Lazy Evaluation in Infinite-Dimensional Function Spaces with Wavelet Basis

Justus Sagemüller
Faculty of Engineering and Science
Western Norway University of Applied Science
Bergen, Norway
jsag@hvl.no

Olivier Verdier
Faculty of Engineering and Science
Western Norway University of Applied Science
Bergen, Norway
olivier.verdier@hvl.no
Department of Mathematics
KTH-Royal Institute of Technology
Stockholm, Sweden
olivierv@kth.se

Abstract

Vectors in numerical computation, i.e., arrays of numbers, often represent continuous functions. We would like to reflect this with *types*. One apparent obstacle is that spaces of functions are typically infinite-dimensional, while the code must run in finite time and memory.

We argue that this can be overcome: even in an infinite-dimensional space, the vectors *can* in practice be stored in finite memory. However, *dual vectors* (corresponding essentially to *distributions*) require infinite data structure. The distinction is usually lost in the finite dimensional case, since dual vectors are often simply represented as vectors (by implicitly choosing a scalar product establishing the correspondence). However, we shall see that an explicit type-level distinction between functions and distributions makes sense and allows directly expressing useful concepts such as the Dirac distribution, which are problematic in the standard finite-resolution picture.

The example implementation uses a very simple local basis that corresponds to a Haar Wavelet transform.

CCS Concepts • **Mathematics of computing → Approximation**; *Distribution functions*; *Arbitrary-precision arithmetic*; *Discretization*; *Continuous functions*; Coding theory; Integral equations; • **Computing methodologies → Representation of mathematical functions**; Image compression; • **Applied computing →** *Mathematics and statistics.*

Keywords wavelet, multiresolution, lazy evaluation

FHPNC '19, August 18, 2019, Berlin, Germany

ACM ISBN 978-1-4503-6814-8/19/08…$15.00
https://doi.org/10.1145/3331553.3342615

ACM Reference Format:
Justus Sagemüller and Olivier Verdier. 2019. Lazy Evaluation in Infinite-Dimensional Function Spaces with Wavelet Basis. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing (FHPNC '19), August 18, 2019, Berlin, Germany*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3331553.3342615

1 Introduction

Consider the unit interval $D^1 = [-1, 1] \subset \mathbb{R}$. This paper discusses functions on that domain, but the methods are written so as to straightforwardly generalise to unbounded multidimensional domains.

The set $\mathbb{R}^{D^1}$ of functions $D^1 \to \mathbb{R}$ is a vector space, but it is, in a sense, too big. To wit, this set contains "pathological" functions that have in every point, even in points very close to each other, a completely unrelated value. (A classical example is the function that assigns each rational number the value 1, each irrational 0.) Pathological functions like that are largely irrelevant for real-world applications; yet not only does $D^1 \to \mathbb{R}$ *contain* such functions, in a cardinality sense *most* of them are pathological. Therefore, for the purposes of an efficient computer implementation, sets of all functions on some domain are quite impractical.

It is helpful and established practice in maths and science to consider subspaces containing only "better behaved" functions. For instance,

- $C^0(D^1)$: continuous functions. These can be characterised thus: to obtain any function value $f(x)$ with at least precision ε, one can instead consider $f(\tilde{x})$ where $\tilde{x}$ needs to be merely *close enough* to x, i.e. within a distance $\delta_{x,\varepsilon}$. By making the range of inputs, i.e. the δ, sufficiently small, one can also limit the range ε of results. So, unlike for general functions, it is not necessary to know the argument *exactly*, it is enough to know it with good accuracy: then the function value will also be known accurately.
- $C^1(D^1)$: continuously differentiable functions. Whilst continuous functions merely guarantee that $f(\tilde{x})$ does

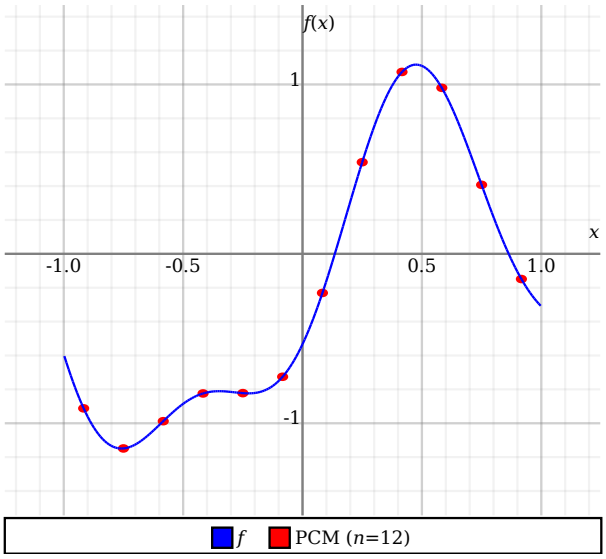

Figure 1. Example of how a function $f : D^1 \to \mathbb{R}$ is represented in PCM.

not deviate too much from $f(x)$ at nearby $\tilde{x}$, differentiability also make some tangible statement of *how* it deviates.

- $\mathcal{L}^2(D^1)$: square-integrable functions. These do not restrict what the function does at individual points, but they guarantee that the behaviour in a whole interval can be "summarised" by the integral over it. Functions that are indistinguishable by any such integration over small intervals are considered equivalent.

Remark 1.1. Continuity has a straightforward physical interpretation. One could argue that a function representing an observable value on a continuous domain *must* be continuous, at least almost everywhere, as a physical setup is never completely exact.

2 The Space of PCM-Sampled Functions

An extremely common way of representing continuous or $\mathcal{L}^2$ functions in computer programs is what we will in the following call the PCM representation[1]. The idea is to divide the domain into equal-sized segments, and store in each of these segments only one function value. That can be done with a simple array, or list.

```
newtype PCM_D¹ y = PCM_D¹ {
        getPCMSampling :: [y] }
```

where the y values correspond to equally-spaced samples of the represented function, i.e.

$$\left[f(-1+\tfrac{2}{n}\cdot i)\,\middle|\, i \leftarrow [\tfrac{1}{2} \mathinner{..} n-\tfrac{1}{2}]\right].$$

There is a body of theory, the *Shannon-Nyquist formalism*, that makes PCM a quite solid method for representing a subspace of $\mathcal{L}^2$, the subspace of *bandlimited* functions. We will not discuss this aspect here.

The space of PCM-sampled functions is a vector space, and many applications make use of this fact all the time: array-languages like Matlab and NumPy implement addition, multiplication and more on arrays, and provided that all functions are sampled with the same resolution, this corresponds perfectly to the conceptually intended extensional addition of functions, i.e.

```
h = PCM_D¹ $ zipWith (+)
    (getPCMSampling f) (getPCMSampling g)
```

$$\overset{\approx}{\Leftrightarrow} h(x) = f(x) + g(x).$$

This does not hold up anymore if f and g are sampled at different resolution. Often, this is simply prevented altogether (made a runtime error or, somewhat better, type-error with indexed-length vector types).

In some applications this is not a problem: for example

- In DSP, the input resolution is fixed by a physical ADC, and subsequent calculations (LTI filters etc.) simply leave it as-is or perhaps double the sampling. Nevertheless, it can be desirable to combine signals from different sources with different sampling rate, which then requires suitable resampling.
- Explicit integrators for hyperbolic PDEs iterate a resolution-preserving propagator-transformation involving the pointwise value of the old state and a numerical approximation (e.g. finite differences) of its spatial derivatives. This calculation will depend on the given resolution, but there is no need to precompute and store it in something like a matrix that would need to be valid for any of the possible resolutions: even if change of resolution should be required, the transformation can just be recomputed from scratch.

Many other applications, however, require coefficients to be stored before the required resolution is even known. This is both necessary for implicit methods / inverse problems – i.e. where there is no explitly programmed way to deduce the result from an input, but rather an input needs to be "guessed" that will match the desired output[2], but also for calculations which simply require a stored-coefficient form

[1] The abbreviation "PCM" stands for *pulse code modulation*. The term is used in digital signal processing for a signal in time domain that is sampled on equal time intervals with a digital value proportional to the analogue voltage it represents. We use "DSP" as a shorthand for general equal-spaced sampling. In applications of numerical differential equations, this might rather be referred to as a *finite differences* (FD) representation.

[2] In practice, implicit methods use heuristics to choose a suitable resolution. A common approach is to just match the data count of the input to that of the intended output, which is reasonable enough since inversion is clearly dependent on some notion of isomorphism – however, even an isomorphism does not necessarily preserve the highest, locally required resolution, but can "concentrate" data in given spots. This can again be taken into account with more heuristics to adaptively refine the mesh, or the

for efficiency. Doing that in a general manner which would work regardless of what resolution turns out to be necessary in the end would amount to choosing an infinite resolution.

Experience with Haskell suggests that the problem of computing infinitely many coefficients should be possible with lazy evaluation, provided only finitely many of them actually need to be evaluated in the end. However, a data structure like `PCM_D¹` is not suitable for this, since adding new coefficients to the end of the list would change the meaning of the already calculated ones (they would be squeezed to the left of the domain). What is required is some form of *progressive* resolution.

3 Multiscale Resolution

The main usefulness of spaces like $\mathcal{L}^2$ is that the infinite dimensionality can be managed easily if one only needs finite-width *integrals* over the function, as these average out small-scale fluctuations. To benefit from this, we must find a way to compute the integral without evaluating all the small-scale structure. That is the idea behind multiscale or wavelet methods. These are often derived as some orthonormal basis of $\mathcal{L}^2$, but we can also give a construction more from first principles.

To directly enable the $O(1)$ evaluation of large-scale integrals, one might consider the following representation of $D^1 \to$ y functions:

```
data PreIntg_D¹ y = PreIntg
  { offset :: y
  , lSubstructure :: PreIntg_D¹ y
  , rSubstructure :: PreIntg_D¹ y
  }
```

The idea is to decompose a function into a constant offset (proportional to the integral $\int_{D^1} \mathrm{d}x\, f(x)$) plus finer-grained fluctuations in each half of the domain, which are in turn recursively represented by the same type.

$$f_{(y_0,f_\mathrm{l},f_\mathrm{r})}(x) = y_0 + \begin{cases} f_\mathrm{l}(x_\mathrm{l}) & \text{if } x \text{ on left} \\ f_\mathrm{r}(x_\mathrm{r}) & \text{if } x \text{ on right} \end{cases}$$

The data structure `PreIntg_D¹` defined above is a binary tree which has always infinite size. This can be handled in a language with lazy evaluation like Haskell, but only if all that is ever requested from the function are integrals over finite-extend subintervals. Pointwise evaluation would recurse infinitely.

Note also that it is not really a function from D^1 to y if it *cannot* be evaluated at any individual point in D^1. In practice, for any given real-world measured function, there will be only finitely many data points available at any given moment, so at a sufficiently small scale one would eventually store *only* the offset, and assume that any smaller fluctuations are negligible.

This eventual cutoff can be implemented by wrapping the substructure fields in **`Maybe`**. Here we define instead a specific constructor for the zero function (to improve the semantics of the generic **`Nothing`** constructor). We now obtain a conventional tree with finite depth. Note that it can now also be strictly evaluated (here enforced by the exclamation marks).

```
data PreIntg_D¹ y
     = PreIntgZero
     | PreIntg !y !(PreIntg_D¹ y)
               !(PreIntg_D¹ y)
```

Pointwise function evaluation is then readily implemented recursively:

```
evalPreIntg_D¹ :: AdditiveGroup y
    => PreIntg_D¹ y -> D¹ -> y
evalPreIntg_D¹ PreIntgZero _ = 0
evalPreIntg_D¹ (PreIntg y0 l r) x
  = y0 + if x < 0
         then evalPreIntg_D¹ l (2*x+1)
         else evalPreIntg_D¹ r (2*x-1)
```

Here, `2*x+1` or `2*x-1` "zoom in" onto the left or right half subinterval, depending on where `x` lies.

One downside of **`data`** `PreIntg_D¹` is that it is redundant: the offset already fixes what the integral over the complete function should be, but there is nothing preventing the subinterval functions from contributing their own part to the integral. One solution is to use a type which represents functions having vanishing integrals. This means there can be no global offset, instead the highest-level structure is the offset *difference* between the domain halves. This changes nothing about the data structure, just about its meaning:

```
data HaarUnbiased y
    = HaarZero
    | HaarUnbiased !y !(HaarUnbiased y)
                      !(HaarUnbiased y)
```

Here, the `y` value now represents the difference in offset between the left and right halves, or, by our convention, the offset in the right half and, implicitly, the negated offset in the left half (which must be the same for the integral to vanish). One can support functions with nonzero integral by simply adding an absolute offset with a wrapper-type at the top level:

```
data Haar_D¹ y = Haar_D¹
   { global_offset :: !y
   , variation :: HaarUnbiased y }
```

The name "Haar" indicates that the basis functions of this data type (meaning those functions represented when exactly one of the fields of type $\mathbb{R}$ in the `HaarUnbiased ℝ` structure is 1, all other zero) are exactly the unnormalised Haar wavelets.

extra detail can just be smoothed out (which can be useful to keep computation effort limited in nonlinear PDE solvers, however it is mathematically not really a correct solution then).

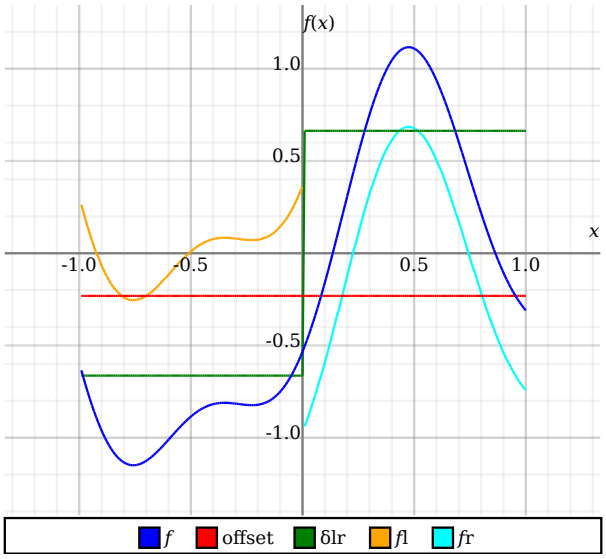

Figure 2. Example of how a function $f : D^1 \to \mathbb{R}$ is decomposed into a constant offset, plus a step-function (Haar wavelet) for the offset-difference between left and right half, plus local fluctuations in each of the halves.

4 Integration

Whether using traditional orthonormal-basis methods, or the domain-decomposition approach introduced in section 3, the numerical representations are obtained from integrating the function on an interval. If the function is given as an analytic expression, it might be possible to calculate this integral exactly, but this is typically impossible, and one resorts to numerical approximations. All such approximations amount to some weighted average of the sample points:

$$\int_{D^1} \mathrm{d}x\, f(x) \approx \sum_i w_i \cdot f(x_i)$$

with $x_i \in D^1$, $\sum_i w_i = 1$. The choice of the evaluation points x_i and weights w_i are subject to considerations of efficiency and accuracy which we will not discuss here.

Crucially for our purposes, the calculation can be split up across the domain just like the recursive `HaarUnbiased` data structure is:

$$\int_{D^1} \mathrm{d}x\, f(x) = \frac{1}{2}\int_{D^1} \mathrm{d}x\, f(\tfrac{x-1}{2}) + \frac{1}{2}\int_{D^1} \mathrm{d}x\, f(\tfrac{x+1}{2})$$

Observe that $\frac{x-1}{2}$ only evaluates on the left-, $\frac{x+1}{2}$ only on the right half of the domain.

This allows constructing the `HaarUnbiased` tree in single pass with bottom-up propagation of the partial integrals, to obtain the offset estimates at each level without redundant computation. The choice of numerical approximation only occurs at the smallest level; the simplest possibility is to only evaluate it at a single point in the middle and give that full weight (rectangular method). Thus the Haar representation can be obtained in $O(n \cdot \log n)$ from a function on the interval:

```
homsampleHaar_D¹ :: ( VectorSpace y
                    , Fractional (Scalar y) )
          => PowerOfTwo -> (D¹ -> y) -> Haar_D¹ y
homsampleHaar_D¹ (TwoToThe 0) f
  = Haar_D¹ (f 0) HaarZero
homsampleHaar_D¹ (TwoToThe i) f
  = case homsampleHaar_D¹ (TwoToThe $ i-1)
          <$> [ f . \x -> (x-1)/2
              , f . \x -> (x+1)/2 ] of
      [Haar_D¹ y0l sfl, Haar_D¹ y0r sfr]
       -> Haar_D¹ ((y0l+y0r)/2)
           $ HaarUnbiased ((y0r-y0l)/2) sfl sfr
```

This algorithm is in DSP called a *fast wavelet transform*, except it normally starts out with a PCM-sampled array instead of a function-to-be-sampled. One advantage of our approach is that it is not necessary to select one global maximum resolution (here the `PowerOfTwo` parameter); instead, a heuristic can be added that refines the resolution *locally* until the function is satisfactorily approximated.

Remark 4.1. Similar adaptive resolution strategies often dramatically improve performance in real-world applications, such as physics/engineering simulations (finite elements or finite volumes methods, where they correspond to adaptive mesh refinement). It is also the main principle behind image compression formats which use quantization on a wavelet expansion. The reason is that images or solutions to nonliner differential equations are often quite smooth in most of the domain, but include sharp edges / transients / shocks confined to a much smaller area.

5 Square-Integrable Functions and Beyond

The `Haar_D¹` structure as given in section 3, with its strict spine and thus finite depth for every tree, can not represent every D^1 function. [3] It *can* approximate arbitrarily well any $\mathcal{L}^2$ function (in the $\mathcal{L}^2$-norm sense). Namely, given a function `f`, the sequence

```
[homsampleHaar_D¹ (TwoToThe n) f | n<-[0..]]
```

converges to `f`. This is much the same for a PCM representation: improving the resolution will allow it to match an $\mathcal{L}^2$ or continuous function ever better.

However, unlike with the PCM array representation, this progressive refinement of resolution does not change the top-level structure but only adds sub-branches at ever deeper levels in the trees. It is alluring to consider allowing the trees to have *infinite* depth, which can readily be had in Haskell

[3] Perhaps most strikingly, all these functions are *dis*continuous. Like with PCM, this could be "fixed" through interpolation post-processing, but that does in neither case enable to exactly fit any function.

if only we drop the strictness annotations in the data structure. Would that then allow representing any $\mathcal{L}^2$ function *exactly*?

It does not quite work this way, at least not practically:

- `evalPreIntg_D1` would recurse infinitely. So even if the infinite tree would theoretically represent the desired function, it would not be possible to evaluate it as such in finite time.
- `homsampleHaar_D¹`, as it stands, would not be able to provide even the top-level node (i.e. the global integral), without first going into the local branches that are needed[4] to compute it.

Mathematically speaking, an infinite tree would correspond to an infinite sum over ever smaller Haar-wavelets. Infinite sums are possible to compute, provided they converge. What this means for practical computer applications is generally that the sum is *not* evaluated completely, but only the finite partial sum that suffices to achieve the required precision (this also applies to other convergent sequences, e.g. Taylor expansion of an analytic function). In other words, if the results need not be exact, then a finite cutoff of the converging-sum infinite tree is also sufficient, which is why we suggest keeping the `Haar_D¹` structure strict/finite.

On the other hand, the coefficients in an infinite tree a in principle not constrained in a way that would require convergence. Thus they can also represent things that are not $D^1 \to \mathbb{R}$ functions at all.

This is not as unreasonable as it may seem. In fact, particularly in physics, "limit functions" that are defined by a not-really-convergent sequence are quite commonly used, albeit often with lack of mathematical explanation. Best known, the "Dirac function", informally defined as

$$\delta : \mathbb{R} \to \text{“}\mathbb{R} \cup \{\infty\}\text{”}$$
$$\delta(x) = \begin{cases} \text{“}\infty\text{”} & \text{if } x = 0 \\ 0 & \text{otherwise} \end{cases}.$$

The idea is that the integral should come out as 1, and crucially for any other function g,

$$\int_{\mathbb{R}} \mathrm{d}x\, \delta(x) \cdot g(x) = g(0)$$

should hold. This allows rewriting pointwise evaluations as integrals vice versa. The integral over the product is also known as the $\mathcal{L}^2$ scalar product.

The above integral equation is quite tractable in a limit sense: consider a sequence of ever narrower and higher box-functions

$$\delta_{[n]}(x) = \begin{cases} n & \text{if } -\frac{1}{n} < x < \frac{1}{n} \\ 0 & \text{else} \end{cases}.$$

Then

$$\int_{\mathbb{R}} \mathrm{d}x\, \delta_{[n]}(x) \cdot g(x) = n \cdot \int_{-1/n}^{1/n} \mathrm{d}x\, g(x).$$

If g is continuous, then it will on the ever-smaller integration interval eventually behave like the constant g_0, and

$$n \cdot \int_{-1/n}^{1/n} \mathrm{d}x\, g_0 = g_0 = g(0).$$

This shows that the idea is sensible. But δ as a function is not: the sequence $(\delta_{[n]})_n$ does not converge, neither pointwise nor in the square-integral.

What does work about it really is just the *contraction* / scalar product with g.

By analogy, we propose that it would be useful to also consider infinite-depth trees, but not as a representation for functions on D^1 but just for contracting against such functions:

```
data CoHaarUnbiased y
    = CoHaarZero
    | CoHaarUnbiased !y (HaarUnbiased y)
                        (HaarUnbiased y)
data CoHaar_D¹ y
    = CoHaar_D¹ !y (CoHaarUnbiased y)
```

Even though this is now non-strict, the following is guaranteed to terminate[5] because `Haar_D¹` has only finite depth and will thus eventually terminate the recursion:

```
(<.>^) :: CoHaar_D¹ ℝ-> Haar_D¹ ℝ-> ℝ
CoHaar_D¹ q0 qFluct <.>^ Haar_D¹ f0 fFluct
   = q0 * f0 + qFluct~<.>^fFluct
 where CoHaarZero ~<.>^ _ = 0
       _ ~<.>^ HaarZero = 0
       CoHaarUnbiased δq ql qr
           ~<.>^ HaarUnbiased δf fl fr
         = δq * δf + ql~<.>^fl + qr~<.>^fr
```

This looks similar enough to a scalar product: corresponding entries in both of the tree structures are multiplied, and the results summed together – like one would also do with the arrays of a PCM-representation. In both cases there are some constant factors missing to make it the actual $\mathcal{L}^2$ scalar product; we will ignore that here since anyways there is no interpretation of `CoHaar_D¹ ℝ` as a type of functions on D^1 anymore.

Rather, it has an interpretation as a type of functions on `Haar_D¹ ℝ`, i.e. (seeing *those* as D^1-functions) higher order functions or *functionals*, specifically linear functionals. (Linearity is important because these form themselves a vector space, the *dual space*.)

[4] This could be overcome if we assume there is some other way of obtaining the target function's integral over a whole interval. However, *if* that is possible, e.g. because the function is given by an analytical expression, then one does not really need to resort to a numerical representation!

[5] This is in striking analogy with the Agda programming language, in which data types are strict by default but there is also *co-data* (coinductive types) allowing for infinite streams.

The Dirac distribution is a very particular example of linear functional, that can indeed be implemented as a value of `CoHaar_D¹ ℝ`. What it does – evaluation at a single point – is a special case of evaluation over a whole interval and averaging: essentially also what the elements of the $\delta_{[n]}$ sequence do, but because there is no explicit integral there is no need to scale up the height to infinity as the width is reduced.

```
boxDistribution :: (D¹, D¹) -- ^ Target interval
                -> ℝ        -- ^ Total weight
                -> Haar_D¹ DistributionSpace ℝ
boxDistribution (D¹ l, D¹ r) y
  | l > r    = boxDistribution (D¹ r, D¹ l) y
boxDistribution (D¹ (-1), D¹ 1) y
             = CoHaar_D¹ y zeroV
boxDistribution (D¹ l, D¹ r) y
  | l<0, r>0 -- intersecting both halves of domain
     = CoHaar_D¹ y $ CoHaarUnbiased (wr-wl) lstru rstru
  | l<0      -- target intersects only left half
     = CoHaar_D¹ y $ CoHaarUnbiased (-wl) lstru 0
  | otherwise -- target intersects only right half
     = CoHaar_D¹ y $ CoHaarUnbiased wr 0 rstru
 where CoHaar_D¹ wl lstru = boxDistribution
                   (D¹ $ l*2 + 1, D¹ $ min 0 r*2 + 1)
                   (y * if r>0 then l/(l-r) else 1)
       CoHaar_D¹ wr rstru = boxDistribution
                   (D¹ $ max 0 l*2 - 1, D¹ $ r*2 - 1)
                   (y * if l<0 then r/(r-l) else 1)
```

The tree generated this way will in general have infinite depth in order to select the desired interval with any delimiters, i.e. this really requires the non-strict data structure. However, the distribution will only narrow in on this selection provided that the function on which we evaluate has any structure at that level. When the function is eventually constant, only the top-level coefficient is evaluated (as that corresponds to the integral, which is what is sought here).

Furthermore, `boxDistribution` itself only builds up the infinitely fine resolution where it is actually required, i.e., at the *boundaries* of the target interval: on those subdivisions that are fully in the interval, again only to top-level coefficient of the function needs to be evaluated, whereas outside of the target the result will simply be zero. Thus, the tree has only two long branches, and `<.>^` has only a complexity of $O(\log n)$ in the resolution of the function which the distribution is contracted against. (Compare this with a PCM implementation, where a box distribution would need to contain $O(n)$ nonzero entries, all of which would need to be evaluated.)

Finally, all of this works even if the target "interval" actually has zero width:

```
dirac :: D¹ -> CoHaar_D¹ ℝ
dirac x0 = boxDistribution (x0,x0) 1
```

That implementation of the Dirac distribution does indeed evaluate functions of arbitrary resolution at one point. We have tested this with QuickCheck:

```
testProperty "Dirac eval of Haar function"
 $ \f p -> dirac p<.>^f ~= evalHaarFunction f p
```

There, the QuickCheck `Arbitrary` instance generates arbitrarily deep tree structures, and picks any point on D^1 for evaluation. The `~=` operator checks equality up to floating-point inaccuracy (in our test suite, the relative error is set to 10^{-9}).

Note that the behaviour, both of `evalHaarFunction` and `dirac` is strictly speaking undefined at the discontinuities created by the Haar representation, but the implementations shown here are in agreement. At any rate this seems safe as long as the `Haar_D¹` function is an approximation of a continuous function, because then the jumps have only very small height.

6 Tensor Products and Linear Maps

One of the most salient aspects about the dual space implementation is that it allows for a *storable* implementation of arbitrary linear mappings.

```
newtype LinearMap v w = LinearMap
          (TensorProduct (DualVector v) w)
```

The `TensorProduct` for a parameterised type like `Haar_D¹` and `CoHaar_D¹` – generally, for any functor in the category of vector spaces[6] – is simply given by instantiating the parameter with the right factor space.

```
type instance Scalar y ~ ℝ
   => TensorProduct (CoHaar_D¹ ℝ) w = CoHaar_D¹ w
```

So specifically, `LinearMap (Haar_D¹ ℝ) (Haar_D¹ ℝ)` is represented by a distribution of functions, i.e. by values of the type `CoHaar_D¹ (Haar_D¹ ℝ)`. This type is important because it would be the type of the identity linear mapping, which is required for `Haar_D¹ ℝ` to be a member of an actual category and prerequisite for generalising several linear algebra algorithm from the finite-Euclidean case to infinite-dimensional spaces like `Haar_D¹ ℝ`. And practically speaking, if `id` is defined then it is easy to sample/convert *any* linear function (defined as a Haskell function) into a tensor-based linear mapping.

`id` is another reason why `CoHaar_D¹` must be non-strict: the identity mapping needs to use an infinite tree in order to properly handle functions with arbitrarily high resolution.

Concretely,

```
id :: LinearMap (Haar_D¹ ℝ) (Haar_D¹ ℝ)
id = LinearMap $ CoHaar_D¹
```

[6] They are in fact also functors in the **Hask** category, but we recommend keeping that instance a private implementation detail because fmapping a nonlinear function is not invariant of the choice of basis, i.e. it is not safe with respect to refactoring to another representations.

```
3           (Haar_D¹ 1 zeroV)
4           (fmap (\ δ-> Haar_D¹ 0 δ) idUnbiased)
5   where idUnbiased :: TensorProduct (CoHaarUnbiased ℝ)
6                                      (HaarUnbiased ℝ)
7         idUnbiased = CoHaarUnbiased
8          (CoHaar_D¹ 1 zeroV zeroV)
9          (fmap (\l -> HaarUnbiased 0 l zeroV) idUnbiased)
10         (fmap (\r -> HaarUnbiased 0 zeroV r) idUnbiased)
```

7 Outlook

Although the Haar-wavelet-expansion type presented in this paper provides a useful starting point for numerical calculations on infinite-dimensional spaces, the fact that the represented functions are inherently discontinuous step-functions limits its usefulness for actual numerical applications. The step functions certainly are not differentiable.

Even for pointwise evaluation alone, the piecewise constant structure means that it is relatively inefficient at approximating continuous functions, namely, the discretisation error $\varepsilon = f_{\text{exact}} - f_{\text{Haar}}$ reduces proportionally to the step size δ, i.e. anti-proportionally to the required tree size. The adaptiveness of resolution can somewhat mitigate this (regions with small gradients have low ε to begin with, so it is sufficient to focus on those with stronger gradient or even discontinuity), however this is limited unless the function really is constant on most of the domain.

By contrast, piecewise linear functions can scale $\varepsilon \propto \delta^2$, cubic ones $\varepsilon \propto \delta^4$ and so on. Thus it would be desirable to combine such a higher-order local model with the tree-based multiscale structure. Comparison with wavelet theory suggests that this may not be as straightforward as it is to employ polynomial interpolation for a PCM sampling or for a finite elements model. Namely, the simplest piecewise-linear orthogonal wavelet is not a hat function by the rather complicated Strömberg wavelet.

However, our approach has the advantage that it does not actually rely on $\mathcal{L}^2$-orthogonality, but uses domain decomposition and direct value readoff for its sampling process. Therefore it is plausible that the "mother wavelet" can be kept much more basic. In particular, a very simple way to construct a continuous function from a Haar-based one is through integration. Because the complete integral can always be evaluated in $O(1)$, this would also still allow efficient random-access pointwise evaluation of the continuous function, unlike the integral of a PCM-sampled function (which would be $O(n)$ for single-point evaluation).

Another important generalisation will be multi-dimensional domains. In fact, `Haar_D¹` already supports those in a sense because a function vector space on a product domain is isomorphic to the tensor product of the function spaces on the factor domains, i.e. `Haar_D¹ (Haar_D¹ ℝ)` represents functions on $D^1 \times D^1$. However, this would largely circumvent the locality properties of the Haar expansions (since nearby points in y-direction would lie in completely different trees of the decomposition in x-direction). An efficient implementation would probably need to intersperse the direction-splittings, to give a kind of kd-tree structure in space an Morton Z order of the leaves.

In summary: we have presented a data structure that can express function types in a way that better represents the mathematical (functional-analysis) notion of such an infinite-dimensional space than the mainstream numerical expansions do. It combines features of techniques from established numerical schemes (wavelets from multiscale analysis, tree backbones from Barnes-Hut style simulations, parametricity/tensor-product from numerical linear algebra), and we expect that it can be extended to be of similar practical use while being more mathematically general and transparent.