
let TURING_COMPUTABLE = new_definition
  `TURING_COMPUTABLE f <=>
  ?m si sf. (!n. HALT m (Conf si 1 (TAPE_OF_NUM n))
                        (Conf sf 1 (TAPE_OF_NUM (f n))))`;;
```

where `` `(m:(alph,num)instr->bool)` ``, `` `si:num` ``, `` `sf:num` `` are the formal counterpart of the machine M and the states s_i and s_f respectively. All the polymorphic definitions of the previous chapter are type istantiated with `A = alph` and `S = num`.

The easier example of a computable function is the identity function `` `\x. x` `` that is computed trivially by the machine M_{ID} that does nothing. Starting in the initial state and in abscissa $x = 1$ on the beginning of the input n it performs two simple steps:

- moves the cursor on the left in the blank box of abscissa $x = 0$ remaining in the initial state,
- moves again the cursor on the right on the beginning of n in abscissa $x = 1$ going in the final state.

The quintuples of M_{ID} are

$$\{0\ || \Leftarrow 0,\ 0\Box\Box \Rightarrow 1\}$$

that is, formally,

```
`M_ID = {Instr 0 One One F 0,
         Instr 0 Blank Blank T 1}`
```

Using our formal framework to simulate the action of a Turing machine we can prove the theorem

```
TURING_COMPUTABLE_ID
  |- TURING_COMPUTABLE (\x. x)
```

where the proof consists essentially in the running of two computation steps from the initial configuration `‘Conf 0 1 (TAPE_OF_NUM n)‘` to the final `‘Conf 1 1 (TAPE_OF_NUM n)‘`.

Another example, not so easy, is the case of the zero function (i.e. $f(n) = 0$ for all $n \in \mathbb{N}$). The machine M_0 that computes this function acts in 3 steps:

1. starting from the initial configuration it jumps all the bars of the input n (remember that we consider n expressed in asticular notation) until it finds the first blank box in the abscissa $x = n + 2$,
2. it comes back in abscissa $x = 0$ deleting all the bars,
3. it goes in abscissa $x = 1$ printing the bar that represent the output 0.

The quintuples are:

$$M_0 = \{0 \,||\!\Rightarrow 0,\ 0\Box\Box \Leftarrow 1,\ 1 \,|\, \Box \Leftarrow 1,\ 1\Box\Box \Rightarrow 22\Box \,|\!\Leftarrow 3,\ 3\Box\Box \Rightarrow 3\}$$

and in our formalization

```
‘M_0 = {Instr 0 One One T 0,
        Instr 0 Blank Blank F 1,
        Instr 1 One Blank F 1,
        Instr 1 Blank Blank T 2,
        Instr 2 Blank One F 3,
        Instr 3 Blank Blank T 3}‘
```

Now, the formal proof of the theorem

```
TURING_COMPUTABLE_ZERO
  |- TURING_COMPUTABLE (\x. 0)
```

is less easy than the identity function case. This happens because the jumping and deleting phases (1 and 2) depend on the input n, so we have to prove that they produce the right configuration by induction.

In the case of the successor function, the situation is similar. After the first jumping phase of the previous example, we simply add another bar in the box of abscissa $x = n+2$ and then we came back to the abscissa $x = 1$ jumping again all the bars. The quintuples that perform this calculation are

$$M_{\mathrm{SUC}} = \{0 \,||\!\Rightarrow 0,\ 0\Box \,|\!\Leftarrow 1,\ 1 \,||\!\Leftarrow 1,\ 1\Box\Box \Rightarrow 2\}$$

that can be translated in our formalization as

```
‘M_SUC = {Instr 0 One One T 0,
          Instr 0 Blank One F 1,
          Instr 1 One One F 1,
          Instr 1 Blank Blank T 2}‘
```

where 0 is the initial state and 2 is the final state.

As before, even if the statement is rather easy, the formal proof is not trivial and the basic idea behind it[1] is to use the transitivity of `‘EXEC‘`, so to divide the execution in five distinct processes:

0. prove that the configuration $(2, 1, n + 1)$ is final for M_{SUC},
1. simulate the execution from configuration $(0, 1, n)$ to configuration $(0, n + 2, n)$,

[1]Look the attached code in the file `computability.hl` for the formal proof.

2. from configuration $(0, n+2, n)$ to configuration $(0, n+1, n+1)$,
3. simulate the execution from configuration $(1, n+1, n+1)$ to configuration $(1, 0, n+1)$
4. simulate the execution from configuration $(1, 0, n+1)$to the final configuration $(2, 1, n+1)$.

We proved easily the phases 0, 2 and 4 only using definition of `‘FINAL‘` and simulating the execution of the machine, but for the phases 1 and 3 we have to use the induction principle.

We can note that an intuitive propriety like the computability of the zero or the successor function requires considerable efforts to be formally proved.

8.2 Conservation of Turing computability under composition

With the restriction to unary functions in mind, we formally prove in HOL Light the following theorem.

Theorem 8.1. *Turing computability is preserved under composition.*

This means that if two functions $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$, both of arity 1, are respectively calculated by two Turing machines M_1 and M_2, then there exists a Turing machine M that calculates the function $h(n) = g(f(n))$.

The informal proof is very easy, it is enough to consider M as the union of all the quintuples of M_1 and M_2. If the final state of M_1 and the initial state of M_2 are equal and all the other states are different, we can easily prove that M compute the function h. Otherwise, we have first to rename appropriately all the states of M_1 and M_2, and then we can prove the theorem. Again, the formalization of this simple idea is not trivial.

Statement. The statement of the theorem 8.1 can be expressed in HOL Light with the term

```
‘!f g. TURING_COMPUTABLE f /\ TURING_COMPUTABLE g
       ==> TURING_COMPUTABLE (g o f)‘.
```

Idea of the formal proof. We call respectively:

- M_1, s_i, s_{f_1} the machine that computes f and its initial and final states,
- M_2, s_{i_1}, s_f the machine that computes g and its initial and final states.

We construct formally the machine M in the following way:

```
‘m = (({Instr (2 * s) u v m (2 * t) | Instr s u v m t IN m1} UNION
       {Instr (2 * s + 1) u v m (2 * t + 1) | Instr s u v m t IN m2}) UNION
      {Instr (2*sf1) One One F (2*sf1),
       Instr (2*sf1) Blank Blank T (2*si1 +1)})‘
```

So, M is the machine that we obtain from the union of the quintuples of the machines M_1 and M_2 with the states appropriately renamed and two auxiliary quintuples. The latter two are used to simplify the transition of the execution control between the two machines.

We will call M_1^* and M_2^* the machines that we obtain after the renaming. Let S_1 and S_2 be the sets of states of M_1 and M_2, then for the states of M_1^* we take the set $S_1^* = \{2s \mid s \in S_1\}$ and for M_2^* the set $S_2^* = \{2s+1 \mid s \in S_2\}$ [2]. After the definition of M, we prove that it computes the composed function $f \circ g$ in four phases:

1. execution of M_1^* that halts in configuration $(2s_{f_1}, 1, f(n))$,

[2] In this way we are sure that $S_1^* \cap S_2^* = \emptyset$.

2. phase "transition of the execution control" from M_1^* to M_2^*, it leads from configuration $(2s_{f_1}, 1, f(n))$ to $(2s_{i_1}+1, 1, f(n))$ and is realized by the two quintuples that we added to M in addition to those of M_1^* and M_2^*,

3. execution of M_2^* that halts in configuration $(2s_f+1, 1, g(f(n)))$,

4. proof that $(2s_f+1, 1, g(f(n)))$ is final for M.

To complete this proof we need some complementary theorems about `EXEC`, `STEP` and `FINAL` that we present in the next paragraph.

Complementary theorems. We need to prove some theorems that we call "conservation theorems" because they show that a given property is preserved after a renaming of the states. Here we discuss only the case of the predicate `STEP`. The cases of the other two predicates `EXEC` and `FINAL` are analogous and can be found in the source code. Such conservation theorems formalize the idea that if we can make a computation step from a configuration `c` to a configuration `c'` with a machine `m`, then we can do it both with a machine that is the union of `m` with an another machine `m'` and with a machine that has the same instructions of `m` appropriately renamed. They are the following.

-
  ```
  STEP_UNION_L
    |- !m m' c c'. STEP m c c' ==> STEP (m UNION m') c c'

  STEP_UNION_R
    |- !m m' c c'. STEP m' c c' ==> STEP (m UNION m') c c'
  ```

-
  ```
  STEP_IMP_STEP2
    |- !m si sf x f g.
         STEP m (Conf si x f) (Conf sf x g)
         ==>
          STEP {Instr (2 * s) u v m (2 * t) | Instr s u v m t IN m}
           (Conf (2 * si) x f)
           (Conf (2 * sf) x g)

  STEP_IMP_STEP2PLUS_1
    |- !m c1 c2.
         STEP m c1 c2
         ==>
          STEP {Instr (2 * s + 1) u v m (2 * t + 1) | Instr s u v m t IN m}
           (Conf (2 * status c1 + 1) (cursor c1) (tape c1))
           (Conf (2 * status c2 + 1) (cursor c2) (tape c2))
  ```

Conclusions and examples. After the proof of similar results for `EXEC` and `FINAL`, we are able to conclude the original demonstration taking the theorem:

```
COMPOSE_TURING_COMPUTABLE
  |- !f g. TURING_COMPUTABLE f /\ TURING_COMPUTABLE g
         ==> TURING_COMPUTABLE (g o f)
```

that represents the goal of this chapter. From this result, we could easily prove that if a function is composition of two Turing computable functions, then it is Turing computable. The corresponding HOL Light theorem is the following.

```
TURING_COMPUTABLE_COMPOSE_SUFFICIENT_CONDITION
  |- !f. (?g h. TURING_COMPUTABLE h /\ TURING_COMPUTABLE g /\ f = g o h)
             ==> TURING_COMPUTABLE f
```

We can conclude with two easy examples where we use this corollary to prove that the constant function, and the function $n \to n + m$ (the same holds for $n \to m + n$), for every $m \in \mathbb{N}$, are Turing computable. The related formal theorems, proved by induction on m, are the following.

```
TURING_COMPUTABLE_CONST
 |- !m. TURING_COMPUTABLE (\n. m)

TURING_COMPUTABLE_RADD
 |- !m. TURING_COMPUTABLE (\n. n + m)
```

8.3 Simulation of execution

In this section we test, with some examples, our formal framework certifying the execution of the machines presented previously. In fact, we prove that such machines initialized on a specific input produce the expected output. More precisely, the machines that we address are:

- the machine M_{ID} that computes the identity function $f(n) = n$,
- the machine M_0 that computes the zero function $f(n) = 0$,
- the machine M_{SUC} that computes the successor function $f(n) = n + 1$,
- the machine "move to the left" M_{LEFT},
- the machine that computes the characteristic function of even number M_{EVEN}.[3]

In order to improve the readability we install a pretty printer for the numeric tape that represent the element of type `` `:alph` `` as

```
One -> 1
Blank -> .
Mark -> M
```

and a tape with the syntax `<|.....|>`. In this context, for example, a tape where is printed the number 2 starting from the box with abscissa $x = 1$, that is,

```
`INS Blank (INS One (INS One (INS One (CONST Blank))))`
```

is printed as `<|.111|>` . We will use the same convention for informal tapes so $\square\ |||\ \square \ldots \square$ becomes $\langle|.111|\rangle$. In the first three cases, we start from a standard configuration $\langle|.111|\rangle$ and then, we apply M_{ID}, M_0 and M_{SUC} obtaining $\langle|.111|\rangle$, $\langle|.1|\rangle$ and $\langle|.1111|\rangle$ respectively. We certify these simple examples using both non-deterministic and deterministic strategy of execution. Moreover, we performed some informal benchmarks to compare the execution time of the two approach using the `time` function available in HOL Light. The results are the following theorems.

1. M_{ID}

```
val it : thm =
  |- HALT
     {Instr 0 One One F 0,
      Instr 0 Blank Blank T 1}
     (Conf 0 1 <|.111|>)
     (Conf 1 1 <|.111|>)
```

[3]The quintuple of this machine have been obtained from an exercise left to the students in the class of Mundici.

with time 0.0 seconds in both cases (note that in this simple case we don't appreciate the benefit of the deterministic evaluation),

2. M_0

```
val it : thm =
  |- HALT
     {Instr 0 One One T 0,
      Instr 0 Blank Blank F 1,
      Instr 1 One Blank F 1,
      Instr 1 Blank Blank T 2,
      Instr 2 Blank One F 3,
      Instr 3 Blank Blank T 3}
     (Conf 0 1 <|.111|>)
     (Conf 3 1 <|.1|>)
```

   with times 0.990562 seconds and 0.533745 seconds in the not-deterministic and deterministic case respectively,

3. M_{SUC}

```
val it : thm =
  |- HALT
     {Instr 0 One One T 0,
      Instr 0 Blank One F 1,
      Instr 1 One One F 1,
      Instr 1 Blank Blank T 2}
     (Conf 0 1 <|.111|>)
     (Conf 2 1 <|.1111|>)
```

   with times 0.523782 seconds and 0.299519 seconds in the not-deterministic and deterministic case respectively.

Note that, even if calculations are very simple (times are in general very small) the working time of the processor is approximately half for the deterministic strategy.

We repeat these tests also for the other machines and we find that this is, more or less, a general relation between calculation times.

For example, the *move to the left* machine, initialized in the configuration $(0, 5, \langle|M....111|\rangle)$, halts in the final configuration $(0, 1, \langle|.111|\rangle)$ and the related theorem needs about 20 s with the not-deterministic strategy, and about 8 seconds in the deterministic case, to be proved.

Finally, we define the conversion HALT_CONV that, given a Turing machine M and a initial configuration c_i, computes the set $C_f(M, c_i)$ of all the possible final configurations resulting from the execution of M initialized on c_i. In fact, it returns the theorem

$$\vdash \text{HALT } m\ c_i = C_f(M, c_i).$$

Note that if M is deterministic, then $C_f(M, c_i)$ is the singleton $\{c_f\}$ where c_f is the unique final configuration, resulting from the execution of M initialized on c_i. Furthermore, if M loops when it is initialized on c_i, so does HALT_CONV that continues to run without producing any theorem.

We test HALT_CONV with the deterministic machine M_{SUC}

```
HALT_CONV ‘HALT {Instr 0 One One T 0,
                 Instr 0 Blank One F 1,
                 Instr 1 One One F 1,
                 Instr 1 Blank Blank T 2}
                (Conf 0 1 <|.111|>)‘ =
|- HALT
     {Instr 0 One One T 0,
      Instr 0 Blank One F 1,
      Instr 1 One One F 1,
      Instr 1 Blank Blank T 2}
     (Conf 0 1 <|.111|>) =
   {Conf 2 1 <|.1111|>}
```

and with a generic non deterministic machine obtaining the following result.

```
HALT_CONV `HALT {Instr 1 Blank Blank T 0,
                 Instr 1 Blank One T 0}
                (Conf 1 1 <||>)` =
|- HALT
     {Instr 1 Blank Blank T 0,
      Instr 1 Blank One T 0}
     (Conf 1 1 <||>) =
   {Conf 0 2 <||>, Conf 0 2 <|.1|>}
```

Such a conversion is based on a more sophisticated one, `COMPUTE_CONV`, due to Maggesi (University of Florence).

Conclusions

We presented a computer formalization of the basic concepts needed for specifying and reasoning on Turing machines and computable functions in HOL. Moreover, we provided an automatic procedure to simulate, in a certified way, the execution of Turing machines. More precisely, given a Turing machine and an initial configuration on which the machine halts, we can automatically prove a theorems that computes the set of all the possible halt configurations of the machine.

A general problem, also observed in the works cited in the introduction, is that, often, the underlying informal proofs, found in most textbooks, are not sufficiently accurate to be directly usable as a guideline for formalization. In our case, even if the description of the Mundici's book is very detailed, some obvious and tedious steps in the middle of a proof, or simple generalizations, are left to the readers but, unfortunately, they have to be made explicit in a computer formalization.

A concrete and more ambitious plan for further developments would be to demonstrate, in our setting, that primitive recursive functions are Turing computable.

Since we have already formally proved that the zero function and the successor function are Turing computable and that Turing computability is preserved under composition (in the case of arity 1) the fundamental missing ingredients for reaching this goal are the formalization of the computability of projector functions and the conservation theorem of Turing computability under recursion. Moreover, we should also generalize the already formalized results to the case of functions of any arity.

Bibliography

[Asperti and Ricciotti, 2012] Asperti, A. and Ricciotti, W. (2012). Formalizing Turing machines. In *Logic, Language, Information and Computation*, volume 7456 of *Lecture Notes in Computer Science*, pages 1–25. Springer.

[Boolos et. al., 2007] Boolos, G., Burgess, J. P., and Jeffrey, R. C. (2007). *Computability and Logic (5th ed.).* Cambridge University Press.

[Ciaffaglione, 2016] Ciaffaglione, A. (2016). Towards turing computability via coinduction. *Sci. Comput. Program.*, 126(C):31–51.

[Harrison, 2011] Harrison, J. (2011). *HOL Light Tutorial (for version 2.20).*

[Mundici, 2013] Mundici, D. (2013). *Dalla macchina di Turing a P/NP.* Education. McGraw-Hill.

[Norrish, 2011] Norrish, M. (2011). Mechanised computability theory. In van Eekelen, M., Geuvers, H., Schmaltz, J., and Wiedijk, F., editors, *Interactive Theorem Proving*, pages 297–311, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Xu et al., 2013] Xu, J., Zhang, X., and Urban, C. (2013). Mechanising Turing machines and computability theory in Isabelle/HOL. In *Interactive theorem proving*, volume 7998 of *Lecture Notes in Computer Science*, pages 147–162. Springer.

Part IV

FORMALIZING BASIC QUANTUM COMPUTING

Introduction

A calculation process is essentially a physical process that is performed on a machine whose operations obey certain physical laws.

Classical computation is based on an abstract machine model (as, for instance, Turing machine) that works according to a set of rules and principles enunciated by Alan Turing in 1936 and elaborated subsequently by John von Neumann in the 1940s. Even if today we can construct devices very more powerful of those we could create in the first half of the 20th century, these principles remained essentially unchanged. The implicit assumption on which such principles are based is that a Turing machine idealizes a mechanical computing device (with a potentially infinite memory) that obeys the laws of classical physics.

Due to the constant decrease of the dimensions of the calculation devices, we must consider an alternative because, at the microscopic level, classical physics fails. At such level, for example inside an atom, the theory that describes and justifies successfully physical phenomena is quantum mechanics. Today, quantum effects has already started to interfere with the functioning of electronic devices as their dimensions became smaller. For these reasons, but not only, quantum computing is born as an alternative paradigm based on the principles of quantum mechanics.

The idea to realize a model of computation as a quantum isolated system began to appear at the beginning of the 80s and in 1985 Deutsch formalized these concepts [Deutsch, 1985] leading to the modern conception of quantum computing.

Moreover, the introduction of the new calculation model causes important effects also in the field of computational complexity, in fact, it changes the notion of "tractability". As shown by P. Shor in 1994, the problem of integer factorization (classically considered intractable) can be solved efficiently (that is, in polynomial time) with a quantum algorithm.

These considerations, together with the technological ones mentioned above, brought to the success of a research field known today as *information theory and quantum computing*.

What is quantum computing?

Quantum computing is, essentially, computing using quantum-mechanical phenomena. The fundamental differences with the classic paradigm derive from the principles of quantum mechanics that regulate the world of the infinitely small.

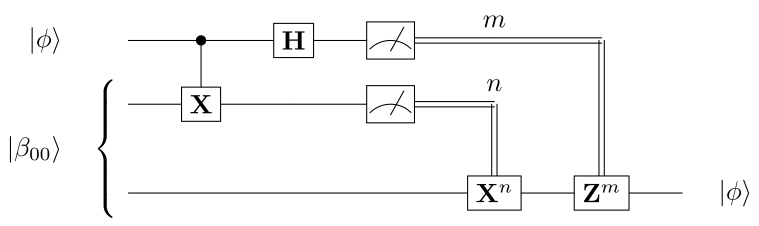
More precisely, quantum computing is based on three quantum phenomena, which are as fundamental as not intuitive, that determine its huge calculation potential. These phenomena are: the principle of superposition of states, the principle of measurement and the phenomenon of entanglement.

The modern mathematical formulation of quantum mechanics is axiomatic, so, forgetting any physical interpretation, it can be treated as a formal system and it can be explored using the mathematical tools of complex linear algebra like, for instance, vectors of complex numbers and Hermitian operators. This allows us to implement in HOL Light the principal concepts of quantum computing (objects and operations) and, in the end, to certify some fundamental quantum protocols and algorithms.

However, at the starting point of our work, HOL Light provided a comprehensive theory only of real linear algebra. We started from the formalization of complex vectors, due to Afshar et al. [Afshar et al., 2014], adding further results about complex linear algebra (e.g. definition of the standard basis vectors and theorems about the extension of complex linear functions defined on such basis vectors) adapting the code, when it was possible, from the real case to the complex case. This has produced a new small library about basics of complex linear algebra. In particular, we formalized the main properties of complex vector spaces (independent complex vectors and their properties) and of complex linear functions (properties of linear and adjoint operators). Since such a theory is well known and serves essentially as a formal support theory for our purposes, we decided, for reasons of time and space, to not report it in this thesis and to consider it as background. Anyway, the related code is reachable in the online repository linked below in the directory `Complex vector`.

A compositional approach

As in the classic case, quantum computations are displayed by quantum circuits. In the following example we report the diagram of the circuit that implements the famous quantum teleportation protocol.



(8.3.1)

Even if the precise meaning of the symbols used in this picture will be introduced in chapter 11, we can still make three very basic remarks.

- From an abstract point of view, a circuit can be thought as a function C that, given an input I, returns the output $O = C(I)$.
- The input and the output are indicated on the left and on the right respectively, so the computation is performed from left to right.
- The circuit is constructed assembling basic components, the so called *quantum logic gates*, represented by the boxes in the picture, that are the simplest operations that can be made on the basic entities of information that is transformed.

The main goal of this part of the thesis is to model and compute, in a certified way, quantum circuits as the one in diagram (8.3.1) using the HOL Light theorem prover. It is clear that such formalization needs a compositional approach. In fact, we have to define a formal framework to express and compute quantum circuits from small (quantum gates) to complex ones.

More precisely, we develop firstly a *compositional* formal language to express quantum circuits assembling quantum gates until the circuit is built. Secondly, we define automatic and certified procedures to simulate the running of quantum circuits. Given a circuit C and an input I, such procedures produce automatically a formal theorem of the form $\vdash C(I) = O$. The latter are also developed in a compositional way, by systematically exploiting the HOL framework of conversions and conversionals. We thus proceed in two steps: first we define the basic conversions for the elementary quantum gates, then we provide a mean to compose them in order to compute arbitrary circuits.

With this formal tools (a suitable language and appropriated automatic procedures), we can formally state and verify (with formal proofs) properties of circuits. For example, we can prove that, if the input of a certain circuit has a specific property, the same holds also for the output (in the spirit of Operational Semantics).

In the next chapters we present our formalization of basics about quantum computing. In chapter 9 we give, firstly, a brief summary of the postulates of quantum mechanics that set the mathematical context on which quantum computing is developed and, secondly, we formalize the basic entities, that is, states of Qbits (quantum bits) and quantum registers. A Qbit is the quantum counterpart of the classical bit (the basic unit of information in classical computing) and a quantum register is a set of Qbits. The latter can be thought as the memory available to perform calculations.

Moreover, along the way, dealing with arbitrary concrete quantum states, we are confronted with some difficulties in using the HOL Light vector algebra (in particular the representation of concrete vectors of arbitrary dimension and the calculation of the components of a vector). Therefore, we have developed a formal theory about types with finite cardinality (*fintypes*) and vectors indexed by such types (*finvec*). This work is discussed in this part of the thesis in chapter 9 (section 9.3) because it is not used elsewhere in this work but, potentially, it has more general applications.

Chapter 10 is dedicated to the main quantum logic gates. We define them formally and we provide, for each of them, a related conversion to compute it. At this point, we are able both to express formally quantum circuits, assembling quantum gates, and to simulate their running, using composed conversions. More precisely, we have an automatic and certified formal calculation system about quantum circuits in which we can specify them and verify their functioning with formal proofs.

Finally, in the last chapter, we compute the certified execution of three fundamental circuits that solve some significant, even if very simple, problems in quantum computing. They are: the already cited *Quantum teleportation protocol*, the *Deutsch's problem* and the *Superdense coding protocol*.

Outline of the code

The source code of this part of the work is reachable at the url

https://bitbucket.org/gabra/phdthesis/src/master/Quantum%20Computing/

and is divided in three subdirectories:

- **Fintypes and Finvec**:
  - **misc.hl** - Miscellanea,
  - **fintype.hl** - introduces a new way to encode finite types,
  - **finvec.hl** - introduces a new way to encode vectors (arrays),
- **Complex Vector**:
  - **matrices.hl** - furter results on real matrices,
  - **misc.hl** - miscellanea,
  - **cvectors.hl** - complex vectors (developed by Sanaz Khan Afshar & Vincent Aravantinos, 2011-13 [Afshar et al., 2014]),
  - **cvectors_more.hl** - further results about cvectors, in particular, theorems about complex linear algebra,
- **Quantum**:
  - **qvectors.hl** - formalization of states of quantum registers, basic definitions and properties of *qvectors* (*quantum* vectors) and the standard computational basis,
  - **quantum.hl** - further properties of *finvec* and *qvectors* needed to deal with formal quantum registers,
  - **quantum_gates.hl** - definition of the main quantum logic gates,
  - **quantum_teleportation_deutsch_problem_superdense_coding.hl** - formal examples of quantum computing, the *Teleportation* and *Superdense coding* protocols and the *Deutsch's* algorithm.

We recall that the code about complex vectors, finite types and vectors indexed by such types has to be loaded to deal with formal quantum computing.

Chapter 9

Basics of quantum computing: Qbits and quantum registers

9.1 Postulates of quantum mechanics

Even if a deep discussion on quantum mechanics is very far from the aim of this work, we recall the axioms on which the formal structure of the whole theory is constructed. We remind that, as usual in quantum mechanics, the Dirac notation *bra-ket* for vectors $|\psi\rangle$, and Hermitian product $\langle\psi|\phi\rangle$, is used.

Postulate 1. *At each instant the state of an isolated physical system is represented by a unit vector (or ket) $|\psi\rangle$ in the space of states associated with such a system.*

The space of states associated to the system is a vector space $\mathcal{H}$ over complex numbers (a separable Hilbert space) equipped with an Hermitian product $\langle\cdot|\cdot\rangle : \mathcal{H} \times \mathcal{H} \to \mathbb{C}$. Every vector $|\psi\rangle$ that represents a possible state of the system is unitary, that is, $\langle\psi|\psi\rangle = 1$. Moreover, the Hilbert space of a composite system is the Hilbert space tensor product of the state spaces associated with the component systems. The states of a composite system that can't be written as the product of the individuals component systems are said to be *entangled*.

Postulate 2. *Every observable $\mathcal{A}$ (a dynamical variable as for instance position, translational momentum, orbital angular momentum, spin, total angular momentum, energy, etc.) of a quantum system is associated with a linear Hermitian operator A whose eigenstates form a complete orthonormal basis for the space of states.*

Since the eigenstates ($|a\rangle$) of A form an orthonormal basis, every state $|\psi\rangle$ of the system can be written in such basis as

$$|\psi\rangle = \sum_a \psi_a\,|a\rangle \qquad \text{with} \qquad \psi_a \in \mathbb{C},\ \sum_a |\psi_a|^2 = 1. \tag{9.1.1}$$

Postulate 3. *The only value which can be obtained as the result of an attempt to measure an observable $\mathcal{A}$ of a quantum system in the state $|\psi\rangle$ is one of the eigenvalues (a) of the Hermitian operator A associated with it.*

Exactly which eigenvalue will be measured cannot generally predicted, but the squared modulus $|\psi_a|^2$ of the coefficient of the eigenstate $|a\rangle$ (associated to the eigenvalue a) in equation (9.1.1) represents the probability that the measurement will yield the result a. This is known as the *Born rule*. Immediately after a measurement, that yields the result a, the state $|\psi\rangle$ of the system collapses in the eigenstate $|a\rangle$. Equivalently, we can interpret $|\psi_a|^2$ as the probability that the system will be in state $|a\rangle$ after the measurement.

Postulate 4. *Between measurements, the state vector $|\psi(t)\rangle$ of a quantum system evolves deterministically according to the equation*

$$i\hbar\frac{d}{dt}\,|\psi(t)\rangle = H\,|\psi(t)\rangle \tag{9.1.2}$$

in which the Hamiltonian operator H is the observable associated with the total energy of the system.

Hence, by the first postulate, it is natural to generalize the classical bit (*Cbit*), i.e. an abstract mathematical object that can have value 0 or 1, with the quantum bit (*Qbit*) that can be described by all the possible unitary states in the Hilbert space spanned by two orthonormal vector $|0\rangle$ and $|1\rangle$. We will see it better in the next sections.

9.2 Formalizing quantum registers

9.2.1 Qbits

A classical computer operates on strings of zeros and ones, such as 100011, converting them into other such strings. Every position of a given string is called *bit*, and it contains either a 0 or a 1. To represent such collection of bits, the computer must contain a corresponding collection of physical systems, each of which can exists in two unambiguously distinguishable physical states.

Therefore, a *Cbit* is an abstract mathematical model for such physical systems that could be in only two different states. For example, a switch that could be open (0) or shut (1), or a magnet whose magnetization could be oriented in two different directions, "up" (0) or "down" (1).

Since a Cbit can have only one of the distinct values 0 or 1, a classical register with n Cbits has value one of the 2^n possible configurations $b_0b_1\dots b_{n-1}$, where $b_i \in \{0,1\}$ for all $i = 0\dots n-1$. For example, a register of five Cbits can have value 10110. It means that the first Cbit has value 1, the second has value 0 and so on.

However, nature provided us with physical systems whose possible states don't suffer from this limitation. For example, if we consider the Hilbert space of the states of an electron spanned by the orthonormal vectors $|0\rangle$ and $|1\rangle$ (interpreted as spin oriented "up" ($|0\rangle$) or "down" ($|1\rangle$) in an uniform magnetic field), we have that, for the first postulate of quantum mechanics, every superposed state

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$$

where α and β are complex numbers such that $|\alpha|^2 + |\beta|^2 = 1$, is a possible state for this system.

At this point, it is natural to extend the concept of Cbits to that of Qbits (quantum bits). A Qbit (or equivalently its state) is described by any unit vector in the two-dimensional complex vector space spanned by the orthonormal vectors $|0\rangle$ and $|1\rangle$, that is,

$$|\psi\rangle = \alpha_0\,|0\rangle + \alpha_1\,|1\rangle = \begin{pmatrix}\alpha_0\\ \alpha_1\end{pmatrix} \in \mathbb{C}^2 \tag{9.2.1}$$

with the only constrain that $|\alpha_0|^2 + |\alpha_1|^2 = 1$.

The basis $\{|0\rangle, |1\rangle\}$, called *standard computational basis*, is orthonormal, that is,

$$\langle i|j\rangle = \delta_{ij} \qquad i,j = 0,1 \tag{9.2.2}$$

and $|\psi\rangle$ is said to be a *superposition* of $|0\rangle$ and $|1\rangle$ with *amplitudes* α_0 and α_1.

For example, physical systems that can represent a Qbit are the electron with oriented spin cited before or a photon with polarization horizontal ($|0\rangle$) or vertical ($|1\rangle$).

From an abstract point of view, the main difference between a Cbit and a Qbit is that a Cbit have value only 0 or 1 whereas every unit vectors $|\psi\rangle$ of the form (9.2.1) is an admissible state for a Qbit.

If we consider n Qbits, things are similar. For the first postulate, the space of states associated to a set of n Qbits is the vector space tensor product $\mathcal{H}$ of the spaces $\mathbb{C}^2$ associated to every Qbit

$$\mathcal{H} = \overbrace{\underbrace{\mathbb{C}^2}_{\text{1-Qbit states}} \otimes \cdots \otimes \mathbb{C}^2}^{n \text{ times}} = \otimes^n \mathbb{C}^2 \cong \mathbb{C}^{2^n}. \tag{9.2.3}$$

Since we do not have an explicit definition of the tensor product in HOL Light, we use the representation as $\mathbb{C}^{2^n}$ for the space of states of a quantum register $\mathcal{H}$. So, any unit vector of the 2^n-dimensional vector space $\mathcal{H} = \mathbb{C}^{2^n}$

$$|\psi\rangle = \sum_{bs} \alpha_{bs} |bs\rangle \tag{9.2.4}$$

$$\sum_{bs} |\alpha_{bs}|^2 = 1 \tag{9.2.5}$$

is a possible state for a *quantum register* with n Qbit.

In equation (9.2.4) the vectors $|bs\rangle = |b_0 b_1 \ldots b_{n-1}\rangle$, with $b_i \in \{0,1\}$ for all $i = 0 \ldots n-1$, are all the possible classical states of a register with n Cbits. The number of all the possible strings with n bits is $|\{0,1\}^n| = 2^n$, that is, the dimension of $\mathcal{H}$ so, the set

$$\{|bs\rangle = |b_0, b_1, \cdots, b_{n-1}\rangle \mid b_i \in \{0,1\} \quad \text{for all} \quad i = 0, \cdots n-1\}$$

has cardinality 2^n and is an orthonormal basis of $\mathcal{H}$. Such a basis is called again *standard computational basis* and, analogously to the case of a single Qbit in equation (9.2.2), the Hermitian product $\langle bs|bs'\rangle$ is 1 if $bs = bs'$ and 0 otherwise.

Therefore, to specify in practice the state of a classical register we have to give n bits (digits in $\{0,1\}$), whereas to describe the state of a quantum register we have to provide 2^n complex numbers (that satisfy the normalization condition (9.2.5)).

For example, considering two Qbits, we have that the general state of a 2-Qbit quantum register, written in the standard computational basis, is

$$|\psi\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle = \begin{pmatrix} \alpha_{00} \\ \alpha_{01} \\ \alpha_{10} \\ \alpha_{11} \end{pmatrix} \in \mathbb{C}^{2^2} = \mathbb{C}^4$$

with the normalization condition $|\alpha_{00}|^2 + |\alpha_{01}|^2 + |\alpha_{10}|^2 + |\alpha_{11}|^2 = 1$.

9.2.2 Formalizing quantum registers: *qvectors*

As already observed in equation (9.2.3), the tensor power $\otimes^n \mathbb{C}^2$ is isomorphic, as a vector space, to $\mathbb{C}^{2^n}$. In our formalism, as explained in section 1.3, the exponent of a cartesian power is encoded by a type, so we need to express the exponentiation 2^n in the language of types. Fortunately, this problem has already been addressed in HOL Light by Harrison for the implementation of Geometric Algebra (file `Multivariate/clifford.ml` of the HOL Light distribution), so we reuse as much as possible the work already done in that context for our purpose. In particular, HOL Light provides the type constructor `‘multivector‘` that, given a type `‘:N‘` with universe of cardinality n, returns the type `‘(:N)multivector‘` with universe of cardinality 2^n. Formally, this property is expressed by the following theorem.

```
DIMINDEX_MULTIVECTOR
 |- dimindex(:(N)multivector) = 2 EXP dimindex(:N)
```

Thus, the vector space formalized by the type `` `:complex^(N)multivector` `` has dimension 2^n so it is the right type to use for our purposes. Because of the original intended application in Geometric Algebra, we call its elements *multivectors*.

However, even if `` `:complex^(N)multivector` `` is suitable to encode the states of a quantum register we have to do some extra work for two main reasons. First of all, the results about *multivectors* provided by the HOL Light library are stated only in the real case, while we need to work with complex coefficients. Secondly, we need to develop a different way to index the components of a *multivector*. More precisely, in Geometric Algebra, *multivectors* are indexed by finite sets of natural numbers. On the other hand, as seen in formula 9.2.4, the components of quantum register, are usually indexed by strings of bits.

The HOL Light indexing operators for *multivectors* is

```
`$$:real^(N)multivector->(num->bool)->real`
```

and it uses the power set $\mathcal{P}(n) = \{S \mid S \subseteq \{1, \ldots, n\}\}$ (since it has cardinality 2^n) to index components of a *multivector*. More precisely, for all $S \in \mathcal{P}(n)$, Harrison has defined the S-th component x_S, of a *multivector* x, as

$$x_S = x_{\text{setcode}(S)} \tag{9.2.6}$$

where setcode is the bijection

$$\text{setcode}\colon \mathcal{P}(n) \to \{1, \ldots, 2^n\} \tag{9.2.7}$$

that has inverse codeset: $\{1, \ldots, 2^n\} \to \mathcal{P}(n)$.
Therefore, following equation 9.2.6, the formal definition of the operator `` `$$` `` is

```
let sindex = new_definition
  `(x:real^(N)multivector)$$s = x$(setcode s)`;;
```

where `` `setcode:(num->bool)->num` `` is the formal counterpart of the function (9.2.7), the operator `` `$:A^(N)->num->real` `` is the standard operator for vectors (presented in section 1.3) and `` `s:num->bool` `` represents an element $S \in \mathcal{P}(n)$.

In quantum computing, it is a common practice to denote the standard computational basis vectors $|bs\rangle$ using natural numbers. More precisely, let $|\psi\rangle$ be a generic state of a quantum register with n Qbits (9.2.4) then it can be written as

$$|\psi\rangle = \sum_{i=0}^{n-1} \alpha_i \, |i\rangle$$

where $|i\rangle$ denotes the basis vector $|bs\rangle$ considering $bs = \text{bin}_n(i)$ as the binary representation (with n bits) of i for every $i \in \{0, \ldots, n-1\}$.

In principle, given this convention, we could think to use natural numbers to index our formal *multivectors*. However, such a choice doesn't allow us to identify and operate easily and directly on the state of a specific Qbit when the register is in the basis state $|i\rangle$. For this reason, we want to index the components of the state of a register `` `x:complex^(N)multivector` `` using the set of all strings of bits with length n that has again cardinality 2^n. In this way, given a basis vector $|bs\rangle = |b_0 b_1 \cdots b_{n-1}\rangle$, we can immediately visualize and operate on the state $|b_i\rangle$ of the i-th Qbit in $|bs\rangle$, for all i in $\{0, \ldots, n-1\}$. Thus, we have to define a new indexing operator, of type `` `:A^(N)multivector->bool^N->A` ``, possibly using the existing code. More precisely, representing the set of string with n bits as $\{0,1\}^n$, the problem is reduced to find a bijection $f\colon \{0,1\}^n \to \{1, \ldots, 2^n\}$ to define the bs-th components of a *multivector* x as

$$x_{bs} = x_{f(bs)} \tag{9.2.8}$$

for every string of n bits bs.

In order to do that, it is very easy to define the bijection bvecset: $\{0,1\} \to \mathcal{P}(n)$ and then consider the following commutative diagram.

$$\begin{tikzcd}
 & \mathcal{P}(n) \arrow[dr, "\text{setcode}"] & \\
\{0,1\}^n \arrow[ur, "\text{bvecset}"] \arrow[rr, "\text{bitscode}"] & & \{1,\dots,2^n\}
\end{tikzcd} \tag{9.2.9}$$

where the searched bijection $f\colon \{0,1\}^n \to \{1,\dots,2^n\}$ is called bitscode and it is defined as setcode $\circ$ bvecset.

Therefore, given a string of n bits $bs = b_0, b_1 \dots b_{n-1} \in \{0,1\}^n$, we define the function bvecset by

$$\text{bvecset}(b_1 b_2 \dots b_n) = \{i \in \{1 \dots n\} \mid b_i = 1\} \in \mathcal{P}(n). \tag{9.2.10}$$

It is clear that, for every $S \in \mathcal{P}(n)$, its inverse is

$$\text{setbvec}(S) = b_1 b_2 \dots b_n \tag{9.2.11}$$

where, for all $i \in \{1,\dots,n\}$, $b_i = 1$ if $i \in S$ and $b_i = 0$ otherwise.

Coding the set $\{0,1\}$ with the type `:bool` and the set of strings of n bits $\{0,1\}^n$ with the type of boolean vectors `:bool^N`, we formalize functions (9.2.11) and (9.2.10) with the following definitions.

```
let setbvec = new_definition
  `setbvec (s:num->bool) = (lambda i. i IN s):bool^N`;;

let bvecset = new_definition
  `bvecset(bs:bool^N) =
    {i | 1 <= i /\ i <= dimindex(:N) /\ bs$i}:num->bool`;;
```

The next theorems show formally that they are one the inverse of the other (with the appropriate conditions since every HOL Light function is total).

```
BVECSET_SETBVEC
  |- !s. bvecset (setbvec s) = s INTER (1..dimindex (:N))

SETBVEC_BVECSET
  |- !bs. setbvec (bvecset bs) = bs
```

Making the previous diagram (9.2.9) commutative, we combine setcode and bvecset getting the function

$$\text{bitscode}(b_1 b_2 \dots b_n) = \text{setcode}(\text{bvecset}(b_1 b_2 \dots b_n)) \tag{9.2.12}$$

that, for every string of n bits, returns a number in $\{1,\dots,2^n\}$. Essentially, given a string of n bits we use bvecset to construct the associated subset of $\{1,\dots n\}$ and then we transform it in a number in $\{1,\dots 2^n\}$ using setcode. Also the function bitscode is a bijection and its inverse is, for all $n \in \{1,\dots,2^n\}$,

$$\text{codebits}(n) = \text{setbvec}(\text{codeset}(n)). \tag{9.2.13}$$

Finally, the function (9.2.12), and its inverse (9.2.13), are formalized by the following definitions

```
let bitscode = new_definition
  `bitscode (bs:bool^N) = setcode (bvecset bs):num`;;

let codebits = new_definition
  `codebits (n:num) = setbvec (codeset n):bool^N`;;
```

and their properties are proved in the next theorems.

```
BITSCODE_BOUNDS
  |- !bs. 1 <= bitscode bs /\ bitscode bs <= 2 EXP dimindex (:N)

BITSCODE_CODEBITS
  |- !n. 1 <= n /\ n <= 2 EXP dimindex (:N) ==> bitscode (codebits n) = n

CODEBITS_BITSCODE
  |- !bs. codebits (bitscode bs) = bs
```

Therefore, following the Harrison style, we use the function ‘bitscode‘ to define our indexing operator (denoted again by the overloaded symbol ‘$$‘) as follows.

```
let qindex = new_definition
  ‘(x:A^(N)multivector)$$bs = x$bitscode bs‘;;
```

From now on, complex *multivectors* (elements of type ‘:complex^(N)multivector‘) indexed by boolean vectors will be called *qvectors* (as mnemonic of *quantum* vectors). The related constructor for *qvectors* ‘(qlambda):(bool^N->A)->A^(N)multivector‘ is given, using the standard one ‘(lambda)‘ for plain vectors, in the following definition.

```
let qlambda = new_definition
 ‘(qlambda) (g:bool^N->A) =
    (lambda i. g(codebits i)):A^(N)multivector‘;;
```

At this point, we prove again the main theorems that allows us to reduce algebraic operations on quantum registers on componentwise calculations expressed with our indexing operator. Essentially, we adapt the existing code about *multivectors* to *qvectors* (that is, to the new definitions of ‘$$‘ and ‘(qlambda)‘). Let ‘bs:bool^N‘ be represent a vector of booleans, then such theorems are the following.

```
QVECTOR_EQ
  |- !u v. u = v <=> (!bs. u $$ bs = v $$ bs)

QLAMBDA_BETA
  |- !f bs. (qlambda) f $$ bs = f bs

QVECTOR_ADD_COMPONENT
  |- !x y bs. (x + y)$$bs = x$$bs + y$$bs

QVECTOR_SUB_COMPONENT
  |- !x y bs. (x - y)$$bs = x$$bs - y$$bs

QVECTOR_NEG_COMPONENT
  |- !x bs. (--x)$$bs = --(x$$bs)

QVECTOR_MUL_COMPONENT
  |- !c x bs. (c % x)$$bs = c * x$$bs
```

9.2.3 Standard computational basis: ‘qbasis‘

Now, we have the right background theory to use the unitary *qvectors* to represent the states of a quantum register with n Qbits. Such *qvectors* ‘x‘ are indexed by boolean vectors ‘bs‘ that indicate to which basis vector $|bs\rangle$ the formal component ‘x $$ bs‘ is related. In fact, every component ‘x $$ bs‘ represents formally the coefficient α_{bs} with respect to the basis-element $|bs\rangle$ in equation (9.2.4). The basis-elements $|bs\rangle$ are also themselves vectors in $\mathbb{C}^{2^n}$ so they can't be formalized by elements of type ‘:bool^N‘. We define a new constant that, given a string of bits bs (an element `‘bs:bool^N‘`), returns the related basis vector $|bs\rangle$ (an element of type `‘:complex^(N)multivector‘`). In order to do this, we adapt again the existing code about *multivectors*.

HOL Light provides a constant ‘`‘mbasis:(num->bool)->real^(N)multivector‘` that formalizes the standard 2^n-dimensional basis vectors of $\mathbb{R}^{2^n}$

$$\begin{pmatrix}1\\0\\\vdots\\0\end{pmatrix},\begin{pmatrix}0\\1\\\vdots\\0\end{pmatrix},\cdots,\begin{pmatrix}0\\0\\\vdots\\1\end{pmatrix} \tag{9.2.14}$$

indexed by the subsets of $\{1,\ldots,n\}$ instead of natural numbers in $\{1,\ldots,2^n\}$. Essentially, given a subset $S \subseteq \{1,\ldots,n\}$, `‘mbasis s‘` is the *multivector* that has the component indexed by S equal to 1 and 0 all the others.

```
MBASIS_COMPONENT;;
  |- !s t. s SUBSET 1..dimindex (:N)
           ==> mbasis t $$ s = (if s = t then &1 else &0)
```

Every *multivector* can be written as linear combination of such vectors.

```
MBASIS_EXPANSION
  |- !x. vsum {s | s SUBSET 1..dimindex (:N)} (\s. x $$ s % mbasis s) = x
```

Correspondingly, for quantum registers, we define a new constant

`‘qbasis:bool^N->complex^(N)multivector‘`

that, for every boolean vector `‘bs:bool^N‘`, returns an element of the basis in (9.2.14) (seen as a vector in $\mathbb{C}^{2^n}$) by embedding in `‘:complex^(N)multivector‘` the related basis vector `‘mbasis (bvecset bs)‘` (we recall that bvecset is the bijection $\{0,1\}^n \to \mathcal{P}(n)$). The formal definition is the following.

```
let qbasis = new_definition
  ‘qbasis (bs:bool^N):complex^(N)multivector =
   vector_to_cvector (mbasis (bvecset bs))‘;;
```

Therefore, in our formal context, the basis vectors $|bs\rangle$ are formalized by the *qvectors* `‘qbasis bs‘`. At this point, we formalize all the main properties of $|bs\rangle$.

- For each basis vector $|bs\rangle$ the decomposition

  $$|bs\rangle = \sum_{bs'} \alpha_{bs'} \, |bs'\rangle$$

  is such that $a_{bs'} = 1$ if $bs = bs'$ and 0 otherwise.

  ```
  QBASIS_COMPONENT
    |- !bs bs’. qbasis bs $$ bs’ = (if bs = bs’ then Cx (&1) else Cx (&0))
  ```

- The vectors $|bs\rangle$ are a basis of $\mathbb{C}^{2^n}$ that is, they are independent and their span is the whole space.

  ```
  CINDEPENDENT_QBASIS
    |- cindependent {qbasis bs | bs IN (:bool^N)}

  CSPAN_QBASIS
    |- cspan {qbasis bs | bs IN (:bool^N)} = (:complex^(N)multivector)
  ```

The constant `cspan` and `cindependent` represents formally the span and the independence of a set of complex vectors respectively.

- The basis vectors $|bs\rangle$ are orthonormal, that is $\langle bs|bs'\rangle = 1$ if $bs = bs'$ and 0 otherwise.

  ```
  QBASIS_CDOT
    |- !bs bs'. qbasis bs cdot qbasis bs' =
                (if bs = bs' then Cx (&1) else Cx (&0))
  ```

  The constant `cdot` represents the Hermitian product $\langle\cdot|\cdot\rangle$.

Then, we prove, as a consequence, that every *qvector* can be written as a linear combination of such basis vectors formalizing, finally, equation (9.2.4).

```
QBASIS_EXPANSION
  |- !x. msum (:bool^N) (\bs. x $$ bs % qbasis bs) = x
```

Here, the constant `msum` encodes sums of complex vectors.[1]

In these formal settings, for example, the elements of the standard computational basis of a register with 2 Qbits

$$|00\rangle\,,\ |01\rangle\,,\ |10\rangle\,,\ |11\rangle$$

are represented respectively by the following formal *qvectors*.

```
`qbasis (vector[F;F]:bool^2)`,  `qbasis (vector[F;T]:bool^2)`,
`qbasis (vector[T;F]:bool^2)`,  `qbasis (vector[T;T]:bool^2)`
```

However, this implementation, although theoretically adequate, has some practical inconveniences.

1. It is cumbersome, at the present stage, to write registers of arbitrary fixed dimension grater than four because, in HOL Light, only the finite types with at most four elements are defined. For example, if we wanted to express the basis state $|10011\rangle$, of a register with five Qbits, we should use the term `qbasis (vector[T;F;F;T;T]:bool^5)` but, unfortunately, HOL Light interprets the type `:5` as a type variable (in the same way of `:N`) because the finite type with five elements is not defined. Hence, the previous term can't represent what we have in mind. As we have seen in section 1.3, we could define the type `:5` and prove the related theorem about its size by it but, if we change the number of Qbits considered (for instance, six Qbits), we have to re-do the job. For this reason, a systematic approach is needed.

2. Even if a concrete finite type is already defined, we haven't a general method to enumerate, automatically, the elements of the finite type of boolean vectors indexed by such type. This turns out to be useful in practice to expand the statement of QBASIS_EXPANSION in an explicit sum when we deal with a specific *qvector* (i.e. a concrete state of a quantum register). For example, let $|x\rangle$ be a generic state of a 2-Qbit register. We are able to write $|x\rangle$ as a sum of the basis vectors

   $$|x\rangle = \sum_{bs\in\{0,1\}^2} x_{bs}\,|bs\rangle$$

   in fact, instantiating QBASIS_EXPANSION with the term `x:complex^(N)multivector`, we obtain the following theorem

   ```
     |- msum (:bool^2) (\bs. x $$ bs % qbasis bs) = x
   ```

[1] Complex vectors, in HOL Light, are element of type `:complex^N`, that is `:real^2^N`. This implies that they are, in fact, real matrices so the prefix `m` in `msum` is mnemonic for *matrix*.

but, we do not have a mean to rewrite it, automatically, in its explicit form

$$|x\rangle = x_{00}\,|00\rangle + x_{01}\,|01\rangle + x_{10}\,|10\rangle + x_{11}\,|11\rangle\,.$$

3. When we write explicitly a basis-vector ‘`qbasis (vector l:bool^N)`‘ we have always to specify the type ‘`:bool^N`‘ of the term ‘`vector l`‘ because, as we have seen in section 1.3, it is not inferred by the length of the list ‘`l`‘. So, it would be much more practical if we give a concrete representation of boolean vectors ‘`bs`‘ such that their type, and also the type of ‘`qbasis bs`‘, is inferred by their form.

4. We need an automatic procedure to compute components of a concrete vector of type ‘`:A^N`‘. HOL Light offers *ad hoc* theorems for the real cases of dimension up to four. Of course, this can be extend case by case to higher dimensions, but for an extensive use a more systematic approach becomes necessary.

   For example, after defining the finite type with seven elements ‘`:7`‘, the basis state $|1000110\rangle$ would be formalized by ‘`qbasis (vector[T;F;F;F;T;T;F]:bool^7)`‘. However, it would be cumbersome to compute that, in such state, the first Qbit (the same holds for all the other Qbits) is in state $|1\rangle$. This happens because there is not an automatic procedure to prove the HOL term ‘`(vector[T;F;F;F;T;T;F]:bool^7)$1 = T`‘.

   Moreover, such a procedure is necessary to deal with quantum logical gates because they works only on a single Qbit (at most on a couple) of the register at time.

For these reasons, before implementing the main quantum logical gates, we develop a formal theory about types with specified finite cardinality and vectors over such types. From now on, we will call them *fintypes* and *finvec* respectively.

9.3 *Fintypes* and *finvec* formal theory

9.3.1 Fintypes

It is well known that HOL Light does not have dependent types. However, for small fragments of language, it is possible to create *ad hoc* structures that simulate, to a certain extent, the paradigm of dependent types. In chapter 1 (section 1.3) we have seen a possible solution to get around the problem in case of vectors. In fact, Harrison used this approach to construct a vector theory (lists with assigned length) encoding positive natural numbers in the type language. This coding is very simple and consists in taking the cardinality of the type. More generally, a fragment of dependent types can be obtained whenever you specify a way to encode a data type in the type language.

In this work, we refine the Harrison technique by coding a slightly more structured language that is, the inductive data type of binary natural numbers. More precisely, we represent every positive natural number into the language of HOL types using a binary encoding. In principle, we could have chosen a different encoding, like the unary notation (using the zero and the successor function), in order to associate a type to each natural number. However, it would have been less adapted when, for instance, we want to compute the component of a vector, because it is not coherent with the HOL Light internal representation of natural numbers that is binary, as explained in chapter 1 (section 1.2).

The basic idea is to build finite types following such representation, so our encoding of binary numerals in the language of types has the same structure. We define two polymorphic types:

- ‘`:(A)tybit0`‘ as the finite sum ‘`:(A,A)finite_sum`‘,
- ‘`:(A)tybit1`‘ as the finite sum ‘`:((A,A)finite_sum),1)finite_sum`‘ where ‘`:1`‘ is the finite type with one element.

The formal definitions are the following.

```
let tybit0_INDUCT,tybit0_RECUR = define_type
  "tybit0 = mktybit0((A,A)finite_sum)";;

let tybit1_INDUCT,tybit1_RECUR = define_type
  "tybit1 = mktybit1(((A,A)finite_sum,1)finite_sum)";;
```

Here, ‘mktybit0‘ and ‘mktybit1‘ are the constructors of ‘:(A)tybit0‘ and ‘:(A)tybit1‘ respectively. So, by the previous definitions it holds that if ‘:A‘ is a type with cardinality n, then the types ‘:(A)tybit0‘ and ‘:(A)tybit1‘ have cardinality $2n$ and $2n + 1$ respectively. It is shown in the following theorems.

```
DIMINDEX_TYBIT0
  |- dimindex (:(A)tybit0) = 2 * dimindex (:A)

DIMINDEX_TYBIT1
  |- dimindex (:(A)tybit1) = 2 * dimindex (:A) + 1
```

As we have seen in chapter 1 (section 1.3), this is a general result indeed, by the definition of ‘dimindex‘, it holds also in the case that ‘:A‘ is an infinite type.

In this setting, for every natural number n, the related *fintype* with n elements is defined by subsequent applications of ‘tybit0‘ or ‘tybit1‘ starting form the basic type ‘:1‘. For example, the type ‘:3‘, with three elements, is constructed as ‘:(1)tybit1‘.

It is clear that the form of a *fintype* is strictly related with the binary representation of its cardinality. More precisely, let be ‘:N‘ a *fintype* written explicitly as subsequent applications of ‘tybit0‘ or ‘tybit1‘. Replacing every ‘tybit0‘ with ‘BIT0‘, ‘tybit1‘ with ‘BIT1‘ and ‘:1‘ with ‘BIT1 _0‘ (we recall that ‘dimindex(:1) = 1‘) we obtain the HOL Light formal representation of the natural number ‘dimindex(:N)‘. Hence, proving these following simple properties

```
pth_num |- dimindex (:A) = n <=> dimindex s = NUMERAL n
pth0 |- dimindex (:A) = n <=> dimindex (:(A)tybit0) = BIT0 n
pth1 |- dimindex (:A) = n <=> dimindex (:(A)tybit1) = BIT1 n
pth_one |- dimindex (:1) = BIT1 _0
```

we can define the conversion DIMINDEX_CONV that computes, by a simple recursive case analysis on the form of the type ‘:N‘, the size of its universe ,that is, ‘dimindex(:N)‘. For example, DIMINDEX_CONV ‘dimindex(:7)‘ returns the theorem |- dimindex(:7) = 7 since ‘:7 = :((1)tybit1)tybit1‘ and, hence, the size of its universe is ‘BIT1 (BIT1 (BIT1 _0))‘, that is, the natural number ‘7‘.

We have extended the parser and the printer of HOL Light so that type identifier that are numerals, are internally encoded as *fintypes*. For example, the type of complex vectors of dimension three is still denoted by ‘:complex^3‘ but, in our settings, it is internally represented by ‘:complex^(1)tybit1‘. This representation is very useful dealing with vectors. For example, the computation of a component of a vector can be automated instead of being performed by providing *ad hoc* theorems as we will show in the next section.

9.3.2 Finvec

Working in a theorem prover that has dependent types, it is easy to define a type of binary natural numbers and then a type of vectors with fixed length dependent on "num". For example, in the Coq's standard library are available the type positive

```
Inductive positive : Set :=
  | xI : positive -> positive
  | xO : positive -> positive
  | xH : positive.
```

for binary natural numbers and we could easily define the type vector by th following code.

```
Inductive vector (A : Set) : positive -> Set :=
  | Vecx (a : A) : vector A xH
  | Vec0 (n: positive) (u v : vector A n) : vector A (x0 n)
  | Vec1 (a : A) (n : positive) (u v : vector A n) : vector A (xI n).
```

In HOL Light, our development of *finvec* corresponds ideally to this setting. Since in our framework binary positive natural numbers are encoded in the language of types by *fintypes* `‘:N‘`, we construct vectors of type `‘:A^1‘`, `‘:A^(N)tybit0‘` and `‘:A^(N)tybit1‘` in the same style, that is, defining three different constructors as shown in the following.

- **The base case.**
  Given an element $a \in A$, the function

  $$\mathrm{vec_x}\colon A \to A^1$$

  returns the vector $\mathrm{vec_x}(a) \in A^1$ such that $\mathrm{vec_x}(a)_0 = a$. The formal definition is the following.

  ```
  let VECX_DEF = new_definition
    ‘vecx a:A^1 = lambda0 i. a‘;;
  ```

  Note that, the the types `‘:A‘` and `‘:A^1‘`, although are commonly identified by an abuse of notation, are formally distinct.

- **The `tybit0` case.**
  Given two vectors $x = (x_0, x_1, \dots x_{n-1}) \in A^n$ and $y = (y_0, y_1, \dots y_{n-1}) \in A^n$, the function

  $$\mathrm{vec_0}\colon A^n \times A^n \to A^{2n}$$

  returns the $2n$-dimensional vector

  $$\mathrm{vec_0}(x, y) = (y_0, x_0, y_1, x_1 \dots y_{n-1}, x_{n-1}) \in A^{2n}$$

  where the odd components are those of x and the even components are those of y, that is, $\mathrm{vec_0}(x, y)_{2i+1} = x_i$ and $\mathrm{vec_0}(x, y)_{2i} = y_i$ for all $i \in \{0, \dots n-1\}$. The formal definition is the following.

  ```
  let VEC0_DEF = new_definition
    ‘vec0 (x:A^N) (y:A^N) : A^N tybit0 =
     lambda0 i. (if ODD i then x else y)$.(i DIV 2)‘;;
  ```

- **The `tybit1` case.**
  Given two vectors $x = (x_0, x_1, \dots x_{n-1}) \in A^n$, $y = (y_0, y_1, \dots y_{n-1}) \in A^n$ and an element $a \in A$, the function

  $$\mathrm{vec_1}\colon A^n \times A^n \times A \to A^{2n+1}$$

  returns the $2n+1$-dimensional vector

  $$\mathrm{vec_1}(x, y, a) = (y_0, x_0, y_1, x_1 \dots y_{n-1}, x_{n-1}, a) \in A^{2n+1}$$

  where the odd components are those of x, the even components are those of y and the last component is a, that is, $\mathrm{vec_1}(x, y, a)_{2i+1} = x_i$, $\mathrm{vec_1}(x, y, a)_{2i} = y_i$ for all $i \in \{0, \dots n-1\}$ and $\mathrm{vec_1}(x, y, a)_{2n} = a$. The formal definition is the following.

  ```
  let VEC1_DEF = new_definition
    ‘vec1 (x:A^N) (y:A^N) (a:A) : A^N tybit1 =
     lambda0 i. if i = BIT0 (dimindex(:N)) then a
                else (if ODD i then x else y)$.(i DIV 2)‘;;
  ```

Usually, the representation of vectors is "sequential", that is, their elements are listed one after the other as, for example, in $\mathbf{v} = (v_0, v_1, \dots, v_n)$. In contrast, our encoding uses a representation of vectors constructed as trees splitting, at each step, even and odd components. As an example, consider a vector $\mathbf{v} = (v_0, v_1, v_2, v_3, v_4) \in A^5$ with five elements. It is represented, in our setting, as

$$\mathrm{vec}_1(\mathrm{vec}_0(v_3, v_1), \mathrm{vec}_0(v_2, v_0), v_4)$$

and its construction can be visualized by the following tree.

$$\begin{tikzcd}
 & & & & \mathbf{v} = \mathrm{vec}_1(\mathrm{vec}_0(v_3, v_1), \mathrm{vec}_0(v_2, v_0), v_4) \in A^5 \arrow[dlll, dash] \arrow[d, dash] \arrow[drrr, dash] & & & \\
 & \mathrm{vec}_0(v_3, v_1) \in A^2 \arrow[dl, dash] \arrow[dr, dash] & & & \mathrm{vec}_0(v_2, v_0) \in A^2 \arrow[dl, dash] \arrow[dr, dash] & & & v_4 \in A \\
v_3 \in A & & v_1 \in A & v_2 \in A & & v_0 \in A & &
\end{tikzcd}$$

In this way, the computation of th i-th component of v, for all $i \in \{0, 1, 2, 3, 4\}$, can be easily made by visiting the proper branch of the tree depending on the parity of i. More precisely, starting from the root of the tree, we have two possibilities:

- if i is even, that is, $i = 2n$ for some $n \in \{0, 1, 2\}$, then we have two subcases:
  - if $n = 2$, then the 4-th component of v is v_4,
  - if $n \in \{0, 1\}$, then we select the central branch of the three and we compute recursively the n-th component of $\mathrm{vec}_0(v_2, v_0)$;
- if i is odd, that is, $i = 2n + 1$ for some $n \in \{0, 1\}$, then we select the left branch of the three and we compute, recursively, the n-th component of $\mathrm{vec}_0(v_3, v_1)$.

This is only an informal idea of the calculation and we will show the formal procedure in details later, but it seems obvious that our approach should allow us to calculate the components of a vector faster then the classical one that scans all the components until the current one.

Even if the HOL Light standard library uses a 1-based indexing for vectors, we choose a more convenient 0-based indexing. We make such a choice for two main reasons. The first is that, often, in quantum computing we refer to the first Qbit, of a certain register, as q_0 or the 0-th Qbit. Secondly, with a 0-based indexing, we can perform efficiently the above computation of the components of a *finvec* depending on the fact that the index is odd or even. So, we define the 0-based indexing operator `‘$.‘` by

```
let index0 = new_definition
  ‘v:A^N $. i = v$(i+1) ‘;;
```

and, consequently, the related constructor `‘(lambda0)‘`.

```
let lambda0 = new_definition
  ‘(lambda0) g:A^N = (lambda) (\i. g (i - 1))‘;;
```

Then, we prove technical theorems, about β-reduction and equality, with respect to them.

```
LAMBDA0_BETA
  |- !i. i < dimindex (:B) ==> (lambda0) g $. i = g i

CART_EQ0
  |- !x y. x = y <=> (!i. i < dimindex (:N) ==> x $. i = y $. i)
```

Note that, the standard requirement that $1 \le i \le n$ is replaced by $i < n$ because, in this context, we use 0-based indexes.

Now, we want to formalize the previous procedure to compute the components of a *finvec*. As said before, the computation is performed depending on the parity of the index considered. Recalling the definitions of the *finvec* constructors we have that:

- considering an element $a \in A$, then

$$\mathrm{vec}_{\mathrm{x}}(a)_0 = a \tag{9.3.1}$$

- considering the vectors $x \in A^n$, $y \in A^n$ and an index $i < n$, then the components of $\mathrm{vec}_0(x, y) \in A^{2n}$ satisfies the conditions

$$\begin{aligned} \mathrm{vec}_0(x, y)_{2i} &= y_i \\ \mathrm{vec}_0(x, y)_{2n+1} &= x_i \end{aligned} \tag{9.3.2}$$

- considering the vectors $x \in A^n$, $y \in A^n$, an element $a \in A$ and an index $i < n$, then the components of $\mathrm{vec}_1(x, y, a) \in A^{2n+1}$ satisfies the conditions

$$\begin{aligned} \mathrm{vec}_1(x, y, a)_{2i} &= y_i \\ \mathrm{vec}_1(x, y, a)_{2i+1} &= x_i \\ \mathrm{vec}_1(x, y, a)_{2n} &= a \end{aligned} \tag{9.3.3}$$

Since even and odd numbers are characterized by the constant ‘`BIT0`‘ and ‘`BIT1`‘ respectively (the previous informal indexes $2i$ and $2i+1$ are formalized by ‘`BIT0 i`‘ and ‘`BIT1 i`‘), we must describe the interaction of ‘`$.`‘ with such constants, for all the constructors ‘`vecx`‘, ‘`vec0`‘ and ‘`vec1`‘ formalizing equations (9.3.1), (9.3.2), (9.3.3). The following formal theorems present these results in a suitable form that can be technically used in the development of our automatic procedure that we will present later.

```
VECX_COMPONENT0
  |- !a. vecx a $. 0 = a

VEC0_COMPONENTS_ARITH
  |- (!x y i. vec0 x y $. NUMERAL i = vec0 x y $. i) /\
     (!x y b i.
          i < dimindex (:N) /\ y $. i = b
          ==> BIT0 i < dimindex (:(N)tybit0) /\ vec0 x y $. BIT0 i = b) /\
     (!x y b i.
          i < dimindex (:N) /\ x $. i = b
          ==> BIT1 i < dimindex (:(N)tybit0) /\ vec0 x y $. BIT1 i = b)

VEC1_COMPONENTS_ARITH
  |- (!a x y i. vec1 x y a $. NUMERAL i = vec1 x y a $. i) /\
     (!a x y i. dimindex (:N) = i
                ==> BIT0 i < dimindex (:(N)tybit1) /\
                    vec1 x y a $. BIT0 (dimindex (:N)) = a) /\
     (!a x y b i.
          i < dimindex (:N) /\ y $. i = b
          ==> BIT0 i < dimindex (:(N)tybit1) /\ vec1 x y a $. BIT0 i = b) /\
     (!a x y b i.
          i < dimindex (:N) /\ x $. i = b
          ==> BIT1 i < dimindex (:(N)tybit1) /\ vec1 x y a $. BIT1 i = b)
```

Such theorems, with `CART_EQ0`, allows us to prove two important formal theorems. The first is that the constructors ‘`vecx`‘, ‘`vec0`‘ and ‘`vec1`‘ are surjective as shown in the following theorem.

```
FINVEC_SURJ
  |- (!v. ?x. v = vecx x) /\
     (!v. ?x y. v = vec0 x y) /\
     (!v. ?x y a. v = vec1 x y a)
```

Therefore, for every *fintype* `‘:N‘`, every *finvec* `‘v:A^N‘` is constructed by subsequent applications of `‘vecx‘`, `‘vec0‘` and `‘vec1‘` depending only on the form of the type `‘:N‘`. It implies that two *finvec*, with the same type `‘:A^N‘`, have necessarily the same decomposition. Moreover, the *finvec* constructors are injective, that is, their images are equal if and only if are equal their arguments. Formally, it is the next theorem.

```
FINVEC_EQ;;
  |- (!a b. vecx a = vecx b <=> a = b) /\
     (!x1 x2 y1 y2. vec0 x1 y1 = vec0 x2 y2 <=> x1 = x2 /\ y1 = y2) /\
     (!x1 x2 y1 y2 c1 c2.
             vec1 x1 y1 c1 = vec1 x2 y2 c2 <=> x1 = x2 /\ y1 = y2 /\ c1 = c2)
```

This implies that, in practice, two *finvec* are equal if and only if their decomposition in the constructors `‘vecx‘`, `‘vec0‘` and `‘vec1‘` (it is the same for both the vectors) has exactly the same arguments.

Using the previous theorems, we can implement the automatic recursive procedure to compute the components of a concrete *finvec*. Let's show in details the formal calculation following the informal one presented before.

Let be `‘v:A^N‘` a *finvec* and suppose we want to calculate its i-th component `‘(v $. i):A‘` with `‘i < dimindex(:N)‘`. Since `‘:N‘` is a *fintype*, it is constructed, starting from the basic type `‘:1‘`, by subsequent applications of `‘tybit0‘` or `‘tybit1‘`. Suppose that the last application is `‘tybit1‘` (for `‘tybit0‘` things are similar). Therefore, the type `‘:N‘` is of the form `‘:(M)tybit1‘` and it implies that `‘v:A^(M)tybit1‘` is of the form `‘vec1 x y a‘` for some `‘x:A^M‘`, `‘y:A^M‘` and `‘a:A‘`. Now, we study the form of `‘i:num‘`. We have two possible cases:

1. `‘i‘` is odd, so it is of the form `‘BIT1 k‘` with `‘k < dimindex(:M)‘`. As shown before, the odd components of `‘v‘` are those of `‘x‘` so we restart recursively the procedure on the term `‘x $. k‘`;

2. `‘i‘` is even, so it is of the form `BIT0 k` with `‘k <= dimindex(:M)‘`. This produces two subcases:

   2.1 `‘k = dimindex(:M)‘`, so it is true that `‘i = BIT0 (dimindex(:M))‘` hence, by `VEC1_COMPONENTS_ARITH`, we can prove the term

   `‘vec1 x y a $. BIT0 (dimindex(:M)) = a‘`

   concluding the calculation,

   2.2 `‘k < dimindex(:M)‘` and, for the previous results, the even components of `‘v‘` are those of `‘y‘` so, we restart recursively the procedure on the term `‘y $. k‘`.

Note that, in case that `‘:N‘` is of the form `‘:(M)tybit0‘`, we haven't subcase 2.1 while everything else remains unchanged.

Repeating recursively this procedure we obtain, at the final step, necessarily a term of the form `‘vecx x $. _0‘` and it is easily proved to be equal to `‘x:A‘`. The procedure just described is implemented by the conversion `‘VEC_COMPONENT_CONV‘`.

As example of a concrete situation, we resume the *finvec* $\mathbf{v} = (v_0, v_1, v_2, v_3, v_4) \in A^5$. Its formal representation is

```
‘(vec1 (vec0 (vecx v3) (vecx v1)) (vec0 (vecx v2) (vecx v0)) v4):A^5‘
```

and suppose we want to calculate the term `‘v $. 3‘`, that is, the 3-th component of `‘v‘`. First of all, note that this is a well defined computation because `‘3 < dimindex(:5)‘`. Then, the *fintype* `‘(:5)‘` is constructed as `‘((:1)tybit0)tybit1‘` and we can consider the index `‘3‘` as `‘BIT1 (BIT1 _0)‘`. Hence, following the procedure, we have to calculate the term `‘vec0 (vecx v3) (vecx v1) $. BIT1 _0‘`. Repeating the procedure recursively on the latter term we obtain the term `‘vecx v3 $. _0‘` and, by the theorem `VECX_COMPONENT0`, it is proved to be equal to `‘v3‘`. Hence, it is proved the term `‘v $. 3 = v3‘`, as we expected.

In order to improve readability, we dedicate a specific syntax `‘<|v0;v1;...;vn|>‘` for *finvec*, nearest to the informal one. For example, the previous term

```
‘vec1 ((vec0 (vecx v3) (vecx v1)) (vec0 (vecx v2) (vecx v0)) v4)‘
```

is printed, in a more familiar form, as `‘<|v0; v1; v2; v3; v4|>‘`. Moreover, as said before, its components are computed by the conversion `VEC_COMPONENT_CONV` and we show some examples.

```
VEC_COMPONENT_CONV‘<|v0; v1; v2; v3; v4|> $. 0‘;;
val it : thm = |- <|v0; v1; v2; v3; v4|> $. 0 = v0

VEC_COMPONENT_CONV‘<|v0; v1; v2; v3; v4|> $. 2‘;;
val it : thm = |- <|v0; v1; v2; v3; v4|> $. 2 = v2

VEC_COMPONENT_CONV‘<|v0; v1; v2; v3; v4|> $. 5‘;;
Exception: Failure "MP: theorems do not agree".
```

Note that, if the index of the component that we want to compute is not in the right range (i.e. it is not less then the dimension of the vector), the conversion fails.

9.4 *Finvec* and quantum registers

Finally, in the context of quantum registers, the formal *fintype* theory allows us to consider concrete vectors with an arbitrary number of elements, that is, quantum registers with an arbitrary number of Qbits. Moreover, the *finvec* formal theory provides some useful tools to represent concrete vectors and to calculate their components. Practically, this allows us both to represent concrete boolean vectors `‘bs‘` that characterize the formal basis-states `‘qbasis bs‘` and to compute the state of the i-th Qbit in such states, that is, terms of the form `‘bs $. i‘`.

In conclusion, in our formalization of quantum computing, we use the vector datatype at two different levels, one to specify formally the basis vectors $|bs\rangle$, and the other to represent general states of quantum registers. We remind, in a brief summary, the notations used.

- ***Bit-strings:*** elements of type `‘:bool^N‘` (*finvec*) constructed by subsequent applications of the constructors `‘vecx‘`, `‘vec0‘` and `‘vec1‘`, depending on the form of the *fintype* `‘:N‘`. Such *finvec* are written with a special notation `‘<|b0;b1;...;bn|>`, their (0-based) indexing operator is `‘$.‘` and their components are computed automatically by the conversion `VEC_COMPONENT_CONV`.

  They are used to define the standard computational basis vectors `‘qbasis bs‘`. For example, in our formal setting, the basis vector (of a 2-Qbit register) $|01\rangle$ is represented by `‘qbasis <|F;T|>‘`.

  This representation is better then that of subsection 9.2.3, involving `‘vector‘` to write concrete boolean vectors, essentially for two reasons. The first is that the type `‘:bool^2‘` of the term `‘<|F;T|>‘` is inferred, by the HOL Light typechecker, from the form of the term itself.

  The second is that, in this representation, the components of `<|F;T|>`, that represent the states of the Qbits of the register in the basis state `‘qbasis <|F;T|>‘`, can be computed automatically by `VEC_COMPONENT_CONV`. In fact, the theorem

  ```
  VEC_COMPONENT_CONV ‘<|F;T|> $. 0‘
  val it : thm = |- <|F;T|> $. 0 = F
  ```

  is interpreted informally as: *the first (0-th) Qbit of a 2-Qbit register, in state* $|01\rangle$*, is in state* $|0\rangle$.

- ***Quantum registers***: elements of type '`:complex^(N)multivector`', that is, complex *multivectors* indexed by boolean vectors (*qvectors*). The indexing operator is represented by the overloaded symbol '`$$`'. In our setting, the quantum state

  $$|x\rangle = \sum_{bs} x_{bs} |bs\rangle$$

  is formalized by an element '`x:complex^(N)multivector`' and the above formulas is given by the following theorem

  ```
  QBASIS_EXPANSION
    |- !x. msum (:bool^N) (\bs. x $$ bs % qbasis bs) = x
  ```

  where '`x $$ bs`' and '`qbasis bs`' formalize x_{bs} and $|bs\rangle$ respectively.

9.4.1 Enumeration of boolean vectors and linear expansion of quantum registers

In case of sums over finite types of boolean vectors '`:bool^N`', it is important to have specific rewrites to expand them explicitly.

For example, if we specialize the theorem `QBASIS_EXPANSION` with a normalized vector `x:complex^(2)multivector`', (i.e. we consider a possible state of a register with 2 Qbits) we obtain the following theorem.

```
|- msum (:bool^2) (\bs. x $$ bs % qbasis bs) = x
```

However, we would be able to rewrite, automatically, this linear combination as

```
x $$ <|F; F|> % qbasis <|F; F|> +
x $$ <|F; T|> % qbasis <|T; F|>) +
x $$ <|T; F|> % qbasis <|F; T|> +
x $$ <|T; T|> % qbasis <|T; T|>.
```

Since every *fintype* is constructed, starting from the base type '`:1`', by subsequent applications of '`tybit0`' or '`tybit1`', it is sufficient to prove a theorem that describes the interaction of '`msum`' in these cases.

We give an informal idea. Let be $\{0,1\}^{2n+1}$ an odd-dimensional vector space of boolean vectors. Following our formal style we can write that $2n+1 = (n)$`tybit1`, so every vector $v \in \{0,1\}^{2n+1}$ is of the form $v = \mathrm{vec}_1(x,y,a)$ with $x, y \in \{0,1\}^n$ and $a \in \{0,1\}$. Then, for every function $f\colon \{0,1\}^{2n+1} \to \mathbb{C}^n$, the sum

$$\sum_{v\in\{0,1\}^{2n+1}} f(v)$$

can be split in the following triple sum.

$$\sum_{x\in\{0,1\}^n} \sum_{y\in\{0,1\}^n} \sum_{a\in\{0,1\}} f(\mathrm{vec}_1(x,y,a)) \tag{9.4.1}$$

A similar split holds for even-dimensional vector spaces of boolean vectors, in fact,

$$\sum_{v\in\{0,1\}^{2n}} f(v) = \sum_{x\in\{0,1\}^n} \sum_{y\in\{0,1\}^n} f(\mathrm{vec}_0(x,y)) \tag{9.4.2}$$

and, finally, it's clear that

$$\sum_{v\in\{0,1\}^1} f(v) = f(\mathrm{vec}_\mathrm{x}(0)) + f(\mathrm{vec}_\mathrm{x}(1)) = f(0) + f(1) = \sum_{v\in\{0,1\}} f(v) \tag{9.4.3}$$

where, with an abuse of notation, we consider $\{0,1\}$ and $\{0,1\}^1$ as the same.

The previous equations (9.4.2), (9.4.1) and (9.4.3) are proved formally in the following theorem.

```
BOOL_MSUM_FINVEC;;
  |- (!f. msum (:bool) f = f F + f T) /\
     (!f. msum (:bool^1) f = f <|F|> + f <|T|>) /\
     (!f. msum (:bool^(N)tybit0) f =
          msum (:bool^N) (\x. msum (:bool^N) (\y. f (vec0 x y)))) /\
     (!f. msum (:bool^(N)tybit1) f =
          msum (:bool) (\a. msum (:bool^N)
                        (\x. msum (:bool^N) (\y. f (vec1 x y a)))))
```

Note that, differently from the informal situation, the base cases with the types `:bool` and `:bool^1` must be considered separately because, from a formal point of view, they are different types.

Now, simply rewriting `BOOL_MSUM_FINVEC`, it is possible to expand the previous sum.

```
REWRITE_CONV[BOOL_MSUM_FINVEC]
   `msum (:bool^2) (\bs. x $$ bs % qbasis bs)`;;
val it : thm =
  |- msum (:bool^2) (\bs. x $$ bs % qbasis bs) =
     (x $$ <|F; F|> % qbasis <|F; F|> +
      x $$ <|T; F|> % qbasis <|T; F|>) +
      x $$ <|F; T|> % qbasis <|F; T|> +
      x $$ <|T; T|> % qbasis <|T; T|>
```

The conversion `QUANTUM_REGISTER_CONV` performs automatically such a rewriting as shown by the following example.

```
QUANTUM_REGISTER_CONV `x:complex^(3)multivector`;;
val it : thm =
  |- x =
     x $$ <|F; F; F|> % qbasis <|F; F; F|> +
     x $$ <|T; F; F|> % qbasis <|T; F; F|> +
     x $$ <|F; T; F|> % qbasis <|F; T; F|> +
     x $$ <|T; T; F|> % qbasis <|T; T; F|> +
     x $$ <|F; F; T|> % qbasis <|F; F; T|> +
     x $$ <|T; F; T|> % qbasis <|T; F; T|> +
     x $$ <|F; T; T|> % qbasis <|F; T; T|> +
     x $$ <|T; T; T|> % qbasis <|T; T; T|>
```

The same result is proved for the constants `sum` and `vsum`

```
BOOL_SUM_FINVEC;;
  |- (!f. sum (:bool) f = f F + f T) /\
     (!f. sum (:bool^1) f = f <|F|> + f <|T|>) /\
     (!f. sum (:bool^(N)tybit0) f =
          sum (:bool^N) (\x. sum (:bool^N) (\y. f (vec0 x y)))) /\
     (!f. sum (:bool^(N)tybit1) f =
          sum (:bool) (\a. sum (:bool^N)
                       (\x. sum (:bool^N) (\y. f (vec1 x y a)))))

BOOL_VSUM_FINVEC;;
  |- (!f. vsum (:bool) f = f F + f T) /\
     (!f. vsum (:bool^1) f = f <|F|> + f <|T|>) /\
     (!f. vsum (:bool^(N)tybit0) f =
          vsum (:bool^N) (\x. sum (:bool^N) (\y. f (vec0 x y)))) /\
     (!f. vsum (:bool^(N)tybit1) f =
          vsum (:bool) (\a. vsum (:bool^N)
                        (\x. vsum (:bool^N) (\y. f (vec1 x y a)))))
```

because it is necessary to rewrite explicitly the Hermitian product (or likewise the squared norm). The Hermitian product is a sum of complex numbers whereas the squared norm of a complex vector is a sum of real numbers, so they are represented formally using the operator `vsum` and `sum` respectively as shown by the following theorems.

```
QVECTOR_CDOT
  |- !x y. x cdot y =
           vsum (:bool^N) (\bs. x $$ bs * cnj (y $$ bs))
QUANTUM_SQUARED_NORM;;
  |- !v. norm v pow 2 =
         sum (:bool^N) (\bs. norm (v $$ bs) pow 2)
```

The conversion `QUANTUM_CDOT_CONV` performs calculations about the Hermitian product in concrete situations. Firstly, it expands the theorem `QVECTOR_CDOT` using the previous theorem `BOOL_VSUM_FINVEC` and, secondly, it makes the appropriate arithmetic simplifications.

For example, we can rewrite explicitly the Hermitian product of two general 2-dimensional *qvectors* `‘x‘,‘y‘` of type `‘:complex^(2)multivector‘`

```
QUANTUM_CDOT_CONV ‘x cdot y‘;;
  val it : thm =
  |- x cdot y = x $$ <|F;F|> * cnj (y $$ <|F;F|>) +
                x $$ <|F;T|> * cnj (y $$ <|F;T|>) +
                x $$ <|T;F|> * cnj (y $$ <|T;F|>) +
                x $$ <|T;T|> * cnj (y $$ <|T;T|>)
```

or, analogously, we can compute the Hermitian product of two concrete *qvectors*.

```
QUANTUM_CDOT_CONV
   ‘(a % qbasis <|F;F|> + b % qbasis <|T;F|>) cdot
    (a % qbasis <|F;F|> + b % qbasis <|T;T|>)‘;;
  val it : thm =
  |- (a % qbasis <|F; F|> + b % qbasis <|T; F|>) cdot
     (a % qbasis <|F; F|> + b % qbasis <|T; T|>) =
     Cx (norm a) pow 2
```

Now, we have the background to define, in our formal framework, the main logical quantum gates and then to certify some simple, but significant, quantum algorithms and protocols.

Chapter 10

Quantum logic gates

We have seen the quantum description of the states of a register with n Qbits, let's see now how these states evolve, giving rise to a quantum computation.

Like a classical computer, a quantum computer is formed from quantum circuits made up of elementary quantum logic gates. A quantum gate (or quantum logic gate) is a basic quantum circuit operating on a small number of Qbits. They are the building blocks of quantum circuits, like classical logic gates are for conventional digital circuits. Unlike many classical logic gates, quantum logic gates are reversible (the computational process can be performed in both verses).

In the classical case there is a single logical (non trivial) one bit port, the NOT gate, which implements the logical negation operation defined by a table of truth where $1 \to 0$ and $0 \to 1$. To define a similar operation on a Qbit, we cannot limit ourselves to establish its action on the basis states $|0\rangle$ and $|1\rangle$, but we must specify also how a superposition of states $|0\rangle$ and $|1\rangle$ must be transformed.

Reversible operations that a quantum computer can perform upon a single Qbit (1-Qbit gates) are represented by the action on the state of the Qbit of any unitary linear operator of the space of states spanned by $\{|0\rangle, |1\rangle\}$. Therefore, a 1-Qbit gate is, in fact, an Hermitian operator $f\colon \mathbb{C}^2 \to \mathbb{C}^2$. By the identification of $\otimes^n\mathbb{C}$ with $\mathbb{C}^{2^n}$, in case of a quantum register with n Qbits, a quantum gate is an Hermitian operator $f\colon \mathbb{C}^{2^n} \to \mathbb{C}^{2^n}$, that is, in our formal framework, a linear and unitary function of type `‘:complex^(N)multivector->complex^(N)multivector‘`.

Moreover, every quantum gates can be defined on the basis states and then extended, by linearity, to the whole space of states. For example the **X** (Quantum NOT) gate can be defined as

$$\mathbf{X}\,|0\rangle = |1\rangle \qquad \mathbf{X}\,|1\rangle = |0\rangle$$

and then can be extended to $\mathbb{C}^2$ by linearity

$$\mathbf{X}(\alpha\,|0\rangle + \beta\,|1\rangle) = \alpha\mathbf{X}\,|0\rangle + \beta\mathbf{X}\,|1\rangle = \alpha\,|1\rangle + \beta\,|0\rangle\,.$$

Analogously, in the case of a register, we can define logical gates on the standard computational basis ($|bs\rangle$) and then extend it, by linearity, on the whole space $\mathbb{C}^{2^n}$. Such an extension result is a basic fact in the theory of linear algebra, but here we consider the form specialized for our computational basis. In particular, since this basis is identified by the set of strings of n bits, that is $\{0,1\}^n$, given any function $f\colon \{0,1\}^n \to \mathbb{C}^{2^n}$ we can extend it, by linearity, to a function $g\colon \mathbb{C}^{2^n} \to \mathbb{C}^{2^n}$ defined on the standard computational basis as

$$g(|bs\rangle) = f(bs) \qquad \text{for all } bs \in \{0,1\}^n.$$

This fundamental property is formalized in the following theorem.

```
CLINEAR_QBASIS_EXTEND
  |- !f. ?g. clinear g /\ (!bs. g (qbasis bs) = f bs)
```

Thanks to this, we can define quantum gates simply giving their action on the computational basis and then proving the existence of the related linear function defined on the whole space of states.

Although there are many possible quantum gates (every Hermitian operator over the space of states of a quantum register is an admissible quantum gate), we decided to formalize only the following gates:

- the quantum *NOT* gate **X** (1-Qbit gate),
- the quantum controlled *NOT* gate **cX** (2-Qbit gate),
- the quantum *SHIFT* gate **Z** (1-Qbit gate),
- the quantum *HADAMARD* gate **H** (1-Qbit gate),
- the quantum *PHASE-SHIFT* gate $\mathbf{R}_\phi$ (1-Qbit gate)

because they are a simple set of gates that form a set of **universal quantum gates**, that is, a set of gates to which any operation possible on a quantum computer can be reduced ([Nielsen and Chuang, 2000]). In other words, any other unitary operation can be expressed as a sequence of gates from this set.

10.1 Quantum NOT gate

As said before, the quantum *NOT* gate (**X**) is the quantum generalization of the classical 1-Cbit gate *NOT*. Thus, it is the linear function, over the space of states of a single Qbit, that acts on the computational basis as

$$\mathbf{X}\left|i\right\rangle = \left|\neg i\right\rangle \qquad \text{for all } i \in \{0,1\} \tag{10.1.1}$$

where $\neg$ is the classical negation over booleans that is, $\neg 0 = 1$ and $\neg 1 = 0$.

In the case of a quantum register, with n Qbits, we defined the function $\mathbf{X}_i$ as the linear function that applies the **X** gate on the i-th Qbit of the register ($i < n$). Practically, it acts on the computational basis by the following rule

$$\mathbf{X}_i\left|bs\right\rangle = \mathbf{X}_i\left|b_0 b_1 \dots b_i \dots b_{n-1}\right\rangle = \left|b_0 b_1 \dots \neg b_i \dots b_{n-1}\right\rangle. \tag{10.1.2}$$

First of all, we define the auxiliary function `BITS_NOT:num->bool^N->bool^N`

```
let BITS_NOT = define
  `BITS_NOT i (x:bool^N):bool^N =
  lambda0 j. if i = j then ~(x$.j) else x$.j`;;
```

that performs negation on the i-th components of a boolean vector. Since `BITS_NOT` acts on a boolean *finvec* (`bs:bool^N`) depending on a natural number (the index of the component on which is applied the negation), we can compute its action, with a strategy similar to that of `VEC_COMPONENT_CONV`, looking at the parity of the index considered. In fact, by the construction of *finvec* presented in the previous chapter (section 9.3), the following computation rules holds, for all $x, y \in \{0,1\}^n$ and $a \in \{0,1\}$.

$$\text{BITS\_NOT}_0(\text{vec}_\text{x}(a)) = \text{vec}_\text{x}(\neg a)$$

$$\text{BITS\_NOT}_i(\text{vec}_1(x,y,a)) = \begin{cases} \text{vec}_1(x, \text{BITS\_NOT}_{\frac{i}{2}}(y), a), & \text{if } i \text{ is even} \\ \text{vec}_1(\text{BITS\_NOT}_{\frac{i-1}{2}}(x), y, a), & \text{if } i \text{ is odd} \\ \text{vec}_1(x, y, \neg a), & \text{if } i = 2n \end{cases}$$

$$\text{BITS\_NOT}_i(\text{vec}_0(x,y)) = \begin{cases} \text{vec}_0(x, \text{BITS\_NOT}_{\frac{i}{2}}(y)), & \text{if } i \text{ is even} \\ \text{vec}_0(\text{BITS\_NOT}_{\frac{i-1}{2}}(x), y), & \text{if } i \text{ is odd} \end{cases}$$

We recall that, in the HOL Light formalism, a natural number i is even (odd) if and only if it is of the form `‘BIT0 n‘` (`‘BIT1 n‘`) and, in this case, the numeral `‘n‘` is the formal counterpart of $\frac{i}{2}$ ($\frac{i-1}{2}$). Therefore, the latter equations are summarized in the next formal theorem.

```
ARITH_BITS_NOT_BIN
  |- (!x v. BITS_NOT (NUMERAL x) v = BITS_NOT x v) /\
     (!x. BITS_NOT _0 (vec x) = vecx (~x)) /\
     (!x y. BITS_NOT _0 (vec0 x y) = vec0 x (BITS_NOT _0 y)) /\
     (!x y a. BITS_NOT _0 (vec1 x y a) = vec1 x (BITS_NOT _0 y) a) /\
     (!i x y.
          BITS_NOT (BIT1 i) (vec0 x y) = vec0 (BITS_NOT i x) y /\
          BITS_NOT (BIT0 i) (vec0 x y) = vec0 x (BITS_NOT i y)) /\
     (!i x y a.
          BITS_NOT (BIT1 i) (vec1 x y a) = vec1 (BITS_NOT i x) y a /\
          BITS_NOT (BIT0 i) (vec1 x y a) =
          (if i = dimindex (:N) then vec1 x y (~a)
           else vec1 x (BITS_NOT i y) a))
```

The function `‘BITS_NOT‘` is an involution, that is, if it is applied twice on the same Qbit of the register, then it works as the identity function as shown by the following HOL theorem.

```
BITS_NOT_BITS_NOT
  |- !bs i. BITS_NOT i (BITS_NOT i bs) = bs
```

The conversion `BITS_NOT_CONV`, defined essentially by rewriting `ARITH_BITS_NOT`, computes terms of the form `‘BITS_NOT i x‘` as shown in the following example.

```
BITS_NOT_CONV ‘BITS_NOT 1 <|T;T;T;T|>‘;;
val it : thm = |- BITS_NOT 1 <|T; T; T; T|> = <|T; F; T; T|>
```

At this point, the generalized gate $\mathbf{X}_i$ is the linear function `‘QNOT‘`, if it exists, of type

`‘:num->complex^(N)multivector->complex^(N)multivector‘`

such that acts on the standard computational basis as

`‘QNOT i (qbasis bs) = qbasis (BITS_NOT i bs)‘`

Using the theorem `CLINEAR_QBASIS_EXTEND` specialized with the function

`‘\bs:bool^N. qbasis (BITS_NOT i bs)‘`

we can easily prove that such a function exists and we can call it `‘QNOT‘`. The next statement, proved in HOL Light, defines formally the gate $\mathbf{X}_i$.

```
QNOT
  |- (!i. clinear (QNOT i)) /\
     (!i bs:bool^N. QNOT i (qbasis bs) = qbasis(BITS_NOT i bs))
```

The quantum *NOT* gate is an involution, in fact it holds that

$$\mathbf{X}_i(\mathbf{X}_i(v)) = v \qquad \text{for all } v \in \mathbb{C}^{2^n}$$

and, as required by quantum mechanics, it is unitary (i.e. it preserves the Hermitian product)

$$\langle v|w\rangle = \langle \mathbf{X}_i(v)|\mathbf{X}_i(w)\rangle\,.$$

The related formal theorems proved in HOL Light are the following.

```
QNOT_QNOT
  |- !v i. QNOT i (QNOT i v) = v

UNITARY_QNOT
  |- !i w v. v cdot w = QNOT i v cdot QNOT i w
```

In order to compute automatically terms of the form `‘QNOT i x‘` we write the conversion `QNOT_CONV` that acts, essentially, by following this simple strategy:

- the state `‘x:complex^(N)multivector‘` is rewritten as linear combination of the standard computational basis using `QUANTUM_REGISTER_CONV`,
- the function `‘QNOT‘` is computed by linearity on the basis vectors and all the necessary arithmetic simplifications are done,
- the conversion `BITS_NOT_CONV` is used inside the term to compute all the subterms `‘BITS_NOT bs‘` resulting from the application of `‘QNOT‘` on the basis states `‘qbasis bs‘`.

We show some examples.

```
QNOT_CONV ‘QNOT 0 (x:complex^(4)multivector)‘;;
val it : thm =
  |- QNOT 0 x =
     x $$ <|F; F; F; F|> % qbasis <|T; F; F; F|> +
     x $$ <|T; F; F; F|> % qbasis <|F; F; F; F|> +
     x $$ <|F; F; T; F|> % qbasis <|T; F; T; F|> +
     x $$ <|T; F; T; F|> % qbasis <|F; F; T; F|> +
     x $$ <|F; T; F; F|> % qbasis <|T; T; F; F|> +
     x $$ <|T; T; F; F|> % qbasis <|F; T; F; F|> +
     x $$ <|F; T; T; F|> % qbasis <|T; T; T; F|> +
     x $$ <|T; T; T; F|> % qbasis <|F; T; T; F|> +
     x $$ <|F; F; F; T|> % qbasis <|T; F; F; T|> +
     x $$ <|T; F; F; T|> % qbasis <|F; F; F; T|> +
     x $$ <|F; F; T; T|> % qbasis <|T; F; T; T|> +
     x $$ <|T; F; T; T|> % qbasis <|F; F; T; T|> +
     x $$ <|F; T; F; T|> % qbasis <|T; T; F; T|> +
     x $$ <|T; T; F; T|> % qbasis <|F; T; F; T|> +
     x $$ <|F; T; T; T|> % qbasis <|T; T; T; T|> +
     x $$ <|T; T; T; T|> % qbasis <|F; T; T; T|>

QNOT_CONV ‘QNOT 1 (a % qbasis <|T;F;T|> + b % qbasis <|T;T;T|>)‘;;
val it : thm =
  |- QNOT 1 (a % qbasis <|T; F; T|> + b % qbasis <|T; T; T|>) =
     a % qbasis <|T; T; T|> + b % qbasis <|T; F; T|>
```

10.1.1 Quantum Controlled NOT gate

The quantum *controlled NOT* gate (**cX**) derives from the **X** gate but differs from this because it is a 2-Qbit port instead of a 1-Qbit port. The **cX** gate applies the **X** gate on a Qbit (called *target*) depending on the state of another Qbit (called *control*). If the state of the control is $|1\rangle$, then the *NOT* operation is performed on the target. Otherwise, nothing is done. Obviously, **cX** is a linear operator and, by this definition, acts on the 2-Qbits standard computational basis as follows (we consider the first Qbit as the control and the second as the target)

$$\mathbf{cX}\,|ij\rangle = \begin{cases} |ij\rangle\,, & \text{if } i=0 \\ |i\neg j\rangle\,, & \text{if } i=1 \end{cases} \qquad \forall i,j\in\{0,1\}$$

therefore

$$\mathbf{cX}\,|00\rangle = |00\rangle \qquad \mathbf{cX}\,|01\rangle = |01\rangle$$
$$\mathbf{cX}\,|10\rangle = |11\rangle \qquad \mathbf{cX}\,|11\rangle = |10\rangle$$

Generalizing to an arbitrary quantum register with n Qbits, the gate $\mathbf{cX}_{ij}$ (with $i, j < n$ and $i \neq j$) applies the **cX** gate to the couple of the i-th (control) and j-th (target) Qbit of the register. Hence, we have that $\mathbf{cX}_{ij}$ acts on the standard computational basis (supposing $i < j$) as follows.

$$\mathbf{cX}_{ij}\,|bs\rangle = \mathbf{cX}_{ij}\,|b_0 \dots b_i \dots b_j \dots b_{n-1}\rangle = \begin{cases} |b_0 \dots b_i \dots b_j \dots b_{n-1}\rangle\,, & \text{if } b_i = 0 \\ |b_0 \dots b_i \dots \neg b_j \dots b_{n-1}\rangle\,, & \text{if } b_i = 1 \end{cases} \tag{10.1.3}$$

Things are similar if $j < i$. Again using `CLINEAR_QBASIS_EXPAND`, we prove the existence of a linear function, that we call ‘QCNOT‘, that acts on the standard basis vectors according to equation (10.1.3).

```
QCNOT
  |- (!i j. clinear (QCNOT i j)) /\
     (!i j bs. QCNOT i j (qbasis bs) =
               qbasis (if bs $. i then BITS_NOT j bs else bs))
```

Also ‘QCNOT‘ is a unitary transformation, that is, $\langle v|w\rangle = \langle \mathbf{cX}_{ij}(v)|\mathbf{cX}_{ij}(w)\rangle$ for all $i \neq j \in \{0, \cdots, n-1\}$ and $v,w \in \mathbb{C}^{2^n}$.

```
UNITARY_QCNOT : thm =
  |- !i j v w. i < dimindex (:N) /\ j < dimindex (:N) /\ ~(i = j)
               ==> v cdot w = QCNOT i j v cdot QCNOT i j w
```

The related conversion `QCNOT_CONV` performs calculations about it, that is, computes term of the form ‘QNOT i j v‘. We give some examples.

```
QCNOT_CONV ‘QCNOT 2 4 (qbasis <|F;T;T;F;T|>)‘;;
val it : thm =
  |- QCNOT 2 4 (qbasis <|F; T; T; F; T|>) = qbasis <|F; T; T; F; F|>

QCNOT_CONV ‘QCNOT 2 1 (a % qbasis <|F;T;T|> + b % qbasis <|T;T;F|>)‘;;
val it : thm =
  |- QCNOT 2 1 (a % qbasis <|F; T; T|> + b % qbasis <|T; T; F|>) =
     b % qbasis <|T; T; F|> + a % qbasis <|F; F; T|>
```

10.2 Quantum SHIFT gate

The quantum *SHIFT* gate ($\mathbf{Z}$), or Pauli-Z gate, is a 1-Qbit gate that maps the base state $|1\rangle$ to $-|1\rangle$ and leaves the base state $|0\rangle$ unchanged. Therefore, by linearity, it acts on a general 1-Qbit state as

$$\mathbf{Z}(\alpha\,|0\rangle + \beta\,|1\rangle) = \alpha\,|0\rangle - \beta\,|1\rangle\,.$$

The generalized gate $\mathbf{Z}_i$ performs, again by linearity, the $\mathbf{Z}$ gate on the i-th Qbit of a quantum register with $n > i$ Qbits. Hence, on the standard computational basis it acts as follows.

$$\mathbf{Z}_i\,|bs\rangle = \mathbf{Z}_i\,|b_0 \dots b_i \dots b_{n-1}\rangle = \begin{cases} |bs\rangle\,, & \text{if } b_i = 0 \\ -\,|bs\rangle\,, & \text{if } b_i = 1 \end{cases} \tag{10.2.1}$$

Following the latter equation, the related formalization can be easily given, without any auxiliary function, proving the following theorem.

```
QSHIFT
  |- (!i. clinear (QSHIFT i)) /\
     (!i bs. QSHIFT i (qbasis bs) =
             (if bs $. i then --qbasis bs else qbasis bs))
```

The existence of the linear function ‘QSHIFT‘ of the latter theorem is proved again by specializing `CLINEAR_QBASIS_EXTEND` with the following function over boolean vectors.

```
‘\bs:bool^N. if bs $. i then --qbasis bs else qbasis bs‘
```

As before, the $\mathbf{Z}_i$ gate is unitary in fact, it holds that

$$\langle v|w\rangle = \langle \mathbf{Z}_i(v)|\mathbf{Z}_i(w)\rangle \qquad \text{for all} i \in \{0, \cdots, n-1\} \text{ and } v, w \in \mathbb{C}^{2^n}$$

and it is an involution, that is,

$$\mathbf{Z}_i(\mathbf{Z}_i(v)) = v \qquad \text{for all } v \in \mathbb{C}^{2^n}.$$

The related formal theorems are the following.

```
UNITARY_QSHIFT
  |- !i w v. v cdot w = QSHIFT i v cdot QSHIFT i w

QSHIFT_QSHIFT
  |- !i v. QSHIFT i (QSHIFT i v) = v
```

The conversion `QSHIFT_CONV` performs calculations about `‘QSHIFT‘` by following a strategy very similar to that explained for `QNOT_CONV`, except for the use of `BITS_NOT_CONV` since `‘BITS_NOT‘` doesn't appear in the definition of `‘QSHIFT‘`. We give some examples.

```
QSHIFT_CONV ‘QSHIFT 0 (x:complex^(2)multivector)‘;;
val it : thm =
  |- QSHIFT 0 x = x $$ <|F; F|> % qbasis <|F; F|> +
                  --(x $$ <|T; F|>) % qbasis <|T; F|> +
                  x $$ <|F; T|> % qbasis <|F; T|> +
                  --(x $$ <|T; T|>) % qbasis <|T; T|>

QSHIFT_CONV ‘QSHIFT 2 (a % qbasis <|F;T;T|> - b % qbasis <|T;T;T|>)‘;;
val it : thm =
  |- QSHIFT 2 (a % qbasis <|F; T; T|> - b % qbasis <|T; T; T|>) =
     --a % qbasis <|F; T; T|> + b % qbasis <|T; T; T|>
```

The **Z** gate is a particular case of a more general gate, the quantum phase *SHIFT* gate ($\mathbf{R}_\phi$), that, given a real number ϕ, maps again $|0\rangle$ in $|0\rangle$ and $|1\rangle$ in $e^{i\phi}|1\rangle$. On a general 1-Qbit state we have that

$$\mathbf{R}_\phi(\alpha|0\rangle + \beta|1\rangle) = \alpha|0\rangle + \beta e^{i\phi}|1\rangle$$

and, in the case of a register with n Qbits, it acts on the basis vectors by the following rule.

$$\mathbf{R}_{(i,\phi)}|bs\rangle = \mathbf{R}_{(i,\phi)}|b_0 \dots b_i \dots b_{n-1}\rangle = \begin{cases} |bs\rangle, & \text{if } b_i = 0 \\ e^{i\phi}|bs\rangle, & \text{if } b_i = 1 \end{cases} \tag{10.2.2}$$

Note that $\mathbf{Z}_i = \mathbf{R}_{(i,\pi)}$ for all $i \in \{0, \dots n-1,\}$.

We define formally a more general function

```
‘QPHASE_SHIFT:num->complex->complex^(N)multivector->complex^(N)multivector‘
```

as before

```
QPHASE_SHIFT
  |- (!i z. clinear (QPHASE_SHIFT i z)) /\
     (!i z bs. QPHASE_SHIFT i z (qbasis bs) =
               (if bs $. i then cexp (ii * z) % qbasis bs else qbasis bs))
```

and then we prove that it is effectively an admissible quantum gate (i.e. a unitary transformation) in the case that `‘z:complex‘` is real, that is, it is of the form `‘Cx x‘` for some `‘x:real‘`. The HOL theorem is the following.

```
UNITARY_QPHASE_SHIFT
  |- !i x w v.
       v cdot w = QPHASE_SHIFT i (Cx x) v cdot QPHASE_SHIFT i (Cx x) w
```

Again, the conversion `QPHASE_SHIFT_CONV` performs calculations automatically as follows.

```
QPHASE_SHIFT_CONV ‘QPHASE_SHIFT 2 (Cx(&2))
                    (qbasis <|T;T;F|> + qbasis <|T;F;T|>)‘;;
val it : thm =
 |- QPHASE_SHIFT 2 (Cx (&2)) (qbasis <|T; T; F|> + qbasis <|T; F; T|>) =
     qbasis <|T; T; F|> + cexp (ii * Cx (&2)) % qbasis <|T; F; T|>

QPHASE_SHIFT_CONV ‘QPHASE_SHIFT 2 (Cx(pi))
                   (qbasis <|T;T;F|> + qbasis <|T;F;T|>)‘;;
val it : thm =
 |- QPHASE_SHIFT 2 (Cx pi) (qbasis <|T; T; F|> + qbasis <|T; F; T|>) =
    qbasis <|T; T; F|> + --qbasis <|T; F; T|>
```

10.3 Quantum HADAMARD gate

The quantum *HADAMARD* gate (**H**) is a 1-Qbit linear transformation that maps the base states $|0\rangle$ and $|1\rangle$ in superposed states with equal coefficients. More precisely, it acts on the 1-Qbit basis states by the following rules

$$\mathbf{H}\,|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$

$$\mathbf{H}\,|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

and its generalization $\mathbf{H}_i$, to a quantum register with n Qbits, performs the **H** gate on the i-th Qbit (for all $i < n$). Thus, for the standard computational basis $|bs\rangle$, we have that

$$\mathbf{H}_i\,|bs\rangle = \mathbf{H}_i\,|b_0 \dots b_i \dots b_{n-1}\rangle = \begin{cases} \frac{1}{\sqrt{2}}(|b_0 \dots 0 \dots b_{n-1}\rangle + |b_0 \dots 1 \dots b_{n-1}\rangle), & \text{if } b_i = 0 \\ \frac{1}{\sqrt{2}}(|b_0 \dots 0 \dots b_{n-1}\rangle - |b_0 \dots 1 \dots b_{n-1}\rangle), & \text{if } b_i = 1 \end{cases} \tag{10.3.1}$$

Defining the auxiliary function $\mathrm{QSET}_{(i,c)} \colon \{0,1\}^n \to \{0,1\}^n$, that sets to $c \in \{0,1\}$ the value of the i-th bit of a string with n bits, that is,

$$\mathrm{QSET}_{(i,c)}(b_0 \dots b_i \dots b_{n-1}) = b_0 \dots c \dots b_{n-1} \qquad \text{for all } c \in \{0,1\},\ i < n \tag{10.3.2}$$

we can rewrite equation (10.3.1) as

$$\mathbf{H}_i\,|bs\rangle = \mathbf{H}_i\,|b_0 \dots b_i \dots b_{n-1}\rangle = \begin{cases} \frac{1}{\sqrt{2}}(|\mathrm{QSET}_{(i,0)}(bs)\rangle + |\mathrm{QSET}_{(i,1)}(bs)\rangle), & \text{if } b_i = 0 \\ \frac{1}{\sqrt{2}}(|\mathrm{QSET}_{(i,0)}(bs)\rangle - |\mathrm{QSET}_{(i,1)}(bs)\rangle), & \text{if } b_i = 1 \end{cases} \tag{10.3.3}$$

First of all, we formalize the auxiliary function QSET giving the following formal definition.

```
let QSET = define
 ‘QSET i (c:bool) (x:bool^N):bool^N =
  lambda0 j. if i = j then c else x$.j‘;;
```

As in the case of BITS_NOT, the function QSET acts on boolean *finvec* depending on the index $i \in \mathbb{N}$, thus the computation is performed looking at the parity of such index. This implies that, for all $x, y \in \{0,1\}^n$ and $a \in \{0,1\}$, properties similar to those of BITS_NOT in section 10.1 hold.

$$\mathrm{QSET}_{(0,c)}(\mathrm{vec}_{\mathrm{x}}(c)) = \mathrm{vec}_{\mathrm{x}}(c)$$

$$\mathrm{QSET}_{(0,c)}(\mathrm{vec}_1(x,y,a)) = \begin{cases} \mathrm{vec}_1(x, \mathrm{QSET}_{(\frac{i}{2},c)}(y), a), & \text{if } i \text{ is even} \\ \mathrm{vec}_1(\mathrm{QSET}_{(\frac{i-1}{2},c)}(x), y, a), & \text{if } i \text{ is odd} \\ \mathrm{vec}_1(x,y,c), & \text{if } i = 2n \end{cases}$$

$$\mathrm{QSET}_{(0,c)}(\mathrm{vec}_0(x,y)) = \begin{cases} \mathrm{vec}_0(x, \mathrm{QSET}_{(\frac{i}{2},c)}(y)), & \text{if } i \text{ is even} \\ \mathrm{vec}_0(\mathrm{QSET}_{(\frac{i-1}{2},c)}(x), y), & \text{if } i \text{ is odd} \end{cases}$$

As in the case of `‘BITS_NOT‘`, the latter equations are formalized in the following HOL theorem, in the same way of the previous `ARITH_BITS_NOT_BIN`.

```
ARITH_QSET_BIN : thm =
  |- (!x c v. QSET (NUMERAL x) c v = QSET x c v) /\
     (!x c. QSET _0 c (vecx x) = vecx c) /\
     (!x y c. QSET _0 c (vec0 x y) = vec0 x (QSET _0 c y)) /\
     (!c x y a.
       QSET _0 c (vec1 x y a) = vec1 x (QSET _0 c y) a) /\
     (!i c x y.
       QSET (BIT1 i) c (vec0 x y) = vec0 (QSET i c x) y /\
       QSET (BIT0 i) c (vec0 x y) = vec0 x (QSET i c y)) /\
     (!i c x y a.
       QSET (BIT1 i) c (vec1 x y a) = vec1 (QSET i c x) y a /\
       QSET (BIT0 i) c (vec1 x y a) =
        (if i = dimindex (:N) then vec1 x y c else vec1 x (QSET i c y) a))
```

The conversion `QSET_CONV` is defined to compute terms of the form `‘QSET i v‘` and the strategy that it follows is similarly to that of `BITS_NOT_CONV`, it rewrites essentially the theorem `ARITH_QSET_BIN`. We show an example.

```
QSET_CONV ‘QSET 3 F <|T;T;T;T|>‘;;
val it : thm = |- QSET 3 F <|T; T; T; T|> = <|T; T; T; F|>
```

Using the function `‘QSET‘`, we can formalize the *HADAMARD* gate proving, by specializing again `CLINEAR_QBASIS_EXTEND`, the existence of a linear function `‘QHADAMARD‘` that satisfy definition (10.3.3). The formal statement, proved in HOL Light, is the following.

```
QHADAMARD
  |- (!i. clinear (QHADAMARD i)) /\
     (!i bs. QHADAMARD i (qbasis bs) =
             inv (Cx (sqrt (&2))) %
              (if bs $. i then qbasis (QSET i F bs) - qbasis (QSET i T bs)
               else qbasis (QSET i F bs) + qbasis (QSET i T bs)))
```

By the properties of `‘QSET‘`, we can prove formally that the latter is a unitary involution.

```
QHADAMARD_QHADAMARD
  |- !i v. i < dimindex (:N)
         ==> QHADAMARD i (QHADAMARD i v) = v

UNITARY_QHADAMARD
  |- !i w v. i < dimindex (:N)
             ==> v cdot w = QHADAMARD i v cdot QHADAMARD i w
```

Moreover, the conversion `QHADAMARD_CONV` performs calculations about the **H** gate by following a strategy very similar to that explained for `QNOT_CONV`, except for the use of `QSET_CONV`, instead of `BITS_NOT_CONV`, at the last step of the procedure. We show some examples of computations.

```
QHADAMARD_CONV ‘QHADAMARD 0 (qbasis <|F;T|>)‘;;
val it : thm =
  |- QHADAMARD 0 (qbasis <|F; T|>) =
     inv (Cx (sqrt (&2))) % qbasis <|F; T|> +
     inv (Cx (sqrt (&2))) % qbasis <|T; T|>

QHADAMARD_CONV ‘QHADAMARD 0 (qbasis <|F;T|> + qbasis <|T;T|>)‘;;
val it : thm =
  |- QHADAMARD 0 (qbasis <|F; T|> + qbasis <|T; T|>) =
     inv (Cx (sqrt (&2))) % qbasis <|F; T|> +
     inv (Cx (sqrt (&2))) % qbasis <|T; T|> +
     inv (Cx (sqrt (&2))) % qbasis <|F; T|> -
     inv (Cx (sqrt (&2))) % qbasis <|T; T|>
```

10.4 Collapsed states and the quantum BORN RULE

The last operation that we deal with is measurement. This is the only one that can't be represented by an Hermitian operator because it involves the collapse of the state subjected to measurement.

From the third postulate of quantum mechanics we have that, given an arbitrary 1-Qbit state

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$$

with $|\alpha|^2 + |\beta|^2 = 1$, the probabilities that, after a measurement, the Qbit is in state $|0\rangle$ or $|1\rangle$ are given by $|\alpha|^2$ and $|\beta|^2$ respectively. In other words, we can say that the state of the Qbit collapses, after the measurement, in state $|0\rangle$ with probability $|\alpha|^2$ and in state $|1\rangle$ with probability $|\beta|^2$.

The same holds in case of a quantum register with n Qbits. Given an arbitrary state of the register

$$|\psi\rangle = \sum_{bs} \alpha_{bs}\,|bs\rangle$$

with the normalization condition

$$\sum_{bs} |\alpha_{bs}|^2 = 1$$

we have that, for every basis state $|bs\rangle = |b_0 \dots b_{n-1}\rangle$, the squared modulus $|\alpha_{bs}|^2$ represents the probability that the register is in state $|bs\rangle$ (i.e. the i-th Qbit of the register is in state $|b_i\rangle$ for all $i = 0 \dots n-1$) after a measurement on the whole register.

Moreover, given a register in the previous state $|\psi\rangle$, we can make a measurement only on a single Qbit. Clearly, if the result of such a measurement is $b \in \{0, 1\}$ then the whole register, immediately after the measurement, collapses in the renormalized state

$$|\psi\rangle_{b_i=b} = \frac{\sum\limits_{bs \in \{b_0 \dots b_i \dots b_{n-1} \mid b_i = b\}} \alpha_{bs}\,|bs\rangle}{\sqrt{\sum\limits_{bs \in \{b_0 \dots b_i \dots b_{n-1} \mid b_i = b\}} |\alpha_{bs}|^2}}. \tag{10.4.1}$$

that is a superposition of the basis states in which the i-th Qbit is in state $|b\rangle$. The question arises: which is the probability that it happens?

In this case, referring to the formalism of the density operator of quantum mechanics (whose presentation is far from the goal of this thesis) we can consider the event that after the measurement on the i-th Qbit, such a Qbit is in state $|b\rangle$, as the union of all the incompatible events that after a measurement on the whole register, it is in state $|bs\rangle$ such that $b_i = b$. Each of the latter have probability equal to $|\alpha_{bs}|^2$ so, by basic probability theory[1], the probability

[1]The probability of an union of incompatible events is the sum of the related probabilities $P(\cup_{a\in A}) = \sum\limits_{a\in A} P(a)$.

considered is

$$P(|\psi\rangle, b_i = b) = \sum_{bs \in \{b_0 \ldots b_i \ldots b_{n-1} \mid b_i = b\}} |\alpha_{bs}|^2. \tag{10.4.2}$$

Since the normalization condition of quantum states and the fact that $b_i = b$ or $b_i = \neg b$, it is obvious that

$$P(|\psi\rangle, b_i = b) + P(|\psi\rangle, b_i = \neg b) = \sum_{bs} |\alpha_{bs}|^2 = 1. \tag{10.4.3}$$

As example, for a generic 2-Qbits state

$$|\psi\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle$$

the probability that, after a measurement on the first Qbit (0-th Qbit), it is in state $|0\rangle$ is

$$P(|\psi\rangle, b_0 = 0) = |\alpha_{00}|^2 + |\alpha_{01}|^2$$

since only the basis states $|00\rangle$ and $|01\rangle$ satisfy the condition $b_0 = 0$. If the result of the measurement on the first Qbit is 0 , then, immediately after the measurement, the whole system collapses in the state

$$|\psi\rangle_{b_0=0} = \frac{\alpha_{00}|00\rangle + \alpha_{01}|01\rangle}{\sqrt{|\alpha_{00}|^2 + |\alpha_{01}|^2}}.$$

In our formal framework, the functions

```
`QUANTUM_COLLAPSE:num->bool->complex^(N)multivector->complex^(N)multivector`

`BORN_RULE:num->bool->complex^(N)multivector->real`
```

are defined such that, for every state of a quantum register `‘v:complex^(N)multivector‘`, index `‘i:num‘` and boolean `‘c:bool‘`, the term

```
`QUANTUM_COLLAPSE i c v:complex^(N)multivector`
```

represents the collapsed state of the register after that the measurement, on the Qbit indexed by `‘i:num‘`, has given the result `‘c:bool‘`, whereas the term

```
`BORN_RULE i c v:real`
```

is the probability that the state `‘v‘` collapses in the state `‘QUANTUM_COLLAPSE i c v‘` after the measurement. Therefore, following equations (10.4.1) and (10.4.2) respectively, their formal definitions are given as follows. The constant `‘mat 0‘` represents the zero vector of a complex vector space.

```
let QUANTUM_COLLAPSE = new_definition
  `QUANTUM_COLLAPSE i c (v:complex^(N)multivector) =
   msum (:bool^N) (\bs. if bs $. i = c then v$$bs % qbasis bs else mat 0)`;;

let BORN_RULE = define
  `BORN_RULE i c (v:complex^(N)multivector) =
   sum (:bool^N) (\bs. if bs$.i = c then norm (v$$bs) pow 2 else &0)`;;
```

Property (10.4.3) is formalized by the following theorem.

```
BORN_RULE_CASES
  |- !i c v.
       norm v pow 2 = &1 ==> BORN_RULE i c v + BORN_RULE i (~c) v = &1
```

Note that the collapsed state `‘QUANTUM_COLLAPSE i c v‘` is not normalized. It is convenient because, if we have to simulate formally subsequent collapses (i.e. subsequent measurements on different Qbits), we can normalize directly at the end of the process simplifying calculations. In order to do this, we define a service function `‘quantum_normalizer‘` that we use to re-normalize states.

```
let quantum_normalizer = new_definition
 `quantum_normalizer (v:complex^(N)multivector) =
  inv(Cx(norm(v))) % v`;;
```

The conversions `QUANTUM_COLLAPSE_CONV` and `BORN_RULE_CONV` are defined to compute automatically the collapsed states and their relative probabilities. The previous example about a 2-Qbit register can be easily formalized, without re-normalization, in one shot.

```
QUANTUM_COLLAPSE_CONV `QUANTUM_COLLAPSE 0 F (x:complex^(2)multivector)`;;
val it : thm =
  |- QUANTUM_COLLAPSE 0 F x =
     x $$ <|F; F|> % qbasis <|F; F|> +
     x $$ <|F; T|> % qbasis <|F; T|>

BORN_RULE_CONV `BORN_RULE 0 F (x:complex^(2)multivector)`;;
val it : thm =
  |- BORN_RULE 0 F x =
     norm (x $$ <|F; F|>) pow 2 + norm (x $$ <|F; T|>) pow 2
```

Finally, the renormalized collapsed state, in this example, is represented by the following term.

```
`quantum_normalizer (QUANTUM_COLLAPSE 0 F (x:complex^(2)multivector))`
```

Note that we can compute also the collapsed state of a register after a measurement on more then one Qbit by iterating the function `` `QUANTUM_COLLAPSE` ``. For instance, the term

```
`QUANTUM_COLLAPSE 0 F (QUANTUM_COLLAPSE 1 T (x:complex^(N)multivector))`
```

formalizes the collapsed state of a register with n (grater than 2) Qbits after two measurements, the first on the 1-th Qbit and the second on the 0-th Qbit, that yield the results $|1\rangle$ and $|0\rangle$ respectively. Again, such a state must be renormalized by using the function `` `quantum_normalizer` ``. In the previous case of a generic state `` `x:complex^(2)multivector` `` of a 2-Qbit register, the latter can be proved to be equal to the basis state $|01\rangle$, that is, formally, `` `qbasis <|F;T|>` ``. However, if the number of the Qbits subjected to the measurements is large this representation becomes cumbersome, so we have to develop a more general one.

Generalizing equations (10.4.1) and (10.4.2) to the case of a measurement on an arbitrary subset $A \subset \{0, \ldots, n-1\}$ of the n Qbits of the register (i.e. a measurement on all the Qbits indexed by the element of A) it holds that the initial state $|\psi\rangle = \sum_{bs} \alpha_{bs} |bs\rangle$ collapses in the final state

$$|\psi\rangle_{(\bigwedge_{a\in A}\ b_a = m_a)} = \frac{\sum\limits_{bs\in\{b_0\ldots b_i\ldots b_{n-1}\ |\ b_a=m_a\ \forall a\in A\}} \alpha_{bs}|bs\rangle}{\sqrt{\sum\limits_{bs\in\{b_0\ldots b_i\ldots b_{n-1}\ |\ b_a=m_a\ \forall a\in A\}} |\alpha_{bs}|^2}} \tag{10.4.4}$$

with probability

$$P(|\psi\rangle, \bigwedge_{a\in A} b_a = m_a) = \sum_{bs\in\{b_0\ldots b_i\ldots b_{n-1}\ |\ b_a=m_a\ \forall a\in A\}} |\alpha_{bs}|^2 \tag{10.4.5}$$

where $m_a \in \{0,1\}$ is the result of the measurement, on the a-th Qbits of the register, for all $a \in A$. Formally we define

```
let MULTI_QUANTUM_COLLAPSE = define
 `MULTI_QUANTUM_COLLAPSE (n:num^(M)) (m:bool^M) (v:complex^(N)multivector) =
   msum (:bool^N) (\bs. if (P_NUMSEG (dimindex(:M) - 1)
                          (\i. bs $.(n$.i) = m $.i)) then v$$bs % qbasis bs
                      else mat 0)`;;
```

```
let MULTI_BORN_RULE = define
 `MULTI_BORN_RULE (n:num^(M)) (m:bool^M) (v:complex^(N)multivector) =
   sum (:bool^N) (\bs. if (P_NUMSEG (dimindex(:M) - 1)
                           (\i. bs $.(n$.i) = m $.i)) then norm(v$$bs) pow 2
                       else &0)`;;
```

where `v` is the initial state of the register (before the measurement) and the vectors `n:num^M, `m:bool^M` formalize the subset A and the vector of the results $m_a \in \{0,1\}$ respectively. The function P_NUMSEG is a function over predicates on natural numbers. Given a predicate $P\colon \mathbb{N} \to \{0,1\}$ and a natural number $n \in \mathbb{N}$ it returns the conjunction $P(n) \wedge P(n-1) \wedge \cdots \wedge P(0)$. Formally it has the following definition.

```
let P_NUMSEG = define
`(!P:num->bool. P_NUMSEG 0 P = P 0) /\
 (!P n. P_NUMSEG (SUC n) P = (P (SUC n) /\ (P_NUMSEG n P)))`;;
```

Therefore, the selection condition

```
`(P_NUMSEG (dimindex(:M) - 1) (\i. bs $.(n$.i) = m $.i))`
```

in the formal definition of MULTI_QUANTUM_COLLAPSE and MULTI_BORN_RULE, means that every component of `bs`, indexed by the i-th element of `n:num^M`, is equal to the i-th element of the vector of the results of the measurements `m:bool^M`. In fact, such a condition selects the elements of the set $\{b_0 \dots b_i \dots b_{n-1} \mid b_a = m_a \ \forall a \in A\}$ where the set of the indexes A is represented by `n:num^M` and $\{m_a\}_{a\in A}$ is coded by `m:bool^M`. Obviously, it doesn't make sense in case that the size of `:M` is greater then the size of `:N`. Moreover, note again that the collapsed state is not renormalized. It happens, essentially, for the same reasons explained above in case of a measurement on a single Qbit.

As before, two conversions are defined to perform, automatically, computations of term like `MULTI_QUANTUM_COLLAPSE n m v` or `MULTI_BORN_RULE n m v`. For example, in case of a 3-Qibit register in a generic state

$$\begin{aligned}|\psi\rangle =&\alpha_{000}\,|000\rangle + \alpha_{001}\,|001\rangle + \alpha_{010}\,|010\rangle + \alpha_{011}\,|011\rangle + \\ &\alpha_{100}\,|100\rangle + \alpha_{101}\,|101\rangle + \alpha_{110}\,|110\rangle + \alpha_{111}\,|111\rangle\end{aligned}$$

we have that the probability that a measurement on the first two Qbits (0-th and 1-th Qbits) gives the results 0 and 1 respectively is

$$P(|\psi\rangle, b_0 = 0 \wedge b_1 = 1) = |\alpha_{010}|^2 + |\alpha_{011}|^2$$

and the related collapsed state is

$$|\psi\rangle_{(b_0=0\wedge b_1=1)} = \frac{\alpha_{010}\,|010\rangle + \alpha_{011}\,|011\rangle}{\sqrt{|\alpha_{010}|^2 + |\alpha_{011}|^2}}.$$

Using our conversions on a formal state `x:complex^(3)multivector`, we get automatically the HOL theorems that formalize, less then normalization, the latter two equations.

```
 MULTI_BORN_RULE_CONV `MULTI_BORN_RULE <|0;1|> <|F;T|> x`;;
  val it : thm =
  |- MULTI_BORN_RULE <|0; 1|> <|F; T|> x =
     norm (x $$ <|F; T; F|>) pow 2 + norm (x $$ <|F; T; T|>) pow 2

MULTI_QUANTUM_COLLAPSE_CONV `MULTI_QUANTUM_COLLAPSE <|0;1|> <|F;T|> x`;;
  val it : thm =
  |- MULTI_QUANTUM_COLLAPSE <|0; 1|> <|F; T|> x =
     x $$ <|F; T; F|> % qbasis <|F; T; F|> +
     x $$ <|F; T; T|> % qbasis <|F; T; T|>
```

Chapter 11

Formalizing some fundamental quantum protocols and algorithms

As said in the introduction of this part of the thesis, quantum computations are implemented by quantum circuits. Ultimately, a quantum circuit is an operator C that transforms the input state $|\psi_i\rangle \in \mathbb{C}^{2^n}$, of a quantum register, in the output state $C(|\psi_i\rangle) = |\psi_f\rangle \in \mathbb{C}^{2^n}$. If C doesn't involve measurements, it is an Hermitian operator $\mathbb{C}^{2^n} \to \mathbb{C}^{2^n}$ and can be formally represented by a Hermitian matrix. Otherwise, if we want to deal also with measurement, we have to consider also operators that are not Hermitian, since it is the only one operation, among those presented in the previous chapter, that is not unitary.

However, this is not a convenient representation because it hides the compositional nature of the circuit. In fact, in quantum computing, circuits are represented by diagrams where the basic logic gates are assembled until the whole circuit is created as shown in the example of teleportation (8.3.1) presented previously.

In our formal setting, a circuit C is represented by a HOL term of type

```
‘:complex^(N)multivector->complex^(N)multivector‘
```

and, following the informal style it is constructed, compositionally, by subsequent applications of the formal logic gates (including the measurement operator, that is, our formal collapse operator) defined in the previous chapter.

In the next section, we explain the symbolism used drawing informal quantum circuit diagrams and, after, we focus on three fundamental quantum circuits and the related formalizations.

11.1 Circuit diagrams

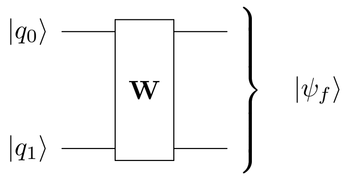
It is the practice in quantum computer science to represent the action of a sequence of gates on n Qbits by a circuit diagram. The initial state of the register appears on the left, the final state on the right, and the gates themselves in the central part of the figure. For example, a circuit diagram representing the action, on a single Qbit, of the 1-Qbit gate $\mathbf{u}$ is drawn as

$$|\psi\rangle \;\text{——}\boxed{\mathbf{u}}\text{——}\; \mathbf{u}\,|\psi\rangle \tag{11.1.1}$$

where $|\psi\rangle$ is the initial state of the Qbit, $\mathbf{u}$ is the gate and $\mathbf{u}\,|\psi\rangle$ is the resulting final state. If the register has more than one Qbit, we can apply different gates to different Qbits. For example, in the simple case of a register with 2 Qbits we can apply the $\mathbf{u}$ gate to the first and the $\mathbf{v}$ gate to the second. The corresponding diagram is the following.

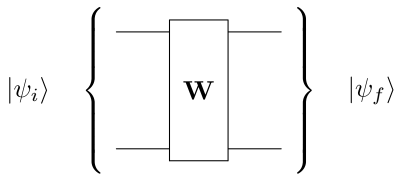
$$\begin{array}{c} |q_0\rangle \;\text{——}\boxed{\mathbf{u}}\text{——}\; \mathbf{u}\,|q_0\rangle \\ \\ |q_1\rangle \;\text{——}\boxed{\mathbf{v}}\text{——}\; \mathbf{v}\,|q_1\rangle \end{array} \tag{11.1.2}$$

Moreover, we represent the action of a 2-Qbit gate $\mathbf{W}$, applied to the previous register, by the diagram



(11.1.3)

where $|\psi_f\rangle = \mathbf{W}(|q_0\rangle\,|q_1\rangle)$ is the final state of the 2-Qbit register after the application of the gate $\mathbf{W}$. However, it is well known that, in the case of an input with two Qbits in an *entangled* state, we can neither operate on the states of the individual Qbits as in (11.1.2), nor represent the action of a 2 Qbits port as in (11.1.3). This happens because such states must be considered as one, in fact, they cannot be written as the tensor product of the states of the single Qbits. In these cases, we fix representation (11.1.3) in this way

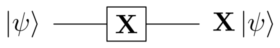


(11.1.4)

where $|\psi_i\rangle$ is the initial entangled state in input and, as before, $|\psi_f\rangle = \mathbf{W}\,|\psi_i\rangle$ is the output state. The generalization to registers with n Qbits is analogous to diagram (11.1.4).

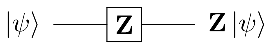
The gates formalized in the previous chapter are all 1-Qbit gate except the Quantum controlled NOT gate. In the standard quantum computing theory, they are represented as follows.

- **Quantum NOT** is represented by the symbol $\mathbf{X}$



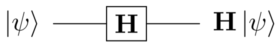
(11.1.5)

- **Quantum SHIFT** is represented by the symbol $\mathbf{Z}$



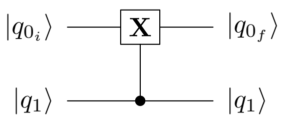
(11.1.6)

- **Quantum HADAMARD** is represented by the symbol $\mathbf{H}$



(11.1.7)

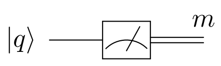
- **Quantum controlled NOT** is represented as **cX** by the diagram



(11.1.8)

where $|q_{0_i}\rangle$ and $|q_{0_f}\rangle$ are the initial and final state of the target Qbit (q_0). The final states $|q_{0_f}\rangle$ depends on the state $|q_1\rangle$ of the control Qbit (q_1).

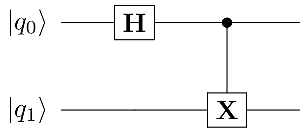
- **Measurement** is represented by the symbol in the following diagram



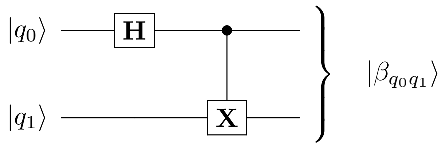
(11.1.9)

where $m \in \{0, 1\}$ is the result of the measurement and the double line means that, after the measurement, the Qbits is collapsed in the classical state $|m\rangle$.

In this settings, for example, the circuit diagram



represents the subsequent action of the quantum *HADAMARD* gate (on the first Qbit q_0) and the quantum controlled *NOT* gate (with target q_1 and control q_0) on a register with 2 Qbits. Such a circuit is very used in quantum computer science because it generates the four maximally entangled states of two Qbits, called *Bell's states*, depending on the value of $q_0, q_1 \in \{0, 1\}$. In fact, given two Qbits $|q_o\rangle$,$|q_1\rangle$ in one of the basis states (i.e. $q_0, q_1 \in \{0, 1\}$), then we have that



(11.1.10)

where $\beta_{q_0q_1}$ is one of the four Bell's states.

$$\beta_{00} = \frac{|00\rangle + |11\rangle}{\sqrt{2}} \qquad \beta_{01} = \frac{|01\rangle + |10\rangle}{\sqrt{2}}$$
$$\beta_{10} = \frac{|00\rangle - |11\rangle}{\sqrt{2}} \qquad \beta_{11} = \frac{|01\rangle - |10\rangle}{\sqrt{2}} \tag{11.1.11}$$

The function `‘bell_state:bool->bool->complex^(N)multivector‘` formalizes the previous states.

```
let bell_state = define
 ‘bell_state F F = Cx(inv(sqrt(&2))) % (qbasis <|F;F|> + qbasis <|T;T|>) /\
  bell_state F T = Cx(inv(sqrt(&2))) % (qbasis <|T;F|> + qbasis <|F;T|>) /\
  bell_state T F = Cx(inv(sqrt(&2))) % (qbasis <|F;F|> - qbasis <|T;T|>) /\
  bell_state T T = Cx(inv(sqrt(&2))) % (qbasis <|F;T|> - qbasis <|T;F|>)‘;;
```

Such a computational states doesn't have any classical counterpart, so they are used to give rise to phenomena which are paradoxical from a classical point of view, but which express the full potential of quantum computing.

Now, we have the right background to present our formalization of some fundamental quantum algorithms and protocols. More precisely, the *Quantum teleportation* and *Superdense coding* protocols and the *Deutsch's algorithm.*

11.2 Quantum teleportation protocol

The *quantum teleportation protocol* is a protocol that permits one to transmit quantum information (i.e. the exact state of a Qbit) from one location to another, with the help of classical communication. More precisely, it is not a form of transportation, as one can image by the name *teleportation*, but of communication: it provides a way of transporting a Qbit from one location to another without having to move a physical particle along with it.

To understand the kind of problems that teleportation can solve, we can imagine the following situation. A person, that we will call Alice, has to let know the state of a Qbit

$$|\phi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle\,, \qquad |\alpha|^2 + |\beta|^2 = 1 \tag{11.2.1}$$

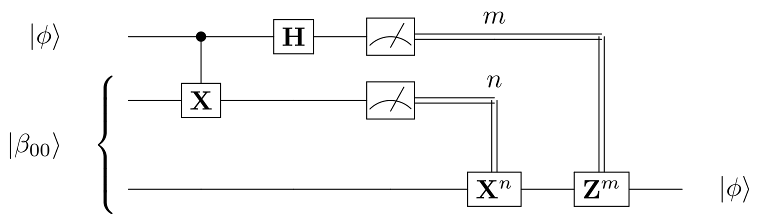
to another person that we will call Bob. Alice does not know the amplitudes α and β and for the no-cloning theorem[1] she can't make a copy of the state $|\phi\rangle$. In addition, Alice can

[1]The no-cloning theorem states that it is impossible to create an identical copy of an arbitrary unknown quantum state

send to Bob only classical information, that is, the values 0 and 1 of a classic bit. In this situation, from a classical point of view, it would seem impossible to transmit the Qbit to Bob. However, it is possible using the properties of the entangled states, in particular the Bell's state presented before. The fundamental hypothesis are:

- Alice and Bob share an entangled couple of Qbits, in state $|\beta_{00}\rangle$, generated previously,
- Alice and Bob can operate on their respective Qbits of such a couple.

The following circuit, where the first two lines represent the Qbits used by Alice (the first is the Qbit that has to be transmitted, the second is the Alice's Qbit entangled with the Bob's one) and the third represent the Qbit of Bob, describes the protocol.



(11.2.2)

Given the state (11.2.1) that Alice wants to transmit, the input of the circuit is the 3-Qbit register in state $|\psi_0\rangle = |\phi\rangle\,|\beta_{00}\rangle$ that is, in computational basis,

$$|\psi_0\rangle = \frac{1}{\sqrt{2}}(\alpha\,|000\rangle + \alpha\,|011\rangle + \beta\,|100\rangle + \beta\,|111\rangle).$$

Alice combines $|\phi\rangle$ with her half of the entangled pair and then she measures her two Qbits after applying the quantum *CNOT* and *HADAMARD* gates. She sends to Bob, via a classical communication channel, the two classical values ($m, n \in \{0,1\}$) obtained after the measurements. At this point, Bob will be able to rebuilt the original state $|\phi\rangle$, using the properties of entangled states.

More in details, after that Alice has applied the *CNOT* gate to her Qbits the whole register is in state

$$|\psi_1\rangle = \mathbf{cX}_{01}\,|\psi_0\rangle = \frac{1}{\sqrt{2}}(\alpha\,|000\rangle + \alpha\,|011\rangle + \beta\,|110\rangle + \beta\,|101\rangle)$$

and then, the application of the *HADAMARD* gate on the first Qbit leads to the state

$$\begin{aligned}|\psi_2\rangle = \mathbf{H}_0\,|\psi_1\rangle = \frac{1}{2}(&\alpha\,|000\rangle + \alpha\,|100\rangle + \alpha\,|011\rangle + \alpha\,|111\rangle + \\ &\beta\,|010\rangle - \beta\,|110\rangle + \beta\,|001\rangle - \beta\,|101\rangle).\end{aligned} \tag{11.2.3}$$

At this point, after the Alice's measurements on her Qbits, the state of the whole register could collapse in one of the four following states:

- $\alpha\,|000\rangle + \beta\,|001\rangle$ if the results are $n = m = 0$,
- $\alpha\,|100\rangle - \beta\,|101\rangle$ if the results are $m = 1$ and $n = 0$,
- $\alpha\,|011\rangle - \beta\,|010\rangle$ if the results are $m = 0$ and $n = 1$,
- $\alpha\,|110\rangle - \beta\,|110\rangle$ if the results are $m = 1$ and $n = 1$.

Now, it easy to check that if Bob applies to his Qbit the appropriate transformation $\mathbf{X}^n\mathbf{Z}^m$, depending on the classical bits $n, m \in \{0,1\}$ received from Alice, then he gets the final state

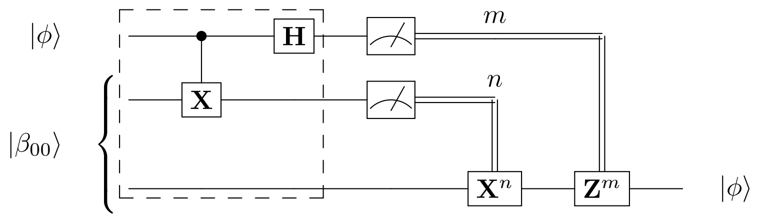
$$|\psi\rangle = \alpha\,|nm0\rangle + \beta\,|nm1\rangle \tag{11.2.4}$$

that can be easily rewritten as $|nm\rangle\,(\alpha\,|0\rangle + \beta\,|1\rangle)$. It proves the theoretical correctness of the teleportation protocol because the state of the third Qbit (Bob's Qbit) is, in any case, equal to that of the original Alice's Qbit, that is, $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$.

In our context, let be `psi0:complex^(3)multivector` the formal counterpart of the input state $|\psi_0\rangle$

```
let psi0 = inv (Cx (sqrt (&2))) %
          (a % qbasis <|F; F; F|> + a % qbasis <|F; T; T|> +
           b % qbasis <|T; F; F|> + b % qbasis <|T; T; T|>)
```

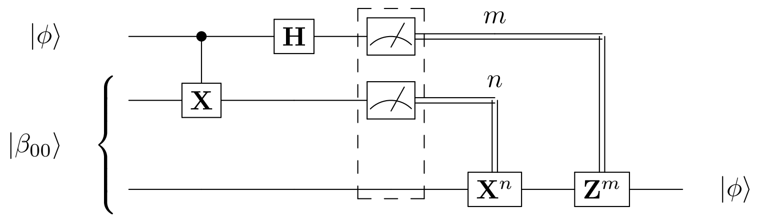
then the first part of the circuit



can be formalized as

```
let psi1 = QHADAMARD 0 (QCNOT 0 1 psi0)
```

and can be easily computed by QCNOT_CONV and QHADAMARD_CONV. We recall that we use a 0-based indexing so, in our formal framework, the first Qbit is the 0-th and so on.
The central part of the protocol, that is the measurements on the Alice's Qbits,



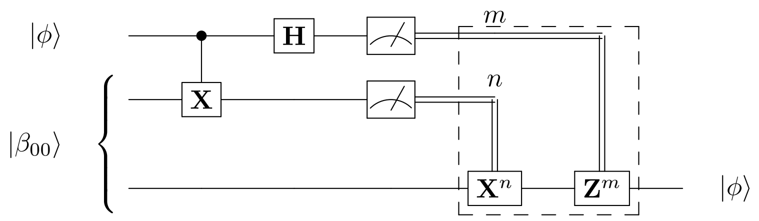
is formalized by the function `QUANTUM_COLLAPSE` that leads to the state

`quantum_normalizer (QUANTUM_COLLAPSE 1 n (QUANTUM_COLLAPSE 0 m psi1))`

where `quantum_normalizer` is needed to re-normalize the collapsed state. Since in such a state the Alice's Qbits (0-th and 1-th) are surely in state $|m\rangle$, $|n\rangle$ ($m, n \in \{0, 1\}$) respectively, we define the linear function

```
QNOT_OR_SHIFT
  |- (!i. clinear (QNOT_OR_SHIFT i)) /\
     (!i bs. QNOT_OR_SHIFT i (qbasis bs) =
            (if bs $. (i - 2) then QSHIFT i (if bs $. (i - 1)
                                             then QNOT i (qbasis bs)
                                             else qbasis bs)
             else (if bs $. (i - 1) then QNOT i (qbasis bs)
                   else qbasis bs)))
```

that, given a basis vector `qbasis bs` and an index `i`, applies on the i-th Qbit the transformation $\mathbf{Z}^m\mathbf{X}^n$ depending on the values $m, n \in \{0, 1\}$ of the $(i-2)$-th and $(i-1)$-th Qbits respectively. Thus, the function `QNOT_OR_SHIFT 2` formalizes the last part of the circuit



applying the appropriate transformation on the Bob's Qbit. Therefore, the final state $|\psi\rangle$ is formally represented by the following HOL term.

```
let psi = QNOT_OR_SHIFT 2
           (quantum_normalizer
            (QUANTUM_COLLAPSE 1 n
             (QUANTUM_COLLAPSE 0 m psi1)))
```

Finally, we certify the theoretical correctness of the teleportation protocol proving that, in any case, the initial state of the first Qbit has been teleported on the third Qbit. More precisely, we prove formally that, for every possible collapse (i.e. for all possible results of the measurements $m, n \in \{0, 1\}$), the final state $|\psi\rangle$ is always of the form (11.2.4), that is, it is provable the following HOL term.

```
‘psi = a % qbasis <|m; n; F|> + b % qbasis <|m; n; T|>
```

Therefore, having in mind the previous formalizations of the intermediate states of the computation, the resulting formal statement that we proved in HOL Light is the following.

```
QUANTUM_TELEPORTATION
  |- !a b m n.
        norm a pow 2 + norm b pow 2 = &1
        ==> (let psi0 = inv (Cx (sqrt (&2))) %
                        (a % qbasis <|F; F; F|> +
                         a % qbasis <|F; T; T|> +
                         b % qbasis <|T; F; F|> +
                         b % qbasis <|T; T; T|>) in
             let psi1 = QHADAMARD 0 (QCNOT 0 1 psi0) in
             let psi = QNOT_OR_SHIFT 2
                        (quantum_normalizer
                         (QUANTUM_COLLAPSE 1 n
                          (QUANTUM_COLLAPSE 0 m psi1))) in
             psi = a % qbasis <|m; n; F|> + b % qbasis <|m; n; T|>)
```

11.3 Deutsch’s problem

In 1992, David Deutsch discovered a problem, solved by the related Deutsch’s algorithm, that is one of the first examples of a problem that can be solved, by a quantum algorithm, faster then by any possible deterministic classical algorithm. Deutsch’s algorithm is also a deterministic algorithm, that is, it always produces an answer, and that answer is always correct.

The Problem. Suppose to have a function $f\colon \{0, 1\} \to \{0, 1\}$, we wish to know if the function is *balanced* ($f(0) \neq f(1)$) or *constant* ($f(0) = f(1)$).

Classically, the problem could be solved by evaluating $f(0) \oplus f(1)$ where $\oplus$ is the logical sum *XOR*. The latter will be equal to 1 if the function is balanced and equal to 0 otherwise. Using this strategy, we need to evaluate the function twice in order to determine whether it is balanced or not and, depending on the cost of the function f, it can be very expensive from a computational point of view.

However, it is very easy to check that there exist only four functions that maps $\{0, 1\}$ into $\{0, 1\}$. Using the lambda calculus notation they are:

- the constant functions $\lambda x.0$ and $\lambda x.1$,
- the balanced functions $\lambda x.x$ and $\lambda x.\neg x$.

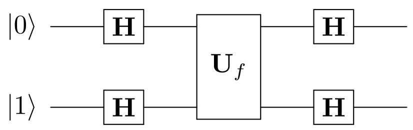
In HOL Light, these facts are summarized in the following theorems.

```
BOOL_FUN_CASES
  |- !f. f = (\x. x) \/ f = (\x. ~x) \/
         f = (\x. T) \/ f = (\x. F)

BOOL_CONSTANT_EXPLICIT
  |- !f. bool_constant f <=> f = (\x. T) \/ f = (\x. F)

BOOL_BALANCED_EXPLICIT
  |- !f. bool_balanced f <=> f = (\x. x) \/ f = (\x. ~x)
```

where `bool_balanced` and `bool_constant` are defined in the standard way.

```
let bool_constant = new_definition
 `bool_constant (f:bool->bool) <=> f T = f F`;;

let bool_balanced = new_definition
 `bool_balanced (f:bool->bool) <=> ~(f T = f F)`;;
```

The Deutsch's algorithm. Let be f a function that maps $\{0,1\}$ into $\{0,1\}$. Deutsch's algorithm is described by the following circuit diagram



(11.3.1)

where $\mathbf{U}_f$ is the 2-Qbit gate defined on the computational basis by

$$\mathbf{U}_f\,|i\rangle\,|j\rangle = |i\rangle\,|j \oplus f(i)\rangle \tag{11.3.2}$$

with $i, j \in \{0,1\}$. Note that, if $j = 0$ the output state of equation (11.3.2) is $|i\rangle\,|f(i)\rangle$. As we see from the diagram, Deutsch's algorithm consists essentially in four steps.

1. Preparing two Qbits , one in state $|0\rangle$ and the other in state $|1\rangle$. Therefore, the initial state of the circuit is $|\psi_0\rangle = |01\rangle$.
2. Apply the *HADAMARD* gate to both Qbits.
3. Apply the $\mathbf{U}_f$ gate to the whole register.
4. Apply, again, the *HADAMARD* gate to both Qbits.

We show the case of the identity function $f = \lambda x.\ x$ more in details. Applying the first *HADAMARD* gates to the initial state we obtain the state

$$|\psi_1\rangle = \mathbf{H}_1\mathbf{H}_0\,|\psi_0\rangle = \frac{1}{2}(|00\rangle - |01\rangle + |10\rangle - |11\rangle)$$

and the consequent application of the $\mathbf{U}_f$ gate produces the state

$$|\psi_2\rangle = \mathbf{U}_f\,|\psi_1\rangle = \frac{1}{2}(|00\rangle - |01\rangle + |11\rangle - |10\rangle).$$

Finally, the last *HADAMARD* gates lead to the final state

$$\begin{aligned}|\psi_f\rangle = \mathbf{H}_1\mathbf{H}_0\,|\psi_2\rangle = \frac{1}{4}(\,&|00\rangle + |01\rangle + |10\rangle + |11\rangle - |00\rangle + |01\rangle - |10\rangle + |11\rangle + \\ &|00\rangle - |01\rangle - |10\rangle + |11\rangle - |00\rangle - |01\rangle + |10\rangle + |11\rangle)\end{aligned}$$

that, by direct computation, becomes $|\psi_f\rangle = |11\rangle$.

This implies that, making a measurement on the first Qbit (for the second is the same), we obtain the result $|1\rangle$ with probability 1 and the related collapsed state is again $|\psi_f\rangle$, that is

$$\begin{array}{ll} P(|\psi_f\rangle, b_0 = 1) = 1 & |\psi_f\rangle_{b_0=1} = |\psi_f\rangle \\ P(|\psi_f\rangle, b_1 = 1) = 1 & |\psi_f\rangle_{b_1=1} = |\psi_f\rangle \end{array}$$

It easy to check the other three cases by direct computation. It turns out that, in the case of a balanced function ($\lambda x.x$ or $\lambda x.\neg x$) the circuit (11.3.1) produces the output states $|11\rangle$ or $-|11\rangle$ whereas, with a constant function ($\lambda x.1$ or $\lambda x.\neg 0$), the possible output state are $|01\rangle$ and $-|01\rangle$. Therefore, we have that the first Qbit is in state $|0\rangle$ with probability 1 in the case of a constant function and is in state $|1\rangle$, again with probability 1, in the case of a balanced function. Thus, making a measurement on the first Qbit, after the computation, we can determine whether the function is balanced or not. Moreover, a very relevant fact is that the function f was only evaluated once during the algorithm, while the classical algorithm evaluates it twice. This could make significant difference if such a function is very complicated and takes a great deal of time to compute.

In our formal setting, the gate $\mathbf{U}_f$ is formalized proving the following theorem.

```
QUANTUM_UGATE
  |- (!f. clinear (QUANTUM_UGATE f)) /\
     (!f bs. QUANTUM_UGATE f (qbasis bs) =
             qbasis <|bs $. 0; bs $. 1 + f (bs $. 0)|>)
```

Here, the symbol `+` is overloaded to represent also the logical sum. As for the logic gates of the previous chapter, we define the conversion `QUANTUM_UGATE_CONV` that computes automatically terms of the form `QUANTUM_UGATE f x`.

The circuit (11.3.1) is formally represented, depending on the function `f:bool->bool`, by the constant `deutsch_algorithm` defined as follows.

```
let deutsch_algorithm = new_definition
 `deutsch_algorithm (f:bool->bool) =
  QHADAMARD 1
   (QHADAMARD 0
    (QUANTUM_UGATE f
     (QHADAMARD 1
      (QHADAMARD 0 (qbasis <|F;T|>)))))`;;
```

Joining togheter `QHADAMARD_CONV`, `QUANTUM_UGATE_CONV` and some appropriate arithmetic simplifications, we can define a further conversion `DEUTSCH_CONV` that is able to compute automatically terms as `deutsch_algorith f` and `BORN_RULE i b (deutsch_algorithm f)`. Essentially, it analyses the form of the input term and, after rewriting the `deutsch_algorithm` definition and computing circuit (11.3.1) with `QHADAMARD_CONV` and `QUANTUM_UGATE_CONV`, it applies the conversion `BORN_RULE_CONV` if it is necessary.

For all the four possible functions of type `:bool->bool`, we can produce the related theorems in one shot.

- **Balanced functions.**

```
DEUTSCH_CONV `deutsch_algorithm (\x. x)`;;
val it : thm =
  |- deutsch_algorithm (\x. x) = qbasis <|T; T|>

DEUTSCH_CONV `BORN_RULE 0 T (deutsch_algorithm (\x. x))`;;
val it : thm =
  |- BORN_RULE 0 T (deutsch_algorithm (\x. x)) = &1
```

```
DEUTSCH_CONV ‘deutsch_algorithm (\x. ~x)‘;;
val it : thm =
  |- deutsch_algorithm (\x. ~x) = Cx (-- &1) % qbasis <|T; T|>

DEUTSCH_CONV ‘BORN_RULE 0 T (deutsch_algorithm (\x. ~x))‘;;
val it : thm =
  |- BORN_RULE 0 T (deutsch_algorithm (\x. ~x)) = &1
```

- **Constant functions.**

```
DEUTSCH_CONV ‘deutsch_algorithm (\x. T)‘;;
val it : thm =
  |- deutsch_algorithm (\x. T) = Cx (-- &1) % qbasis <|F; T|>

DEUTSCH_CONV ‘BORN_RULE 0 F (deutsch_algorithm (\x. T))‘;;
val it : thm =
  |- BORN_RULE 0 F (deutsch_algorithm (\x. T)) = &1

DEUTSCH_CONV ‘deutsch_algorithm (\x. F)‘;;
val it : thm =
  |- deutsch_algorithm (\x. F) = qbasis <|F; T|>

DEUTSCH_CONV ‘BORN_RULE 0 F (deutsch_algorithm (\x. F))‘;;
val it : thm =
  |- BORN_RULE 0 F (deutsch_algorithm (\x. F)) = &1
```

At this point, we can easily prove the theorem that formalizes the Deutsch's problem. It consists of a conjunction of three statements.

```
DEUTSCH_PROBLEM
  |- (!f b. BORN_RULE 0 b (deutsch_algorithm f) = &1 \/
            BORN_RULE 0 b (deutsch_algorithm f) = &0) /\
     (!f. bool_constant f <=>
          BORN_RULE 0 F (deutsch_algorithm f) = &1) /\
     (!f. bool_balanced f <=>
          BORN_RULE 0 T (deutsch_algorithm f) = &1)
```

The first statement means that the first Qbit of the output state of the Deutsch's algorithm is in state $|0\rangle$ or $|1\rangle$ (depending on the form of the function f) with probability 1. The second and the third state that such a Qbit is in state $|0\rangle$ if and only if the function f is constant and, it is in state $|1\rangle$ if and only if f is balanced. This proves definitely, in our formal context, that the circuit represented by the function ‘`deutsch_algorithm`‘ solves the *Deutsch's problem.*

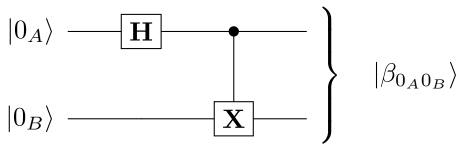
11.4 Superdense coding protocol

Another significant example, in which quantum computing is able to solve a problem that doesn't have a classical solution, is the *Superdense coding protocol.*

Imagine a situation where two people (named Alice and Bob) are in different parts of the world. Alice has two bits $b_0, b_1 \in \{0,1\}$ and she would like to communicate them to Bob by sending him just a single Qbit. It turns out that there is no way they can accomplish this task without additional resources (see [Nielsen and Chuang, 2000] for more details). This is not obvious, but it is true, Alice cannot encode two classical bits into a single Qbit in any way that would give Bob more than just one bit of information about the pair b_0, b_1. However, let us imaging that Alice and Bob share an entangled pair of Qbits generated previously in the Bell's state

$$|\beta_{0_A 0_B}\rangle = \frac{1}{\sqrt{2}}(|0_A 0_B\rangle + |1_A 1_B\rangle) \tag{11.4.1}$$

where the subscripts A and B indicate the Qbit of Alice and Bob respectively. So, after generating the input state $|\beta_{0_A0_B}\rangle$ with the circuit

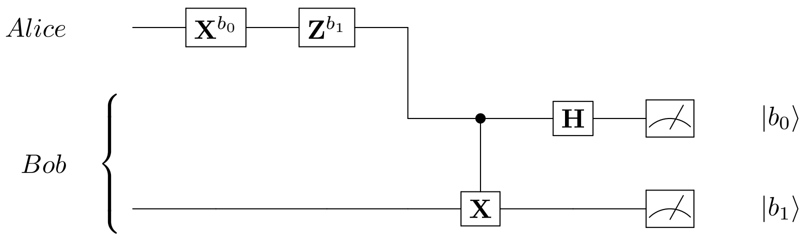


(11.4.2)

the protocol, called *Superdense coding*, can start. It consists in four steps.

1. Alice applies to her Qbit (of the entangled pair) a transformation depending on the value of the bits that she want to send to Bob:
   - if $b_0 = 0$ and $b_1 = 0$ then she does nothing,
   - if $b_0 = 0$ and $b_1 = 1$ then she applies the quantum *NOT* gate $\mathbf{X}$,
   - if $b_0 = 1$ and $b_1 = 0$ then she applies the quantum *SHIFT* gate $\mathbf{Z}$,
   - if $b_0 = 1$ and $b_1 = 1$ then she applies first the $\mathbf{X}$ gate and then the $\mathbf{Z}$ gate.
2. Alice sends her Qbit to Bob (this is the only Qbit that is sent during the protocol).
3. Bob applies a quantum *controlled NOT* operation to the pair of Qbits (A, B) where A (the Qbits received by Alice) is the control and B (the Bob's Qbit) is the target. Consecutively, Bob applies the *HADAMARD* gate $\mathbf{H}$ to the Qbit A.
4. Bob measures both Qbits A and B. The result will be (b_0, b_1) with certainty because the output state of the register at the end of the protocol is surely $|b_0b_1\rangle$.

All these steps are summarized in the following diagram



(11.4.3)

and we recall that the input state is $|\beta_{0_A0_B}\rangle$ generated, for example, with the circuit (11.4.2).

As example, we show more in details the case of $b_0 = 0$ and $b_1 = 1$, the others (three) cases are very similar, so we avoid to report them explicitly.

If $b_0 = 0$ and $b_1 = 1$, then the first step of the protocol produces the state

$$|\psi_1\rangle = \mathbf{X}_A\,|\beta_{0_A0_B}\rangle = \frac{1}{\sqrt{2}}(|1_A0_B\rangle + |0_A1_B\rangle).$$

At this point, Bob receives the Qbit from Alice and applies the *controlled-NOT* gate obtaining the intermediate state

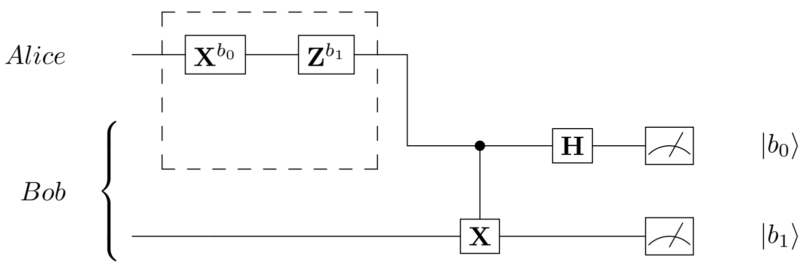
$$|\psi_2\rangle = \mathbf{cX}_{AB}\,|\psi_1\rangle = \frac{1}{\sqrt{2}}(|1_A1_B\rangle + |0_A1_B\rangle).$$

Finally, with the application of the last *HADAMARD* gate, he gets the final state

$$|\psi_f\rangle = \mathbf{H}_A\,|\psi_2\rangle = \frac{1}{2}(|0_A1_B\rangle - |1_A1_B\rangle + |0_A1_B\rangle + |1_A1_B\rangle) = |0_A1_B\rangle\,.$$

Note that, for every gate $\mathbf{V}$, the notation $\mathbf{V}_A$ or $\mathbf{V}_B$ means that the transformation is applied on the Alice's or Bob's Qbit respectively. Thus, at the end of the protocol, if Bob measures both the Qbits he gets the results $b_0 = 1$ for the first and $b_1 = 0$ for the second with certainty (i.e. with probability 1). In this way, he can know the values of the two classic bits that Alice wanted to send him.

In our formalization the first part of the protocol (i.e. Alice's action)



(11.4.4)

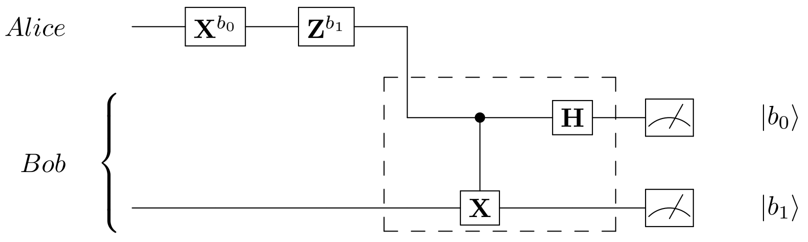
is encoded by the constant

```
‘ALICE_SUPERDENSE_CONDING_PROTOCOL:bool->bool->complex^(2)multivector‘
```

defined case by case by the following HOL terms.

```
let ALICE_SUPERDENSE_CONDING_PROTOCOL = define
 ‘ALICE_SUPERDENSE_CODING_PROTOCOL F F = bell_state F F /\
  ALICE_SUPERDENSE_CODING_PROTOCOL F T = QNOT 0 (bell_state F F) /\
  ALICE_SUPERDENSE_CODING_PROTOCOL T F  = QSHIFT 0 (bell_state F F) /\
  ALICE_SUPERDENSE_CODING_PROTOCOL T T  = QSHIFT 0
                                        (QNOT 0 (bell_state F F))‘;;
```

In the latter definition, the booleans arguments represent the values of the two classical bits b_0 and b_1 that Alice wants to send. Furthermore, the second part of the protocol (Bob's action)



(11.4.5)

doesn't depend on the value of b_0 and b_1 and it is formalized by

```
let BOB_SUPERDENSE_CODING_PROTOCOL = new_definition
 ‘BOB_SUPERDENSE_CODING_PROTOCOL (v:complex^(2)multivector) =
  QHADAMARD 0 (QCNOT 0 1 v)‘;;
```

where `‘v:complex^(2)multivector‘` is a generic state of a 2-Qbit register.

Finally, the whole protocol (11.4.3) is represented by the term

```
‘BOB_SUPERDENSE_CODING_PROTOCOL (ALICE_SUPERDENSE_CODING_PROTOCOL b0 b1)‘
```

with `‘b0:bool‘` and `‘b1:bool‘`. We can certify, in our formal framework, that the output of such a protocol is always $|b_0 b_1\rangle$ that is, we can prove formally that the previous HOL term is equal to the term `‘qbasis <|b0;b1|>‘`. The resulting theorem, that we proved in HOL Light, is the following.

```
SUPERDENSE_CODING_OUTPUT
  |- !b0 b1. BOB_SUPERDENSE_CODING_PROTOCOL
              (ALICE_SUPERDENSE_CODING_PROTOCOL b0 b1) =
             qbasis <|b0; b1|>
```

This implies that, after a measurement on the first (second) Qbit, it will be in state $|b_0\rangle$ ($|b_1\rangle$) with probability equal to one. The following theorem formalizes this result.

```
SUPERDENSE_CODING
  |- !b0 b1. BORN_RULE 0 b0
              (BOB_SUPERDENSE_CODING_PROTOCOL
               (ALICE_SUPERDENSE_CODING_PROTOCOL b0 b1)) = &1 /\
             BORN_RULE 1 b1
              (BOB_SUPERDENSE_CODING_PROTOCOL
               (ALICE_SUPERDENSE_CODING_PROTOCOL b0 b1)) = &1
```

The latter two formal theorems definitively certify the validity and the theoretical correctness of the *Superdense coding* protocol.

Conclusions

We developed a formal automatic and certified calculation system about quantum circuits. More precisely, we can specify, in our language, any quantum circuits (written using the set of universal quantum gates $\mathbf{X}$, $\mathbf{cX}$, $\mathbf{Z}$, $\mathbf{H}$, $\mathbf{R}_\phi$) certifying, with formal proofs, their functioning and their properties. As examples, or better as tests for our framework, we formalized some of the basics quantum algorithms and protocols, more precisely, the *Teleportation protocol*, the *Superdense coding protocol* and the *Deutsch's algporithm.*

Moreover, since the theory relies in a fundamental way on the algebra of complex vectors, we worked to improve the current implementation of vectors in HOL Light in two ways. On one hand, we extended the standard library with new basic results about complex vectors, including the case of complex *multivectors.* On the other hand, we provided a new way to represent types with finite cardinality (*Fintypes*) and to encode concrete vectors indexed by such types (*Finvec*), which accommodate for working with arbitrary fixed finite dimension and for writing procedures for automatic calculations. In fact, we also defined a conversion that calculates automatically components of a concrete vector.

Afterall, this part of the work takes about 4,000 lines of code and consists in about 350 theorems without considering the background material about complex vectors and complex linear algebra.

A possible line of improvement for this work is to formalize further mathematical results in complex linear algebra, in particular, properties of Hermitian operators. Definitively, dealing with quantum computing is easier if a considerable part of complex linear algebra is available in HOL Light. For example, we could prove formally the theoretical result that the gates that we have formalized form a set of universal quantum gates, that is, any other unitary operation can be expressed as a sequence of gates from this set.

Moreover, it could be convenient to find a method to define explicitly the tensor product in HOL Light. Simulating the action of a circuit, it is not strictly necessary because every state, during a computation, can be written as a linear combination of the standard computational basis. However, it would allow us to reason on the metatheory, for example, we could define formally the *entanglement* and make reasoning about it.

Last but not least, our code can be used to certify new algorithms or circuits when it is tedious to do by hand. An interesting problem could be the following. Let C be a circuit that, with input I, produces an output $O = C(I)$. How does O vary if I varies slightly?

In principle, formalizing C in our settings we can perform calculations automatically producing certified tests for this problem. The latter is a practical interesting problem because often, even if a protocol or an algorithm is well designed and produces a deterministic result, it is physically very difficult to prepare exactly the needed input state. Therefore, a formal method that investigates and checks the effects on the result of an algorithm (or protocol) of an error, even if small, in such preparation could be very useful.

Bibliography

[Afshar et al., 2014] Afshar, S., Aravantinos, V., Hasan, O., and Tahar, S. (2014). Formalization of complex vectors in higher-order logic. In S.M., W., J.H., D., A.P., S., P., S., and J., U., editors, *Intelligent Computer Mathematics*, volume 8543 of *Lecture Notes in Computer Science*, pages 123–137. Springer, Cham.

[Bennett et al., 1993] Bennett, C. H., Brassard, G., Crépeau, C., Jozsa, R., Peres, A., and Wootters, W. K. (1993). Teleporting an unknown quantum state via dual classical and einstein-podolsky-rosen channels. *Phys. Rev. Lett.*, 70:1895–1899.

[Deutsch and Jozsa, 1992] Deutsch, D. and Jozsa, R. (1992). Rapid solution of problems by quantum computation. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 439:553 – 558.

[Deutsch, 1985] Deutsch, D. (1985). Quantum theory, the Church-Turing Principle and the Universal Quantum computer. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 400.

[Mermin, 2007] Mermin, N. D. (2007). *Quantum Computer Science: An Introduction.* Cambridge University Press New York, NY USA.

[Nielsen and Chuang, 2000] Nielsen, M. and Chuang, I. (2000). *Quantum Computation and Quantum Information.* Cambridge University Press.

[Wilde, 2017] Wilde, M. M. (2017). *Quantum Information Theory.* Cambridge University Press, 2 edition.