A Functional Algorithm for Exact Real Integration with Invariant Measures

A. Scriven [1]

School of Computer Science
University of Birmingham
Birmingham, England

Abstract

We develop an algorithm for exact real integration over a class of self-similar spaces and measures defined by Hutchinson. We construct the algorithm in an idealised lazy functional programming language and prove its correctness using domain theory. This generalises an algorithm developed by Alex Simpson for exact Riemann integration over the real line. We also implement the algorithm in the functional language Haskell and give some preliminary experimental results.

Keywords: Exact real number computation; integration; self-similar; iterated function system; invariant measure; domain theory; lazy functional programming; streams; Haskell.

1 Introduction

Exact real number computation is useful from both a practical and theoretical point of view. On the practical side, it allows calculations to be made without the propagation of errors inherent in floating point arithmetic, albeit at the inevitable loss of efficiency. Theoretically, the study of exact arithmetic sheds light on questions regarding the computability of real numbers and function(al)s. One functional in particular that has been the focus of detailed study is integration over the reals [5,21]. Edalat and Escardó, and later Simpson constructed algorithms that performed integration. There were significant differences in the implementation, but in both cases the essential ingredient was the following:

$$\int_0^1 f\ dx = \tfrac{1}{2}\int_0^1 f(x/2)\ dx + \tfrac{1}{2}\int_0^1 f((x+1)/2)\ dx \tag{1}$$

$$\int_0^1 f(x) + c\ dx = \int_0^1 f(x)\ dx + c \tag{2}$$

[1] Email: msc38ams@cs.bham.ac.uk

Intuitively, the idea is to recursively apply (1) splitting the integrand into simpler and simpler fragments until constants can be extracted using (2). This method has successfully been used to develop exact real algorithms for Riemannian integration on the real line [5,21].

In this paper we generalise Simpson's algorithm [21] over a large class of measurable spaces and measures. The spaces in question are the so called *invariant spaces* [13], which we discuss in Section 2.6. Informally these spaces have the defining property that when a segment is viewed, it resembles the original space topologically. As such, these spaces can be defined recursively, making them computationally appealing. Such spaces arise naturally in mathematics and nature. The real line exhibits this property of invariance, indeed part of this study goes toward showing how it is this characteristic of the real line that allows algorithms such as in [5,21] to work. Other invariant spaces include many fractal spaces such as the Sierpinski gasket and Falconers fractal.

This paper is a condensed version of the masters thesis by the author[20].

1.1 Related Work

In [5], Edalat and Escardó used a form of PCF extended with an extensional datatype `real` to represent the reals, called `RealPCF`. This language required special primitive constructs to work with the abstract dataset `real`, in particular parallel constructs were found to be a necessary component. Moreover in order to evaluate (1) it was necessary to evaluate both sub-integrals in parallel. This non-sequentiality made their integration algorithm difficult to implement in practice, and it was unclear how a sequential equivalent could be constructed. Indeed Gianantonio conjectured in his thesis that no such algorithm existed. However Simpson [21] achieved a sequential integration algorithm using an intensional data type, representing the reals as a stream of digits. This language required no additional primitives, as arithmetic is performed by working directly on the representations and through the use of Bergers universal quantifier [3].

2 Background

2.1 Computational model

In this paper we develop our algorithm in an informal lazy functional programming language, much alike to that used in [21]. As Simpson remarks, such a language "provides a natural implementation style for exact real algorithms", as well as being readable and intuitive. We make use of the following type structure:

$$\sigma, \tau ::= \texttt{Int} \mid \texttt{Bool} \mid \sigma \times \tau \mid \sigma \to \tau \mid [\sigma]$$

Where `Int` and `Bool` are the datatypes of natural numbers and booleans respectively. The constructors application $\to$ and product $\times$ are familiar, the term $[\sigma]$ denotes the datatype of lazy streams of elements of type σ. We shall also make use of finite types for clarity and convenience later in this paper.

The denotational semantics of our language is the standard interpretation from

Scott model of partial continuous functionals P, which form cartesian closed category of dcpos. We assume the reader is familiar with domain theory, and refer them to [1,10,22] for a detailed survey. Denotationally, streams of type $[\sigma]$ can be interpreted as elements of type $P_{\mathtt{Int}} \to P_\sigma$. Thus, in proving correctness we treat our programs as expressions in PCF interpreted in the Scott model, thus maintaining full mathematical rigour. Informally, we may consider our functional programming language to be PCF extended with lazy streams.

2.2 Denotation of streams

For a set X, denote the set of finite sequences of elements in X by X^*, and the set of infinite sequences by X^ω. We write $X^\infty := X^* \cup X^\omega$ for the set of all sequences. For a sequence $\alpha \in X^\infty$ we write $|\alpha|$ for the (possibly infinite) length of α. For $0 \leq i < |\alpha|$, the i^{th} element of α is denoted by α_i. The concatenation of a finite sequence $\alpha \in X^*$ and an arbitrary sequence $\beta \in X^\infty$, is denoted $\alpha\beta$. We write $\alpha\lceil_n$ for the largest prefix β, of α, such that $|\beta| \leq n$. As with domains, for a finite stream α we denote the set of all streams with α as a prefix by $\uparrow \alpha$, and define $\Box\alpha = \uparrow \alpha \cap X^\omega$ to be the subset of infinite streams with α as a prefix. Finally, for $x \in X$, we write x_ω for the infinite constant sequence $xxx...$.

2.3 Universal quantification on streams

In a recent paper, Escardó [7] showed how to extend Bergers universal quantifier to the class of so called *searchable* spaces, including the generalised Cantor spaces over n objects, denoted n^ω. In particular, Escardíntroduces an algorithm `cantor`,

```
type Quantifier a = (a -> Bool) → Bool
cantor :  N → Quantifier (N → N)
```

such that, for any $n \in \mathbb{N}$,

$$[\![\mathtt{cantor}(n)]\!](p) = \begin{cases} \text{true} & \text{if } p(x) = \text{true, for all } x \in n^\omega \\ \text{false} & \text{otherwise.} \end{cases}$$

We shall make use of this algorithm in Section 3 to develop our generalised integration algorithm.

2.4 Exact real arithmetic

There have been numerous approaches to representing the real numbers within a computational framework. Some methods seek to represent the reals extensionally i.e. as an abstract datatype such as the interval domain, while others are intensional in that the reals are represented using existing type structures (see e.g. [9,8] for a summary). In this paper we favour the latter approach, in particular infinite streams of digits as studied by Kreitz et. al. [14] and used by Simpson [21].

For our purposes, a representation of X is a surjective (partial) map $\delta : A \subseteq \mathbb{F} \to X$, where $\mathbb{F} = (\mathbb{N} \to \mathbb{N}) \equiv \mathbb{N}^\omega$. The Baire space $\mathbb{F}$ has the product topology, which coincides with the subspace topology of the Scott topology on $\mathbb{N}^\infty$. Given representations $\delta : A \to X$, $\gamma : B \to Y$ we say that a continuous $f : A \to B$ is

real [2] if there exists a continuous $\tilde{f} : X \to Y$, such that $\gamma \cdot f = \delta \cdot \tilde{f}$. We call $\tilde{f}$ the realization of f. Furthermore we say f is *real-total* if it is total and the restriction to the total function is real. The following definition is due to Kreitz et. al. [14]

Definition 2.1 Let (X, τ) be a T_0-space with countable base. A representation $\delta :\subseteq \mathbb{F} \to X$ is *admissible* (with respect to τ) if it is continuous and for all continuous functions $\phi :\subseteq \mathbb{F} \to X$, there exists a continuous $\theta : \mathbb{F} \to \mathbb{F}$ such that $\phi = \delta \cdot \theta$.

The admissible representations have remarkable properties, namely, every continuous function $f : A \to B$ is real i.e. the representation contains no "junk", and conversely, every continuous $g : X \to Y$ is the realization of some continuous $f : A \to B$ i.e. the representation is large enough. In particular, the quotient topology induced on X by an admissible representation $\delta : A \to X$ corresponds precisely to the existing topology on X, i.e. the representation preserves topological structure. Many typical representations such as decimal expansions and Cauchy sequences fail to be admissible. A typical example of such a case is the lack of a computable algorithm for multiplication by 3 in the decimal representation (see e.g. [9]).

2.5 The signed binary representation

Perhaps the simplest admissible representation is the signed binary representation. Let $\mathbf{3} := \{-1, 0, 1\}$, then the map $q : \mathbf{3}^\omega \to [-1, 1]$ defined by $q(\alpha) = \sum_{i=1}^{\infty} \alpha_i 2^{-i}$ is a surjection, called the *signed binary representation*. This representation is admissible, and has been used successfully by many people in performing exact arithmetic [18,21]. In our lazy functional language we implement 3^ω using an informal finite type

```
type three = {-1,0,1}
type interval = [three]
```

Thus $[\![\texttt{interval}]\!] = \mathbf{3}^\infty$, where only the (total) elements $\mathbf{3}^\omega$ can be interpreted as real numbers. Similarly, only the (total) functions $f : \mathbf{3}^\infty \to \mathbf{3}^\infty$ correspond to functions on real numbers. If this were to be implemented in PCF say then we would perceive $\mathbf{3}^\infty$ as a (closed) subset of $(\mathtt{N} \to \mathtt{N})$, and the corresponding (partial) functionals would only be necessarily defined on $\mathbf{3}\infty$.

The admissibility of the signed binary representation allows us to guarantee that every continuous real valued function has a continuous real-total counterpart in our domain model. The following results are generalisations of those given in [21, Proposition 1].

Lemma 2.2

(i) Given a representation $\delta : A \to X$, any continuous $\theta : X \to [0, 1]$ can be realised by a map $\phi : A \to \mathcal{3}^\omega$ such that $\theta \cdot \delta = q \cdot \phi$.

(ii) Let σ be a type in our language. Then for any continuous $\phi : T_\sigma \to \mathcal{3}^\omega$, there exists a total $\phi : D_\sigma \to \mathcal{3}^\infty$ such that ϕ restricts to θ.

Proof.

[2] this terminology stems from [21]

(i) Since $q : \mathbf{3}^\omega \to [-1,1]$ is admissible, for any surjective $f :\subseteq \mathbb{F} \to [-1,1]$ there exists a continuous $\phi : \mathbb{F} \to \mathbb{F}$ such that $f = q \cdot \phi$ on $\mathrm{dom}(f)$. Taking $f = \delta \cdot \theta$ suffices.

(ii) Immediate from T_σ being dense in D_σ, and 3^∞ being a Scott domain and therefore densely injective.

□

2.6 Invariant spaces and measures

Many interesting spaces can be constructed by repeatedly applying a set of simple transformations to an initial space. For example, the Cantor space embedded in $\mathbb{R}$, as shown in Figure 1, can be constructed by repeatedly applying the following set of maps

$$\mathcal{C} = \{S_1 : x \mapsto \tfrac{x}{3}, S_2 : x \mapsto \tfrac{x+2}{3}\} \text{ where } \mathrm{dom}(S_i) = [0,1] \tag{3}$$

Fig. 1. The first 3 iterations generating the Cantor space, embedded in the real line.

The following definition generalised this concept, and is due to Hutchinson [13].

Definition 2.3 Let (X,d) be a complete metric space. A closed, bounded, non-empty $K \subseteq X$ is called *invariant* w.r.t a (finite) set of contraction maps $\mathscr{S} = \{S_0, ..., S_n\}$ if:

$$K = \bigcup_{i=1}^{n} S_i(K)$$

The set $\mathscr{S}$ is often referred to as an *Iterated Function System* or *IFS* [12]. The startling result discovered by Hutchinson in [13] is that a (unique) invariant set always exists for any given $\mathscr{S}$.

Theorem 2.4 *Let (X,d) be a complete metric space. Then for any (finite) set of contractions $\mathscr{S}$ on X, there exists a unique closed bounded non-empty $K \subseteq X$ which is invariant w.r.t $\mathscr{S}$. Moreover K is compact.*

Throughout this paper we denote the invariant space w.r.t $\mathscr{S}$ by $|\mathscr{S}|$, or by K when there is no ambiguity as to the IFS.

2.7 The co-ordinate map

In order to perform constructive analysis on invariant spaces, we need a suitable representation. A natural candidate is the co-ordinate map $\pi : n^\omega \to |\mathscr{S}|$, defined

by

$$\{\pi(\alpha)\} = \bigcap_{i=1}^{\infty} S_{\alpha_1 \ldots \alpha_i}(X)$$

This map is surjective and therefore a representation of $|\mathscr{S}|$. Moreover it is continuous w.r.t the product topology on n^ω and the subspace topology on $|\mathscr{S}|$ inherited from the metric space (X, d). However, in general π is not admissible, an obvious example being the familiar binary representation $\pi : \mathbf{2}^\omega \to [0, 1]$ where $S_0 : x \mapsto \frac{x}{2}$ and $S_1 : x \mapsto \frac{x+1}{2}$

2.8 Invariant measures

It was shown in [13] that just as $|\mathscr{S}|$ is exhibited as the fixed point of $\mathscr{S}$ applied to X, a natural measure on $|\mathscr{S}|$ exists as the fixed point of an analogous application of $\mathscr{S}$ on $\mathcal{M}$, the set of Borel regular measures on X having bounded support and finite mass. Hutchinson used these "invariant measures" to differentiate between IFS's that gave the same invariant space but were inherently different. However the same measures are natural candidates for integration.

Suppose we have a set $\rho = \{\rho_1, ..., \rho_n\}$ such that $\rho_i \in (0, 1)$ and $\sum \rho_i = 1$. Given a set of contractions $\mathscr{S}$, define $(\mathscr{S}, \rho) : \mathcal{M} \to \mathcal{M}$ by:

$$(\mathscr{S}, \rho)(\mu)(A) = \sum_{i=1}^{n} \rho_i \mu(S_i^{-1}(A))$$

It is clear from the definition that $M((\mathscr{S}, \rho)(\mu) = M(\mu)$ so the map restricts to $(\mathscr{S}, \rho) : \mathcal{M}^1 \to \mathcal{M}^1$, where $\mathcal{M}^1 \subseteq \mathcal{M}$ is the subset of measures having mass $M(\mu) = 1$.

Theorem 2.5 *There exists a unique $\mu \in \mathcal{M}^1$ such that $(\mathscr{S}, \rho)\mu = \mu$.*

As an immediate corollary we obtain the following identity, which motivates the next section.

$$\int_{|\mathscr{S}|} f \, d\|\mathscr{S}, \rho\| = \sum_{i=1}^{n} \rho_i \int_{|\mathscr{S}|} (f \cdot S_i) \, d\|\mathscr{S}, \rho\| \tag{4}$$

3 Exact integration on invariant spaces

The definition of invariant spaces captures the property of the real line used by [21,5] to compute integration. Equation (4) generalises the identity $\int_0^1 f(x)\,dx = \frac{1}{2}\int_0^1 f(x/2)\,dx + \frac{1}{2}\int_0^1 f((x+1)/2)\,dx$. This special case is arguably the simplest form of integration on a self-similar space, and it suffices to only compute averages of two real numbers. For general invariant spaces, it is clear from (4) that we need an algorithm for weighted sums of arbitrarily many real numbers. In order to develop an algorithm for this, we consider an alternative representation of the unit interval.

3.1 A "streams-of-streams" representation of the unit interval

In order to be able to develop a Riemann integration algorithm, Simpson needed the averaging algorithm to have the property that $|\texttt{avg}(x, y)| \geq \min(|x|, |y|)$ i.e.

for every digit of input from x and y, `avg` outputs at least a single digit. We will call such programs *stable*. Simpson showed that no stable algorithm for averaging existed in the signed binary expansion, and solved the problem by making use of an intermediate representation, namely the dyadics $\mathbb{D} := \mathbb{Q}_d \cap [-1,1]$ with representation $q_d : \mathbb{D} \to [-1,1]$ defined by $q_d(\alpha) = \sum_{i=1}^{n} \alpha_i 2^{-i}$ as a natural extension of q. In our more general setting, we need yet another intermediate representation, as the following result illustrates.

Lemma 3.1 *There is no stable algorithm for performing weighed averaging in the signed binary* or *dyadic arithmetic.*

Proof. The existence of a stable weighted averaging algorithm would also imply the existence of a stable multiplication algorithm. A simple counter example is to consider the width of the interval $q(\Box 11) \times q(\Box 10) = [\frac{1}{8}, \frac{3}{4}]$. □

It is clear that this problem is not unique to the signed binary expansion, but relates to carry overs in the arithmetic. While a possible solution would be to the expand the digit set to include all of $\mathbb{Q} \cap [-1,1]$, allowing multiplication to be performed digit-wise, this can lead to "integers explosion" in the denominator and numerator. To sidestep this problem, we make use of a stream-of-streams representation i.e. our digits are themselves streams representing real numbers.

Definition 3.2 [q^ω-representation] Define the space $\mathbf{3}^{\omega\omega} := (\mathbf{3}^\omega)^\omega$ and the representation $q^\omega : \mathbf{3}^{\omega\omega} \to [-1,1]$ by,

$$q^\omega(\alpha) = \sum_{i=1}^{\infty} q(\alpha_i) 2^{-i}$$

Since $\mathbf{3}^{\omega\omega}$ has the same cardinality as the continuum we can embed it in $\mathbb{F}$ and the machinery from [15] carries over with no difficulty. In particular, q^ω is admissible: it is continuous, and for any continuous $\phi :\subseteq \mathbb{F} \to [-1,1]$, there is a continuous $\theta : \mathbb{F} \to \mathbb{F}$ such that $\phi = q \cdot \theta$, thus $\phi = q^\omega \cdot \theta^\omega$ where $\theta^\omega(\alpha) = \theta(\alpha)^\omega$. We implement $\mathbf{3}^{\omega\omega}$ in our functional programming language, using the data type:

```
type w-interval = [interval]
```

In this representation is is possible to implement a stable weighted sum algorithm, presented in Figure 2.

Lemma 3.3

(i) $[\![mul]\!]$ *is real-total with* $q^\omega([\![mul]\!](a,\alpha)) = q(a) \times q^\omega(\alpha)$ *for all* $a \in 3^\omega$, $\alpha \in 3^{\omega\omega}$.

(ii) *On the domain* $\{(x,y) : |q(x_i) + q(y_i)| \leq 1$ *for all* $i\ \}$, $[\![add]\!]$ *is real-total with* $q^\omega([\![add]\!](x,y)) = q^\omega(x) + q^\omega(y)$

Proof. Immediate from the definitions of `add` and `mul`. □

Lemma 3.4 *For all* $p : \mathbb{N} \to \mathbf{3}^\omega$, $f : \mathbb{N} \to \mathbf{3}^{\omega\omega}$ *with* p_i *and* f_i *defined for* $i = 1, ..., n$,

(i) `w-sum` *is total, i.e.* $[\![w\text{-}sum]\!](n,p,f) \in 3^{\omega\omega}$

(ii) `w-sum` *is correct, i.e.* $q^\omega([\![w\text{-}sum]\!](n,p,f)) = \sum_{i=1}^{n} \tilde{p}_i.\tilde{f}_i$

```
mul ::  interval → w-interval → w-interval
a `mul` (b : y) = (a ∗ b :  a `mul` y)

add ::  w-interval → w-interval → w-interval
(a : x) `add` (b : y) = (a + b :  (x `add` y) )

w-sum ::  ( N , N → interval , N → w-interval ) → w-interval
w-sum(0,_,_) = 0_ωω
w-sum(n,p,f) = p(n)∗f(n) `add` w-sum(n − 1,p,f)
```

Fig. 2. The weighted sum algorithm: The operators + and ∗ denote addition and multiplication on the datatype `interval` respectively, as implemented in [18] for example. Note that there is no bounds checking on +.

(iii) ***w-sum*** *is stable, i.e.* $|[\![w\text{-}sum]\!](n,p,f)| \geq \min_{0\leq i\leq n} |f_i|$

Proof. [Sketch] Both (i) and (ii) follow by induction on n. For (iii), it is clear that for both `mul` and `sum`, are stable, where here the "digits" of $\mathbf{3}^{\omega\omega}$ are the streams $\mathbf{3}^{\omega}$. Since `w-sum` is a finite composition of these functions, it is also stable. □

We can easily convert from `w-intervals` to `intervals` using the algorithm `shift` in Figure 3. We make use of a secondary nine-digit representation $q' : \mathbf{9}^{\omega} \to [-1,1]$ given by $q'(\alpha) = \sum_{i=1}^{\infty} \alpha_i 2^{-(i+2)}$.

```
type nine = {−4,−3,−2,−1,0,1,2,3,4}
type nine-stream = [nine]
coerce ::  nine-stream → interval
coerce(a:b:x) = let c = 2 ∗ a + b in cases
   c < −4     then −1:coerce(c + 8:x)
   c > 4      then  1:coerce(c − 8:x)
   otherwise  then  0:coerce(c:x)

shift' ::  w-interval → nine-stream
shift'( (a : b : x) :  (c : y) :  z ) = let d = 2a + b + c in
   d:shift'( (x `q-avg` y) :  z)

shift = coerce.shift'
```

Fig. 3. An algorithm for converting from $\mathbf{3}^{\omega\omega}$ to $\mathbf{3}^{\omega}$.

Lemma 3.5

(i) For all $\alpha \in \mathbf{9}^{\omega}$, ***coerce***$(\alpha) \in \mathbf{3}^{\omega}$ *and* $[\![\widetilde{coerce}]\!] = id$ *i.e.* $q([\![\mathbf{coerce}]\!](\alpha)) = q'(\alpha)$.

(i) For all $\gamma \in \mathbf{3}^{\omega\omega}$, ***shift***$(\gamma) \in \mathbf{3}^{\omega}$, *and* $[\![\widetilde{shift}]\!] = id$ *i.e.* $q([\![shift]\!](\gamma)) = q_{\omega}(\gamma)$.

Proof. It is a simple exercise to show (i). Whence for (ii) it suffices to prove that `shift'` is real-total with $[\![\widetilde{\texttt{shift'}}]\!] = \mathrm{id}$. Since `shift'` outputs one digit for every two digits[3] input, it is total. To prove (ii), let $\gamma = ((a : b : \alpha) : (c : \beta) : \delta)$, then:

[3] where here a digit is an element in $\mathbf{3}^{\omega}$.

$$\begin{aligned} q_\omega(\gamma) &= \tfrac{1}{2}q(a:b:\alpha) + \tfrac{1}{4}q(c:\beta) + \tfrac{1}{4}q_\omega(\delta) \\ &= \frac{\frac{1}{2}a + \frac{1}{4}b + \frac{1}{4}q(\alpha)}{2} + \frac{\frac{1}{2}c + \frac{1}{2}q(\beta)}{4} + \frac{q_\omega(\delta)}{4} \\ &= \frac{1}{8}(2a+b+c) + \frac{1}{2}\left(\frac{q(\alpha)\oplus q(\beta)}{2} + \frac{q_\omega(\delta)}{2}\right) \\ &= \tfrac{1}{8}d + \tfrac{1}{2}q_\omega([\![\text{q-avg}]\!](\alpha,\beta):\delta) \end{aligned}$$

where $d = 2a + b + c$. The result follows. □

It is worth noting that we could use the dyadic representation [4] from [21] instead of `nine-stream`, since this has the added benefit of greater computational efficiency in performing multiplication. The only modification to the algorithm would be that `shift'` outputs $d/4$ rather than d. We can convert from dyadics to signed binary using the `coerce` algorithm from [21].

3.2 A first integration algorithm

With an intensionally stable weighted sum algorithm, we are now able to produce a generalised integration program over any (computable) invariant space. We present this algorithm in Figure 4.

```
type Baire = [N]

w-int ::  ((Baire → interval) , N , N → interval) → w-interval
w-int(f,n,p) = let d = head(f(1_ω)) in
    if (forall(n))(λv.head(f(v)) == d)
       then d^ωω  :  w-int(tail.f,n,p)
       else w-sum(n,p,λi.w-int(λv.f(i:v),n,p))

integrate = shift.w-int
```

Fig. 4. The generalised integration algorithm: Here `Baire` has denotation $[\![\text{Baire}]\!] = \mathbb{N}^\infty$ and so is the datatype containing the Baire space $\mathbb{N}^\omega$. **forall(n)** is the universal quantifier over the Cantor space on $\{1,...,n\}$, which is a subset of $\mathbb{N}^\omega$.

Proposition 3.6 *Let $\rho : \mathbb{N} \to 3^\omega$ be such that $\tilde{\rho}(i) \in [0,1]$ for $i = 1,...,n$ and $\sum_i \tilde{\rho}(i) = 1$. Then, for any continuous real-total $\phi : n^\omega \to 3^\omega$ it holds that $[\![integrate]\!](\phi, n, \rho) \in \mathbb{3}^w$ and*

$$q([\![integrate]\!](\phi,n,\rho)) = \int_K \tilde{\phi}\, d\mu$$

where $\mu = \|\mathscr{S}, \rho\|$ is the invariant measure of K.

Note that Lemma 2.2 guarantees that any continuous function $f : K \to [0,1]$ has a continuous real-total realizer of the form $\phi : n^\omega \to 3^\omega$. The proof is similar to a sketch given in [21].

[4] In fact this was our first approach, however the nine digit method is more elegant and simpler to implement.

Proof. `integrate` is total since `shift` is total and `w-sum` is stable. By Lemma 3.5 it is sufficient to prove $q^\omega([\![\texttt{w-int}]\!](\phi)) = \int_K \tilde{\phi}\, d\mu$. We will prove by induction on m that

$$\left| q^\omega([\![\text{w-int}]\!](\phi))\lceil_m - \int_K \tilde{\phi}(x)\, d\mu \right| \leq 2^{-m}$$

Where we have used the abuse of notation $q^\omega(\alpha)\lceil_m := \sum_{i=1}^m q(\alpha_i)2^{-i}$ for convenience[5] The case $m = 0$ is trivial. For $m > 0$, consider $h(\phi) : n^\omega \to \mathbf{3}$ defined by $h(\phi)(\alpha) = \text{head}(\phi(\alpha))$. Since ϕ is total, so is $h(\phi)$. We now proceed by inner induction on $emc(h(\phi))$, the extensional modulus of continuity.

If $emc(h(\phi)) = 0$ then $h(\phi) = d$ is constant. In particular, for all v

$$\phi(v) = d : [\![\text{tail}]\!](\phi(v)) \tag{5}$$

$$\Rightarrow \tilde{\phi}(v) = \tfrac{1}{2}d + \tfrac{1}{2}\widetilde{[\![\text{tail}]\!]}(\tilde{\phi}(v)) \tag{6}$$

In this case, `w-int` outputs:

$$\texttt{w-int}(\phi) = d : \texttt{w-int}(\texttt{tail}.\phi)$$

Thus,

$$[\![\text{w-int}]\!](\phi)\lceil_m = d : [\![\text{w-int}]\!](\text{tail}.\phi)\lceil_{m-1} \tag{7}$$

$$\Rightarrow q^\omega([\![\text{w-int}]\!](\phi))\lceil_m = \tfrac{1}{2}d + \tfrac{1}{2}q^\omega([\![\text{w-int}]\!](\text{tail}.\phi))\lceil_{m-1} \tag{8}$$

Where we have again used the notation $q(\alpha)\lceil_m$ to denote the partial evaluation $\sum_{i=1}^m \alpha_i 2^{-i}$. By the inductive hypothesis on m.

$$\left| q^\omega([\![\text{w-int}]\!]([\![\text{tail}]\!](\phi)))\lceil_{m-1} - \int_K \widetilde{[\![\text{tail}(\phi)]\!]}\, d\mu \right| \leq 2^{1-m}$$

by 5 and 7 $$\left| 2q^\omega([\![\text{w-int}]\!](\phi))\lceil_m + d - \int_K (2\tilde{\phi}(x) - d)\, d\mu \right| \leq 2^{1-n}$$

$$\left| q^\omega([\![\text{w-int}]\!](\phi))\lceil_m - \int_K \tilde{\phi}(x)\, d\mu \right| \leq 2^{-m}$$

where we have made use of the identity,

$$\int_K f(x) + c\, d\mu = \int_K f(x)\, d\mu + c \tag{9}$$

For $emc(h(\phi)) > 0$, we have:

$$\texttt{w-int}(f) = \texttt{w-sum}(n, p, \lambda i.\texttt{w-int}(\lambda v.f(i : v), n, p))$$

Since $emc(f.\texttt{cons}_i) < emc(f)$, it follows by the inner inductive hypothesis that $q^\omega([\![\texttt{w-int}]\!](\phi \cdot \texttt{cons}_i)) = \int_K (\tilde{\phi} \cdot \widetilde{[\![\texttt{cons}_i]\!]})(x)\, d\mu$. By Lemma 3.4 we can say that, for all m:

$$\left| \sum_{i=1}^n \tilde{\rho}_i \int_K \widetilde{f \cdot \texttt{cons}_i}\, d\mu - q^\omega\Big([\![\text{w-sum}]\!](n, \rho, \lambda i.(f \cdot \texttt{cons}_i))\Big)\Big\lceil_m \right| \leq 2^{-m}$$

By (4) the result now follows. □

[5] it should be noted that we are not extending q over $\mathbf{3}^\infty$, this is discussed in e.g [9].

3.3 A more general integration algorithm

Most "interesting" functions of the form $f : |\mathscr{S}| \subseteq \mathbb{R}^m \to \mathbb{R}$ are very hard to realize in the form $f : n^\omega \to \mathbf{3}^\omega$. In order to integrate over more useful functions we modified the algorithm to integrate over (a representation of) the metric space X containing $\mathscr{S}|$. This still gives the same result, as $\mathrm{spt}(\|\mathscr{S}, \rho\|) = |\mathscr{S}|$. The only condition on the representation of X is that it is *searchable*, in order that we can quantify over it.

We summarise the modified algorithm in Figure 5. The data type `IFS` defined therein encapsulates the necessary information of the IFS in order to perform integration. Note that the original algorithm can be obtained from `x-integrate` given input $(0_\omega, \mathtt{cantor}(n), n, \lambda i.\mathtt{cons}_i, \rho)$, so we may think of `x-integrate` as a generalised algorithm.

```
type IFS a =
(a, Quantifier a, N, N → (interval → interval), N → interval)

x-int ::  IFS a → (Baire → interval) → w-interval
x-int 𝒮 f = let d = head(f(ι)) in
   if ∀_X(λv.head(f(v)) == d)
      then d_ωω :  x-int(tail.f,𝒮)
      else w-sum(n,ρ,λi.x-int(f.S_i,𝒮))
   where (ι, ∀_X, n, S, ρ) = 𝒮

x-integrate = shift.x-int
```

Fig. 5. A more practical integration algorithm.

There is a subtle complication to be addressed here: When K is represented by n^ω, the contractions S_i are realized as $\mathtt{cons}_i$, which is vital for the inductive step in reducing the *emc* of the integrand. Formally, for a given representation q we need the S_i to be such that $|S_i(x)| > |x|$. We call such functions *computationally contractive* with respect to q, or simply computationally contractive (c.c.) when there is no ambiguity as to the representation.

Unfortunately, under a general representation $\delta :\subseteq \mathbb{F} \to X$, the maps $\mathscr{S}$ are not guaranteed to be c.c. under δ. For example the map $x \mapsto \frac{x}{3}$ has no c.c. representation in the signed binary arithmetic. This seems to potentially limit the success of using an arbitrary representation space. However the following result counters this problem.

Theorem 3.7 *Given any set of functions $S_i : \mathbf{3}^\infty \to \mathbf{3}^\infty$ corresponding to and IFS $\mathscr{S}$, as well as coefficients ρ, we can construct a set of computationally contractive functions $S'_i : \mathbf{3}^\infty \to \mathbf{3}^\infty$ corresponding to an IFS $\mathscr{S}'$, and a set of coefficients ρ' such that:*

(i) *$\mathscr{S}$ and $\mathscr{S}'$ induce the same invariant space i.e. $|\mathscr{S}'| = |\mathscr{S}|$*

(ii) *$\mathscr{S}, \rho$ and $\mathscr{S}', \rho'$ induce the same invariant measure i.e. $\|\mathscr{S}', \rho'\| = \|\mathscr{S}, \rho\|$*

In short this result states that any invariant space we care to define can be

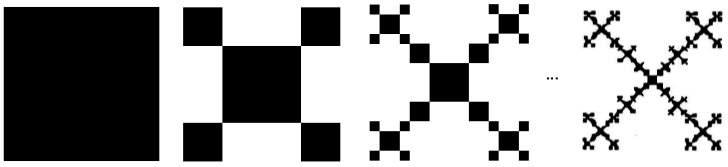
Space	Partial output	Approx. value	Analytic value	Time (secs)	Memory (bytes)
Falconer	$[1,-1,1,-1,1]$	$0.34375 \pm \frac{1}{32}$	$1/3$	20.65	$7.1 \cdot 10^8$
Sierpinski	$[1,-1,1,0,1]$	$0.40625 \pm \frac{1}{32}$	$11/27$	113.39	$4.2 \cdot 10^9$

Fig. 6. Results from the integration program applied to various self-similar spaces: The tests were run on a Linux machine with 4, 2.40 GHz Intel® Xeon™ CPU's, with 4GB's of RAM and 8GB's of swap.

generated by a set of contractions that have c.c. realizers. We prove the result in section 3.5, and for now satisfy ourselves with some examples.

3.4 Experimental results

We implemented a variation of our algorithm in Haskell and ran some initial experiments on various functions. For our experiments we took $X = [-1,1]^2$, represented by $q^2 : \mathbf{3}^\omega \times \mathbf{3}^\omega \to [-1,1]^2$ in the obvious way. We implemented IFS's for the Sierpinski gasket, and Falconers Fractal (see Figure 7). The integrand itself was $(x,y) \mapsto x^2$.



(a) Falconers Fractal

(b) A variant of the Sierpinski gasket.

Fig. 7. invariant spaces.

3.5 Proof of Theorem 3.7

We now provide a proof to Theorem 3.7. The is proof constructive, which is essential from a practical point of view: If we simply knew that a suitable $\mathscr{S}'$ existed but did not know how to construct it, then we would be no better off than before. We shall prove Theorem 3.7 in an informal manner, consisting of two steps:

(i) We show how, given any set of contractions $\mathscr{S}$, we can produce an equivalent (albeit larger) set of contractions with arbitrarily small contractivity.

(ii) We show how the realizer of a function with sufficiently small contractivity can be manipulated so that it is computationally contractive.

Step 1

We start first with some definitions from [13]: Let I be a finite subset of finite ordered tuples i.e. $I \subseteq \{1,...,n\}^*$. Define $\hat{I} = \{\alpha_1...\alpha_p...|\alpha_i \in I\} \subseteq n^\omega$ where we are concatenating finite tuples from I in the obvious way. We say I is *secure* if for

every $\beta \in n^\omega$ there exists a $\alpha \in I$ such that $\alpha \sqsubseteq \beta$ i.e. α is a prefix of β. Moreover I is *secure* if this α is unique. The following result is due to Hutchinson [13].

Theorem 3.8 *Let I be a finite set as above and $\mathscr{S}$ a set of contractions,*

(i) *Let $\mathscr{S}_I = \{S_\alpha | \alpha \in I\}$. Then $|\mathscr{S}_I| = \{k_\beta | \beta \in \hat{I}\}$ where $k_\alpha := \bigcap_{p=1}^{\infty} S_{\alpha_1 \ldots \alpha_p}(K)$. Moreover if I is secure then $|\mathscr{S}_I| = |\mathscr{S}|$.*

(ii) *Let ρ be coefficients inducing a measure on $\mathscr{S}$. Define $\rho_I : I \to (0,1)$ by $\rho_I(i_1, ..., i_p) = \rho(i_1)...\rho(i_p)$. Then if I is tight one can check that $\sum_{\alpha \in I} \rho_I(\alpha) = 1$ and furthermore $\|\mathscr{S}_I, \rho_I\| = \|\mathscr{S}, \rho\|$.*

So now given a set $\mathscr{S}$ with $L = \max_{i=1,..,n} \mathrm{Lip}(S_i) < 1$, let $I = \{\text{all permutations of } < 1, ..., m >\}$ for some integer $m > 0$. Then I is tight, $|\mathscr{S}_I| = |\mathscr{S}|$, $\|\mathscr{S}_I, \rho_I\| = \|\mathscr{S}, \rho\|$ for any suitable ρ and $L_I := \max_{\alpha \in I} \mathrm{Lip}(S_\alpha) = L^m$. Since L is necessarily less than 1, we can therefore make L_I arbitrarily small.

Step 2

For the second step, we start by outlining some algorithms and stating some results. The functions `addone` and `subone` in Figure 3.5 are due to Plume [18], and have the following properties:

```
addone,subone ::  I → I

addone (1:x) = 1_ω
addone (0:x) = 1:addone(x)
addone (−1:x) = 1:x

subone (1:x) = −1:x
subone (0:x) = −1:subone(x)
subone (−1:x) = −1_ω
```

Fig. 8. The `addone` and `subone` algorithms.

Lemma 3.9 *For all $x \in [-1, 1]$.*

(i) $\widetilde{[\![\texttt{subone}]\!]}(x) = \begin{cases} x - 1 & \text{when } x > 0 \\ -1 & \text{otherwise} \end{cases}$

(ii) $\widetilde{[\![\texttt{addone}]\!]}(x) = \begin{cases} x + 1 & \text{when } x < 0 \\ 1 & \text{otherwise} \end{cases}$

Proof. See e.g. [18] □

Next define the q-equivalence relation, $x \equiv_q y \iff q(x) = q(y)$. From Lemma 3.9 the following result is apparent,

Corollary 3.10 *For any $x \in \mathscr{Z}^\omega$,*

(i) if $q(x) \geq 0$ then $1 : \texttt{p-one}(x) \equiv_q x$

(ii) if $q(x) \leq 0$ then $-1 : \texttt{p-negone}(x) \equiv_q x$

```
p-one, p-negone, p-zero ::  I → I

p-one(0:x) = subone(x)
p-one(1:x) = x

p-negone(1:x) = addone(x)
p-negone(−1:x) = x

p-zero(0:x) = x
p-zero(1:x) = addone(x)
p-zero(−1:x) = subone(x)
```

(iii) if $-\frac{1}{2} \leq q(x) \leq \frac{1}{2}$ *then* $0 : \texttt{p-zero}(x) \equiv_q x$

Now suppose we have a function $S : 3^\infty \to 3^\infty$ with contractivity less than $\frac{1}{4}$. Let $c = 4S(0_\omega)_0 + 2S(0_\omega)_1 + S(0_\omega)$.

- If $c > 3$ then $S(0_\omega) > \frac{1}{4}$ and since S has contractivity less than $\frac{1}{4}$ it is true that $S(x) > 0$ for all $x \in 3^\omega$. Thus the function $S' \equiv \lambda x. -1 : \text{p-negone} \cdot S(x)$ is q-equivalent to S i.e. $S' \equiv_q S$. Moreover S' is *computationally contractive.*
- Similarly if $c < -3$ then $S(0_\omega < -\frac{1}{4}$ and $S \equiv_q \lambda x.1 : \text{p-one} \cdot S(x)$
- Finally if $-2 \leq c \leq 2$ then $-\frac{1}{4} \leq S(0_\omega) \leq \frac{1}{4}$ and $S \equiv_q \lambda x.0 : \text{p-zero} \cdot S(x)$

So for suitably contractive functions, we can find an equivalent c.c. set of realizers. Thus provided we have prior knowledge as to the contractivity of each $S_i \in \mathscr{S}$ we can construct an equivalent $\mathscr{S}_I$ with contractivity $< \frac{1}{4}$ and then manipulate these functions into a computationally contractive form. This makes our generalised algorithm universally applicable, with the only condition being that we have some domain knowledge, which is not an unreasonable assumption.

The crucial point here is that given any $\mathscr{S}$ we can either find a suitable c.c. representation of the functions in $(\mathbf{3}^\infty \to \mathbf{3}^\infty)$ or, failing that, generate an equivalent c.c. set that induces the same invariant space and measure.

4 Conclusion

In this study we have extended Simpson's algorithm in two ways. Firstly integration over all computable invariant spaces $|\mathscr{S}|$, and secondly w.r.t all computable invariant measures $\|\mathscr{S}, \rho\|$. In proving correctness we made essential use of domain theory. This is are purely theoretical result, and to the best of our knowledge is the first case of such an algorithm in a functional sequential language.

The crucial step in this study was the use an intermediate "streams-of-streams" representation, in order to compute multiplication digit-wise and thus make our algorithm total. We have shown how the signed binary arithmetic cannot contain a suitable multiplication algorithm, and hinted at how this may generalise to other representations. We may think of our stream-of-streams representation as the natural extension of the dyadics, as used by Simpson, to achieve integration in [21].

The characteristic of the real line that motivated the integration algorithms in

[21,5] is a fundamental trait of all invariant spaces. Moreover it is the defining characteristic of the invariant measures on invariant spaces i.e. every space that can be integrated in this way is an invariant space w.r.t some $\mathscr{S}$ and conversely.

From a practical point of view, the algorithm `x-int` provides more flexibility in the definition of the integrands, and so is more useful. Initial results indicate that the algorithm performs very slowly in practice. This is to be expected however, since when an IFS $\mathscr{S}$ contains n contractions, the algorithm can potentially branch exponentially with order n. This problem was also observed by Simpson, who's algorithm corresponds to the case $n = 2$.

4.1 Further Work

Theorem 3.7 was originally part of an algorithm that implemented the co-ordinate map $\pi : n^\omega \to X$, for an arbitrary representation X. The intention was to use the original integration algorithm `integrate` and convert the integrand $f : X \to \mathbf{3}^\omega$ to $g = f \cdot \pi : N^\omega \to \mathbf{3}^\omega$. However, the additional complexity of g proved to be unacceptable in practice. From a theoretical perspective, the existence of an algorithm for π is of interest in itself, and we hope to explore this further in later work.

Acknowledgments

I am deeply grateful to Martín Escardó for the time, effort and continued encouragement he has given me. In particular I would like to credit him with the original idea for the algorithm `integrate`, and the idea of using the stream-of-streams representation.

References

[1] Abramsky, S. and A. Jung, *Domain theory*, , **3**, Clarendon Press, 1994 pp. 1–168.

[2] Bauer, A., M. Escardó and A. Simpson, *Comparing functional paradigms for exact-real computation*, in: *Automata, landuages and programming*, Lecture notes in Computer Science **2380**, Springer, 2007 pp. 489–500.

[3] Berger, U., "Totale Objecte und Mengen in Bereichstheorie," Ph.D. thesis, University of Munich (1990), phD Thesis.

[4] Brattka, V. and P. Hertling, *Topological properties of real number representations*, Theoretical Computer Science **284** (2002), pp. 241–257.

[5] Edalat, A. and M. Escardó, *Integration in Real PCF*, Information and Computation **160** (2000), pp. 128–166.

[6] Escardó, M., *PCF extended with real numbers*, Theoretical Computer Science **162** (1996), pp. 79–115.

[7] Escardó, M., *Infinite sets that admit fast exhaustive search*, Proceedings of LICS'2007 (2007), pp. 443–452.

[8] Geuvers, H., M. Niqui, B. Spitters and F. Wiedijk, *Constructive analysis, types and exact real numbers*, Mathematical Structures in Computer Science **17** (2007), pp. 3–36.

[9] Gianantonio, P. D., *Real number computability and domain theory*, Information and Computation **127** (1996), pp. 12–25.

[10] Gierz, G., K. Hofmann, K. Keimel, J. Lawson, M. Mislove and D. Scott, "Continuous Lattices and Domains," Cambridge University Press, 2003.

[11] Henrikson, J., *Completeness and total boundedness of the hausdorff metric*, MIT Undergraduate Journal of Mathematics (1999).

[12] Hoggar, S. G., "Mathematics for Computer Graphics," Cambridge University Press, 1994.

[13] Hutchinson, J. E., *Fractals and self similarity*, Indiana University Mathematics Journal **30** (1981), pp. 713–747.

[14] Kreitz, C. and K. Weihrauch, *Theory of representations*, Theoretical Computer Science **38** (1985), pp. 35–53.

[15] Kreitz, C. and K. Weihrauch, *Representations of the real numbers and of the open subsets of the set of real numbers.*, Annals of Pure and Applied Logic **35** (1987), pp. 247–260.

[16] Normann, D., *Computability over the partial continuous functionals*, Journal of Symbolic Logic **65** (2000), pp. 1133–1142.

[17] Plotkin, G., *LCF considered as a programming language*, Theoretical Computer Science **5** (1977), pp. 223–255.

[18] Plume, D., *A calculator for exact real number computation* (1998), 4th year project.

[19] Scott, D., *A theory of computable functions of higher types* (1969), unpublished seminar notes, University of Oxford.

[20] Scriven, A., "Functional algorithms for Exact Real Integration over Invariant Measures," Master's thesis, Department of Computer Science, University of Birmingham (2007).

[21] Simpson, A. K., *Lazy functional algorithms for exact real functionals*, Lecture Notes in Computer Science **1450** (1998), pp. 456–465.

[22] Streicher, T., "Domain-Theoretic Foundations of Functional Programming," World Scientific Publishing, 2006.