Mechanizing Hyperdual Numbers in Isabelle/HOL

Filip Smola

MInf Project (Part 1) Report
Master of Informatics
School of Informatics
University of Edinburgh

2020

Abstract

Second-order hyperdual numbers, a number system based on the dual numbers [8], have been proposed as method of computing accurate second derivatives [10]. However, the properties of hyperdual numbers on which this is based have not been formally proven.

In this project we investigate the properties of hyperdual numbers in detail, formalize them and mechanize the definitions and proofs in the interactive theorem prover Isabelle. We do this to provide theoretical support for investigation of their uses, including among others differentiation.

Acknowledgements

I would like to thank Dr. Jacques Fleuriot who is supervising this project. His support throughout all parts of this project was invaluable. The recommendations and feedback he provided kept me on course and are much appreciated.

I would also like to thank all members of the AI Modelling Lab[1] for their support through the challenges of this project and all their feedback.

[1] https://aiml.inf.ed.ac.uk/

Table of Contents

Chapter 1

Introduction

Hyperdual numbers consist of a real part and a number of infinitesimal parts. These infinitesimal components have the useful property of having zero squares while being non-zero themselves. This leads to an interesting behaviour with uses such as automatic differentiation, representation of rigid body dynamics or skinning of models.

In this project, we investigate properties of hyperdual numbers, formalize them and mechanize the definitions and proofs in the interactive theorem prover Isabelle. With this we build formal theory allowing investigation of the uses of hyperdual numbers. We only look at their mathematical properties and ignore details of representation and implementation. The main application we focus on in this work is the use of hyperdual numbers for differentiation.

1.1 Motivation and Goals

The original inspiration for this project is the work of Fike and Alonso [10], where the authors describe basic properties of hyperdual numbers and their use for second derivative computation. This work only briefly establishes the mathematical properties of hyperdual numbers and quickly moves on to the application and benchmarks. To provide a more solid formal basis for the properties and their use, we decided to mechanize the properties of hyperdual numbers used in that work as well as any further properties that we deem useful for other applications.

Our aim is to mechanize a well-rounded theory of hyperdual numbers in which their practical applications can eventually be verified. This includes proving hyperdual numbers to be an instance of multiple algebraic structures, which allows us to inherit all the already proven properties of those structures. When the behaviour of hyperdual numbers does not fall into some known algebraic structure (e.g. their division not forming a division algebra – see Section 3.5), we prove as many properties as possible of the closest known structure and find counter-examples to the properties they violate. In this way we hope to clearly delineate how hyperdual numbers “lack” the structure in question.

In our mechanization we also hope to uncover any hidden assumptions in the work of Fike and Alonso [10]. This is especially true with the hyperdual extension (see Chapter 4), which forms the basis of the authors' approach to automatic differentiation.

The mechanization concerns only second-order hyperdual numbers, that is ones with two distinct infinitesimals (ε_1, ε_2) and their product ($\varepsilon_1\varepsilon_2$). Second-order hyperduals were selected because the expressions involved are simple enough to be mechanized within the scope of this project, while still yielding the properties we are interested in (see for example the hyperdual extension in Chapter 4). Despite this limitation, most of the mechanized definitions and properties should be generalizable to higher-order hyperdual numbers.

1.2 Report Organisation

In Chapter 2 we provide context on Isabelle, the interactive theorem prover used for this project, and also mention the mechanisation of other number systems that bear some relation to the hyperduals. In Chapter 3, we define second-order hyperdual numbers and a number of operations on them. We also show them to be an instance of multiple algebraic structures. In Chapter 4, we describe the hyperdual extension of real functions, motivations for its definition and show its properties. In Chapter 5, we summarize conclusions about hyperdual numbers and outline future work. In the Appendix (Chapter 6), we provide the full definitions and proofs mentioned in the document.

Chapter 2

Context

2.1 Mechanization in Isabelle/HOL

To ensure correctness of our reasoning, we decided to mechanize our definitions and claims in a theorem prover. This validates all our proofs and makes sure there are no hidden assumptions, lending further credence to our conclusions.

There are two main types of theorem provers, *automatic* and *interactive*. Automatic theorem provers usually take a first-order logic statement and attempt to prove it fully automatically, without any input from the user [31] [32]. Interactive theorem provers function more as a dialogue between the user and the theorem prover [2]. The user instructs the theorem prover which rules to apply and the theorem prover computes the result and verifies the correctness of that step. Despite the recent rapid improvement of automated theorem provers, they are not suitable to projects such as this one. The reasoning required for some of the proofs is not necessarily straightforward for a computer and the rule search space is vast, which means that automatic methods without human guidance often struggle to find a proof. For example, using induction is still intractable for fully automated theorem provers [7]. Furthermore, most automatic theorem provers are limited to first-order logic, while we require a more expressive formal system.

Another reason why interactive theorem provers are well suited to this project is the exploration aspect. We are not mechanizing already developed proofs of known properties, but instead exploring what properties hyperdual numbers have and then proving those. This is where the dialogue aspect of an interactive theorem prover shines. When attempting to prove a claim, we can use trial and error to explore different approaches to the proof with the theorem prover making sure we never make an incorrect step, never miss a case. If the claim is false, this process can help highlight which of its aspects is the problem. This for example led us to realizing that hyperduals have non-trivial zero divisors (see Section 3.3.1).

We chose to use the Isabelle interactive theorem prover for this project. Isabelle is a generic interactive theorem prover supporting many object logics: various first-order logics, ZF set theory and higher-order logic to name a few [27]. We use the higher-order logic (HOL) and base our work on the Analysis formal theory (HOL-Analysis). Thanks to years of work by Isabelle's community, there is a sizeable library of already proven theorems to work from. On top of that, Isabelle also offers a more readable proof language caller *Isar* [40]. This language is close to the style of mathematical pen and paper proofs and makes proof development much more convenient and intuitive.

Another invaluable feature of Isabelle is the proof search tool *sledgehammer* [6]. This tool takes the current goal and some selected theorems, found via machine learning, and invokes a number of first-order automatic provers on those. If the automated provers are able to prove the goal, Isabelle uses the information they produce to reconstruct its proof. Even when such reconstruction is not fully successful, the result can sometimes be enough to show a possible path forward to the user. This saves a lot of time that would otherwise be spent meticulously proving "trivial" things.

In Isabelle, we begin claims to be proven with the keyword **lemma**, often followed with a name for the claim. This may be followed by a number of variables that the proof **fixes** (along with their type), and a number of statements that it **assumes** to be true. Then follows the keyword **shows** and the conclusion being proven. If no variables are fixed and no statements assumed, the conclusion can directly follow the lemma name. For example:

lemma *divisors-re-zero*:
  fixes $a\ b :: hyperdual$
  assumes $Re\ (a * b) = 0$
  shows $Re\ a = 0 \lor Re\ b = 0$

is equivalent to the higher-order logic statement:

$\bigwedge a\ b :: hyperdual\ .\ Re\ (a * b) = 0 \Longrightarrow Re\ a = 0 \lor Re\ b = 0$

There are two kinds of proofs in Isar, application of automatic methods and proof blocks. An automatic method can be applied to a statement by following it with:

using *[facts/rules]* **by** *[method]*

This takes all the facts stated after **using** and invokes the stated method with those facts on the statement. Note that the method can also use other facts that are not stated at the same point, if they are globally added to its consideration.

Proof blocks are a way for the user to split the proof of the statement into smaller steps. They start with **proof** and conclude with **qed**. Within this block are usually multiple statements, preceded with **have**, with their own proof (automatic or block). They can be chained with **then**, which adds the prior statement to be considered by the automatic method applied to the following statement. We also often use **moreover** and **ultimately**, where **moreover** keywords collect multiple statements and then the **ultimately** keyword adds them all to the be considered by the automatic method applied to the following statement. This represents the confluence of multiple lines of reasoning into one step. Facts containing meta-variables can be instantiated with specific terms using *rule[of x y z]*. This is often helpful in constraining the search space for the method applying the rules.

Sometimes we have a number of lemmas that share some fixed values and assumptions. We can collect these in what is called a **locale**. In simple terms, if an object satisfies the assumptions of a locale (i.e. it is an *instance* of the locale) then all of the lemmas within that locale apply to it. This is very useful for keeping statements concise and organising lemmas. For example, we do this when describing division of hyperdual numbers to collect the assumptions of our division (see Section 3.5 for details):

```
locale division-hyperdual =
  assumes left-inverse:  Re a ≠ 0 ⟹ inverse a ∗ a = 1
  assumes right-inverse: Re a ≠ 0 ⟹ a ∗ inverse a = 1
  assumes divide-inverse: a / b = a ∗ inverse b
  assumes inverse-zero: Re a = 0 ⟹ Re (inverse a) = 0
```

Isabelle also supports Haskell-style type classes [12] to achieve a form of overloading. These type classes are essentially locales with a single type variable. They are used to establish that a type together with some set of parameters (usually functions) satisfy some specification. However, the type class's parameters are publicly available constants, which means that they can then be used polymorphically with any instance of the respective type class. In this project we do not introduce any new type classes, but we make extensive use of existing ones when proving hyperdual numbers to be an instance of certain algebraic structures. For example, consider the following definition of the type class for a commutative (abelian) group under addition:

```
class ab-group-add = minus + uminus + comm-monoid-add +
  assumes ab-left-minus: − a + a = 0
  assumes ab-diff-conv-add-uminus: a − b = a + (− b)
begin

subclass group-add
by standard (simp-all add: ab-left-minus ab-diff-conv-add-uminus)

subclass cancel-comm-monoid-add
proof...

lemma uminus-add-conv-diff [simp]: − a + b = b − a
by (simp add: add.commute)

lemma minus-add-distrib [simp]: − (a + b) = − a + − b
by (simp add: algebra-simps)

lemma diff-add-eq [algebra-simps, field-simps]: (a − b) + c = (a + c) − b
by (simp add: algebra-simps)
end
```

This class assumes that the type in question is an instance of all *minus*, *uminus* and *comm-monoid-add* type classes, establishing it as a commutative monoid under addition with a notion of (unary) minus. It then extends that with the assumption that minus is an additive inverse and that subtracting is adding the minus. Within its body, it is proven to be a subclass of two further type classes and a couple new theorems are introduced and proven.

To instantiate a type class with a type, we need to supply definitions of the parameters and prove any assumptions. Consider for example the instantiation of hyperdual numbers as the above class (see Appendix 6.1 for details):

instantiation *hyperdual* :: *ab-group-add*
begin
primcorec *zero-hyperdual*
  where ...
primcorec *plus-hyperdual*
  where ...
primcorec *uminus-hyperdual*
  where ...
primcorec *minus-hyperdual*
  where ...
instance
by *standard simp-all*
end

Isabelle also allows us to use corecursion [5]. We define most of the functions discussed as primitive corecursive. These definitions start with the keyword **primcorec**, followed by the name of the function, type and defining expressions. We mostly use destructor view notation, that is expressing the function result in terms of the output of destructors applied to it. In our case the destructors are the component projections, leading to essentially a componentwise definition. This suits our definitions well and keeps them readable. For this reason, we define hyperdual numbers as a coinductive datatype (see Section 3.1).

2.2 Similar Number Systems

As the name suggests, hyperdual numbers came about from dual numbers, which (in contrast) only have one infinitesimal part. Along with complex numbers [37] and higher-order versions of those, such as quarternions [38], all these number systems are tightly intertwined. These are all instances of hypercomplex number systems as described by Kantor and Solodovnikov [19] (see Section 3.7 for the hyperdual case). Here we give a quick summary of the most important of these and their uses.

Perhaps most familiar of these are complex numbers. These can be expressed as $a+bi$ where a,b are real coefficients and i is the solution to $x^2 = -1$ – the imaginary unit. This yields the following multiplication behaviour:

$$(a+bi)\cdot(c+di) = ac+adi+bci+bdi^2 = (ac-bd)+(ad+bc)i$$

Complex number have been heavily studied and thus have a great number of known uses. Highlights include Fourier analysis [29] and complex-step derivative approximation [24]. They also play a fundamental role in quantum mechanics [39].

Complex numbers were mechanized in Isabelle in early 2000s by Fleuriot, with some additions by Paulson[1]. This mechanization is now included as part of standard Isabelle

[1]https://isabelle.in.tum.de/dist/library/HOL/HOL/Complex.html

distributions. Due to the similarity of the number systems, we use this mechanization to inspire our approach to hyperdual numbers (see Chapter 3). We note in passing that there are also mechanizations of complex numbers in the HOL Light [15] theorem prover and in Coq[2].

By taking pairs of complex numbers and applying a similar multiplication to them one can construct quarternions (this is known as the Cayley-Dickson construction [1]). Quaternions can be expressed as $a+bi+cj+dk$ where a,b,c,d are real coefficients and i,j,k are the non-real units. In this case, the multiplication is given by the distributive law and the following basis multiplication table:

×	1	i	j	k
1	1	i	j	k
i	i	-1	k	-j
j	j	-k	-1	i
k	k	j	-i	-1

The most notable use of quaternions is representing rotations in 3D space [36] [22]. They are more compact compared to a matrix representation, but most importantly they avoid gimbal lock, which can occur in some other rotation systems and leads to loss of one degree of freedom.

Quaternions were mechanized in Isabelle in 2018 by Paulson [28], in a similar way to how complex numbers are mechanized there. This mechanization is not included as part of standard Isabelle distributions, but is available on the Archive of Formal Proofs. There is also a mechanization of quaternions in the HOL Light theorem prover [11].

Dual numbers [8] are defined similarly to complex numbers. They can be expressed as $a+b\varepsilon$ where a,b are real coefficients but the non-real unit ε is in this case nilpotent (i.e. $\varepsilon^2=0$). This yields the following multiplication behaviour:

$$(a+b\varepsilon)\cdot(c+d\varepsilon)=ac+ad\varepsilon+bc\varepsilon+bd\varepsilon^2=ac+(ad+bc)\varepsilon$$

Dual numbers themselves can be used to perform first-order differentiation [30]. Dual-complex numbers (either dual numbers with complex parts or complex numbers with dual parts, both are equivalent) are used to represent combinations of rotation and displacement in 2D space [25]. Dual quaternions (dual number whose parts are quaternions) are used to represent rigid transformations in 3D space, for example in computer graphics [20] and robotics [26].

Due to dual numbers' use in automatic differentiation, there exists a number of implementations in various languages. For example, there are C++ and Matlab implementations by Fike and Alonso, and a Fortran implementation by Edwin van der Weide[3]. There is also a Julia implementation as part of the Julia ForwardDiff package [30].

[2]https://www.cs.umd.edu/~rrand/vqc/Complex.html
[3]http://adl.stanford.edu/hyperdual/

2.3 Summary

We described our motivations for mechanizing this theory and the choice of the interactive theorem prover Isabelle. We summarized the key features of Isabelle and described the relevant syntax that we use. We also summarized three other number systems with similar features: complex numbers, quaternions and dual numbers.

In the next chapter, we will describe in detail the properties of hyperdual numbers. We will provide both a mathematical characterization and a description of the mechanization in Isabelle.

Chapter 3

Properties of Hyperdual Numbers

We now proceed to define hyperdual numbers and their properties. When indicated, the described definition follows Fike and Alonso [10], but is usually further formalized. Otherwise it is an original definition.

3.1 Hyperdual Number

We define a hyperdual number as:

$$a = a_1 + a_2 \cdot \varepsilon_1 + a_3 \cdot \varepsilon_2 + a_4 \cdot \varepsilon_1\varepsilon_2$$

where all $a_i \in \mathbb{R}$, and ε_1 and ε_2 are non-zero but nilpotent infinitesimals (i.e. $\varepsilon_1 \neq 0$, $\varepsilon_2 \neq 0$, $\varepsilon_1\varepsilon_2 \neq 0$ but $\varepsilon_1^2 = \varepsilon_2^2 = (\varepsilon_1\varepsilon_2)^2 = 0$) [10]. We denote the set of all hyperdual numbers with H.

We call the first term a_1 the real component, the second a_2 and third terms a_3 the first-order hyperdual components, and the fourth term a_4 the second-order hyperdual component. We also define the projections $Re(a) = a_1$, $Eps_1(a) = a_2$, $Eps_2(a) = a_3$ and $Eps_{12}(a) = a_4$.

We define two hyperdual numbers to be equal when all their components are equal. That is for any $a, b \in H$:

$$a = b \leftrightarrow a_1 = b_1 \wedge a_2 = b_2 \wedge a_3 = b_3 \wedge a_4 = b_4 \tag{3.1}$$

Any real number can be expressed as a hyperdual number with zero hyperdual components. That is, for any $x \in \mathbb{R}$, we have:

$$x = x + 0 \cdot \varepsilon_1 + 0 \cdot \varepsilon_2 + 0 \cdot \varepsilon_1\varepsilon_2 \tag{3.2}$$

3.1.1 Mechanization

In Isabelle, we define the hyperdual numbers as a coinductive data type with four real-valued components:

codatatype *hyperdual* = *Hyperdual* (*Re*: *real*) (*Eps1*: *real*) (*Eps2*: *real*) (*Eps12*: *real*)

This definition gives us among other properties the four projections. Note that we could have defined hyperdual numbers as an inductive datatype, but using a coinductive one allows us to define functions corecursively, which greatly improves the readability of the definitions (this approach is also used for defining the complex numbers in Isabelle).

We then prove statement (3.1) i.e. that two hyperdual numbers are equal if and only if their respective components are equal as follows:

lemma *hyperdual-eq-iff*:
 $x = y \longleftrightarrow ((Re\ x = Re\ y) \wedge (Eps1\ x = Eps1\ y) \wedge (Eps2\ x = Eps2\ y) \wedge (Eps12\ x = Eps12\ y))$
using *hyperdual.expand* **by** *auto*

3.2 Addition

As is done by Fike and Alonso [10], we define the addition of two hyperdual numbers a and b as the sum of the expanded expressions, resulting in a componentwise sum. That is for any $a, b \in H$:

$$a + b = (a_1 + b_1) + (a_2 + b_2)\varepsilon_1 + (a_3 + b_3)\varepsilon_2 + (a_4 + b_4)\varepsilon_1\varepsilon_2$$

We then define the additive inverse of a as:

$$-a = -a_1 - a_2\varepsilon_1 - a_3\varepsilon_2 - a_4\varepsilon_1\varepsilon_2$$

and the hyperdual zero $0 \in H$ such that:

$$0 = 0 + 0 \cdot \varepsilon_1 + 0 \cdot \varepsilon_2 + 0 \cdot \varepsilon_1\varepsilon_2$$

Then hyperdual numbers form a commutative group under addition with identity 0. In particular, this means that:

- Hyperduals are closed under addition
- Addition is associative and commutative
- Zero is identity for addition
- Each hyperdual has an inverse with which it adds to zero

3.2.1 Mechanization

We proved that hyperdual numbers are an instance of Isabelle's commutative (or abelian) group under addition type class *ab-group-add*. For this, we defined addition corecursively as follows:

```
primcorec plus-hyperdual
  where
   Re (x + y) = Re x + Re y
  | Eps1 (x + y) = Eps1 x + Eps1 y
  | Eps2 (x + y) = Eps2 x + Eps2 y
  | Eps12 (x + y) = Eps12 x + Eps12 y
```

with the remaining operations (inverse and subtraction) and unit constant (zero) defined similarly (see Appendix 6.1). The proof of the instantiation is quickly performed by automatic methods, because the operation behaves as real addition on each component.

3.3 Multiplication

We define the product of two hyperdual numbers a and b as the product of the expanded expressions [10], keeping in mind the properties of the infinitesimals ε_1 and ε_2. That is for any $a, b \in H$:

$$a \cdot b = (a_1 b_1) + (a_1 b_2 + a_2 b_1)\varepsilon_1 + (a_1 b_3 + a_3 b_1)\varepsilon_2 + (a_1 b_4 + a_2 b_3 + a_3 b_2 + a_4 b_1)\varepsilon_1\varepsilon_2 \tag{3.3}$$

We then define the hyperdual one $1 \in H$ such that:

$$1 = 1 + 0 \cdot \varepsilon_1 + 0 \cdot \varepsilon_2 + 0 \cdot \varepsilon_1\varepsilon_2$$

Then hyperdual numbers form a commutative ring with identity 1. This means that:

- Hyperduals are a commutative group under addition
- Multiplication is associative and commutative
- One is identity for multiplication
- Multiplication distributes over addition

After formalizing multiplication, we investigated its inverse and the related division operation. During that investigation, we identified the several notable properties for which multiplication over the hyperduals differs from that over the reals. These differences are discussed next.

3.3.1 Zero Divisors

Unlike with real numbers, there exist non-zero hyperdual numbers that multiply to zero (i.e. $a, b \in H$ such that $a \cdot b = 0$ while $a \neq 0$ and $b \neq 0$). Intuitively, this is because the non-real parts of a hyperdual number express infinitesimal components which have a tendency to vanish under multiplication.

As we will see in Section 3.4, hyperduals form an associative algebra. Because associative division algebras have no non-trivial zero divisors, we can deduce that hyperduals are not a division algebra [19]. This agrees with our conclusions in Section 3.5.

The precise conditions for a pair of hyperdual numbers to multiply to zero are as follows:

$$a \cdot b = 0 \leftrightarrow a = 0 \vee b = 0 \vee (a_1 = 0 \wedge b_1 = 0 \wedge a_2 b_3 = -a_3 b_2) \tag{3.4}$$

It is trivial to see that when either of the factors is zero then their product is zero, because all terms of the multiplication involve zero and are therefore themselves zero. What is more interesting is the third disjunct. By the definition of multiplication (3.3), if either of the factors' real components is zero, the real component of the product is zero. If both are zero, the first-order hyperdual components of the product are zero. Moreover, then the second-order hyperdual component of the product is $a_2 b_3 + a_3 b_2$ which leads to the last part of the condition for it to be zero.

Thus, if the real components of *both* factors are zero, the conditions on the rest of their components are somewhat relaxed.

We can derive these conditions by solving $a \cdot b = 0$ for $a, b \in H$. We split that into equations for components $a_i, b_i \in \mathbb{R}$ using the hyperdual equality (3.1) and multiplication (3.3) definitions:

$$\begin{aligned}
0 &= a_1 b_1 \\
0 &= a_1 b_2 + a_2 b_1 \\
0 &= a_1 b_3 + a_3 b_1 \\
0 &= a_1 b_4 + a_2 b_3 + a_3 b_2 + a_4 b_1
\end{aligned}$$

The first equation means that $a_1 = 0$ or $b_1 = 0$, because there are no non-trivial real zero divisors.

We use this to split the second equation into three cases. When both a_1, b_1 are zero, then that equation is satisfied without any further requirements. When only one of a_1, b_1 is zero, then that number's second component also has to be zero for the equation to be satisfied. Same reasoning applies to the third equation.

The same case split is used for the last equation. When both a_1, b_1 are zero, we know nothing further about a_2, a_3, b_2, b_3 and thus we can only simplify the equation by removing the first and last terms. When only one of a_1, b_1 is zero, then that number's second and third component also have to be zero and after simplification this means that number's last component is also zero or the other number's first component is also zero. By the case assumption we know that the other number's first component cannot also be zero, therefore the number with zero first component must be the hyperdual zero (all components zero).

All of this yields the following equivalence:

$$\begin{aligned} a\cdot b=0 \leftrightarrow &(a_1=0\vee b_1=0)\wedge(a_1=0\wedge b_1=0\rightarrow a_2b_3=-a_3b_2)\wedge\\ &(a_1\neq 0\wedge b_1=0\rightarrow b_2=0\wedge b_3=0\wedge b_4=0)\wedge\\ &(a_1=0\wedge b_1\neq 0\rightarrow a_2=0\wedge a_3=0\wedge a_4=0)\end{aligned}$$

As a result of how the derivation proceeds, this equivalence has some redundancy. By the first conjunct, the precondition of one of the remaining conjuncts has to be true. Therefore:

$$\begin{aligned} a\cdot b=0 \leftrightarrow &(a_1=0\wedge b_1=0\wedge a_2b_3=-a_3b_2)\vee\\ &(a_1\neq 0\wedge b_1=0\wedge b_2=0\wedge b_3=0\wedge b_4=0)\vee\\ &(a_1=0\wedge b_1\neq 0\wedge a_2=0\wedge a_3=0\wedge a_4=0)\end{aligned}$$

Simplifying with the definition of hyperdual zero we get the following:

$$\begin{aligned} a\cdot b=0 \leftrightarrow &(a_1=0\wedge b_1=0\wedge a_2b_3=-a_3b_2)\vee\\ &(a_1\neq 0\wedge b=0)\vee(b_1\neq 0\wedge a=0)\end{aligned}$$

Next, note that the first disjunct contains the cases $a_1=0\wedge b=0$ and $b_1=0\wedge a=0$. We can use this to "fill out" the second and third disjunct, removing their first conjuncts:

$$a\cdot b=0 \leftrightarrow (a_1=0\wedge b_1=0\wedge a_2b_3=-a_3b_2)\vee b=0\vee a=0$$

Finally, this equivalence can be reordered as (3.4).

3.3.2 Multiplication Cancellation

There are rules for multiplication cancellation in real numbers, taking the general form $a=b\cdot a\leftrightarrow(a=0\vee b=1)$. Similar rules can be derived for hyperduals. These turn out to have an added option compared to the real version in a similar vein to the zero divisors described above. The derivation of these rules mirrors that of the zero divisor condition.

The precise conditions for this cancellation are:

$$a=b\cdot a\leftrightarrow a=0\vee b=1\vee(a_1=0\wedge b_1=1\wedge a_2b_3=-a_3b_2) \tag{3.5}$$

It is again trivial to see that when $a=0$ or $b=1$ then their product is a, by substitution into the definition of hyperdual multiplication (3.3). What is again more interesting is the third disjunct. If either $a_1=0$ or $b_1=1$, the real component of the product is equal to that of a. If both $a_1=0$ and $b_1=1$, the first-order hyperdual components of the product are equal to those of a. Moreover, then the second-order hyperdual component of the equation is $a_4=a_4+a_2b_3+a_3b_2$, or after simplifying $0=a_2b_3+a_3b_2$, which again leads to the third part of that disjunct.

Thus, if the real components of a and b are 0 and 1 respectively, the conditions on the rest of their components are again somewhat relaxed.

3.3.3 Mechanization

We proved that hyperdual numbers are an instance of the commutative ring under multiplication type class *comm-ring-1*. For this, we defined multiplication corecursively as follows:

primcorec *times-hyperdual*
  where
   $Re\ (x * y) = Re\ x * Re\ y$
  $|\ Eps1\ (x * y) = (Re\ x * Eps1\ y) + (Eps1\ x * Re\ y)$
  $|\ Eps2\ (x * y) = (Re\ x * Eps2\ y) + (Eps2\ x * Re\ y)$
  $|\ Eps12\ (x * y) = (Re\ x * Eps12\ y) + (Eps1\ x * Eps2\ y) + (Eps2\ x * Eps1\ y) + (Eps12\ x * Re\ y)$

with the unit constant (one) defined similarly to the addition unit (see Appendix 6.2). The proof of the instantiation is quickly performed by automatic methods with addition of the set of algebraic simplifications that is available in Isabelle.

We mechanized both the zero divisor (3.4) equivalence:

lemma *divisiors-hyperdual-zero*:
  fixes $a\ b$ :: *hyperdual*
  shows $a * b = 0 \longleftrightarrow (a = 0 \vee b = 0 \vee (Re\ a = 0 \wedge Re\ b = 0 \wedge Eps1\ a * Eps2\ b = -\ Eps2\ a * Eps1\ b))$

and the multiplication cancellation (3.5) equivalence:

lemma *hyperdual-mult-cancel-right1*:
  fixes $a\ b$ :: *hyperdual*
  shows $a = b * a \longleftrightarrow a = 0 \vee b = 1 \vee (Re\ a = 0 \wedge Re\ b = 1 \wedge Eps1\ b * Eps2\ a = -\ Eps2\ b * Eps1\ a)$

For the zero divisors, the proof that hyperdual numbers satisfying the conditions do multiply to zero (right-to-left implication) is easily dispatched by automatic methods. To prove the converse we consider each of the four combinations of truth values for statements $Re(a) = 0$ and $Re(b) = 0$. We then verify in each case, that either one of the numbers is zero, both their real components are zero with the correct relation of the first-order hyperdual components, or the case leads to a contradiction. We then prove one of the multiplication cancellation rules in the same way and use symmetry of equality and commutativity of multiplication to extend that to the three other possible reorderings of the statement. See Appendix 6.3 and 6.4 for full proofs.

3.4 Scalar Multiplication and Real Algebra

We define the scalar product of a hyperdual number with a real factor as the componentwise scalar product. That is for any $h \in H$ and $f \in \mathbb{R}$:

$$f \cdot a = fa_1 + fa_2\varepsilon_1 + fa_3\varepsilon_2 + fa_4\varepsilon_1\varepsilon_2$$

With hyperdual addition and multiplication, hyperdual numbers form an algebra over the real numbers with unit 1. This means that:

- Hyperduals are a commutative ring under multiplication and addition
- Scalar multiplication by two real numbers is scalar multiplication by their product (for $a, b \in \mathbb{R}$ and $h \in H$ have $a(bh) = (ab)h$)
- Real one is the identity for scalar multiplication
- Scalar multiplication distributes over hyperdual and real addition (for $a, b \in \mathbb{R}$ and $h, g \in H$ have $a(h+g) = ah + ag$ and $(a+b)h = ah + bh$)
- Scalar multiplication is compatible with hyperdual multiplication (for $a, b \in \mathbb{R}$ and $h, g \in H$ have $(ah) \cdot (bg) = (ab)(h \cdot g)$)

3.4.1 Mechanization

We proved that hyperdual numbers are an instance of Isabelle's type class of algebras with unit 1 over real numbers – *real-algebra-1*. Given the previous two instantiations, this only requires the definition of scalar multiplication:

primcorec *scaleR-hyperdual*
 where
  Re (*scaleR f x*) = *f* ∗ *Re x*
 | *Eps1* (*scaleR f x*) = *f* ∗ *Eps1 x*
 | *Eps2* (*scaleR f x*) = *f* ∗ *Eps2 x*
 | *Eps12* (*scaleR f x*) = *f* ∗ *Eps12 x*

The proof of the instantiation is again performed by automatic methods with the help of Isabelle's set of algebraic simplification rules (see Appendix 6.5). Note that in Isabelle this operation is usually denoted by the infix operator $*_R$.

With this instantiation we also get the *of-real* function, defined in Isabelle as follows:

definition *of-real* :: *real* ⇒ *'a*::*real-algebra-1*
 where *of-real r* = *scaleR r 1*

This function defines an embedding from of any real number r into the instantiated type, which corresponds to the conversion (3.2) we defined when introducing the type. Now we can prove this as a theorem, which is then added to the simplifier to allow it to be automatically used from this point on:

lemma [*simp*]: *of-real a* = *Hyperdual a 0 0 0*
by (*simp add*: *of-real-def one-hyperdual.code*)

3.5 Multiplicative Inverse and Division

First we define the inverse of a hyperdual number. We want the inverse of any $a \in H$ to be some $\frac{1}{a} \in H$ such that $a \cdot \frac{1}{a} = 1$. Solving this equation for $\frac{1}{a}$ we get:

$$\frac{1}{a} = \frac{1}{a_1} - \frac{a_2}{a_1^2}\varepsilon_1 - \frac{a_3}{a_1^2}\varepsilon_2 + \left(2\frac{a_2 a_3}{a_1^3} - \frac{a_4}{a_1^2}\right)\varepsilon_1\varepsilon_2$$

From this, we can see that the inverse of a hyperdual number is only defined when its real component is non-zero. All other components can have any value, the only constraint is on the real part. This means that while hyperdual numbers have a notion of division, they do not form a division algebra, because their division is not defined for all $a \neq 0$. This agrees with the Frobenius theorem [19] for real division algebras, which states that any finite-dimensional associative division algebra over the real numbers is isomorphic to the real numbers, complex numbers or quaternions.

Then we can define division of two hyperdual numbers $\frac{a}{b}$ when $b_1 \neq 0$ as follows:

$$\begin{aligned}\frac{a}{b} = \frac{a_1}{b_1} + \frac{a_2 b_1 - a_1 b_2}{b_1^2}\varepsilon_1 + \frac{a_3 b_1 - a_1 b_3}{b_1^2}\varepsilon_2 + \\ \frac{2a_1 b_2 b_3 - a_1 b_1 b_4 - a_2 b_1 b_3 - a_3 b_1 b_2 + a_4 b_1^2}{b_1^3}\varepsilon_1\varepsilon_2\end{aligned}$$

3.5.1 Mechanization

First, we proved hyperdual numbers are an instance of the type class *inverse* by supplying the definition of inverse (see Appending 6.6 for the full instantiation):

```
primcorec inverse-hyperdual
  where
   Re (inverse a) = 1 / Re a
  | Eps1 (inverse a) = − Eps1 a / (Re a)^2
  | Eps2 (inverse a) = − Eps2 a / (Re a)^2
  | Eps12 (inverse a) = 2 ∗ (Eps1 a ∗ Eps2 a / (Re a)^3) − Eps12 a / (Re a)^2
```

Note that in Isabelle/HOL all functions have to be total. In our case the inverse becomes zero when not well defined, which is inherited from Isabelle's real inverse being defined to be zero for zero arguments.

Most of the division properties in Isabelle are proven as part of the *division-ring* type class. Because hyperdual numbers do not satisfy its assumptions, we cannot take advantage of this type class to inherit these properties. We instead chose to build a new locale specialised to hyperduals which mirrors *division-ring*. In this locale we attempt the same lemmas, proving those that are true and providing counter-examples to those that are false. We use a locale instead of a type class because we intend to use it with only one specific type, hyperduals. Thus this is more of a collection of theorems than polymorphic specification, and falls more withing the purview of a locale.

The locale definition is as follows:

```
locale division-hyperdual =
  assumes left-inverse:  Re a ≠ 0 ⟹ inverse a ∗ a = 1
  assumes right-inverse: Re a ≠ 0 ⟹ a ∗ inverse a = 1
  assumes divide-inverse: a / b = a ∗ inverse b
  assumes inverse-zero: Re a = 0 ⟹ Re (inverse a) = 0
```

The assumptions are the same as for *division-ring*, but the $a \neq 0$ conditions are replaced with $Re(a) \neq 0$ instead. Similar adjustments are made to the lemmas considered, otherwise their definitions are kept as close to original as possible to make the reuse of proofs easier.

While some lemmas are easily proven to still hold, there are a few properties of the original class that do not translate to this locale. These differences mainly show up when we talk about intervals where the inverse is not well-defined. Due to the fact that multiple hyperdual numbers have inverse equal to 0 (that is any $h \in H$ such that $Re(h) = 0$), inverse is not an involution nor a bijection in the general case. However, it is both an involution and a bijection when it is well-defined.

Proving hyperduals to be an instance of this locale was mostly done by automatic methods when supplied with some algebraic simplification rules. The one difficult part was proving that division is exactly multiplication by inverse. This required a series of handwritten expression transformations to break down the proof into parts manageable by the automatic methods (see Appendix 6.7).

3.6 Real Normed Vector Space

With the above properties of addition and scalar multiplication, the hyperduals form a vector space. A basis of this vector space is formed by $\{1, \varepsilon_1, \varepsilon_2, \varepsilon_1\varepsilon_2\}$ and each hyperdual number can be expressed as a four-dimensional real vector:

$$a_1 + a_2\varepsilon_1 + a_3\varepsilon_2 + a_4\varepsilon_1\varepsilon_2 = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix}$$

We extend this into a real normed vector space by defining a norm on hyperduals as the Euclidian distance:

$$||a|| = \sqrt{a_1^2 + a_2^2 + a_3^2 + a_4^2}$$

This being a norm [35] means that:

- Only hyperdual zero has zero norm, all other hyperduals have positive norm
- Scalar multiplication scales the norm by the absolute value of the real factor
- Triangle inequality holds (the norm of a sum is at most the sum of norms of the summands)

3.6.1 Alternative Considered Norm

We also considered the alternative norm defined only on the real component, that is:

$$||a|| = ||a_1||$$

The motivation was that this would be closer to the interpretation that the infinitesimal components should not have as much impact as the real component. It is also the absolute value function used by Fike and Alonso [10]. However, this definition does not satisfy one of the assumptions of a norm – that only the zero element should have zero norm. It is trivial to see that any hyperdual number with a zero real component would have zero norm, not just the hyperdual zero. Thus this is not a valid norm, although it is a seminorm [35].

3.6.2 Real Normed Algebra

As part of our investigation, we also examined whether the hyperduals form a real normed algebra. This structure adds an additional assumption over the real normed vector space which describes a triangle inequality for a product, that is for any $x, y \in H$:

$$\|x \cdot y\| \leq \|x\| \cdot \|y\|$$

Hyperduals do not have this property though. For example, $x = y = 1 + 1\varepsilon_1 + 1\varepsilon_2 + 1\varepsilon_1\varepsilon_2$ gives $\|x \cdot y\| = \|1 + 2\varepsilon_1 + 2\varepsilon_2 + 4\varepsilon_1\varepsilon_2\| = 5$ whereas $\|x\| \cdot \|y\| = 2 \cdot 2 = 4$.

3.6.3 Mechanization

As part of the mechanization to show that the hyperduals form a real normed vector space, we first defined four constants corresponding to the four basis vectors:

definition *re* **where** *re* = *Hyperdual 1 0 0 0*
definition *e1* **where** *e1* = *Hyperdual 0 1 0 0*
definition *e2* **where** *e2* = *Hyperdual 0 0 1 0*
definition *e12* **where** *e12* = *Hyperdual 0 0 0 1*

We then proved the following theorems:

- **fixes** *x* :: *hyperdual*
  shows $\exists$*a-1 a-2 a-3 a-4* :: *real* . *x* = *a-1* $*_R$ *re* + *a-2* $*_R$ *e1* + *a-3* $*_R$ *e2* + *a-4* $*_R$ *e12*
- **shows** *Hyperdual a-1 a-2 a-3 a-4* = *a-1* $*_R$ *re* + *a-2* $*_R$ *e1* + *a-3* $*_R$ *e2* + *a-4* $*_R$ *e12*
- **fixes** *x* :: *hyperdual*
  shows *x* = *Re x* $*_R$ *re* + *Eps1 x* $*_R$ *e1* + *Eps2 x* $*_R$ *e2* + *Eps12 x* $*_R$ *e12*

We also proved the following theorems about the infinitesimal basis elements, establishing the desired behaviour:

- **shows** *e1* $*$ *e1* = *0*
- **shows** *e1* $\neq$ *0*
- **shows** *e2* $*$ *e2* = *0*
- **shows** *e2* $\neq$ *0*
- **shows** *e12* $*$ *e12* = *0*
- **shows** *e12* $\neq$ *0*

See Appendix 6.8 for the full proofs.

We then proved hyperdual numbers to be an instance of the real normed vector type class *real-normed-vector* (see Appendix 6.9 for the full instantiation). All definitions required by this instantiation (e.g. uniformity filter) except for the norm were left as they are defined in other instantiations (e.g. the complex numbers). That is either in terms of the norm or of each other. The norm itself was defined as the Euclidian distance.

The proof of the instantiation consisted for the most part of short sub-proofs, except for the triangle inequality. Probably due to the complexity of the expression involved, this part required significant number of handwritten expression transformations to break down the proof into parts manageable by the automatic methods. This was a very slow process, as at one point the expression in question grew to over 600 characters, but thanks to the theorem prover all errors were immediately caught and resolved.

We also mechanized our counter-example to hyperduals being a real normed algebra (see Appendix 6.10).

3.7 Hypercomplex Numbers

The similarity of the hyperdual numbers to the complex numbers is not just coincidental. They are both instances of what are called hypercomplex numbers, as described by Kantor and Solodovnikov [19]. In the context of that book, second-order hyperdual numbers would be defined as a hypercomplex number system with $n = 3$ "imaginary units" ε_1, ε_2, $\varepsilon_1\varepsilon_2$ whose multiplication is defined by the following multiplication table:

	ε_1	ε_2	$\varepsilon_1\varepsilon_2$
ε_1	0	$\varepsilon_1\varepsilon_2$	0
ε_2	$\varepsilon_1\varepsilon_2$	0	0
$\varepsilon_1\varepsilon_2$	0	0	0

With this, the addition, subtraction and multiplication we define here follow from the definitions in the book. This was expected, because Fike and Alonso (who we took those definitions from) cite this book and so probably made sure to match the definitions from there.

Chapter 7 of the book talks about how hypercomplex numbers can be viewed as an algebra. This validates our view of hyperdual numbers as an algebra over the reals with basis $\{1, \varepsilon_1, \varepsilon_2, \varepsilon_1\varepsilon_2\}$ (see Section 3.6).

Chapter 9 of the book contains a proof that a hypercomplex system with non-trivial zero divisors does not form a division algebra (i.e. does not admit division on all non-zero elements). This is consistent with our finding that hyperduals have non-trivial zero divisors and do not form a division algebra (see Section 3.3.1).

Chapter 12 of the book talks about introducing a scalar (dot) product to the hypercomplex space and about the norm of hypercomplex numbers. The norm defined there is the same as the the one we use, which further validates our choice.

3.8 Bounded Linearity of Projections

Due to hyperdual addition and scalar multiplication being defined component-wise, and the hyperdual norm being a Euclidian distance, we can prove that the projections Re, Eps_1, Eps_2 and Eps_{12} are bounded linear maps [21].

More specifically, the linear part requires that they commute with addition and scalar multiplication, that is for any $x, y \in H$ and $r \in \mathbb{R}$:

$$\begin{aligned} Re(x+y) &= Re(x) + Re(y) \\ Re(r \cdot x) &= r \cdot Re(x) \end{aligned}$$

and similarly for all the other projections. This can be easily seen to be true from the definition of those two operations.

The bounded part requires that for any hyperdual the ratio of the norm of its projection to its norm is bounded from above. That is, there exists some $M \in \mathbb{R}$ with $M \geq 0$ such that for any $h \in H$:

$$\|Re(h)\| \leq M\|h\|$$

and similarly for all the other projections. In this case, we will see that $M = 1$. Recall the definition of the norm with projections instead of indices:

$$\|a\| = \sqrt{(Re(a))^2 + (Eps_1(a))^2 + (Eps_2(a))^2 + (Eps_{12}(a))^2}$$

The squared norm of each projection result is part of the argument of the square root on the right hand side. Moreover the argument of the square root is a sum of non-negative terms and square root is strictly increasing. Thus we have for any $h \in H$:

$$\begin{aligned} &\|Re(h)\|^2 \leq (Re(h))^2 + (Eps_1(h))^2 + (Eps_2(h))^2 + (Eps_{12}(h))^2 \\ \Rightarrow &\|Re(h)\| \leq \sqrt{(Re(h))^2 + (Eps_1(h))^2 + (Eps_2(h))^2 + (Eps_{12}(h))^2} \\ \Rightarrow &\|Re(h)\| \leq \|h\| \end{aligned}$$

and similarly for all the other projections. See Appendix 6.11 for the full proofs.

Bounded linearity of the projection is useful, because there are already proven theorems about bounded linear functions in Isabelle. We are in particular interested in the following properties:

Consider a bounded linear function f. Given another function g such that g tends to some value a, the composition $f \circ g$ tends to the value $f(a)$. This fact is useful when deriving behaviour of limits of hyperdual-valued functions (see Section 3.10).

Moreover, given a function g which has derivative g', the composition $f \circ g$ has derivative $f \circ g'$. This fact is useful when deriving behaviour of derivative of hyperdual-valued functions (see Section 3.11).

3.9 Filters

In order to introduce our remaining definitions related to limits (see Section 3.10) and derivatives (see Section 3.11), we need a momentary diversion to briefly introduce mathematical filters [34]. These are used to abstractly define these notions in Isabelle. While filters are not the only way of formalizing these concepts [14], Isabelle takes this as a general approach [17].

We spent quite a lot of time reading various introductions to the subject as well as investigating how filters are used in Isabelle, before we were comfortable working on the limits and derivatives of hyperdual functions. The most useful resource was the description by Hölzl et al. [17] which accompanied the introduction of these filter-based formalizations to Isabelle.

A filter generalizes the notion of the set of tails of a sequence. Given some set X, a filter F on X is a non-empty subset of the powerset $\mathcal{P}(X)$ of X such that:

- If A and B are in F then so is $A \cap B$ (F is closed under finite intersection)
- If A is in F and $A \subseteq B$ for some $B \subseteq X$ then B is in F (F is upward-closed)

In Isabelle, filters are defined as predicates on predicates:

locale *is-filter* =
 fixes $F :: ('a \Rightarrow bool) \Rightarrow bool$
 assumes *True*: $F\ (\lambda x.\ True)$
 assumes *conj*: $F\ (\lambda x.\ P\ x) \Longrightarrow F\ (\lambda x.\ Q\ x) \Longrightarrow F\ (\lambda x.\ P\ x \wedge Q\ x)$
 assumes *mono*: $\forall x.\ P\ x \longrightarrow Q\ x \Longrightarrow F\ (\lambda x.\ P\ x) \Longrightarrow F\ (\lambda x.\ Q\ x)$

typedef *'a filter* $= \{F :: ('a \Rightarrow bool) \Rightarrow bool.\ is\text{-}filter\ F\}$

The assumption *conj* corresponds to closure under finite intersection, while the assumption *mono* corresponds to being upward-closed. The assumption *True* corresponds to the set itself always being in the filter, which is subsumed by the filter being non-empty and upward-closed. Because Isabelle does not require filters to be non-empty (proper), the *True* assumption has to be stated explicitly. Otherwise this definition is equivalent to the one based on sets.

For our use, the most important filters are those based around neighbourhoods of a point. The neighbourhoods filter for a point a is defined in Isabelle as follows:

definition (**in** *topological-space*) *nhds* :: *'a* ⇒ *'a filter*
 where $nhds\ a = (INF\ S{\in}\{S.\ open\ S \wedge a \in S\}.\ principal\ S)$

In this definition, *open* is a predicate that is true if and only if the set is open, and *principal* refers to the principal filter of a set which is the smallest filter that contains the set – the set of all of its supersets.

This filter holds for all open sets in the space that contain a. It corresponds to the predicate "for all y in some open neighbourhood of a". Then the *punctured* neighbourhood filter for a point a within some set S is defined as follows:

definition (**in** *topological-space*) *at-within* :: *'a* ⇒ *'a set* ⇒ *'a filter*
  (*at* (-)/ *within* (-) [*1000*, *60*] *60*)
 where $at\ a\ within\ S = inf\ (nhds\ a)\ (principal\ (S - \{a\}))$

This filter corresponds to the predicate "for all $y \in S$ and $y \neq a$ in some neighbourhood of x" [17].

There is also a special case of the punctured neighbourhood filter *at a within S* for when S is the universal open set (i.e. the whole type):

abbreviation (**in** *topological-space*) *at* :: *'a ⇒ 'a filter* (*at*)
  where *at x ≡ at x within* (*CONST UNIV*)

Next we describe the convergence relation *tendsto* ($\longrightarrow$). Consider any function $f : \alpha \to \beta$ where α is an arbitrary type and β is a topological space. Then f converging to $l \in \beta$ on a filter F on α is expressed in Isabelle as $(f \longrightarrow l)\ F$ This can express many kinds of limits by using different filters, for example:

- $(f \longrightarrow L)$ $(at\ a)$ expresses $\lim_{x \to a} f(x) = L$
- $(f \longrightarrow L)$ $(at\text{-}right\ a)$ expresses $\lim_{x \to a^+} f(x) = L$
- $(f \longrightarrow L)$ $(at\text{-}bot)$ expresses $\lim_{x \to -\infty} f(x) = L$

3.10 Limits

Consider any function $j : \alpha \to H$ from any arbitrary type α to the hyperduals. Let it be corecursively defined via real-valued functions $f, g, h, i : \alpha \to \mathbb{R}$:

$$j(x) = f(x) + g(x) \cdot \varepsilon_1 + h(x) \cdot \varepsilon_2 + i(x) \cdot \varepsilon_1 \varepsilon_2$$

Then if f, g, h, i converge to $a, b, c, d \in \mathbb{R}$ respectively, j converges to $e \in H$ such that:

$$e = a + b \cdot \varepsilon_1 + c \cdot \varepsilon_2 + d \cdot \varepsilon_1 \varepsilon_2$$

On the other hand, if j converges to some $e \in H$, then $Re \circ j$ converges to $Re(e)$ and similarly for all the other projections. This is because the projections only project onto the relevant axis.

From these two implications, for any hyperdual-valued function $j : \alpha \to H$ we have:

$$\begin{aligned} (j \longrightarrow e) \longleftrightarrow (&Re \circ j \longrightarrow Re(e) \wedge \\ &Eps_1 \circ j \longrightarrow Eps_1(e) \wedge \\ &Eps_2 \circ j \longrightarrow Eps_2(e) \wedge \\ &Eps_{12} \circ j \longrightarrow Eps_{12}(e)) \end{aligned} \tag{3.6}$$

3.10.1 Mechanization

The equivalence (3.6) was mechanized in Isabelle using the *tendsto* ($\longrightarrow$) convergence relation for an arbitrary filter (see Section 3.9):

lemma *tendsto-hyperdual-iff*:
 ((*f* :: *'a ⇒ hyperdual*) ⟶ *x*) *F*
  ⟷ (((λ*x*. *Re* (*f x*)) ⟶ *Re x*) *F*
     ∧ ((λ*x*. *Eps1* (*f x*)) ⟶ *Eps1 x*) *F*
     ∧ ((λ*x*. *Eps2* (*f x*)) ⟶ *Eps2 x*) *F*
     ∧ ((λ*x*. *Eps12* (*f x*)) ⟶ *Eps12 x*) *F*)

The proof first separately establishes the right to left implication. This is then extended to the full equivalence using bounded linearity of projections (see Section 3.8). The skeleton of this proof was inspired by how this property is proven for complex numbers in Isabelle. See Appendix 6.12 for the full proof.

With the notion of hyperdual limits we can also prove that hyperduals are an instance of Isabelle's Banach space [33] (complete normed vector space) type class *banach*. For this instantiation we only need to prove completeness, which is that every Cauchy sequence [33] converges to a hyperdual number. We prove this similarly to how it is proven for complex numbers in Isabelle. See Appendix 6.13 for full instantiation.

3.11 Derivatives

We use the Fréchet derivative [9] as the notion of derivative for hyperdual-valued functions. This choice is motivated by Isabelle's mechanization of derivative for real normed vector spaces being defined that way. Also, because hyperdual numbers do not form a real normed field (see 3.5) we cannot use the simpler notion of field derivative, which is also defined in Isabelle.

Let $\beta \subseteq \alpha$ be an open subset of a real normed vector space α. The Fréchet derivative of $j : \beta \to H$ at some $x \in \beta$ is a bounded linear map $A : \alpha \to H$ such that:

$$\lim_{\|h\|_\alpha \to 0} \frac{\|j(x+h) - j(x) - A(h)\|_H}{\|h\|_\alpha} = 0 \tag{3.7}$$

If such an A exists, we write $Dj(x) = A$. Note that the type of this function is:

$$Dj : \beta \to L(\alpha, H)$$

where $L(\alpha, H)$ is the space of all bounded linear maps from α to H.

From this point on, when referring to a derivative for hyperdual-valued functions we mean the Fréchet derivative.

One notable property of this notion of derivative is that the chain rule is subsumed in the composition of derivatives. That is, if $j : \beta \to \gamma$ has derivative $Dj(x)$ at some $x \in \beta$ and $k : \gamma \to \delta$ has derivative $Dk(j(x))$ at $y = j(x)$, then the composition $k \circ j$ has derivative $Dk(j(x)) \circ Dj(x)$.

Using the limit equivalence (3.6), we can decompose definition (3.7) in terms of limits on the component functions. This then gives us an equivalence between the derivative relation of the overall functions and those of the component functions. That is, any hyperdual-valued function $j : \beta \to H$ has derivative $Dj(x) : \alpha \to H$ if and only if:

- $Re \circ j$ has derivative $Re \circ Dj(x)$, and
- $Eps_1 \circ j$ has derivative $Eps_1 \circ Dj(x)$, and
- $Eps_2 \circ j$ has derivative $Eps_2 \circ Dj(x)$, and
- $Eps_{12} \circ j$ has derivative $Eps_{12} \circ Dj(x)$

3.11.1 Mechanization

The difficulty with the mechanization of this property was our initial lack of familiarity with Fréchet derivatives. These are quite different from the usual notion of derivative of real functions in both their statement and behaviour. We spent multiple weeks reading theory, working through examples and investigating how they are used in Isabelle. Once again, the description by Hölzl et al. [17] was very useful here. The different nature of this notion of derivative also shows later in one mechanization of the hyperdual extension (see Section 4.4.2), where it complicates our ability to express second derivatives.

The general derivative is defined in Isabelle by the following relation:

definition *has-derivative* ::
$('a::\mathit{real\text{-}normed\text{-}vector} \Rightarrow 'b::\mathit{real\text{-}normed\text{-}vector}) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a\ \mathit{filter} \Rightarrow \mathit{bool}$
where $(f\ \mathit{has\text{-}derivative}\ f')\ F \longleftrightarrow \mathit{bounded\text{-}linear}\ f' \wedge$
$((\lambda y.\ ((f\ y - f\ (\mathit{Lim}\ F\ (\lambda x.\ x))) - f'\ (y - \mathit{Lim}\ F\ (\lambda x.\ x)))\ /_R\ \mathit{norm}\ (y - \mathit{Lim}\ F\ (\lambda x.\ x)))$
$\longrightarrow 0)\ F$

This relates the function f to the bounded linear function f' if f' is a derivative of f on the filter F. Usually this filter is some form of *at a within S*, where a is the point of differentiation (see Section 3.9). Then f' is the Fréchet derivative of f at a in the sense of equation (3.7) where β represents S.

There is also a notion of field derivative defined in Isabelle, which is simpler to use in some aspects but assumes the functions are operators on a real normed field:

definition *has-field-derivative* :: $('a::\mathit{real\text{-}normed\text{-}field} \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a\ \mathit{filter} \Rightarrow \mathit{bool}$
where $(f\ \mathit{has\text{-}field\text{-}derivative}\ D)\ F \longleftrightarrow (f\ \mathit{has\text{-}derivative}\ (*)\ D)\ F$

Due to hyperduals not forming a real normed field, the derivative relation we have to use for general hyperdual-valued functions is *has-derivative*.

We mechanize the derivative equivalence as follows:

lemma *has-derivative-hyperdual-iff*: $(f\ \mathit{has\text{-}derivative}\ f')\ F \longleftrightarrow$
$((\lambda x.\ \mathit{Re}\ (f\ x))\ \mathit{has\text{-}derivative}\ (\lambda x.\ \mathit{Re}\ (f'\ x)))\ F \wedge$
$((\lambda x.\ \mathit{Eps1}\ (f\ x))\ \mathit{has\text{-}derivative}\ (\lambda x.\ \mathit{Eps1}\ (f'\ x)))\ F \wedge$
$((\lambda x.\ \mathit{Eps2}\ (f\ x))\ \mathit{has\text{-}derivative}\ (\lambda x.\ \mathit{Eps2}\ (f'\ x)))\ F \wedge$
$((\lambda x.\ \mathit{Eps12}\ (f\ x))\ \mathit{has\text{-}derivative}\ (\lambda x.\ \mathit{Eps12}\ (f'\ x)))\ F$

The proof of the equivalence first establishes the left to right implication, decomposing the relation onto the component functions. This uses the fact that the projections are bounded linear (see Section 3.8).

Going from the relations of the component functions to the relation of the hyperdual-valued functions is more difficult and has two parts. First, the equivalence for limits (see Section 3.10.1) is used with the definition of the relation to transfer the convergence part from the component functions to the hyperdual functions. Second, the bounded linearity of the component derivatives is used to establish the bounded linearity of the overall derivative. This second step is quite complicated, mainly because of the chain of complex expression rewriting it requires. See Appendix 6.14 for the full proof.

3.12 Summary

In this chapter we described the main operations on hyperdual numbers and their properties. To maximize reuse of known facts, we showed that hyperdual numbers are an instance of known algebraic structures where possible. We then derived rules about limits and derivatives of hyperdual-valued functions. We also described how we mechanized these definitions and proofs in Isabelle.

In the next chapter, we describe hyperdual extensions of real functions. These appear to be a natural way of translating properties of real operations to the realm of hyperduals.

Chapter 4

Hyperdual Extension of Real Functions

We define a useful class of hyperdual functions, each based on a real function, which we call the hyperdual extensions of those real functions. These extensions have properties which are useful for automatic differentiation of the underlying real function.

The interesting aspect is that for a lot of operations, the operation on hyperduals is a hyperdual extension of the same operation performed on the reals. For example this holds for addition, multiplication and multiplicative inverse. This suggests that this is not as much a structure built on hyperdual numbers as their inherent property.

First, we describe the general context of automatic differentiation and two other popular approaches to the problem. Second, we motivate the desired properties of the hyperdual extension. Third, we derive the definition of the hyperdual extension that obeys those properties. Fourth, we discuss some further properties of the hyperdual extension. Last, we describe the mechanization.

The definition discussed here is due to Fike and Alonso [10], but we go beyond this and further formalize its derivation and formally prove its the properties.

4.1 Automatic Differentiation

In a number of applications, we desire highly-accurate first and second derivatives. Using symbolic differentiation is not always desirable, for example due to complex control flow or changing data structures [4]. Numerical methods like finite-difference [23] and complex-step [24] approximations, which we review briefly further down, are subject to truncation [23] and subtraction-cancellation [16] errors.

Automatic differentiation [3] instead follows the execution of the program, using known derivatives of the elementary operations performed and the chain rule to compute the derivative of the function computed by the program. This is inherently compatible with complicated control flow and changing data, because only the path truly executed is differentiatied. Using hyperdual numbers for this task in the way we describe also introduces no truncation or subtraction-cancellation errors.

Finite-difference method finds the first derivative of a function by evaluating it at two sufficiently close points and computing the gradient between them. This can be derived from the Taylor series. For example the central-difference approximation is given by subtracting the expansions of $f(x+d)$ for real steps $d = \pm h$:

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2)$$

Complex-step method takes instead a complex step $d = ih$. The Taylor series derivation then yields the following expression:

$$f(x+ih) = \left(f(x) - \frac{1}{2!}h^2 f''(x) + \ldots\right) + ih\left(f'(x) - \frac{1}{3!}h^3 f'''(x) + \ldots\right)$$

The complex-step approximation of first derivative is then:

$$f'(x) = \frac{Imag[f(x+ih)]}{h} + O(h^2)$$

and for the second derivative it is:

$$f''(x) = \frac{2\left(f(x) - Real[f(x+ih)]\right)}{h^2} + O(h^2)$$

All the above approximations involve truncation error, because of the infinite nature of the Taylor series. All the higher-order terms (than the derivative we are approximating) are neglected yielding the truncation error, in all these cases of the order $O(h^2)$.

Finite-step first derivative approximation and complex-step second derivative approximation also involve subtraction-cancellation error. As we make the step smaller (decreasing h), the subtracted values get closer. Because of finite precision of the real number representation in computers, for a sufficiently low h two distinct value will yield zero difference. This puts a limit on how accurate this approximation can be.

We will show that using hyperdual numbers we can avoid both of these errors. This makes them well suited to high-precision automatic differentiation.

4.2 Basic Desired Properties

The properties we desire of the resulting function are rooted in the target use, automatic differentiation. They are meant to:

- Ensure the extension preserves the behaviour of the original function (allowing safe substitution for the real function in computations)
- Provide access to accurate derivatives after evaluation
- Preserve the properties on composition, allowing construction for arbitrary combinations

First, we want the real component to behave exactly as it would under the original function, i.e. $Re(g(x)) = f(Re(x))$. This means that if we replace f by g in a computation (and appropriately change relevant types) and then only look at the real component of the result, nothing changes. Thus the same computation still happens on the real component of the hyperdual variable as on the original real variable.

Second, we want to be able to extract exact values of the first and second derivatives from the result of evaluating the extension. This means that there should be no error terms or infinite sums which would introduce *truncation error*. The derivatives should also be the leading terms in their components, otherwise they would be subject to *subtraction cancellation error*.

Last, these properties should be preserved under function composition. This is important because we wish to compose simple functions to construct arbitrarily complex functions, and we wish the result to have the above properties as well. This is at the core of how automatic differentiation works.

4.3 Derivation

The derivation of the extension definition is based on that described by Fike and Alonso [10] which is quite informal. We try to make it as formal as possible in order to get closer to mechanizing it. Although our improvements are still not sufficient to fully formalize the derivation, some gaps in the original derivation have been fixed. We assume the existence of the two functions and proceed to verify under what conditions g has the desired properties relative to f.

Let $f : \mathbb{R} \to \mathbb{R}$ and $g : H \to H$. By the hyperdual Taylor's expansion[1]we have for all $y, b \in H$:

$$g(y+b) = g(y) + \frac{b}{1!}g'(y) + \frac{b^2}{2!}g''(y) + \frac{b^3}{3!}g'''(y) + \dots$$

Next, note that for all $x \in H$ and $n \in \mathbb{N}$ such that $x_1 = 0$ and $n \geq 3$ we have $x^n = 0$, because all components of the product involve a square of some infinitesimal.

Also, label the components of y and b as done previously for other hyperduals:

$$y = y_1 + y_2\varepsilon_1 + y_3\varepsilon_2 + y_4\varepsilon_1\varepsilon_2$$
$$b = b_1 + b_2\varepsilon_1 + b_3\varepsilon_2 + b_4\varepsilon_1\varepsilon_2$$

Thus if we constrain $b_1 = 0$, we get the *finite* expansion:

$$g(y+b) = g(y) + \frac{b}{1!}g'(y) + \frac{b^2}{2!}g''(y)$$

[1]Here we assume a certain form of the Taylor's expansion for hyperduals, as well as skip any conditions it might have (e.g. radius of convergence). This is also glossed over by Fike and Alonso. We note this for future investigation in Section 5.1.3.

Then expanding b and b^2, simplifying the expression and factoring it in terms of the components, we get:

$$g(y+b) = g(y) + b_2 g'(y) \cdot \varepsilon_1 + b_3 g'(y) \cdot \varepsilon_2 + (b_4 g'(y) + b_2 b_3 g''(y)) \cdot \varepsilon_1\varepsilon_2 \tag{4.1}$$

Next we constrain $y_2 = y_3 = y_4 = 0$, then assume that:

$$\begin{aligned} g(y) &= f(y_1) + 0\varepsilon_1 + 0\varepsilon_2 + 0\varepsilon_1\varepsilon_2 \\ g'(y) &= f'(y_1) + 0\varepsilon_1 + 0\varepsilon_2 + 0\varepsilon_1\varepsilon_2 \\ g''(y) &= f''(y_1) + 0\varepsilon_1 + 0\varepsilon_2 + 0\varepsilon_1\varepsilon_2 \end{aligned}$$

and note that multiplying a hyperdual number by $a + 0\varepsilon_1 + 0\varepsilon_2 + 0\varepsilon_1\varepsilon_2$ for some $a \in \mathbb{R}$ is the same as scaling that number by a.

Then we get the following by substituting into equation (4.1) and simplifying:

$$g(y+b) = f(y_1) + b_2 f'(y_1) \cdot \varepsilon_1 + b_3 f'(y_1) \cdot \varepsilon_2 + (b_4 f'(y_1) + b_2 b_3 f''(y_1)) \cdot \varepsilon_1\varepsilon_2 \tag{4.2}$$

Next we reinterpret the argument of g in this equation as a single number instead of the addition of two numbers. Consider an arbitrary hyperdual number $x \in H$ with components labeled as usual:

$$x = x_1 + x_2\varepsilon_1 + x_3\varepsilon_2 + x_4\varepsilon_1\varepsilon_2$$

Then we can get $x = y + b$ by setting the remaining unconstrained components of y and b as follows:

$$y_1 = x_1 \qquad b_2 = x_2 \qquad b_3 = x_3 \qquad b_4 = x_4$$

Note that this decomposition satisfies the constraints we put on y and b throughout this derivation for any x. Then substituting into equation (4.2) we get:

$$g(x) = f(x_1) + x_2 f'(x_1) \cdot \varepsilon_1 + x_3 f'(x_1) \cdot \varepsilon_2 + (x_4 f'(x_1) + x_2 x_3 f''(x_1)) \cdot \varepsilon_1\varepsilon_2 \tag{4.3}$$

Now if we evaluate g at $a + 1\varepsilon_1 + 1\varepsilon_2 + 0\varepsilon_1\varepsilon_2$ for some $a \in \mathbb{R}$ we get:

$$g(a + 1 \cdot \varepsilon_1 + 1 \cdot \varepsilon_2 + 0 \cdot \varepsilon_1\varepsilon_2) = f(a) + f'(a) \cdot \varepsilon_1 + f'(a) \cdot \varepsilon_2 + f''(a) \cdot \varepsilon_1\varepsilon_2 \tag{4.4}$$

Note that in the above expression, each derivative (f, f', f'') is the leading term in at least one component and the overall expression is finite. Moreover, the real component of g is equal to f and it is just a matter of substitution to see that this form is preserved under composition. Thus g satisfies the desired properties relative to f.

4.4 Mechanization

After working through the derivation above, we mechanized the resulting definition as a locale in Isabelle. This locale relates a real function f and its first and second derivatives, f' and f'', with a hyperdual function g, when g is a valid hyperdual extension

f. Within this locale we prove general properties of hyperdual extensions, for example how the values of derivatives can be extracted and that it is preserved by composition. Then we prove a number of concrete cases of extensions. For the operations we examined in Chapter 3 (e.g. addition) we prove that they are hyperdual extensions of the corresponding operations on the real numbers. When considering additional functions over the hyperduals (e.g. sine), we define new functions based on the hyperdual extension definition.

We have two versions of the locale representing the hyperdual extension. One uses the *has-field-derivative* relation (see Section 3.11.1) and thus assumes that the components of hyperdual numbers are from a real normed field. This version is much more complete, because this assumption allows us to more easily express the second derivative. Therefore we use this version for our case studies in Section 4.4.3.

The other version uses the more general *has-derivative* relation (see Section 3.11.1) which only assumes the components of hyperdual numbers are from a real normed vector space. This version is more general, allowing some future experimentation with component types. The drawback is that because this is the pure Fréchet derivative, the second derivative is taken on the space of bounded linear functions. This makes it difficult to work with in Isabelle, because it requires we wrap the function type with the bounded linearity assumption for it to form a real normed vector space and be compatible with the relation. For that reason, this version is still very much in development. We will come back to this when we discuss future work in Section 5.1.1.3.

Next we describe how the field derivative version of the locale is mechanized, including the theorems proven within it. We then give an overview of the general version of the locale, the known challenges and outline our plan for it. Last we describe the concrete cases of extensions we have proven, going into detail on a selected few of those proofs.

4.4.1 Field Version of the Extension Locale

This version of the locale uses the *has-field-derivative* relation (see Section 3.11.1) which requires the function to be within one real normed field, in this case the reals. This allows us to express the second derivative more intuitively than with the more general *has-derivative* relation (as mentioned at the end of the previous section), because the first derivative function has the same type as the original function.

First we set up a locale for the second derivative of a function:

```
locale has-snd-field-derivative =
  fixes f :: ('a :: real-normed-field) ⇒ 'a
    and f' :: 'a ⇒ 'a
    and f'' :: 'a ⇒ 'a
    and S :: 'a set
    and a :: 'a
  assumes deriv-f: (f has-field-derivative f' a) (at a within S)
      and deriv-f': (f' has-field-derivative f'' a) (at a within S)
```

This locale simply establishes that the functions f', f'' are successive derivatives of f. It does this given the point of differentiation a and a set S. These are used in the filters (see Section 3.9) on which the derivative is assumed. Exposing these as parameters allows us to control the filter when instantiating this locale. This is for example used in the preservation on composition theorem later in this section, which assumes the outer function extension is valid (at least) on the range of the inner function.

We then prove that this locale is preserved on taking a subset for the filter set (see Appendix 6.15 for proof):

lemma *has-snd-field-derivative-subset*:
 assumes $T \subseteq S$
  and *has-snd-field-derivative* $f\,f'\,f''\,S\,a$
 shows *has-snd-field-derivative* $f\,f'\,f''\,T\,a$

Next we extend that locale to tie in the hyperdual extension, as defined in (4.3):

locale *hyperdual-ext* =
 has-snd-field-derivative $f\,f'\,f''\,S\,a$
  for f **and** f' **and** f'' **and** S **and** a +
 fixes g :: *hyperdual* $\Rightarrow$ *hyperdual*
 assumes *re-g*: $Re\ (g\ x) = f\ (Re\ x)$
  and *eps1-g*: $Eps1\ (g\ x) = Eps1\ x * f'\ (Re\ x)$
  and *eps2-g*: $Eps2\ (g\ x) = Eps2\ x * f'\ (Re\ x)$
  and *eps12-g*: $Eps12\ (g\ x) = Eps12\ x * f'\ (Re\ x) + Eps1\ x * Eps2\ x * f''\ (Re\ x)$

This extension establishes componentwise that the proposed extension function is equal to the definition (4.3) we derived. We then proceed to prove its properties within that locale (see Appendix 6.16 for full proofs).

First, we prove three lemmas about how each derivative can be extracted from the result of evaluating the extension on the argument indicated in the derivation (4.4):

lemma *extract-f*:
 shows $Re\ (g\ (Hyperdual\ x\ 1\ 1\ 0)) = f\ x$

lemma *extract-f'*:
 shows $Eps1\ (g\ (Hyperdual\ x\ 1\ 1\ 0)) = f'\ x$
  and $Eps2\ (g\ (Hyperdual\ x\ 1\ 1\ 0)) = f'\ x$

lemma *extract-f''*:
 shows $Eps12\ (g\ (Hyperdual\ x\ 1\ 1\ 0)) = f''\ x$

These ensure that the values of the derivatives can be extracted in the expected way.

Next, we prove that an extension stays an extension if we use the subset of the original set for the filter:

lemma *subset*:
 assumes $T \subseteq S$
 shows *hyperdual-ext* $f\,f'\,f''\,T\,a\,g$

The proof uses the same fact we showed earlier in this section for the second field derivative locale and the fact that we are not changing anything about the functions involved. This theorem is vital when we need the sets used by different extensions to be the same (see for example case study in Section 4.4.3.4).

Next, we prove that the composition with another hyperdual extension is the hyperdual extension of the composition of the underlying functions:

lemma *compose*:
  assumes *hyperdual-ext* $m\ m'\ m''\ (f`S)\ (f\,a)\ n$
  shows *hyperdual-ext* $(\lambda x.\ m(f\,x))\ (\lambda x.\ m'(f\,x) * f'\,x)\ (\lambda x.\ m'(f\,x) * f''\,x + m''(f\,x) * f'\,x * f'\,x)\ S\ a\ (\lambda x.\ n(g\,x))$

This allows us to take two proven hyperdual extensions, compose them and easily prove that the result is also a hyperdual extension (see case study in Section 4.4.3.4).

Last, we prove the uniqueness of an extension given the function with its derivatives:

lemma *unique*:
  assumes *hyperdual-ext* $f\,f'\,f''\ S\ a\ h$
  shows $g = h$

The proof uses the fact that both g and h have to be equal to the same componentwise definition, and thus are equal. However, so far we have been unable to prove the uniqueness of the derivative functions of f, therefore we are not able prove a stronger claim than this. We do however note this for future investigation (see Section 5.1.1.2).

4.4.2 General Version of the Extension Locale

As mentioned, the version of the locale using the more general *has-derivative* relation is still in development. It is planned to follow the same lines as the field version, proving variants of the same lemmas. Its validation will be all the examples proven for the field version, thus checking that it is at least as good. This will also provide a demonstration of whether this version leads to simpler proofs.

The advantage of using the *has-derivative* relation is that the chain rule is subsumed in function composition, while for *has-field-derivative* it has to be explicitly applied. We have found that the chain rule considerably complicates expressions within proofs using the field locale. For example, the preservation under composition proof for (the current version of) the general locale is less than half the length of the field locale proof (17 lines instead of 39 lines).

The difficulty with this approach is expressing the second derivative. The derivative of a function $f : U \to W$ with $U \subset V$ is $Df : U \to L(V,W)$ where $L(V,W)$ is the space of all bounded linear maps from V to W. To be able to talk about second derivatives of f, we first need to be able to talk about the functions we get as the first derivative as elements of $L(V,W)$ with $L(V,W)$ proven to be a real normed vector space. Fortunately, Isabelle includes the theory *Bounded Linear Function* that allows us to talk about such functions. This theory is for example used in the mechanization of ordinary differential equations by Immler and Hölzl [18].

We go over our plans for future work on this implementation in Section 5.1.1.3.

4.4.3 Case Studies

We have mechanized hyperdual extensions for a number of functions using the field derivative version of the extension locale. We first give a brief overview of all the proven cases, then we describe details of a few select proofs. At the end of this section, we outline our effort to mechanize the analytical test function used by Fike and Alonso [10].

The following hyperdual operations were proven to be extensions of their real counterparts (see Appendix 6.17 for the proofs):

- Constant function
- Identity function
- Scalar multiplication
- Addition
- Linear function
- Natural power
- Multiplication
- Multiplicative inverse
- Finite polynomial

The fact that the functions were defined independently of the hyperdual extension still satisfy it strongly suggests that it is a property of the number system rather than structure built on them. We intend to investigate this relationship further in future, because it might lead to some interesting results (see Section 5.1.1.4.

Moreover, we defined new functions to form the extensions of the following real functions (see Appendix 6.17 for definitions and proof):

- Square root
- Exponential function
- Sine and cosine

We tried to perform any proofs where it is possible using the preservation under composition and prior extensions, in order to better demonstrate the viability of this approach. This ensured that we have proven all the theorems required to easily perform such proofs.

We will now give detailed descriptions of proofs of some of the extensions.

4.4.3.1 Identity Function

First we look at the proof that the hyperdual identity function is an extension of the real identity function:

lemma *hyperdual-ext-identity*:
  hyperdual-ext $(\lambda x.\ x)$ $(\lambda x.\ 1)$ $(\lambda x.\ 0)$ *UNIV* x $(\lambda x.\ x)$
proof

This proof is a great example of proving an extension directly, that is by the assumptions of the locale(s) (see Section 4.4.1 for definitions). First, we need to show that the real functions f' and f'' are first and second derivatives of f. In this case, these facts are already proven in Isabelle as part of the derivative mechanization, so we only need to refer to those:

  show $((\lambda x.\ x)$ *has-field-derivative* $1)$ $($*at* $x)$
    using *has-derivative-ident* **by** *simp*
  show $((\lambda x.\ 1)$ *has-field-derivative* $0)$ $($*at* $x)$
    by *simp*

Then we need to show that the hyperdual function is equal to what the locale assumes it to be. In this case, these are simplifications that the automatic simplifier can perform without assistance:

  show $\bigwedge x.\ Re\ x = Re\ x$
    and $\bigwedge x.\ Eps1\ x = Eps1\ x * 1$
    and $\bigwedge x.\ Eps2\ x = Eps2\ x * 1$
    and $\bigwedge x.\ Eps12\ x = Eps12\ x * 1 + Eps1\ x * Eps2\ x * 0$
    by *simp-all*
qed

This proof is one of the easiest among the extensions. It does not require any complicated reasoning, because all the assumptions of the locales are already proven by other theorems. We only need to bring those together.

4.4.3.2 Addition

Next we look at the proof that hyperdual addition of extensions is an extension of real addition. This is a good example of how to approach binary functions. For unary functions we do not need to assume that the function we are working with is being composed with another one, because we have the preservation on composition theorem to then derive a rule for that case. This is however required for binary functions. Intuitively, we approach binary functions as combining two extensions into one. In this we take inspiration from how derivatives of such functions are already mechanized in Isabelle.

The claim is then as follows:

lemma *hyperdual-ext-add*:
  assumes *hyperdual-ext* $f\ f'\ f''\ S\ a\ fE$
    and *hyperdual-ext* $g\ g'\ g''\ S\ a\ gE$
  shows *hyperdual-ext* $(\lambda x.\ f\ x + g\ x)$ $(\lambda x.\ f'\ x + g'\ x)$ $(\lambda x.\ f''\ x + g''\ x)$ $S\ a$ $(\lambda x.\ fE\ x + gE\ x)$
proof

This proof again proceeds directly by verifying the assumptions of the locales, starting with the real functions f' and f'' being first and second derivatives of f. In this case, the relevant facts are already proven theorems in Isabelle. We do however instantiate them with the relevant terms before applying the automated method to make the proof tractable for the simplifier:

```
 show ((λx. f x + g x) has-field-derivative f′ a + g′ a) (at a within S)
   using assms hyperdual-ext-def
       Deriv.field-differentiable-add[of f f′ a at a within S g g′ a]
   by (simp add: has-snd-field-derivative-def)
 show ((λx. f′ x + g′ x) has-field-derivative f″ a + g″ a) (at a within S)
   using hyperdual-ext-def assms
       Deriv.field-differentiable-add[of f′ f″ a at a within S g′ g″ a]
   by (simp add: has-snd-field-derivative-def)
```

Finally, we again need to show that the hyperdual function is equal to what the locale assumes it to be. We first fix the variable, otherwise we would have to reason under a binder which complicates the reasoning required. With the use of properly instantiated facts about the extensions we are adding, this proof can then be handled by the automatic methods:

```
 fix x :: hyperdual
 show Re (fE x + gE x) = f (Re x) + g (Re x)
   using assms hyperdual-ext.re-g by auto
 show Eps1 (fE x + gE x) = Eps1 x * (f′ (Re x) + g′ (Re x))
   using assms
       hyperdual-ext.eps1-g[of f f′ f″ S a fE x]
       hyperdual-ext.eps1-g[of g g′ g″ S a gE x]
   by (simp add: distrib-left)
 show Eps2 (fE x + gE x) = Eps2 x * (f′ (Re x) + g′ (Re x))
   using assms
       hyperdual-ext.eps2-g[of f f′ f″ S a fE x]
       hyperdual-ext.eps2-g[of g g′ g″ S a gE x]
   by (simp add: distrib-left)
 show Eps12 (fE x + gE x) = Eps12 x * (f′ (Re x) + g′ (Re x)) + Eps1 x * Eps2 x * (f″ (Re
x) + g″ (Re x))
   using assms
       hyperdual-ext.eps12-g[of f f′ f″ S a fE x]
       hyperdual-ext.eps12-g[of g g′ g″ S a gE x]
   by (simp add: distrib-left)
qed
```

This proof is still relatively simple, because the differentiation rule for addition itself is quite simple. The complicated steps are already proven as theorems in Isabelle and the only complication is properly instantiating them so that the automatic methods can handle the rest.

4.4.3.3 Multiplication

Next we will look at the proof that hyperdual multiplication of extensions is an extension of real multiplication. We will contrast this proof to the previous one for addition, as they are both similar in structure. However, multiplication has a considerably more complicated differentiation rule than addition, which will be reflected in considerably more complicated expressions.

The claim is as follows:

lemma *hyperdual-ext-times*:
  assumes *hyperdual-ext* $f\,f'\,f''\,S\,a\,fE$
    and *hyperdual-ext* $g\,g'\,g''\,S\,a\,gE$
  shows *hyperdual-ext* $(\lambda x.\, f\,x * g\,x)\ (\lambda x.\, f'\,x * g\,x + f\,x * g'\,x)\ (\lambda x.\, f''\,x * g\,x + f'\,x * g'\,x + f'\,x * g'\,x + f\,x * g''\,x)\ S\ a\ (\lambda x.\, fE\,x * gE\,x)$
proof

As with addition. we proceed directly by verifying the assumptions of the locales. The first derivative assumption only requires the instantiation of the general multiplication derivative rule already mechanized in Isabelle and facts about the assumed extensions being multiplied. The second derivative is more complicated, because it now involves addition of two multiplications. However, with careful instantiations of the relevant addition and multiplication differentiation rules and facts about the extensions being multiplied, this can also be solved by automatic methods:

  show $((\lambda x.\, f\,x * g\,x)$ *has-field-derivative* $f'\,a * g\,a + f\,a * g'\,a)$ $($*at a within S*$)$
   using *assms has-snd-field-derivative.deriv-f add.commute*
     DERIV-mult$'$[*of* $f\,f'\,a\,a\,S\,g\,g'\,a$]
   by (*metis* (*full-types*) *hyperdual-ext-def*)
  show $((\lambda x.\, f'\,x * g\,x + f\,x * g'\,x)$ *has-real-derivative* $f''\,a * g\,a + f'\,a * g'\,a + f'\,a * g'\,a + f\,a * g''\,a)$ $($*at a within S*$)$
   using *assms has-snd-field-derivative.deriv-f hyperdual-ext-def*
     DERIV-add[*of* $\lambda x.\, f'\,x * g\,x\,f''\,a * g\,a + f'\,a * g'\,a\,a\,S\,\lambda x.\, f\,x * g'\,x\,f'\,a * g'\,a + f\,a * g''\,a$]
     DERIV-mult$'$[*of* $f'\,f''\,a\,a\,S\,g\,g'\,a$]
     DERIV-mult$'$[*of* $f\,f'\,a\,a\,S\,g'\,g''\,a$]
     add.assoc
     has-snd-field-derivative-def
   by *smt*

For the extension function equality proofs, we once again fix the variable to avoid reasoning under a binder. We then carefully instantiate relevant facts about the multiplied extensions for each equation. With the addition of the set of algebraic simplifications available in Isabelle, each of these equations can then be proven by the automatic simplifier:

  fix x :: *hyperdual*
  show $Re\ (fE\,x * gE\,x) = f\ (Re\,x) * g\ (Re\,x)$
   using *hyperdual-ext.re-g assms*
   by *simp*
  show $Eps1\ (fE\,x * gE\,x) = Eps1\,x * (f'\ (Re\,x) * g\ (Re\,x) + f\ (Re\,x) * g'\ (Re\,x))$
   using *add-mult-distrib mult.commute add.commute*
     hyperdual-ext.eps1-g[*of* $f\,f'\,f''\,S\,a\,fE\,x$]
     hyperdual-ext.re-g[*of* $f\,f'\,f''\,S\,a\,fE\,x$]
     hyperdual-ext.eps1-g[*of* $g\,g'\,g''\,S\,a\,gE\,x$]
     hyperdual-ext.re-g[*of* $g\,g'\,g''\,S\,a\,gE\,x$]
     assms
   by (*simp add*: *algebra-simps*)

show *Eps2* $(fE\ x * gE\ x) = Eps2\ x * (f'\ (Re\ x) * g\ (Re\ x) + f\ (Re\ x) * g'\ (Re\ x))$
using *add-mult-distrib mult.commute add.commute*
hyperdual-ext.eps2-g$[of\ f\ f'\ f''\ S\ a\ fE\ x]$
hyperdual-ext.re-g$[of\ f\ f'\ f''\ S\ a\ fE\ x]$
hyperdual-ext.eps2-g$[of\ g\ g'\ g''\ S\ a\ gE\ x]$
hyperdual-ext.re-g$[of\ g\ g'\ g''\ S\ a\ gE\ x]$
assms
by (*simp add*: *algebra-simps*)
show *Eps12* $(fE\ x * gE\ x) = Eps12\ x * (f'\ (Re\ x) * g\ (Re\ x) + f\ (Re\ x) * g'\ (Re\ x)) +$
$Eps1\ x * Eps2\ x * (f''\ (Re\ x) * g\ (Re\ x) + f'\ (Re\ x) * g'\ (Re\ x) + f'\ (Re\ x) * g'\ (Re\ x) +$
$f\ (Re\ x) * g''\ (Re\ x))$
using *hyperdual-ext.eps12-g*$[of\ f\ f'\ f''\ S\ a\ fE\ x]$
hyperdual-ext.eps2-g$[of\ f\ f'\ f''\ S\ a\ fE\ x]$
hyperdual-ext.eps1-g$[of\ f\ f'\ f''\ S\ a\ fE\ x]$
hyperdual-ext.re-g$[of\ f\ f'\ f''\ S\ a\ fE\ x]$
hyperdual-ext.eps12-g$[of\ g\ g'\ g''\ S\ a\ gE\ x]$
hyperdual-ext.eps2-g$[of\ g\ g'\ g''\ S\ a\ gE\ x]$
hyperdual-ext.eps1-g$[of\ g\ g'\ g''\ S\ a\ gE\ x]$
hyperdual-ext.re-g$[of\ g\ g'\ g''\ S\ a\ gE\ x]$
assms
by (*simp add*: *algebra-simps*)
qed

While this proof proceeds in the same way as the addition proof, it is almost double the length. This gives an indication of how the complexity of the proof quickly increases once more complicated expressions are involved. This can be felt even more keenly when the considered function is a composition of multiple functions, which can lead to very large second derivative expressions. We plan to investigate ways to alleviate this problem in the future (see Section 5.1.1.1).

4.4.3.4 Linear Function

Next we will look at the proof that a hyperdual linear function is an extension of a real linear function:

lemma *hyperdual-ext-linear*:
hyperdual-ext $(\lambda x.\ k *_R x + a)\ (\lambda x.\ k)\ (\lambda x.\ 0)\ UNIV\ x\ (\lambda x.\ k *_R x + of\text{-}real\ a)$
proof –

This proof is a simple example of how the *compose* and *subset* theorems from the hyperdual extension locale can be used to construct hyperdual extensions by composition. We will therefore not be directly proving the locale's assumption, but deriving the claim itself. We consider a linear function as a composition of identity, scalar multiplication, addition and constant.

First, we show the extension of the addition of a constant with the filter set being the range of multiplication by a scalar and the filter variable being the overall filter variable multiplied by that scalar:

have *hyperdual-ext* ($\lambda x.\ x + a$) ($\lambda x.\ 1$) ($\lambda x.\ 0$) (*range* (($*_R$) *k*)) (*k* $*_R$ *x*) ($\lambda x.\ x +$ *of-real a*)
using *hyperdual-ext-add-const*[*of a k* $*_R$ *x*]
hyperdual-ext.subset[*of* $\lambda x.\ x + a$ $\lambda x.\ 1$ $\lambda x.\ 0$ *UNIV k* $*_R$ *x* $\lambda x.\ x +$ *of-real a range* (($*_R$) *k*)]
subset-UNIV[*of range* (($*$) *k*)]
by *simp*

This peculiar form of the statement is required to work with the composition theorem. It is derived from the addition of constant extension (proved separately) by the subset theorem, keeping in mind that any set is a subset of the universal set.

Then we use the composition theorem to combine the scalar multiplication extension with the one we just derived, proving the claim:

then show *?thesis*
using *hyperdual-ext-scaleR*[*of k x*]
hyperdual-ext.compose[*of* ($*$) *k* $\lambda x.\ k$ $\lambda x.\ 0$ *UNIV x* ($*_R$) *k* $\lambda x.\ x + a$ $\lambda x.\ 1$ $\lambda x.\ 0$ $\lambda x.\ x +$ *of-real a*]
by *simp*
qed

This shows that using composition to prove extensions can simplify the proofs involved. The one complication is that the domain of the outer function needs to match the codomain of the inner function, which introduces the condition on the filter set and variable. This is where the subset theorem is vital, it allows us to adjust the filter set in a consistent way in order to match the requirement of the composition theorem.

4.4.3.5 Sine

Next we look at the hyperdual extension of sine. We first need to define a new function to serve as the extension. To do this, we simply follow the definition (4.3) of the hyperdual extension and use the known derivatives of sine:

primcorec *hyperdual-sin* :: *hyperdual* $\Rightarrow$ *hyperdual*
where
Re (*hyperdual-sin x*) = *sin* (*Re x*)
| *Eps1* (*hyperdual-sin x*) = (*Eps1 x*) $*$ (*cos* (*Re x*))
| *Eps2* (*hyperdual-sin x*) = (*Eps2 x*) $*$ (*cos* (*Re x*))
| *Eps12* (*hyperdual-sin x*) = (*Eps12 x*) $*$ (*cos* (*Re x*)) + (*Eps1 x*) $*$ (*Eps2 x*) $*$ ($-$ *sin* (*Re x*))

Then we need to prove that this function we defined is indeed the extension of real sine:

lemma *hyperdual-ext-sin*:
hyperdual-ext sin cos ($-$ *sin*) *UNIV x hyperdual-sin*
proof

Because we directly defined this function, we cannot take advantage of composition and instead need to use the direct approach starting with the derivative relation. However, this is made simpler by the required facts once again being already-proven theorems in Isabelle and thus the proof can be handled by automatic methods:

```
  show (sin has-field-derivative cos x) (at x)
    by simp
  show (cos has-field-derivative (− sin) x) (at x)
    by simp
```

Finally, the proof of the extension function being equal to what is assumed by the locale is made simple by the fact that we defined it to be exactly that:

```
  fix x :: hyperdual
  show Re (hyperdual-sin x) = sin (Re x)
    by simp
  show Eps1 (hyperdual-sin x) = Eps1 x ∗ cos (Re x)
    by simp
  show Eps2 (hyperdual-sin x) = Eps2 x ∗ cos (Re x)
    by simp
  show Eps12 (hyperdual-sin x) = Eps12 x ∗ cos (Re x) + Eps1 x ∗ Eps2 x ∗ (− sin) (Re x)
    by simp
qed
```

This proof shows that if care is taken when defining a new function to serve as the hyperdual extension of some real function, the proof can be quite simple. It is even simpler when the relevant derivative statements are already proven theorems. However, this does not necessarily mean that the function we defined behaves as hyperdual analogue of sine, only that it fits as an extension of the real one.

4.4.3.6 Finite Polynomial

Last we look at the proof that finite hyperdual polynomials are extensions of real finite polynomials. This proof uses composition of scalar multiplication, natural power and addition. Moreover, we prove it by induction on the number of terms. However, because the expressions involved initially included subtraction of up to two from natural numbers, we had to split this into three cases: zero terms, one term, two or more terms. This is because subtraction of two is not well defined for natural numbers below two.

First, we prove the case with zero terms:

```
lemma hyperdual-ext-polyn0:
  fixes coef :: nat ⇒ real
  shows hyperdual-ext (λx. (∑i<0. coef i ∗ xˆi)) (λx. (∑j<(0−1). coef (j+1) ∗ (j+1) ∗ xˆj))
        (λx. (∑k<(0−2). coef (k+2) ∗ (k+2) ∗ (k+1) ∗ xˆk)) UNIV a (λx. (∑i<0. coef i ∗R
xˆi))
proof −
```

For this proof, we first show that the extension function is equal to the constant zero function:

```
  have (λx::hyperdual. (∑i<0. coef i ∗R xˆi)) = (λx. of-real 0)
  proof
    fix x::hyperdual
    show (∑i<0. coef i ∗R x ˆ i) = of-real 0
      by simp
  qed
```

Then we can use the extension of a constant zero function to prove the claim:

```
 then show ?thesis
   using hyperdual-ext-const[of 0 a]
   by simp
qed
```

Next, we prove the case with one term:

```
lemma hyperdual-ext-polyn1:
 fixes coef :: nat ⇒ real
 shows hyperdual-ext (λx. (∑i<1. coef i ∗ xˆi)) (λx. (∑j<(1−1). coef (j+1) ∗ (j+1) ∗ xˆj))
       (λx. (∑k<(1−2). coef (k+2) ∗ (k+2) ∗ (k+1) ∗ xˆk)) UNIV a (λx. (∑i<1. coef i ∗R
xˆi))
proof −
```

For this proof, we first show that the extension function is equal to the constant function of the first coefficient:

```
 have (λx::hyperdual. (∑i<1. coef i ∗R xˆi)) = (λx. of-real (coef 0))
 proof
   fix x::hyperdual
   show (∑i<1. coef i ∗R x ˆ i) = of-real (coef 0)
     by simp
 qed
```

Then we can use the extension of a constant function of the first coefficient to prove the claim:

```
 then show ?thesis
   using hyperdual-ext-const[of coef 0 a]
   by simp
qed
```

Finally, we prove the remaining cases with two or more terms. We induct on the number of terms of the second derivative, thus avoiding subtraction of natural numbers:

```
lemma hyperdual-ext-polyn-ge2:
 fixes coef :: nat ⇒ real
   and n :: nat
 shows hyperdual-ext (λx. (∑i<(n+2). coef i ∗ xˆi)) (λx. (∑j<(n+1). coef (j+1) ∗ (j+1) ∗
xˆj))
       (λx. (∑k<n. coef (k+2) ∗ (k+2) ∗ (k+1) ∗ xˆk)) UNIV a (λx. (∑i<(n+2). coef i ∗R
xˆi))
proof (induct n)
```

For the base case, we first show that the real function is equal to a linear function:

```
 case 0
 have (λx::real. (∑i<0+2. coef i ∗ xˆi)) = (λx. coef 1 ∗ x + coef 0)
 proof
   fix x::real
   have (∑i<1+1. coef i ∗ x ˆ i) = coef 1 ∗ x + coef 0
     by simp
```

  then show $(\sum i{<}0{+}2.\ coef\ i * x\ \hat{}\ i) = coef\ 1 * x + coef\ 0$
    using *nat-1-add-1*
    by *simp*
  qed

We then show the same for the extension function:

  moreover have $(\lambda x{::}hyperdual.\ (\sum i{<}0{+}2.\ coef\ i *_R x\hat{}i)) = (\lambda x.\ coef\ 1 *_R x + of\text{-}real\ (coef\ 0))$
  proof
    fix $x{::}hyperdual$
    have $(\sum i{<}1{+}1.\ coef\ i *_R x\ \hat{}\ i) = coef\ 1 *_R x + of\text{-}real\ (coef\ 0)$
      by *simp*
    then show $(\sum i{<}0{+}2.\ coef\ i *_R x\ \hat{}\ i) = coef\ 1 *_R x + of\text{-}real\ (coef\ 0)$
      using *nat-1-add-1*
      by *simp*
  qed

Then we can use automatic methods to prove the base case using the extension of a linear function:

  ultimately show *?case*
    using *hyperdual-ext-linear*[*of coef 1 coef 0 a*]
    by *simp*
next

For the inductive step, we assume an extension for a polynomial with $n+2$ terms. We then want to add to it the $coef_{n+2} \cdot x^{n+2}$ term using the extension of addition. First, we use the composition theorem, subset theorem, and the extensions of scalar multiplication and natural power to derive the extension of the term we intend to add:

  case *hyp*: (*Suc n*)

  have $hyperdual\text{-}ext\ (\lambda x.\ coef\ (n+2) *_R x\ \hat{}\ (n+2))\ (\lambda x.\ coef\ (n+2) * ((n+2) * x\ \hat{}\ (n+1)))$
      $(\lambda x.\ coef\ (n+2) * ((n+2) * (n+1) * x\ \hat{}\ n) + 0 * ((n+2) * x\ \hat{}\ (n+1)) * ((n+2) * x\ \hat{}\ (n+1)))$
      $UNIV\ a\ (\lambda x.\ coef\ (n+2) *_R x\ \hat{}\ (n+2))$
    using *hyperdual-ext-power-ge2*[*of n a*]
      $hyperdual\text{-}ext.compose[of\ \lambda x.\ x\ \hat{}\ (n+2)\ \lambda x.\ (n+2) * x\ \hat{}\ (n+1)\ \lambda x.\ (n+2) * (n+1) * x\ \hat{}\ n\ UNIV\ a\ \lambda x.\ x\ \hat{}\ (n+2)$
      $(*_R)\ (coef\ (n+2))\ \lambda x.\ coef\ (n+2)\ \lambda x.\ 0\ (*_R)\ (coef\ (n+2))]$
      $hyperdual\text{-}ext\text{-}scaleR[of\ coef\ (n+2)\ a\ \hat{}\ (n+2)]$
      hyperdual-ext.subset
    by *blast*

Then we use the set of algebraic simplifications available in Isabelle to simplify the involved expressions:

  then have $hyperdual\text{-}ext\ (\lambda x.\ coef\ (n+2) *_R x\ \hat{}\ (n+2))\ (\lambda x.\ coef\ (n+2) * (n+2) * x\ \hat{}\ (n+1))\ (\lambda x.\ coef\ (n+2) * (n+2) * (n+1) * x\ \hat{}\ n)\ UNIV\ a\ (\lambda x.\ coef\ (n+2) *_R x\ \hat{}\ (n+2))$
    by (*simp add*: *algebra-simps*)

Then we show that adding this intended term to the real function is equal to the target polynomial:

moreover have $(\lambda x.\ \sum i{<}Suc\ n + 2.\ coef\ i * x\ \hat{}\ i) =$
$(\lambda x.\ (\sum i{<}n + 2.\ coef\ i * x\ \hat{}\ i) + coef\ (n{+}2) *_R x\ \hat{}\ (n{+}2))$
by *simp*

Next we show that adding the first derivative of the intended term to the first derivative function is equal to the first derivative of the target polynomial:

moreover have $(\lambda x.\ \sum j{<}Suc\ n + 1.\ coef\ (j + 1) * real\ (j + 1) * x\ \hat{}j) =$
$(\lambda x.\ (\sum j{<}n + 1.\ coef\ (j + 1) * real\ (j + 1) * x\ \hat{}j) + coef\ (n + 1 + 1) * real\ (n + 1 + 1) * x\ \hat{}\ (n{+}1))$
by *simp*

Next we show that adding the second derivative of the intended term to the second derivative function is equal to the second derivative of the target polynomial:

moreover have $(\lambda x.\ \sum k{<}Suc\ n.\ coef\ (k + 2) * real\ (k + 2) * real\ (k + 1) * x\ \hat{}\ k) =$
$(\lambda x.\ (\sum k{<}n.\ coef\ (k + 2) * real\ (k + 2) * real\ (k + 1) * x\ \hat{}\ k) + coef\ (n + 2) * real\ (n + 2) * real\ (n + 1) * x\ \hat{}\ n)$
by *simp*

Next we show that adding the intended hyperdual term to the extension function is equal to the target hyperdual polynomial:

moreover have $(\lambda x.\ \sum i{<}Suc\ n + 2.\ coef\ i *_R x\ \hat{}\ i) =$
$(\lambda x.\ (\sum i{<}n + 2.\ coef\ i *_R x\ \hat{}\ i) + coef\ (n{+}2) *_R x\ \hat{}\ (n{+}2))$
by *simp*

Finally, we combine all the previous steps with the relevant instantiation of the addition extension and the inductive hypothesis, which allows the automatic simplifier with the addition of the set of algebraic simplifications available in Isabelle to prove the inductive step:

ultimately show *?case*
using *hyp*
hyperdual-ext-add$[of\ \lambda x.\ \sum i{<}n + 2.\ coef\ i * x\ \hat{}\ i\ \lambda x.\ \sum j{<}n + 1.\ coef\ (j + 1) * (j + 1) * x\ \hat{}j$
$\lambda x.\ \sum k{<}n.\ coef\ (k + 2) * (k + 2) * (k + 1) * x\ \hat{}k\ UNIV\ a\ \lambda x.\ \sum i{<}n + 2.\ coef\ i *_R x\ \hat{}\ i$
$\lambda x.\ coef\ (n + 2) *_R x\ \hat{}\ (n + 2)\ \lambda x.\ coef\ (n + 2) * (n + 2) * x\ \hat{}\ (n + 1)$
$\lambda x.\ coef\ (n + 2) * (n + 2) * (n + 1) * x\ \hat{}\ n\ \lambda x.\ coef\ (n + 2) *_R x\ \hat{}\ (n + 2)]$
by (*simp add*: *algebra-simps*)
qed

This proof combines the composition and subset theorems with a number of proven extensions within an induction proof. We consider this a great demonstration of proving an extension of a composite function without having to directly work with its derivatives. On the other hand, it also demonstrates well how the expressions involved can quickly get complicated (see Section 5.1.1.1 on our plans regarding this).

4.4.3.7 Analytic Test Function

To evaluate expressivity and usability of the formal system we set up, we have started on the mechanization of the analytic test function used by Fike and Alonso [10], given as follows:

$$f(x) = \frac{e^x}{\sqrt{sin(x)^3 + cos(x)^3}} \tag{4.5}$$

To this end, we define a hyperdual function using the same composition of hyperdual extensions of the respective functions:

definition *fa2011-test* :: *hyperdual* ⇒ *hyperdual*
 where *fa2011-test x* = (*hyperdual-exp x*) / (*hyperdual-sqrt* ((*hyperdual-sin x*)ˆ*3* + (*hyperdual-cos x*)ˆ*3*))

Our hope is that this function is a hyperdual extension of the real function (4.5). Then evaluating this hyperdual function would in essence perform automatic differentiation of that real function. We would also like it to replicate exactly the results mentioned by Fike and Alonso [10] in that same section. There are two possible approaches to this:

- Prove this function we define to be a hyperdual extension of the real function (4.5) with the derivatives as specified by Fike and Alonso
- Evaluate this function and compare the results to those given by Fike and Alonso

We have started work on both these approaches, but neither is complete yet. See Section 5.1.1.5 where we go over future work.

The second approach may seem easy at first glance because it does not involve a proof, but there is a hidden problem. All of sine, cosine, exponential and square root are defined in Isabelle in uncomputable terms. In particular, sine, cosine and exponential are defined as power series i.e. "infinite sums", and square root is defined using the choice function. This means that Isabelle is not able to directly evaluate the function we defined.

We plan to define hyperdual extensions of computable *approximations* of these uncomputable functions. By composing these approximation extensions in the same manner as the original analytic test function extension, we should get an extension that is computable and approximates the results.

Not even the first approach is without problems. While we can use the composition theorem and previous extensions to prove that the function we define is a hyperdual extension of the real function (4.5), we are not yet able to do so with the same derivatives that Fike and Alonso specify. This problem lies in establishing equality of two very large expressions. It is also compounded by these expressions involving square root, which is not well defined on all numbers and thus we require further lemmas to perform the required simplifications. We plan to keep working on this problem, more so because it could potentially lead to a set of further useful theorems for hyperdual extensions.

4.5 Summary

In this chapter we introduced and motivated the hyperdual extensions of real functions. We derived definition for these extensions based on the properties we want them to have. We then described how we mechanized this concept in Isabelle.

There we described details about the current mechanization based on the field derivative, as well as outlined the difficulties with implementing a mechanization based on the general derivative. We then briefly summarized concrete cases of hyperdual extensions we have proven, followed by detailed description of a chosen few of those proofs. We finished by describing the analytic test function used by Fike and Alonso [10] and the challenges that need to be tackled when it comes to mechanizing it.

Next we summarize our results and lay out future work.

Chapter 5

Conclusion

We mechanized the following definitions due to Fike and Alonso [10]:

- Hyperdual number
- Addition
- Multiplication
- Multiplicative inverse

On top of this we proved hyperduals to be an instance of the following algebraic structures, defining further operations as required:

- Commutative group under addition
- Commutative ring under multiplication
- Algebra over the reals
- Real-normed vector space

Furthermore we proved important facts about:

- Hyperdual zero divisors
- Multiplication cancellation
- Hyperdual division
- Limits and derivatives of hyperdual-valued functions

We also defined a notion of a hyperdual extension of a real function, exploring some of its properties and providing examples of extensions of basic functions.

5.1 Future Work

This was the first year of our two-year project. For the next year, we want to focus on both improving the theory already mentioned as well as finding further potentially useful properties. There are also some things outside the scope of this project that could nevertheless be interesting avenues of further research.

5.1.1 Improving the Hyperdual Extension

Foremost is the improvement of the current theory around the hyperdual extension of a real function. This is both improving the field version of the locale as well as working on the general version. It also includes forming a better understanding of how it arises.

5.1.1.1 Choice Function for Derivatives

One big drawback of the extension locales as they are currently implemented is the need to specify the first and second derivatives. The expressions involved in these tend to quickly grow in complexity as functions are composed.

Currently, we find the derivatives to use in a claim by substituting the relevant terms into the theorem we intend to use and then copying the derivatives out of its conclusion. Given this process, it should be possible to automate this step. Isabelle allows choosing some arbitrary derivative (given that the function is differentiable) using the choice function. If successful, this would lead to much simpler claim statements, and cleaner and faster proofs.

5.1.1.2 Stronger Uniqueness Theorem

The current uniqueness theorem mentioned in Section 4.4.1 assumes the two extensions are proven with same derivatives. It would be much more useful if we could prove a stronger version of this theorem that only assumes the real function, and the filter variable and set are the same. This would for example allow us to skip the long proofs of equality of different expressions for the same function often seen in proofs using composition of multiple extensions (see Section 4.4.3). We plan to investigate this further, as successfully proving this stronger variant would considerably simplify future proofs.

5.1.1.3 General Version Implementation

Currently we are still in process of implementing the more general *has-derivative* version of the hyperdual extension locale. We will now outline our plan for this implementation.

The first step in building this locale is building a second derivative locale, similar to the one Immler and Hölzl use [18]. This includes building familiarity with the bounded linear function representation, and working through our examples to see how best to implement the locale to fit our needs.

The next step then is to build a hyperdual extension locale on top of that second derivative locale similarly to how it is done for the field version (see Section 4.4.1). We will do this alongside proving extensions already proven for the field version using this more general locale. This should ensure we are converging on the most useable version of the locale as well as catching any inadequacies early.

5.1.1.4 Connection to Real Function

As mentioned in Chapter 4, there seems to be some underlying relationship between real and hyperdual versions of various operations. This points to the hyperdual extension being a property of the number system. We would like to further investigate this property and any underlying basis it might have.

One thing we noticed is that it may be possible to define most of the properties of hyperdual numbers as extensions of the real properties. Mechanizing an independent theory of hyperdual numbers based entirely on hyperdual extensions may give us further insight into this. It may even be possible to define other interesting number systems using variations of the hyperdual extension.

5.1.1.5 Analytic Test Function

As discussed in Section 4.4.3.7, our mechanization of the analytic test function used by Fike and Alonso [10] is incomplete. This function serves as a good test of the usability of our theory and therefore we intend to work further on completing this mechanization.

5.1.2 General Second Derivative

As part of our above mentioned plans for the general hyperdual extension locale we will be creating tools to work with general second order Fréchet derivatives. These tools are not specific to this project and thus they could be useful for other projects in the Isabelle community. It should be possible to extract a small library of useful locales and theorems from our work for general use.

5.1.3 Hyperdual Taylor's Expansion

When deriving the hyperdual extension in Section 4.3, we assume a certain form of the Taylor's expansion for hyperduals. We have not proven that this form is valid which represents a potential gap in the derivation. We plan to investigate and potentially prove the form of the hyperdual Taylor's expansion.

5.1.4 Verification of Algorithms

The hyperdual extension is intended to be used in computing derivatives of functions specified by programmes. It would be good to include formal verification of the extension's use in some simpler algorithm. This would both serve as an example for future efforts, and an opportunity to test the usability of the theory for such endeavours.

Related to this is also the representation of hyperduals. Currently the proofs assume the components are real numbers. It would be interesting to see how much of the behaviour persists when they are instead represented with floating point representations that computers use.

5.1.5 Further Properties

Currently the only use case we explore is automatic differentiation. Investigating alternative use cases and developing the theory around the properties they use could lead to a more well rounded theory. It might also uncover some underlying connections about why the approaches work or are better than alternatives.

Another possible source of further properties is Kantor and Solodovnikov's description of hypercomplex numbers [19]. As hyperduals are an instance of these, the properties should hold for them. It should be interesting to mechanize these properties for the particular case of hyperduals and see if they have any immediate use.

5.1.6 Code Extraction

An interesting possibility is posed by the ability of Isabelle/HOL to generate code from definitions [13]. Most of the corecursive definitions of properties of hyperdual numbers should require little adjustment for code generation. However we also define a number of uncomputable functions (e.g. hyperdual extension of sine). We would have to use appropriate approximations instead of these to be able to generate working code.

Once generated, this code could then be compared to existing implementations of hyperduals in various languages. This comparison could be enlightening, potentially even suggesting some enhancements to the implementations in question.

5.1.7 Higher-Order Hyperduals

As eluded to in the introduction, we only concern ourselves with second-order hyperdual numbers. If a similar theory was developed for arbitrary higher-order hyperdual numbers, it could uncover some further uses for the numbers.

One possible approach could be through hypercomplex numbers. While investigating the hypercomplex point of view on second-order hyperduals, we noticed that arbitrary order hyperduals seem to form a hypercomplex system with basis the set resulting from mapping a product over the power set of the set of infinitesimals. That is, products of all combinations of infinitesimals (including the empty product – multiplicative unit). The multiplication table is then constructed so that each infinitesimal is nilpotent (i.e. every term involving a square of some infinitesimal is 0).

5.2 Final Remarks

We have mechanized properties of hyperdual numbers from addition up to derivatives. Then we mechanized a notion of a hyperdual extension, its properties and some cases. Finally we outlined several future extensions to this project.

We believe that this work sets the scene for a comprehensive framework in which to investigate hyperdual numbers and their applications.

Bibliography

[1] John Baez. The Cayley–Dickson construction, in The Octonions. *Bull. Amer. Math. Soc. 39*, 2002.

[2] Henk Barendregt and Herman Geuvers. Chapter 18 - proof-assistants using dependent type systems. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, Handbook of Automated Reasoning, pages 1149 – 1238. North-Holland, Amsterdam, 2001.

[3] Michael Bartholomew-Biggs, Steven Brown, Bruce Christianson, and Laurence Dixon. Automatic differentiation of algorithms. *Journal of Computational and Applied Mathematics*, 124(1):171 – 190, 2000. Numerical Analysis 2000. Vol. IV: Optimization and Nonlinear Equations.

[4] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research*, 18(153):1–43, 2018.

[5] Julian Biendarra, Jasmin Blanchette, Martin Desharnais, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. Defining (co)datatypes and primitively (co)recursive functions in isabelle/hol. https://isabelle.in.tum.de/dist/Isabelle2020/doc/datatypes.pdf, 2020. Accessed: 2020-04-17.

[6] Jasmin Blanchette and Lawrence Paulson. Hammering away: A user's guide to sledgehammer for isabelle/hol. https://isabelle.in.tum.de/dist/Isabelle2020/doc/sledgehammer.pdf, 2020. Accessed: 2020-04-17.

[7] Alan Bundy. Chapter 13 - the automation of proof by mathematical induction. In *Handbook of Automated Reasoning*, pages 845–911. Elsevier B.V, 2001.

[8] Clifford. Preliminary sketch of biquaternions. *Proceedings of the London Mathematical Society*, s1-4(1):381–395, 1871.

[9] Rodney Coleman. *Differentiation*, pages 35–60. Springer New York, New York, NY, 2012.

[10] J. A. Fike and J. J. Alonso. The development of hyper-dual numbers for exact second-derivative calculations. In *AIAA paper 2011-886, 49th AIAA Aerospace Sciences Meeting*, 2011.

[11] Andrea Gabrielli and Marco Maggesi. Formalizing basic quaternionic analysis. In *International Conference on Interactive Theorem Proving*, pages 225–240. Springer, 2017.

[12] Florian Haftmann. Haskell-style type classes with Isabelle/Isar. https://isabelle.in.tum.de/dist/Isabelle2020/doc/classes.pdf, 2020. Accessed: 2020-04-17.

[13] Florian Haftmann and Lukas Bulwahn. Code generation from Isabelle/HOL theories. https://isabelle.in.tum.de/dist/Isabelle2020/doc/codegen.pdf, 2020. Accessed: 2020-04-21.

[14] John Harrison. A HOL theory of euclidean space. In *In Hurd and Melham [7*, pages 114–129. Springer, 2005.

[15] John Harrison. Formalizing basic complex analysis. In R. Matuszewski and A. Zalewska, 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, 2007.

[16] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.

[17] Johannes Hölzl, Fabian Immler, and Brian Huffman. Type classes and filters for mathematical analysis in Isabelle/HOL. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, pages 279–294, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[18] Fabian Immler and Johannes Hölzl. Ordinary differential equations. *Archive of Formal Proofs*, April 2012. http://isa-afp.org/entries/Ordinary_Differential_Equations.html, Formal proof development.

[19] I. L. Kantor and A. S. Solodovnikov. *Hypercomplex numbers: an elementary introduction to algebras, Translated by A. Shenitzer*. Springer, 1989.

[20] Ladislav Kavan, Steven Collins, Jiří Žára, and Carol O'Sullivan. Geometric skinning with approximate dual quaternion blending. *ACM Trans. Graph.*, 27(4), November 2008.

[21] Anthony W. Knapp. *Lebesgue Measure and Abstract Measure Theory*, pages 231–295. Birkhäuser Boston, Boston, MA, 2005.

[22] Jack B Kuipers. *Quaternions and Rotation Sequences: A Primer with Applications to Orbits, Aerospace and Virtual Reality*. Princeton University Press, 1999.

[23] Randall J LeVeque. *Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems*, volume 98. Siam, 2007.

[24] Joaquim R. R. A. Martins, Peter Sturdza, and Juan J. Alonso. The complex-step derivative approximation. *ACM Trans. Math. Softw.*, 29(3):245–262, September 2003.

[25] Genki Matsuda, Shizuo Kaji, and Hiroyuki Ochiai. *Anti-commutative Dual Complex Numbers and 2D Rigid Transformation*, pages 131–138. Springer Japan, Tokyo, 2014.

[26] J.M. McCarthy. *An Introduction to Theoretical Kinematics*. MIT Press, 1990.

[27] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. *CoRR*, cs.LO/9301106, 1993.

[28] Lawrence C. Paulson. Quaternions. *Archive of Formal Proofs*, September 2018. http://isa-afp.org/entries/Quaternions.html, Formal proof development.

[29] L. R. Rabiner and B. Gold. *Theory and application of digital signal processing*. 1975.

[30] J. Revels, M. Lubin, and T. Papamarkou. Forward-mode automatic differentiation in Julia. *arXiv:1607.07892 [cs.MS]*, 2016.

[31] Alan J.A. Robinson and Andrei Voronkov. *Handbook of Automated Reasoning*, volume 1 and 2. Elsevier B.V, 2001.

[32] Stephan Schulz. E - a brainiac theorem prover. *AI Commun.*, 15(2,3):111–126, August 2002.

[33] Mícheál Ó Searcóid. *Completeness*, pages 245–268. Springer London, London, 2002.

[34] Mícheál Ó Searcóid. *Convergence*, pages 215–230. Springer London, London, 2002.

[35] Mícheál Ó Searcóid. *Geometric Structure*, pages 133–158. Springer London, London, 2002.

[36] Ken Shoemake. Animating rotation with quaternion curves. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH ’85, page 245–254, New York, NY, USA, 1985. Association for Computing Machinery.

[37] John Vince. *Complex Numbers*, pages 5–16. Springer London, London, 2011.

[38] John Vince. *Quaternions*, pages 59–65. Springer London, London, 2011.

[39] John Von Neumann. *Mathematical foundations of quantum mechanics: New edition*. Princeton university press, 2018.

[40] M Wenzel and L Paulson. Isabelle/Isar. *Seventeen Provers Of The World*, 3600:41–49, 2006.

Chapter 6

Appendix - Proofs

In this appendix we give full versions of some proofs from our theory which are mentioned in this report.

6.1 *ab-group-add* Instantiation

In the following, we define the required operations and unit to prove that hyperduals form an instance of the commutative group under addition (see Section 3.2) type class *ab-group-add*:

instantiation *hyperdual* :: *ab-group-add*
begin

primcorec *zero-hyperdual*
  where
   $Re\ 0 = 0$
  $|\ Eps1\ 0 = 0$
  $|\ Eps2\ 0 = 0$
  $|\ Eps12\ 0 = 0$

primcorec *plus-hyperdual*
  where
   $Re\ (x + y) = Re\ x + Re\ y$
  $|\ Eps1\ (x + y) = Eps1\ x + Eps1\ y$
  $|\ Eps2\ (x + y) = Eps2\ x + Eps2\ y$
  $|\ Eps12\ (x + y) = Eps12\ x + Eps12\ y$

primcorec *uminus-hyperdual*
  where
   $Re\ (-x) = -\ Re\ x$
  $|\ Eps1\ (-x) = -\ Eps1\ x$
  $|\ Eps2\ (-x) = -\ Eps2\ x$
  $|\ Eps12\ (-x) = -\ Eps12\ x$

primcorec *minus-hyperdual*

where
Re $(x - y) =$ *Re x* $-$ *Re y*
| *Eps1* $(x - y) =$ *Eps1 x* $-$ *Eps1 y*
| *Eps2* $(x - y) =$ *Eps2 x* $-$ *Eps2 y*
| *Eps12* $(x - y) =$ *Eps12 x* $-$ *Eps12 y*

instance
by *standard simp-all*
end

6.2 *comm-ring-1* Instantiation

In the following, we define the required operations and unit to prove that hyperduals form an instance of the commutative ring under multiplication (see Section 3.3) type class *comm-ring-1*:

instantiation *hyperdual* :: *comm-ring-1*
begin

primcorec *one-hyperdual*
where
Re 1 $=$ *1*
| *Eps1 1* $=$ *0*
| *Eps2 1* $=$ *0*
| *Eps12 1* $=$ *0*

primcorec *times-hyperdual*
where
Re $(x * y) =$ *Re x* $*$ *Re y*
| *Eps1* $(x * y) = ($*Re x* $*$ *Eps1 y*$) + ($*Eps1 x* $*$ *Re y*$)$
| *Eps2* $(x * y) = ($*Re x* $*$ *Eps2 y*$) + ($*Eps2 x* $*$ *Re y*$)$
| *Eps12* $(x * y) = ($*Re x* $*$ *Eps12 y*$) + ($*Eps1 x* $*$ *Eps2 y*$) + ($*Eps2 x* $*$ *Eps1 y*$) + ($*Eps12 x* $*$ *Re y*$)$

instance
by *standard* (*simp-all add*: *algebra-simps*)
end

6.3 Non-Trivial Zero Divisors Proof

First, we use automatic methods to establish the behaviour on the real component is as with real numbers:

lemma *divisors-re-zero*:
fixes *a b* :: *hyperdual*
assumes *Re* $(a * b) = 0$
shows *Re a* $= 0 \lor$ *Re b* $= 0$
using *assms* **by** *auto*

```
lemma mult-eq-0-iff:
 fixes a b :: hyperdual
 shows Re (a ∗ b) = 0 ⟷ Re a = 0 ∨ Re b = 0
by simp
```

We then use these theorems to prove the condition (3.4):

```
lemma divisiors-hyperdual-zero [simp]:
 fixes a b :: hyperdual
 shows a ∗ b = 0 ⟷ (a = 0 ∨ b = 0 ∨ (Re a = 0 ∧ Re b = 0 ∧ Eps1 a ∗ Eps2 b = − Eps2
a ∗ Eps1 b))
proof
 assume mult: a ∗ b = 0
 then have split: Re a = 0 ∨ Re b = 0
  by (simp add: divisors-re-zero)
 show a = 0 ∨ b = 0 ∨ Re a = 0 ∧ Re b = 0 ∧ Eps1 a ∗ Eps2 b = − Eps2 a ∗ Eps1 b
 proof (cases Re a = 0)
  case aT: True
  then show ?thesis
  proof (cases Re b = 0)
   case bT: True
   then have Eps12 (a ∗ b) = Eps1 a ∗ Eps2 b + Eps2 a ∗ Eps1 b
    by (simp add: aT)
   then show ?thesis
    by (simp add: aT bT mult)
  next
   case bF: False
   then have e1: Eps1 a = 0
   proof −
    have Eps1 (a ∗ b) = Eps1 a ∗ Re b
     by (simp add: aT)
    then show ?thesis
     by (simp add: bF mult)
   qed
   moreover have e2: Eps2 a = 0
   proof −
    have Eps2 (a ∗ b) = Eps2 a ∗ Re b
     by (simp add: aT)
    then show ?thesis
     by (simp add: bF mult)
   qed
   moreover have Eps12 a = 0
   proof −
    have Eps12 (a ∗ b) = Eps1 a ∗ Eps2 b + Eps2 a ∗ Eps1 b
     by (simp add: e1 e2 mult)
    then show ?thesis
     by (simp add: aT bF)
   qed
   ultimately show ?thesis
```

```
      by (simp add: aT)
    qed
  next   case aF: False
    then show ?thesis
    proof (cases Re b = 0)
      case bT: True
      then have e1: Eps1 b = 0
      proof -
        have Eps1 (a * b) = Re a * Eps1 b
          by (simp add: bT)
        then show ?thesis
          by (simp add: aF mult)
      qed
      moreover have e2: Eps2 b = 0
      proof -
        have Eps2 (a * b) = Re a * Eps2 b
          by (simp add: bT)
        then show ?thesis
          by (simp add: aF mult)
      qed
      moreover have Eps12 b = 0
      proof -
        have Eps12 (a * b) = Eps1 a * Eps2 b + Eps2 a * Eps1 b
          by (simp add: e1 e2 mult)
        then show ?thesis
          by (simp add: bT aF)
      qed
      ultimately show ?thesis
        by (simp add: bT)
    next
      case bF: False
      then have False
        using split aF by blast
      then show ?thesis
        by simp
    qed
  qed
next
  show a = 0 ∨ b = 0 ∨ Re a = 0 ∧ Re b = 0 ∧ Eps1 a * Eps2 b = − Eps2 a * Eps1 b ⟹ a *
b = 0
    by auto
qed
```

6.4 Multiplication Cancellation Proof

First, we prove the multiplication cancellation rule (3.5) for one configuration:

```
lemma hyperdual-mult-cancel-right1 [simp]:
  fixes a b :: hyperdual
```

```
 shows a = b ∗ a ⟷ a = 0 ∨ b = 1 ∨ (Re a = 0 ∧ Re b = 1 ∧ Eps1 b ∗ Eps2 a = − Eps2 b
∗ Eps1 a)
proof
 assume mult: a = b ∗ a
 then have Re a = Re b ∗ Re a
  by (metis times-hyperdual.simps(1))
 then have split: Re a = 0 ∨ Re b = 1
  using mult-cancel-right1 by blast
 show a = 0 ∨ b = 1 ∨ (Re a = 0 ∧ Re b = 1 ∧ Eps1 b ∗ Eps2 a = − Eps2 b ∗ Eps1 a)
 proof (cases Re a = 0)
  case aT: True
  then show ?thesis
  proof (cases Re b = 1)
   case bT: True
   then show ?thesis
    by (smt aT mult mult-cancel-right1 mult-minus-left times-hyperdual.simps(4))
  next
   case bF: False
   then have e1: Eps1 a = 0
   by (metis aT add.left-neutral mult mult.commute mult-cancel-left1 times-hyperdual.simps(2))
   moreover have e2: Eps2 a = 0
   by (metis aT add.left-neutral bF mult mult.commute mult-cancel-left1 times-hyperdual.simps(3))
   moreover have Eps12 a = 0
   by (metis aT add.left-neutral bF e1 e2 mult mult.commute mult-cancel-left1 times-hyperdual.simps(4))
   ultimately show ?thesis
    by (simp add: aT)
  qed
 next
  case aF: False
  then show ?thesis
  proof (cases Re b = 1)
   case bT: True
   then have e1: Eps1 b = 0
   proof −
    have Eps1 a = Re b ∗ Eps1 a + Eps1 b ∗ Re a
     using mult by (metis times-hyperdual.simps(2))
    then have 0 = Eps1 b ∗ Re a
     by (simp add: bT)
    then show ?thesis
     using aF by auto
   qed
   moreover have e2: Eps2 b = 0
   proof −
    have Eps2 a = Re b ∗ Eps2 a + Eps2 b ∗ Re a
     using mult by (metis times-hyperdual.simps(3))
    then have 0 = Eps2 b ∗ Re a
     by (simp add: bT)
    then show ?thesis
     using aF by auto
```

```
      qed
      moreover have Eps12 b = 0
      proof –
        have Eps12 a = Re b ∗ Eps12 a + Eps1 b ∗ Eps2 a + Eps2 b ∗ Eps1 a + Eps12 b ∗ Re a
          by (metis mult times-hyperdual.simps(4))
        then show ?thesis
          using aF bT e1 e2 by auto
      qed
      ultimately show ?thesis
        by (simp add: bT)
    next
      case bF: False
      then have False
        using split aF by blast
      then show ?thesis
        by simp
    qed
  qed
next
  have a = 0 ⟹ a = b ∗ a
    by simp
  moreover have b = 1 ⟹ a = b ∗ a
    by simp
  moreover have Re a = 0 ∧ Re b = 1 ∧ Eps1 b ∗ Eps2 a = − Eps2 b ∗ Eps1 a ⟹ a = b ∗ a
    by simp
  ultimately show a = 0 ∨ b = 1 ∨ (Re a = 0 ∧ Re b = 1 ∧ Eps1 b ∗ Eps2 a = − Eps2 b ∗
Eps1 a) ⟹ a = b ∗ a
    by blast
qed
```

We then extend this to the remaining three configurations:

```
lemma hyperdual-mult-cancel-right2 [simp]:
  fixes a b :: hyperdual
  shows b ∗ a = a ⟷ a = 0 ∨ b = 1 ∨ (Re a = 0 ∧ Re b = 1 ∧ Eps1 b ∗ Eps2 a = − Eps2 b
∗ Eps1 a)
using hyperdual-mult-cancel-right1 by metis

lemma hyperdual-mult-cancel-left1 [simp]:
  fixes a b :: hyperdual
  shows a = a ∗ b ⟷ a = 0 ∨ b = 1 ∨ (Re a = 0 ∧ Re b = 1 ∧ Eps1 b ∗ Eps2 a = − Eps2 b
∗ Eps1 a)
by (metis hyperdual-mult-cancel-right1 mult.commute)

lemma hyperdual-mult-cancel-left2 [simp]:
  fixes a b :: hyperdual
  shows a ∗ b = a ⟷ a = 0 ∨ b = 1 ∨ (Re a = 0 ∧ Re b = 1 ∧ Eps1 b ∗ Eps2 a = − Eps2 b
∗ Eps1 a)
using hyperdual-mult-cancel-left1 by metis
```

6.5 *real-algebra-1* Instantiation

In the following we define the last operation required to prove that hyperdual numbers are an instance of the algebra over real numbers (see Section 3.4) type class *real-algebra-1*:

```
instantiation hyperdual :: real-algebra-1
begin

primcorec scaleR-hyperdual
  where
   Re (scaleR f x) = f ∗ Re x
  | Eps1 (scaleR f x) = f ∗ Eps1 x
  | Eps2 (scaleR f x) = f ∗ Eps2 x
  | Eps12 (scaleR f x) = f ∗ Eps12 x

instance
by standard (simp-all add: algebra-simps)
end
```

6.6 *inverse* Instantiation

In the following we define hyperdual multiplicative inverse and division (see Section 3.5) and prove hyperdual numbers to be an instance of the type class *inverse*:

```
instantiation hyperdual :: inverse
begin

primcorec inverse-hyperdual
  where
   Re (inverse a) = 1 / Re a
  | Eps1 (inverse a) = − Eps1 a / (Re a)^2
  | Eps2 (inverse a) = − Eps2 a / (Re a)^2
  | Eps12 (inverse a) = 2 ∗ (Eps1 a ∗ Eps2 a / (Re a)^3) − Eps12 a / (Re a)^2

primcorec divide-hyperdual
  where
   Re (divide a b) = Re a / Re b
  | Eps1 (divide a b) = (Eps1 a ∗ Re b − Re a ∗ Eps1 b) / ((Re b)^2)
  | Eps2 (divide a b) = (Eps2 a ∗ Re b − Re a ∗ Eps2 b) / ((Re b)^2)
  | Eps12 (divide a b) = (2 ∗ Re a ∗ Eps1 b ∗ Eps2 b −
      Re a ∗ Re b ∗ Eps12 b −
      Eps1 a ∗ Re b ∗ Eps2 b −
      Eps2 a ∗ Re b ∗ Eps1 b +
      Eps12 a ∗ ((Re b)^2)) / ((Re b)^3)
instance
by standard
end
```

6.7 *division-hyperdual* Interpretation

The hyperdual division locale is defined as follows:

```
locale division-hyperdual =
  assumes left-inverse [simp]:  Re a ≠ 0 ⟹ inverse a ∗ a = 1
  assumes right-inverse [simp]: Re a ≠ 0 ⟹ a ∗ inverse a = 1
  assumes divide-inverse: a / b = a ∗ inverse b
  assumes inverse-zero [simp]: Re a = 0 ⟹ Re (inverse a) = 0
```

We then prove hyperduals to satisfy this locale:

```
interpretation hyperdual: division-hyperdual
proof
  fix a b :: hyperdual
  show Re a ≠ 0 ⟹ inverse a ∗ a = 1
    by (simp add: inverse-hyperdual-def times-hyperdual-def algebra-simps power2-eq-square
power3-eq-cube)
  show Re a ≠ 0 ⟹ a ∗ inverse a = 1
    by (simp add: inverse-hyperdual-def times-hyperdual-def algebra-simps power2-eq-square
power3-eq-cube)
  show a / b = a ∗ inverse b
  proof (rule hyperdual-eqI, simp-all add: inverse-hyperdual-def times-hyperdual-def power2-eq-square
power3-eq-cube)
    show (Eps1 a ∗ Re b − Re a ∗ Eps1 b) / (Re b ∗ Re b) = Eps1 a / Re b − Re a ∗ Eps1 b /
(Re b ∗ Re b)
      by (simp add: diff-divide-distrib)
    show (Eps2 a ∗ Re b − Re a ∗ Eps2 b) / (Re b ∗ Re b) = Eps2 a / Re b − Re a ∗ Eps2 b /
(Re b ∗ Re b)
      by (simp add: diff-divide-distrib)
    have Re a ∗ (2 ∗ (Eps1 b ∗ Eps2 b) / (Re b ∗ Re b ∗ Re b) −
        Eps12 b / (Re b ∗ Re b)) −
        Eps1 a ∗ Eps2 b / (Re b ∗ Re b) −
        Eps2 a ∗ Eps1 b / (Re b ∗ Re b) +
        Eps12 a / Re b =
        Re a ∗ 2 ∗ (Eps1 b ∗ Eps2 b) / (Re b ∗ Re b ∗ Re b) −
        Re a ∗ Eps12 b / (Re b ∗ Re b) −
        Eps1 a ∗ Eps2 b / (Re b ∗ Re b) −
        Eps2 a ∗ Eps1 b / (Re b ∗ Re b) +
        Eps12 a / Re b
      by (simp add: right-diff-distrib′)
    moreover have ... = Re a ∗ 2 ∗ (Eps1 b ∗ Eps2 b) / (Re b ∗ Re b ∗ Re b) −
        Re a ∗ Eps12 b ∗ Re b / (Re b ∗ Re b ∗ Re b) −
        Eps1 a ∗ Eps2 b ∗ Re b / (Re b ∗ Re b ∗ Re b) −
        Eps2 a ∗ Eps1 b ∗ Re b / (Re b ∗ Re b ∗ Re b) +
        Eps12 a ∗ Re b ∗ Re b / (Re b ∗ Re b ∗ Re b)
      by simp
    moreover have ... = (Re a ∗ 2 ∗ (Eps1 b ∗ Eps2 b) − Re a ∗ Eps12 b ∗ Re b − Eps1 a ∗
Eps2 b ∗ Re b − Eps2 a ∗ Eps1 b ∗ Re b + Eps12 a ∗ Re b ∗ Re b) /
            (Re b ∗ Re b ∗ Re b)
      by (simp add: add-divide-distrib diff-divide-distrib)
```

ultimately show $(2 * Re\ a * Eps1\ b * Eps2\ b - Re\ a * Re\ b * Eps12\ b - Eps1\ a * Re\ b * Eps2\ b - Eps2\ a * Re\ b * Eps1\ b + Eps12\ a * (Re\ b * Re\ b))\ /$
$(Re\ b * Re\ b * Re\ b) =$
$Re\ a * (2 * (Eps1\ b * Eps2\ b)\ /\ (Re\ b * Re\ b * Re\ b) - Eps12\ b\ /\ (Re\ b * Re\ b))$
$-$
$Eps1\ a * Eps2\ b\ /\ (Re\ b * Re\ b) -$
$Eps2\ a * Eps1\ b\ /\ (Re\ b * Re\ b) +$
$Eps12\ a\ /\ Re\ b$
by *auto*
qed
show $Re\ a = 0 \Longrightarrow Re\ (inverse\ a) = 0$
by *simp*
qed

6.8 Vector Space Preliminaries

We first define the four basis elements:

definition *re* **where** $re = Hyperdual\ 1\ 0\ 0\ 0$
definition *e1* **where** $e1 = Hyperdual\ 0\ 1\ 0\ 0$
definition *e2* **where** $e2 = Hyperdual\ 0\ 0\ 1\ 0$
definition *e12* **where** $e12 = Hyperdual\ 0\ 0\ 0\ 1$

We then prove that any hyperdual number is a linear combination of these four elements:

lemma *hyperdual-linear-comb*:
fixes $x :: hyperdual$
shows $\exists a\text{-}1\ a\text{-}2\ a\text{-}3\ a\text{-}4 :: real\ .\ x = a\text{-}1 *_R re + a\text{-}2 *_R e1 + a\text{-}3 *_R e2 + a\text{-}4 *_R e12$
by (*simp add*: *e12-def e1-def e2-def re-def*)

Next, we prove that the hyperdual constructor is equivalent to using these four elements and that the projections give exactly the linear combination coefficients:

lemma *Hyperdual-eq*:
shows $Hyperdual\ a\text{-}1\ a\text{-}2\ a\text{-}3\ a\text{-}4 = a\text{-}1 *_R re + a\text{-}2 *_R e1 + a\text{-}3 *_R e2 + a\text{-}4 *_R e12$
by (*simp add*: *e12-def e1-def e2-def plus-hyperdual.code re-def*)

lemma *hyperdual-eq*:
fixes $x :: hyperdual$
shows $x = Re\ x *_R re + Eps1\ x *_R e1 + Eps2\ x *_R e2 + Eps12\ x *_R e12$
using *Hyperdual-eq hyperdual.collapse* **by** *auto*

Last we show that the non-real basis elements are nilpotent and non-zero:

lemma *e1-square*:
shows $e1 * e1 = 0$
by (*simp add*: *e1-def*)
lemma *e2-square*:
shows $e2 * e2 = 0$
by (*simp add*: *e2-def*)
lemma *e12-square*:

```
  shows e12 ∗ e12 = 0
by (simp add: e12-def)

lemma e1-nonzero:
  shows e1 ≠ 0
by (simp add: e1-def)
lemma e2-nonzero:
  shows e2 ≠ 0
by (simp add: e2-def)
lemma e12-nonzero:
  shows e12 ≠ 0
by (simp add: e12-def)
```

6.9 *real-normed-vector* Instantiation

In the following we define the operations required and prove that hyperduals are an instance of the real normed vector (see Section 3.6) type class *real-normed-vector*:

```
instantiation hyperdual :: real-normed-vector
begin

definition norm-hyperdual :: hyperdual ⇒ real
  where norm-hyperdual x = sqrt((Re x)^2 + (Eps1 x)^2 + (Eps2 x)^2 + (Eps12 x)^2)

definition sgn-hyperdual :: hyperdual ⇒ hyperdual
  where sgn-hyperdual x = x /R norm x

definition dist-hyperdual :: hyperdual ⇒ hyperdual ⇒ real
  where dist-hyperdual a b = norm(a − b)

definition uniformity-hyperdual :: (hyperdual × hyperdual) filter
  where uniformity-hyperdual = (INF e∈{0 <..}. principal {(x, y). dist x y < e})

definition open-hyperdual :: hyperdual set ⇒ bool
  where  open-hyperdual U ⟷ (∀x∈U. eventually (λ(x', y). x' = x ⟶ y ∈ U) uniformity)

instance
proof
  fix x y :: hyperdual
  fix a :: real
  show dist x y = norm (x − y)
    by (simp add: dist-hyperdual-def)
  show sgn x = x /R norm x
    by (simp add: sgn-hyperdual-def)
  fix U :: hyperdual set
  show open U = (∀x∈U. ∀F (x', y) in uniformity. x' = x ⟶ y ∈ U)
    using open-hyperdual-def by blast
  show (norm x = 0) = (x = 0)
  proof
```

assume $\mathit{norm}\ x = 0$
then have $(\mathit{Re}\ x)\hat{}2 + (\mathit{Eps1}\ x)\hat{}2 + (\mathit{Eps2}\ x)\hat{}2 + (\mathit{Eps12}\ x)\hat{}2 = 0$
by (*simp add*: *norm-hyperdual-def*)
then have $\mathit{Re}\ x = 0 \wedge \mathit{Eps1}\ x = 0 \wedge \mathit{Eps2}\ x = 0 \wedge \mathit{Eps12}\ x = 0$
using *power2-less-eq-zero-iff sum-power2-ge-zero*
by (*metis add.commute add.left-neutral add-nonneg-nonneg le-add-same-cancel1 zero-eq-power2*)
then show $x = 0$
by *simp*
next
assume $x = 0$
then show $\mathit{norm}\ x = 0$
by (*simp add*: *norm-hyperdual-def*)
qed
show $\mathit{norm}\ (x + y) \leq \mathit{norm}\ x + \mathit{norm}\ y$
proof –
have *norm-ge-zero*: $\forall h :: \mathit{hyperdual}\ .\ \mathit{norm}\ h \geq 0$
by (*simp add*: *norm-hyperdual-def*)
have $(\mathit{norm}\ (x + y))\hat{}2 = (\mathit{norm}\ x)\hat{}2 + (\mathit{norm}\ y)\hat{}2 + 2{*}(\mathit{Re}\ x * \mathit{Re}\ y + \mathit{Eps1}\ x * \mathit{Eps1}\ y +$
$\mathit{Eps2}\ x * \mathit{Eps2}\ y + \mathit{Eps12}\ x * \mathit{Eps12}\ y)$
proof –
have $(\mathit{norm}\ (x + y))\hat{}2 = (\mathit{Re}\ x + \mathit{Re}\ y)\hat{}2 + (\mathit{Eps1}\ x + \mathit{Eps1}\ y)\hat{}2 + (\mathit{Eps2}\ x + \mathit{Eps2}\ y)\hat{}2$
$+ (\mathit{Eps12}\ x + \mathit{Eps12}\ y)\hat{}2$
by (*simp add*: *norm-hyperdual-def*)
moreover have $\ldots = (\mathit{Re}\ x)\hat{}2 + (\mathit{Eps1}\ x)\hat{}2 + (\mathit{Eps2}\ x)\hat{}2 + (\mathit{Eps12}\ x)\hat{}2 + (\mathit{Re}\ y)\hat{}2 +$
$(\mathit{Eps1}\ y)\hat{}2 + (\mathit{Eps2}\ y)\hat{}2 + (\mathit{Eps12}\ y)\hat{}2 + 2{*}(\mathit{Re}\ x * \mathit{Re}\ y + \mathit{Eps1}\ x * \mathit{Eps1}\ y + \mathit{Eps2}\ x * \mathit{Eps2}$
$y + \mathit{Eps12}\ x * \mathit{Eps12}\ y)$
by (*simp add*: *power2-sum algebra-simps*)
moreover have $\ldots = (\mathit{norm}\ x)\hat{}2 + (\mathit{norm}\ y)\hat{}2 + 2{*}(\mathit{Re}\ x * \mathit{Re}\ y + \mathit{Eps1}\ x * \mathit{Eps1}\ y +$
$\mathit{Eps2}\ x * \mathit{Eps2}\ y + \mathit{Eps12}\ x * \mathit{Eps12}\ y)$
by (*simp add*: *norm-hyperdual-def*)
ultimately show *?thesis*
by *simp*
qed
moreover have $(\mathit{norm}\ x + \mathit{norm}\ y)\hat{}2 = (\mathit{norm}\ x)\hat{}2 + (\mathit{norm}\ y)\hat{}2 + 2{*}(\mathit{norm}\ x){*}(\mathit{norm}\ y)$
using *power2-sum* **by** *blast*
moreover have $\mathit{Re}\ x * \mathit{Re}\ y + \mathit{Eps1}\ x * \mathit{Eps1}\ y + \mathit{Eps2}\ x * \mathit{Eps2}\ y + \mathit{Eps12}\ x * \mathit{Eps12}\ y \leq$
$(\mathit{norm}\ x){*}(\mathit{norm}\ y)$
proof –
have $((\mathit{norm}\ x){*}(\mathit{norm}\ y))\hat{}2 - (\mathit{Re}\ x * \mathit{Re}\ y + \mathit{Eps1}\ x * \mathit{Eps1}\ y + \mathit{Eps2}\ x * \mathit{Eps2}\ y + \mathit{Eps12}$
$x * \mathit{Eps12}\ y)\hat{}2 \geq 0$
proof –
have $(\mathit{Re}\ x * \mathit{Re}\ y + \mathit{Eps1}\ x * \mathit{Eps1}\ y + \mathit{Eps2}\ x * \mathit{Eps2}\ y + \mathit{Eps12}\ x * \mathit{Eps12}\ y)\hat{}2 = (\mathit{Re}\ x$
$* \mathit{Re}\ y + \mathit{Eps1}\ x * \mathit{Eps1}\ y)\hat{}2 + 2{*}(\mathit{Re}\ x * \mathit{Re}\ y + \mathit{Eps1}\ x * \mathit{Eps1}\ y){*}(\mathit{Eps2}\ x * \mathit{Eps2}\ y + \mathit{Eps12}$
$x * \mathit{Eps12}\ y) + (\mathit{Eps2}\ x * \mathit{Eps2}\ y + \mathit{Eps12}\ x * \mathit{Eps12}\ y)\hat{}2$
by (*smt power2-sum*)
moreover have $\ldots = (\mathit{Re}\ x)\hat{}2 * (\mathit{Re}\ y)\hat{}2 + 2 * \mathit{Re}\ x * \mathit{Re}\ y * \mathit{Eps1}\ x * \mathit{Eps1}\ y + (\mathit{Eps1}$
$x)\hat{}2 * (\mathit{Eps1}\ y)\hat{}2 + 2{*}(\mathit{Re}\ x * \mathit{Re}\ y * \mathit{Eps2}\ x * \mathit{Eps2}\ y + \mathit{Re}\ x * \mathit{Re}\ y * \mathit{Eps12}\ x * \mathit{Eps12}\ y +$
$\mathit{Eps1}\ x * \mathit{Eps1}\ y * \mathit{Eps2}\ x * \mathit{Eps2}\ y + \mathit{Eps1}\ x * \mathit{Eps1}\ y * \mathit{Eps12}\ x * \mathit{Eps12}\ y) + (\mathit{Eps2}\ x)\hat{}2 *$
$(\mathit{Eps2}\ y)\hat{}2 + 2 * \mathit{Eps2}\ x * \mathit{Eps2}\ y * \mathit{Eps12}\ x * \mathit{Eps12}\ y + (\mathit{Eps12}\ x)\hat{}2 * (\mathit{Eps12}\ y)\hat{}2$

by (*simp add*: *distrib-left mult.commute mult.left-commute power2-sum power-mult-distrib right-diff-distrib′ scaleR-add-left scaleR-add-right*)
moreover have ((*norm x*)∗(*norm y*))ˆ2 = ((*Re x*)ˆ2 + (*Eps1 x*)ˆ2 + (*Eps2 x*)ˆ2 + (*Eps12 x*)ˆ2)∗((*Re y*)ˆ2 + (*Eps1 y*)ˆ2 + (*Eps2 y*)ˆ2 + (*Eps12 y*)ˆ2)
by (*smt add.left-commute hyperdual.sel*(*1*) *mult-numeral-1 mult-zero-left real-sqrt-pow2 sum-power2-ge-zero norm-hyperdual-def power-mult-distrib*)
moreover have ... = (*Re x*)ˆ2 ∗ (*Re y*)ˆ2 + (*Re x*)ˆ2 ∗ (*Eps1 y*)ˆ2 + (*Re x*)ˆ2 ∗ (*Eps2 y*)ˆ2 + (*Re x*)ˆ2 ∗ (*Eps12 y*)ˆ2 + (*Eps1 x*)ˆ2 ∗ (*Re y*)ˆ2 + (*Eps1 x*)ˆ2 ∗ (*Eps1 y*)ˆ2 + (*Eps1 x*)ˆ2 ∗ (*Eps2 y*)ˆ2 + (*Eps1 x*)ˆ2 ∗ (*Eps12 y*)ˆ2 + (*Eps2 x*)ˆ2 ∗ (*Re y*)ˆ2 + (*Eps2 x*)ˆ2 ∗ (*Eps1 y*)ˆ2 + (*Eps2 x*)ˆ2 ∗ (*Eps2 y*)ˆ2 + (*Eps2 x*)ˆ2 ∗ (*Eps12 y*)ˆ2 + (*Eps12 x*)ˆ2 ∗ (*Re y*)ˆ2 + (*Eps12 x*)ˆ2 ∗ (*Eps1 y*)ˆ2 + (*Eps12 x*)ˆ2 ∗ (*Eps2 y*)ˆ2 + (*Eps12 x*)ˆ2 ∗ (*Eps12 y*)ˆ2
by (*simp add*: *add.left-commute distrib-left mult.commute*)
ultimately have ((*norm x*)∗(*norm y*))ˆ2 − (*Re x* ∗ *Re y* + *Eps1 x* ∗ *Eps1 y* + *Eps2 x* ∗ *Eps2 y* + *Eps12 x* ∗ *Eps12 y*)ˆ2 = (*Re x*)ˆ2 ∗ (*Eps1 y*)ˆ2 + (*Re x*)ˆ2 ∗ (*Eps2 y*)ˆ2 + (*Re x*)ˆ2 ∗ (*Eps12 y*)ˆ2 + (*Eps1 x*)ˆ2 ∗ (*Re y*)ˆ2 + (*Eps1 x*)ˆ2 ∗ (*Eps2 y*)ˆ2 + (*Eps1 x*)ˆ2 ∗ (*Eps12 y*)ˆ2 + (*Eps2 x*)ˆ2 ∗ (*Re y*)ˆ2 + (*Eps2 x*)ˆ2 ∗ (*Eps1 y*)ˆ2 + (*Eps2 x*)ˆ2 ∗ (*Eps12 y*)ˆ2 + (*Eps12 x*)ˆ2 ∗ (*Re y*)ˆ2 + (*Eps12 x*)ˆ2 ∗ (*Eps1 y*)ˆ2 + (*Eps12 x*)ˆ2 ∗ (*Eps2 y*)ˆ2 − 2∗(*Re x* ∗ *Re y* ∗ *Eps1 x* ∗ *Eps1 y* + *Re x* ∗ *Re y* ∗ *Eps2 x* ∗ *Eps2 y* + *Re x* ∗ *Re y* ∗ *Eps12 x* ∗ *Eps12 y* + *Eps1 x* ∗ *Eps1 y* ∗ *Eps2 x* ∗ *Eps2 y* + *Eps1 x* ∗ *Eps1 y* ∗ *Eps12 x* ∗ *Eps12 y* + *Eps2 x* ∗ *Eps2 y* ∗ *Eps12 x* ∗ *Eps12 y*)
by (*simp add*: *field-simps*)
moreover have ... = (*Re x* ∗ *Eps1 y* − *Eps1 x* ∗ *Re y*)ˆ2 + (*Re x* ∗ *Eps2 y* − *Eps2 x* ∗ *Re y*)ˆ2 + (*Re x* ∗ *Eps12 y* − *Eps12 x* ∗ *Re y*)ˆ2 + (*Eps1 x* ∗ *Eps2 y* − *Eps2 x* ∗ *Eps1 y*)ˆ2 + (*Eps1 x* ∗ *Eps12 y* − *Eps12 x* ∗ *Eps1 y*)ˆ2 + (*Eps2 x* ∗ *Eps12 y* − *Eps12 x* ∗ *Eps2 y*)ˆ2
proof −
have (*Re x*)ˆ2 ∗ (*Eps1 y*)ˆ2 + (*Eps1 x*)ˆ2 ∗ (*Re y*)ˆ2 − 2 ∗ *Re x* ∗ *Eps1 y* ∗ *Eps1 x* ∗ *Re y* = (*Re x* ∗ *Eps1 y* − *Eps1 x* ∗ *Re y*)ˆ2
by (*simp add*: *power2-diff power-mult-distrib*)
moreover have (*Re x*)ˆ2 ∗ (*Eps2 y*)ˆ2 + (*Eps2 x*)ˆ2 ∗ (*Re y*)ˆ2 − 2 ∗ *Re x* ∗ *Eps2 y* ∗ *Eps2 x* ∗ *Re y* = (*Re x* ∗ *Eps2 y* − *Eps2 x* ∗ *Re y*)ˆ2
by (*simp add*: *power2-diff power-mult-distrib*)
moreover have (*Re x*)ˆ2 ∗ (*Eps12 y*)ˆ2 + (*Eps12 x*)ˆ2 ∗ (*Re y*)ˆ2 − 2 ∗ *Re x* ∗ *Eps12 y* ∗ *Eps12 x* ∗ *Re y* = (*Re x* ∗ *Eps12 y* − *Eps12 x* ∗ *Re y*)ˆ2
by (*simp add*: *power2-diff power-mult-distrib*)
moreover have (*Eps1 x*)ˆ2 ∗ (*Eps2 y*)ˆ2 + (*Eps2 x*)ˆ2 ∗ (*Eps1 y*)ˆ2 − 2 ∗ *Eps1 x* ∗ *Eps2 y* ∗ *Eps2 x* ∗ *Eps1 y* = (*Eps1 x* ∗ *Eps2 y* − *Eps2 x* ∗ *Eps1 y*)ˆ2
by (*simp add*: *power2-diff power-mult-distrib*)
moreover have (*Eps1 x*)ˆ2 ∗ (*Eps12 y*)ˆ2 + (*Eps12 x*)ˆ2 ∗ (*Eps1 y*)ˆ2 − 2 ∗ *Eps1 x* ∗ *Eps12 y* ∗ *Eps12 x* ∗ *Eps1 y* = (*Eps1 x* ∗ *Eps12 y* − *Eps12 x* ∗ *Eps1 y*)ˆ2
by (*simp add*: *power2-diff power-mult-distrib*)
moreover have (*Eps2 x*)ˆ2 ∗ (*Eps12 y*)ˆ2 + (*Eps12 x*)ˆ2 ∗ (*Eps2 y*)ˆ2 − 2 ∗ *Eps2 x* ∗ *Eps12 y* ∗ *Eps12 x* ∗ *Eps2 y* = (*Eps2 x* ∗ *Eps12 y* − *Eps12 x* ∗ *Eps2 y*)ˆ2
by (*simp add*: *power2-diff power-mult-distrib*)
ultimately show *?thesis*
by (*simp add*: *field-simps*)
qed
moreover have ... ≥ *0*
by *simp*
ultimately show *?thesis*

```
          by simp
        qed
        then show ?thesis
          by (smt norm-ge-zero mult-nonneg-nonneg power2-le-imp-le)
      qed
      ultimately have (norm (x + y))^2 ≤ (norm x + norm y)^2
        by auto
      then show ?thesis
        using power2-le-imp-le norm-ge-zero by (metis add-nonneg-nonneg)
    qed
    show norm (a *R x) = |a| ∗ norm x
    proof −
      have norm (a *R x) = sqrt(a^2 ∗ ((Re x)^2 + (Eps1 x)^2 + (Eps2 x)^2 + (Eps12 x)^2))
        by (simp add: norm-hyperdual-def power-mult-distrib distrib-left)
      then show ?thesis
        by (simp add: norm-hyperdual-def real-sqrt-mult)
    qed
    show uniformity = (INF e∈{0<..}. principal {(x :: hyperdual, y). dist x y < e})
      using uniformity-hyperdual-def by blast
  qed
  end
```

6.10 *real-normed-algebra* Counter-Example

In the following we mechanize the counter-example to hyperduals forming a real normed algebra (see Section 3.6.2):

```
lemma not-normed-algebra:
  shows ¬(∀x y :: hyperdual . norm (x ∗ y) ≤ norm x ∗ norm y)
proof −
  have norm (Hyperdual 1 1 1 1) = 2
    by (simp add: norm-hyperdual-def)
  moreover have (Hyperdual 1 1 1 1) ∗ (Hyperdual 1 1 1 1) = Hyperdual 1 2 2 4
    by (simp add: hyperdual.expand)
  moreover have norm (Hyperdual 1 2 2 4) = 5
    by (simp add: norm-hyperdual-def)
  ultimately have ∃x y :: hyperdual . norm (x ∗ y) > norm x ∗ norm y
    by (smt power2-eq-square real-sqrt-four real-sqrt-pow2)
  then show ?thesis
    by (simp add: not-le)
qed
```

6.11 Bounded Linearity of Projections

In the following we prove all four projections to be bounded linear maps (see Section 3.8):

```
lemma bounded-linear-Re: bounded-linear Re
```

```
proof
  show ⋀b1 b2. Re (b1 + b2) = Re b1 + Re b2
    by simp
  show ⋀r b. Re (r *_R b) = r *_R Re b
    by simp
  have ∀x. norm (Re x) ≤ norm x
  proof (simp add: norm-hyperdual-def)
    show ∀x. |Re x| ≤ sqrt ((Re x)^2 + (Eps1 x)^2 + (Eps2 x)^2 + (Eps12 x)^2)
      by (metis add.assoc real-sqrt-ge-abs1 real-sqrt-pow2 sum-power2-ge-zero)
  qed
  then show ∃K. ∀x. norm (Re x) ≤ norm x * K
    by (metis linordered-field-class.sign-simps(24) mult-1s(1))
qed

lemma bounded-linear-Eps1: bounded-linear Eps1
proof
  show ⋀b1 b2. Eps1 (b1 + b2) = Eps1 b1 + Eps1 b2
    by simp
  show ⋀r b. Eps1 (r *_R b) = r *_R Eps1 b
    by simp
  have ∀x. norm (Eps1 x) ≤ norm x
  proof (simp add: norm-hyperdual-def)
    show ∀x. |Eps1 x| ≤ sqrt ((Re x)^2 + (Eps1 x)^2 + (Eps2 x)^2 + (Eps12 x)^2)
      using add.assoc real-sqrt-ge-abs1 real-sqrt-pow2 sum-power2-ge-zero Groups.add-ac(2)
      by (metis (no-types, hide-lams))
  qed
  then show ∃K. ∀x. norm (Eps1 x) ≤ norm x * K
    by (metis linordered-field-class.sign-simps(24) mult-1s(1))
qed

lemma bounded-linear-Eps2: bounded-linear Eps2
proof
  show ⋀b1 b2. Eps2 (b1 + b2) = Eps2 b1 + Eps2 b2
    by simp
  show ⋀r b. Eps2 (r *_R b) = r *_R Eps2 b
    by simp
  have ∀x. norm (Eps2 x) ≤ norm x
  proof (simp add: norm-hyperdual-def)
    show ∀x. |Eps2 x| ≤ sqrt ((Re x)^2 + (Eps1 x)^2 + (Eps2 x)^2 + (Eps12 x)^2)
      using add.assoc real-sqrt-ge-abs1 real-sqrt-pow2 sum-power2-ge-zero Groups.add-ac(3)
      by metis
  qed
  then show ∃K. ∀x. norm (Eps2 x) ≤ norm x * K
    by (metis linordered-field-class.sign-simps(24) mult-1s(1))
qed

lemma bounded-linear-Eps12: bounded-linear Eps12
proof
  show ⋀b1 b2. Eps12 (b1 + b2) = Eps12 b1 + Eps12 b2
```

by *simp*
show $\bigwedge r\ b.\ Eps12\ (r *_R b) = r *_R Eps12\ b$
by *simp*
have $\forall x.\ norm\ (Eps12\ x) \le norm\ x$
proof (*simp add*: *norm-hyperdual-def*)
show $\forall x.\ |Eps12\ x| \le sqrt\ ((Re\ x)^2 + (Eps1\ x)^2 + (Eps2\ x)^2 + (Eps12\ x)^2)$
using *add.commute real-sqrt-ge-abs1 real-sqrt-pow2 sum-power2-ge-zero*
by *metis*
qed
then show $\exists K.\ \forall x.\ norm\ (Eps12\ x) \le norm\ x * K$
by (*metis linordered-field-class.sign-simps*(*24*) *mult-1s*(*1*))
qed

6.12 Limits

First, we prove the tendsto relation of a hyperdual function given those of its component functions:

lemma *tendsto-Hyperdual*:
assumes $(f \longrightarrow a)\ F$
and $(g \longrightarrow b)\ F$
and $(h \longrightarrow c)\ F$
and $(i \longrightarrow d)\ F$
shows $((\lambda x.\ Hyperdual\ (f\ x)\ (g\ x)\ (h\ x)\ (i\ x)) \longrightarrow Hyperdual\ a\ b\ c\ d)\ F$
proof –
have $((\lambda x.\ (i\ x) *_R e12) \longrightarrow d *_R e12)\ F$
using *tendsto-scaleR*[*of i d F* ($\lambda y.\ e12$) *e12*] *tendsto-const*[*of e12 F*] *assms*(*4*) **by** *simp*
then have $((\lambda x.\ (h\ x) *_R e2 + (i\ x) *_R e12) \longrightarrow c *_R e2 + d *_R e12)\ F$
using *tendsto-scaleR*[*of h c F* ($\lambda y.\ e2$) *e2*] *tendsto-const*[*of e2 F*] *assms*(*3*) *add.assoc tendsto-add* **by** *simp*
then have $((\lambda x.\ (g\ x) *_R e1 + ((h\ x) *_R e2 + (i\ x) *_R e12)) \longrightarrow b *_R e1 + (c *_R e2 + d *_R e12))\ F$
using *tendsto-scaleR*[*of g b F* ($\lambda y.\ e1$) *e1*] *tendsto-const*[*of e1 F*] *assms*(*2*) *tendsto-add* **by** *simp*
then have $((\lambda x.\ (f\ x) *_R re + ((g\ x) *_R e1 + ((h\ x) *_R e2 + (i\ x) *_R e12))) \longrightarrow a *_R re + (b *_R e1 + (c *_R e2 + d *_R e12)))\ F$
using *tendsto-scaleR*[*of f a F* ($\lambda y.\ re$) *re*] *tendsto-const*[*of re F*] *assms*(*1*) *add.assoc tendsto-add* **by** *simp*
then show *?thesis*
unfolding *Hyperdual-eq* **by** (*simp add*: *add.assoc*)

Then we prove the full equivalence for limits of hyperdual-valued functions (see Section 3.10):

lemma *tendsto-hyperdual-iff*:
$((f :: {'a} \Rightarrow hyperdual) \longrightarrow x)\ F$
$\longleftrightarrow (\ \ ((\lambda x.\ Re\ (f\ x)) \longrightarrow Re\ x)\ F$
$\wedge ((\lambda x.\ Eps1\ (f\ x)) \longrightarrow Eps1\ x)\ F$
$\wedge ((\lambda x.\ Eps2\ (f\ x)) \longrightarrow Eps2\ x)\ F$
$\wedge ((\lambda x.\ Eps12\ (f\ x)) \longrightarrow Eps12\ x)\ F)$

```
proof safe
 assume (f ⟶ x) F
 then show ((λx. Re (f x)) ⟶ Re x) F
      and ((λx. Eps1 (f x)) ⟶ Eps1 x) F
      and ((λx. Eps2 (f x)) ⟶ Eps2 x) F
      and ((λx. Eps12 (f x)) ⟶ Eps12 x) F
   by (simp-all add: tendsto-Re tendsto-Eps1 tendsto-Eps2 tendsto-Eps12)
next
 assume ((λx. Re (f x)) ⟶ Re x) F
   and ((λx. Eps1 (f x)) ⟶ Eps1 x) F
   and ((λx. Eps2 (f x)) ⟶ Eps2 x) F
   and ((λx. Eps12 (f x)) ⟶ Eps12 x) F
 moreover have (λx. Hyperdual (Re (f x)) (Eps1 (f x)) (Eps2 (f x)) (Eps12 (f x))) = f
   by simp
 ultimately show (f ⟶ x) F
   using tendsto-Hyperdual[of λx. Re (f x) Re x F λx. Eps1 (f x) Eps1 x λx. Eps2 (f x) Eps2 x
λx. Eps12 (f x) Eps12 x]
   by simp
qed
```

6.13 *banach* Instantiation

We prove hyperdual numbers to be an instance of the Banach space type class *banach* by proving that every Cauchy sequence converges:

```
instance hyperdual :: banach
proof
 fix X :: nat ⇒ hyperdual
 assume Cauchy X
 then have (λn. Hyperdual (Re (X n)) (Eps1 (X n)) (Eps2 (X n)) (Eps12 (X n))) ⟶
     Hyperdual (lim (λn. Re (X n))) (lim (λn. Eps1 (X n))) (lim (λn. Eps2 (X n))) (lim (λn.
Eps12 (X n)))
   using Cauchy-Re[of X] Cauchy-Eps1[of X] Cauchy-Eps2[of X] Cauchy-Eps12[of X]
       Cauchy-convergent-iff convergent-LIMSEQ-iff
        tendsto-Hyperdual[of λn. Re (X n) lim (λn. Re (X n)) sequentially λn. Eps1 (X n) lim
(λn. Eps1 (X n)) λn. Eps2 (X n) lim (λn. Eps2 (X n)) λn. Eps12 (X n) lim (λn. Eps12 (X n))]
   by blast
 then show convergent X
   using convergentI by simp
qed
```

6.14 Derivatives

We prove the derivative equivalence for hyperdual-valued functions (see Section 3.11):

```
lemma has-derivative-hyperdual-iff: (f has-derivative f′) F ⟷
   ((λx. Re (f x)) has-derivative (λx. Re (f′ x))) F ∧
```

((λ*x*. *Eps1* (*f x*)) *has-derivative* (λ*x*. *Eps1* (*f′ x*))) *F* ∧
((λ*x*. *Eps2* (*f x*)) *has-derivative* (λ*x*. *Eps2* (*f′ x*))) *F* ∧
((λ*x*. *Eps12* (*f x*)) *has-derivative* (λ*x*. *Eps12* (*f′ x*))) *F*
proof *safe*
assume *assm*: (*f has-derivative f′*) *F*
show ((λ*x*. *Re* (*f x*)) *has-derivative* (λ*x*. *Re* (*f′ x*))) *F*
using *assm has-derivative-Re* **by** *blast*
show ((λ*x*. *Eps1* (*f x*)) *has-derivative* (λ*x*. *Eps1* (*f′ x*))) *F*
using *assm has-derivative-Eps1* **by** *blast*
show ((λ*x*. *Eps2* (*f x*)) *has-derivative* (λ*x*. *Eps2* (*f′ x*))) *F*
using *assm has-derivative-Eps2* **by** *blast*
show ((λ*x*. *Eps12* (*f x*)) *has-derivative* (λ*x*. *Eps12* (*f′ x*))) *F*
using *assm has-derivative-Eps12* **by** *blast*
next
assume *assmRe*: ((λ*x*. *Re* (*f x*)) *has-derivative* (λ*x*. *Re* (*f′ x*))) *F*
and *assmEps1*: ((λ*x*. *Eps1* (*f x*)) *has-derivative* (λ*x*. *Eps1* (*f′ x*))) *F*
and *assmEps2*: ((λ*x*. *Eps2* (*f x*)) *has-derivative* (λ*x*. *Eps2* (*f′ x*))) *F*
and *assmEps12*: ((λ*x*. *Eps12* (*f x*)) *has-derivative* (λ*x*. *Eps12* (*f′ x*))) *F*
have ((λ*y*. ((*Re* (*f y*) − *Re* (*f* (*Lim F* (λ*x*. *x*)))) − *Re* (*f′* (*y* − *Lim F* (λ*x*. *x*)))) $/_R$ *norm* (*y* − *Lim F* (λ*x*. *x*))) ⟶ *0*) *F*
using *assmRe has-derivative-def*[*of* (λ*x*. *Re* (*f x*)) (λ*x*. *Re* (*f′ x*)) *F*] **by** *blast*
then have *re*: ((λ*y*. *Re* (((*f y* − *f* (*Lim F* (λ*x*. *x*))) − *f′* (*y* − *Lim F* (λ*x*. *x*))) $/_R$ *norm* (*y* − *Lim F* (λ*x*. *x*)))) ⟶ *Re 0*) *F*
by *simp*

have ((λ*y*. ((*Eps1* (*f y*) − *Eps1* (*f* (*Lim F* (λ*x*. *x*)))) − *Eps1* (*f′* (*y* − *Lim F* (λ*x*. *x*)))) $/_R$ *norm* (*y* − *Lim F* (λ*x*. *x*))) ⟶ *0*) *F*
using *assmEps1 has-derivative-def*[*of* (λ*x*. *Eps1* (*f x*)) (λ*x*. *Eps1* (*f′ x*)) *F*] **by** *blast*
then have *eps1*: ((λ*y*. *Eps1* (((*f y* − *f* (*Lim F* (λ*x*. *x*))) − *f′* (*y* − *Lim F* (λ*x*. *x*))) $/_R$ *norm* (*y* − *Lim F* (λ*x*. *x*)))) ⟶ *Re 0*) *F*
by *simp*

have ((λ*y*. ((*Eps2* (*f y*) − *Eps2* (*f* (*Lim F* (λ*x*. *x*)))) − *Eps2* (*f′* (*y* − *Lim F* (λ*x*. *x*)))) $/_R$ *norm* (*y* − *Lim F* (λ*x*. *x*))) ⟶ *0*) *F*
using *assmEps2 has-derivative-def*[*of* (λ*x*. *Eps2* (*f x*)) (λ*x*. *Eps2* (*f′ x*)) *F*] **by** *blast*
then have *eps2*: ((λ*y*. *Eps2* (((*f y* − *f* (*Lim F* (λ*x*. *x*))) − *f′* (*y* − *Lim F* (λ*x*. *x*))) $/_R$ *norm* (*y* − *Lim F* (λ*x*. *x*)))) ⟶ *Re 0*) *F*
by *simp*

have ((λ*y*. ((*Eps12* (*f y*) − *Eps12* (*f* (*Lim F* (λ*x*. *x*)))) − *Eps12* (*f′* (*y* − *Lim F* (λ*x*. *x*)))) $/_R$ *norm* (*y* − *Lim F* (λ*x*. *x*))) ⟶ *0*) *F*
using *assmEps12 has-derivative-def*[*of* (λ*x*. *Eps12* (*f x*)) (λ*x*. *Eps12* (*f′ x*)) *F*] **by** *blast*
then have *eps12*: ((λ*y*. *Eps12* (((*f y* − *f* (*Lim F* (λ*x*. *x*))) − *f′* (*y* − *Lim F* (λ*x*. *x*))) $/_R$ *norm* (*y* − *Lim F* (λ*x*. *x*)))) ⟶ *Re 0*) *F*
by *simp*

have ((λ*y*. ((*f y* − *f* (*Lim F* (λ*x*. *x*))) − *f′* (*y* − *Lim F* (λ*x*. *x*))) $/_R$ *norm* (*y* − *Lim F* (λ*x*. *x*))) ⟶ *0*) *F*
using *re eps1 eps2 eps12* **by** (*simp add*: *tendsto-hyperdual-iff*)

```
moreover have bounded-linear f'
proof
  have bl-ref': bounded-linear (λx. Re (f' x))
    using assmRe has-derivative-def by blast
  have bl-eps1f': bounded-linear (λx. Eps1 (f' x))
    using assmEps1 has-derivative-def by blast
  have bl-eps2f': bounded-linear (λx. Eps2 (f' x))
    using assmEps2 has-derivative-def by blast
  have bl-eps12f': bounded-linear (λx. Eps12 (f' x))
    using assmEps12 has-derivative-def by blast

  have l-ref': linear (λx. Re (f' x))
    using bl-ref' bounded-linear.linear by blast
  have l-eps1f': linear (λx. Eps1 (f' x))
    using bl-eps1f' bounded-linear.linear by blast
  have l-eps2f': linear (λx. Eps2 (f' x))
    using bl-eps2f' bounded-linear.linear by blast
  have l-eps12f': linear (λx. Eps12 (f' x))
    using bl-eps12f' bounded-linear.linear by blast

  show ⋀x y. f' (x + y) = f' x + f' y
  proof –
    fix x y :: 'a
     have f' (x + y) = Hyperdual (Re (f' (x + y))) (Eps1 (f' (x + y))) (Eps2 (f' (x + y)))
(Eps12 (f' (x + y)))
      by simp
    moreover have Re (f' (x + y)) = Re (f' x) + Re (f' y)
      using l-ref' by (simp add: linear-iff)
    moreover have Eps1 (f' (x + y)) = Eps1 (f' x) + Eps1 (f' y)
      using l-eps1f' by (simp add: linear-iff)
    moreover have Eps2 (f' (x + y)) = Eps2 (f' x) + Eps2 (f' y)
      using l-eps2f' by (simp add: linear-iff)
    moreover have Eps12 (f' (x + y)) = Eps12 (f' x) + Eps12 (f' y)
      using l-eps12f' by (simp add: linear-iff)
    ultimately have f' (x + y) = Hyperdual (Re (f' x) + Re (f' y)) (Eps1 (f' x) + Eps1 (f' y))
(Eps2 (f' x) + Eps2 (f' y)) (Eps12 (f' x) + Eps12 (f' y))
      by simp
    then have f' (x + y) = Hyperdual (Re (f' x)) (Eps1 (f' x)) (Eps2 (f' x)) (Eps12 (f' x)) +
Hyperdual (Re (f' y)) (Eps1 (f' y)) (Eps2 (f' y)) (Eps12 (f' y))
      by (simp add: plus-hyperdual.code)
    then show f' (x + y) = f' x + f' y
      by simp
  qed

  show ⋀x y. f' (x *R y) = x *R f' y
  proof –
    fix x :: real
    fix y
     have f' (x *R y) = Hyperdual (Re (f' (x *R y))) (Eps1 (f' (x *R y))) (Eps2 (f' (x *R y)))
```

```
(Eps12 (f′ (x ∗R y)))
      by simp
    moreover have Re (f′ (x ∗R y)) = x ∗R Re (f′ y)
      using l-ref′ by (simp add: linear-iff)
    moreover have Eps1 (f′ (x ∗R y)) = x ∗R Eps1 (f′ y)
      using l-eps1f′ by (simp add: linear-iff)
    moreover have Eps2 (f′ (x ∗R y)) = x ∗R Eps2 (f′ y)
      using l-eps2f′ by (simp add: linear-iff)
    moreover have Eps12 (f′ (x ∗R y)) = x ∗R Eps12 (f′ y)
      using l-eps12f′ by (simp add: linear-iff)
    ultimately have f′ (x ∗R y) = Hyperdual (x ∗R Re (f′ y)) (x ∗R Eps1 (f′ y)) (x ∗R Eps2 (f′
y)) (x ∗R Eps12 (f′ y))
      by simp
    then have f′ (x ∗R y) = x ∗R Hyperdual (Re (f′ y)) (Eps1 (f′ y)) (Eps2 (f′ y)) (Eps12 (f′
y))
      by (simp add: scaleR-hyperdual.code)
    then show f′ (x ∗R y) = x ∗R f′ y
      by simp
  qed

  show ∃K. ∀x. norm (f′ x) ≤ norm x ∗ K
  proof −
    obtain k-re where ∀x. (norm (Re (f′ x)))^2 ≤ (norm x ∗ k-re)^2
      using bl-ref′ bounded-linear.bounded norm-ge-zero power-mono by metis
    moreover obtain k-eps1 where ∀x. (norm (Eps1 (f′ x)))^2 ≤ (norm x ∗ k-eps1)^2
      using bl-eps1f′ bounded-linear.bounded norm-ge-zero power-mono by metis
    moreover obtain k-eps2 where ∀x. (norm (Eps2 (f′ x)))^2 ≤ (norm x ∗ k-eps2)^2
      using bl-eps2f′ bounded-linear.bounded norm-ge-zero power-mono by metis
    moreover obtain k-eps12 where ∀x. (norm (Eps12 (f′ x)))^2 ≤ (norm x ∗ k-eps12)^2
      using bl-eps12f′ bounded-linear.bounded norm-ge-zero power-mono by metis
    moreover have ∀x. (norm (f′ x))^2 = (norm (Re (f′ x)))^2 + (norm (Eps1 (f′ x)))^2 +
(norm (Eps2 (f′ x)))^2 + (norm (Eps12 (f′ x)))^2
      using norm-hyperdual-def by (simp add: norm-hyperdual-def)
    ultimately have ∀x. (norm (f′ x))^2 ≤ (norm x ∗ k-re)^2 + (norm x ∗ k-eps1)^2 + (norm
x ∗ k-eps2)^2 + (norm x ∗ k-eps12)^2
      by smt
    then have ∀x. (norm (f′ x))^2 ≤ (norm x)^2 ∗ (k-re^2 + k-eps1^2 + k-eps2^2 + k-eps12^2)
      by (simp add: distrib-left power-mult-distrib)
    then have final: ∀x. norm (f′ x) ≤ norm x ∗ sqrt(k-re^2 + k-eps1^2 + k-eps2^2 + k-eps12^2)
      using real-le-rsqrt real-sqrt-mult real-sqrt-pow2 by fastforce
    then show ∃K. ∀x. norm (f′ x) ≤ norm x ∗ K
      by blast
  qed
 qed
 ultimately show (f has-derivative f′) F
  by (simp add: has-derivative-def)
qed
```

6.15 Second Field Derivative

We define the second field derivative locale as follows:

locale *has-snd-field-derivative* =
  fixes f :: ($'a$ :: *real-normed-field*) ⇒ $'a$
    and f' :: $'a$ ⇒ $'a$
    and f'' :: $'a$ ⇒ $'a$
    and S :: $'a$ *set*
    and a :: $'a$
  assumes *deriv-f*: (f *has-field-derivative* f' a) (*at* a *within* S)
      and *deriv-f'*: (f' *has-field-derivative* f'' a) (*at* a *within* S)

Then we prove that this relation is preserved on taking a subset of the given set:

lemma *has-snd-field-derivative-subset*:
  assumes $T \subseteq S$
      and *has-snd-field-derivative* f f' f'' S a
  shows *has-snd-field-derivative* f f' f'' T a
proof
  show (f *has-field-derivative* f' a) (*at* a *within* T)
    using *assms has-snd-field-derivative.deriv-f has-field-derivative-subset* **by** *blast*
  show (f' *has-field-derivative* f'' a) (*at* a *within* T)
    using *assms has-snd-field-derivative.deriv-f' has-field-derivative-subset* **by** *blast*
qed

6.16 Hyperdual Extension - Field Version

We define the field derivative version of the hyperdual extension locale as follows:

locale *hyperdual-ext* =
  has-snd-field-derivative f f' f'' S a
    for f **and** f' **and** f'' **and** S **and** a +
  fixes g :: *hyperdual* ⇒ *hyperdual*
  assumes *re-g*: *Re* (g x) = f (*Re* x)
      and *eps1-g*: *Eps1* (g x) = *Eps1* x ∗ f' (*Re* x)
      and *eps2-g*: *Eps2* (g x) = *Eps2* x ∗ f' (*Re* x)
      and *eps12-g*: *Eps12* (g x) = *Eps12* x ∗ f' (*Re* x) + *Eps1* x ∗ *Eps2* x ∗ f'' (*Re* x)

Next, we prove the derivative extraction theorems:

lemma *extract-f*:
  shows *Re* (g (*Hyperdual* x *1 1 0*)) = f x
by (*simp add*: *re-g*)

lemma *extract-f'*:
  shows *Eps1* (g (*Hyperdual* x *1 1 0*)) = f' x
    and *Eps2* (g (*Hyperdual* x *1 1 0*)) = f' x
by (*simp-all add*: *eps1-g eps2-g*)

lemma *extract-f''*:
  shows *Eps12* (g (*Hyperdual* x *1 1 0*)) = f'' x

by (*simp add*: *eps12-g*)

Next, we prove that this locale is preserved on taking subset of the given set:

lemma *subset*:
  assumes $T \subseteq S$
  shows *hyperdual-ext* $f\, f'\, f''\, T\, a\, g$
proof
  show (f *has-real-derivative* $f'\, a$) (*at* a *within* T)
    using *assms deriv-f has-field-derivative-subset* **by** *blast*
  show (f' *has-real-derivative* $f''\, a$) (*at* a *within* T)
    using *assms deriv-f' has-field-derivative-subset* **by** *blast*
  fix x :: *hyperdual*
  show *Re* $(g\, x) = f$ (*Re* x)
    and *Eps1* $(g\, x) =$ *Eps1* $x * f'$ (*Re* x)
    and *Eps2* $(g\, x) =$ *Eps2* $x * f'$ (*Re* x)
    and *Eps12* $(g\, x) =$ *Eps12* $x * f'$ (*Re* x) $+$ *Eps1* $x *$ *Eps2* $x * f''$ (*Re* x)
    using *re-g eps1-g eps2-g eps12-g* **by** *simp-all*
qed

Next, we prove that this locale is preserved on composition with another instance:

lemma *compose*:
  assumes *hyperdual-ext* $m\, m'\, m''$ (f‘S) ($f\, a$) n
    shows *hyperdual-ext* ($\lambda x.\ m(f\, x)$) ($\lambda x.\ m'(f\, x) * f'\, x$) ($\lambda x.\ m'(f\, x) * f''\, x + m''(f\, x) * f'\, x * f'$
x) $S\, a$ ($\lambda x.\ n(g\, x)$)
proof
  show (($\lambda x.\ m\ (f\, x)$) *has-real-derivative* $m'\ (f\, a) * f'\, a$) (*at* a *within* S)
    using *deriv-f assms*
        has-field-derivative-imp-has-derivative[*of* $f\, f\, f'\, a$ *at* a *within* S]
        has-snd-field-derivative.deriv-f[*of* $m\, m'\, m''\, f$‘$S\, f\, a$]
        has-field-derivative-imp-has-derivative[*of* $m\, m'$ ($f\, a$) *at* ($f\, a$) *within* f‘S]
        has-derivative-in-compose[*of* $f\, f$ $(*)$ ($f'\, a$) $a\, S\, m$ $(*)$ (m' ($f\, a$))]
    by (*simp add*: *has-derivative-imp-has-field-derivative hyperdual-ext-def*)
  show (($\lambda x.\ m'\ (f\, x) * f'\, x$) *has-real-derivative* $m'\ (f\, a) * f''\, a + m''\ (f\, a) * f'\, a * f'\, a$) (*at* a
within S)
    using *deriv-f deriv-f' assms*
        has-field-derivative-imp-has-derivative[*of* $f\, f\, f'\, a$ *at* a *within* S]
        has-snd-field-derivative.deriv-f'[*of* $m\, m'\, m''\, f$‘$S\, f\, a$]
        has-field-derivative-imp-has-derivative[*of* $m'\, m''$ ($f\, a$) *at* ($f\, a$) *within* f‘S]
        has-derivative-in-compose[*of* $f\, f$ $(*)$ ($f'\, a$) $a\, S\, m'$ $(*)$ (m'' ($f\, a$))]
        has-field-derivative-imp-has-derivative[*of* $f'\, f''\, a$ *at* a *within* S]
        has-derivative-mult[*of* $\lambda x.\ m'\ (f\, x)$ $\lambda x.\ m''\ (f\, a) * (f'\, a * x)$ $a\, S\, f'$ $(*)$ ($f''\, a$)]
        has-derivative-imp-has-field-derivative[*of* $\lambda x.\ m'\ (f\, x) * f'\, x$
          $\lambda h.\ m'\ (f\, a) * (f''\, a * h) + m''\ (f\, a) * (f'\, a * h) * f'\, a$ *at* a *within* S $m'\ (f\, a) * f''\, a + m''$
($f\, a$) $* f'\, a * f'\, a$]
    by (*simp add*: *distrib-left hyperdual-ext-def*)
  fix x :: *hyperdual*
  show *Re* (n ($g\, x$)) $= m$ (f (*Re* x))
    using *assms hyperdual-ext.re-g re-g* **by** *auto*
  show *Eps1* (n ($g\, x$)) $=$ *Eps1* $x *$ (m' (f (*Re* x)) $* f'$ (*Re* x))

```
    using assms eps1-g hyperdual-ext.eps1-g re-g by auto
  show Eps2 (n (g x)) = Eps2 x ∗ (m′ (f (Re x)) ∗ f′ (Re x))
    using assms eps2-g hyperdual-ext.eps2-g re-g by auto
  have Eps12 (n (g x)) = (Eps12 x ∗ f′ (Re x) + Eps1 x ∗ Eps2 x ∗ f″ (Re x)) ∗ m′ (f (Re x)) +
      Eps1 x ∗ f′ (Re x) ∗ Eps2 x ∗ f′ (Re x) ∗ m″ (f (Re x))
    using assms re-g[of x] eps1-g[of x] eps2-g[of x] eps12-g[of x]
      hyperdual-ext.eps1-g[of m m′ m″ f‘S f a n g x]
      hyperdual-ext.eps2-g[of m m′ m″ f‘S f a n g x]
      hyperdual-ext.eps12-g[of m m′ m″ f‘S f a n g x]
    by simp
  moreover have ... = Eps12 x ∗ f′ (Re x) ∗ m′ (f (Re x)) +
          Eps1 x ∗ Eps2 x ∗ f″ (Re x) ∗ m′ (f (Re x)) +
          Eps1 x ∗ Eps2 x ∗ m″ (f (Re x)) ∗ f′ (Re x) ∗ f′ (Re x)
    by (simp add: distrib-right mult.assoc)
  ultimately show Eps12 (n (g x)) = Eps12 x ∗ (m′ (f (Re x)) ∗ f′ (Re x)) +
      Eps1 x ∗ Eps2 x ∗ (m′ (f (Re x)) ∗ f″ (Re x) + m″ (f (Re x)) ∗ f′ (Re x) ∗ f′ (Re x))
    by (simp add: distrib-left)
qed
```

Last, we prove uniqueness of the extension given the real functions, variable and set:

```
lemma unique:
  assumes hyperdual-ext f f′ f″ S a h
  shows g = h
proof
  fix x :: hyperdual
  have Re (g x) = Re (h x)
    using assms hyperdual-ext.re-g re-g by simp
  moreover have Eps1 (g x) = Eps1 (h x)
    using assms hyperdual-ext.eps1-g eps1-g by simp
  moreover have Eps2 (g x) = Eps2 (h x)
    using assms hyperdual-ext.eps2-g eps2-g by simp
  moreover have Eps12 (g x) = Eps12 (h x)
    using assms hyperdual-ext.eps12-g eps12-g by simp
  ultimately show g x = h x
    by simp
qed
```

6.17 Hyperdual Extension - Examples

We prove a number of concrete extensions of real functions.

6.17.1 Constant Function

We prove the *of-real* function to be the hyperdual extension of a real constant:

```
lemma hyperdual-ext-const:
  hyperdual-ext (λx. a) (λx. 0) (λx. 0) UNIV x (λx. of-real a)
proof
  show ((λx. a) has-field-derivative 0) (at x)
```

```
    by simp
  show ((λx. 0) has-field-derivative 0) (at x)
    by simp
  show ⋀x. Re (of-real a) = a
    and ⋀x. Eps1 (of-real a) = Eps1 x ∗ 0
    and ⋀x. Eps2 (of-real a) = Eps2 x ∗ 0
    and ⋀x. Eps12 (of-real a) = Eps12 x ∗ 0 + Eps1 x ∗ Eps2 x ∗ 0
    by simp-all
qed
```

6.17.2 Identity Function

We prove the hyperdual identity to be the extension of the real identity:

```
lemma hyperdual-ext-identity:
  hyperdual-ext (λx. x) (λx. 1) (λx. 0) UNIV x (λx. x)
proof
  show ((λx. x) has-field-derivative 1) (at x)
    using has-derivative-ident by simp
  show ((λx. 1) has-field-derivative 0) (at x)
    by simp
  show ⋀x. Re x = Re x
    and ⋀x. Eps1 x = Eps1 x ∗ 1
    and ⋀x. Eps2 x = Eps2 x ∗ 1
    and ⋀x. Eps12 x = Eps12 x ∗ 1 + Eps1 x ∗ Eps2 x ∗ 0
    by simp-all
qed
```

6.17.3 Addition

We prove addition of hyperdual extensions to be the hyperdual extension of real addition:

```
lemma hyperdual-ext-add:
  assumes hyperdual-ext f f′ f″ S a fE
    and hyperdual-ext g g′ g″ S a gE
  shows hyperdual-ext (λx. f x + g x) (λx. f′ x + g′ x) (λx. f″ x + g″ x) S a (λx. fE x + gE x)
proof
  show ((λx. f x + g x) has-field-derivative f′ a + g′ a) (at a within S)
    using assms hyperdual-ext-def
      Deriv.field-differentiable-add[of f f′ a at a within S g g′ a]
    by (simp add: has-snd-field-derivative-def)
  show ((λx. f′ x + g′ x) has-field-derivative f″ a + g″ a) (at a within S)
    using hyperdual-ext-def assms
      Deriv.field-differentiable-add[of f′ f″ a at a within S g′ g″ a]
    by (simp add: has-snd-field-derivative-def)
  fix x :: hyperdual
  show Re (fE x + gE x) = f (Re x) + g (Re x)
    using assms hyperdual-ext.re-g by auto
  show Eps1 (fE x + gE x) = Eps1 x ∗ (f′ (Re x) + g′ (Re x))
```

```
    using assms
        hyperdual-ext.eps1-g[of f f′ f″ S a fE x]
        hyperdual-ext.eps1-g[of g g′ g″ S a gE x]
    by (simp add: distrib-left)
  show Eps2 (fE x + gE x) = Eps2 x ∗ (f′ (Re x) + g′ (Re x))
    using assms
        hyperdual-ext.eps2-g[of f f′ f″ S a fE x]
        hyperdual-ext.eps2-g[of g g′ g″ S a gE x]
    by (simp add: distrib-left)
  show Eps12 (fE x + gE x) = Eps12 x ∗ (f′ (Re x) + g′ (Re x)) + Eps1 x ∗ Eps2 x ∗ (f″ (Re
x) + g″ (Re x))
    using assms
        hyperdual-ext.eps12-g[of f f′ f″ S a fE x]
        hyperdual-ext.eps12-g[of g g′ g″ S a gE x]
    by (simp add: distrib-left)
qed
```

6.17.4 Addition of Constant

We prove by preservation under composition the hyperdual extension of the special case where a constant is one of the summands:

```
lemma hyperdual-ext-add-const:
  hyperdual-ext (λx. x + a) (λx. 1) (λx. 0) UNIV x (λx. x + of-real a)
using hyperdual-ext-add[of λx. x λx. 1 λx. 0 UNIV x λx. x λx. a λx. 0 λx. 0 λx. of-real a]
      hyperdual-ext-const[of a x]
      hyperdual-ext-identity
  by simp
```

6.17.5 Scalar Multiplication

We prove hyperdual scalar multiplication to be the hyperdual extension of real multiplication by a constant:

```
lemma hyperdual-ext-scaleR:
  hyperdual-ext (λx. k ∗R x) (λx. k) (λx. 0) UNIV x ((∗R) k)
proof
  show ((λx. k ∗R x) has-field-derivative k) (at x)
    by simp
  show ((λx. k) has-field-derivative 0) (at x)
    by simp
  show ⋀x. Re (k ∗R x) = k ∗R Re x
    and ⋀x. Eps1 (k ∗R x) = Eps1 x ∗ k
    and ⋀x. Eps2 (k ∗R x) = Eps2 x ∗ k
    and ⋀x. Eps12 (k ∗R x) = Eps12 x ∗ k + Eps1 x ∗ Eps2 x ∗ 0
    by simp-all
qed
```

6.17.6 Linear Function

We prove the hyperdual linear function to be the hyperdual extension of the real linear function by composing the identity, scalar multiplication and addition of constant extensions:

lemma *hyperdual-ext-linear*:
  hyperdual-ext (λ*x*. *k* $*_R$ *x* + *a*) (λ*x*. *k*) (λ*x*. *0*) *UNIV* *x* (λ*x*. *k* $*_R$ *x* + *of-real a*)
proof –
  have *hyperdual-ext* (λ*x*. *x* + *a*) (λ*x*. *1*) (λ*x*. *0*) (*range* (($*_R$) *k*)) (*k* $*_R$ *x*) (λ*x*. *x* + *of-real a*)
    using *hyperdual-ext-add-const*[*of a k* $*_R$ *x*]
      hyperdual-ext.subset[*of* λ*x*. *x* + *a* λ*x*. *1* λ*x*. *0* *UNIV* *k* $*_R$ *x* λ*x*. *x* + *of-real a range* (($*_R$) *k*)]
      subset-UNIV[*of range* ((∗) *k*)]
    by *simp*
  then show *?thesis*
    using *hyperdual-ext-scaleR*[*of k x*]
      hyperdual-ext.compose[*of* (∗) *k* λ*x*. *k* λ*x*. *0* *UNIV* *x* ($*_R$) *k* λ*x*. *x* + *a* λ*x*. *1* λ*x*. *0* λ*x*. *x* + *of-real a*]
    by *simp*
qed

6.17.7 Exponential Function

First we define the hyperdual extension of the exponential function:

primcorec *hyperdual-exp* :: *hyperdual* ⇒ *hyperdual*
  where
   Re (*hyperdual-exp x*) = *exp* (*Re x*)
  | *Eps1* (*hyperdual-exp x*) = *Eps1 x* ∗ *exp* (*Re x*)
  | *Eps2* (*hyperdual-exp x*) = *Eps2 x* ∗ *exp* (*Re x*)
  | *Eps12* (*hyperdual-exp x*) = *Eps12 x* ∗ *exp* (*Re x*) + *Eps1 x* ∗ *Eps2 x* ∗ *exp* (*Re x*)

Then we prove this to be the hyperdual extension of the real exponential function:

lemma *hyperdual-ext-exp*:
  hyperdual-ext exp exp exp UNIV x hyperdual-exp
proof
  show (*exp has-field-derivative exp x*) (*at x*)
    by *simp*
  fix *x* :: *hyperdual*
  show *Re* (*hyperdual-exp x*) = *exp* (*Re x*)
    by *simp*
  show *Eps1* (*hyperdual-exp x*) = *Eps1 x* ∗ *exp* (*Re x*)
    by *simp*
  show *Eps2* (*hyperdual-exp x*) = *Eps2 x* ∗ *exp* (*Re x*)
    by *simp*
  show *Eps12* (*hyperdual-exp x*) = *Eps12 x* ∗ *exp* (*Re x*) + *Eps1 x* ∗ *Eps2 x* ∗ *exp* (*Re x*)
    by *simp*
qed

6.17.8 Sine

First we define the hyperdual extension of sine:

primcorec *hyperdual-sin* :: *hyperdual* ⇒ *hyperdual*
  where
  Re (*hyperdual-sin x*) = *sin* (*Re x*)
  | *Eps1* (*hyperdual-sin x*) = (*Eps1 x*) ∗ (*cos* (*Re x*))
  | *Eps2* (*hyperdual-sin x*) = (*Eps2 x*) ∗ (*cos* (*Re x*))
  | *Eps12* (*hyperdual-sin x*) = (*Eps12 x*) ∗ (*cos* (*Re x*)) + (*Eps1 x*) ∗ (*Eps2 x*) ∗ (− *sin* (*Re x*))

Then we prove this to be the hyperdual extension of the real sine:

lemma *hyperdual-ext-sin*:
  hyperdual-ext sin cos (− *sin*) *UNIV x hyperdual-sin*
proof
  show (*sin has-field-derivative cos x*) (*at x*)
    by *simp*
  show (*cos has-field-derivative* (− *sin*) *x*) (*at x*)
    by *simp*
  fix *x* :: *hyperdual*
  show *Re* (*hyperdual-sin x*) = *sin* (*Re x*)
    by *simp*
  show *Eps1* (*hyperdual-sin x*) = *Eps1 x* ∗ *cos* (*Re x*)
    by *simp*
  show *Eps2* (*hyperdual-sin x*) = *Eps2 x* ∗ *cos* (*Re x*)
    by *simp*
  show *Eps12* (*hyperdual-sin x*) = *Eps12 x* ∗ *cos* (*Re x*) + *Eps1 x* ∗ *Eps2 x* ∗ (− *sin*) (*Re x*)
    by *simp*
qed

6.17.9 Cosine

First we define the hyperdual extension of cosine:

primcorec *hyperdual-cos* :: *hyperdual* ⇒ *hyperdual*
  where
  Re (*hyperdual-cos x*) = *cos* (*Re x*)
  | *Eps1* (*hyperdual-cos x*) = (*Eps1 x*) ∗ (− *sin* (*Re x*))
  | *Eps2* (*hyperdual-cos x*) = (*Eps2 x*) ∗ (− *sin* (*Re x*))
  | *Eps12* (*hyperdual-cos x*) = (*Eps12 x*) ∗ (− *sin* (*Re x*)) + (*Eps1 x*) ∗ (*Eps2 x*) ∗ (− *cos* (*Re x*))

Then we prove this to be the hyperdual extension of the real cosine:

lemma *hyperdual-ext-cos*:
  hyperdual-ext cos (λ*x*. − *sin x*) (λ*x*. − *cos x*) *UNIV x hyperdual-cos*
proof
  show (*cos has-field-derivative* − *sin x*) (*at x*)
    by *simp*
  show ((λ*x*. − *sin x*) *has-field-derivative* − *cos x*) (*at x*)
    using *DERIV-sin*[*of x*] *has-field-derivative-scaleR-right*[*of sin cos x at x* −*1*]
    by *simp*

```
  fix x :: hyperdual
  show Re (hyperdual-cos x) = cos (Re x)
    by simp
  show Eps1 (hyperdual-cos x) = Eps1 x ∗ − sin (Re x)
    by simp
  show Eps2 (hyperdual-cos x) = Eps2 x ∗ − sin (Re x)
    by simp
  show Eps12 (hyperdual-cos x) = Eps12 x ∗ − sin (Re x) + Eps1 x ∗ Eps2 x ∗ − cos (Re x)
    by simp
qed
```

6.17.10 Square Root

First we define the hyperdual extension of square root:

```
primcorec hyperdual-sqrt :: hyperdual ⇒ hyperdual
  where
    Re (hyperdual-sqrt x) = sqrt (Re x)
  | Eps1 (hyperdual-sqrt x) = Eps1 x ∗ inverse (2 ∗ sqrt (Re x))
  | Eps2 (hyperdual-sqrt x) = Eps2 x ∗ inverse (2 ∗ sqrt (Re x))
  | Eps12 (hyperdual-sqrt x) = Eps12 x ∗ inverse (2 ∗ sqrt (Re x)) + Eps1 x ∗ Eps2 x ∗ −
inverse (4 ∗ sqrt (Re x) ^ 3)
```

Then we prove this to be the hyperdual extension of the real square root:

```
lemma hyperdual-ext-sqrt:
  assumes x > 0
  shows hyperdual-ext sqrt (λx. inverse (2 ∗ sqrt x)) (λx. − inverse (4 ∗ (sqrt x)^3)) UNIV x
hyperdual-sqrt
proof
  show (sqrt has-field-derivative inverse (2 ∗ sqrt x)) (at x)
    using DERIV-real-sqrt assms by auto
  then have ((λx. 2 ∗ sqrt x) has-field-derivative 2 ∗ inverse (2 ∗ sqrt x)) (at x)
    using DERIV-cmult by blast
  then have ((λx. 2 ∗ sqrt x) has-field-derivative inverse (sqrt x)) (at x)
    using inverse-mult-distrib by simp
  moreover have − (inverse (sqrt x) ∗ inverse (4 ∗ (sqrt x) ^ 2)) = − inverse (4 ∗ (sqrt x)^3)
    using inverse-mult-distrib mult.commute mult.assoc
    by (simp add: power2-eq-square power3-eq-cube)
  ultimately show ((λx. inverse (2 ∗ sqrt x)) has-field-derivative − inverse (4 ∗ (sqrt x)^3))
(at x)
    using DERIV-inverse-fun[of λx. 2 ∗ sqrt x inverse (sqrt x) x UNIV] assms by simp
  fix x :: hyperdual
  show Re (hyperdual-sqrt x) = sqrt (Re x)
    by simp
  show Eps1 (hyperdual-sqrt x) = Eps1 x ∗ inverse (2 ∗ sqrt (Re x))
    by simp
  show Eps2 (hyperdual-sqrt x) = Eps2 x ∗ inverse (2 ∗ sqrt (Re x))
    by simp
  show Eps12 (hyperdual-sqrt x) = Eps12 x ∗ inverse (2 ∗ sqrt (Re x)) + Eps1 x ∗ Eps2 x ∗ −
inverse (4 ∗ sqrt (Re x) ^ 3)
```

```
  by simp
qed
```

6.17.11 Multiplicative Inverse

We prove hyperdual inverse to be the hyperdual extension of the real inverse:

```
lemma hyperdual-ext-inverse:
 assumes x ≠ 0
 shows hyperdual-ext inverse (λx. − inverse (x^2)) (λx. 2 ∗ inverse (x^3)) UNIV x inverse
proof
 show (inverse has-field-derivative − inverse (x^2)) (at x)
  using assms DERIV-inverse[of x UNIV]
  by (simp add: power2-eq-square)
 have ((λx. inverse (x^2)) has-real-derivative − (2 ∗ x ∗ inverse ((x ^ 2) ^ 2))) (at x)
  using assms
      DERIV-inverse-fun[of λx. x^2 2∗x x UNIV]
      DERIV-pow[of 2 x UNIV]
      zero-eq-power2[of x] diff-Suc-Suc[of Suc 0 0] power-Suc0-right[of x]
  by simp
 then have ((λx. − inverse (x^2)) has-real-derivative 2 ∗ x ∗ inverse ((x ^ 2) ^ 2)) (at x)
  using Deriv.field-differentiable-minus by force
 moreover have 2 ∗ x ∗ inverse ((x ^ 2) ^ 2) = 2 ∗ inverse (x ^ 3)
  using assms power2-eq-square[of x^2] power2-eq-square[of x] power3-eq-cube[of x]
      times-divide-eq-right[of x 1 x ∗ x ∗ x ∗ x] inverse-eq-divide[of x ∗ x ∗ x ∗ x]
      add.inverse-inverse
  by (simp add: divide-inverse)
 ultimately show ((λx. − inverse (x^2)) has-real-derivative 2 ∗ inverse (x ^ 3)) (at x)
  by metis
 fix x :: hyperdual
 show Re (inverse x) = inverse (Re x)
  by (simp add: inverse-eq-divide)
 show Eps1 (inverse x) = Eps1 x ∗ − inverse ((Re x)^2)
  by (simp add: inverse-eq-divide)
 show Eps2 (inverse x) = Eps2 x ∗ − inverse ((Re x)^2)
  by (simp add: inverse-eq-divide)
 show Eps12 (inverse x) = Eps12 x ∗ − inverse ((Re x)^2) + Eps1 x ∗ Eps2 x ∗ (2 ∗ inverse
(Re x ^ 3))
  by (simp add: inverse-eq-divide)
qed
```

6.17.12 Multiplication

We prove multiplication of hyperdual extension to be the hyperdual extension of real multiplication:

```
lemma hyperdual-ext-times:
 assumes hyperdual-ext f f′ f″ S a fE
   and hyperdual-ext g g′ g″ S a gE
```

shows *hyperdual-ext* $(\lambda x.\ f\ x * g\ x)$ $(\lambda x.\ f'\ x * g\ x + f\ x * g'\ x)$ $(\lambda x.\ f''\ x * g\ x + f'\ x * g'\ x + f'\ x * g'\ x + f\ x * g''\ x)$ *S a* $(\lambda x.\ fE\ x * gE\ x)$
proof
show $((\lambda x.\ f\ x * g\ x)$ *has-field-derivative* $f'\ a * g\ a + f\ a * g'\ a)$ *(at a within S)*
using *assms has-snd-field-derivative.deriv-f add.commute*
DERIV-mult'[of f f' a a S g g' a]
by *(metis (full-types) hyperdual-ext-def)*
show $((\lambda x.\ f'\ x * g\ x + f\ x * g'\ x)$ *has-real-derivative* $f''\ a * g\ a + f'\ a * g'\ a + f'\ a * g'\ a + f\ a * g''\ a)$ *(at a within S)*
using *assms has-snd-field-derivative.deriv-f hyperdual-ext-def*
DERIV-add[of $\lambda x.\ f'\ x * g\ x\ f''\ a * g\ a + f'\ a * g'\ a\ a\ S\ \lambda x.\ f\ x * g'\ x\ f'\ a * g'\ a + f\ a * g''\ a$*]*
DERIV-mult'[of f' f'' a a S g g' a]
DERIV-mult'[of f f' a a S g' g'' a]
add.assoc
has-snd-field-derivative-def
by *smt*
fix *x :: hyperdual*
show *Re (fE x * gE x) = f (Re x) * g (Re x)*
using *hyperdual-ext.re-g assms*
by *simp*
show *Eps1 (fE x * gE x) = Eps1 x * (f' (Re x) * g (Re x) + f (Re x) * g' (Re x))*
using *add-mult-distrib mult.commute add.commute*
hyperdual-ext.eps1-g[of f f' f'' S a fE x]
hyperdual-ext.re-g[of f f' f'' S a fE x]
hyperdual-ext.eps1-g[of g g' g'' S a gE x]
hyperdual-ext.re-g[of g g' g'' S a gE x]
assms
by *(simp add: algebra-simps)*
show *Eps2 (fE x * gE x) = Eps2 x * (f' (Re x) * g (Re x) + f (Re x) * g' (Re x))*
using *add-mult-distrib mult.commute add.commute*
hyperdual-ext.eps2-g[of f f' f'' S a fE x]
hyperdual-ext.re-g[of f f' f'' S a fE x]
hyperdual-ext.eps2-g[of g g' g'' S a gE x]
hyperdual-ext.re-g[of g g' g'' S a gE x]
assms
by *(simp add: algebra-simps)*
show *Eps12 (fE x * gE x) = Eps12 x * (f' (Re x) * g (Re x) + f (Re x) * g' (Re x)) +*
*Eps1 x * Eps2 x * (f'' (Re x) * g (Re x) + f' (Re x) * g' (Re x) + f' (Re x) * g' (Re x) + f (Re x) * g'' (Re x))*
using *hyperdual-ext.eps12-g[of f f' f'' S a fE x]*
hyperdual-ext.eps2-g[of f f' f'' S a fE x]
hyperdual-ext.eps1-g[of f f' f'' S a fE x]
hyperdual-ext.re-g[of f f' f'' S a fE x]
hyperdual-ext.eps12-g[of g g' g'' S a gE x]
hyperdual-ext.eps2-g[of g g' g'' S a gE x]
hyperdual-ext.eps1-g[of g g' g'' S a gE x]
hyperdual-ext.re-g[of g g' g'' S a gE x]
assms

```
  by (simp add: algebra-simps)
qed
```

6.17.13 Natural Power

We prove that raising a hyperdual number to the power of some natural number is the hyperdual extension of the same operation on real numbers. First we establish this when the natural number is zero or one:

```
lemma hyperdual-ext-power0:
 shows hyperdual-ext (λx. x^0) (λx. 0) (λx. 0) UNIV a (λx. x^0)
using hyperdual-ext-const[of 1 a]
  by (simp add: one-hyperdual.code)

lemma hyperdual-ext-power1:
 shows hyperdual-ext (λx. x^1) (λx. 1) (λx. 0) UNIV a (λx. x^1)
using hyperdual-ext-identity
 by simp
```

Next, we prove the remaining cases (i.e. exponent being two or greater) by induction and composition with the multiplication extension:

```
lemma hyperdual-ext-power-ge2:
 fixes n :: nat
  and f f' f'' :: real ⇒ real
 shows hyperdual-ext (λx. x^(n+2)) (λx. (n+2) ∗ x^(n+1)) (λx. (n+2) ∗ (n+1) ∗ x^n) UNIV
a (λx. x^(n+2))
proof (induct n)
 case 0
 show ?case
  using hyperdual-ext-identity
      hyperdual-ext-times[of λx. x λx. 1  λx. 0 UNIV a λx. x λx. x λx. 1  λx. 0 λx. x]
  by (simp add: algebra-simps)
next
 case (Suc n)
 then show ?case
  using hyperdual-ext-identity
      hyperdual-ext-times[of λx. x λx. 1  λx. 0 UNIV a λx. x λx. x^(n+2) λx. (n+2) ∗ x^(n+1)
        λx. (n+2) ∗ (n+1) ∗ x^n λx. x ^ (n+2)]
  by (simp add: algebra-simps)
qed
```

Last, we prove the particular case when the exponent is equal to three:

```
lemma hyperdual-ext-cube:
 hyperdual-ext (λx. x^3) (λx. 3∗x^2) (λx. 6∗x) UNIV a (λx. x^3)
proof –
 have (λx::real. x ^ (1 + 2)) = (λx. x ^ 3)
  by (metis one-plus-numeral semiring-norm(3))
 moreover have (λx::hyperdual.  x ^ (1 + 2)) = (λx. x ^ 3)
  by (metis one-plus-numeral semiring-norm(3))
 moreover have ⋀x :: real. real (1 + 2) ∗ x ^ (1 + 1) ∗ 1 = 3 ∗ x ^ 2
```

```
  proof –
    fix x :: real
    have (1 + 2) ∗ x ˆ (1 + 1) ∗ 1 = (1 + 2) ∗ x ˆ 2
      using mult.right-neutral[of (1 + 2) ∗ x ˆ (1 + 1)] nat-1-add-1
      by smt
    then show real (1 + 2) ∗ x ˆ (1 + 1) ∗ 1 = 3 ∗ x ˆ 2
      by simp
  qed
  ultimately show ?thesis
    using hyperdual-ext-power-ge2[of 1 a]
        hyperdual-ext-identity[of a]
    by (simp add: algebra-simps)
qed
```

6.17.14 Finite Polynomial

We prove hyperdual finite polynomials to be the hyperdual extension of real finite polynomials. First we establish this when the number of terms is zero or one:

```
lemma hyperdual-ext-polyn0:
  fixes coef :: nat ⇒ real
  shows hyperdual-ext (λx. (∑i<0. coef i ∗ xˆi)) (λx. (∑j<(0−1). coef (j+1) ∗ (j+1) ∗ xˆj))
        (λx. (∑k<(0−2). coef (k+2) ∗ (k+2) ∗ (k+1) ∗ xˆk)) UNIV a (λx. (∑i<0. coef i ∗R
xˆi))
proof –
  have (λx::hyperdual. (∑i<0. coef i ∗R xˆi)) = (λx. of-real 0)
  proof
    fix x::hyperdual
    show (∑i<0. coef i ∗R x ˆ i) = of-real 0
      by simp
  qed
  then show ?thesis
    using hyperdual-ext-const[of 0 a]
    by simp
qed

lemma hyperdual-ext-polyn1:
  fixes coef :: nat ⇒ real
  shows hyperdual-ext (λx. (∑i<1. coef i ∗ xˆi)) (λx. (∑j<(1−1). coef (j+1) ∗ (j+1) ∗ xˆj))
        (λx. (∑k<(1−2). coef (k+2) ∗ (k+2) ∗ (k+1) ∗ xˆk)) UNIV a (λx. (∑i<1. coef i ∗R
xˆi))
proof –
  have (λx::hyperdual. (∑i<1. coef i ∗R xˆi)) = (λx. of-real (coef 0))
  proof
    fix x::hyperdual
    show (∑i<1. coef i ∗R x ˆ i) = of-real (coef 0)
      by simp
  qed
  then show ?thesis
    using hyperdual-ext-const[of coef 0 a]
  by simp
qed
```

Next, we prove the remaining cases (i.e. two or more terms) by induction and composition with addition, scalar multiplication and natural power extensions:

```
lemma hyperdual-ext-polyn-ge2:
 fixes coef :: nat ⇒ real
  and n :: nat
 shows hyperdual-ext (λx. (∑i<(n+2). coef i ∗ xˆi)) (λx. (∑j<(n+1). coef (j+1) ∗ (j+1) ∗
xˆj))
       (λx. (∑k<n. coef (k+2) ∗ (k+2) ∗ (k+1) ∗ xˆk)) UNIV a (λx. (∑i<(n+2). coef i ∗R
xˆi))
proof (induct n)
 case 0
 have (λx::real. (∑i<0+2. coef i ∗ xˆi)) = (λx. coef 1 ∗ x + coef 0)
 proof
  fix x::real
  have (∑i<1+1. coef i ∗ x ˆ i) = coef 1 ∗ x + coef 0
   by simp
  then show (∑i<0+2. coef i ∗ x ˆ i) = coef 1 ∗ x + coef 0
   using nat-1-add-1
   by simp
 qed
 moreover have (λx::hyperdual. (∑i<0+2. coef i ∗R xˆi)) = (λx. coef 1 ∗R x + of-real (coef
0))
 proof
  fix x::hyperdual
  have (∑i<1+1. coef i ∗R x ˆ i) = coef 1 ∗R x + of-real (coef 0)
   by simp
  then show (∑i<0+2. coef i ∗R x ˆ i) = coef 1 ∗R x + of-real (coef 0)
   using nat-1-add-1
   by simp
 qed
 ultimately show ?case
  using hyperdual-ext-linear[of coef 1 coef 0 a]
  by simp
next
 case hyp: (Suc n)

 have hyperdual-ext (λx. coef (n + 2) ∗R x ˆ (n + 2)) (λx. coef (n + 2) ∗ ((n + 2) ∗ x ˆ (n +
1)))
        (λx. coef (n + 2) ∗ ((n + 2) ∗ (n + 1) ∗ x ˆ n) + 0 ∗ ((n + 2) ∗ x ˆ (n + 1)) ∗ ((n +
2) ∗ x ˆ (n + 1)))
        UNIV a (λx. coef (n + 2) ∗R x ˆ (n + 2))
  using hyperdual-ext-power-ge2[of n a]
     hyperdual-ext.compose[of λx. x ˆ (n + 2) λx. (n + 2) ∗ x ˆ (n + 1) λx. (n + 2) ∗ (n +
1) ∗ x ˆ n UNIV a λx. x ˆ (n + 2)
      (∗R) (coef (n + 2)) λx. coef (n + 2) λx. 0 (∗R) (coef (n + 2))]
     hyperdual-ext-scaleR[of coef (n + 2) a ˆ (n + 2)]
```

hyperdual-ext.subset
by *blast*
then have *hyperdual-ext* $(\lambda x.\ coef\ (n+2) *_R x\ \hat{}\ (n+2))$ $(\lambda x.\ coef\ (n+2) * (n+2) * x\ \hat{}\ (n+1))$
$(\lambda x.\ coef\ (n+2) * (n+2) * (n+1) * x\ \hat{}\ n)$ *UNIV a* $(\lambda x.\ coef\ (n+2) *_R x\ \hat{}\ (n+2))$
by (*simp add*: *algebra-simps*)
moreover have $(\lambda x.\ \sum i<Suc\ n+2.\ coef\ i * x\ \hat{}\ i) =$
$(\lambda x.\ (\sum i<n+2.\ coef\ i * x\ \hat{}\ i) + coef\ (n{+}2) *_R x\ \hat{}\ (n{+}2))$
by *simp*
moreover have $(\lambda x.\ \sum j<Suc\ n+1.\ coef\ (j+1) * real\ (j+1) * x\ \hat{}\ j) =$
$(\lambda x.\ (\sum j<n+1.\ coef\ (j+1) * real\ (j+1) * x\ \hat{}\ j) + coef\ (n+1+1) * real\ (n+1+1) * x\ \hat{}\ (n{+}1))$
by *simp*
moreover have $(\lambda x.\ \sum k<Suc\ n.\ coef\ (k+2) * real\ (k+2) * real\ (k+1) * x\ \hat{}\ k) =$
$(\lambda x.\ (\sum k<n.\ coef\ (k+2) * real\ (k+2) * real\ (k+1) * x\ \hat{}\ k) + coef\ (n+2) * real\ (n+2) * real\ (n+1) * x\ \hat{}\ n)$
by *simp*
moreover have $(\lambda x.\ \sum i<Suc\ n+2.\ coef\ i *_R x\ \hat{}\ i) =$
$(\lambda x.\ (\sum i<n+2.\ coef\ i *_R x\ \hat{}\ i) + coef\ (n{+}2) *_R x\ \hat{}\ (n{+}2))$
by *simp*
ultimately show *?case*
using *hyp*
hyperdual-ext-add[*of* $\lambda x.\ \sum i<n+2.\ coef\ i * x\ \hat{}\ i$ $\lambda x.\ \sum j<n+1.\ coef\ (j+1) * (j+1) * x\ \hat{}\ j$
$\lambda x.\ \sum k<n.\ coef\ (k+2) * (k+2) * (k+1) * x\ \hat{}\ k$ *UNIV a* $\lambda x.\ \sum i<n+2.\ coef\ i *_R x\ \hat{}\ i$
$\lambda x.\ coef\ (n+2) *_R x\ \hat{}\ (n+2)$ $\lambda x.\ coef\ (n+2) * (n+2) * x\ \hat{}\ (n+1)$
$\lambda x.\ coef\ (n+2) * (n+2) * (n+1) * x\ \hat{}\ n$ $\lambda x.\ coef\ (n+2) *_R x\ \hat{}\ (n+2)$]
by (*simp add*: *algebra-simps*)
qed