A constructive and formal proof of Lebesgue's Dominated Convergence Theorem in the interactive theorem prover Matita

CLAUDIO SACERDOTI COEN
and
ENRICO TASSI
Department of Computer Science, University of Bologna

We present a formalisation of a constructive proof of Lebesgue's Dominated Convergence Theorem given by Sacerdoti Coen and Zoli in [CSCZ]. The proof is done in the abstract setting of ordered uniformities, also introduced by the two authors as a simplification of Weber's lattice uniformities given in [Web91, Web93]. The proof is fully constructive, in the sense that it is done in Bishop's style and, under certain assumptions, it is also fully predicative. The formalisation is done in the Calculus of (Co)Inductive Constructions using the interactive theorem prover Matita [ASTZ07]. It exploits some peculiar features of Matita and an advanced technique to represent algebraic hierarchies previously introduced by the authors in [ST07]. Moreover, we introduce a new technique to cope with duality to halve the formalisation effort.

Both authors were supported by DAMA (Dimostrazione Assistita per la Matematica e l'Apprendimento), a strategic project of the University of Bologna.

Contents

1 Introduction **3**

2 Pen&paper proof: pitfalls and formalisation choices **4**
2.1 Ordered sets . . . 5
2.2 Uniform spaces . . . 9
2.3 Ordered uniform spaces . . . 10
2.4 Uniformities with property (σ) . . . 11
2.5 Exhaustive order uniformities . . . 12
2.6 Lebesgue's dominated convergence theorem . . . 12

3 Technical devices **13**
3.1 Manifesting coercion . . . 13
3.2 Reflected duality . . . 16
3.2.1 Partial solution . . . 16
3.2.2 The problem . . . 18
3.2.3 The solution . . . 18
3.2.4 A more intuitive but incorrect solution . . . 20
3.2.5 The solution at work . . . 21
3.2.6 Drawbacks . . . 23
3.3 The Russell language . . . 23

4 Formalising the proof **24**
4.1 Sets equipped with an order or an equivalence relation . . . 24
4.2 Dual definitions over sets . . . 26
4.3 Uniformities and ordered uniformities . . . 28
4.4 Order continuity, property (σ) and exhaustivity . . . 32
4.5 Lebesgue's dominated convergence theorems . . . 33
4.6 A model based on the discrete uniformity over $\mathbb{N}$ . . . 34

5 Conclusions **36**

1. INTRODUCTION

We present a formalisation of the proof of Lebesgue's Dominated Convergence Theorem in the interactive theorem prover Matita [ASTZ07]. The formalization has been added to the the standard library of Matita[1]. The formalization problems addressed in the present paper have been different from those tackled in the rest of the Matita library that mainly deals with number theory [?].

The theorem represents a real milestone in integration theory and probability theory, and it is the major justification for the introduction of Lebesgue's integral. Formalising probability theory is a pre-requisite for many probabilistic analyses of computer systems and thus an interesting test case in formalisation of abstract results with immediate applications.

The proof we follow is the one given by the first author and Enrico Zoli in [CSCZ]. We will recall here all definitions and statements of the proof, but we expect the reader to refer to that paper for the informal proofs, examples and the intuitive justifications for the definitions and lemmas.

The proof given in [CSCZ] has some peculiarities. First of all, it is a novel proof of Lebesgue's Dominated Convergence Theorem in the very abstract setting of *ordered uniformities* that we introduced. Actually, what we did from this point of view is just to relax the lattice structure assumptions from the proof given by Hans Weber in [Web91, Web93] in the context of uniform lattices. A formalisation of the proof is the ultimate proof that the novel proof is correct. Indeed, one minor and one major error were spotted in the informal proof during formalisation and one of them was already present in the proof by Weber.

The second peculiarity of the proof is that we made it fully constructive in the following sense. First of all, it has been done in Bishop's style. Thus the computationally non-informative notion of partial large order "less or equal" has been replaced with the informative notion of partial excess relation introduced by Von Plato [?] and throughly investigated by Baroni in his Ph.D. thesis [Bar04]

Moreover, we had to change the assumptions of the proof with locatedness hypotheses that are examples of sentences that are tautologies only classically, but not intuitionistically. Second, under the assumptions that all uniformity bases are represented as set-indexed families, the proof is also fully predicative. Finally, some of the proofs are given in a peculiar style, showing the computational content first and proving the statement by showing that the computational content, seen as a program, fulfils its specification.

Matita implements the Calculus of (Co)Inductive Constructions (CIC) that has both a predicative and an impredicative fragment. Moreover, via the Curry-Howard isomorphism, CIC is an intuitionistic higher order logic. Finally, as the Coq system [Soz06], it helps the user in developing proofs by giving as a proof skeleton the computational content of the proof. All these characteristics are exploited to fully capture the constructive content of the proof. However, it must be said that Lebesgue's theorem has no major useful constructive content in itself (whereas several lemmas have). Thus we did not even state the theorem in its converse form

[1] The interested reader can browse proof scripts online at http://matita.cs.unibo.it/library/dama/ or download the Matita LiveCD at http://matita.cs.unibo.it/FILES/matita-svnhead.iso

to make the content evident by means of the excess relation in place of its negated version[2].

The last peculiarity is that the informal proof was developed in the first place having in mind that we wanted to formalise it later on. Thus the authors were extremely precise in all details and consequently the De Bruijn factor of the formalisation [Wie00] has remained quite low. Nevertheless, they only tried to keep all details explicit, but they still exploited several mathematical constructions that are difficult to capture formally. In particular they exploited duality of the excess connective to halve the size of the proof and they assumed a way to represent mathematical structures and to have multiple inheritance between them. The major issue in the formalisation, and the major contribution of the paper, is exactly an explanation of how we were able to capture these constructions in Matita in a faithful way. We must immediately advise the reader that the solutions proposed can be applied, in theory, using any interactive theorem prover based on the same logic or a similar one. Nevertheless, in practice the type inference algorithm of the system must be chosen carefully and must implement some heuristics that are implemented in Matita only and that were described in previous papers like [ST07]. Thus, for example, we would be unable to reproduce our formal proof in the Coq system [Coq], which is the interactive theorem prover close to Matita and based on the same logic. Anyway Coq is able to type check the proof object once it has been generated by Matita. Our success in exploiting these heuristics and algorithms to obtain a natural formalisation is a major motivation for the techniques itself.

In Sect. 2 we briefly sketch the informal proof, spotting the major sources of difficulties in the formalisation. The division in subsections partially reflects the one given in the informal paper [CSCZ]. In Sect. 3 we present the technical devices we adopted to faithfully represent inside Matita the mathematical concepts involved in the proof inside Matita. Finally, in Sect. 4 we show all the formal definitions, the formal statements of the lemmas and we glance to some interesting formal proof snippets. The division in subsections parallels the one of Sect.2. In Sect. 5 we draw some conclusions.

2. PEN&PAPER PROOF: PITFALLS AND FORMALISATION CHOICES

We present now, in the yellow boxes, all the definitions and statements of the informal proof [CSCZ], highlighting in bold font those steps that are responsible for the main difficulties in the formalisation process. The reader should refer to [CSCZ] for motivations, details and explanations about the informal proof.

[2] Consequently, we did not try to explicitly extract the computational content from the proof. If we did, we would obtain an algorithm that, given an evidence of the fact that a sequence uniformly diverges from a limit point, it produces an evidence of the fact that it also order diverges from the same point. According to our knowledge, we did not find yet any concrete application of this computational content.

2.1 Ordered sets

Definition 2.1. Ordered set. *An ordered set $(C, \nleq)$ is a data type C together with a* ***propositional operation*** *$\nleq$ (called excess [Bar04]) such that the following properties hold:*

(*1*) *Co-reflexivity:* $\forall x : C. \neg(x \nleq x)$

(*2*) *Co-transitivity:* $\forall x, y, z : C. x \nleq y \Rightarrow x \nleq z \vee z \nleq y$

We call C a data type and not a set since we will ignore its equality. Correspondingly, we require $\nleq$ to be only a propositional operation, in Bishop's sense, and not a relation, since we are not interested in the preservation of any equivalence relation on C. Any ordered set will turn out to be a set with an excess relation when we will induce an equality on C starting from the excess propositional operation.

Bishop's style mathematics is always developed in an extensional setting built on top of an intensional one. This means that every set is equipped with its apartness relation, which induces an equality relation on the set. The apartness relation captures negation of equality in a positive and computationally relevant way, whereas the computational content of equality is squashed into the unit or the empty type. The book by Bishop and Bridges [BE85] is the reference guide for the practice of Bishop's style mathematics.

According to the informal text, we initially diverge from Bishop's style by ignoring any pre-defined apartness and equality relations over the carrier of the ordered set. Nevertheless, the excess relation will induce an apartness and an equality. Thus this is just a technical trick to avoid asking for compatibility between the predefined relations and the induced ones.

As in the case of equality and apartness, any semi-decidable (but not decidable) property that is defined in classical mathematics as the negation of some other undecidable property must be directly axiomatised in Bishop's style. In our case, the model of partially ordered set we have in mind is that of real functions and pointwise order. The latter is not decidable, whereas its "classical negation", here called excess, is semi-decidable. Indeed only a finite amount of information is required to show that there exists one point (or one open segment if we want to be point-free) such that one function is strictly above the other in that point.

In order to axiomatise the excess relation, we must decide how to represent a propositional operation. In intuitionistic logic, a propositional operation is a function taking two arguments in the carrier and returning a proposition that, according to the Curry-Howard isomorphism employed in Matita, is just a data type. In order to distinguish between real data types and those meant to be propositions, it is customary to put them in different universes. Universes are types for types that were initially introduced in type theory in order to avoid paradoxes and are reminiscent of Russel's ramified type theory. CIC inherits from Luo's Extended Calculus of Constructions [Luo89] two kind of universes: the impredicative universe **Prop** and a hierarchy of predicative universes $\mathbf{Type}_i$ whose set theoretical strength corresponds to a hierarchy of strongly inaccessible cardinals [Wer97]. For each propositional operation we want to represent we must choose the universe to put it in, reasonably among the impredicative universe **Prop** and the first predica-

tive universe $\mathbf{Type}_0$.

In particular, we must choose a universe both for the positive (and computationally relevant) excess relation and for its negation. In order to make the second choice, we observe that, according to the Brower-Heiting-Kolmogorov semantics for intuitionistic logic, the computational content of any proof of $\neg P$ is a procedure that, given an inhabitant of P, returns an inhabitant of the empty set. If we identify procedures with functions (ignoring their intensional aspects), we immediately notice that the content of a proof of $\neg P$ is a function whose codomain is empty. Such a function exists only when P is an empty type and, in such case, the function is the empty relation. As a procedure, it will never be called since nobody will be able to produce an input for it in the empty context. Thus we can identify all procedures that compute the empty function. The resulting semantics assigns to every negated formula either the empty set of procedures or a singleton set. Thus it is isomorphic to the classical semantics.

For this reason, and since the proof of the Fundamental Theorem of Algebra [CFGW04, FS03], it has become an habit to define negation of a proposition P as $P \Rightarrow \bot$ where the empty type $\bot$ is declared in the impredicative universe $\mathbf{Prop}$. As a consequence of the typing rule for products (see again [Luo89]) the negation of a proposition is automatically put in the same impredicative universe. Because of impredicativity in the first place, the set-theoretic semantics of all data types defined in $\mathbf{Prop}$ is equivalent to the classical semantics and thus bears no computational content. In particular all subterms whose type is in $\mathbf{Prop}$ are erased during code extraction [PM89]. In this way the user can achieve a fine tuned control on the computationally relevant parts of the proof, and he can exploit impredicativity when interested only in provability. Notice that this kind of fine control is absolutely necessary to extract reasonable and efficient programs from proofs (see [FS03] for a discussion).

Having decided to put all negated propositions in $\mathbf{Prop}$, and since we are interested in the computational content of all remaining propositions like excess and apartness, we have to put them in a predicative universe and, since the mathematical proof only involves sets and not classes, the smallest predicative universe, $\mathbf{Type}_0$ in ECC, is sufficient. However, all data types are already put in $\mathbf{Type}_0$ and we must record the difference between types that represent data types and types that represent propositions in order to render proofs and statements in the expected way. For instance, a dependent product instead as an universal quantification is usually displayed as $\forall x.P(x)$ in place of $\Pi x.P(x)$, which is reserved for dependent maps between data types. In order to solve this problem, in the current version of Matita we depart from the tradition of ECC (and Coq too) by introducing two parallel hierarchies of predicative universes $\mathbf{Type}_i$ for data types and $\mathbf{CProp}_i$ for computationally relevant propositions. Every level of one hierarchy is cumulatively contained in the next level of both hierarchies, even if $\mathbf{CProp}_i$ is not convertible with $\mathbf{Type}_i$. The trivial (non injective) map that identifies $\mathbf{CProp}_i$ with $\mathbf{Type}_i$ preserves the typing and conversion judgements.

Thus we put all propositional operations, and in particular the excess and apartness relations, in $\mathbf{CProp}_0$.

To summarise, the excess relation will be defined as a computationally relevant function taking two arguments in the carrier and returning a proposition in

$\mathbf{CProp}_0$. Every model will instantiate it with a function producing some evidence that witnesses the fact that the first argument exceeds the second. The negated relation $\leq$ will be a function taking two arguments and returning a proposition in **Prop**.

> As in the classical case, if $\nleq$ is an excess operation, the same holds for $\nleq^{-1}$. This allows to **omit dualized definitions** and statements in the sequel.

Duality is a meta-linguistic device usually employed to omit redundant lemmas and definition that can be derived by a mechanical transformation. Although this seems a field where interactive theorem prover should assist the users dualizing results for them, according to the authors knowledge no interactive theorem prover provides such facility. Moreover, no particular attention has been devoted to this problem in the literature. Having to double by hand every definition and every proof is not only a tedious task when done the first time, but imposes on the user an additional burden of work when his definition will be eventually reworked and proofs fixed. Avoiding pollution of the search space by keeping only non-redundant theorems is also an issue to be considered when dealing with duality.

O'Connor in [**?**] has exploited the non first class module system of Coq for duality. However he needs to state again the statements of every dualized lemma.

> Definition 2.2. Apartness, equality, less or equal. *Let* $(C, \nleq)$ *be an ordered set.*
>
> (1) $x \neq y$ *iff* $x \nleq y \vee y \nleq x$.
> (2) $x = y$ *iff* $\neg(x \neq y)$.
> (3) $x \leq y$ *iff* $\neg(x \nleq y)$.
>
> $(C, \neq)$ endowed with the equality relation induced by $\nleq$ is a **set in Bishop's terminology.** Moreover, the excess and less or equal propositional operations are **relations w.r.t. the equality** . From the co-reflexivity and co-transitivity properties of $\nleq$ it immediately follows reflexivity and transitivity of $\leq$ and $=$, and co-reflexivity and co-transitivity of $\neq$.

Sets obtained in mathematics by quotienting them over some equivalence relations are hard to formalise in intensional type theories like the one implemented in Matita. Traditionally, the problem is addressed by working with the original un-quotiented set and keeping the equivalence relation in place of the equality. However, rewriting of a term with an equivalent one in a given expression is no longer granted to work, unless the relation is a congruence with respect to all operators in the expression. Moreover, even in that case an explicit proof of the latter fact is required in order to proceed with the rewriting. In practice, the system must provide proper assistance, like the one presented in [Sac04] for Coq, in order to automatically provide such proofs.

Surprisingly, the equality relation induced by the excess relation does not play any role at all in our proof. Thus we can completely ignore this issue.

Finally, we note that the short piece of text under examination hides quite a number of very simple proofs that were not difficult to formalise, but that are mostly responsible for the doubling in the number of objects during the formalisation.

LEMMA 2.3. *Let* $(C, \nleq)$ *be an ordered set and* $a, b, a', b' \in C$ *such that* $a \nleq b$, $a \leq a'$, $b' \leq b$. *Then* $a' \nleq b'$.

DEFINITION 2.4. STRONG SUPREMUM. *Let* $(C, \nleq)$ *be an ordered set and* (a_i) *a sequence in* C. $a \in C$ *is a strong supremum of* (a_i) *if* $\forall i \in \mathbb{N}. a_i \leq a$ *and* $\forall b \in C. a \nleq b \Rightarrow \exists i \in \mathbb{N}. a_i \nleq b$.

We write $a_i \uparrow a$ when (a_i) is an increasing sequence, whose strong supremum is a.

LEMMA 2.5. *Let* $(C, \nleq)$ *be an ordered set and* (m_n) *a strictly increasing sequence of natural numbers. If* a *and* (a_n) *are in* C *and* $a_n \uparrow a$, *then* $a_{m_n} \uparrow a$.

Definition 2.4 and Lemma 2.5 are the first examples of definitions and properties that are implicitly dualized in the informal proof.

DEFINITION 2.6. ORDER CONVERGENCE. *Let* $(C, \nleq)$ *be an ordered set and* a *and* (a_i) *in* C. *We say that* (a_i) *order converges to* a *(written* $a_i \stackrel{o}{\to} a$*) iff there exist an increasing sequence* (l_i) *and a decreasing sequence* (u_i) *in* C *such that* $l_i \uparrow a$ *and* $u_i \downarrow a$ *and for all* $i \in \mathbb{N}$ *the strong infimum of* $(a_{i+n})_{n \in \mathbb{N}}$ *is* l_i *and the strong supremum is* u_i.

DEFINITION 2.7. SEGMENT. *Let* $(C, \nleq)$ *be an ordered set and* $a, b \in C$. *The segment* $[a, b]$ *is the set* $\{x | a \leq x \text{ and } x \leq b\}$.

Clearly, **the restriction of an ordered set to a segment is itself canonically endowed with an order structure**.

The representation of sub-sets in type theory is another source of technical problems. We adopt the usual solution of representing a sub-set — in this case the segment — with a Σ-type that packs together elements of the super-set and proofs that the element also belong to the sub-set. Coercions are used to automatically cast elements of the sub-set to elements of the super-set and vice-versa. In the latter case the insertion of the coercion opens a proof obligation that the user must fill, in the style of PVS predicate subtyping [SO99].

What is more problematic to formalise is the latter statement. Indeed, inducing an order relation on the Σ-type is simple but, a priori, there is no reason why $x \leq_C y$ should be convertible to $x \leq_{[a,b]} y$ for x, y elements of the sub-set. As we will see, this will not be the case in our formalisation and the two formulae will only be logically equivalent, i.e. $x \leq_C y \iff x \leq_{[a,b]} y$. Since the two order relations will be given the same unqualified notation "$\leq$", this may result in a source of confusion during formalisation. Declaring a coercion from each formula to the other equivalent one was sufficient to avoid further user intervention.

LEMMA 2.8. *Let $(C, \not\leq)$ be an ordered set, $l, u \in C$ and (a_i) and a in $C \cap [l, u]$. If $a_i \uparrow a$ in C, then $a_i \uparrow a$ in $C \cap [l, u]$.*

DEFINITION 2.9. CONVEX SET. *Let $(C, \not\leq)$ be an ordered set. We say that a set $U \subseteq C \times C$ is convex iff $\forall (a, b) \in U.a \leq b \Rightarrow [a, b]^2 \subseteq U$.*

DEFINITION 2.10. UPPER LOCATEDNESS. *Let $(C, \not\leq)$ be an ordered set. The sequence (a_i) is upper located [Bar04] if $\forall x, y \in C.y \not\leq x \Rightarrow (\exists i \in \mathbb{N}.a_i \not\leq x) \vee (\exists b \in C.y \not\leq b \wedge \forall i \in \mathbb{N}.a_i \leq b)$.*

LEMMA 2.11. *Let $(C, \not\leq)$ be an ordered set and (a_i) and a in C such that $a_i \uparrow a$. Then (a_i) is upper located in C.*

2.2 Uniform spaces

DEFINITION 2.12. UNIFORM SPACE. *A uniform space $(C, \neq, \Phi)$ is a set $(C, \neq)$ equipped with a* ***family*** *Φ (called uniformity base) of* ***sub-sets*** *of the cartesian product $C \times C$ (called basic entourages) with the following properties:*

(1) $\forall U \in \Phi.\{(x, y) | \neg(x \neq y) \in C\} \subseteq U$
(2) $\forall U, V \in \Phi.\exists W \in \Phi.W \subseteq U \cap V$
(3) $\forall U \in \Phi.\exists V \in \Phi.V \circ V \subseteq U$
(4) $\forall U \in \Phi.U = U^{-1}$

The usual definition of uniform spaces is in terms of (not necessarily basic) entourages. An entourage is any superset of some basic entourage. We do not follow this approach since the family of all entourages is necessarily a proper class in an impredicative setting. Indeed, the class Φ of all entourages is closed w.r.t. the following property: $\forall U \in \Phi.\forall V \in 2^{C \times C}.U \subseteq V \Rightarrow V \in \Phi$ where the quantification of V is on the power set of the $C \times C$. On the contrary, to work in a predicative setting it is sufficient to assume that the class Φ of all basic entourages is set indexed and that all quantifications in the definition of Φ are on the set of indexes. In what follows, we will tacitly assume this.

The formal definition of a uniform space is the most challenging one in the whole development, since it involves at once two difficult concepts: that of *family* and that of *sub-set*. Both of them can be represented in type theory in at least two alternative and predicatively non equivalent ways: as propositional predicates over some set/family or as indexed enumerations of the elements. The different lattice theoretic properties of these two representations are studied, for instance, in Sect. 2 of [HH08]. We will discuss the consequences of the different choices in Section 4.3, where we will present the unorthodox choice made in the formalisation.

Definition 2.13. Cauchy sequence. *A sequence* (a_i) *of points of a uniform space* $(C, \neq, \Phi)$ *is Cauchy iff* $\forall U \in \Phi. \exists n \in \mathbb{N}. \forall i, j \geq n.(a_i, a_j) \in U$.

Definition 2.14. Uniform convergence. *A sequence* (a_i) *of points of a uniform space* $(C, \neq, \Phi)$ *converges to a point* $a \in C$ *(written* $a_i \rightarrow a$*) if* $\forall U \in \Phi. \exists n \in \mathbb{N}. \forall i \geq n.(a, a_i) \in U$.

Lemma 2.15. *Let* $(C, \neq, \Phi)$ *be a uniform space and* (a_i) *and* a *in* C *such that* $a_i \rightarrow a$. *Then* (a_i) *is Cauchy.*

Definition 2.16. Restricted uniformity. *Let* $(C, \neq, \Phi)$ *be a uniform space and* X *a sub-set of* C. *We call the family* $\{U \cap X \times X | U \in \Phi\}$ *the restricted uniformity base on* X.

The definition is well posed, as the properties listed in Definition 2.12 hold.

Fact 2.17. *Let* $(C, \neq \Phi)$ *be a uniform space,* X *a sub-set of* C *and* (a_i) *in* X. *If* (a_i) *is Cauchy in* X, *then* (a_i) *is Cauchy in* C.

The proof that the definition of restricted uniformity is well posed is simple, but hides another large number of lemmas.

2.3 Ordered uniform spaces

Definition 2.18. Ordered uniform space. *A triple* $(C, \nleq, \Phi)$ *is an ordered uniform space iff* $(C, \nleq)$ *is an ordered set,* (C, Φ) *is a uniform space and every basic entourage* $U \in \Phi$ *is convex.*

This is the classical example of the definition of an algebraic structure by composition of two pre-existing structures and addition of axioms that relate the two of them. In particular, the definition implicitly says that the uniform space and the ordered set are not completely generic, since they must be defined on the same carrier, nor completely instantiated, since $\nleq$ and Φ are still abstract. We will be able to formalise the definition using the technique presented in [ST07] by packing together a totally abstract ordered set and a uniform space with one manifest field (the carrier, set to the carrier of the ordered set).

Moreover, we note a mismatch in the informal definition between the two carriers C. The C of the ordered uniform space is a Bishop set, i.e. a data type together with an equivalence relation. In the following we equip every Bishop set with a tight apartness relation omitting the equivalence relation that we induce from it. We adopted this solution since we are interested in the computational content carried only by the former relation on ideal objects.

The C of the ordered set is just a data type and it becomes a Bishop set only by endowing it with the apartness induced by the excess relation. Thus, to be precise, we will have to set the carrier of the uniform space, that must be a Bishop set, to the Bishop set induced on C by the excess relation.

Lemma 2.19. *Let* $(C, \nleq, \Phi)$ *be an ordered uniform space and* $l, u \in C$. *Let* (a_i) *and* a *in* $C \cap [l, u]$. *If* $a_i \rightarrow a$ *in* C *then* $a_i \rightarrow a$ *in* $C \cap [l, u]$.

Theorem 2.20. Sandwich. *Let $(C, \not\leq, \Phi)$ be an ordered uniform space. Let $l \in C$ and $(a_i), (x_i), (b_i)$ be sequences in C such that $\forall i \in \mathbb{N}. a_i \leq x_i \leq b_i$ and $a_i \to l$ and $b_i \to l$. Then $x_i \to l$.*

Definition 2.21. Order continuity. *Let the triple $(C, \not\leq, \Phi)$ be an ordered uniform space. We say that the uniformity is order continuous iff for all (a_i) and a in C, $a_i \uparrow a \Rightarrow a_i \to a$ and $a_i \downarrow a \Rightarrow a_i \to a$.*

2.4 Uniformities with property (σ)

Definition 2.22. Property (σ). *Let $(C, \not\leq, \Phi)$ be an ordered uniform space. The uniformity satisfies property (σ) iff $\forall U \in \Phi. \exists (U_n). \forall (a_n). \forall a. a_n \uparrow a \Rightarrow (\forall n. \forall i, j \geq n. (a_i, a_j) \in U_n) \Rightarrow (a_1, a) \in U$.*

The definition of property (σ) only deals with increasing sequences. However, in the rest of the proof it is implicitly assumed that a uniformity that has property (σ) also has the dual property on decreasing sequences. A posteriori, this is an error in the definition of property (σ) itself. In the formalisation we decided to fix it by assuming $a_n \uparrow a \vee a_n \downarrow a$ in the definition of (σ). Another possibility would have been to split property (σ) into two properties and requiring both of them where necessary. In Section 4.4 we will discuss the advantages and disadvantages of both choices.

Lemma 2.23. *Let $(C, \not\leq, \Phi)$ be an ordered uniform space with property (σ). Suppose (a_i), a in C such that $a_i \uparrow a$. If (a_i) is Cauchy, then $a_i \to a$.*

Proof. Fix $U \in \Phi$. We need to prove $\exists m \in \mathbb{N}. \forall i \geq m. (a_i, a) \in U$. Let (U_n) as in Definition 2.22 and let (m_n) in $\mathbb{N}$ be the sequence defined by recursion as follows. For the base case, since (a_i) is Cauchy, there exists $k \in \mathbb{N}$ such that $\forall j, j' \geq k. (a_j, a_{j'}) \in U_0$; take k for m_0. For the inductive case, since (a_i) is Cauchy, there exists $k \in \mathbb{N}$ such that $\forall j, j' \geq k. (a_j, a_{j'}) \in U_{n+1}$. Take $\max\{k, m_n + 1\}$ for m_{n+1}. The sequence (m_n) is strictly increasing by construction. Thus $a_{m_n} \uparrow a$ by Lemma 2.5. Thus, by property (σ), $(a_{m_1}, a) \in U$. Take m_1 and let $i \geq m_1$. Since (a_{m_n}) is increasing, $a_i \in [a_{m_1}, a]$. Since U is convex and $(a_{m_1}, a) \in U$, also $(a_i, a) \in U$. □

This is the first example of a proof that explicitly gives the computational content first and then the proof that it satisfies the desired property. The same proof style was already used in [Web93].

In particular, the text exhibits a program $m : \mathbf{N} \to \mathbf{N}$ defined by structural recursion on natural numbers. The definition is presented in mathematical style as the explicit construction of a sequence (m_n). Later on, it is proved that (m_n) is strictly increasing and then (m_n) is used together with the definition of (σ) to prove $(a_{m_1}, a) \in U$. However, the latter derivation requires a proof of $\forall n, \forall j, j' > m_n, (a_j, a'_j) \in U_n$ that is silently assumed in the text, but that we will need to prove explicitly. The missing proof is by construction of (m_n), but it was not completely trivial to formalise. Thus we consider the omission a fault in the informal proof, since the authors claimed that all details were made explicit in view of a future formalization. This omission was already present in [Web93].

Another important observation is that the proof implicitly uses the axiom of countable dependent choice to extract the sequence (U_n) from U by property (σ). We know (see for instance [ML06]) that the axiom of extensional choice does not hold in an extensional setting built on top of an intensional one. Thus the construction of (m_n) from U yields only an operation and not a function. Nevertheless, this does not give us any problem since we only use (m_n) to build m_1 that is only used to prove $(a_i, a) \in U$.

Readers that for philosophical reasons prefer to avoid any axiom of choice at all can just, instead of assuming (σ) in the rest of the paper, assume a function f mapping an entourage to a sequence of entourages such that

$$\forall U \in \Phi . \forall (a_n). \forall a. a_n \uparrow a \Rightarrow (\forall n. \forall i, j \geq n. (a_i, a_j) \in (f\ U)_n) \Rightarrow (a_1, a) \in U$$

All the theorems in this paper will continue to hold with this new definition, but we have not checked this formally.

2.5 Exhaustive order uniformities

DEFINITION 2.24. EXHAUSTIVITY. *The uniformity Φ of the ordered uniform space $(C, \nleq, \Phi)$ is exhaustive if any increasing sequence that is upper located, and any decreasing sequence that is lower located, is Cauchy.*

LEMMA 2.25. *Let $(C, \nleq, \Phi)$ be an ordered uniform space with property (σ). Let $l, u \in C$ such that the uniformity induced on $C \cap [l, u]$ is exhaustive. If (a_i) is a sequence in $C \cap [l, u]$ and a a point in C such that $a_i \uparrow_C a$, then $a \in [l, u]$ and $a_i \to a$ in $C \cap [l, u]$.*

Lemma 2.25 is the only dualizable lemma on ordered uniformities, while all the previous dualizable ones were on ordered sets only. In the formalisation we have not set up the dualization machinery for ordered uniformities since one single application of it did not pay back the extra effort.

2.6 Lebesgue's dominated convergence theorem

The two final results are proofs of Lebesgue's Dominated convergence theorem under incomparable assumptions.

THEOREM LEBESGUE UNDER (σ) AND EXHAUSTIVITY ASSUMPTIONS. *Let $(C, \nleq, \Phi)$ be an ordered uniform space with property (σ) and such that, for all $l, u \in C$, the uniformity induced on $C \cap [l, u]$ is exhaustive. Let (a_i) be a sequence in C and $l, u \in C$ such that $\forall i \in \mathbb{N}. a_i \in C \cap [l, u]$. Finally, let a be a point in C such that $a_i \xrightarrow{o} a$ in C. Then $a \in C \cap [l, u]$ and $a_i \to a$ in $C \cap [l, u]$.*

THEOREM LEBESGUE UNDER ORDER CONTINUITY ASSUMPTION. *Let $(C, \nleq, \Phi)$ be an ordered uniform space such that for all $l, u \in C$ the uniformity induced on $C \cap [l, u]$ is order continuous. Let (a_i) be a sequence in C and $l, u \in C$ such that $\forall i \in \mathbb{N}. a_i \in C \cap [l, u]$. Finally, let a be a point in C such that $a_i \xrightarrow{o} a$ in C. Then $a \in C \cap [l, u]$ and $a_i \to a$ in $C \cap [l, u]$.*

We summarise the main difficulties expected in the formalisation as follows:

—Duality helps the authors of the informal proof in keeping the proof compact. The interactive theorem prover needs to provide some kind of support to duality to avoid duplication by hand of definition and lemmas.

—Multiple inheritance is used by the authors when defining the ordered uniform space structure. The CIC type system has no built-in sub-typing relation to mimic inheritance and no first-order module system with manifest type equations (in the style of ML) to combine together different structures forcing some of their fields to be equal.

—Some of the lemmas require the proof that some explicit construction satisfy some properties. An adequate support by the system is necessary to help the user in this task which is technically closer to proving the correctness of programs than to standard mathematical proofs.

3. TECHNICAL DEVICES

Here we introduce the techniques we adopted to overcome the difficulties spotted in Section 2.

3.1 Manifesting coercion

Manifesting coercions are a device introduced by the authors in [ST07] to formalise algebraic structures in an intensional type theory like CIC in a way that looks quite close to what mathematicians do in regular algebra textbooks. It requires no notions of sub-typing to be part of the logic and no module system allowing to combine two structures forcing some some of their fields to be the same. On the contrary it requires the Interactive Theorem Prover (ITP from now on) to implement the coercion mechanism [Luo99, Sai97] and some peculiar unification heuristics [ST07].

Usually a hierarchy of algebraic structures is built starting from its basic, components that are then extended and combined to build more complex and rich structures. Let us take for example the definition of the ring structure given by Wikipedia[3]:

Definition Ring. *A ring is a set R equipped with two binary operations $+ : R \times R \to R$ and $* : R \times R \to R$ (where $\times$ denotes the Cartesian product), called addition and multiplication, such that:*

— *$(R, +)$ is an abelian group with identity element 0*
— *$(R, *)$ is a monoid with identity element 1*
— *Multiplication distributes over addition*

We recall that a monoid is a structure equipped with an internal associative operation and a unit element, while the group adds to that the existence of inverse elements.

Assuming we have already defined what monoid and abelian group are, we want a simple mechanism to combine them together, forcing their carrier set to be the same and adding the additional distributivity property among their operations.

[3] As of November 28, 2008 at http://en.wikipedia.org/wiki/Ring_(mathematics)

Forcing the carrier to be the same is necessary for stating the property of distributivity, since the expression $a * (b + c)$ requires the output type of the + operation to be convertible with the input type of the $*$ operation.

As discussed in [ST07] the literature offers many not so satisfactory solutions, like parametrising any field that needs to be constrained out of the algebraic structures or using induction recursion to break the group and monoid structures down into pieces and building on the fly a brand new ring structure. For a detailed discussion of these alternative approaches the reader should refer to [ST07, Pol02].

The solution we propose in [ST07] requires no induction recursion and uses the coercion mechanism [Luo99] to mimic subtyping. Our approach allows to almost completely reuse the previously given definitions of group, with no necessity of abstracting it over the carrier. It requires the definition of an intermediate structure "ring_" that simply puts together a group and a monoid and forces the carrier of the latter to be provably equal to the carrier of the former structure.

```
record ring_ : Type := {
  r_group :> group;
  r_monoid_ : monoid;
  r_with_  : g_carr group = m_carr monoid_
}.
```

Here "g_carr" and "m_carr" are projections (also declared as coercions) extracting the carrier from the monoid and the group structure. We also state that "r_group" is a coercion with the syntax ":>". Note that he "r_monoid_" projection is not declared as a coercion. We just recall that the coercion mechanism consists in tagging some terms (like the projections above) that the system is allowed to insert to fix user provided terms. For example, passing an argument that inhabits the "ring_" type to a function that expects a "group" is not allowed in CIC, the "g_carr" coercion is automatically inserted by the type inference subsystem of the ITP to explicitly cast the argument to the expected type.

We explicitly define the "r_monoid" coercion, that given a "ring_" structure extracts a "monoid" structure, taking the fields of the "r_monoid_" component and building a new monoid whose carrier is by definition the carrier of the "r_group" field and its other components are the ones of the "r_monoid_" field whose type has been properly rewritten to the right one thanks to "r_with_". If we denote with "[[_]]" the dependent rewriting principle associated to the "r_with_" equation, the "r_monoid" coercion can be defined as follows:

```
definition r_monoid := λ r.
  mk_monoid (g_carr (r_group r)) [[m_times (r_monoid_ r)]]
    [[ m_unit (r_monoid_ r)]]  [[ m_unit_prop (r_monoid_ r)]]
    [[ m_assoc (r_monoid_ r)]]
```

Where "m_times", "m_unit", "m_unit_prop" and "m_assoc" are projections extracting the monoid operation, the unit element, the property of the unit element and the associativity property of the monoid operation. For details on "[[_]]" the reader should refer to [ST07].

After attaching infix notation to "m_times" and "g_plus" (the additive operation of the group structure) we can define the ring structure.

```
record ring : Type := {
  r_stuff  :> ring_;
  r_distr  : ∀a,b,c. a ∗ (b + c) = a ∗ b + a ∗ c
}.
```

The “r_distr” field types correctly since the “m_times” projection will not “extract” from “r_stuff” the real “r_monoid_” field (that was not declared as a coercion) but its manifest version “r_monoid” that takes in input and gives in output objects of type “(g_carr (r_group r_stuff))” by construction.

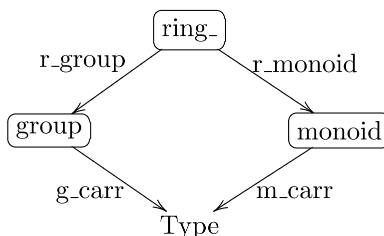
To make an interactive theorem prover like Matita accept that definition, some smart type inference algorithm needs to be implemented. What is hidden by the “a ∗ (b + c)” notation is

```
m_times ?1 a (g_plus ?2 b c)
```

where “$?_1$” and “$?_2$” are implicit arguments the system has to infer. The expected type for the second argument of the multiplication is “m_carr $?_1$” (since “m_times” has type “∀m:monoid.m_carr m → m_carr m → m_carr m”), while its inferred type is “g_carr $?_2$”.

The unification problem “m_carr $?_1$” versus “g_carr $?_2$” usually leads to a failure since the two head terms are non convertible constants. Anyway there is a solution, since what the unification problem is really asking for is the smallest structure (if any) containing a monoid and a group whose carrier is the same.

If we draw the graph of coercions declared so far we obtain the following picture:



We thus extended our unification algorithm (used by the type inference one) making it aware of the coercions graph and allowing it to look for the pullback (in categorical terms) of the morphisms “m_carr” and “g_carr”. It is thus able to give the following solution to the previous unification problem:

```
?1 := r_monoid ?3
?2 := r_group ?3
```

For some “$?_3$” that in our case will be later instantiated with “r_stuff”. Note that, even before instantiating “$?_3$”, the terms “m_carr (r_monoid $?_3$)” and “g_carr (r_group $?_3$)” are convertible (by construction of “r_monoid”, they both reduce to “g_carr (r_group $?_3$)”). This amounts to the coherence property of the coercion graph (i.e. the two paths in the graph above commute) defined in [Luo99].

This technique will be adopted to put together the ordered set structure and the uniform space structure.

3.2 Reflected duality

Duality is usually understood as a meta-linguistic feature, and is thus reasonable to expect that it is a duty of the ITP to provide such facility. Although, to the authors knowledge, no tool provides proper support for duality.

One of the main reasons for using an ITP based on a powerful higher order language like CIC is its computational behaviour. Since the type system identifies types up to reduction, many tasks usually relegated to the user or to external tools can be reflected into the system and performed by means of reduction.

The main sources of duality in our development are the excess and the order relation. We consider here the latter since it is more familiar. The usual way $\leq$ and $\geq$ are formalised in ITP grants some simple duality, but is not general enough to fulfil our needs. What is usually done in ITPs is to define $\leq$ as a primitive notion on a fixed set (like $\mathbb{N}$), and $\geq$ as a dumb constant on top of $\leq$ that simply swaps its arguments. Thus $a \geq b$ reduces to $b \leq a$, as one would expect, and a proof of $b \leq a$ is also a proof of $a \geq b$. A great limitation of that approach is that it does not scale to more complex statements. For example the proof of the transitivity property for $\leq$ and $\geq$ cannot be the same. If we have an assumption "H" that

```
∀ a,b,c.  a ≤ b → b ≤ c → a ≤ c
```

H is not also a proof of

```
∀ a,b,c.  a ≥ b → b ≥ c → a ≥ c
```

since the two assumptions have to be swapped.

Our solution to that problem will be presented in two steps, first we will describe a partial solution to introduce the overall technique, then we will show why this solution is only partial and finally we will refine it to solve the encountered issue.

3.2.1 *Partial solution.* We say that a structure is composed of two halves, intuitively corresponding to the primitive and the dual structure. We associate the "$\lll$" infix notation to the "half_pred" field.

```
record half : Type :={
  half_carr :> Type;
  half_pred :   half_carr  → half_carr  → CProp;
  half_property : ∀ x,y,z.x ⋘ y → y ⋘ z → x ⋘ z
}.
```

We then define the dualizer as follows, note that the arguments of "$\lll$" are swapped in the dual structure, as well as "P1" and "P2" in the proof of the property.

```
definition dualize_half  :  half  → half := λ H:half.
  mk_half ( half_carr  H) (λ x,y.y ⋘ x) (λ x,y,z,P1,P2.half_property H x y z P2 P1)
```

Using the technique of manifest coercions described in the previous section we define the full structure, composed by two halves such that one is the dual of the other, with the difference that none of the projections is declared as a coercion.

```
record full  :  Type := {
  half_l  :  half;
```

```
  half_r_ : half;
  with_ : half_r_ = dualize_half half_l
}.

lemma half_r : full → half. intros; apply (dualize_half ( half_l f )); qed.
```

The "with_" constraint here is so strong that allows us to completely remove the "half_r_" field, since its content has to be exactly what the "dualize_half" produces in output. Equivalently the second halve is completely concrete and thus the manifesting coercion must not preserve anything from "half_r_". Thus the definition of the full structure boils down to a degenerated record, that we need just in order to distinguish the half structure from the full one. When we want to use the full structure as an half structure, the system expects us to explicitly project it with either "half_l" or "half_r".

```
record full : Type := { half_l : half }.
```

To use the "half_property" to prove both versions of transitivity it is sufficient to: 1) introduce the notation "≤" for "half_pred (half_l ?)" and "≥" for "half_pred (half_r ?)", following the intuition that the relation inside the primary structure ("half_l") is the ≤ relation, while the one in the dual structure ("half_r") corresponds to ≥; 2) taking out the "half_property" from the appropriate half structure.

```
lemma test_l : ∀F:full. ∀x,y,z:F. x ≤ y → y ≤ z → x ≤ z.
intro F; apply half_property ( half_l F). qed.

lemma test_r : ∀F:full. ∀x,y,z:F. x ≥ y → y ≥ z → x ≥ z.
intro F; apply half_property ( half_r F). qed.
```

We also adopt the notation "le_transitive" for "half_property (half_l ?)" and "ge_transitive" for the dual one, so that, even if there is just on "half_property" object defined, the user has the illusion of having two distinct versions of the transitivity axiom.

Notation here plays a double role. First, it is used for pretty printing purposes, to display "≥" when the relation in question comes from the dual half of a structure. We are thus obliged to attach the notation not only the to the head constant "half_pred", but also to its first argument, in case it is projecting out from a full structure the primitive or the dual half. Second, it is used for input purposes. Like in the previous example, the "half_property" constant can be used for proving both the transitivity of the "≤" relation and the transitivity of "≥", but as users we clearly prefer to type "**apply** le_transitive" when facing a thesis like "a ≤ b" and "**apply** ge_transitive" when the thesis is "a ≥ b". If we associated the input notation to just the head constant "half_property", the user were allowed to type "**apply** le_transitive" when facing the conclusion "a ≤ b", but nothing prevented him to type "**apply** ge_transitive", since the two notations were just aliases for the same constant. Moreover, even if the user did not specify the first argument of "ge_transitive", the type inference algorithm of the interactive theorem prover was likely to infer from the conclusion (that under the notation "a ≤ b" hides the projection "half_l F") that the lemma "half_property" is applied to "half_l F".

Even worse, a command like "**apply** (le_transitive (half_r F))" would had made sense to the system. To allow only a meaningful usage of the " le_transitive " and " ge_transitive " names, we impose them to be partially instantiated to the respective projections " half_l " and " half_r ". We decide to use the notation mechanism instead of defining concrete objects for these partial instantiations mainly to avoid pollution of the space of theorems and definitions.

This methodology works also for lemmas and not only axioms of our dual structures. Dual lemmas have to be proved for the "half" structure, and two different notations are attached to them, partially instantiating their first argument (as for the "∗ _transitive " pair). As an example we consider the following trivial lemma.

```
lemma half_trivial : ∀H:half. ∀x,y:H. x ≪ y → y ≪ x → x ≪ x.
intros; apply (half_property h ??? H H1). qed.
... (* Omitted declarations of notations for le_trivial and ge_trivial *)
check ( le_trivial ). (* : ∀x,y: half_carr (half_l ?_F). x ≤ y → y ≤ x → x ≤ x *)
check ( ge_trivial ). (* : ∀x,y: half_carr (half_r ?_F). x ≥ y → y ≥ x → x ≥ x *)
```

Again, we attached the notation " le_trivial " to "half_property (half_l ?)" and " ge_trivial " "half_property (half_r ?)" to have different names for the partial instantiations of the same " trivial " lemma. We report in comment the inferred type, where "$?_F$" is an implicit argument of type " full " the system is expected to infer when the lemmas are applied to a goal. Since " half_carr " is declared as a coercion, the system actually prints "$?_F$" in place of both " half_carr (half_l $?_F$)" and " half_carr (half_r $?_F$)". Remember that the two terms that are printed in the same way are convertible by construction of " half_r ", thus no confusion arises.

3.2.2 *The problem.* This technique worked fine in our development until we reached the concept of *segment*. For example consider the following statement, asserting that a sequence "a" of points in a segment "{[l,u]}" has "u" as an upper bound (or better "u" is the upper bound of the sequence of first projections of "a", since a segment is the Σ-type of points that lay in the segment). In the following snippet we denote "$(\pi_1(a\ n))_n$" with "$\lfloor n,\pi_1(a\ n)\rfloor$". The latter notation is required to enter formulae in Matita, but the former is the one pretty printed by the system.

```
lemma segment_upperbound:
  ∀C:half_ordered_set.∀l,u:C.∀a:sequence {[l,u]}. u is_upper_bound ⌊n,π₁ (a n)⌋.
```

To obtain the dual statement is clearly not enough to change "upper_bound" (defined in terms of "≤") with "lower_bound". We also want "l" to replace "u" in the conclusion of the lemma. This suggested us that the dualization of the relation "half_pred" cannot be hidden (as we did) in the " dualize_half " construction. Here some parts of the statement, not only arguments of a "≪" expression, have to be changed w.r.t. the structure in input. More to the point we would like to replace "u" with something like " if is_dual C then l else u".

3.2.3 *The solution.* Our solution consisted in adding two fields to the "half" structure:

```
  wloss: ∀A,B:Type. (A → A → B) → A → A → B;
  wloss_prop:
```

```
  (∀T,R,P,x,y.P x y = wloss T R P x y) ∨ (∀T,R,P,x,y.P y x = wloss T R P x y);
```

The "wloss" field (that stands for "without loss of generality", inspired by the wloss tactic, part of the SSR [GM] proof shell) is a function that the "wloss_prop" field forces to be (in a closed context) extensionally equal to one of the following:

— "λ A,B,P,x,y.P x y"

— "λ A,B,P,x,y.P y x"

The first is just (η-equivalent to) "P", the latter swaps the arguments before calling "P".

The complete definition of the half structure and its dual follows

```
record half : Type :={
  half_carr :> Type;
  wloss: ∀A,B:Type. (A → A → B) → A → A → B;
  wloss_prop:
    (∀T,R,P,x,y.P x y = wloss T R P x y) ∨ (∀T,R,P,x,y.P y x = wloss T R P x y);
  half_pred_:   half_carr  → half_carr  → CProp;
  half_property:
    ∀x,y,z.wloss ?? half_pred_ x y → wloss ?? half_pred_ y z → wloss ?? half_pred_ x z
}.
definition half_pred := λH,x,y. wloss H ?? (half_pred_ H) x y.
```

Note that, attaching to "half_pred" the infix notation "≤", "half_property" simply states "∀x,y,z.x ≤ y → y ≤ z → x ≤ x" as expected.

We then build the dualizer, such that the carrier and the 'half_pred_" relation are preserved, while the "wloss" field swaps its arguments w.r.t. the "wloss" field of the input structure "H".

```
definition dualize_half : half → half := λH.
 mk_half (half_carr H) (λA,B,P,x,y. wloss H A B P y x)
   (match wloss_prop H with
    [ or_intro_l p ⇒ or_intro_r ?? p
    | or_intro_r p ⇒ or_intro_l ?? p ])
   (half_pred_ H) (λx,y,x,H1,H2.half_property H z y x H2 H1)
```

The wloss property is easily proved introducing the disjunction the other way around: we need to prove that

```
(∀T,R,P,x,y. P x y = (λA,B,P,x,y. wloss H A B P y x) T R P x y) ∨
(∀T,R,P,x,y. P y x = (λA,B,P,x,y. wloss H A B P y x) T R P x y)
```

That is equivalent, up to commutativity of disjunction, to the wloss property of "H".

```
(∀T,R,P,x,y. P x y = wloss H T R P y x) ∨
(∀T,R,P,x,y. P y x = wloss H T R P y x)
```

3.2.4 *A more intuitive but incorrect solution.* Intuitively, the previous solution corresponds to adding to the structure a boolean that discriminates between $\leq$ and $\geq$. Moreover, the computational content of "wloss_prop" is isomorphic to that boolean being a disjoint union of unit types. This suggests a simpler implementation that does not work properly. The implementation is the following:

```
record half : Type :={
  ...
  wloss: bool
  ...
}.
definition half_pred := match b with [ true ⇒ half_pred_ y x | _ ⇒ half_pred_ x y ]

definition dualize_half : half → half := λ H.
  ...
   (match (wloss H) with [ true ⇒ false | false ⇒ true ])
  ...
```

The problem is that "a ≤ b" and "b ≥ a" are no longer convertible when the structure is not concrete. Indeed, without notation, and unfolding the definition of "half_pred",
"a ≤ b" becomes

```
match wloss (half_l F) with
[ true   ⇒ half_pred_ ( half_l F) b a
| false  ⇒ half_pred_ ( half_l F) a b ]
```

If we adopt the notation "a $\lll_{F_L}$ b" for "half_pred_ (half_l F) a b" we have

```
match wloss (half_l F) with [ true ⇒ b ⋘_{F_L} a | false ⇒ a ⋘_{F_L} b ]
```

Similarly, if we expand the definition of "b ≥ a" we obtain

```
match wloss (half_r F) with [ true ⇒ b ⋘_{F_R} a | false ⇒ a ⋘_{F_R} b ]
```

where "a $\lll_{F_R}$ b" stands for "half_pred_ (half_r F) a b". Expanding the definition of the wloss field of the dual structure we obtain the following

```
match (match (wloss F) with [ true ⇒ false | false ⇒ true ])
with [ true ⇒ b ⋘_{F_R} a | false ⇒ a ⋘_{F_R} b ]
```

Thus, what we obtained is not convertible, in a context where "F" is not a closed term (i.e. it is a variable), with "a ≤ b".

Adding commuting conversions to the system (see for instance [**?**]) would solve the issue but, unfortunately, commuting conversions do not mix well with the dependently typed discipline of CIC.

On the contrary, our solution introduces no pattern matching construct in the definition of the dual relation, it just adds some β-redexes that swap arguments. The notation "a ≤ b", just hides

```
half_pred ( half_l F) a b
```

that reduces to the following term (we substituted two types with question marks to make it more readable)

```
wloss ( half_l F) ?? (half_pred_ ( half_l F)) a b
```

The same term is obtained starting from "b ≥ a", that hides

```
half_pred ( half_r F) b a
```

Note that the "half_r" (manifesting) projection is defined as "(dualize_half (half_l F))". Since the dualizer does not change the "half_pred_" relation we obtain

```
wloss ( dualize_half ( half_l F)) ?? (half_pred_ ( half_l F)) b a
```

The wloss field of the dualized structure is defined in terms of the wloss field of the input structure but with its arguments swapped. Thus we have

```
(λ A,B,P,x,y.wloss (half_l F) A B P y x) ?? (half_pred_ ( half_l F)) b a
```

Firing all β-redexes we obtain exactly what we obtained starting from "a ≤ b".

3.2.5 *The solution at work.* Using the "wloss" device we can easily define the upper/lower segment w.r.t. an half structure "H" delimited by

```
definition seg_u := λ H,l,u. wloss H ?? (λ x,y.x) u l
definition seg_l := λ H,l,u. wloss H ?? (λ x,y.x) l u
```

Note that "wloss" can just swap arguments, thus an alternative definition like the following

```
definition seg_u := λ H,l,u. wloss H ?? (λ x,y.y) l u
```

is not equivalent, since when "H" is instantiated with something like "half_r O" the definition reduces to

```
wloss ( half_l O) ?? (λ x,y.y) u l
```

that is not convertible with the following term

```
wloss ( half_l O) ?? (λ x,y.x) l u
```

In a similar way, the predicate of being in a segment is defined as follows. Note the use of "wloss" to avoid the non convertibility of "A∧ B" with "B∧ A". Also note that coercions can be used to map proofs of "A∧ B" to proofs of "B∧ A", but that they are not propagated automatically under CIC constructors (e.g. a record type) unless additional coercions are declared (e.g. for each record type). Forcing convertibility as we do completely solves the problem.

```
definition in_segment := λ H,l,u,x.
  wloss H ?? (λ p1,p2.p1∧ p2) (seg_l H l u ≪ x) (x ≪ seg_u H l u)
```

The motivation for the definition is the following. Since we will be interested in the Σ-type of points in a segment (denoted with "{[l,u]}"), and we will prove lemmas about these objects, we have to be sure that the type

```
Σx:half_carr (half_l O).in_segment (half_l O) l u x
```

is convertible with the type

```
Σx:half_carr (half_r O).in_segment (half_r O) l u x
```

If this was not the case, we would be unable to apply primitive and dual lemmas on the same points. Since "half_carr (half_r O)" is by construction "half_carr (half_l O)" we are left to show that "in_segment (half_r O) l u x" and "in_segment (half_l O) l u x" are convertible. Unfolding the definition of "in_segment" we obtain

```
wloss (half_r O) ?? (λ p1,p2.p1∧ p2)
  (seg_l (half_r O) l u ≥ x) (x ≥ seg_u (half_r O) l u)
```

By unfolding definitions of "seg_l" and "seg_u" we obtain the following:

```
wloss (half_r O) ?? (λ p1,p2.p1∧ p2)
  (wloss (half_r O) ?? (λ x,y.x) l u ≥ x) (x ≥ wloss (half_r O) ?? (λ x,y.x) u l)
```

Since the "half_r O" (manifesting) projection is defined as "(dualize_half (half_l O))" we obtain

```
wloss (dualize_half (half_l O)) ?? (λ p1,p2.p1∧ p2)
  (wloss (dualize_half (half_l O)) ?? (λ x,y.x) l u ≥ x)
  (x ≥ wloss (dualize_half (half_l O)) ?? (λ x,y.x) u l)
```

If we project out the wloss field from the outermost dualized structure and we fire all β redexes we obtain

```
wloss (half_l O) ?? (λ p1,p2.p1∧ p2)
  (x ≥ wloss (dualize_half (half_l O)) ?? (λ x,y.x) u l)
  (wloss (dualize_half (half_l O)) ?? (λ x,y.x) l u ≥ x)
```

Doing the same for the occurrences in the arguments we have

```
wloss (half_l O) ?? (λ p1,p2.p1∧ p2)
  (x ≥ wloss (half_l O) ?? (λ x,y.x) l u) (wloss (half_l O) ?? (λ x,y.x) u l ≥ x)
```

Note that the notation "≥" hides an occurrence of "(half_r O)" too, and as shown before "x≥ y" converts with "y≤ x". We thus obtain

```
wloss (half_l O) ?? (λ p1,p2.p1∧ p2)
  (wloss (half_l O) ?? (λ x,y.x) l u ≤ x)  (x ≤ wloss (half_l O) ?? (λ x,y.x) u l)
```

As expected, folding back the definition of "seg_l" and "seg_u" we obtain the definition of "in_segment (half_l O) l u x".

```
wloss (half_l O) ?? (λ p1,p2.p1∧ p2)
  (seg_l (half_l O) l u ≤ x)  (x ≤ seg_u (half_l O) l u)
```

3.2.6 *Drawbacks.* Duality is here embedded in our objects to be applied by the reduction mechanism when needed, but this technique for handling duality forces us to put the half structure in the third predicative universe. Indeed the field "wloss" is declared with type

$\forall\,\mathrm{A}{:}\mathbf{Type}_i.\forall\,\mathrm{B}{:}\mathbf{Type}_j.\ (\mathrm{A} \to \mathrm{A} \to \mathrm{B}) \to \mathrm{A} \to \mathrm{A} \to \mathrm{B}$

and to state, for example, the co-reflexivity property we write

```
coreflexive ? (wloss ?? hos_excess_)
```

We thus force the conversion of "$\mathrm{A} \to \mathrm{A} \to \mathrm{B}$" (the expected type of the third "wloss" argument) with the type of "hos_excess_" that is "hos_carr $\to$ hos_carr $\to \mathbf{CProp}_0$". This forces the type "$\mathbf{Type}_j$" of "B" to be at least the type of "$\mathbf{CProp}_0$" that is "$\mathbf{Type}_1$". The type of "wloss" is thus forced to be

$\forall\,\mathrm{A}{:}\mathbf{Type}_0.\forall\,\mathrm{B}{:}\mathbf{Type}_1.\ (\mathrm{A} \to \mathrm{A} \to \mathrm{B}) \to \mathrm{A} \to \mathrm{A} \to \mathrm{B}$

CIC typing rules require the sort of an inductive type to be grater or equal to the sort of all the constructors of the inductive type. Thus, the inductive type representing the half structure must be put in "$\mathbf{Type}_2$" instead of "$\mathbf{Type}_1$".

This lifting phenomenon is recurrent when reflection mechanisms are employed to prove statements, but the statements themselves are not usually lifted up. Indeed, what usually requires higher universes is the statement of the correctness theorem of the reflection procedure, whose application is usually hidden in the proofs, letting statements live in lower type universes. Instead, in our case, we quantify over half structures lifting this way every statement.

3.3 The Russell language

The Russell language [Soz06] allows to write programs using simple (non dependent) types and let the system annotate them with user provided proofs retyping them with a more sophisticated specification (i.e. a dependent type). For example one may write a function sorting a list as a term of type "list $\to$ list", and let the system generate the proof obligations necessary to decorate it such that it inhabits the more precise type "$\forall$l: list .$\exists$l1. sorted l1 $\wedge$ l1 is_permutation l".

This technology amounts at having support for declaring sub-set coercions (in the style of PVS [SO99]) like

```
in_sigma : ∀l: list . P l → Σl:list.P l
```

and their respective projections

```
out_sigma : (Σl:list.P l) → list
```

When a function defined by recursion on a list like the following

```
let rec f (l : list ) on l : list :=
  match l with
  [ nil ⇒ g1
  | cons x tl ⇒ g2 x (f tl) ]
```

is casted to a more precise type like "∀l: list .∃l1.P l l1" the system has to propagate the "in_sigma" coercion under the fix point and pattern match constructions, applying "out_sigma" to every recursive call.

```
let rec f (l : list) on l : ∃l1.P l l1 :=
  match l with
  [ nil ⇒ in_sigma g1 ?1
  | cons x tl ⇒ in_sigma (g2 x (out_sigma (f tl))) ?2 ]
```

Note that question marks represent missing proofs the user will be later asked to provide: "$?_1$" corresponds to the proof of "P nil g1", while "$?_2$" corresponds to the proof of "P (cons x tl) (g2 x (out_sigma (f tl)))" under the extra assumption that "P tl (out_sigma (f tl))" holds.

The main advantage of this technology is to ease the writing of programs that use dependent types, splitting the activity in two separate phases: the former in which the user writes a program using simple types, and the latter in which every proof annotation is solved using the regular proof language of the ITP.

Writing together the program and its proof annotations is quite hard, since the user cannot benefit from the ITP proof language and has to provide all proof terms at once writing them explicitly.

In constructive mathematics the simply typed user provided program becomes the explicitly given computational content, and the program specification becomes the intuitionistic statement to be proved.

The Russell language has been implemented in Matita as part of the second author Ph.D. thesis [Tas08].

4. FORMALISING THE PROOF

4.1 Sets equipped with an order or an equivalence relation

The first structure defined is the "half_ordered_set". The reflected duality technique explained in Section 3.2 has been adopted to avoid the duplication of every result about ordered sets.

```
record half_ordered_set: Type2 :={
  hos_carr:> Type0;
  wloss: ∀A:Type0.∀B:Type1. (A → A → B) → A → A → B;
  wloss_prop:
   (∀T,R,P,x,y.P x y = wloss T R P x y) ∨ (∀T,R,P,x,y.P y x = wloss T R P x y);
  hos_excess_: hos_carr → hos_carr → CProp0;
  hos_coreflexive : coreflexive ? (wloss ?? hos_excess_);
  hos_cotransitive : cotransitive ? (wloss ?? hos_excess_)
}.
definition hos_excess :=λO:half_ordered_set.wloss O ?? (hos_excess_ O).
```

As explained in Section 2, since the excess is a positive predicate with an interesting computational content we put it in "$\mathbf{CProp}_0$".

The "dual_hos" construction builds the dual half ordered set structure, swapping the arguments of the "wloss" projection while preserving the same "hos_excess_" field. We build it with the tactic language of Matita. Square brackets are the concrete syntax for the branching LCF tactical (implemented in a small step fashion in

Matita [STZ06]), thus the second proof line builds the carrier of the dual structure, the third and forth the "wloss" related fields, the fifth provides the excess relation and the latter two proofs of its properties.

```
definition dual_hos : half_ordered_set → half_ordered_set.
intro; constructor 1;
[ apply (hos_carr h);
| apply (λ T,R,f,x,y.wloss h T R f y x);
| intros; cases (wloss_prop h);[right|left ]intros;apply H;
| apply (hos_excess_ h);
| apply (hos_coreflexive h);
| intros 4 (x y z H); simplify in H ⊢%; cases (hos_cotransitive h y x z H);
  [right|left ] assumption;]
qed.
```

We then define the "ordered_set" structure and its manifest coercion "os_r" using the technique explained in Sections 3.1 and 3.2.

```
(* Definition 2.1 *)
record ordered_set : Type :={
  os_l : half_ordered_set ;
}.

definition os_r : ordered_set → half_ordered_set.
intro o; apply (dual_hos (os_l o)); qed.
```

What follows is the first lemma that is proved on the "half_ordered_set" structure, to be used for both the ≤ and the ≥ relations. Here we denote with "a ≰ b" the application "hos_excess_ H a b" for some half ordered set 'H".

```
definition le :=λ E:half_ordered_set.λ a,b:E. ¬ (a ≰ b).

lemma hle_reflexive: ∀ E.reflexive ? (le E).
unfold reflexive; intros 3; apply (hos_coreflexive ? x H); qed.
```

After that, different names are associated to the same lemma: if the lemma is applied to the primitive structure (projected out with "or_l") the name " le_reflexive " is used, while " ge_reflexive " is used when the lemma is applied to the dual substructure. The "**notation**" command associates a name ("le_reflexive" in the first case) to an abstract syntax tree (a single node called "' le_reflexive " in the first case). The "**interpretation**" command relates an abstract syntax tree with a CIC term (where "_" is a placeholder for any argument). This works for both input and output: if the user types in " le_reflexive " it is interpreted as " hle_reflexive (os_l ?)" for some implicit argument "?"; if the system prints out " hle_reflexive (os_l C)" for some "C" the user sees " le_reflexive ".

```
notation "'le_reflexive'" non associative with precedence 90 for @{'le_reflexive }.
notation "'ge_reflexive'" non associative with precedence 90 for @{'ge_reflexive}.
interpretation "le reflexive" 'le_reflexive = (hle_reflexive (os_l _)).
interpretation "ge reflexive" 'ge_reflexive = (hle_reflexive (os_r _)).
```

A "bishop_set" is a type equipped with an apartness relation (denoted with "#"). In [BE85] Bishop requires every set to be equipped with an equivalence relation while the apartness (there called inequality) may not be present. Since we are interested in models where only the latter carries a computational content (for example real functions) we always require the apartness relation and we induce the equality from it in order to make it *tight*, i.e. $\neg(x \# y) \Rightarrow x = y$.

We again use the universe of constructive propositions to represent the apartness relation, that carries with it some computational content, while its negation will defining the equality relation.

```
(* Definition 2.2 *)
record bishop_set: Type₁ :={
  bs_carr:> Type₀;
  bs_apart: bs_carr → bs_carr → CProp₀;
  bs_coreflexive : coreflexive ? bs_apart;
  bs_symmetric: symmetric ? bs_apart;
  bs_cotransitive : cotransitive ? bs_apart
}.
```

Every "ordered_set" give rise to a "bishop_set" when the apartness relation is defined by "λ a,b.a ≰ b ∨ b ≰ a". The equality relation is obtained negating apartness as expected.

```
definition bishop_set_of_ordered_set : ordered_set → bishop_set.
intros (E); apply (mk_bishop_set E (λ a,b:E. a ≰ b ∨ b ≰ a)); . . . qed.

definition eq :=λ A:bishop_set.λ a,b:A. ¬ (a # b).
```

4.2 Dual definitions over sets

What follows is the first set of completely dualized definitions. Note that they are all defined using the half structure, and a new infix or postfix notation is attached to their partial instantiations.

```
definition upper_bound :=λ O:half_ordered_set.λ a:sequence O.λ u:O.
  ∀ n:ℕ.a n ≪ u.

(* Definition 2.4 *)
definition supremum :=λ O:half_ordered_set.λ s:sequence O.λ x.
  upper_bound ? s x ∧ (∀ y:O.x ≰≱ y → ∃ n.s n ≰≱ y).

definition increasing :=λ O:half_ordered_set.λ a:sequence O.
  ∀ n:ℕ.a n ≪ a (S n).

definition uparrow :=λ C:half_ordered_set.λ s:sequence C.λ u:C.
  increasing ? s ∧ supremum ? s u.
```

The type constructor sequence, is an inductive type parametrised over the type of elements.

```
inductive sequence (O:Type) : Type :=mk_seq : (nat → O) → sequence O.
```

```
definition fun_of_seq: ∀O:Type.sequence O → nat → O :=λO.λx:sequence O.
  match x with [ mk_seq f ⇒f ].
```

Its constructor simply wraps a function from natural numbers to the parametrised type. Then, a coercion from the sequence type to the space of functions had been declared, allowing us to simply apply an object of type sequence to a number n to obtain the n-th element of the sequence.

We decided to wrap sequences inside that constructor to be able to attach a notation to it. The output notation is the usual one for sequences "$(\mathrm{a})_\mathrm{n}$". On the contrary, some limitations in the notation system of Matita forced us to adopt different parenthesis for the input syntax. We will thus write "⌊n, (f a) n⌋" for the sequence "$(\mathrm{f\ a})_\mathrm{n}$".

By convention, every dualized lemma has its name prefixed with "h_" and the remaining part of it refers to the primitive version of its statements. For example the following lemma states that given a strictly increasing sequence of natural numbers m, and a sequence a of an ordered set, if $(a)_n \uparrow u$ then $(a)_{(m\ n)} \uparrow u$. Its primitive name would include the uparrow word, while its dual would use downarrow; we thus call the dualized version "h_selection_uparrow", and we attach to its partial instantiation the names "selection_downarrow" and "selection_uparrow".

```
(* Lemma 2.5 *)
lemma h_selection_uparrow:
  ∀C:half_ordered_set.∀m:sequence nat_ordered_set. m is_strictly_increasing →
    ∀a:sequence C.∀u.uparrow ? a u → uparrow ? ⌊x,a (m x)⌋ u.
```

Note that, non dualized notions in the statement, like the fact that the sequence of naturals is strictly increasing, are expressed using the postfix version of the increasing predicate, and thus refer to the primitive substructure of the ordered set of natural numbers (i.e. the sequence is strictly increasing w.r.t. $\leq$ and not $\geq$).

The following definition has no interesting dualization, moreover it involves at once one notion (infimum) and its dual (supremum). Thus we express it over the full ordered set structure.

```
(* Definition 2.6 *)
definition order_converge :=
  λO:ordered_set.λa:sequence O.λx:O.
    ∃l,u. l ↑ x ∧ u ↓ x ∧
      ∀i:ℕ. (l i) is_infimum ⌊w,a (w+i)⌋ ∧ (u i) is_supremum ⌊w,a (w+i)⌋.
```

As already explained in Section 3.2 the definition of segment deserves special care. Here we pack the bounds of a segment inside a record, to ease the quite frequent operation of quantification over a segment. This also give us named projections for its components, whose semantics is clear and does not depend on the order of their arguments (that we reduced to a single record).

```
record segment (O : Type) : Type :={ seg_l_ : O; seg_u_ : O }.

definition seg_u :=λO:half_ordered_set.λs:segment O.
  wloss O ?? (λl,u.l) (seg_u_ ? s) (seg_l_ ? s).
```

```
definition seg_l :=λ O:half_ordered_set.λ s:segment O.
  wloss O ?? (λ l,u.l) ( seg_l_ ? s) (seg_u_ ? s).

(* Definition 2.7 *)
definition in_segment :=λ O:half_ordered_set.λ s:segment O.λ x:O.
    wloss O ?? (λ p1,p2.p1 ∧ p2) (seg_l ? s ≤≤ x) (x ≤≤ seg_u ? s).
```

As we detailed before, the following test lemma grants the property that being in a segment w.r.t an order relation or its dual is the same, in the strong sense that the two statements are convertible.

```
lemma test: ∀ O:ordered_set.∀ s: segment (os_l O).∀ x:O.
   in_segment (os_l O) s x = in_segment (os_r O) s x.
intros; reflexivity; qed.
```

A pair of technical lemmata are needed to properly manipulate objects living in a segment. Given an half ordered set and a segment on it, two points in that segment are in the excess relation of the ordered set if and only if they are in the excess relation of the ordered set restricted to the segment. In the following snippet "half_segment_ordered_set" is the half ordered set restricted to the segment "s"; we state this technical lemma on the half structure since is has an interesting dual. Later we will adopt the notation "{[s]}" for ordered set restricted to the segment "s".

```
lemma x2sx: ∀ O:half_ordered_set. ∀ s:segment O.
  ∀ x,y:half_segment_ordered_set ? s.   π1 x ≰≰ π1 y → x ≰≰ y.
```

This technical lemma shows a shortcoming of our approach to duality. Even if the excess for objects inside the segment is defined as the excess of their projections, these two notions are not convertible because of the wloss projection on a possibly abstract ordered set. The excess relation of the ordered set restricted to the segment, when applied to two points of the segment, is the following:

```
wloss O ?? (λ x,y.hos_excess_ O (π1 x) (π1 y)) a b
```

That term is not convertible, when O is abstract, with the excess relation outside the segment, since the two points are projected outside the excess relation:

```
wloss O ?? (hos_excess_ O) (π1 a) (π1 b)
```

The problem is ameliorated by the fact that Matita allows to declare coercions from and to the same type, thus the lemma "x2sx" can be declared as a coercion and the system inserts it to fix terms written by the user. Nevertheless, the proof object still contains application of this lemma.

4.3 Uniformities and ordered uniformities

The formal definition of a uniform space is the most challenging one in the whole development. According to the informal definition, a uniform space is characterised by a *family* Φ of entourages that are *sub-sets* of some cartesian product $C \times C$.

In Type Theory, both sub-sets and families can be represented in two different ways: as a propositional function (that recognises elements of the set/collection)

or as an indexed collection (that enumerates the elements of the set/collection). In constructive mathematics, the two representations have very different properties (see, for instance, Sect. 2 of [HH08]). In particular, only the second one can be handled predicatively and only under the assumption that the collection is set indexed. The proof given in [CSCZ] is fully predicative only under the assumption that Φ is represented as a set indexed collection.

The computational content of the two representations is also completely different. In the second representation each element of the collection is identified by its name (the index) and all functions extracted from the proofs only manipulate names. In the first representation we identify Φ with a propositional function that maps elements of the collection to propositions. Since propositions have in general no computational representation (no names), there is no computational content associated with the first representation.

What representation seem the most appropriate for Φ? In order to maintain the predicative spirit of the proof and to be able to preserve the computational content, it seems natural to represent Φ as a set indexed family of sub-sets (called basic entourages) of $C \times C$. On the other hand, we do not see any benefit in representing the sub-sets themselves as indexed families, since we are never interested (in the Lebesgue's proof) in providing an evidence that some point is in some basic entourage. Thus we decide to represent basic entourages with propositional functions over $C \times C$. The final representation for Φ would then be $S \to (C^2 \to \mathbf{Prop})$ where S is a set of indexes.

Adopting the latter representation is certainly possible. On the other hand, mathematicians (at least the classical ones) are not used to work with indexed families and they prefer to stick with propositional functions (i.e. characteristic functions). The type system of CIC suggest an intermediate representation: instead of representing a family over T as an enumeration $S \to T$ for some S, we can represent it as a function $T \to \mathbf{CProp}$ where $\mathbf{CProp}$ is the universe of propositions that have a computational content. From the user point of view, this is not very different from having a propositional function $T \to \mathbf{Prop}$. However, it makes possible to have both computationally irrelevant models and models where the function maps every element of T to a Σ-type (a constructive existential) in $\mathbf{CProp}$. In our case, the elements of Φ have no constructive content, being basic entourages represented as $C^2 \to \mathbf{Prop}$. Thus, in those models, Φ of type $(C^2 \to \mathbf{Prop}) \to \mathbf{CProp}$ computationally maps the unit type to some Σ-type that, computationally, is just a set of elements (the first projections of the inhabitants of the Σ-type). Thus it is possible to apply the computational content of Φ to the inhabitant of the unit type to retrieve a set of names (i.e. an enumeration) for the elements of Φ.

As an experiment, in the formalisation in Matita we have adopted the latter representation, obtaining the following definition.

```
(* Definition 2.12 *)
record uniform_space : Type :={
  us_carr:> bishop_set;
  us_unifbase: (us_carr ² → Prop) → CProp;
  us_phi1: ∀U:us_carr² → Prop. us_unifbase U → (λx:us_carr².π₁ x ≈ π₂x) ⊆U;
  us_phi2: ∀U,V:us_carr² → Prop. us_unifbase U → us_unifbase V →
    ∃W:us_carr² → Prop.us_unifbase W ∧ (W ⊆(λx.U x ∧ V x));
```

```
  us_phi3: ∀U:us_carr² → Prop. us_unifbase U →
    ∃W:us_carr² → Prop.us_unifbase W ∧ (W ∘W) ⊆U;
  us_phi4: ∀U:us_carr² → Prop. us_unifbase U → ∀x.(U x → U⁻¹x) ∧ (U⁻¹ x → U x)
}.
```

When we will show the definition of uniform space restricted to a segment, the user will see the construction of a model based on the Σ-type construction. Another trivial model of that kind will be given in Section 4.6. Nevertheless, we must notice that this representation does not force the user to build only models based on the Σ-type construction, and that the inspection of the computational content to retrieve the enumeration can only be performed at the meta-level, outside the logic. Thus it may not be fully satisfactory to every constructive and predicative mathematician. We conjecture that a translation of our formalisation to adopt the less controversial set indexed representation poses no major problems.

Uniform convergence is the only notion for uniform spaces we are interested in. Here we use the notation "⟨x,y⟩" to build the pair whose first element is "x" and second is "y".

```
(* Definition 2.14 *)
definition uniform_converge :=λ C:uniform_space.λ a:sequence C.λ u:C.
  ∀U.us_unifbase C U → ∃n. ∀i. n ≤ i → U ⟨u,a i⟩.
```

We now define uniformities whose carrier is also an ordered set by inheriting from both the structure of uniform spaces and the structure of ordered sets. We constrain the carrier (a Bishop set) of the uniform space to be the one induced by the ordered set.

```
record ordered_uniform_space_ : Type :={
  ous_os:> ordered_set;
  ous_us_: uniform_space;
  with_ : us_carr ous_us_ = bishop_set_of_ordered_set ous_os
}.
```

We then build the manifesting coercion for the uniform space sub structure and define the final ordered uniform space structure assuming the convexity property.

```
lemma ous_unifspace: ordered_uniform_space_ → uniform_space.

(* Definition 2.18 *)
record ordered_uniform_space : Type :={
   ous_stuff :>ordered_uniform_space_;
  ous_convex_l: ∀U.us_unifbase ous_stuff U → convex (os_l ous_stuff) U;
}.
```

One of the most involved constructions in the whole development is the definition of the ordered uniform space restricted to a segment, that is an ordered uniform space whose carrier is an ordered set restricted to a segment (i.e. "{[s]}" for some segment "s"). The key point of the construction is the definition of the family of basic entourages that will build the restricted uniform space. Computationally, that means that we must compute a set of indexes (names) for the basic entourages of the restricted uniformity from the set of indexes (names) for the basic entourages of

the unrestricted uniformity. Intuitively, the idea is that we can just reuse the same set of indexes, interpreting them as the original propositional functions restricted to elements in the square of the segment.

Technically, since we defined the uniformity base as (us_carr ^{2} → **Prop**) → **CProp**, we actually need to produce a function that, given a candidate basic entourage for the segment, recognises if it is extensionally equivalent to one of the functions associated to some index in the restricted uniformity. We do that by introducing the definition of restriction agreement (i.e. extensional equivalence in the square of the segment) between a basic entourage for the unrestricted space and a basic entourage for restricted space. We use the notation "‡O" for a segment of points in "O".

```
definition restriction_agreement :=
 λO:ordered_uniform_space.λs:‡O.λP:{[s]}²→Prop. λOP:O²→Prop.
  ∀b:{[s]} ² .(P b →OP b) ∧(OP b →P b))
```

Here we see in action coercions, since "b" lives in the restricted square ordered set, and is injected in the not restricted square ordered set by the following coercion.

```
definition ordered_set_square_of_segment_square :=λO:ordered_set.λs:‡O.λb:{[s]}².
  ⟨π₁(π₁ b),π₁(π₂ b)⟩.
```

This coercion allows us to keep the statement of the restriction agreement readable.

When we build the ordered uniform space restricted to a segment, we inhabit its family of entourages as follows. We ask that for every restricted entourage there exists a non restricted one such that the two agree on the segment.

```
(* Definition 2.16 applied to ordered uniform spaces *)
lemma segment_ordered_uniform_space:
  ∀O:ordered_uniform_space.∀s:‡O.ordered_uniform_space.
intros (O s); apply mk_ordered_uniform_space;
[1: apply (mk_ordered_uniform_space_ {[s]});
     [1: letin f :=(λP:{[s]} ² →Prop. ∃OP:O²→Prop.
                        (us_unifbase O OP) ∧restriction_agreement ?? P OP);
...
```

The existential notation "∃OP:O²→**Prop**" hides the Σ-type "ΣOP:O²→**Prop**" whose first projection is indeed the name (index) of the entourage. The requirement "(us_unifbase O OP)" constrains the name to be the one for some entourage in the unrestricted uniform space "O", i.e. the two sets of indexes are the same.

The first milestone in the proof is the so called sandwich theorem, stating that when two sequences a, b topologically converge to a point l and a third sequence x is always between a and b w.r.t. the order relation, then x also topologically converges to l.

```
(* Theorem 2.20 *)
theorem sandwich:
  ∀O:ordered_uniform_space.∀l:O.∀a,b,x:sequence O. (∀i:ℕ.a i ≤x i ∧x i ≤b i) →
    a uniform_converges l →b uniform_converges l →x uniform_converges l.
```

The proof poses no problems; the theorem is a consequence of the halving property of the uniform space and its convexity property.

4.4 Order continuity, property (σ) and exhaustivity

The order continuity property poses no difficulties, since the statement has no interesting dualization.

```
(* Definition 2.21 *)
definition order_continuity :=
  λ C:ordered_uniform_space.∀ a:sequence C.∀ x:C.
    (a ↑ x → a uniform_converges x) ∧ (a ↓ x → a uniform_converges x).
```

On the contrary, the (σ) property can be expressed in two variants, respectively assuming $a \uparrow x$ or $a \downarrow x$.

```
(* Definition 2.22 *)
definition property_sigma :=λ C:ordered_uniform_space.
 ∀ U.us_unifbase ? U →
  ∃ V:sequence (C^2 → Prop).
   (∀ i.us_unifbase ? (V i)) ∧
   (∀ a:sequence C.∀ x:C.
    (a ↑ x ∨ a ↓ x) → (∀ n.∀ i,j.n ≤ i → n ≤ j → V n ⟨a i,a j⟩) → U ⟨a 0,x⟩).
```

As explained in Section 2.4, the spirit of the definition of property (σ) in the informal proof required the fix consisting in assuming that either $a \uparrow x$ or $a \downarrow x$. From the point of view of our approach to duality this has not been the optimal solution since statements that involve property (σ) can no longer be dualized.

The alternative would have been to define property (σ) only on increasing sequences with respect to an half ordered uniform space. As a consequence, the final result would have required both (σ) and its dual to hold. The benefit would have been to exploit dualization for just two more lemmas. However, future extensions of the library could benefit from this alternative approach.

The following lemma employs the Russell technology explained in Section 3.3 to define the program "m" as done in the proof of lemma 2.23.

```
(* Lemma 2.23 *)
lemma sigma_cauchy:
  ∀ C:ordered_uniform_space.property_sigma C →
    ∀ a:sequence C.∀ l:C.(a ↑ l ∨ a ↓ l) → a is_cauchy → a uniform_converges l.
...
letin spec :=(λ z,k:ℕ.∀ i,j,l:ℕ.k ≤ i → k ≤ j → l ≤ z → w l ⟨a i,a j⟩);
```

It first defines a specification for a program that will be the computational content of the proof. Here "w" is a sequence of entourages of the uniform space, "H2" is the Cauchy hypothesis.

```
letin m :=(let rec aux (i:ℕ) : ℕ:=
  match i with
  [ O ⇒π1 (H2 (w i) ?)
  | S i' ⇒ max (π1 (H2 (w i) ?)) (S (aux i'))
  ] in aux
```

```
  : ∀z.∃k. spec z k);
```

The program "m" is there defined by the recursive function "aux" of type "ℕ → ℕ", that is then forced to be of type "∀z.∃k. spec z k".

There are two question marks in the body of "aux", both in place of a proof that "(w i)" is an entourage. They are presented to the user later, together with proof obligations generated by Russell. Even if their proof is not that complex (around 10 lines of proof script) the corresponding proof terms (that decorate the three lines recursive function written above) do not fit this sheet of paper. Since their type lives in the sort "**Prop**" (entourages like "(w i)" are of type "(us_carr ^{2} → **Prop**)"), an extraction process would erase them, obtaining the original program written by the user.

The exhaustivity property is defined as follows:

```
(* Definition 2.24 *)
definition exhaustive :=λ C:ordered_uniform_space.
  ∀ a,b:sequence C.
    (a is_increasing → a is_upper_located → a is_cauchy ) ∧
    (b is_decreasing → b is_lower_located → b is_cauchy ).
```

The following lemma is the only lemma in the full proof of the dominated convergence theorem we choose to not dualize. Its dualization, or better a proper handling of its dual, requires the "wloss" approach. To state its dual, we would need both the half ordered uniform space structure and a change to the definition of exhaustivity. Since no other lemma on ordered uniform spaces has an interesting dual, it is cheaper to state and prove the lemma twice than build all the machinery needed to dualize the lemma.

```
(* Lemma 2.25 *)
lemma restrict_uniform_convergence_uparrow:
 ∀ C:ordered_uniform_space.property_sigma C →
  ∀ s:segment (os_l C).exhaustive {[s]} →
    ∀ a:sequence {[s]}. ∀ x:C. ⌊n,π₁ (a n)⌋ ↑x →
      x ∈ s ∧ ∀ h:x ∈s.a uniform_converges ⟨x,h⟩.
```

The expected conclusion of the statement would have been the following:

```
Σh:x ∈s.a uniform_converges ⟨x,h⟩.
```

We stated it in our alternative and stronger formulation to stress that the proof of "a uniform_converges ⟨x,h⟩" does not depend in any way on the proof object for "x ∈ s", i.e, that the quantification over "h:x ∈ s" is (proof) irrelevant. We also note that this lemma has no computational content, since it is the proof of a conjunction of two negative ("≤") statements.

4.5 Lebesgue's dominated convergence theorems

Finally we present the two statements of the Lebesgue's dominated convergence theorem, the former one assuming that every segment is order continuous, the latter assuming (σ) and exhaustivity.

```
theorem lebesgue_oc:
  ∀ C:ordered_uniform_space.
    (∀ s:‡C.order_continuity {[s]}) →
     ∀ a:sequence C.∀ s:‡C.∀ H:∀ i:ℕ.a i ∈s.
      ∀ x:C.a order_converges x →
        x ∈ s ∧ ∀ h:x ∈s. uniform_converge {[s]} ⌊n,⟨a n,H n⟩⌋ ⟨x,h⟩.
theorem lebesgue_se:
  ∀ C:ordered_uniform_space.property_sigma C →
    (∀ s:‡C.exhaustive {[s]}) →
     ∀ a:sequence C.∀ s:‡C.∀ H:∀ i:ℕ.a i ∈s.
      ∀ x:C.a order_converges x →
        x ∈ s ∧ ∀ h:x ∈s. uniform_converge {[s]} ⌊n,⟨a n,H n⟩⌋ ⟨x,h⟩.
```

We again avoided using a Σ-type for the conclusion to stress proof irrelevance on the proof "h". The two proofs have no computational content, as discussed in Section 2.

4.6 A model based on the discrete uniformity over ℕ

We present now the formalisation of a very simple model of ordered uniformity: we just consider natural numbers endowed with the discrete uniformity and the usual excess relation (here identified with $<$).

The model is not mathematically interesting for Lebesgue's Dominated Convergence Theorem, since both uniform convergence and order convergence just mean that the sequence eventually stabilises. In particular, property (σ) is trivially true. On the contrary, to prove exhaustivity or order continuity, we need to prove the following constructively and computationally interesting property: every increasing sequence of natural numbers that has a strong supremum eventually stabilises, i.e. there exists a limit point l and an index j such that $\forall i \geq j.\ a_i = l$.

This simple model is useful to show the logical consistency of the axioms of ordered uniformities and the way to inhabit the family of basic entourages in the degenerate case where there is only one entourage.

The simple model based on the discrete uniformity over ℕ is built first defining the ordered set of natural numbers (where the excess relation is the decidable relation $<$).

Then we define the set, in Bishop's terminology, using the " bishop_set_of_ordered_set " construction shown in Section 4.1. Finally to build a uniform space we need to exhibit the family of basic entourages, that in our encoding is a term of type "(C^2 → **Prop**) → **CProp**" for a Bishop set "C". Since the discrete uniformity is the one generated by just one basic entourage that is the diagonal, we index the family of basic entourages on the unit type.

```
definition discrete_uniform_space_of_bishop_set : bishop_set → uniform_space.
intro C; apply (mk_uniform_space C);
[1: intro P; apply (∃ d:unit.∀ n:C².(π₁ n ≈ π₂ n → P n) ∧ (P n → π₁ n ≈ π₂ n));
```

The most difficult theorem in the whole model is the lemma stating that given a (not strictly) increasing sequence "a" in a segment "s", if "X" is the supremum of "a" there exists an index "i" such that "a_i" is equal to "X" (since here "X"

lives in the Σ-type of points belonging to the segment, we are interested in its first projection to be equal to the first projection of the i-th sequence element, we call that projection "x"). Let us recall that a point "x" is the supremum of a sequence "a" when it is the upper bound of the sequence and if "$\forall y.x \not\leq y \to \exists n.a_n \not\leq y$". A constructive proof has to entails a program that exhibits such an index. The idea we follow is that we can construct a strictly increasing subsequence of "a", that will thus reach every point in the segment below (or equal to) "x" in less than "U_s" steps (where "U" is the notation for the upper boundary of a segment) and then be constant once "x" is reached. Actually also "$x-a_0$" steps would be enough.

We build a sequence of indices "m" such that "a_{m_n}" has the desired property. The first index returned by "m" is "0". To compute the next one, let say $n+1$, we compute the n-th index by recursion obtaining "pred" and we evaluate "m_{pred}". If the first projection of "$a_{m_{pred}}$" is already equal to "x" we return "pred" (the sequence has already stabilised). Otherwise we use the supremum property to find an index "next" for "a" such that "a_{next}" exceeds "$a_{m_{pred}}$" and we return "next". Intuitively we are using the strong supremum assumption to skip over any constant subsequence.

```
lemma increasing_supremum_stabilizes: ∀s:‡ℕ.∀a:sequence {[s]}. a is_increasing →
  ∀X.X is_supremum a → ∃i.∀j.i ≤ j → π₁ X = π₁ (a j).
intros 4;cases X (x Hx);clear X;letin X :=⟨x,Hx⟩;fold normalize X;intros;cases H1;
letin spec :=(λi,j:ℕ.(U_s ≤ i ∧ x = π₁ (a j)) ∨ (i < U_s ∧ x + i ≤ U_s + π₁ (a j)));
(* 'spec i j' corresponds to the invariant: x − a_j ≤ max 0 (U_s − i) *)
letin m :=(
  let rec aux i :=
    match i with
    [ O ⇒O
    | S i' ⇒
        let pred :=aux i' in
        let apred :=a pred in
        match cmp_nat x (π₁ apred) with
        [ cmp_le _ ⇒pred
        | cmp_gt nP ⇒π₁ (H3 apred ?)]]
  in aux : ∀i:nat.∃j:nat.spec i j);
```

The specification for the program "m" is tricky: if the input "i" is greater than "U_s" we say that "$x = \pi_1(a\ j)$", otherwise we say that for an input "i" the output "j" is such that "$x - a_j \leq \max 0\ (\mathsf{U}_s - i)$". This will allow us to easily prove that "U_s" steps are enough to reach "x".

We now have a strictly increasing sub sequence of "a", but we still need to compute how many steps it needs to reach "x". Note that "m_{U_s}" is already a valid witness for the conclusion we have to prove, but we can do better.

We thus write another program that finds the *least* (up to "U_s") input "i" for "m" such that "a_{m_i}" is equal to "x".

```
letin find :=(
 let rec find i u on u : nat :=
  match u with
  [ O ⇒m i
  | S w ⇒match eqb (π₁ (a (m i))) x with
```

```
        [ true  ⇒(m i:nat)
        | false  ⇒find (S i) w]]
in find : ∀i,bound.∃j.i + bound = U_s→ x = π_1(a j));
```

The specification of the find program states that given a bound and an accumulator such that they sum up to "U_s", the program finds a "j" such that"$x = \pi_1(a\ j)$". Note that this specification is probably too weak to prove that this program computes the lowest "j", but the theorem we are proving does not require that (we decided to provide the smallest index, but the statement just requires an index).

The computational complexity of the resulting algorithm is clearly non optimal (it may be liner while it is quadratic) but the result is the lowest index "i" that makes "a_{m_i}" equal to "x". The proof of the optimal algorithm would have been more involved.

5. CONCLUSIONS

The formalisation presented in the paper has been a major test bench for some peculiar features of Matita and for the technique introduced by the authors in [ST07] to represent algebraic hierarchies. Historically, an early version of the present work based on [Spi05, Spi06], where inheritance is heavily used in the algebraic hierarchy, has been the initial motivation for the introduction of the technique itself. In turn, the technique forced us to enhance management of coercions in Matita, as described in the second author's Ph.D. thesis [Tas08]. During that enhancement it became obvious how to simply integrate in Matita the "Russel language" of [Soz06]. A posteriori, it became obvious how the Russel language was a very useful tool also for the formalisation presented here.

We also note that, from the user point of view, our implementation of Russel is more satisfactory than the one in Coq. Indeed, in Coq the Russel language is not fully integrated with the regular proof flow, since it requires ad-hoc commands that cannot be part of a proof. In Matita, instead, it is just part of the normal handling of coercions and it can be triggered anywhere in the middle of some proof. For instance, in the proof that natural numbers are a model of an ordered uniformity, we have exploited this featured twice.

The original proof in [CSCZ] is 7 pages long. Considering that an average page counts around 35 lines, we obtain a total length of 245 lines. The number of definitions and lemmas explicitly given amounts to 38. However, some "obvious" proofs and definitions are left to the user, like the definition of negated notions or the proof that the definition of restricted ordered uniformity is well posed. All dual definitions and proofs are also omitted.

The formal development counts 1450 lines, of which 200 are for the discrete uniformity model. Notations (also used to handle duality) amount to 180 lines. The number of objects (definition and theorems) in the proof is 108, of which 32 are purely technical lemmas and 76 are genuine mathematical lemmas and definitions. Thus, in retrospect, about half of the required definitions and lemmas were left to the reader in the pen&paper proof. Our treatment of duality allowed us to avoid 25 mathematical objects, two less than in the original proof.

Since the proof was done in a novel and very abstract setting, almost no part of the Matita standard library was reused outside the formalisation of the given

model. Nevertheless, the Bishop's style theory of partially ordered sets is a major contribution to the library that we expect to be largely reused in the future.

The De Bruijn intrinsic factor [Wie00] amounts to 2.2 (redundancy free factor between the number of lines), while the apparent one amounts to 5. Both numbers are quite satisfactory. Moreover all definitions and proofs follow very closely the one given in the pen&paper version, fully confirming in this example the benefits of the formalisation techniques we adopted.

We were able to spot one minor and one major mistake in the original proof. Their fixes did increase the size of the proof by few lines. Due to the still unfinished referral process, we do not know if they could have been spotted by a careful reader.

References

[ASTZ07] Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. User interaction with the Matita proof assistant. *Journal of Automated Reasoning*, 39(2):109–139, 2007.

[Bar04] Marian Alexandru Baroni. *The constructive theory of Riesz spaces and applications in mathematical economics*. PhD thesis, University of Canterbury, department of mathematics and statistics, 2004.

[BE85] Bridges D. Bishop E. *Constructive Analysis*. Springer Verlag, 1985.

[CFGW04] Luís Cruz-Filipe, Herman Geuvers, and Freek Wiedijk. C-corn, the constructive coq repository at nijmegen. In *Mathematical Knowledge Management*, volume 3119/2004 of *LNCS*, pages 88–103. Springer-Verlag, 2004.

[Coq] The Coq proof-assistant. http://coq.inria.fr.

[CSCZ] Claudio Sacerdoti Coen and Enrico Zoli. Lebesgue's dominated convergence theorem in bishop's style. Technical Report UBLCS-2008-18, University of Bologna (Italy), Department of Computer Science. Submitted to Annals of Pure and Applied Logic, Special Issue on 3rd Workshop on Formal Topology.

[FS03] Luís Cruz Filipe and Bas Spitters. Program extraction from large proof developments. In D. Basin and B. Wolff, editors, *Proceedings of TPHOLS 2003*, volume 2758 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[GM] Georges Gonthier and Assia Mahboubi. A small scale reflection extension for the coq system. Technical report N RR-6455 (2008), http://hal.inria.fr/inria-00258384/fr/.

[HH08] Peter Hancock and Pierre Hyvernat. Programming interfaces and basic topology. *Annals of Pure and Applied Logic*, 137:189–239, 2008.

[Luo89] Zhaohui Luo. Ecc, and extended calculus of constructions. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 385–395, Piscataway, NJ, USA, 1989. IEEE Press.

[Luo99] Zhaohui Luo. Coercive subtyping. *J. Logic and Computation*, 9(1):105–130, 1999.

[ML06] Per Martin-Löf. 100 years of zermelo's axiom of choice: what was the problem with it? *Comput. J.*, 49(3):345–350, 2006.

[PM89] Christine Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin, January 1989. ACM.

[Pol02] Robert Pollack. Dependently typed records in type theory. *Formal Aspects of Computing*, 13:386–402, 2002.

[Sac04] Claudio Sacerdoti Coen. A semi-reflexive tactic for (sub-)equational reasoning. In *Types for Proofs and Programs*, volume 3839/2006 of *LNCS*, pages 98–114. Springer-Verlag, 2004.

[Sai97] Amokrane Saibi. Typing algorithm in type theory with inheritance. In *The 24th Annual ACM SIGPLAN - SIGACT Symposium on Principle of Programming Language (POPL)*, 1997.

[SO99] Natarajan Shankar and Sam Owre. Principles and pragmatics of subtyping in pvs. In *Recent Trends in Algebraic Development Techniques*, volume 1827/2000 of *LNCS*, pages 37–52. Springer-Verlag, 1999.

[Soz06] Matthieu Sozeau. Subset coercions in coq. In *Types for Proofs and Programs*, volume 4502/2007 of *LNCS*, pages 237–252. Springer-Verlag, 2006.

[Spi05] Bas Spitters. Constructive algebraic integration theory without choice. In Thierry Coquand, Henri Lombardi, and Marie-Françoise Roy, editors, *Mathematics, Algorithms, Proofs*, number 05021 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005.

[Spi06] B Spitters. Constructive algebraic integration theory. *Ann. Pure Appl. Logic*, 137:380–390, 2006.

[ST07] Claudio Sacerdoti Coen and Enrico Tassi. Working with mathematical structures in type theory. In *Proceedins of TYPES 2007*, volume 4941/2008 of *LNCS*, pages 157–172. Springer-Verlag, 2007.

[STZ06] Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. Tinycals: step by step tacticals. In *Proceedings of User Interface for Theorem Provers 2006*, volume 174 of *Electronic Notes in Theoretical Computer Science*, pages 125–142. Elsevier Science, 2006.

[Tas08] Enrico Tassi. *Interactive Theorem Provers: issues faced as a user and tackled as a developer.* PhD thesis, University of Bologna, 2008.

[Web91] Hans Weber. *Uniform lattices I. A generalization of topological Riesz spaces and topological Boolean rings.* Ann. Mat. Pure Appl., 1991.

[Web93] Hans Weber. *Uniform lattices II. Order continuity and exhaustivity.* Ann. Mat. Pure Appl., 1993.

[Wer97] Benjamin Werner. Sets in types, types in sets. In Martin Abadi and Takahashi Ito editors, editors, *Theoretical Aspect of Computer Software TACS'97, Lecture Notes in Computer Science*, volume 1281, pages 530–546. Springer-Verlag, 1997.

[Wie00] Freek Wiedijk. The "De Bruijn factor". http://www.cs.ru.nl/~freek/factor/, 2000.