A computer verified, monadic, functional implementation of the integral.

Russell O'Connor[a], Bas Spitters[a]

[a]*Radboud University Nijmegen*

Abstract

We provide a computer verified exact monadic functional implementation of the Riemann integral in type theory. Together with previous work by O'Connor, this may be seen as the beginning of the realization of Bishop's vision to use constructive mathematics as a programming language for exact analysis.

1. Introduction

Integration is one of the fundamental techniques in numerical computation. However, its implementation using floating point numbers requires continuous effort on the part of the user in order to ensure that the results are correct. This burden can be shifted away from the end-user by providing a library of *exact* analysis in which the computer handles the error estimates. For high assurance we use computer verified proofs that the implementation is actually correct; see [GNSW07] for an overview. It has long been suggested that by using *constructive mathematics* exact analysis and provable correctness can be unified [Bis67, Bis70]. Constructive mathematics provides a high level framework for specifying computations (Section 2.1). However, Bishop [Bis67] p.357 writes:

> *As written, this book is person-oriented rather than computer-oriented. It would be of great interest to have a computer-oriented version. Without such a version, it is hard to predict with any confidence what form computer-oriented abstract analysis will eventually assume. A thoughtful computer-oriented presentation should uncover many interesting phenomena.*

Our aim is to provide such a presentation for Riemann integration. In fact, we provide much more. We provide an implementation in dependent type theory (Section 2.2). Type theory is a formal framework for constructive mathematics [ML98, ML82, NPS90]. It supports the development of formal proofs, while, at the same time, being an efficient functional programming language with a dependent type system. We use the Coq [Tea08, BC04] proof assistant, which is an implementations of the Calculus of Inductive Constructions (CIC) [CH88, CP90]. However, we believe that the ideas presented in this paper are general enough to easily be developed in other implementations of type theory, such as Martin-Löf type theory[1] [ML98, ML82, NPS90], so our presentation is mostly done in a type-theoretic agnostic way.

Coq includes a compiler [GL02] based on OCaml's virtual machine to allow efficient evaluation[2]. As a feasibility study, we have implemented Riemann integration. Our implementation is functional and structured in a monadic way. This structure greatly simplifies the integrated development of the program together with its correctness proof.

In constructive analysis one approximates real numbers by rational, or dyadic numbers. Rational numbers, as opposed to the real numbers, can be represented exactly in a computer. The real numbers are the completion of the rationals. The completion construction can be organized in a monad, a familiar construct from functional programming (Section 2.8). This completion monad provides an efficient combination of proving and computing [O'C07]. In this paper, we use a similar technique: the integrable functions are in the completion of rational step functions (Section 3.1), and the same monadic implementation is reused.

Our contributions include:

- We show that the step functions form a monad itself (Section 3.2) that distributes over the completion monad (Section 3.9).
- Using the applicative functor interface of the step function monad we lift functions and relations to step functions (Section 3.3).
- Using combinators we also lift theorems to reason about these functions and relations on step functions (Section 4.6).
- We define both L^1 and L^∞ metrics on step functions (Section 3.5) and define integration on the completion of the L^1 space (Section 3.6).
- We show how to embed uniformly continuous functions into this space in order to integrate them (Section 3.7).
- We extend our definition of Riemann integral to a Stieltjes integral (Section 3.8).

1.1. Notation

We will use traditional notation from functional programming for this paper. Thus fx will represent function application. We will typically use curried functions, so fxy will represent $(fx)y$, and f will have type $X \Rightarrow Y \Rightarrow Z$ (meaning $X \Rightarrow (Y \Rightarrow Z)$).

[1]In particular we do not believe that we make any essential use of impredicativity of propositions in Coq.

[2]We copy the conclusions from the benchmarks carried out in [GL02]:'...our reducer runs about as fast as OCaml's bytecode interpreter; the speed ratio varies between 1.4 and 0.95. Compiling the extracted Caml code with the OCaml native-code compiler results in speed ratios between 3.5 and 5.6, which is typical of the speed-ups obtained by going from bytecode interpretation to native-code generation.'

We will mostly gloss over details about equivalence relations for types. We will use $\asymp$ to represent the equivalence relation to be used with the types in question. We will use :=for defining functions and constants.

We denote the type of the closed unit interval as $[0, 1]$, and $]0, 1[$ will be the type of the open interval. We denote the the open interval restricted to the rational numbers by $]0, 1[_{\mathbb{Q}}$.

2. Background

2.1. Constructive mathematics and type theory

We wish to use constructive reasoning because constructive proofs have a computational interpretation. For example, a constructive proof of $\varphi \vee \psi$ tells which of the two disjuncts hold. A proof of $\exists n : \mathbb{N}.Pn$ gives an explicit value for n that makes Pn hold. Most importantly, we have a functional interpretation of $\Rightarrow$ and $\forall$. A proof of $\forall n : \mathbb{N}.\exists m : \mathbb{N}.Rnm$ is interpreted as a function with an argument n that returns an m paired with a proof of Rnm. A proof of $\neg\varphi$, which is equal to $\varphi \Rightarrow \bot$ by definition, is a function taking an arbitrary proof of φ to a proof of $\bot$ (false)—which means there should not be any proofs of φ.

The connectives in constructive logic come equipped with their constructive rules of inference (given by natural deduction)[SU98]. Excluded middle ($\varphi \vee \neg\varphi$) cannot be deduced in general, and proof by contradiction, $\neg\neg\varphi \Rightarrow \varphi$, is also not provable in general.

2.2. Dependently typed functional programming

The functional interpretation of constructive deductions is given by the Curry-Howard isomorphism [SU98]. This isomorphism associates formulas with dependent types, and proofs of formulas with functional programs of the associated dependent types. For example, the identity function $\lambda x : A.x$ of type $A \Rightarrow A$ represents a proof of the tautology $A \Rightarrow A$. Table 1 lists the association between logical connectives and type constructors.

Logical Connective	**Type Constructor**
implication: $\Rightarrow$	function type: $\Rightarrow$
conjunction: $\wedge$	product type: $\times$
disjunction: $\vee$	disjoint union type: $+$
true: $\top$	unit type: $()$
false: $\bot$	void type: $\emptyset$
for all: $\forall x.Px$	dependent function type: $\Pi x.Px$
exists: $\exists x.Px$	dependent pair type: $\Sigma x.Px$

Table 1: The association between formulas and types given by the Curry-Howard isomorphism.

In dependent type theory, functions from values to types are allowed. Using types parametrized by values, one can create dependent pair types, $\Sigma x : A.Px$, and dependent function types, $\Pi x : A.Px$. A dependent pair consists of a value

x of type A and a value of type Px. The type of the second value depends on the first value, x. A dependent function is a function from the type A to the type Px. The type of the result depends on the value of the input.

The association between logical connectives and types can be carried over to constructive mathematics. We associate mathematical structures, such as the natural numbers, with inductive types in functional programming languages. We associate atomic formulas with functions returning types. For example, we can define equality on the natural numbers, $x =_{\mathbb{N}} y$, as a recursive function:

$$\begin{array}{rcl} 0 =_{\mathbb{N}} 0 & := & \top \\ Sx =_{\mathbb{N}} 0 & := & \bot \\ 0 =_{\mathbb{N}} Sy & := & \bot \\ Sx =_{\mathbb{N}} Sy & := & x =_{\mathbb{N}} y \end{array}$$

One catch is that general recursion is not allowed when creating functions. The problem is that general recursion allows one to create a fixed-point operator, $\mathsf{fix} : (\varphi \Rightarrow \varphi) \Rightarrow \varphi$, that corresponds to a proof of a logical inconsistency. To prevent this, we allow only well-founded recursion over an argument with an inductive type. Because well-founded recursion ensures that functions always terminate, the language is not Turing complete. However, one can still express fast-growing functions, such as the Ackermann function, without difficulty by using higher-order functions [Tho91].

Because proofs and programs are written in the same language, we can freely mix the two. For example, in previous work [O'C07], the real numbers are presented by the type

$$\exists f : \mathbb{Q}^+ \Rightarrow \mathbb{Q}. \forall \varepsilon_1 \varepsilon_2. |f\varepsilon_1 - f\varepsilon_2| \le \varepsilon_1 + \varepsilon_2. \tag{1}$$

A value of this type is a pair of a function $f : \mathbb{Q}^+ \Rightarrow \mathbb{Q}$ and a proof of $\forall \varepsilon_1 \varepsilon_2. |f\varepsilon_1 - f\varepsilon_2| \le \varepsilon_1 + \varepsilon_2$. The idea is that a real number is represented by a function f that maps any requested precision $\varepsilon : \mathbb{Q}^+$ to a rational approximation of the real number. Not every function of type $\mathbb{Q}^+ \Rightarrow \mathbb{Q}$ represents a real number. Only those functions that have coherent approximations should be allowed. The proof object paired with f witnesses the fact that f has coherent approximations. This is one example of how mixing functions and formulas allows one to create precise data-types.

2.3. Extensional Equality

In this paper, we will use the equality sign $(=)$ for **extensional equality**. Two functions f, g of the same type are considered extensionally equal when, for any input given to both functions, the outputs of the functions are extensionally equal:

$$f = g := \forall a. f(a) = g(a).$$

Two values of an inductive type are extensionally equal when their constructors are the same and all parameters are extensionally equal.

Extensional equality is the finest equality we will need. However, `Coq` uses a finer equality called intensional equality for its fundamental equality.

Another sort of equality that we will frequently use is setoid equality (see Section 2.4), which is generally coarser than extensional equality.

2.4. Setoids Instead of Quotients

A quotient type is a type modulo a given equivalence relation on that type. For instance, the type $\mathbb{Q}$ is often considered as a quotient of the type $\mathbb{Z} \times \mathbb{N}^+$. Coq does not have quotient types. One reason for this is that it would destroy the decidability of type checking. One instead passes around the equivalence relation in question. To do this, one often uses a data structure called a setoid, or a Bishop set [Bis67, Hof97, BCP03]. A setoid $(A, \asymp_A)$ is a type paired with an equivalence relation on that type. Functions between setoids that preserve their equivalence relations are called **respectful**. Proving that a function is respectful consists of the same work in traditional mathematics needed to prove that a function over quotients is well-defined. Respectful functions are also called **morphisms**.

2.4.1. Rewrite Automation

Coq supports reasoning about setoids through its tactics `setoid_rewrite` and `setoid_replace` [Coe04]. These tactics will automatically create the deductions for substitution of setoid equivalent terms into respectful functions and relations. This support makes reasoning about setoid equivalence almost as easy as reasoning about equality in Coq.

Furthermore, Coq has the ability to define a database of rewrite lemmas. These lemmas have terms of the form $a \asymp_A b$ for their conclusions. When they are added to the database the user indicates which way substitution should be performed (the same lemma can be added to different databases with different directions). The user can then use the database as a rewrite system to process a hypothesis or goal. The `autorewrite` *`<database>`* tactic will repeatedly try to use the lemmas in the named database to rewrite the goal. Well crafted rewrite databases can be used to quickly transform or simplify expressions.

2.5. Metric spaces

Traditionally, a metric space is defined as a set X with a metric function $d : X \times X \Rightarrow \mathbb{R}^{0+}$ satisfying certain axioms. The usual constructive formulation requires d be a computable function. In previous work [O'C07], it was useful to take a more relaxed definition for a metric space that does not require the metric be a function. A similar construction can be found in the work by Richman [Ric08]. Instead, the metric is represented via a (respectful) ball relation $\boldsymbol{B} : \mathbb{Q}^+ \Rightarrow X \Rightarrow X \Rightarrow \star$, where $\star$ is the type of propositions, satisfying five axioms:

1. $\forall x \varepsilon . \boldsymbol{B}_\varepsilon x x$
2. $\forall x y \varepsilon . \boldsymbol{B}_\varepsilon x y \Rightarrow \boldsymbol{B}_\varepsilon y x$

3. $\forall xyz\varepsilon_1\varepsilon_2 . \boldsymbol{B}_{\varepsilon_1} xy \Rightarrow \boldsymbol{B}_{\varepsilon_2} yz \Rightarrow \boldsymbol{B}_{\varepsilon_1+\varepsilon_2} xz$
4. $\forall xy\varepsilon . (\forall \delta . \varepsilon < \delta \Rightarrow \boldsymbol{B}_{\delta} xy) \Rightarrow \boldsymbol{B}_{\varepsilon} xy$
5. $\forall xy . (\forall \varepsilon . \boldsymbol{B}_{\varepsilon} xy) \Rightarrow x \asymp y$

The ball relation $\boldsymbol{B}_{\varepsilon} xy$ expresses that the points x and y are within ε of each other. We call this a ball relationship because the partially applied relation $\boldsymbol{B}^X_{\varepsilon} x : X \Rightarrow \star$ is a predicate that represents the closed ball of radius ε around the point x.

For example, $\mathbb{Q}$ can be equipped with the usual metric by defining the ball relation as

$$\boldsymbol{B}^{\mathbb{Q}}_{\varepsilon} xy := |x - y| \leq \varepsilon .$$

This definition satisfies all the required axioms.

2.6. Uniform continuity

We are interested in the category of metric spaces with uniformly continuous functions between them. A function $f : X \Rightarrow Y$ between two metric spaces is **uniformly continuous with modulus** $\mu_f : \mathbb{Q}^+ \Rightarrow \mathbb{Q}^+$ if

$$\forall x_1 x_2 \varepsilon . \boldsymbol{B}^X_{\mu_f \varepsilon} x_1 x_2 \Rightarrow \boldsymbol{B}^Y_{\varepsilon} (f x_1)(f x_2).$$

A function is **uniformly continuous** if it is uniformly continuous with some modulus. We use the notation $X \to Y$ with a single bar arrow to denote the type of uniformly continuous functions from X to Y. This record type consists of three parts, a function f of type $X \Rightarrow Y$, a modulus of continuity, and a proof that f is uniformly continuous with the given modulus. We will leave the projection to the function type implicit and allow us to write fx when $f : X \to Y$ and $x : X$. Our definition of uniform continuity implies that the function is respectful.

2.7. Monads

Moggi [Mog89] recognized that many non-standard forms of computation may be modeled by monads[3]. Wadler [Wad92a] popularized their use in functional programming. Monads are now an established tool to structure computation with side-effects. For instance, programs with input X and output Y which have access to a mutable state S can be modeled as functions of type $X \times S \Rightarrow Y \times S$, or equivalently $X \Rightarrow (Y \times S)^S$. The type constructor $\mathfrak{M}Y := (Y \times S)^S$ is an example of a monad. Similarly, partial functions may be modeled by maps $X \Rightarrow Y_\perp$, where $Y_\perp := Y + ()$ is a monad. The reader monad, $\mathfrak{M}Y := Y^E$, for passing an environment implicitly will play an important role in this paper.

[3]In category theory one would speak about the Kleisli category of a (strong) monad.

The formal definition of a (strong) monad is a triple $(\mathfrak{M}, \mathsf{return}, \mathsf{bind})$ consisting of a type constructor $\mathfrak{M}$ and two functions:

$$\begin{aligned}\mathsf{return} &: X \Rightarrow \mathfrak{M}X \\ \mathsf{bind} &: (X \Rightarrow \mathfrak{M}Y) \Rightarrow \mathfrak{M}X \Rightarrow \mathfrak{M}Y\end{aligned}$$

We will denote $(\mathsf{return}\, x)$ as $\hat{x}$, and $(\mathsf{bind}\, f)$ as $\check{f}$. These two operations must satisfy the following laws:

$$\begin{aligned}\mathsf{bind}\ \mathsf{return}\ a &\asymp a \\ \check{f}\hat{a} &\asymp fa \\ \check{f}(\check{g}a) &\asymp \mathsf{bind}(\check{f} \circ g)a\end{aligned}$$

Alternatively, we can define a (strong) monad using three functions:

$$\begin{aligned}\mathsf{return} &: X \Rightarrow \mathfrak{M}X \\ \mathsf{map} &: (X \Rightarrow Y) \Rightarrow (\mathfrak{M}X \Rightarrow \mathfrak{M}Y) \\ \mathsf{join} &: \mathfrak{M}(\mathfrak{M}X) \Rightarrow \mathfrak{M}X\end{aligned}$$

satisfying certain laws. These can be obtained from the previous presentation of a monad by defining

$$\begin{aligned}\mathsf{map}\, fm &:= \mathsf{bind}(\mathsf{return} \circ f)m \\ \mathsf{join}\, m &:= \check{\mathbf{I}}m.\end{aligned}$$

where $\mathbf{I}$ is the identity function. Conversely, given the $(\mathsf{return}, \mathsf{map}, \mathsf{join})$ presentation we define

$$\mathsf{bind}\, f := \mathsf{join} \circ (\mathsf{map}\, f).$$

2.8. Completion monad

The first monad that we will meet in this paper is O'Connor's completion monad $\mathfrak{C}$ [O'C07]. Given a metric space X, the completion of X is defined by

$$\mathfrak{C}X := \exists f : \mathbb{Q}^+ \Rightarrow X.\forall \varepsilon_1 \varepsilon_2.\boldsymbol{B}^X_{\varepsilon_1+\varepsilon_2}(f\varepsilon_1)(f\varepsilon_2).$$

The real numbers defined as the completion, $\mathbb{R} := \mathfrak{C}\mathbb{Q}$, is exactly the type given in equation 1.

The function $\mathsf{return} : X \to \mathfrak{C}X$ is the embedding of a metric space in its completion. The function $\mathsf{join} : \mathfrak{C}(\mathfrak{C}X) \to \mathfrak{C}X$ is half of this isomorphism between $\mathfrak{C}(\mathfrak{C}X)$ and $\mathfrak{C}X$ (with return being the other half). Finally, a uniformly continuous function $f : X \to Y$ can be lifted to operate on complete metric spaces, $\mathsf{map}\, f : \mathfrak{C}X \to \mathfrak{C}Y$. Uniformly continuity is essential in this definition of map. This means that $\mathfrak{C}$ is a monad on the category of metric spaces with uniformly continuous functions. One advantage of this approach is that it helps us to work with simple representations. To specify a function from $\mathbb{R} \to \mathbb{R}$, one can simply define a uniformly continuous function $f : \mathbb{Q} \to \mathbb{R}$, and then $\check{f} : \mathbb{R} \to \mathbb{R}$ is the required function. Hence, the completion monad allows us to do in a structured way what was already folklore in constructive mathematics: to work with simple, often decidable, approximations to continuous objects; see e.g. [Sch08].

3. Informal Presentation of Riemann Integration

In this section, we present our work in informal constructive mathematics. Everything presented here has been formalized in Coq, except where otherwise noted.

We will implement Riemann integration as follows:

1. Define step functions;
2. Introduce applicative functors and show that step functions form an applicative functor;
3. Show that the step functions form a metric space under both the L^1 and L^∞ norms;
4. Define integrable functions as the completion of the step functions under the L^1 norm;
5. Define integration first on step functions and lift it to operate on integrable functions;
6. Define an injection from the uniformly continuous functions to the integrable functions in order to integrate them.

At the end, we will see that it is natural to generalize our Riemann integral to a Stieltjes integral.

3.1. Step functions

Our first goal will be to define (formal) step functions and some important operations on them. For any type X, we first define the inductive data type of (rational) step functions from the unit interval to X, denoted by $\mathfrak{S}X$. A step function is either a constant function, $\mathsf{const}\, x$, for some $x : X$, or two step functions, $f : \mathfrak{S}X$ and $g : \mathfrak{S}X$ glued at a point in o, $\mathsf{glue}\, o f g$, where o must be a rational number strictly between 0 and 1. We will sometimes write $(\mathsf{const}\, x)$ as $\hat{x}$, and $(\mathsf{glue}\, o f g)$ as $f \rhd o \lhd g$.

Definition 1. *The rules for constructing the inductive data type $\mathfrak{S}$:*

$$\frac{x : X}{\mathsf{const}\, x : \mathfrak{S}(X)} \qquad \frac{o : (0,1)_{\mathbb{Q}} \quad f : \mathfrak{S}(X) \quad g : \mathfrak{S}(X)}{f \rhd o \lhd g : \mathfrak{S}(X)}$$

The elements of this inductive type are intended to be interpreted as step functions on $[0, 1]$. The interpretation of $\hat{x}$ is the constant function on $[0, 1]$ returning x. The interpretation of $f \rhd o \lhd g$ is f squeezed into the interval $[0, o]$ and g squeezed into the interval $[o, 1]$. In this sense f and g are "glued" together.

Even though we call step functions "functions", they are not really functions, and we never formally interpret them as functions. They are a formal structure that takes the place of step functions from classical mathematics. It does not

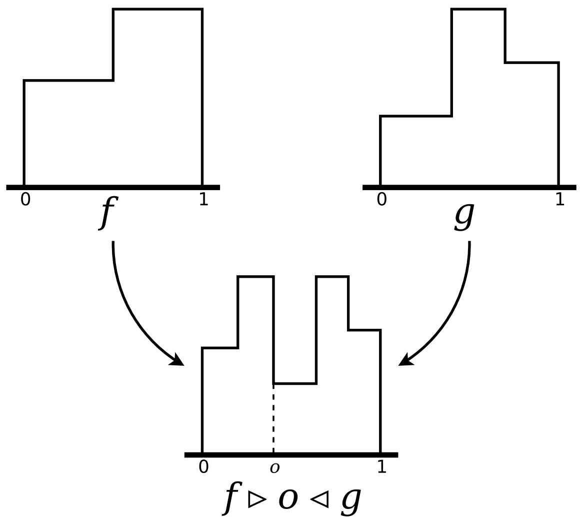

Figure 1: Given two step functions f and g, the step function $f \rhd o \lhd g$ is f squeezed into $[0, o]$ and g squeezed into $[o, 1]$.

matter that our informal interpretation of $f \rhd o \lhd g$ is not well defined at o, because the step functions are intended for integration, not for evaluation at a point.

One can see that this inductive type is a binary tree whose nodes hold data of type $(0,1)_{\mathbb{Q}}$, and whose leaves have type X. We work with an equivalence relation on this binary tree structure that identifies different ways of constructing the same step function. Informally, this is the equivalence relation induced by our interpretation; the formal equivalence relation is defined in Section 3.4.

We define two sorts of inverses to glue which we call left-split and right-split. Given $f : \mathfrak{S}X$ and $a : (0,1)_{\mathbb{Q}}$ we define left-split (written as $f \blacktriangleright a : \mathfrak{S}X$) and right-split (written as $a \blacktriangleleft f : \mathfrak{S}X$) as follows:

Definition 2.

$$\begin{aligned}
\widehat{x} \blacktriangleright a &:= \widehat{x} \\
(f_l \rhd o \lhd f_r) \blacktriangleright a &:= \begin{cases} f_l \blacktriangleright \frac{a}{o} & \text{(if } a < o\text{)} \\ f_l & \text{(if } a = o\text{)} \\ f_l \rhd \frac{o}{a} \lhd (f_r \blacktriangleright \frac{a-o}{1-o}) & \text{(if } a > o\text{)} \end{cases} \\
a \blacktriangleleft \widehat{x} &:= \widehat{x} \\
a \blacktriangleleft (f_l \rhd o \lhd f_r) &:= \begin{cases} (\frac{a}{o} \blacktriangleleft f_l) \rhd \frac{o-a}{1-a} \lhd f_r & \text{(if } a < o\text{)} \\ f_r & \text{(if } a = o\text{)} \\ \frac{a-o}{1-o} \blacktriangleleft f_r & \text{(if } a > o\text{).} \end{cases}
\end{aligned}$$

Informally, the left split $(f \blacktriangleright a)$ takes the portion of f on the interval $[0, a]$ and scales it up to the full interval $[0, 1]$. The right split $(a \blacktriangleleft f)$ does the same thing for the portion of f on the interval $[a, 1]$. We have that

$$(f \blacktriangleright a) \rhd a \lhd (a \blacktriangleleft f) \asymp f$$

holds, which means that gluing back the left and right pieces of a step function split at a returns an equivalent function back. However, this process does not generally return an identical representation. The formal definition of the equivalence relation is defined later in Section 3.4.

The inductive type for step functions has an associated catamorphism which we call fold.

Definition 3.

$$\begin{array}{rcl} \mathsf{fold} & : & (X \Rightarrow Y) \Rightarrow ((0,1)_{\mathbb{Q}} \Rightarrow Y \Rightarrow Y \Rightarrow Y) \Rightarrow \mathfrak{S}X \Rightarrow Y \\ \mathsf{fold}\,\varphi\psi\hat{x} & := & \varphi x \\ \mathsf{fold}\,\varphi\psi(f \rhd o \lhd g) & := & \psi o(\mathsf{fold}\,\varphi\psi f)(\mathsf{fold}\,\varphi\psi g). \end{array}$$

This fold operation is used in many places. For instance, it is used to define two metrics on step functions (Section 3.5) or to check whether a property holds globally on $[0, 1]$ (Section 3.4). Not every fold respects the equivalence relation on step functions, so we need to prove that each fold instance we use respects the equivalence relation.

3.2. Step functions form a monad

The step function type constructor $\mathfrak{S}$ forms a monad similar to the reader monad $\lambda X.X^{[0,1]}$ [Wad92b]. The return of $\mathfrak{S}$ is the constant function, map is defined in the obvious way using fold, and the join from $\mathfrak{S}(\mathfrak{S}X)$ to $\mathfrak{S}X$ is the formal variant of the join function from the reader monad, $\mathsf{join}\,fz := fzz$, which considers a step function of step functions as a step function of two inputs and returns the step function of its diagonal:

Definition 4.

$$\begin{array}{rcl} \mathsf{join}\,\widehat{f} & := & f \\ \mathsf{join}\,(f \rhd o \lhd g) & := & \mathsf{join}(\mathsf{map}(\lambda x.x \blacktriangleright o)f) \rhd o \lhd \mathsf{join}(\mathsf{map}(\lambda x.o \blacktriangleleft x)g). \end{array}$$

Rather than use these monadic functions, we use the applicative functor interface to this monad.

3.3. Applicative functors

Let $\mathfrak{M}$ be a strong monad. To lift a function $f : X \Rightarrow Y$ to a function $\mathfrak{M}X \Rightarrow \mathfrak{M}Y$, we use $\mathsf{map} : (X \Rightarrow Y) \Rightarrow \mathfrak{M}X \Rightarrow \mathfrak{M}Y$. Lifting a function with two curried arguments is possible using a similar function $\mathsf{map2}$. However, to avoid having to write a function $\mathsf{map}\,n$ for each natural number n, one can use the theory of applicative functors. An consists of a type constructor $\mathfrak{T}$ and two functions:

$$\begin{array}{rcl} \mathsf{pure} & : & X \Rightarrow \mathfrak{T}X \\ \mathsf{ap} & : & \mathfrak{T}(X \Rightarrow Y) \Rightarrow \mathfrak{T}X \Rightarrow \mathfrak{T}Y \end{array}$$

The function pure lifts any value inside the functor. The ap function applies a function inside the functor to a value inside the functor to produce a value inside the functor. We denote $(\mathsf{pure}\, x)$ by $\widehat{x}$, as was done for monads, and we denote $(\mathsf{ap}\, f x)$ by $f@x$. An applicative functor must satisfy the following laws [MP08]:

$$\begin{array}{ll} \widehat{\mathbf{I}}@v \asymp v & \text{Identity} \\ \widehat{\mathbf{B}}@u@v@w \asymp u@(v@w) & \text{Composition} \\ \widehat{f}@\widehat{x} \asymp \widehat{fx} & \text{Homomorphism} \\ u@\widehat{y} \asymp \widehat{\mathrm{ev}_y}@u & \text{Interchange} \end{array}$$

Where $\mathbf{B}$ and $\mathbf{I}$ are the composition and identity combinators respectively (see Section 4.5) and $\mathrm{ev}_y := \lambda f. f y$ is the function which evaluates at y.

Every strong monad induces the canonical applicative functor [MP08] where

$$\begin{array}{rcl} \mathsf{pure} & := & \mathsf{return} \\ f@x & := & \mathsf{bind}(\lambda g.\, \mathsf{map}\, g x) f. \end{array}$$

As the name suggests, every applicative functor can be seen as a functor. Given an applicative functor $\mathfrak{T}$, we define $\mathsf{map} : (X \Rightarrow Y) \Rightarrow \mathfrak{T}X \Rightarrow \mathfrak{T}Y$ as

$$\mathsf{map}\, f x := \hat{f}@x.$$

When $\mathfrak{T}$ is generated from a monad, this definition of map is equivalent to the definition of map associated with the monad.

3.4. The step function applicative functor

The ap function for step functions $\mathfrak{S}$ applies a step function of functions to a step function of argument pointwise. It is formally defined as follows:

Definition 5.

$$\begin{array}{rcl} \widehat{f}@\widehat{x} & := & \widehat{f(x)} \\ \widehat{f}@(x_l \rhd o \lhd x_r) & := & (\widehat{f}@x_l) \rhd o \lhd (\widehat{f}@x_r) \\ (f_l \rhd o \lhd f_r)@x & := & (f_l@(x \blacktriangleright o)) \rhd o \lhd (f_r@(o \blacktriangleleft x)). \end{array}$$

For step functions $\mathfrak{S}$, we denote $(\mathsf{map}\, f x)$ by $f \circ\!\!\!\nearrow x$. This notation is meant to suggest the similarity with the composition operation, which is the definition of map for the reader monad $\lambda X. X^{[0,1]}$.

Definition 6. *The binary version of* map *is defined in terms of* map *and* ap.

$$\mathsf{map2}\, f a b := f \circ\!\!\!\nearrow a@b.$$

Higher arity maps can be defined in a similar way; however, we found it more natural to simply use map and ap everywhere.

We will often use map2 to lift infix operations. Because of this, we give it a special notation.

Definition 7. *If $\circledast$ is some infix operator such that $\lambda xy.x \circledast y : X \Rightarrow Y \Rightarrow Z$, then we define*

$$f \langle \circledast \rangle \, g := (\lambda xy.x \circledast y) ♂ f @ g,$$

where $f : \mathfrak{S}X$, $g : \mathfrak{S}Y$, and $f \langle \circledast \rangle \, g : \mathfrak{S}Z$.

For example, if $f, g : \mathfrak{S}\mathbb{Q}$ are rational step functions, then $f \langle - \rangle \, g$ is the pointwise difference between f and g as a rational step function.

We can lift relations to step functions as well. A relation is simply a function to $\star$, the type of propositions. Thus a binary relation $\propto$ has a type $\lambda xy.x \propto y : X \Rightarrow Y \Rightarrow \star$. If we use map2, we end up with an function $\lambda fg.f \langle \propto \rangle \, g : \mathfrak{S}X \Rightarrow \mathfrak{S}Y \Rightarrow \mathfrak{S}\star$. The result is not a proposition, but rather a step function of propositions. Classically, this corresponds to a step function of Booleans. In other words, $\mathfrak{S}\star$ represents a type of step characteristic functions on $[0, 1]$.

Each way of turning a characteristic function into a proposition determines a different kind of predicate lifting [Sch05]. For our purposes, we are interested in the one that asks the characteristic function to hold everywhere. The function $\mathsf{fold}_\star : \mathfrak{S}\star \Rightarrow \star$ does this by folding conjunction over a step function.

Definition 8. $\mathsf{fold}_\star := \mathsf{fold}(\mathbf{I}, \lambda opq.p \wedge q)$.

When this function is composed with map2, the result lifts a relation to a relation on step functions.

Definition 9. $f \, \{\propto\} \, g := \mathsf{fold}_\star(f \langle \propto \rangle \, g)$.

For example, we define equivalence on step functions by lifting the equivalence relation on X.

Definition 10. $f \asymp_{\mathfrak{S}X} g := f \, \{\asymp_X\} \, g$.

Two step functions are equivalent if they are pointwise equivalent everywhere. Similarly, we define a partial order on step functions by lifting the inequality relation on $\mathbb{Q}$.

Definition 11. $f \le_{\mathfrak{S}\mathbb{Q}} g := f\{\le_{\mathbb{Q}}\}g$.

A step function f is less than a step function g if f is pointwise less than g everywhere.

3.5. Two metrics for step functions

The step functions over the rational numbers, $\mathfrak{S}\mathbb{Q}$, form a metric space in two ways, with the L^∞ metric and the L^1 metric. We first define the two norms on the step functions.

Definition 12.

$$\begin{aligned}\|f\|_\infty &:= \mathsf{fold}_{\mathrm{sup}}(\mathsf{abs} \,\circ\!\!\nearrow f)\\ \|f\|_1 &:= \mathsf{fold}_{\mathrm{affine}}(\mathsf{abs} \,\circ\!\!\nearrow f)\end{aligned}$$

where

$$\begin{aligned}\mathsf{fold}_{\mathrm{sup}} &:= \mathsf{fold}\,\mathbf{I}(\lambda oxy.\mathrm{max} xy)\\ \mathsf{fold}_{\mathrm{affine}} &:= \mathsf{fold}\,\mathbf{I}(\lambda oxy.ox + (1-o)y)\end{aligned}$$

and $\mathsf{abs} : \mathbb{Q} \Rightarrow \mathbb{Q}$ *is the absolute value function on* $\mathbb{Q}$*.*

The function $\mathsf{fold}_{\mathrm{sup}} : \mathfrak{S}\mathbb{Q} \Rightarrow \mathbb{Q}$ returns the supremum of the step function, while the function $\mathsf{fold}_{\mathrm{affine}} : \mathfrak{S}\mathbb{Q} \Rightarrow \mathbb{Q}$ returns the integral of a step function.

Next, the metric distance between two step functions is defined.

Definition 13.

$$\begin{aligned}d^\infty fg &:= \|f \langle - \rangle g\|_\infty\\ d^1 fg &:= \|f \langle - \rangle g\|_1.\end{aligned}$$

Finally, the distance relations are defined in terms of the distance functions.

Definition 14.

$$\begin{aligned}\boldsymbol{B}_\varepsilon^{\mathfrak{S}^\infty\mathbb{Q}} fg &:= d^\infty fg \leq \varepsilon\\ \boldsymbol{B}_\varepsilon^{\mathfrak{S}^1\mathbb{Q}} fg &:= d^1 fg \leq \varepsilon.\end{aligned}$$

When we need to be clear which metric space is being used, we will use the notation $\mathfrak{S}^\infty\mathbb{Q}$ or $\mathfrak{S}^1\mathbb{Q}$.

The two fold functions defined in this section are uniformly continuous for their respective metrics.

$$\begin{aligned}\mathsf{fold}_{\mathrm{sup}} &: \mathfrak{S}^\infty\mathbb{Q} \to \mathbb{Q}\\ \mathsf{fold}_{\mathrm{affine}} &: \mathfrak{S}^1\mathbb{Q} \to \mathbb{Q}\end{aligned}$$

The identity function is uniformly continuous in one direction, $\iota : \mathfrak{S}^\infty\mathbb{Q} \to \mathfrak{S}^1\mathbb{Q}$; however, the other direction is not uniformly continuous.

The metrics $\mathfrak{S}^\infty X$ and $\mathfrak{S}^1 X$ can be defined for any metric space X:

$$\begin{aligned}\boldsymbol{B}_\varepsilon^{\mathfrak{S}^\infty X} fg &:= \mathsf{fold}_\star(\boldsymbol{B}_\varepsilon^X \,\circ\!\!\nearrow f @ g)\\ \boldsymbol{B}_\varepsilon^{\mathfrak{S}^1 X} fg &:= \exists h : \mathfrak{S}\mathbb{Q}^+.\, \mathsf{fold}_\star(\boldsymbol{B}^X \,\circ\!\!\nearrow h @ f @ g) \wedge \|h\|_1 \leqslant \varepsilon\end{aligned}$$

We have implemented the generic $\mathfrak{S}^\infty X$ metric in our formalization. However, for the L^1 space, we have only implemented the specific $\mathfrak{S}^1\mathbb{Q}$ metric.

3.6. Integrable functions and bounded functions

The bounded functions and the integrable functions are defined as the completion of the step functions under the L^∞ and the L^1 metrics respectively.

Definition 15.

$$\begin{aligned} \mathfrak{B} &:= \mathfrak{C} \circ \mathfrak{S}^\infty \\ \mathfrak{I} &:= \mathfrak{C} \circ \mathfrak{S}^1. \end{aligned}$$

In Section 3.1, we informally interpreted elements of $\mathfrak{S}X$ as (partially defined) functions on $[0,1]$. Similarly, we can informally interpret each bounded function as a (partially defined) function. Consider $f : \mathfrak{B}\mathbb{Q}$. Define $g_n := f\left(\frac{1}{n}\right)$. Then $\lim_{n\to\infty} g_n(x)$ exists for all points x in $[0,1]$ except perhaps for the (rational) splitting points of the step functions g_n. At the points where this limit is defined, it is (classically) continuous.

To every Riemann integrable function on $[0,1]$ we can associate an element in $\mathfrak{I}\mathbb{Q}$. Moreover, functions f and g such that $\int |f-g| \asymp 0$ will be assigned to equivalent elements in $\mathfrak{I}\mathbb{Q}$. This definition can be extended to every *generalized* Riemann integrable function, where a function h is generalized Riemann integrable if $h_n := \max\left(\min h\hat{n}\right)\left(\widehat{-n}\right)$ is integrable for each n and the limit of$\int h_n$ converges (even though h_n may not converge pointwise everywhere). Conversely, we can informally interpret every element f of $\mathfrak{I}\mathbb{Q}$ as a generalized Riemann integrable function. Define g_n as the sequence

$$g_n := f\left(\frac{1}{2^{2n+1}}\right).$$

By the fundamental lemma of integration [Lan93], g_n converges pointwise almost everywhere. Let g be this pointwise limit. Then g is a generalized Riemann integrable function associated with f.

The bounded functions have a supremum operation, $\mathsf{sup} : \mathfrak{B}\mathbb{Q} \to \mathbb{R}$ and, similarly, the integrable functions have an integration operation, $\int : \mathfrak{I}\mathbb{Q} \to \mathbb{R}$ which are defined by lifting the two folds from the previous section.

Definition 16.

$$\begin{aligned} \mathsf{sup}\, f &:= \mathsf{map}_{\mathfrak{C}}\, \mathsf{fold}_{\mathsf{sup}}\, f \\ \int f &:= \mathsf{map}_{\mathfrak{C}}\, \mathsf{fold}_{\text{affine}}\, f \end{aligned}$$

There is an injection from the bounded functions into the integrable functions defined by lifting the injection on step functions: $\mathsf{map}\, \iota : \mathfrak{B}\mathbb{Q} \to \mathfrak{I}\mathbb{Q}$. However, there is no injection from integrable functions to bounded functions. Thus bounded functions can be integrated, but integrable functions may not have a supremum.

3.7. Riemann integral

The process for integrating a function is as follows. Given a function f, one needs to find an equivalent representation of f as an integrable function and then this integrable function can be integrated. We will consider how to integrate uniformly continuous functions on $[0, 1]$, which is a useful class of functions to integrate.

We convert a uniformly continuous function to an integrable function by a two step process. First, we will convert it to a bounded function, and then the bounded function can be converted to an integrable function using the injection defined in the previous section.

To produce a bounded function, one needs to create a step function that approximates f within ε for any value $\varepsilon : \mathbb{Q}^+$. The usual way of doing this is to create a step function where each step has width no more than $2\,(\mu_f \varepsilon)$. The value at each step is taken by sampling the function at the center of the step.

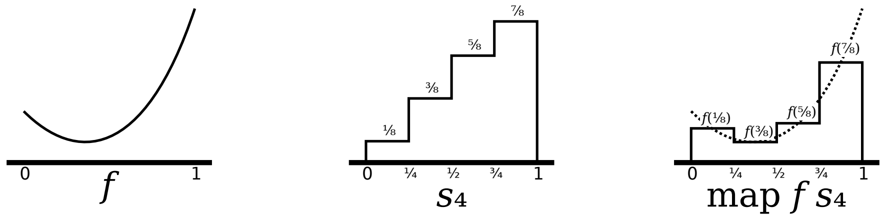

Figure 2: Given a uniformly continuous function f and a step function s_4 that approximates the identity function, the step function $(\mathsf{map}\, f s_4)$ (or $f ♂ s_4$) approximates f in the familiar Riemann way.

When developing the above, it became clear that one can achieve the desired result by creating a step function whose values are the sample inputs, and then mapping f over these "sampling step-functions" (see Figure 3.7). In fact, the limit of these "sampling step-functions" is simply the identity function on $[0, 1]$ represented as a bounded function, $\mathbf{I}_{[0,1]} : \mathfrak{B}\mathbb{Q}$ (see Section 4.7). Given any uniformly continuous function $f : \mathbb{Q} \to \mathbb{Q}$, we can prove that $\mathsf{map}_{\mathfrak{S}^\infty}\, f : \mathfrak{S}^\infty\mathbb{Q} \to \mathfrak{S}^\infty\mathbb{Q}$ is uniformly continuous. We can then lift again to operate on bounded functions, $\mathsf{map}_{\mathfrak{C}}\,(\mathsf{map}_{\mathfrak{S}^\infty}\, f) : \mathfrak{B}\mathbb{Q} \to \mathfrak{B}\mathbb{Q}$. Applying this to $\mathbf{I}_{[0,1]}$ yields f restricted to $[0, 1]$ as a bounded function, which can then be converted to an integrable function and integrated.

Definition 17. $\int_{[0,1]} f := \int \left(\mathsf{map}_{\mathfrak{C}}\, \iota \left(\mathsf{map}_{\mathfrak{C}}\, (\mathsf{map}_{\mathfrak{S}^\infty}\, f)\, \mathbf{I}_{[0,1]}\right)\right)$.

With a small modification, this process will also work for $f : \mathbb{Q} \to \mathbb{R}$. In this case $\mathsf{map}\, f$ has type $\mathfrak{S}\mathbb{Q} \Rightarrow \mathfrak{S}\mathbb{R}$, Fortunately, there is an injection $\mathsf{dist} : \mathfrak{S}\mathbb{R} \Rightarrow \mathfrak{B}\mathbb{Q}$, that interprets a step function of real values as a bounded function (see Definition 20). We can prove that the composition $\mathsf{dist} \circ (\mathsf{map}_{\mathfrak{S}}\, f) : \mathfrak{S}^\infty\mathbb{Q} \to \mathfrak{B}\mathbb{Q}$ is uniformly continuous. Then, proceeding in a similar fashion, this can be lifted with bind and applied to $\mathbf{I}_{[0,1]}$ to yield f restricted to $[0, 1]$ as a bounded function, which can then be integrated.

Definition 18. $\int_{[0,1]} f := \int \left(\mathsf{map}_{\mathfrak{C}}\, \iota \left(\mathsf{bind}_{\mathfrak{C}} \left(\mathsf{dist} \circ (\mathsf{map}_{\mathfrak{S}}\, f)\right) \mathbf{I}_{[0,1]}\right)\right)$.

An arbitrary uniformly continuous function $f : \mathbb{R} \to \mathbb{R}$ can be integrated on $[0,1]$ by integrating $f \circ \mathsf{return}_{\mathfrak{C}} : \mathbb{Q} \to \mathbb{R}$ because the Riemann integral only depends on the value of functions at rational points.

3.8. Stieltjes integral

Given the previous presentation, any bounded function could be used in place of $\mathbf{I}_{[0,1]}$. A natural question arises: what happens when $\mathbf{I}_{[0,1]}$ is replaced by another bounded function, $g : \mathfrak{B}\mathbb{Q}$? An analysis shows that the result is the Stieltjes integral with respect to g^{-1}, when g is non-decreasing.

Definition 19. $\int f \mathrm{d}g^{-1} := \int \left(\mathsf{map}_{\mathfrak{C}}\, \iota \left(\mathsf{bind}_{\mathfrak{C}} \left(\mathsf{dist} \circ (\mathsf{map}_{\mathfrak{S}}\, f)\right) g\right)\right)$.

We never intended to develop the Stieltjes integral; however, it practically falls out of our work for free. This is not quite as general as the Stieltjes integral for three reasons. Because g is defined on $[0,1]$, this means that g^{-1}'s range must go from 0 to 1. Essentially, g^{-1} must be a cumulative distribution function and, hence, g is a quantile function. Secondly, because g is a bounded function, g^{-1} must have compact support (meaning g^{-1} must be 0 to the left of its support and 1 to the right of its support). Thirdly, our bounded functions can only have discontinuities at rational points.

We have tried to allow g to be an arbitrary integrable function (this would remove some of the previous restrictions); however, we have been unable to constructively show that $\mathsf{dist} \circ (\mathsf{map}_{\mathfrak{S}}\, f) : \mathfrak{S}^1\mathbb{Q} \Rightarrow \mathfrak{I}\mathbb{Q}$ is uniformly continuous when f is. We have generated counterexamples where f is uniformly continuous with modulus μ and $\mathsf{dist} \circ (\mathsf{map}_{\mathfrak{S}}\, f)$ is *not* uniformly continuous with modulus μ; however, for our particular counterexamples, $\mathsf{dist} \circ (\mathsf{map}_{\mathfrak{S}}\, f)$ is still uniformly continuous with a different modulus.

Still, our integral should allow one to integrate with respect to some interesting distributions such as the Dirac distribution and the Cantor distribution.

3.9. Distributing monads

The function $\mathsf{dist} : \mathfrak{S}\mathbb{R} \Rightarrow \mathfrak{B}\mathbb{Q}$ combines two monads on metric spaces, $\mathfrak{C}$ and $\mathfrak{S}$. The function dist has type $\mathfrak{S}(\mathfrak{C}\mathbb{Q}) \Rightarrow \mathfrak{C}(\mathfrak{S}\mathbb{Q})$. In general, the composition of two monads $\mathfrak{M} \circ \mathfrak{N}$ forms a monad when there is a distribution function $\mathsf{dist} : \mathfrak{N}(\mathfrak{M}X) \to \mathfrak{M}(\mathfrak{N}X)$ satisfying certain laws [Bec69, BW05]. Below we state the laws in a more familiar function style [JD93]: [4]

$$\begin{aligned}
\mathsf{dist} \circ \mathsf{map}_{\mathfrak{N}}\, (\mathsf{map}_{\mathfrak{M}}\, f) &\asymp \mathsf{map}_{\mathfrak{M}}\, (\mathsf{map}_{\mathfrak{N}}\, f) \circ \mathsf{dist} \\
\mathsf{dist} \circ \mathsf{return}_{\mathfrak{N}} &\asymp \mathsf{map}_{\mathfrak{M}}\, \mathsf{return}_{\mathfrak{N}} \\
\mathsf{dist} \circ \mathsf{map}_{\mathfrak{N}}\, \mathsf{return}_{\mathfrak{M}} &\asymp \mathsf{return}_{\mathfrak{M}} \\
\mathsf{prod} \circ \mathsf{map}_{\mathfrak{N}}\, \mathsf{dorp} &\asymp \mathsf{dorp} \circ \mathsf{prod}
\end{aligned}$$

[4] For the $\mathfrak{S}$ and $\mathfrak{C}$ monads, we formally checked all of these rules apart from the last one which was too tedious; however, the correctness of the integral does not depend on the proofs of these laws.

where

$$\begin{aligned}\mathsf{prod} &:= \mathsf{map}_{\mathfrak{M}}\,\mathsf{join}_{\mathfrak{N}} \circ \mathsf{dist}\\ \mathsf{dorp} &:= \mathsf{join}_{\mathfrak{M}} \circ \mathsf{map}_{\mathfrak{M}}\,\mathsf{dist}\,.\end{aligned}$$

Definition 20. *In our case, the distribution function is defined as*

$$\begin{aligned}\mathsf{dist} &: \mathfrak{S}^{\infty}(\mathfrak{C}X) \to \mathfrak{C}(\mathfrak{S}^{\infty}X)\\ \mathsf{dist}\,f &:= \lambda\varepsilon.\,\mathsf{map}_{\mathfrak{S}^{\infty}}(\lambda x.x\varepsilon)\,f.\end{aligned}$$

The function dist maps a step function f with values in the completion of X to a collection of approximations $f_\varepsilon : \mathfrak{S}^{\infty}X$ to the function f such that for all ε in $\mathbb{Q}^+$, $|f - f_\varepsilon| \le \varepsilon$ “pointwise”.

4. Implementation in Coq

In this section, we treat aspects related to our implementation in Coq.

4.1. Formalization in Coq

Formalizing the previous in Coq is done in a straightforward manner. We interpret $\star$ as `Prop`, the universe of propositions. Thus, for example, the ball relation on rational numbers has type `Qball : Qpos -> Q -> Q -> Prop`.

The metric space structure is packaged up as a dependent record, a Σ-type. This record contains a field for the domain of the metric space, which is a setoid, a ball relation over that domain with a proof that the ball relation respects the equivalence relation of the domain. Lastly the record contains a collection of proofs of the five axioms of a metric space (see Section 2.5) which are themselves packed into their own record type.

The completion monad is a function from the record type of metric spaces to the record type of metric spaces. In Section 2.8 the domain of the completion is given with an existential quantifier. We use Coq’s `Set` based existential quantifier (essentially a Σ-type) to implement this quantifier.

As a rule, we use `Prop` based objects only for types that would (extensionally) have at most one value, these are essentially the Harrop formulas [CFS03]. Thus negative types such as function types/implications whose result type is $\bot$ or $\top$ go into the `Prop` universe, and all other types are put into the `Set` or `Type` universes. We chose to have the ball relation return `Prop` because the closed sets are typically negative predicates.

Step functions are represented by an inductive data type which is effectively a labeled binary tree. The Coq declaration for this structure is the following:

```
Inductive StepF : Type:=
|constStepF : X -> StepF
|glue : OpenUnit -> StepF -> StepF -> StepF
```

```
Record is_MetricSpace (X:Setoid)(B: Qpos -> relation X):Prop :=
{ msp_refl: forall e, reflexive _ (B e)
; msp_sym: forall e, symmetric _ (B e)
; msp_triangle: forall e1 e2 a b c, B e1 a b -> B e2 b c ->
              B (e1 + e2)%Qpos a c
; msp_closed: forall e a b,(forall d, B(e+d)%Qpos a b)->B e a b
; msp_eq: forall a b, (forall e, B e a b) -> st_eq a b
}.

Record MetricSpace : Type :=
{ msp_is_setoid :> Setoid
; ball : Qpos -> msp_is_setoid -> msp_is_setoid -> Prop
; ball_wd : forall (e1 e2:Qpos), (QposEq e1 e2) ->
            forall x1 x2, (st_eq x1 x2) ->
            forall y1 y2, (st_eq y1 y2) ->
            (ball e1 x1 y1 <-> ball e2 x2 y2)
; msp : is_MetricSpace msp_is_setoid ball
}.
```

Figure 3: The formal definition of a metric space as a dependent record.

```
Lemma Integrate01_correct : forall F (H01:Zero[<=](One:IR))
 (HF:Continuous_I H01 F) (f:Q_as_MetricSpace --> CR),
 (forall (o:Q) H, (0 <= o <= 1)->
 (f o == IRasCR (F (inj_Q IR o) H)))%CR ->
 (IRasCR (integral Zero One H01 F HF)==Integrate01 f)%CR.
```

Figure 4: The theorem stating that our definition of integral is correct.

Eventually we defined the intended equivalence relation on step functions (see Section 3.4) as a binary predicate, but first we define the split (Section 4.2) and basic applicative functor functions. For example, `Ap` is defined as:

```
Fixpoint Ap (X Y:Type)(f:StepF (X->Y))(a:StepF X):StepF Y :=
match f with
|constStepF f0 => Map f0 a
|glue o f0 f1=>let (l,r):=Split a o in (glue o(Ap f0 l)(Ap f1 r))
end.
```

We created proofs of the various laws and relationships between our definitions. This cumulates with an ultimate proof that our definition of integration coincides with a previous reference implementation from the CoRN library [CF03]:

Loosely speaking this says "for any function `F` over CoRN's real number which is continuous on $[0, 1]$ and for any function `f` from the rationals to our real numbers that agrees with `F` for rational inputs between 0 and 1, then

CoRN's integral of `F` over $[0, 1]$ is equivalent to our integral of `f`. The proof of this lemma is 300 lines long and mostly consists of translating facts about the fast implementation of the reals to the C-CoRN library and vice versa. The actual proof is quite general because it only uses certain general properties of the integral, such as linearity and monotonicity.

As a by-product of our development, we can also compute the supremum of any uniformly continuous function on $[0, 1]$.

This has been a small glimpse into our Coq development. For full details there is no better source than the source; see ⟨`http://c-corn.cs.ru.nl`⟩.

4.2. Glue and split

As discussed in Section 4.1, step functions are an inductive structure defined by two constructors. One constructor creates constant step functions, and the other constructor, `glue`, squeezes two step functions together, joining them together at a given point $o : (0, 1)_{\mathbb{Q}}$. One of the first operations we defined on step functions (after defining `fold`) was `Split`, which is like the opposite of `glue`. Recall from Section 3.1 that, given a step function f and a point $a : (0, 1)_{\mathbb{Q}}$, `Split` splits f into two pieces at a. The functions `SplitL` and `SplitR` return the left step function and the right step function respectively. Table 2 lists the association between our mathematical notation and the concrete syntax used in Coq.

Mathematical Notation	**Coq Syntax**
$\widehat{x}$	`constStepF x`
$f \rhd o \lhd g$	`glue o f g`
$f \blacktriangleright a$	`SplitL f a`
$a \blacktriangleleft f$	`SplitR f a`
$(f \blacktriangleright a, a \blacktriangleleft f)$	`Split f a`

Table 2: The concrete syntax used in Coq for our step function notation.

The key to reasoning about `Split` was to prove the `Split-Split` lemmas:

$$\begin{aligned} ab = c \quad &\Rightarrow \quad f \blacktriangleright a \blacktriangleright b \asymp f \blacktriangleright c \\ a + b - ab = c \quad &\Rightarrow \quad b \blacktriangleleft a \blacktriangleleft f \asymp c \blacktriangleleft f \\ a + b - ab = c \Rightarrow dc = a \quad &\Rightarrow \quad (a \blacktriangleleft f) \blacktriangleright b \asymp d \blacktriangleleft (f \blacktriangleright c) \end{aligned}$$

This collection of lemmas shows how the splits combine and distribute over each other. With sufficient case analysis, one can prove the above lemmas. These lemmas, combined with a few other useful lemmas (such as `Split-Map` lemmas) provided enough support to prove the laws for applicative functors without difficulty.

4.3. Equivalence of step functions

The work in the previous section defined an applicative functor of step functions over any type X. From this point on, we will require that X be a setoid (see Section 3.4). In order to help facilitate this, in our development we define new functions, `constStepF`, `glue`, `Split`, etc., that operate on step functions of setoids rather than step functions of types. These functions are definitionally equal to the previous functions, but their types now carry the setoid relation from their argument types to their result types. These new function names shadow the old function names, and the lemmas about them need to be repeated; however, their proofs are trivial by using previous proofs.

Perhaps the biggest challenge we encountered in our formalization was to prove that lifting setoid equivalence to step functions (Section 3.3) is indeed an equivalence relation—in particular showing that it is transitive. We eventually succeeded after creating some lemmas about the interaction between the equivalence relation and `Split`, etc.

4.4. Common partitions

When reasoning about two (or more) step functions, it is common to split up one of the step functions so that it shares the same partition structure as the other step function. This allows one to do induction over two step functions and have both step functions decompose the same way. Eventually, we abstracted this pattern of reasoning into an induction-like principle.

`Lemma StepF_ind2`:

$$\begin{array}{lllll}
\forall XY.\forall\Psi : X \Rightarrow Y \Rightarrow \star. & & & & \\
(\forall s_0 s_1 t_0 t_1 : \mathfrak{S}X. s_0 \asymp s_1 \Rightarrow t_0 \asymp t_1 \Rightarrow \Psi s_0 t_0 \Rightarrow \Psi s_1 t_1 & & & &) \Rightarrow \\
(\forall x : X.\forall y : Y. & \Psi & \widehat{x} & \widehat{y} &) \Rightarrow \\
(\forall o : (0,1)_{\mathbb{Q}}.\forall s_l s_r : \mathfrak{S}X.\forall t_l t_r : \mathfrak{S}Y.\Psi s_l t_l \Rightarrow \Psi s_r t_r \Rightarrow & \Psi & s_l \rhd o \lhd s_r & t_l \rhd o \lhd t_r &) \Rightarrow \\
\forall s : \mathfrak{S}X.\forall t : \mathfrak{S}Y. & \Psi & s & t &
\end{array}$$

This lemma may look complex, but it is as easy to use in Coq as an induction principle for an inductive family. Normally one would reason about two step functions by assuming, without loss of generality, that they have a common partition, then doing induction over that partition. Our lemma above combines these two steps into one. In one step, one does induction as if the two functions have a common partition. This lemma was inspired by McBride and McKinna's work on views in dependent type theory [MM04]. It allows one to "view" two step functions as having a common partition.

The lemma is used by applying it to a goal of the form `forall (s t : StepF X), <expr>`, which can be created by generalizing two step functions. There are only two cases to consider. One case is when `s` and `t` are both constant step functions. The other case is when `s` and `t` are each glued together from two step functions *at the same point*. There is, however, a side condition to be proved. One has to show that `<expr>` respects the equivalence relation on step functions for `s` and `t`. Fortunately, `<expr>` is typically constructed from respectful functions, and proving this side condition is easy.

For example, we used this lemma in the proof that $\mathsf{fold}_{\text{affine}}$ is additive.

Theorem 1. *For all step functions $f, g : \mathfrak{S}\mathbb{Q}$,*

$$\mathsf{fold}_{\text{affine}}\, f + \mathsf{fold}_{\text{affine}}\, g = \mathsf{fold}_{\text{affine}}(f \,\langle + \rangle\, g)$$

Proof The predicate $\lambda f g.\mathsf{fold}_{\text{affine}}\, f + \mathsf{fold}_{\text{affine}}\, g \asymp \mathsf{fold}_{\text{affine}}(f \,\langle + \rangle\, g)$ is a respectful predicate because $\mathsf{fold}_{\text{affine}}$ and addition are respectful functions. Therefore, we can apply `StepF_ind2`. There are only two cases to consider.

The first case is when $f = \widehat{x}$ and $g = \widehat{y}$. In this case, the problem reduces to $x + y = x + y$ after evaluating $\mathsf{fold}_{\text{affine}}$ and $\widehat{x} \,\langle + \rangle\, \widehat{y}$.

The second case is when $f = f_l \rhd o \lhd f_r$ and $g = g_l \rhd o \lhd g_r$. In this case, the problem reduces to

$$o(\mathsf{fold}_{\text{affine}}\, f_l + \mathsf{fold}_{\text{affine}}\, g_l) + (1 - o)(\mathsf{fold}_{\text{affine}}\, f_r + \mathsf{fold}_{\text{affine}}\, g_r)$$
$$=$$
$$o(\mathsf{fold}_{\text{affine}}(f_l \,\langle + \rangle\, g_l) + (1 - o)(\mathsf{fold}_{\text{affine}}(f_r \,\langle + \rangle\, g_r))$$

after evaluation. This then follows from the inductive hypothesis. □

This induction lemma was also very useful for proving the combinator equations in Section 4.5.

The proof of `StepF_ind2` is not very difficult.

Proof Suppose Ψ is a respectful binary predicate on step functions. Suppose it also satisfies the two other hypothesis of the lemma. We need to show $\forall st, \Psi st$. We proceed first by induction on s.

Consider the case when $s = \widehat{x}$. Now we do induction on t. Consider the case when $t = \widehat{y}$. This is exactly the situation of our first hypothesis, so we are done. Consider the case when $t = t_l \rhd o \lhd t_r$. We need to prove $\Psi\widehat{x}(t_l \rhd o \lhd t_r)$ assuming that $\Psi\widehat{x}t_l$ and $\Psi\widehat{x}t_r$ both hold. We know that $(\widehat{x} \blacktriangleright o) \rhd o \lhd (o \blacktriangleleft \widehat{x}) \asymp \widehat{x}$ holds, and because Ψ is respectful we can replace $\widehat{x}$ using this equivalence. Also $\widehat{x} \blacktriangleright o$ and $o \blacktriangleleft \widehat{x}$ both reduce to $\widehat{x}$ by evaluation. This leaves us with needing to show $\Psi(\widehat{x} \rhd o \lhd \widehat{x})(t_l \rhd o \lhd t_r)$. This follows from our second hypothesis and our two inductive hypotheses.

Now consider the case when $s = s_l \rhd o \lhd s_r$. We need to prove $\forall t.\Psi(s_l s_l \rhd o \lhd s_r)t$ assuming that $\forall t.\Psi s_l t$ and $\forall t.\Psi s_r t$. Again, we know that $(t \blacktriangleright o) \rhd o \lhd (o \blacktriangleleft t) \asymp t$ holds, and because Ψ is respectful we can replace t using this equivalence. The proof proceeds similar to before. □

4.5. Combinators

The combinators **B** and **I** are preserved by every applicative functor (see Section 3.3). For the applicative functor $\mathfrak{S}$, all lambda expressions are preserved. To show this, it is sufficient to show that each of the **BCKW** combinators are preserved. These are the combinators defined by:

- $\mathbf{B}fgx := f(gx)$ (compose)
- $\mathbf{C}fxy := fyx$ (interchange)
- $\mathbf{I}x := x$ (identity)
- $\mathbf{K}xy := x$ (discard)

- $\mathbf{W}fx := fxx$ (duplicate)

The identity combinator is redundant because $\mathbf{I} \asymp \mathbf{WK}$, but it is still useful.

All lambda expressions can be rewritten in a "point free" form using these combinators. Using combinators allows us to reason about the lambda calculus without worrying about binders, which are notoriously difficult to do by hand. In fact, it is one of the main issues in the POPLmark challenge [ABF+05].

Theorem 2. *The combinators,* **CKW**, *are preserved by the* $\mathfrak{S}$ *monad.*

$$\begin{aligned} \mathbf{C}\male f@x@y &\asymp_{\mathfrak{S}X} f@y@x \\ \mathbf{K}\male x@y &\asymp_{\mathfrak{S}X} x \\ \mathbf{W}\male f@x &\asymp_{\mathfrak{S}X} f@x@x \end{aligned}$$

This means that we can lift any function definable with the λ-calculus to step functions.

4.6. Lifting theorems

During our development, we often needed to prove statements like the transitivity of the order relation on the step functions:

$$\forall fgh : \mathfrak{S}\mathbb{Q}. f \{\leq_{\mathbb{Q}}\} g \Rightarrow g \{\leq_{\mathbb{Q}}\} h \Rightarrow f \{\leq_{\mathbb{Q}}\} h$$

We would like to deduce this statement from the transitivity of the corresponding pointwise relation:

$$\forall xyz : \mathbb{Q}. x \leq_{\mathbb{Q}} y \Rightarrow y \leq_{\mathbb{Q}} z \Rightarrow x \leq_{\mathbb{Q}} z$$

First, we use a lemma that lifts universal statements about an arbitrary predicate $R : X \Rightarrow Y \Rightarrow Z \Rightarrow \star$ to a universal statement about step functions:

$$(\forall x : X. \forall y : Y. \forall z : Z. Rxyz) \Rightarrow \forall f : \mathfrak{S}X. \forall g : \mathfrak{S}Y. \forall h : \mathfrak{S}Z. \mathsf{fold}_\star(R\male f@g@h)$$

This yields

$$\forall fgh : \mathfrak{S}\mathbb{Q}. \mathsf{fold}_\star((\lambda xyz. x \leq_{\mathbb{Q}} y \Rightarrow y \leq_{\mathbb{Q}} z \Rightarrow x \leq_{\mathbb{Q}} z)\male f@g@h).$$

Next, we would like to "evaluate" the lambda expression as "applied" to the step functions f, g, and h. Because f, g, and h are variables, we need to symbolically evaluate the expression. We avoid dealing with binders by converting the lambda expression into the combinator expression

$$\mathbf{S}(\mathbf{BS}(\mathbf{B}(\mathbf{B}(\mathbf{BB}(\Rightarrow)))(\leq_{\mathbb{Q}})))(\mathbf{B}(\mathbf{C}(\mathbf{BS}(\mathbf{B}(\mathbf{B}(\Rightarrow))(\leq_{\mathbb{Q}}))))(\leq_{\mathbb{Q}}))\male f@g@h,$$

where $\mathbf{S} := \mathbf{B}(\mathbf{B}(\mathbf{BW})\mathbf{C})(\mathbf{BB})$ and $(\Rightarrow)$ and $(\leq_{\mathbb{Q}})$ are prefix versions of these infix functions. This substitution is sound because the combinator term and lambda expression can easily be shown to be extensionally equivalent (by normalization), and map and ap are well-defined with respect to extensional equality.

We found the required combinator form by using lambdabot [BJ06], a standard tool for Haskell programmers. It would have been interesting to implement the algorithm for finding the combinator form of a lambda term in Coq; however, this was not the aim of our current research.

Now that the lambda term is expressed in combinator form, we can repeatedly apply the combinator equations from Section 3.3 and Section 4.5. These equations are exactly the rules of "evaluation" of this expression "applied" to step functions. We put these equations into a database of rewrite rules and used Coq's `autorewrite` system as part of a small custom tactic to automatically reduce this entire expression in one command, yielding

$$\forall fgh : \mathfrak{S}\mathbb{Q}.\, \mathsf{fold}_\star(f \,\langle\leq_\mathbb{Q}\rangle\, g \,\langle\Rightarrow\rangle\, g \,\langle\leq_\mathbb{Q}\rangle\, h \,\langle\Rightarrow\rangle\, f \,\langle\leq_\mathbb{Q}\rangle\, h).$$

Finally, we push the $\mathsf{fold}_\star$ inside. To do so, we have proved a lemma which allows us to distribute implication over $\mathsf{fold}_\star$:

$$\forall PQ : \mathfrak{S}(\star).(\mathsf{fold}_\star(P \,\langle\Rightarrow\rangle\, Q)) \Rightarrow \mathsf{fold}_\star\, P \Rightarrow \mathsf{fold}_\star\, Q$$

Repeated application of this lemma yields

$$\forall fgh : \mathfrak{S}\mathbb{Q}. f \,\{\leq_\mathbb{Q}\}\, g \Rightarrow g \,\{\leq_\mathbb{Q}\}\, h \Rightarrow f \,\{\leq_Q\}\, h$$

as required.

4.7. The Identity Bounded Function

In order to integrate uniformly continuous functions, we compose them with the identity bounded function to create a bounded function that can be integrated (see Section 3.7). This requires defining the identity bounded function on $[0, 1]$.

The bounded functions are the completion of step functions under the L^∞ metric. To create a bounded function, we need to generate a step function within ε of the identity function for every $\varepsilon : \mathbb{Q}^+$. The number of steps used in the approximation will determine the number of samples of the continuous function f that will be used. For efficiency, we want the approximation to have the fewest number of steps possible. Therefore, we defined a function `stepSample` : `positive` $\Rightarrow \mathfrak{S}\mathbb{Q}$, where `positive` is the binary positive natural numbers, such that `stepSample`n produces the best approximation of the identity function with n steps.

It is unfortunate that the width of each step is computed during integration, because we know that the result will always be equivalent to $\frac{1}{n}$ for these particular step functions. Perhaps some other data structure for step functions could be used that explicitly stores the length of each step. However, the time spent computing the length of the interval is usually much smaller that the time it takes to sample the continuous function f.

Function	**Time**
`(answer 3 (Integrate01 Cunit))`	0.18s
`(answer 2 (Integrate01 cos_uc))`	0.52s
`(answer 3 (Integrate01 cos_uc))`	8.55s
`(answer 3 (Integrate01 sin_uc))`	7.48s

Table 3: `Time Eval vm_compute in ...` carries out the reduction using Coq's virtual machine. The expression `answer` n asks for an answer to within 10^{-n}. All computations where carried out on an IBM Thinkpad X41.

4.8. Timings

The version of Riemann integration that we implemented applies to *general* continuous functions and hence has bad complexity behavior. If we knew more about the function, for instance if it is differentiable, faster algorithms could be used [Eda99].

When extracted to OCaml, the functions run approximately five times faster when compiled and optimized.

5. Future and related work

Many optimizations are possible. Most time is spend on evaluating the function at many points, as can be seen by comparing the timings for the sin function and the identity function (`CUnit`) which have the same modulus of continuity and hence the same partition.

Some ways of speeding up the computation of these functions are discussed in [O'C08a]. Most notable are:

- the use of dyadic rationals;
- the use of machine integers, (which will enter Coq in the near future);
- the use of forward propagation of errors instead of our a priori estimates of convergence [BK09];
- the use of parallelism. Our use of maps and folds makes it easy to run the algorithm in parallel. In fact, adding parallelism to the extacted O'Caml code by hand speeds up the evalutation by a factor three on a four processor machine. This only required making a single function, `DistrComplete` (a fold), be evaluated in parallel.

  We hope that the technology of parallel functional programming will included in Coq in the future.

Because of the way that we have defined uniform continuity, one modulus of continuity applies to an entire function. Even for those parts of the domain where the function changes slowly, we still must approximate the input to the same precision that is needed for those parts where the function changes quickly.

This reduces performance somewhat for evaluation of these functions (at the segments where the function changes slowly), but this causes particularly bad performance for integration.

Because we only have a global modulus of continuity, we must use uniform partitions when creating an integrable function from a uniformly continuous function. This means that the function is sampled just as often where the function changes slowly as where the function changes quickly. This uniform sampling can be quite expensive for integration.

There is some potential to increase efficiency by using a "non-uniform" definition of uniform continuity. That is to say, using a definition of uniform continuity that allows different segments of the domain to have local moduli associated with them. Ulrich Berger uses such a definition of uniform continuity to define integration [Ber09]. Simpson also defines an integration algorithm that uses a local modulus for a function that is computed directly from the definition of the function [Sim98]. However, implementing his algorithm directly in Coq is not possible because it relies on bar induction, which is not available in Coq unless one adds an axiom such as bar induction to it or one treats the real numbers as a formal space [Sam87][Bau08].

The constructive real numbers have already been used to provide a semi-decision procedure for inequalities of real numbers. Not only for the constructive real numbers, but also for the non-computational real numbers in the Coq standard library [KO08]. The same technique can be applied here.

Previously, the CoRN project [CFGW04] showed that the formalization of constructive analysis in a type theory is feasible. However, the extraction of programs from such developments is difficult [CFS03]. On the contrary, in the present article we have shown that if one takes an algorithmic attitude from the start it *is* possible to obtain feasible programs.

6. Conclusions

We have implemented Riemann integration in constructive mathematics based on type theory. Type checking guarantees that the implementation meets its formal specification. The use of the completion and the step function monads helped to structure the program/proof, as did the use of applicative functors.

Building on the previous implementation of the completion of a metric space [O'C08a] and the library [CF04], the current implementation was completed in four man-months. The program/proof consists of 1155 lines of specifications, 3380 lines of proof, and 170,137 total characters. The size of the gzipped tarball (`gzip -9`) of all the source files is 37,039 bytes, which is an estimate of the information content.

Together with the work in [O'C07, O'C08a, O'C08b], the current project may be seen as the beginning of the realization of Bishop's program to use constructive mathematics, based on type theory, as a programming language for exact analysis.

7. Acknowledgements

We thank Cezary Kaliszyk for helping us to implement parallelism in OCaml.

[ABF$^+$05] B. Aydemir, A. Bohannon, M. Fairbairn, J. Foster, B. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *Proceedings of the Eighteenth International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, 2005.

[Bau08] Andrej Bauer. Efficient computation with dedekind reals. Extended abstract for CCA2008, 2008.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.

[BCP03] Gilles Barthe, Venanzio Capretta, and Olivier Pons. Setoids in type theory. *J. Funct. Programming*, 13(2):261–293, 2003. Special issue on "Logical frameworks and metalanguages".

[Bec69] J. Beck. Distributive laws. In B. Eckman, editor, *Seminar on Triples and Categorical Homology Theory*, number 80 in Lecture Notes in Mathematics, pages 119–140. Springer, Berlin, 1969.

[Ber09] Ulrich Berger. From coinductive proofs to exact real arithmetic. In *Computer Science Logic*, pages 132–146. Springer, 2009.

[Bis67] Errett A. Bishop. *Foundations of constructive analysis.* McGraw-Hill Publishing Company, Ltd., 1967.

[Bis70] Errett Bishop. Mathematics as a numerical language. In *Intuitionism and Proof Theory (Proceedings of the summer Conference at Buffalo, N.Y., 1968)*, pages 53–71. North-Holland, Amsterdam, 1970.

[BJ06] Andrew J. Bromage and Thomas Jäger. Lambdabot. `http://www.cse.unsw.edu.au/~dons/lambdabot.html`, 2006.

[BK09] Andrej Bauer and Iztok Kavkler. A constructive theory of continuous domains suitable for implementation. *Ann. Pure Appl. Logic*, 159(3):251–267, 2009.

[BW05] Michael Barr and Charles Wells. Toposes, triples and theories. *Repr. Theory Appl. Categ.*, (12):x+288 pp., 2005. Corrected reprint of the 1985 original [MR0771116].

[CF03] L. Cruz-Filipe. A constructive formalization of the fundamental theorem of calculus. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs*, volume 2646 of *LNCS*, pages 108–126. Springer–Verlag, 2003.

[CF04] L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications.* PhD thesis, University of Nijmegen, April 2004.

[CFGW04] L. Cruz-Filipe, H. Geuvers, and F. Wiedijk. C-corn: the constructive coq repository at nijmegen. In A. Asperti, G. Bancerek, and A. Trybulec, editors, *Mathematical Knowledge Management, Third International Conference, MKM 2004*, volume 3119 of *LNCS*, pages 88–103. Springer–Verlag, 2004.

[CFS03] L. Cruz-Filipe and B. Spitters. Program extraction from large proof developments. In D. Basin and B. Wolff, editors, *Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2003*, volume 2758 of *LNCS*, pages 205–220. Springer–Verlag, 2003.

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Inform. and Comput.*, 76(2-3):95–120, 1988.

[Coe04] Claudio Sacerdoti Coen. A semi-reflexive tactic for (sub-)equational reasoning. In Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors, *TYPES*, volume 3839 of *Lecture Notes in Computer Science*, pages 98–114. Springer, 2004.

[CP90] Thierry Coquand and Christine Paulin. Inductively defined types. In *COLOG-88 (Tallinn, 1988)*, volume 417 of *Lecture Notes in Comput. Sci.*, pages 50–66. Springer, Berlin, 1990.

[Eda99] Abbas Edalat. Numerical integration with exact real arithmetic. In *Automata, Languages and Programming, 26th International Colloquium, ICALP99, Prague, Czech 227 Republic, July 11-15, 1999, Proceedings, volume 1644 of Lecture Notes in Computer Science*, pages 90–104. Springer, 1999.

[GL02] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *ICFP*, pages 235–246, 2002.

[GNSW07] Herman Geuvers, Milad Niqui, Bas Spitters, and Freek Wiedijk. Constructive analysis, types and exact real numbers (overview article). *Mathematical Structures in Computer Science*, 17(1):3–36, 2007.

[Hof97] Martin Hofmann. *Extensional constructs in intensional type theory.* CPHC/BCS Distinguished Dissertations. Springer-Verlag London Ltd., London, 1997.

[JD93] Mark P. Jones and Luke Duponcheel. Composing monads. Technical Report YALEU/DCS/RR-1004, Yale University, 1993.

[KO08] Cezary Kaliszyk and Russell O'Connor. Computing with classical real numbers. Submitted for publication to the Journal of Automated Reasoning, 2008.

[Lan93] Serge Lang. *Real and Functional Analysis*. Springer, 1993.

[ML82] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, methodology and philosophy of science, VI (Hannover, 1979)*, volume 104 of *Stud. Logic Found. Math.*, pages 153–175. North-Holland, Amsterdam, 1982.

[ML98] Per Martin-Löf. An intuitionistic theory of types. In *Twenty-five years of constructive type theory (Venice, 1995)*, volume 36 of *Oxford Logic Guides*, pages 127–172. Oxford Univ. Press, 1998.

[MM04] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[Mog89] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 14–23, Piscataway, NJ, USA, 1989. IEEE Press.

[MP08] Conor McBride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, 2008.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's type theory*, volume 7 of *International Series of Monographs on Computer Science*. The Clarendon Press Oxford University Press, New York, 1990. An introduction.

[O'C07] Russell O'Connor. A monadic, functional implementation of real numbers. *Mathematical Structures in Computer Science*, 17(1):129–159, 2007.

[O'C08a] Russell O'Connor. Certified exact transcendental real number computation in Coq. In Otmane Ait-Mohamed, editor, *TPHOLs*, volume 5170 of *Lecture Notes in Computer Science*, pages 246–261. Springer, 2008.

[O'C08b] Russell O'Connor. A computer verified theory of compact sets. In Bruno Buchberger, Tetsuo Ida, and Temur Kutsia, editors, *SCSS 2008*, number 08-08 in RISC-Linz Report Series, pages 148–162, Castle of Hagenberg, Austria, July 2008. RISC.

[Ric08] Fred Richman. Real numbers and other completions. *Math. Log. Q.*, 54(1):98–108, 2008.

[Sam87] Giovanni Sambin. Intuitionistic formal spaces - a first communication. In D. Skordev, editor, *Mathematical logic and its Applications*, pages 187–204. Plenum, 1987.

[Sch05] L. Schröder. Expressivity of coalgebraic modal logic: The limits and beyond. In V. Sassone, editor, *Foundations of Software Science and Computational Structures*, number 3441 in Lecture Notes in Mathematics, pages 440–454. Springer, Berlin, 2005.

[Sch08] Helmut Schwichtenberg. Realizability interpretation of proofs in constructive analysis. *Theor. Comp. Sys.*, 43(3):583–602, 2008.

[Sim98] Alex K. Simpson. Lazy functional algorithms for exact real functionals. *Lecture Notes in Computer Science*, 1450:456–464, 1998.

[SU98] M. Sörensen and P. Urzyczyn. *Lectures on the Curry-Howard isomorphism.* Elsevier, 1998.

[Tea08] The Coq Development Team. *The Coq Proof Assistant Reference Manual.* INRIA-Rocquencourt, 2008.

[Tho91] S. Thompson. *Type Theory and Functional Programming.* Addison Wesley, 1991.

[Wad92a] P. Wadler. Monads for functional programming. In *Proceedings of the Marktoberdorf Summer School on Program Design Calculi*, August 1992.

[Wad92b] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, 1992.