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

Russell O'Connor
Radboud University Nijmegen

Bas Spitters
Eindhoven University of Technology

22 August 2008

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]

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. 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]. 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 implementation of type theory [Tea08]. 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 2.6). 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).

1. Bas Spitters is supported by NWO.

- Using combinators we also lift theorems to reason about these functions and relations on step functions (Section 4.5).
- 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 $X \Rightarrow (Y \Rightarrow Z)$).

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 $\exists n\colon \mathbb{N}.\, Pn$ gives an explicit value for n that makes Pn hold. Most importantly, we have a functional interpretation of $\Rightarrow$ and $\forall$. A proof of $\forall n\colon \mathbb{N}.\exists m\colon \mathbb{N}.\, Rnm$ 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 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)[Tho91]. 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 [Tho91]. This isomorphism associates formulas with dependent types, and proofs of formulas with functional programs of the associated dependent types. For example, the identity function $\lambda x\colon A.\, x$ 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, $\Sigma x\colon A.\, Px$, and dependent function types, $\Pi x\colon A.\, Px$. 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:

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

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, $\mathrm{fix} : (\varphi \Rightarrow \varphi) \Rightarrow \varphi$, 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, the language can still express fast growing functions like the Ackermann function without difficulty [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

$$\exists f\colon \mathbb{Q}^+ \Rightarrow \mathbb{Q}. \forall \varepsilon_1\, \varepsilon_2. |f\varepsilon_1 - f\varepsilon_2| \leq \varepsilon_1 + \varepsilon_2. \tag{1}$$

Values of this type are pairs of a function $f\colon \mathbb{Q}^+ \Rightarrow \mathbb{Q}$ and a proof of $\forall \varepsilon_1\, \varepsilon_2. |f\varepsilon_1 - f\varepsilon_2| \leq \varepsilon_1 + \varepsilon_2$. The idea is that a real number is represented by a function f that maps any requested precision $\varepsilon\colon \mathbb{Q}^+$ 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 datatypes.

2.3. Metric Spaces. Traditionally, a metric space is defined as a set X with a metric function $d\colon X \times X \Rightarrow \mathbb{R}^{0+}$ 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. Instead, the metric is represented via a (respectful) ball relation $B\colon \mathbb{Q}^+ \Rightarrow X \Rightarrow X \Rightarrow \star$, where $\star$ is the type of propositions, satisfying five axioms:

1. $\forall x\, \varepsilon. B_\varepsilon\, x\, x$
2. $\forall x\, y\, \varepsilon. B_\varepsilon\, x\, y \Rightarrow B_\varepsilon\, y\, x$
3. $\forall x\, y\, z\, \varepsilon_1\, \varepsilon_2. B_{\varepsilon_1} x\, y \Rightarrow B_{\varepsilon_2} y\, z \Rightarrow B_{\varepsilon_1+\varepsilon_2} x\, z$
4. $\forall x\, y\, \varepsilon. (\forall \delta. \varepsilon < \delta \Rightarrow B_\delta\, x\, y) \Rightarrow B_\varepsilon x\, y$
5. $\forall x\, y. (\forall \varepsilon. B_\varepsilon x\, y) \Rightarrow x \asymp y$

The ball relation $B_\varepsilon\ x\, y$ expresses that the points x and y are within ε of each other. We call this a ball relationship because the partially applied relation $B_\varepsilon\, x\colon X \Rightarrow \star$ 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

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

This definition satisfies all the required axioms.

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

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

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 $f\colon X \to Y$ and $x\colon X$.

2.5. Monads. Moggi [Mog89] and Wadler [Wad92] recognized that many non-standard forms of computation may be modeled by monads[2]. 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 $MY := (Y \times S)^S$ 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, $MY := Y^E$, for passing an environment implicitly will play an important role in this paper.

The formal definition of a (strong) monad is a triple (M, return, bind) consisting of a type constructor M and two functions:

$$\begin{aligned} \text{return} &: X \Rightarrow MX \\ \text{bind} &: (X \Rightarrow MY) \Rightarrow MX \Rightarrow MY \end{aligned}$$

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

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

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

$$\begin{aligned} \text{return} &: X \Rightarrow MX \\ \text{map} &: (X \Rightarrow Y) \Rightarrow (MX \Rightarrow MY) \\ \text{join} &: M(MX) \Rightarrow MX \end{aligned}$$

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

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

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

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

2.6. 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

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

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

The function return: $X \to \mathfrak{C}X$ is the embedding of a metric space in its completion. The function 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 $f\colon X \to Y$ can be lifted to operate on complete metric spaces, map $f\colon \mathfrak{C}X \to \mathfrak{C}Y$. 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 $f\colon \mathbb{Q} \to \mathbb{R}$, 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].

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

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 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 have defined 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 const x for some $x\colon X$, or two step functions, $f\colon \mathfrak{S}X$ and $g\colon \mathfrak{S}X$ glued at a point in o, glue $o\,f\,g$, where o must be a rational number strictly between 0 and 1. We will write const x as $\hat{x}$, and glue $o\,f\,g$ as $f \triangleright o \triangleleft g$.

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

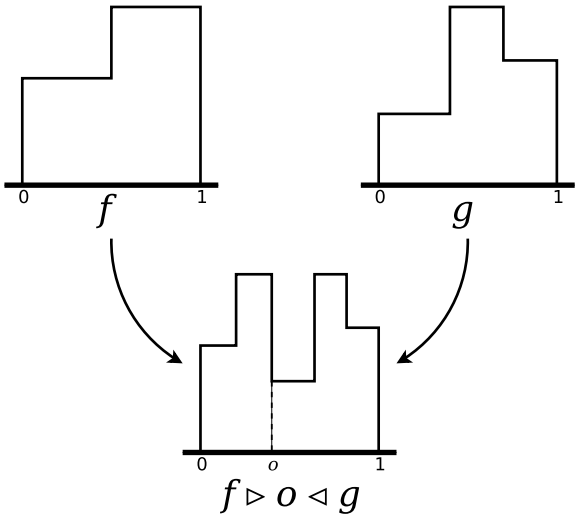

Figure 1. Given two step functions f and g, the step function $f \triangleright o \triangleleft g$ is f squeezed into $[0, o]$ and g squeezed into $[o, 1]$.

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 take the place of step functions in classical mathematics. It does not matter that our informal interpretation of $f \triangleright o \triangleleft g$ 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 with leaves of 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; more formally, the equivalence relation is defined in Section 3.4.

The inductive type for step functions has an associated catamorphism[3] which we call fold:

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

This fold operation is used in many places. For instance to define two metrics on step functions (3.5) or to check whether a property holds globally on [0,1]. Not every fold respects the equivalence relation on step functions, so one needs to prove that the generated functions respect this equivalence relation.

3.2. Step functions are a monad. Step functions form a monad, $\mathfrak{S}$, similar to the familiar reader monad $\lambda X.X^{[0,1]}$. The unit of $\mathfrak{S}$ is the constant function, map is defined in the obvious way, 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, 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.

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

3.3. Applicative functors. Let T be a (strong) monad. To lift a function $f\colon X \Rightarrow Y$ to a function $TX \Rightarrow TY$, we use map: $(X \Rightarrow Y) \Rightarrow TX \Rightarrow TY$. Lifting a function with two curried arguments is possible using a similar function map2. However, to avoid having to write a function map$n$ for each natural number n, one uses the theory of applicative functors. An *applicative functor* contains two functions:

$$\begin{aligned} \text{pure} &: X \Rightarrow TX \\ \text{ap} &: T(X \Rightarrow Y) \Rightarrow TX \Rightarrow TY \end{aligned}$$

The function pure lifts any value inside the functor. The ap function applies a function inside the functor to a values inside the functor to produce a value inside the functor. We denote pure x by $\hat{x}$, as was done for monads, and we denote ap fx by $f \mathbin{@} x$. An applicative functor must satisfy the following laws:

$\hat{\mathbf{I}} \mathbin{@} v \asymp v$	Identity
$\hat{\mathbf{B}} \mathbin{@} u \mathbin{@} v \mathbin{@} w \asymp u \mathbin{@} (v \mathbin{@} w)$	Composition
$\hat{f} \mathbin{@} \hat{x} \asymp \widehat{fx}$	Homomorphism
$u \mathbin{@} \hat{y} \asymp \widehat{\text{ev}_y} \mathbin{@} u$	Interchange

Where $\mathbf{B}$ and $\mathbf{I}$ are the composition and identity combinators respectively (see Section 4.4) and $\text{ev}_y := \lambda f.\, fy$ is the function which evaluates at y.

Every monad can be seen as an applicative functor [MP08]. Every monad induces the canonical applicative functor

$$\begin{aligned} \text{pure} &:= \text{return} \\ f \mathbin{@} x &:= \text{bind}\,(\lambda g.\,\text{map}\,g\,x)\,f. \end{aligned}$$

As the name suggests, every applicative functor can be seen as a functor. Given an applicative functor we define $map : (X \Rightarrow Y) \Rightarrow TX \Rightarrow TY$ as

$$\text{map}\,fx := \hat{f} \mathbin{@} x.$$

When T is generated from a monad, this definition of map is equivalent to the definition of map associated with the monad.

3.4. Step functions as an applicative functor.

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

The binary version of map is defined as

$$\text{map2}\,fab := f \male a \mathbin{@} b.$$

3. In functional programming, a catamorphism is a generalization of the fold on lists to arbitrary abstract data types which can be described as initial algebras.

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. If $\circledast$ is some infix operator such that $\lambda x\, y.\, x \circledast y \colon X \Rightarrow Y \Rightarrow Z$, then we define

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

where $f \colon \mathfrak{S} X$, $g \colon \mathfrak{S} Y$, and $f \langle \circledast \rangle g \colon \mathfrak{S} Z$. For example, if $f, g \colon \mathfrak{S}\mathbb{Q}$ are rational step function, 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 $\lambda x\, y.\, x \propto y \colon X \Rightarrow Y \Rightarrow \star$. If we use map2, we end up with an function $\lambda f g.\, f \langle \propto \rangle g \colon \mathfrak{S} X \Rightarrow \mathfrak{S} Y \Rightarrow \mathfrak{S} \star$. The result is not a proposition, rather it is 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 $[0, 1]$.

We can turn a characteristic function into a proposition by asking it to hold everywhere. The function $\mathrm{fold}_\star \colon \mathfrak{S} \star \Rightarrow \star$ does this by folding the conjunction over a step function.

$$\mathrm{fold}_\star := \mathrm{fold}\, \mathbf{I}\, (\lambda o p q.\, p \wedge q)$$

This function can be composed with the map2, lifting a relation to a relation on step functions:

$$f \{ \propto \} g := \mathrm{fold}_\star (f \langle \propto \rangle g)$$

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

$$f \asymp_{\mathfrak{S} X} g := f \{ \asymp_X \} g$$

Two functions are equivalent if they are pointwise equivalent everywhere. Similarly, we define inequality for $f g \colon \mathfrak{S}\mathbb{Q}$ by lifting the inequality relation on $\mathbb{Q}$:

$$f \leq_{\mathfrak{S}\mathbb{Q}} g := f \{ \leq_{\mathbb{Q}} \} g$$

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

3.5. Two metric spaces of 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. Formally, we first define the two norms on the step functions:

$$\begin{aligned} \|f\|_\infty &:= \mathrm{fold}_{\sup} (\mathrm{abs} \mathbin{♂} f) \\ \|f\|_1 &:= \mathrm{fold}_{\mathrm{affine}} (\mathrm{abs} \mathbin{♂} f) \end{aligned}$$

where

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

and $\mathrm{abs} : \mathbb{Q} \Rightarrow \mathbb{Q}$ is the absolute value function on $\mathbb{Q}$. The function $\mathrm{fold}_{\sup} : \mathfrak{S}\mathbb{Q} \Rightarrow \mathbb{Q}$ returns the supremum of the step function, while the function $\mathrm{fold}_{\mathrm{affine}} : \mathfrak{S}\mathbb{Q} \Rightarrow \mathbb{Q}$ returns the integral of a step function.

Next, the metric distance between two step functions is simply defined as

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

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

$$\begin{aligned} B_\varepsilon^{\mathfrak{S}^\infty \mathbb{Q}} f g &:= d^\infty f g \leq \varepsilon \\ B_\varepsilon^{\mathfrak{S}^1 \mathbb{Q}} f g &:= d^1 f g \leq \varepsilon \end{aligned}$$

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.

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

The identity function is uniformly continuous in one direction, $\iota\colon \mathfrak{S}^\infty\mathbb{Q} \to \mathfrak{S}^1\mathbb{Q}$; 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:

$$\begin{aligned} B_\varepsilon^{\mathfrak{S}^\infty X}(f,g) &:= \mathrm{fold}_\star(B_\varepsilon^X \eth f @ g) \\ B_\varepsilon^{\mathfrak{S}^1 X}(f,g) &:= \exists h\colon \mathfrak{S}\mathbb{Q}^+.\, \mathrm{fold}_\star(B^X \eth h @ f @ g) \wedge \|h\|_1 \leqslant \varepsilon \end{aligned}$$

Then one can prove that the monad $\mathfrak{S}^\infty$ is a submonad of $\mathfrak{S}^1$. However, we have only pursued the first monad in our formalization.

3.6. Integrable Functions and Bounded Functions. The bounded functions and the integrable functions are defined to be the completion of the step functions under the L^∞ and the L^1 metrics respectively.

$$\begin{aligned} \mathfrak{B}\mathbb{Q} &:= \mathfrak{C}(\mathfrak{S}^\infty\mathbb{Q}) \\ \mathfrak{I}\mathbb{Q} &:= \mathfrak{C}(\mathfrak{S}^1\mathbb{Q}) \end{aligned}$$

In section 3.1, we interpreted elements of $\mathfrak{S}X$ as (partially defined) functions on [0,1]. Similarly, we can associate a (partially defined) function to each bounded function. Let $f\colon \mathbb{Q}^+ \Rightarrow \mathfrak{S}^\infty\mathbb{Q}$ be an element of $\mathfrak{B}\mathbb{Q}$. Define $g_n := f(\frac{1}{n})$. Then $\lim g_n(x)$ exists for all points x in [0,1] except perhaps for the (rational) splitting points of the step functions g_n. In the points where this limit is defined, it is continuous.

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

$$h_n := \lambda\varepsilon.\mathrm{map}\,(\lambda x.\max(\min x\, n)(-n))(h\,\varepsilon)$$

To each h_n we associate a partial function $\widetilde{h_n}$ by taking the pointwise limit of the step functions. These functions are bounded and continuous almost everywhere. Consequently, they are Riemann integrable. It is clear that the limit $\int \widetilde{h_n}$ converges.

The bounded functions have a supremum operation, $\sup : \mathfrak{B}\mathbb{Q} \to \mathbb{R}$ 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.

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

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

3.7. Riemann Integral. This 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 bounded functions 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 $\varepsilon\colon \mathbb{Q}^+$. 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 values at each step is taken by sampling the function at the center of each step, $f(x_i)$.

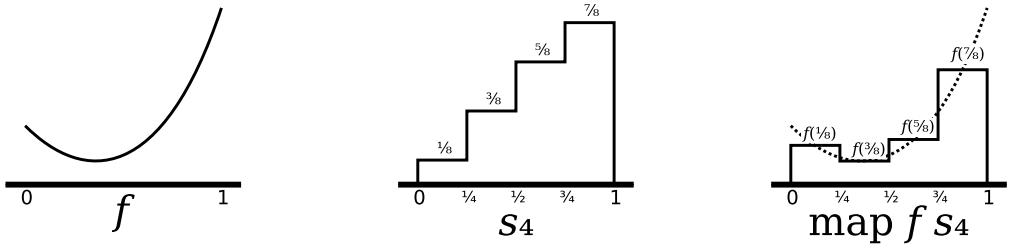

Figure 2. Given a uniformly continuous function f and a step function s_4 that approximates the identity function, the step function map $f s_4$ (or $f \mathbin{♂} s_4$) approximates f in the familiar Riemann way.

When developing the above, it became clear that one can achieve the above results by creating a step function whose values are x_i, and then mapping f over these "sampling step-functions" (see Figure 2). 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 4.6). Given any uniformly continuous function $f\colon \mathbb{Q} \to \mathbb{Q}$, we can prove that $\mathrm{map}_{\mathfrak{S}^\infty} f\colon \mathfrak{S}^\infty\mathbb{Q} \to \mathfrak{S}^\infty\mathbb{Q}$ is uniformly continuous. Then we can lift again to operate on bounded functions, $\mathrm{map}_{\mathfrak{C}}\,(\mathrm{map}_{\mathfrak{S}^\infty} f)\colon \mathfrak{B}\mathbb{Q} \to \mathfrak{B}\mathbb{Q}$. Applying this to $\mathbf{I}_{[0,1]}$ yields f restricted to $[0,1]$ as a bounded function, which then can be converted to an integrable function and integrated.

$$\int_{[0,1]} f := \int (\mathrm{map}_{\mathfrak{C}}\, \iota\, (\mathrm{map}_{\mathfrak{C}}\, (\mathrm{map}_{\mathfrak{S}^\infty} f)\, \mathbf{I}_{[0,1]}))$$

With a small modification, this process will also work for $f\colon \mathbb{Q} \to \mathbb{R}$. In this case map $f\colon \mathfrak{S}\mathbb{Q} \Rightarrow \mathfrak{S}\mathbb{R}$; however, we haven't given a metric structure for $\mathfrak{S}\mathbb{R}$. Fortunately, there is an injection $\mathrm{dist} : \mathfrak{S}\mathbb{R} \Rightarrow \mathfrak{B}\mathbb{Q}$, that interprets a step function of real values as a bounded function (see section 3.9). We can prove that the composition $\mathrm{dist} \circ (\mathrm{map}\, f)\colon \mathfrak{S}^\infty\mathbb{Q} \to \mathfrak{B}\mathbb{Q}$ 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 then can be integrated.

$$\int_{[0,1]} f := \int (\mathrm{map}_{\mathfrak{C}}\, \iota\, (\mathrm{bind}_{\mathfrak{C}}\, (\mathrm{dist} \circ (\mathrm{map}_{\mathfrak{S}}\, f))\, \mathbf{I}_{[0,1]}))$$

An arbitrary uniformly continuous function $f\colon \mathbb{R} \to \mathbb{R}$ can be integrated on [0,1] by integrating $\lambda x.\, f\, \hat{x} : \mathbb{Q} \to \mathbb{R}$ 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, $g\colon \mathfrak{B}\mathbb{Q}$? An analysis shows that the result is Stieltjes integral with respect to g^{-1}, when g is non-decreasing.

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

The more general Stieltjes integral practically falls out of our work for free. This is not quite as general as 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 (if g is not monotonic, then g^{-1} is a bizarre cumulative distribution non-function that can double back on itself). 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 only have discontinuities at rational points.

We have tried to allow g to be an arbitrary integrable function (this would lift some of the previous restrictions); however, we have been unable to constructively show that $\mathrm{dist} \circ \mathrm{map}_{\mathfrak{S}} f\colon \mathfrak{S}^1\mathbb{Q} \Rightarrow \mathfrak{I}\mathbb{Q}$ is uniformly continuous when f is. We have generated counterexamples where f is uniformly continuous with modulus μ and $\mathrm{dist} \circ \mathrm{map}_{\mathfrak{S}} f$ is *not* uniformly continuous with modulus μ; however, for our particular counterexamples, $\mathrm{dist} \circ \mathrm{map}_{\mathfrak{S}} f$ 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 $\mathrm{dist} : \mathfrak{S}\mathbb{R} \Rightarrow \mathfrak{B}\mathbb{Q}$ combines two monads on metric spaces $\mathfrak{C}$ and $\mathfrak{S}$: dist has type $\mathfrak{S}\mathfrak{C} \Rightarrow \mathfrak{C}\mathfrak{S}$. In general, the composition of two monads $M \circ N$ forms a monad when there is a distribution function $\mathrm{dist} : N(MX) \to M(NX)$ satisfying[4][JD93]

$$\begin{aligned} \mathrm{dist} \circ \mathrm{map}_N\,(\mathrm{map}_M\, f) &\asymp \mathrm{map}_M\,(\mathrm{map}_N\, f) \circ \mathrm{dist} \\ \mathrm{dist} \circ \mathrm{return}_N &\asymp \mathrm{map}_M\, \mathrm{return}_N \\ \mathrm{dist} \circ \mathrm{map}_N\, \mathrm{return}_M &\asymp \mathrm{return}_M \\ \mathrm{prod} \circ \mathrm{map}_N\, \mathrm{dorp} &\asymp \mathrm{dorp} \circ \mathrm{prod} \end{aligned}$$

where

$$\begin{aligned} \mathrm{prod} &:= \mathrm{map}_M \mathrm{join}_N \circ \mathrm{dist} \\ \mathrm{dorp} &:= \mathrm{join}_M \circ \mathrm{map}_M \mathrm{dist}. \end{aligned}$$

In our case, $\mathfrak{B}X := (\mathfrak{C} \circ \mathfrak{S}^\infty)X$ is the collection of bounded (integrable) functions. The distribution function $\mathrm{dist}\colon \mathfrak{S}^\infty(\mathfrak{C}X) \to \mathfrak{C}(\mathfrak{S}^\infty X)$ is defined as

$$\mathrm{dist}\, x := \lambda\varepsilon.\mathrm{map}_{\mathfrak{S}^\infty}\,(\lambda z.z(\varepsilon))\, x.$$

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

3.10. Correctness. We have proved our implementation correct by showing that our Riemann integral is equivalent to the definition of the integral in the C-CoRN library [CFGW04]. This library contains many machine verified facts about the Riemann integral closely following [BB85]. The correctness is one lemma with a 300-line proof mostly consisting 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 define the supremum of any uniformly continuous function on $[0, 1]$.

4. Implementation in Coq

In this section, we treat aspects related to the implementation in type theory.

4.1. Glue and Split. As discussed in section 3.1, step functions are an inductive structure defined by two constructors. One constructor `constStepF` creates constant step functions, and the other constructor, `glue`, squeezes two step functions together, joining them together at a given point $o\colon]0, 1[_{\mathbb{Q}}$. One of the first operations we defined on step functions (after defining `fold`) was `Split`, which is like the opposite of `glue`. Given a step function f and a point $a\colon]0, 1[_{\mathbb{Q}}$, `Split` splits f into two pieces at a. The functions `SplitL` and `SplitR` return the left step function and the right step function respectively. The idea is that $\texttt{SplitL}\ fa \rhd a \lhd \texttt{SplitR}\ fa$ will be equivalent to the original step function f, although we have not defined the equivalence relation at this point yet.

This split function was the key to defining the `Ap` component of the applicative functor. Recall that `Ap` takes a step function of functions and a step function of arguments and applies the functions to the arguments pointwise. `Split` is used recursively to refine one of the arguments until its structure is compatible with the the other argument.

$$\begin{aligned} \texttt{Ap}\,(\texttt{constStepF}\, f)\, x &:= \texttt{Map}\, fx \\ \texttt{Ap}\,(f_l \rhd o \lhd f_r)\, x &:= \mathrm{let}\,(x_l, x_r) := \texttt{Split}\, x\, o\, \mathrm{in}\,(\texttt{Ap}\, f_l\, x_l) \rhd o \lhd (\texttt{Ap}\, f_r\, x_r) \end{aligned}$$

4. We formally checked all of these rules apart from last one which was too tedious; however, the correctness of the integral does not depend on the proofs of these laws.

The function `Map` is defined in the obvious way:

$$\begin{aligned} \texttt{Map}\, f\, (\texttt{constStepF}\, x) &:= \texttt{constStepF}\, (fx) \\ \texttt{Map}\, f\, (x_l \rhd o \lhd x_r) &:= (\texttt{Map}\, f x_l) \rhd o \lhd (\texttt{Map}\, f x_r) \end{aligned}$$

The key to reasoning about `Split` was to prove the `Split-Split` lemmas. This collection of lemmas show how the splits combine and distribute over each other.

$$\begin{aligned} a\, b = c &\Rightarrow \texttt{SplitL}\, (\texttt{SplitL}\, f a)\, b \asymp \texttt{SplitL}\, f c \\ a + b - a\, b = c &\Rightarrow \texttt{SplitR}\, (\texttt{SplitR}\, f a)\, b \asymp \texttt{SplitR}\, f c \\ a + b - a\, b = c \to d\, c = a &\Rightarrow \texttt{SplitL}\, (\texttt{SplitR}\, f a)\, b \asymp \texttt{SplitR}\, (\texttt{SplitL}\, f c)\, d \end{aligned}$$

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 functions without difficulty.

4.2. Equivalence of step functions. The work in the previous section defined an applicative functor of step functions over any type X. In this section, we will require that X be a setoid — a type with an equivalence relation. 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 for 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 with the equivalence relation and `Split`, etc.

4.3. 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 functions. This allows one to do induction over one step function and have both step function decompose the same way. Eventually, we abstracted this pattern of reasoning into an induction-like principle.

$$\begin{array}{l} \texttt{Lemma StepF\_ind2 :} \\ \quad \forall X Y.\, \forall \Psi\colon X \Rightarrow Y \Rightarrow \star. \\ \quad (\forall s_0\, s_1\, t_0\, t_1 : \mathfrak{S}X.\, s_0 \asymp s_1 \Rightarrow t_0 \asymp t_1 \Rightarrow \Psi s_0\, t_0 \Rightarrow \Psi\, s_1\, t_1 \qquad\qquad\qquad\quad) \Rightarrow \\ \quad (\forall x\colon X.\, \forall y\colon Y. \qquad\qquad \Psi \quad \hat{x} \qquad\quad \hat{y} \quad) \Rightarrow \\ \quad (\forall o\colon]0,1[.\, \forall s_l\, s_r\colon \mathfrak{S}X.\, \forall t_l\, t_r\colon \mathfrak{S}Y.\, \Psi\, s_l\, t_l \Rightarrow \Psi\, s_r\, t_r \Rightarrow \Psi (s_l \rhd o \lhd s_r)\, (t_l \rhd o \lhd t_r)) \Rightarrow \\ \quad \forall s\colon \mathfrak{S}X.\, \forall t\colon \mathfrak{S}Y. \qquad\qquad \Psi \quad s \qquad\quad t \end{array}$$

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 and 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 function *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, typically `<expr>` is constructed from morphisms, and proving this side condition is easy.

This induction lemma was very useful for proving the combinator equations in section 4.4.

4.4. Combinators. In order to work with expressions like

$$\forall x\, y\, z.\, x \leqslant y \Rightarrow y \leqslant z \Rightarrow x \leqslant y,$$

one might first be inclined to reason with binders. However, this is notoriously difficult to do by hand [O'C05]. Although a number of solutions have been proposed [ABF+05], we decided to avoid the problem and use the **BCKW**-combinator presentation of the λ-calculus. 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. The combinators $\mathbf{B}$ and $\mathbf{I}$ are preserved by every applicative functor (see Section 3.3). For the applicative functor $\mathfrak{S}$, all of the combinators are preserved:

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

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

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

$$\forall fgh\colon \mathfrak{S}\,\mathbb{Q}.\, f\{\le_{\mathbb{Q}}\}g \Rightarrow g\{\le_{\mathbb{Q}}\}h \Rightarrow f\{\le_{Q}\}h \tag{2}$$

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

$$\forall xyz\colon \mathbb{Q}.\, x \le_{\mathbb{Q}} y \Rightarrow y \le_{\mathbb{Q}} z \Rightarrow x \le_{\mathbb{Q}} z$$

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

$$(\forall(x\colon X)(y\colon Y)(z\colon Z).\, Rxyz) \Rightarrow \forall(f\colon \mathfrak{S}X)(g\colon \mathfrak{S}Y)(h\colon \mathfrak{S}Z).\, \mathrm{fold}_\star\,(R \mathbin{♂} f \mathbin{@} g \mathbin{@} h) \tag{3}$$

This yields

$$\forall\, fgh\colon \mathfrak{S}\mathbb{Q}.\, \mathrm{fold}_\star((\lambda xyz.x \le_{\mathbb{Q}} y \Rightarrow y \le_{\mathbb{Q}} z \Rightarrow x \le_{\mathbb{Q}} z) \mathbin{♂} f \mathbin{@} g \mathbin{@} h).$$

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 variable, we would like to symbolically evaluate the expression. We avoid dealing with binders by converting the lambda expression into the combinator expression

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

where $\mathbf{S} := \mathbf{B}(\mathbf{B}(\mathbf{BW})\mathbf{C})(\mathbf{BB})$ and $(\Rightarrow)$ and $(\le_{\mathbb{Q}})$ 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[5], a standard tool for Haskell programmers. It would have been interesting to implement the algorithm for finding the combinator form of a λ-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 4.4. 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 use Coq's `autorewrite` system as part of a small custom tactic to automatically reduce this entire expression in one command, yielding

$$\forall fgh\colon \mathfrak{S}\mathbb{Q}.\, \mathrm{fold}_\star(f\langle\le_{\mathbb{Q}}\rangle g\langle\Rightarrow\rangle g\langle\le_{\mathbb{Q}}\rangle h\langle\Rightarrow\rangle f\langle\le_{\mathbb{Q}}\rangle h).$$

5. `http://www.cse.unsw.edu.au/~dons/lambdabot.html`

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

$$\forall PQ\colon \mathfrak{S}\star.\,(\mathrm{fold}_\star\,(P\,\langle\Rightarrow\rangle\,Q))\Rightarrow \mathrm{fold}_\star P\Rightarrow \mathrm{fold}_\star Q \tag{4}$$

Repeated application of this lemma yields

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

as required.

4.6. 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 $\varepsilon\colon\mathbb{Q}^+$. 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` are the binary positive natural numbers, such that `stepSample`n produces the best approximation of the identity function with n steps.

Using binary positive natural numbers makes it easy to create a well-balanced tree to represent the step function by recursion. This is important because when a step function is integrated, the glue points are recursively multiplied together to compute the length of each step. Having a balanced tree means that $O(n\,(\log n))$ multiplications are done rather than $O(n^2)$.

It is unfortunate that the width of each step is computed 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, we expect the time spent computing the length of the interval is much smaller that the time it takes to sample the continuous function f.

4.7. Timings. The version of Riemann integration that we implemented applies to *general* continuous functions and hence has bad complexity behavior. If we know more about the function, for instance if it is differentiable, faster algorithms can be used, also in the context of exact real arithmetic [Eda99].

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

Table 2. `Time Eval vm_compute in ...` carries out the reduction using Coq's virtual machine. `answer` n asks for an answer to within 10^{-n}. All computations where carried out on an IBM Thinkpad X41.

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.

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 [BK08];
- 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.

An interesting algorithm for Riemann integration is suggested by Simpson [Sim98]. However, the verification of the termination of this algorithm is not possible in the type theory of 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 is correct. 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.

Bibliography

[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.

[BB85] Errett Bishop and Douglas Bridges. *Constructive Analysis*. Number 279 in Grundlehren der mathematischen Wissenschaften. Springer-Verlag, 1985.

[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.

[BK08] Andrej Bauer and Iztok Kavkler. A constructive theory of domains suitable for implementation. http://math.andrej.com/wp-content/uploads/2008/01/constructive-domains.pdf, 2008.

[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.

[Eda99] Abbas Edalat. Numerical integration with exact real arithmetic. In *Automata, Languages and Programming, 26th International Colloquium, ICALPâĂŹ99, Prague, Czech 227 Republic, July 11-15, 1999, Proceedings, volume 1644 of Lecture Notes in Computer Science*, pages 90–104. Springer, 1999.

[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.

[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.

[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.

[O'C05] Russell O'Connor. Essential incompleteness of arithmetic verified by coq. In Joe Hurd and Thomas F. Melham, editors, *TPHOLs*, volume 3603 of *Lecture Notes in Computer Science*, pages 245–260. Springer, 2005.

[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.

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

[Sch08] Helmut Schwichtenberg. Realizability interpretation of proofs in constructive analysis. To appear in ToCS, 2008.

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

[Wad92] P. Wadler. Monads for functional programming. In *Proceedings of the Marktoberdorf Summer School on Program Design Calculi*, August 1992.