Theory and implementation of exact real number computation

Peter Berry BSc (Swansea)

A thesis submitted to Swansea University in candidature for the degree of Master of Research

Swansea University
Prifysgol Abertawe

Department of Computer Science
Swansea University

March 2010

Declaration

This work has not been previously accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed ..... (candidate)

Date .....

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed ..... (candidate)

Date .....

Statement 2

I hereby give my consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed ..... (candidate)

Date .....

Abstract

This thesis investigates the theoretical basis of computation with exact real numbers: the basics of real analysis, including metric and topological spaces and domain theory; type-2 Turing machines and the resulting notion of computability on infinite data; and methods of representing real numbers, in particular decimal expansions and signed digit streams. The topic is additionally discussed in the context of proof theory, in particular program extraction via realisability. Finally, an application of the theory is demonstrated by Haskell code (improved from the original by Ulrich Berger) extracted by hand, justified by realisability, from definitions on the proof level.

Contents

Chapter 1

Introduction

Throughout the history of computation, many kinds of mathematical objects have been manipulated with algorithms, but one in particular has been persistently elusive: real numbers. The standard theory of computation is based on halting Turing machines, which provides admirably for finite objects such as integers, rational numbers and words in finitary languages; but it fails where objects potentially contain infinite amounts of information, in particular real numbers, despite the fact that real numbers were the original motivating example for Turing machines [Tur36]. Those wanting to compute with "continuous" data have had to make do with fractions: rationals, or finite-precision floating point numbers.

To define a data type for true real numbers, we need to define exactly what quality of real numbers it is that halting Turing machines can't handle, and decide what kind of relaxation of the "must halt" restriction is necessary. In the process it becomes clear that we need a non-traditional way of writing them down; infinite decimal expansions are insufficient. Specifically, the representation of a Euclid-continuous function by a function on infinite decimal expansions is not Cantor-continuous. There is a characterisation of "admissible" representations in which a representation of a Euclid-continuous function is Cantor-continuous; this includes streams of signed binary digits.

Ulrich Berger has developed a first order definition of uniformly continuous functions of signed digit streams [Ber09a], and translated it into Haskell code, following the realisability interpretation of first order logic [BS10]. The code additionally makes use of memoization via an isomorphism with terminal coalgebras, described by Altenkirch [Alt01] and Hinze [Hin00b]. In this thesis, I investigate different representations of real numbers and real functions by infinite data structures, and use advanced extensions to Haskell to improve this code. In the process, it becomes clear that there is a close relationship between the realisability method used in the code and a more direct method of memoizing (with the Altenkirch/Hinze method) representations of real functions by monotone functions on finite prefixes.

Chapter 2 describes the basic theory of real numbers, specifically Euclidean and Cantor metrics and their associated topologies. This is followed by a discussion of Weihrauch's Type-2 Turing machines, which provide a notion of computability on infinite data, including real numbers. I also give a tentative definition, in terms of Type-2 TMs, of a certain class of computable corecursive

functions, the primitive corecursive functions, by analogy with primitive recursive functions. Finally, a brief description is given of the signed digit stream representation of real numbers, which is the representation used in our implementation.

Chapter 3 describes the extraction of programs about real numbers from proofs of their properties in first order logic. This involves the proofs-as-programs paradigm or Curry-Howard correspondence, in which dependently typed functional programs correspond to first order proofs, and making use of the realisability interpretation of logic to remove the non-computational aspects of such proofs, in order to produce more efficient, untyped or simply typed programs [Ber09b]. Our implementation consists of programs extracted by this process by hand.

Chapter 4 is concerned with memoization by means of generalised tries, by which functions on algebraic domains can be represented as non-wellfounded tree data structures, improving their time efficiency. Real numbers themselves can be described as functions on algebraic data, namely as streams (functions whose domain is the natural numbers) of signed digits, so memoization of these functions yields a representation of real numbers as part of a more general framework.

In Chapter 5 I bring together the various theoretical threads into a concrete implementation in Haskell. Specific extensions and libraries are used to improve Berger's original code, making the connection between the proofs and manually extracted programs more clear.

Finally, Chapter 6 summarises the research contribution of this thesis, and suggests some possible further avenues of research beyond it.

1.1 Related work

Previous approaches to exact real number computation have involved representing a computation on real numbers by constructing an abstract expression operating on abstract objects that represent real numbers or (in the case of functions) real-valued variables:

- RealLib [Lam07], by Branimir Lambov, yields real number objects from which the programmer can extract finite approximations in any of several representations, including IEEE 754 floating point numbers, decimal representations, and strict comparisons. The floating point representation allows for very fast hardware computations for applications where their precision is sufficient.

- IC-Reals [EH02b], by Addas Ebalat and others at Imperial College, uses a representation based on linear fractional transformations on signed digit streams.

- Norbert Müller's iRRAM [Mül01] is based on simulation of Turing machines computing real numbers represented as converging series of open intervals with rational endpoints. The model is in turn based on Brattka and Hertling's concept of a "Real RAM" [BH96].

All of these libraries are written in C++, although IC-Reals also has a Haskell implementation.

This thesis describes a way of computing directly with representations of real numbers as streams of signed binary digits. There is no abstract symbolic representation; rather, algorithms are defined directly using general language constructs. However, when implemented in a lazy functional language (such as our Haskell implementation), such computations are expressions (closures, or thunks — suspended computations) built up in the object language and evaluated at need, which bears some resemblance to the symbolic approach.

Russell O'Connor has created a library for reasoning and computing with complete metric spaces in Coq [O'C08], using which he has implemented constructive real numbers and elementary real functions together with proofs of their correctness. This metric-space-based approach is similar to the more recent directions of Ulrich Berger's project, which also involves metric spaces on digit systems.

Chapter 2

Theoretical background in computable analysis

This chapter discusses the theoretical framework of real number computation: the fundamentals of real analysis and its intersection with computability theory. This is essential for understanding what form programs manipulating real numbers must take.

2.1 Real numbers

Real numbers are, after the natural numbers, probably the most fundamental of mathematical objects. While the natural numbers are an abstraction of finite, discrete quantities (finite cardinals) and finite positions in a well-founded ordering (finite ordinals), real numbers are an abstraction of finite continuous quantities and finite positions along a non-wellfounded continuum. It's not difficult to understand how natural numbers, integers and rationals can be modelled with a computer and operations performed on them, and there are well-known models of computation — particularly Turing machines — that can be used to analyse such properties of algorithms as totality (termination) and complexity (of both time and space usage). This is because all of these sets are *countable*: there exists from each of these sets $S \in \{\mathbb{N}, \mathbb{Z}, \mathbb{Q}\}$ a surjective function

$$f : \mathbb{N} \twoheadrightarrow S$$

also called an *enumeration*. In other words, the cardinality of the sets S is at most that of $\mathbb{N}$, namely $\aleph_0$, the smallest transfinite cardinal number.

Generalising this concept of enumeration to partial functions $f :\subseteq \Sigma^* \to S$ from *words over a finite alphabet* Σ further gives rise to the concept of a *language* or *notation* [Wei00]. Intuitively, this means we can write down textual descriptions of countable sets in a uniform way, since if Σ is a finite set, the set Σ^* of words over Σ is also countable. (Of course, this might not be a *computable* enumeration — as in the case of all non-terminating Turing machines — in which case any purported "Turing machine" enumerating it would have to be infinitely complex (have infinitely many states, thus need to be described using an infinite amount of information).)

Fundamental as they are, real numbers unfortunately are not countable: there is no surjective function $f : \mathbb{N} \twoheadrightarrow \mathbb{R}$. This can easily be shown by Georg Cantor's famous diagonalisation method: given any such enumeration f, we can construct a real number x such that there is no $n \in \mathbb{N}$ with $f(n) = x$. This x can be constructed using the following (nonterminating but productive) algorithm on infinite decimal expansions:

1. Output $\mathtt{0}$[1] followed by a decimal point.
2. For each $n \in \mathbb{N}$, beginning with 0 and proceeding with $n + 1$:
   2.1. Compute d_n, the nth digit of $f(n)$ after the decimal point.
   2.2. Choose a digit $d'_n \in \{\mathtt{0}, \ldots, \mathtt{9}\}$ with $d'_n \neq d_n$, and output it.

This algorithm obviously constructs a decimal representation of a real number x such that for all n, $f(n) \neq x$. (Note that we need to avoid, say, outputting $0.1999\ldots$ if computing f produces the representation $0.2000\ldots$. This property of multiple representations exists for almost all schemes for representing the real numbers. For the decimal expansion there are only ever two possibilities, which are always adjacent (interpreting 0 and 9 as adjacent), so we can exclude that digit and the adjacent ones, only choosing from among the other 8 digits.)

Although there is no enumeration of the real numbers, there are surjective partial functions $f :\subseteq \Sigma^\omega \twoheadrightarrow \mathbb{R}$ for various Σ. Whereas Σ^* means "words over Σ", where by "word" we mean a finite list (describable as a function $w : n \to S$ where n, the length of w, is a finite ordinal), Σ^ω means "streams over Σ", where by "stream" we mean an infinite list defined by a first element and a tail (describable as a function $s : \omega \to S$, where ω is the smallest transfinite ordinal). In particular there is one for $\Sigma = \{\mathtt{0}, \ldots, \mathtt{9}, \mathtt{.}\}$, namely the standard infinite decimal place-value expansions: infinite strings consisting of digits $d \in \{0, \ldots, 9\}$ and one decimal point. This notation however has some problems, making it unsuitable for use in computation. In this project we instead use the *signed digit stream* representation, which solves some of these problems. This issue is discussed fully in section 2.4.

2.2 Computability on uncountable sets

Computability on countable sets is conventionally defined in terms of languages and Turing machines (TMs): a function $f : S \to T$ (where S and T are countable) is computable if and only if there is a TM that, starting with such a representation of $x \in S$ (i.e. a word over some finite alphabet) on its tape, halts with such a representation of $f(x)$ on its tape. This is not a theorem, but rather a (philosophical) definition of computability, called the Church-Turing thesis (an equivalent formulation exists replacing TMs with Church's untyped lambda calculus). The set of TMs is countable, so under this definition the set of computable functions is as well. Since a TM computing a computable function must halt, its output can only constitute finite words in any language, so this definition of computability is not appropriate for computation on uncountable sets — which (over a finite alphabet) necessarily contain some elements with

[1] I use typewriter face for symbols in an alphabet, to distinguish them from actual numbers.

infinite representations, because there are only countably many finite strings over a finite alphabet.

Of course, a computation in the real world cannot make use of *all* the information available in a (representation of a) real number, since that would take infinite time, while we can only make use of finite time. We might, however, want to run a non-terminating computation as long as we need to to get *sufficient* information in the output. If we want a machine that performs such an infinite computation, we want to be sure that if we stop it at any point, any output it has already produced wouldn't be overwritten if we ran it for a bit longer. In other words, any finite portion of the output depends only on a finite portion of the input (since reading all the input would also take infinite time).

To make this concrete, consider the *Cantor space* $\mathbb{C}$ of infinite sequences of binary digits (i.e. $\mathbb{C} = \mathbb{N} \to \{0, 1\}$, also called 2^ω or $2^{\mathbb{N}}$). If we use the informal definition of computability above, this translates into the following definition on the Cantor space:

Lemma 2.2.1 (equality up to k)**.** *For $k \in \mathbb{N}$ and $p, q \in \mathbb{C}$ define*

$$p =_k q \iff \forall i \leqslant k.\ p(i) = q(i)$$

Then $(=_i)_{i \in \mathbb{N}}$ is a series of successively finer equivalence relations on $\mathbb{C}$, that is, each $=_i$ is an equivalence relation and

$$i \leqslant j \implies (p =_j q \implies p =_i q).$$

Definition 2.2.2 (continuity of functions on Cantor space)**.** A function $f : \mathbb{C} \to \mathbb{C}$ is continuous if and only if the following holds: For all $k \in \mathbb{N}$ and sequences $p \in C$, there exists an $l \in \mathbb{N}$ such that for all sequences $q \in \mathbb{C}$, if $p =_l q$ then $f(p) =_k f(q)$.

Intuitively, this means that the first l digits of the sequence p contain enough information to compute k digits of $f(p)$; the fact that the equivalence relations $=_i$ are successively finer means that further specifying the input beyond l does not affect the output up to k. A stronger version is as follows:

Definition 2.2.3 (uniform continuity of functions on Cantor space)**.** A function $f : \mathbb{C} \to \mathbb{C}$ is uniformly continuous if and only if the following holds: For all $k \in \mathbb{N}$, there exists an $l \in \mathbb{N}$ such that for all sequences $p, q \in \mathbb{C}$, if $p =_l q$ then $f(p) =_k f(q)$. The Skolem function giving this l for each k is called the *modulus of uniform continuity.*

Lemma 2.2.4. *If f is uniformly continuous, then it is continuous.*

Proof. This is a special case of the tautology

$$\forall p.(\forall a \exists x \forall b.p(a, b, x) \implies \forall a \forall b \exists x.p(a, b, x)).$$ □

(This theorem is more general than Cantor space: it is true for any topological space (more about which later).)

Lemma 2.2.5. *If $f : \mathbb{C} \to \mathbb{C}$ is continuous, then it is uniformly continuous.*

Proof. This is a consequence of König's lemma. The proof is folklore. □

2.2.1 Metric spaces

The above definition of continuity essentially asserts that given a continuous function $f : \mathbb{C} \to \mathbb{C}$, if two sequences p, q are "close enough", then $f(p)$ and $f(q)$ are similarly "close". This idea of "closeness" can be quantified using a *metric*.

Definition 2.2.6 (metric on Cantor space)**.** Define a function $d_{\mathbb{C}} : \mathbb{C} \times \mathbb{C} \to \mathbb{R}$ by

$$\begin{aligned} d_{\mathbb{C}}(p,p) &= 0 \\ d_{\mathbb{C}}(p,q) &= 2^{-n} \end{aligned}$$

where n is the smallest i for which $p(i) \neq q(i)$.

We can use this to define continuity as follows:

Definition 2.2.7 (metric continuity on Cantor space)**.** Define

$$B_\epsilon(p) = \{q \mid d_{\mathbb{C}}(p,q) < \epsilon\},$$

the *open ball of radius ϵ around p*. A function $f : \mathbb{C} \to \mathbb{C}$ is continuous if for any point x and positive distance ϵ, there is some positive distance δ such that the image under f of any open ball of size ϵ around x is contained within the the open ball of size δ around $f(x)$:

$$\forall x \forall \epsilon > 0. \exists \delta > 0. f[B_\delta(x)] \subseteq B_\epsilon(f(x))$$

Equivalently:

$$\forall x \forall \epsilon > 0. \exists \delta > 0. B_\delta(x) \subseteq f^{-1}[B_\epsilon(f(x))] \tag{2.1}$$

In other words, if some y is within δ of x, then $f(y)$ is within ϵ of $f(x)$:

$$\forall x \forall \epsilon > 0. \exists \delta > 0. \forall y. d(x,y) \leq \delta \to d(f(x), f(y)) \leq \epsilon$$

It is well known that these two typical definitions of continuity coincide:

Lemma 2.2.8. *Definitions 2.2.2 and 2.2.7 are equivalent.*

The Cantor space is an example of the general construction of *metric space*, a set equipped with a notion of *distance* between elements.

Definition 2.2.9 (metric spaces)**.** A *metric space* is a pair (A, d) where A is a set and $d : A \times A \to \mathbb{R}$, the *metric*, is a function satisfying

$$\begin{array}{rcll} d(x,y) & \geqslant & 0 & \text{(non-negativity)} \\ d(x,y) = 0 & \iff & x = y & \text{(identity of indiscernibles)} \\ d(x,y) & = & d(y,x) & \text{(commutativity)} \\ d(x,y) + d(y,z) & \geqslant & d(x,z) & \text{(triangle inequality)} \end{array}$$

Intuitively, the metric laws express that two different points can't be in the same place, each point is in only one place, direction is irrelevant to distance, and detours are not shortcuts.

Lemma 2.2.10. *The Cantor metric really is a metric.*

Proof. By definition $d(p,p) = 0$, and $d(p,q) = 2^{-n}$ (in case $p \neq q$) is always positive for any $n \in \mathbb{N}$, proving non-negativity and identity of indiscernibles. The definition is symmetric with respect to the arguments, so d is also commutative. Without loss of generality assume $d(p,q) \leqslant d(q,r)$. This means precisely $p =_i q$ and $q =_j r$ for some $j \leqslant i$. By Lemma 2.2.1 also $p =_k r$ for some $k \geqslant j$ (in fact $k = i$ because d is an ultrametric); in other words, $d(p,r) \leqslant \max(d(p,q), d(q,r))$ and *a forteriori* the triangle inequality holds. □

In general metric spaces, continuity is not necessarily equivalent to uniform continuity, but uniformly continuous functions are always continuous (by the same logical tautology). The important question is whether the space is topologically compact, which is a sort of finiteness quality.

2.2.2 Computability and continuity

Denotational semantics

When studying the denotational semantics of functional programming languages, we assign a *Scott domain* [AJ94] to each type in the language. In this usage a domain is a set of partial values of that type, partially ordered by definedness, or alternatively by what information they contain. It is a sort of meet-semilattice that has some joins: a meet of a set of elements is the element containing all the information that all of them share (possibly none), and *no less*; while a join of a set of elements is the element that contains all the information that any of them have, but *no more*, which obviously only makes sense if their information is mutually consistent. Thus 'meet' is analogous to the binary connective 'and' (for $x, y \in X$, information in $\bigsqcap X$ is in x *and* in y), 'join' is analogous to 'or' (information in $\bigsqcup^{\uparrow} X$ may be in x *or* in y); and the 'bottom' element $\bot = \bigsqcup \emptyset$ (the element containing all the information contained in any of the elements of the empty set, i.e. none) is analogous to absurdity (if X is inconsistent, information in $\bigsqcap X$ is nonexistent). The definedness ordering (more usually called the specialisation ordering) is $x \sqsubseteq y \iff x = x \sqcup y$, expressing the idea that all the information contained in x is also contained in y.

As a simple example, the semantics $[\![\texttt{()}]\!]$ of a singleton type `()` (a type with only one value) is the *Sierpinski space*, a domain 1 with (despite its name) two elements: $\bot$, the undefined element; and $*$, an arbitrary value distinct from $\bot$ with $\bot \sqsubset *$. There are concepts of product and sum on domains, as well as least and greatest fixed point operations for strictly positive functions of domains, allowing domains representing algebraic data types (see 4.1.1 below) to be defined using equations. For example the semantics of a type of natural numbers is defined as the least solution to the domain equation

$$\mathbb{N} = 1 + \mathbb{N}$$

Using this kind of semantics, a function $[\![f]\!] : [\![\sigma]\!] \to [\![\tau]\!]$ (in other words, a function between possibly incompletely evaluated values of the types σ and τ) can be correctly defined in the language (as f) only if it is *Scott-continuous*. Specifically, $[\![f]\!]$ is Scott-continuous if it preserves directed suprema; in other words, $[\![f]\!](\bigsqcup^{\uparrow} X) = \bigsqcup^{\uparrow}([\![f]\!][X])$ whenever X is directed ($\bigsqcup^{\uparrow}$ means "directed join", that is, join with the precondition that its parameter be directed). All

continuous functions are also *monotone* (more correctly monotonically increasing), meaning $f(x) \sqsubseteq f(y)$ whenever $x \sqsubseteq y$, that is, f may be able to make use of extra information, but more information does not make it 'change its mind'. Continuing the simple example, consider the functions on $[\![()]\!]$. Since this set has two elements, there are, set-theoretically, four functions on it:

x	$f_1(x)$	$f_2(x)$	$f_3(x)$	$f_4(x)$
$\bot$	$\bot$	$\bot$	$*$	$*$
$*$	$\bot$	$*$	$\bot$	$*$

Of these, f_3 is not continuous, in fact it is not even monotone, because it reverses the ordering, adding extra information to $\bot$ but not to $*$. $\bot$ represents, among other things, *nonterminating* computations that yield no information; for f_3 to be computable we would need to solve the halting problem, which is well known to be undecidable. Although f_4 adds information, it does not do so by 'magically' detecting that the value is $\bot$ (say, by solving the halting problem) but rather behaves the same on all inputs, and this does not prevent it from being continuous. It is just a constant function, which is computable, and definable in non-strict languages like Haskell (but not strict languages like ML). f_1 is another constant function. Finally, f_2 is the identity function, doing nothing with its input and passing it out unaltered and uninspected.[2]

Topological spaces

Continuity is the province of a branch of mathematics called *topology*. Topology is usually considered in connection with geometry, where it provides a systematic study of boundaries, that is, limits. The original motivation is to understand what it means for, say, two two-dimensional shapes to be 'the same', modulo stretching of boundaries without tearing them. This 'sameness modulo stretching' is represented mathematically by bijective continuous functions (*homeomorphisms*) between the two spaces. The term 'continuous' is defined first on functions $f : \mathbb{R} \to \mathbb{R}$, where it means that f's graph has no 'jumps': as x and y get arbitrarily close together, so $f(x)$ and $f(y)$ also get arbitrarily close. Extending the concept to $\mathbb{R}^2$, a continuous function is one that 'tears or merges no holes'. We can also mix spaces: a continuous function $f : \mathbb{R}^2 \to \mathbb{R}$ can be described as a two-dimensional surface in 3D space that does not overlap itself and 'has no holes'. In any case, the proper intuition about continuous functions is that a small change in the input gives a correspondingly small change in the output. If a shape X has a certain number of 'holes' in its boundary, then given a shape Y, if a continuous function $f : X \to Y$ exists, then Y has the same number of holes in its boundary; if Y had more, there would be a point where two points that are very close together on X's boundary are far apart, on opposite sides of the hole, on Y's boundary (*vice versa* if X has more), so no such f could exist. In short, a continuous function is one which *preserves boundaries*.

This sounds similar to the definition of 'Scott continuity' on domains as defined above, of a continuous function being one which preserves 'boundaries' in the sense of directed suprema. In fact, if we define a *Scott topology* and apply it to all domains, the two definitions of 'continuous' are the same. A continuous

[2] The example given here is adapted from [Pip08].

function is thus one in which any finite precision of output can be obtained with some finite precision of input; that is, we can observe finite properties with a finite amount of effort.

Definition 2.2.11 (topological spaces)**.** A topological space is a set S together with a set of subsets $\tau_S \subseteq \mathcal{P}(S)$ (the *topology*, whose elements are called *open sets*) satisfying:

- $\emptyset \in \tau_S$ and $S \in \tau_S$;
- if $X, Y \in \tau_S$, then also $X \cap Y \in \tau_S$;
- if $T \subseteq \tau_S$ is a set of open sets, $\bigcup T$ is an open set.

By induction, a topology is closed under finite intersection. $X \subseteq \tau_S$ is a *base* for τ_S if $\tau_S = \{\bigcup O \mid O \subseteq X\}$ and a *subbase* for τ_S if $\{\bigcap O \mid O \subseteq X, O \text{ finite}\}$ is a base for τ_S. If X is a subbase for τ_S then τ_S is the *topology generated by* X.

Samson Abramsky [Abr87] (quoted by Stephen Vickers [Vic89]) remarked that, in light of domain theory, topology is the logic of finitely observable properties. (This was inspired by Mike Smyth's slogan "open sets are semidecidable properties".) Interpreting an open set as a finitely observable property — that is, a property that, whenever it is true, we can observe that it's true in finite time, using a finite amount of information — we see that the axioms for a topological space reflect this; the idea is connected to the discipline of *geometric logic*.

- The open set $\emptyset$ represents a property that is never true. By 'X is finitely observable' we really mean 'if X is ever observable, it is finitely observable', which is vacuously true in this case.
- The open set S represents a property that is always true. It is finitely observable because if we know it is always true, we need not bother spending any resources to test for it; and no resources at all is of course finite.
- To observe $X \cap Y$, we need to observe both X and Y. If observing X takes resources t_X, and similarly for Y and t_Y, then observing $X \cap Y$ takes resources $t_X + t_Y$ at worst. If t_X and t_Y are both finite, so is $t_X + t_Y$. This extends to any finite number of finite observations, but not infinitely many ones.
- To observe $\bigcup X$ where X is a set of finitely observable properties, we only need to observe one $x \in X$, no matter how many there are. In principle the resources we need to observe $\bigcup X$ is the minimum of the resources needed to observe any of the x. (In practice there is some overhead in organising X so it can be exhaustively searched.)

As an example, consider the standard topology on the Cantor space:

Definition 2.2.12 (Cantor topology on Cantor space)**.** Let

$$P = \{p :\subseteq \mathbb{N} \to \{0,1\} \mid \exists k \in \mathbb{N}.(\forall i < k.p(i)\downarrow) \wedge (\forall i \geqslant k.\neg(p(i)\downarrow)\}.$$

be the set of all prefixes of binary sequences ($p(i)\downarrow$ means "p is defined at i"). For each prefix $p \in P$ define the set of its infinite extensions

$$\sigma_p := \{\sigma \in \mathbb{C} \mid \forall i \in \mathbb{N}.\ p(i)\downarrow \implies p(i) = \sigma(i)\}$$

Then $\beta := \{\sigma_p \mid p \in P\}$ (the set of basic opens) is a base for the *Cantor topology.*

As is well known, the notion of continuity induced by this topology is the same as those defined before. For another example, the real numbers $\mathbb{R}$:

Definition 2.2.13 (standard topology on $\mathbb{R}$)**.** A subset $X \subseteq \mathbb{R}$ is an *open interval* iff

$$X = \{z \in \mathbb{R} \mid x < z < y\}$$

for some $x, y \in \mathbb{R}$ with $x < y$. This is usually abbreviated (x, y). The set of all open intervals is a base for the *Euclidean topology* on $\mathbb{R}$.

Note that the term 'open interval' is an arithmetic notion specific to $\mathbb{R}$ and is defined without reference to topology; whereas 'open set' is a topological notion, so called for the historical reason that $\mathbb{R}$ was among the first topological spaces defined and its open sets happen to be 'open' in the arithmetic sense.

As it happens, both $\mathbb{C}$ and $\mathbb{R}$ are topological spaces in a very natural way: both have a notion of distance (they are metric spaces), and any metric space has a canonical topology.

Lemma 2.2.14 (metric on $\mathbb{R}$)**.** *Define*

$$d_{\mathbb{R}}(x, y) = |x - y|$$

This function is a metric.

Definition 2.2.15 (topology generated by a metric)**.** Let (X, d) be a metric space. Define

$$\beta = \{B_\epsilon(x) \mid x \in X, \mathbb{R} \ni \epsilon > 0\}$$

where $B_\epsilon(x)$ is the open ball of radius ϵ (with respect to the metric d) around x. Then β is a subbase for a topology τ_d, the topology generated by the metric d.

Lemma 2.2.16. *The standard topologies on $\mathbb{R}$ (Euclidean) and $\mathbb{C}$ (Cantor) are the ones generated by their respective metrics.*

There are various standard ways to construct new topologies, for instance:

Definition 2.2.17 (subspace topology)**.** Let (X, τ) be a topological space and $X' \subseteq X$. The *subspace topology* on X' is

$$\tau_{X'} = \{o \cap X' \mid o \in X\}.$$

That is, the open sets in the subspace topology are all the open sets of the parent topology restricted to X'.

Note that some of the opens in the subspace topology may not be open in the parent set. For example:

Definition 2.2.18 (standard topology on the unit interval)**.** $\mathbb{I} := [-1, 1] = \{x \mid -1 \leqslant x \leqslant 1\}$ has the subspace topology as a subset of $\mathbb{R}$.

Lemma 2.2.19. $[-1, 0) = \{-1\} \cup (-1, 0)$ *is open in $\mathbb{I}$ but not in $\mathbb{R}$.*

Proof. $[-1, 0)$ is $(-2, 0) \cap \mathbb{I}$ and is therefore open in $\mathbb{I}$, but is not a union of open intervals. □

The theory of topology is set up specifically to make the following definition:

Definition 2.2.20 (continuity)**.** Let S and T be topological spaces and $f : S \to T$ a function between points of the space. The *inverse image* under f of a subset $t \subseteq T$ is defined by

$$f^{-1}[t] := \{x \in S \mid f(x) \in t\}.$$

Then f is *continuous* iff the inverse image of every T-open set is an S-open set: for all $t \in \tau_T$, $f^{-1}[t] \in \tau_S$.

It is immediately evident that this definition is the same as the definition for metric spaces (2.1).

For a function to be computable (as defined in the next section), it must at least be continuous:

Proposition 2.2.21. *If $f : \mathbb{C} \to \mathbb{C}$ is computable, then it is Cantor-continuous.*

Similar results apply to many sets, for example:

Proposition 2.2.22. *If $f : \mathbb{I} \to \mathbb{I}$ is computable, then it is Euclid-continuous.*

This second result holds for similar reasons to the first. Now the Euclidean topology on $\mathbb{I}$ does not look much like a Cantor topology, so one would think the arguments are inapplicable. This is because there is no specification here that the real numbers are a sequence of some sort for which a Cantor topology might make sense. Actually, real numbers as mathematicians think of them are not usually concretely pinned down like this, but rather they just posit the existence of some set of abstract "real numbers" and assume it has the properties that they want. The question of whether a set with these properties actually exists is a foundational question; it must be answered in the affirmative for results about real numbers to make sense, but once that is done, what the set concretely consists of is not interesting for most applications. As computer scientists we are however deeply interested in foundational questions like this, since to implement a mathematical idea it has to be built completely from the ground up.

2.3 Type-2 computability

As noted previously, the usual definition of computability, using terminating Turing machines operating on finite strings over a finite alphabet, is inappropriate for uncountable sets, where necessarily some data must be represented with infinite strings. Weihrauch [Wei00] similarly extends the concept of computability to uncountable sets in a fairly simple manner: where a function ranges over an *uncountable* set (represented by infinite strings), he relaxes the requirement that a TM computing a computable function must terminate, but it must still produce output. At the same time, it still makes no sense for a computable function on *countable* sets to not terminate. This dichotomy is resolved by imposing a type on the input and output of a TM, and computability is defined differently at different types according to whether it ought to halt or be infinitely productive. The requirement for a reliable output also means that the definition of TM must be modified to stop it from overwriting output that has already

been produced. The result is *type-2 Turing machines* and the associated *type-2 theory of effectivity*. (The property of a machine that neither halts nor loops without output was called *circle-free* by Turing himself in his original paper on *a-machines* [Tur36]).

Weihrauch's Turing machines are defined as follows.

Definition 2.3.1 (Turing machines)**.** A Turing machine consists of

- a finite input/output alphabet Σ with a special symbol $\sqcup$ ("blank"), and a work alphabet $\Gamma \supseteq \Sigma \cup \{\sqcup\}$;
- k input tapes, numbered 1 to k, for some k;
- $N - k$ work tapes, for some $N \geqslant k$, numbered $k + 1$ to N, infinite in both directions;
- an output tape, numbered 0;
- a flowchart[3] operating on the tapes, with actions
  - HALT;
  - i:LEFT and i:RIGHT, which move the head on tape i respectively left or right;
  - for each $a \in \Gamma$, i:WRITE(a), which writes a on tape i;
  - for each $a \in \Gamma$, i:IF(a), which branches depending on whether the head of tape i is over the symbol a.

  The following restrictions apply to the flowchart:
  - i:LEFT is only allowed if i is a work tape.
  - i:WRITE(a) is not allowed if i is an input tape.
  - 0:WRITE(a) is only allowed for $a \in \Sigma$, and must be followed by 0:RIGHT.

The restrictions guarantee that the input tapes are one-way, read-only tapes, and that output once written cannot be erased or overwritten; in other words, input is immutable and output is final. Since the definition is meant to be equivalent to conventional Turing machines, it is implied that the flowchart must be finite (corresponding to finitely many states and hence a finite transition function). (The restrictions on the input tape do not restrict the set of computable functions since input that has been read can just be copied to a work tape, but they make the machines easier to reason about.)

Definition 2.3.2 (type-2 machines)**.** A Type-2 Turing machine is a Turing machine with input/output alphabet Σ and k input tapes together with a type specification $(Y_1, \ldots, Y_k, Y_0)$ $(Y_i \in \{\Sigma^*, \Sigma^\omega\})$ specifying whether each input tape contains finite or infinite strings and whether the output is finite or infinite.

[3] The definition in [Wei00] does not explain more precisely than this what a "flowchart" is. We assume what is meant is a state transition diagram with each edge labelled by a sequence of these actions; and nodes labelled i:IF(a) which have two outward transitions, corresponding to "yes" and "no".

Definition 2.3.3 (type-2 computability)**.** The initial configuration for a Turing machine on input $(y_1, \ldots, y_k)$ is as follows: each y_i is placed immediately to the right of the head on input tape i, and all other cells are blank. Define

1. if $Y_0 = \Sigma^*$: $f_M(y_1, \ldots, y_k) = y_0 \in \Sigma^*$ if and only if, on input $(y_1, \ldots, y_k)$, M halts with y_0 on its output tape.
2. if $Y_0 = \Sigma^\omega$: $f_M(y_1, \ldots, y_k) = y_0 \in \Sigma^\omega$ if and only if, on input $(y_1, \ldots, y_k)$, M computes forever and writes y_0 on its output tape.

Then a partial function $f :\subseteq Y_1 \times \cdots \times Y_k \to Y_0$ is called *computable* if and only if for some type-2 machine M, $f(x) = f_M(x)$ for all $x \in \mathrm{dom}(f)$.

This definition excludes two cases in particular:

1. If $Y_0 = \Sigma^\omega$ and M halts on input x, $f_M(x)$ is undefined (because its output $y_0 \notin \Sigma^\omega$).
2. If, on input x, M writes some output (maybe none) and then computes forever without writing any output, $f_M(x)$ is undefined (because it does not halt, and its output $y_0 \notin \Sigma^\omega$).

Theorem 2.3.4. *If $f :\subseteq \Sigma^{a_1} \times \cdots \times \Sigma^{a_k} \to \Sigma^{a_0}$ ($a_i \in \{*, \omega\}$) is computable (according to Definition 2.3.3), then it is continuous (with respect to the product topology for the input, using the discrete topology if $a_i = *$, and the Cantor topology if $a_i = \omega$).*

Proof. Let f be computed by the type-2 machine M. We show that f is continuous, meaning $f^{-1}[S]$ is open in the input space for all sets S that are open in the output space. It suffices to show this for all basic opens S (associated with either a point (for $a_0 = *$) or a prefix (for $a_0 = \omega$)). We choose S defined by the word w_0, which is either the complete result (for $*$) or a finite prefix of it (for ω); in each case we show that the inverse image of this set is open. (A basic open that *isn't* defined by any result or finite prefix of a result lies entirely ouside f's range, so the inverse image of such an open is empty and therefore open.) Fix input $(y_1, \ldots, y_k) \in \mathrm{dom}(f) \subseteq Y := \Sigma^{a_1} \times \cdots \times \Sigma^{a_k}$.

Case $a_0 = *$: A basic open in Σ^* is a singleton $\{w\}$. Let M halt with w_i to the left of the head on each tape i ($i \in \{0, \ldots, k\}$). Since the input and output are immutable, w_i is a prefix of y_i; furthermore $w_0 = y_0 = f(\vec{y})$ by assumption. Because the input tapes are one-way and immutable, w_i is exactly the portion of y_i that M makes use of, in particular it does not use any part of y_i beyond this prefix, so $f^{-1}[\{w_0\}] = w_1\Sigma^{a_1} \times \cdots \times w_k\Sigma^{a_k}$, which is open in $\Sigma^{a_1} \times \cdots \times \Sigma^{a_k}$.

Case $a_0 = \omega$: A basic open in Σ^ω is the set $w\Sigma^\omega$ of all infinite extensions of a word w. Consider a configuration of M during computation of $f(\vec{y})$ (i.e. after some finite number of steps) such that w_i is to the left of the head on each tape $i \in \{0, \ldots, k\}$. By finality of the output tape, w_0 is a prefix of $f(\vec{y})$, i.e. $f(\vec{y}) \in w_0\Sigma^\omega$ (a basic open set in Σ^ω). By the fact that the input tapes are one-way and immutable, M has read precisely the prefix $w_i \sqsubseteq y_i$ for each input tape i, in other words $f_M^{-1}[w_0\Sigma^\omega] = w_1\Sigma^{a_1} \times \cdots \times w_k\Sigma^{a_k}$, which is open in $\Sigma^{a_1} \times \cdots \times \Sigma^{a_k}$. □

2.3.1 Computably continuous vs. primitive recursive

Any computable function has a computable modulus of continuity. It is interesting to ask whether we can reverse this implication, or if not, what extra condition is needed. In standard computability theory we define a *primitive recursive* function $f : Y \to \Sigma^*$ ($Y := (\Sigma^*)^k$) as one which either is one of a given set of basic functions, or (inductively) can be defined either by iteration at most $\sigma(x)$ times (where $\sigma(x)$ is the Gödel number of the input x) of some previously defined primitive recursive function or by composition of previously defined primitive recursive functions. More generally, a non-basic f is p.r. iff it can be defined either by composition or by at most $A(n)(\sigma(x))$-fold iteration (for some n) of some previously defined p.r. function(s), where A is the Ackermann function (every function $A(n)$ is p.r. by construction). That is, once we know x we can substantially restrict the time available to M_f for computing $f(x)$, such that M will nonetheless always halt before we force it to. Consider the following oracle TM with type $(\Sigma^*, \Sigma^*, \Sigma^*)$, where $\Sigma := \Sigma_{M_f} \cup \{\bot\}$ ($\bot \notin \Sigma_{M_f}$), and a numeric representation σ of the domain of the word function computed by M_f:

Input: a description of a TM M_f of type $((\Sigma_{M_f})^*, (\Sigma_{M_f})^*)$; a number n representing a branch of the Ackermann function; a string $w \in (\Sigma_{M_f})^*$ encoding a set of inputs for M_f. Any TM computing a p.r. function with strictly finite input and output can be translated into such a TM (given an appropriate Σ).

1. Compute $A(n)(\sigma(w))$, storing the result on a work tape.
2. Simulate M_f on input y, writing its output on a second work tape. After each transition, check whether the value on the first work tape is 0. If so, output $\bot$ and halt. Otherwise, decrement it.
3. If M_f halts (namely after not more than $c_f(x)$ transitions), copy the output work tape to the output tape and halt.

$A(n)$ is like an alarm clock: we set the alarm for $A(n)(x)$, and it goes off when it reaches 0. If f is primitive recursive, M_f exists and there is a branch $A(n)$ of the Ackermann function which tells us how high to set the alarm such that this TM never outputs $\bot$, but always halts with the value of $f(x)$ on its output tape before the alarm goes off. In other words, it is a universal TM for primitive recursive functions, but an 'anti-universal' TM for non-primitive recursive functions (it fails to compute any such).

Similarly, for functions f with infinite input, we can define a function $f : \mathbb{C} \to \mathbb{C}$ to be 'primitive corecursive' if it is computed by some TM M_f with some c_f, a "modulus of primitive corecursion", such that $c_f(i)$ is an upper limit on the number of steps required to compute the ith symbol of the output of M_f, and $A(n)$ majorises c_f for some n. It is equivalent to say that $A(n)$ itself is a modulus. Consider the following TM with type $(\Sigma^*, \Sigma^*, \Sigma^\omega, \Sigma^\omega)$:

Input: a description of a type-2 TM M_f of type $((\Sigma_{M_f})^\omega, (\Sigma_{M_f})^\omega)$; a number n representing a branch of the Ackermann function; a string w encoding a set of inputs for M_f (3 input tapes). Any type-2 machine with infinite output and at least one infinite input can be translated into a TM of this type (given an appropriate Σ).

1. Set $i := 0$ (i records the number of symbols of output produced so far).

2. Compute $c := A(n)(i) - A(n)(i-1)$ (setting $A(n)(-1) := 0$).

3. Simulate M_f on input y until it produces one symbol of output. Send this output straight to the output tape. After each transition, check whether $c = 0$. If so, halt. Otherwise, decrement it.

4. Increment i and go to step 2.

Again, this is a universal TM for primitive corecursive functions but anti-universal for non-primitive corecursive ones. Here is the connection to computable continuity:

Theorem 2.3.5. *If f is primitive corecursive, then it is computably continuous (has a computable modulus of uniform continuity).*

Proof. Any TM takes at least i steps to read i symbols of input, so a modulus of primitive corecursion is also a modulus of Cantor (uniform) continuity. Since the $A(n)$ that majorises the modulus is also a modulus and computable, there is also a computable modulus. □

Just as not all computable functions on countable sets are primitive recursive, it is probable that not all computable functions on uncountable sets are primitive corecursive. All that is required for a function to be computable is that after outputting a symbol, the machine runs for a finite number of steps before outputting another symbol. There is no particular reason why its modulus of primitive corecursion must be computable, or even exist: while the number of steps required to output the $(n+1)$th symbol must be finite, it need not be bounded by a computable function of n. But we do know that running for a finite number of steps implies reading a finite number of input symbols, which obviously implies Cantor continuity as a prerequisite.

Another notion of primitive corecursion has been defined by Leivant and Ramyaa [LR], in an equational style, in contrast to this machine-oriented style. At present it is unknown how the two definitions are related.

2.4 Representation of real numbers

There are various methods for constructing the real numbers from other sets; Dedekind cuts and Cauchy sequences of rational numbers are well known ones.

Definition 2.4.1 (Cauchy sequences)**.** A Cauchy sequence is a sequence $(x_i)_{i\in\mathbb{N}}$ of elements of a metric space such that for any $\epsilon > 0$, there is a k such that $d(x_i, x_j) < \epsilon$ for all $i, j > k$.

Cauchy sequences in general metric spaces do not always converge to an element of the space; the property that they do is called *Cauchy completeness*. $\mathbb{R}$ is Cauchy complete, but $\mathbb{Q}$ with the same metric is not. In fact, $\mathbb{R}$ is the *Cauchy completion* of $\mathbb{Q}$: the set of (equivalence classes of) Cauchy sequences of rational numbers is isomorphic to $\mathbb{R}$. A symbolic representation of real numbers in the form of infinite digit streams is essentially a notation for Cauchy sequences of rational numbers.

2.4.1 Infinite decimal expansions

The 'standard' representation of real numbers is as infinite strings of decimal digits $d_i \in \{\mathtt{0}, \ldots, \mathtt{9}\}$. There are a finite number (say $n+1$) of digits before the decimal point, and an infinite number after it. Call this language IDE, for 'infinite decimal expansion'. Each digit has an index: those before the point have a non-negative integer index $i < n$, those after the point a negative integer index $i \in \mathbb{Z} \setminus \mathbb{N}$. The streams are interpreted via the semantic functions $\sigma : \{\mathtt{0}, \ldots, \mathtt{9}\} \to \mathbb{R}$ and $\sigma : \mathrm{IDE} \to \mathbb{R}$ (given the same name for convenience):

$$\begin{aligned} \sigma(\mathtt{0}) &= 0 \\ \sigma(\mathtt{1}) &= 1 \text{ etc.} \\ \sigma(a) &= \sum_{-\infty}^{i=n} \sigma(a_i) \times 10^i \end{aligned}$$

With this semantics, if we interpret a finite prefix as a real number by setting all the remaining digits to 0, an infinite decimal expansion a_i with $n+1$ digits left of the decimal point notates the Cauchy sequence

$$\begin{aligned} c_0 &= a_n \times 10^n \\ c_i &= c_{i-1} + a_{n-i} \times 10^{n-i} \quad i > 0 \end{aligned}$$

The decimal representation is convenient, but only because it is familiar. If we use it with Type-2 TMs, we get a notion of computability that excludes some functions that should obviously be computable. For example, $x \mapsto 3x$ is not computable. $\mathtt{0.333}\ldots$ should map to either $\mathtt{0.999}\ldots$ or $\mathtt{1.000}\ldots$. Suppose a machine attempting to compute this function does output $\mathtt{0.999}$. If the next digit is $\mathtt{4}$, the machine must go back and overwrite the $\mathtt{0}$ with a $\mathtt{1}$, but cannot because the output tape is one-way. Nor can it delay producing output until it is sure the output is unambiguous, because for some inputs, in particular $\mathtt{0.333}\ldots$, it may delay forever and so not produce any output at all. Allowing the output tape to be overwritten allows this function to be computed, but, as Weihrauch says, "such machines are useless in practice, since no finite initial part of an infinite computation gives a reliable result, in general."[4] In short, the Euclidean-continuous function $x \mapsto 3x$ is not Cantor-continuous for this representation (two close input values do not require commensurately similar amounts of inspection to produce each digit of their corresponding outputs).

The problem is as follows. Define the successor function on digits in the obvious way, up to $\mathtt{9}$ ($\mathrm{suc}\,\mathtt{0} = \mathtt{1}$, $\mathrm{suc}\,\mathtt{8} = \mathtt{9}$). Note, again, the distinction between symbols (in typewriter face, e.g. $\mathtt{9}$) and actual numbers (in ordinary serif face, e.g. 9). We can interpret finite prefixes a_i ($-\infty < i \leq n$) as real intervals by extending the semantic function σ as follows (simplifying slightly to omit the decimal point):

$$\begin{aligned} \sigma\prime(a_n \ldots a_0.a_{-1} \ldots a_{-m}) &= \textstyle\sum_{-m}^{i=n} a_i \times 10^i \\ \sigma(wa_m\mathtt{9}^k) &= \left[\sigma\prime(wa_m\mathtt{9}^k),\ \sigma\prime(w(\mathrm{suc}\,a_m))\right) \quad a_m \neq \mathtt{9} \\ \sigma(wa_m) &= \left[\sigma\prime(wa_m),\ \sigma\prime(w(\mathrm{suc}\,a_m))\right) \quad a_m \neq \mathtt{9} \\ \sigma(\mathtt{9}^k) &= \left[\sigma\prime(\mathtt{9}^k),\ \sigma\prime(\mathtt{10}^k)\right) \end{aligned}$$

This defines the meaning of a finite prefix w in terms of the range of meanings of its infinite extensions. Figure 2.1 illustrates the situation.

[4] This, including the quotation, is Example 2.1.4.7 in [Wei00].

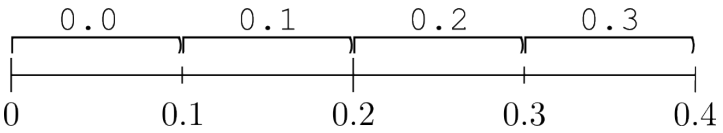


Figure 2.1: Half-open intervals defined by decimal prefixes

The problem with the decimal representation is precisely that the intervals don't overlap. When a TM outputs a digit, it knows that the value it is computing is definitely in the associated interval. But if the value it is computing is very close to the boundary between the two intervals, it cannot commit to either interval until it has enough information — and in the case of the extreme lower end of one of these half-closed intervals, no finite amount of information is enough.

The signed digit stream representation (previously used in [EH02a] and [MRE07]) solves this problem.

2.4.2 Signed digit streams

An infinite stream of digits $d_i \in \{-1, 0, 1\}$ represents a real number in the unit interval $x \in \mathbb{I}$:

$$\sigma(a) = \sum_{i \geq 0} a_i \times 2^{-(i+1)}$$

To represent real numbers, we pair a signed digit stream a with a rational number q, and interpret $\sigma(q, a) = q + \sigma(a)$. The intervals described by these digits are illustrated in Figure 2.2.

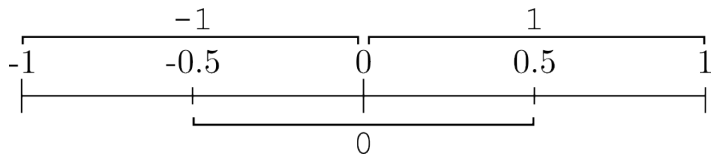


Figure 2.2: Half-open intervals defined by signed digit stream prefixes

Like the decimal representation, a TM producing a signed binary digit commits to one of these intervals. Unlike the decimal representation, if the value it is computing is close to a boundary, it can still commit to a digit, because there is some digit such that only finite information is required to determine that the value is definitely within that digit's interval.

Chapter 3

Realisability and program extraction

In this chapter I discuss the process of extracting programs from constructive proofs, using the realisability interpretation of constructive first order logic, which was used to produce the implementation described in chapter 5. The process makes use of the Curry-Howard correspondence between logic and functional programs, which I therefore also describe.

3.1 The Curry-Howard correspondence

When writing a program, we often want to prove properties of it. For example, we might want to guarantee that the output of a sorting function is a list with the same elements as its input in order; or, less trivially, that a concurrent algorithm has no race conditions and cannot deadlock. The former is an example of an algebraic property, which I will restrict myself to for now. (Given an appropriate calculus, the latter might also be an algebraic property, but such calculi are beyond the scope of this thesis.) Here are some examples of algebraic properties:

Definition 3.1.1. Let X be a set. Define lists of X of length n as functions $l : \bar{n} \to X$ where $\bar{n} = \{0, 1, \ldots, n-1\}$. Abbreviate this by $\mathrm{List}(X, n)$. Define $\mathrm{List}(X) = \{l \mid \exists n \in \mathbb{N}.\ l : \mathrm{List}(X, n)\}$. Let $f : \mathrm{List}(X) \to \mathrm{List}(X)$ be a function. Let $\leqslant$ be a linear ordering on X.

1. f is a *permutation* if any list l contains the same elements as $f(l)$. Concretely, for any list $l : \bar{n} \to X$ there is an injective function $\tilde{f}_l : \bar{n} \to \bar{n}$ that translates indexes of $f(l)$ to indexes of l, i.e. such that $f(l)(i) = l(\tilde{f}_l(i))$ for all $i \in \bar{n}$.

2. A list l is *ordered* if it is monotone, i.e. if for all $i, j \in \mathbb{N}$, if $i \leqslant j$ then $l(i) \leqslant l(j)$.

3. f is a *sorting* if it is a permutation and $f(l)$ is ordered for all lists l.

We can prove order properties of f, l etc. using predicate logic. The (normal[1]) form of a proof depends on the form of the definition:

1. To prove that f is a permutation, let l be a list, without assuming anything else about it, and define a function $\tilde{f}_l : \mathbb{N} \to \mathbb{N}$ and prove that it is injective; then let $i \in \bar{n}$ and prove $f(l)(i) = l(\tilde{f}_l(i))$.

2. To prove that a list l is ordered, let $i, j \in \bar{n}$, further assume that $i \leqslant j$, and prove that $l(i) \leqslant l(j)$.

3. To prove that f is a sorting, first prove that it is a permutation; then let l be an arbitrary list and prove that $f(l)$ is ordered.

As a thought experiment, suppose we make proofs first-class algebraic objects. Then we can think of proofs as objects, and implications as translating proofs into proofs:

1. To prove that f is a permutation, define a function $\tilde{f}_l : \mathbb{N} \to \mathbb{N}$ *and also* construct a proof of the conjunction

$$f \text{ is injective} \wedge (\forall i \in \bar{n}.l(i) = l(\tilde{f}_l(i))).$$

2. To prove that l is ordered, define a function that accepts a proof that $i \leqslant j$ and uses it to construct a proof that $l(i) \leqslant l(j)$.

3. To prove that f is a sorting, construct a proof of the conjunction

$$(f \text{ is a permutation} \wedge \forall l \in \mathrm{List}(X).f(l) \text{ is ordered}),$$

that is, a proof that f is a permutation and a function that, given a list l, constructs a proof that $f(l)$ is ordered.

3.1.1 Propositions vs. types, proofs vs. values

If we follow through with the thought experiment, we see how proofs begin to look like functional programs. A proof resembles a function that takes as parameters proofs of whatever assumptions it makes and parameters instantiating any universally quantified variables, and returns a proof of the property it is trying to prove, including witnesses to any existentially quantified propositions. A proof proves a proposition, which is the 'type' of the proof, resembling the type of the object that the proof resembles. This resemblance is called the *Curry-Howard correspondence*, after the logicians Haskell Curry [Cur34] and William Alvin Howard [How80]. Consider the ordinary types in a purely functional language, say, the simply typed lambda calculus with algebraic data types:

- Truth $\top$, which has only one (trivial) normal-form proof, resembles a singleton type 1.
- Falsehood $\bot$, which (in a consistent logic) has no proof, resembles an empty type 0.
- A conjunction $P \wedge Q$ resembles a product $P \times Q$.

[1] Proofs do not always have normal forms, but for the sake of illustration we pretend they do.

- A disjunction $P \vee Q$ resembles a sum $P + Q$.
- An implication $P \Rightarrow Q$ resembles a function type $P \to Q$.
- A negated proposition $\neg P$ is the same as a proposition $P \Rightarrow \bot$ asserting that P implies the absurd, which resembles a function type $P \to 0$.

To prove a proposition P, we construct a value[2] of the type that P resembles. So to prove a conjunction $P \wedge Q$ we prove P by constructing a value of the type corresponding to P, and Q by constructing a value of the type corresponding to Q, and pair them together (such a pair being a value of type $P \times Q$, the type corresponding to $P \wedge Q$). If P corresponds to X, the circumstances under which we can prove P resemble the form of values of type X:

- We can prove $\top$ under any circumstances. A value of type 1 is available everywhere, namely its only value $*$.
- We can never prove $\bot$, unless we made inconsistent assumptions (like $P \wedge \neg P$), the proof isn't syntactically valid, or we used an invalid argument (for example we used *modus ponens* with the wrong premise, or circular reasoning), which isn't really a proof. Values of type 0 have no form at all, unless our parameters couldn't possibly all exist at the same time and we're actually living in a fantasy world (like $(x, f) : P \times (P \to 0)$, then $f\ x : 0$); or the program has a syntax error, is ill-typed, or contains an infinite, unproductive loop, and so isn't really a value.
- We can prove $P \wedge Q$ whenever we can prove both P and Q with no extra assumptions. A value of $P \times Q$ takes the form of a pair, whose constituents are a value of type P and a value of type Q, which we produce using the same set of parameters.
- We can prove $P \vee Q$ whenever we can prove one of P or Q with no extra assumptions; then we use the left or right introduction rule. A value of type $P + Q$ takes the form of a value of type P or a value of type Q, usually together with a label saying which it is, which we produce with (some of) the current parameters. (Note the remark about normal forms; in non-constructive logics we can prove a disjunction without proving either of its components.)
- We can prove $P \Rightarrow Q$ whenever, given exactly one extra assumption P in addition to whatever other assumptions we've made, we can prove Q. A value of $P \to Q$ takes the form of a function which takes a parameter of type P and uses it, along with whatever other parameters are in scope, to produce a value of type Q. ('Scope' here means static lexical scope.)

(To be precise, a logical introduction rule for P resembles a data constructor with P as result type.)

Dependent type theory (introduced by Per Martin-Löf [ML80]) extends the correspondence to predicate logic. At this stage the boundaries between the

[2]Mathematicians usually talk about 'objects'. Here I use the term 'value' instead, in deference to the tradition in the functional programming community of avoiding confusion with the term 'object' from object-oriented programming languages; and also because propositions are 'objects' in a sense, but certainly not values.

universes of proposition, proof, type and value begin to blur; in particular, propositions begin to refer inextricably to types and values, and types and proofs begin to refer to values as well, and even treat propositions and types in some ways as if they were values.

- A predicate P applied to a value, type, proposition or proof x (i.e. $P(x)$) resembles a polymorphic type $F\ a$. The circumstances under which we can prove a proposition $P(x)$ depends on what exactly x is and what exactly P asserts about it. The form taken by a value of type $F\ x$ depends on what exactly x is and what use F makes of it.

- A universal quantification $\forall x \in X.P(x)$ resembles a dependent product $\Pi x : X.F\ x$. To prove $\forall x \in X.P(x)$, we let x be an arbitrary value in the set X and prove $P(x)$. A value of type $\Pi x : X.F\ x$ takes the form of a function that takes a parameter x of type X and uses it, along with whatever other parameters are in scope, to produce a value of type $F\ x$. Notice that the precise type $F\ x$ depends on the parameter x.

- An existential quantification $\exists x \in X.P(x)$ resembles a dependent sum $\Sigma x : X.F\ x$. To prove $\exists x \in X.P(x)$ we construct a value x using the current assumptions and prove $x \in X$ and $P(x)$. A value of type $\Sigma x : X.F\ x$ takes the form of a pair, whose constituents are a value x of type X and a value of type $F\ x$. Notice that the type of the second constituent depends on the first constituent (not just its type).

3.1.2 Proof-irrelevance and computational content

Dependently typed languages (an example being Agda [Agd]) take full advantage of the Curry-Howard correspondence as it applies to predicate logic. Since types can refer to values, they allow a huge variety of properties (propositions about data) to be stated (as types), and proved (as programs). Many types may be empty when all their variables are instantiated, and for many of those, all the programmer cares about a value of that type is that it exists, not what form it takes. These correspond to pure *propositional types*, as opposed to *data types*.

The idea that one proof is as good as another, that we only care whether it exists or not, is called *proof irrelevance*. For example, as I mentioned earlier, most mathematicians only care that a set $\mathbb{R}$ exists that has the properties of the real numbers, not how those properties are proved (at least in their day to day work with real numbers). But if we want to define a data type corresponding to the 'proposition' $\mathbb{R}$, we need to know the details of some of those proofs, because different proofs will correspond to different values/programs and result in a different implementation. Nonetheless, some properties are only extra information that we can use to optimise the implementation or prove that it is correct, and we don't need them after compiling the program. Such properties are said to have no *computational content*, the programming counterpart to proof irrelevance.

3.2 Program extraction

3.2.1 Using Curry-Howard directly

Since proofs resemble programs, it is appropriate to ask what program any given proof resembles. For example, take the following proposition:

Proposition 3.2.1. *For any $l \in \mathrm{List}(\mathbb{N}, n)$ ($n \in \mathbb{N}$), there exists a list $l' \in \mathrm{List}(\mathbb{N}, n)$ which is a sorted permutation of l.*

Definition 3.2.2. A list $l : \mathrm{List}(X, n)$ is a *sublist* of a list $o : \mathrm{List}(X, m)$ iff there is a monotone injective function $\tilde{o}_l : \bar{n} \to \bar{m}$ with $l(i) = o(\tilde{o}_l(i))$ for all i. In other words all the elements of l are in o in the same order.

Definition 3.2.3. The *tail* of a list l of length $n+1$ is its sublist l' of length n defined by $l'(i) = l(i+1)$ for all i. In other words the tail is just l with its first element removed.

Proof of proposition 3.2.1. Let $l \in \mathrm{List}(\mathbb{N})$. Proof proceeds by strong induction on n, the length of l.

- *Case* $n = 0$. Set $l' = l$. This is trivially a permutation and vacuously sorted.
- *Case* $n = n' + 1$. Let $l_{\leqslant}$ be the largest sublist of the tail of l such that $l_{\leqslant}(i) \leqslant l(0)$ for all $i > 0$, similarly $l_{>}$. Let $m_{\leqslant}$ and $m_{>}$ be their respective lengths. $m_{\leqslant}$ and $m_{>}$ are both strictly less than n, so by the induction hypothesis each sublist has a sorted permutation, respectively $l'_{\leqslant}$, $l'_{>}$. Then set
$$l'(i) = \begin{cases} l'_{\leqslant}(i) & \text{if } i < m_{\leqslant} \\ l(0) & \text{if } i = m_{\leqslant} \\ l'_{>}(i - m_{\leqslant} - 1) & \text{if } i > m_{\leqslant} \end{cases}$$
Since $\leqslant$ is transitive, $l'_{\leqslant}(i) \leqslant l(0) < l'_{>}(j)$ for all i, j. Further, since $l'_{\leqslant}$ and $l'_{>}$ are both sorted, so is l'. Since $\leqslant$ is total, $l(i) = l'_{\leqslant}(j)$ or $l(i) = l'_{>}(j)$ for some j for all $i > 0$. Therefore l' is a permutation of l. □

What program does this resemble? Examine the proof step by step. First, we figure out what type of program it is. The proposition is a universal quantification over natural numbers and lists of that length, so the program is a (dependently typed) function taking a number and a list as arguments:

$$\Pi n : \mathbb{N}.\Pi l : \mathrm{List}(\mathbb{N}, n).\cdots$$

The quantified proposition is an existential, so the function's output is a (dependently typed) pair of a list and a value corresponding to the property that it satisfies, which is that the output list is a sorted permutation of the input. Semi-formally, the type is:

$$\mathrm{SortedList}(l, n) := \Sigma l' : \mathrm{List}(\mathbb{N}, n).l' \text{ is a permutation of } l \times l' \text{ is sorted}$$

So the whole type of the program is

$$\Pi n : \mathbb{N}.\Pi l : \mathrm{List}(\mathbb{N}, n).\mathrm{SortedList}(l, n)$$

Now, to the proof. First of all, it is a function:

$$\lambda n : \mathbb{N}.\lambda l : \mathrm{List}(\mathbb{N}, n).\cdots$$

The proof then proceeds by induction on n. The strong induction (or well-founded induction) axiom asserts: if $Q(n)$ whenever $Q(i)$ for all $i < n$, then $Q(i)$ for all i. The strong induction axiom schema is

$$(\forall n \in \mathbb{N}.(\forall i \in \mathbb{N}.i < n \Rightarrow Q(i)) \Rightarrow Q(n)) \Rightarrow \forall i \in \mathbb{N}.Q(i)$$

for any given Q. This resembles the type of the wellfounded recursion operator for natural numbers, which we will call wfrec; in fact, it is identical apart from replacing '$\forall$' with 'Π' and '$\in$' with ':':

$$(\Pi n : \mathbb{N}.(\Pi i : \mathbb{N}.i < n \to Q(i)) \to Q(n)) \to \Pi i : \mathbb{N}.Q(i)$$

(This operator is not extracted directly from a proof of wellfoundedness of $<$, but can be derived indirectly. It is a sort of fixpoint operator that provides an inductive proof that the program will terminate. The base case for induction is provided by the fact that $i < 0$ is false for all i.) We need to use it with $Q(i) = \Pi l : \mathrm{List}(\mathbb{N}, i).\mathrm{SortedList}(l, i)$, that is, at the following type:

$$\begin{array}{c}(\Pi n : \mathbb{N}.(\Pi i : \mathbb{N}.i < n \to \Pi l : \mathrm{List}(\mathbb{N}, i).\mathrm{SortedList}(l, i)) \\ \to \Pi l : \mathrm{List}(\mathbb{N}, n).\mathrm{SortedList}(l, n)) \\ \to \Pi i : \mathbb{N}.\Pi l : \mathrm{List}(\mathbb{N}, i).\mathrm{SortedList}(l, i)\end{array}$$

This is an exceedingly complex type, containing a lot of sub-types with no computational content. Similarly, the program of this type (quicksort) is littered with proof-terms that we only need to be sure the program is correct. These correctness annotations, as we can think of them, are only needed when we are writing the program; we could remove the proofs and have a perfectly valid program.

3.2.2 Using realisability

I mentioned earlier that in non-constructive logics we can prove a disjunction without proving either of its components. The converse property of a constructive logic, that every proof of a disjunction is a proof of one of its components, is called the disjunction property. Similarly, in constructive logics but not non-constructive ones, every proof of an existential includes a witness (the existential property). S. C. Kleene, an intuitionist, invented realisability in an attempt to characterise this without resorting to meta-proofs [Kle45]. Kreisel's modified realisability simplifies the idea by using typed lambda calculus [Kre59]. A symbolic object realises a proposition, and in modified realisability additionally each proposition P has an associated type $\tau(P)$ of potential realisers. Berger's uniform realisability [Ber09b] is the variant used here. Here are some examples:

- $2 : \mathtt{Nat}$ realises the proposition $2 \in \mathbb{N}$;
- if $x : X$ realises a proposition P and $y : Y$ realises Q, then the pair $\langle x, y\rangle : X \times Y$ (of a product type) realises the conjunction $P \wedge Q$, and $\mathrm{inl}\, x : X + Y$ and $\mathrm{inr}\, y : X + Y$ (of a disjoint union type) realise the disjunction $P \vee Q$;

- if $f(x) : Y$ realises Q whenever $x : X$ realises P, then the function $f : X \to Y$ (of a function type) realises a proposition $P \to Q$;
- if $x : X$ realises $A(y)$ for some y, then x realises $\exists y.A(y)$;
- if $x : X$ realises $A(y)$ for all y, then x realises $\forall y.A(y)$.

Notice that in the existential and universal cases, the type of potential realisers does not depend on the quantified variable y, as it does in Kleene and Kreisel's versions of realisability; a realiser of $\forall x : X.\exists y : Y.P(x, y)$ is not a Skolem function $f : X \to Y$ satisfying $\forall x : X.P(x, f(x))$, but rather just a realiser of $P(x, y)$ for all x and some y that depends on x.

There are also rules that discard proof-irrelevant, or more precisely non-computational, formulae: if A is non-computational, then (for example) the type $\tau(A \wedge B)$ of realisers of $A \wedge B$ is the same as the type $\tau(B)$ of realisers of B. Essentially, the process discards the first order and non-computational parts, yielding simple (as opposed to dependent), non-trivial types of realisers.

We can use modified realisability as a process for extracting programs from proofs. A proof is not only a definition of an algorithm, but also a proof that it is correct. We only need the proof to show that the algorithm is correct, but the proof and definition may be intertwined. With realisability we can extract just the algorithm. An interesting example is the wellfounded recursion operator mentioned before. Let $\mathrm{WfR}(a) := \Sigma <: a \times a.\mathrm{Wf}(<)$ be the proposition "there exists a wellfounded relation on a" ($\mathrm{Wf}(<)$ meaning "$<$ is wellfounded"). Then, given a set a and a predicate P over a, the wellfounded induction schema (which essentially says "if there is a wellfounded relation on a then induction is valid for a") is the proposition

$$\begin{aligned}&\forall \langle <, p\rangle \in \mathrm{WfR}(a).\\ &\quad (\forall x \in a.(\forall y \in a.y < x \to P(y)) \to P(x)) \to \forall x \in a.P(x)\end{aligned}$$

Using Curry-Howard directly, given types a and P realising respectively a and P, a wellfounded recursion operator should have the type

$$\begin{aligned}&\Pi\langle <, p\rangle : \mathrm{WfR}(a).\\ &\quad (\Pi x : a.(\Pi y : a.y < x \to P(y)) \to P(x)) \to \Pi x : a.P(x)\end{aligned}$$

Applying realisability, on the other hand, we find $\mathrm{WfR}(a)$ is non-computational, and the propositions involving its components are discarded along with the first order part (the Πs), leaving us with this type:

$$(P \to P) \to P$$

which is the type of a fixpoint combinator. In other words, the program extracted from the wellfounded induction schema is general recursion! The fact that a program extracted from an application of wellfounded induction is terminating is not needed at run time, and is therefore not present.

3.3 Coinduction

In §2.3.1 I used the term 'primitive corecursive'. By analogy with 'primitive recursive' this suggests there is a class of 'corecursive' functions of which it is a

subset, as primitive recursive functions are a subset of the recursive functions. The counterpart for corecursive functions of the property of termination for recursive functions is *productivity*: a correct corecursive function must produce some output (but not necessarily all of it) after a finite amount of time.

On the logic level, we define induction and coinduction on strictly positive operators $\Phi := \lambda X.\mathcal{P}$ ($\mathcal{P}$ a predicate, in which X may appear free but not in the premise of any implications) via least and greatest fixed points (with respect to set inclusion) of these operators ($\mu\Phi$ and $\nu\Phi$, respectively). These are defined via the following axioms:

Closure	$\Phi(\mu\Phi) \subseteq \mu\Phi$	Induction	$\Phi(\mathcal{Q}) \subseteq \mathcal{Q} \to \mu\Phi \subseteq \mathcal{Q}$
Coclosure	$\nu\Phi \subseteq \Phi(\nu\Phi)$	Coinduction	$\mathcal{Q} \subseteq \Phi(\mathcal{Q}) \to \mathcal{Q} \subseteq \nu\Phi$

(Co)closure and (co)induction simply define μ and ν. So for example the set of natural numbers

$$\mathbb{N} := \mu X.\{0\} \cup \{x + 1 \mid X(x)\}$$

is the least fixpoint of the strictly positive operator

$$\Phi(X) := \{0\} \cup \{x + 1 \mid X(x)\}.$$

For example, $0 \in 1 = \Phi(\emptyset) = \{0\}$ and also $0 \in \Phi(\Phi(1))$ — Φ yields a set that is a superset of its argument. The induction axiom expresses the fact that $\mu\Phi$ is the least fixed point of Φ, the smallest set for which Φ does not yield a strictly larger set.

The application for our purposes is in the definition of real numbers. The unit interval is the set

$$C_0 := \nu X.\{(i + x)/2 \in \mathbb{I} \mid \mathrm{SD}(i) \wedge X(x)\}$$

where $\mathrm{SD} := \{-1, 0, 1\}$, the signed digits. A realiser of this set is a coinductive data type τ whose elements consist of a digit and a τ, i.e. streams of digits. Note that C_0 defines a set of real numbers, but its realisers are not reals but streams whose relation to the reals is defined by the realisability interpretation to be exactly the semantic function σ defined earlier.

This is in fact just a special case, the general case being:

$$C_n = \nu X.\mu Y.\{f \mid \exists d(f[\mathbb{I}^n] \subseteq \mathbb{I}_d \wedge X(\mathrm{va}_d \circ f)) \vee \exists i \forall d Y(f \circ \mathrm{av}_{i,d})\}$$

which is a coinductive definition of uniform continuity for n-ary functions; its realisers are coinductive data structures which are isomorphic to functions $f : \mathbb{I}^n \to \mathbb{I}$, via an isomorphism similar to the memoization isomorphism described in Chapter 4. (va_d and $\mathrm{av}_{i,d}$ are defined in Section 5.2.2; C_n is introduced and treated more fully in [Ber09a].)

Chapter 4

Background on memo tries

This chapter describes the method used for memoization of functions on real numbers. Real numbers are interpreted as streams of digits; mathematically, they are functions $p : \mathbb{N} \to D$. We exploit the interpretation of functions with algebraic domain of [Hin00b] as coinductive data structures, specifically generalised tries, to recover the 'data-ness' of real numbers. The approach additionally uses this memoisation scheme for functions over real numbers, making them more time-efficient for close values.

4.1 Memoization à la Hinze

Hinze [Hin00b] describes a general method for defining a memo function $f : A \to Z$ for any (inductive) algebraic data type A and any type Z, by defining an associated type of *generalised trie* (also called prefix trees). This is a lookup table whose shape is determined by the domain A; it is similar to finite maps implemented as ordered trees (e.g. Haskell's `Data.Map`), except that the trees are infinite in general (each trie has as many nodes as the domain has elements).

4.1.1 Memo tries

An algebraic data type is one of the sets generated from the following operations:

- singleton sets $1 = \{*\}$ ("unit type");
- sums, or coproducts, $X + Y$ for algebraic data types X and Y;
- products $X \times Y$ for algebraic data types X and Y.

Sums and products are unique up to isomorphism. With morphisms inherited from $\mathfrak{Set}$, algebraic data types therefore form a category with all finite products and all nonempty finite coproducts (all finite coproducts with the addition of empty types).[1] The motivation behind [Alt01] is, in essence, to discover whether

[1] Data types in a programming language are usually concrete categories over $\mathfrak{Dcpo}$, the category of directed-complete partial orders and monotone functions, or its subcategory with only Scott-continuous functions; but there are no coproducts in such a category because they are not unique — both separated and coalesced sums exist. I am glossing over this technicality here.

the category of algebraic data types also contains certain exponentials Y^X (or $X \Rightarrow Y$). The conclusion is that it does contain those exponentials where X is inductive, and therefore such function types are represented by algebraic data types (specifically, terminal coalgebras).

In a Cartesian closed category $\mathfrak{C}$ with finite coproducts, the following laws apply (for all objects X, Y and Z): [AHS90]

- $Z^1 \simeq Z$
- $Z^{A+B} \simeq Z^A \times Z^B$
- $Z^{A\times B} \simeq (Z^B)^A$

where $\simeq$ means "is isomorphic to". More generally, these laws hold whenever the relevant objects exist, regardless of Cartesian closure, by the definition of exponential objects. By structural induction, if A and Z are algebraic, then Z^A is isomorphic to an algebraic data type $\hat{A}(Z)$. We can interpret $\hat{A}(Z)$ as a container of values of type Z. Then we don't care that Z is algebraic any more, and use $\hat{A}$ simply as a lookup table for arbitrary types, keyed by the type A. We call these lookup tables *generalised tries* (after [Hin00a]) or *tries* for short.

Obviously for types with infinitely many elements, such as trees and lists, such tries will be infinitely deep; that is, the type is not inductive, rather it is coinductive. (Algebraic types can be described quite generally as finitely branching trees, meaning that the values in an infinite set of values of the same type will be arbitrarily deep; by König's lemma this means the representative tries will be *infinitely* deep.) Categorically speaking the type of tries keyed on a type which is an initial algebra (i.e. least fixed point) for some functor is the terminal coalgebra (i.e. greatest fixed point) for a 'canonical' other functor. Specifically, if X is the initial algebra of the functor F_X, define $\hat{F}_X$ as this associated functor, namely:

- $\hat{F}_1$ (1 being any terminal object) is the identity functor $\mathrm{id}_{\mathfrak{C}}$. (Every object is a fixed point of $\mathrm{id}_{\mathfrak{C}}$.)
- $\hat{F}_{X+Y}$ is the product of functors $\hat{F}_X \times \hat{F}_Y$, i.e. $\Lambda Z.\hat{F}_X(Z) \times \hat{F}_Y(Z)$.
- $\hat{F}_{X\times Y}$ is the composition of functors $\hat{F}_X \circ \hat{F}_Y$, i.e. $\Lambda Z.\hat{F}_X(\hat{F}_Y(Z))$.
- $\hat{F}_{\mu X.F_{A'}(X)}$ (the functor for the least fixed point of the functor $F_{A'}$) is the greatest fixed point functor $\Lambda Z.\nu X.\hat{F}_{A'}(Z)(X)$.

(These definitions come from [Alt01]. Any of these definitions can be replaced by naturally isomorphic functors.) The functor $\hat{F}_A$ is the *generalised trie* functor for A, and $\hat{F}_A(Z)$ is the type of generalised tries keyed by values of type A and storing values in Z.

Examples

The Peano natural numbers $\mathbb{N}$ are defined algebraically as

$$\mathbb{N} \quad = \quad \mu X.1 + X$$

which is the initial algebra (i.e. least fixed point) of the functor

$$F_{\mathbb{N}}(X) \quad = \quad 1 + X.$$

The above isomorphisms, more formally stated, imply that

$$\begin{aligned}\hat{F}_{\mathbb{N}}(Z) &= \hat{F}_{\mu X.1+X}(Z)) \\ &= \nu X.Z \times X\end{aligned}$$

i.e., the type of infinite streams of Z's. In this case replacing ν with μ yields the empty set, showing that tries cannot in general be inductive types.

The type of finite lists of natural numbers is

$$\begin{aligned}\text{List}(\mathbb{N}) &= \mu X.1 + \mathbb{N} \times X \\ &= \mu X.1 + (\mu Y.1 + Y) \times X\end{aligned}$$

and its associated trie type is

$$\begin{aligned}\hat{F}_{\text{List}(\mathbb{N})}(Z) &= \hat{F}_{\mu X.1+(\mu Y.1+Y)\times X}(Z) \\ &= \nu X.Z \times (\Lambda W.\nu Y.W \times Y)(X) \\ &= \nu X.Z \times (\nu Y.X \times Y)\end{aligned}$$

In this table we have:

- an entry for the empty list, which gives a single result in Z;
- an entry for nonempty lists. This is a table with
  - an entry for lists starting with 0, which is a table for the rest of the list;
  - an entry for lists starting with $S(n)$, which is a table for n.

4.1.2 Memoization using tries

Suppose we have such a table $\tau_f : \hat{F}_A(Z)$, with the property that the position indexed by $a : A$ stores the value $f(a) : Z$ (where $f : A \to Z$). Then we can define a function $\hat{f} : A \to Z$ which takes the parameter $a : A$ and looks up the corresponding $f(a) : Z$ stored in τ_f. Now $f = \hat{f}$ extensionally, but not intensionally, since $\hat{f}$ is defined in terms of $\hat{F}_A(Z)$, so it may have different operational semantics. This operational semantics can be much more time-efficient since $\hat{f}$ doesn't need to do any work, but only needs to look up a value in a table.

Obviously if we are to implement such a function in a programming language, the type $\hat{F}_A(Z)$ has to be lazy, with $\hat{f}$ building only those parts of τ_f that it actually looks at, or else it will enter an infinite loop trying to build τ_f. Then computing $\hat{f}(a)$ for the first time has the side effect of computing the value $f(a)$ that it is expecting to find. This gains nothing immediately; but if $\hat{f}(a)$ is computed again, the computation of $f(a)$ is avoided. This technique, of caching computed values of a function for use later, or more precisely, of converting a function to do this, is called *memoization*.

With $\hat{F}_A(Z)$ in particular, we know that there exist isomorphisms

$$\begin{aligned}\text{trie} &: Z^A \to \hat{F}_A(Z) \\ \text{untrie} &: \hat{F}_A(Z) \to Z^A.\end{aligned}$$

By 'isomorphism' we mean 'isomorphism up to extensional equality'. In other words untrie $\circ$ trie is not the (intensional) identity, but rather it maps functions to functions that are extensionally equal. This implies the existence of a function

$$\text{memoize} : (A \to Z) \to (A \to Z)$$

defined by

$$\text{memoize}(f) = \text{untrie}(\text{trie}(f))$$

The *specification* of this function is that it is extensionally equal to the identity function, but produces $\tilde{f}$ from f by building a trie representing f and returning the lookup function for this trie as a closure. In a lazy language this instead creates a thunk (suspended computation) for the root of the trie, which is then run when $\tilde{f}$ (more specifically, the function untrie used in its definition) tries to pattern-match on it. This thunk then creates the skeleton for the top level, then creates thunks for each branch and continues running only the branch it needs for the specific value it is currently looking up. In this way only those parts of the trie actually exist in memory which have been visited on the way to values of f which have already been computed.

4.1.3 Memoization of functions on signed digit streams

Let D be a finite set of digits (for example the signed binary digits $SD = \{-1, 0, 1\}$), D^* the set of finite lists of digits (i.e. finite approximations of real numbers), D^ω the set of streams of signed digits (representing a real number, as described in chapter 2), and $\bar{x} : D^\omega$ a digit stream representing x. (Convention: the bar $\bar{}$ is a representation by infinite streams, the tilde $\tilde{}$ an approximation of the representation by finite prefixes.) A uniformly continuous function $f : \mathbb{I} \to \mathbb{I}$, interpreted as an (abstract) function $\bar{f} : D^\omega \to D^\omega$, can be represented by a particular

$$\tilde{f} \quad : \quad D^* \to D^*$$

such that

$$\begin{aligned} &\forall x \in \mathbb{R}.\bar{f}(\bar{x}) = \overline{f(x)} \wedge \\ &\quad \lim_{|\tilde{x}| \to \infty} f(\tilde{x}) = \bar{x} \wedge \\ &\quad \forall \tilde{x}, \tilde{x}'.\tilde{x} \sqsubseteq \tilde{x}' \sqsubseteq \bar{x} \Rightarrow \tilde{f}(\tilde{x}) \sqsubseteq \tilde{f}(\tilde{x}') \sqsubseteq \bar{f}(\bar{x}) \end{aligned}$$

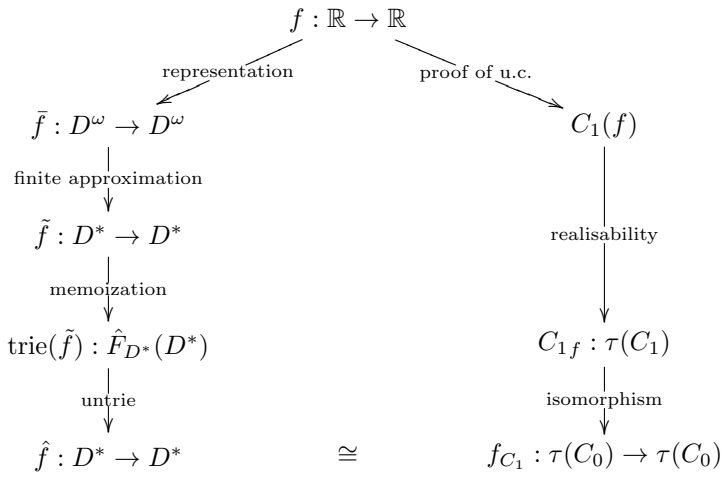
where $\sqsubseteq$ denotes prefixes. That is, $\tilde{f}$ approximates and converges to $\bar{f}$, and is monotone with respect to prefixes: giving it longer input does not make it 'change its mind'.

Then a memoized version $\hat{f}$ of $\tilde{f}$ is a function such that

1. $\hat{f}$ is extensionally equal to $\tilde{f}$;
2. for every $x : D^*$, for every $x' : D^*$ with $x \sqsubseteq x'$, whenever $\hat{f}(x)$ has already been computed, the computation of $\hat{f}(x')$ looks up the part of the computation already done for x rather than doing it again.

This combination is possible precisely because f is uniformly continuous: at a particular output precision, variations in the input that are within the precision given by the modulus of continuity for f (that is, that only affect digits after a particular length of prefix) do not affect the value of f up to that precision (that is, do not change the prefix of the value up to that length).

Realisability extracts a tree from a proof of a proposition $C_n(f)$ which is very similar to the memo tries $\hat{F}_{D^*}(\xi)$ (where ξ is the trie for C_{n-1}). The following diagram illustrates the idea:



Put another way, $\tau(C_1)$, which is a coinductive type of trees representing uniformly continuous functions with domain $\mathbb{I}$ of infinitely large objects, seems to be isomorphic to $\hat{F}_{D^*}(D^*)$, a coinductive type of trees representing continuous functions with domain D^* of finite objects. A data structure essentially identical to $\tau(C_1)$ is described in [HPG09].

Chapter 5

Prototype implementation

In this chapter I describe Haskell code implementing the predicates C_n, in particular the improvements I made. These mostly take the form of organising the code and using advanced language extensions to make the code take on a form that more closely resembles the proofs.

5.1 Implementing memo tries in Haskell

5.1.1 Generic Haskell: polytypic programming

Shortly before Altenkirch's paper [Alt01], Ralf Hinze covered much the same ground from a more practical point of view [Hin00b]. Andres LÃ¶h et al have created *Generic Haskell* (GH), a dialect of Haskell which enables what he calls *polytypic programming*. This is a form of polymorphism in which functions, and associated data types, are defined by induction over the structure of the type. For example, the type of memo tries is defined as follows (note this is a definition on the meta-level, not GH itself):

$$\begin{array}{lcl}
TABLE\langle * \rangle & = & * \to * \\
TABLE\langle k \to l \rangle & = & TABLE\langle k \rangle \to TABLE\langle l \rangle \\
Table\langle a \rangle & = & table_a \\
Table\langle t :: k \rangle & = & TABLE\langle k \rangle \\
Table\langle t\ u \rangle & = & (Table\langle t \rangle)(Table\langle u \rangle) \\
Table\langle \Lambda x.t \rangle & = & \Lambda table_x.Table\langle u \rangle \\
Table\langle \mu a.t \rangle & = & \mu table_a.Table\langle t \rangle
\end{array}$$

where a is a type that has its memo trie $trie_a$ defined separately. Essentially this extends the Haskell concept of pattern matching on data to the type (and kind) level. Like pattern matching, this generic approach is only suitable for types where there is no significant abstraction from the concrete representation. For abstract types this approach may distinguish values we want to consider equal; for example given a rational number type implemented as a pair of integers, two rationals may be the same but not use the same integers if one is in lowest terms and the other isn't. The atomic type clause considers this by allowing trie types to be defined in an *ad hoc* fashion for such types (where in contrast the generic portion is in a sense parametric). Similarly it allows definition of

tries for (some) types that don't fit this pattern, or that are more efficient than the default, such as for hardware integers.

5.1.2 Multi-parameter type classes with functional dependencies

In my undergraduate dissertation I was unable to get Generic Haskell to work, so I instead implemented memo tries using relatively 'ordinary' multi-parameter type classes with functional dependencies. This is a relational approach to declaring associated types, borrowed from the theory of relational databases: A class *MemIso* with two parameters is interpreted as a relation on types a and f, and a functional dependency, written $a \to f$, asserts that this relation is functional, i.e. for each a there is at most one f. This then allows type inference, when it finds an instance involving a, to choose the f given by this instance unambiguously. In the case of *MemIso*, the relation is that the functor is (extensionally) naturally isomorphic to the codomain functor $(\to)\ a$:[1]

```
data MIso a f z
  = MIso { trie   :: (a -> z) -> f z
         , untrie :: f z -> (a -> z) }
class MemIso a f | a -> f where
  memIso :: Iso (a -> z) (f z)
```

Then primitive type constructors are defined for each of the type operations (unit, sum, product) and associated combinators for their isomorphisms (whose definitions are elided here for space):

```
type One   = ()
data x :+ y = Inl x | Inr y
type x :* y = (x, y)
newtype OneT   z = OneT   { fromOne   :: z }
data (f :+: g) z = PlusT  { outl      :: f z
                          , outr      :: g z }
newtype (f :*: g) z = TimesT { fromTimes :: f (g z) }
oneI   :: MIso One OneT a
plusI  :: MIso a f z -> MIso b g z -> MIso (a :+ b) (f :+: g) z
timesI :: MIso a f (g z) -> MIso b g z -> MIso (a :* b) (f :*: g) z
instance MemIso One OneT where
  memIso = oneI
instance (MemIso a f, MemIso b g) => MemIso (a :+ b) (f :+: g) where
  memIso = memIso `plusI` memIso
instance (MemIso a f, MemIso b g) => MemIso (a :* b) (f :*: g) where
  memIso = memIso `timesI` memIso
```

The last two instances look ambiguous, but the type (and hence implementation) of *memIso* as used on the right hand sides of the equations can be inferred automatically in all four cases, due to the functional dependency.

[1] Of course, the isomorphism constraint *trie* . *untrie* = *id* is not expressible in Haskell, so it is left as an informal proof obligation for the programmer. In the *Data.MemoTrie* library GHC is told to rewrite *trie* (*untrie* x) $\to x$ during optimisation (if rewrite rules are turned on), and apart from anything else, the constraint is required for this rewriting to be sound.

Currently, multi-parameter type classes with functional dependencies are the best-established method of declaring associated types; they have been implemented in major Haskell implementations since about 2001 (having been proposed by Mark Jones in 2000 [Jon00]).

5.1.3 Indexed type families

As I was completing my dissertation in 2007, another approach to associated types called *indexed type families* [Cha+05][CKP05][Sch+08] was beginning to be implemented in the Glasgow Haskell Compiler (GHC). Here an associated type is declared as part of a class, in the same way as class methods (although they can be declared outside classes). Support was unstable and experimental at this stage, so I opted not to use them, but they have since become more stable and usable. In July 2008, Conal Elliott uploaded to Hackage [Hac] (the official repository of Haskell packages) the module `Data.MemoTrie`. This defines the following class:

```
class HasTrie a where
  data (:->:) a :: * -> *
  trie   :: (a ->  b) -> (a :->: b)
  untrie :: (a :->: b) -> (a ->  b)
```

Infix type operators such as (:→:) are themselves an extension, not available in ordinary Haskell; they make 'function-like' types such as this a lot easier to read (otherwise we would have something like *Trie a b*)). In the library are defined instances for standard algebraic types: `()` (unit), binary tuples (i.e. products) and `Either` (binary sums / disjoint unions):

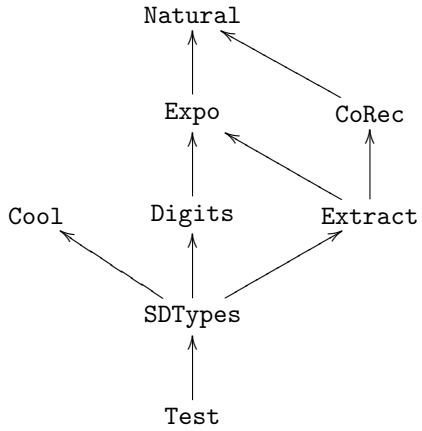
```
instance HasTrie () where
  data () :->: a = UnitTrie a
  trie f = UnitTrie (f ())
  untrie (UnitTrie a) = λ() -> a

instance (HasTrie a, HasTrie b) => HasTrie (Either a b) where
  data (Either a b) :->: x = EitherTrie (a :->: x) (b :->: x)
  trie f                   = EitherTrie (trie (f . Left)) (trie (f . Right))
  untrie (EitherTrie s t)  = either (untrie s) (untrie t)

instance (HasTrie a, HasTrie b) => HasTrie (a, b) where
  data (a, b) :->: x    = PairTrie (a :->: (b :->: x))
  trie f                = PairTrie (trie (trie . curry f))
  untrie (PairTrie t) = uncurry (untrie . untrie t)
```

This leads to much clearer type signatures: with functional dependencies one must choose an arbitrary type variable to represent the trie functor, and it may be unclear which variable is the functor for which domain type; whereas with type families the relationship is made explicit 'inline' by the functional notation. The declaration is also more compact, with the trie types and their methods grouped together. Examples illustrating this are given in the next section.

5.2 The code

When I started this project, my supervisor had been preparing a paper entitled "From coinductive proofs to exact real arithmetic". This was a very large paper, containing a lot of ideas, definitions, lemmas and proofs, all in one document. (His published paper by this title, [Ber09a], includes only some portions of the draft paper.) He had implemented a prototype of the ideas in the draft in an *ad hoc* fashion, by manually extracting programs from the definitions and proofs in his draft paper using realisability (see Chapter 3).

As I received it, Berger's code was in two files, `sdtest.hs` and `Cool.hs`. The first thing I did was to split the former into logically related chunks. The current module hierarchy looks like this (arrows indicate dependencies):



These modules are:

- `Natural`: Oddly, there is no data type of natural numbers in the standard library. This provides one. (Previously a type synonym for *Integer* was used.)
- `Expo`: This is a front-end to the `Deta.MemoTrie` module, with the addition of a *HasTrie* instance for *Natural*.
- `CoRec`: This contains the classes *Fix* (type constructors that have a fixpoint), *Ind* (recursion and strong recursion operators for least fixpoints), and *Coind* (corecursion and strong corecursion operators for greatest fixpoints), and instances for them. It also contains *wfrec*, the recursion (fixpoint) operator extracted from a proof of the wellfounded induction principle.
- `Digits` defines signed digits and their memo tries.
- `Extract` contains the inductive-coinductive data types extracted from the predicates C_n, and memo tries for them. These predicates are separate from the digits because they are in fact independent of the digit system used. In this module, in the functions *compC* and *timesC* scoped type variables make it possible to put type signatures on their subfunctions; these signatures are additionally made much clearer with the use of associated types instead of functional dependencies.

- `SDTypes` exports a type of signed digit streams and functions on them. Functions are defined in terms of open balls with rational centres and radii, which are subjected to the function *cool* (from `Cool.hs`) to eliminate excess precision (improving space efficiency). Scoped type variables are again used to make type signatures possible.

- `Test` contains a number of example programs. Here I attempted to do some optimisation by introducing some strictness with bang patterns.

5.2.1 `CoRec`

In `CoRec` are defined classes for fixpoints of type operators and induction and coinduction combinators for them:

```
class Fix op where
  type Fixpoint op
  fixin  :: op (Fixpoint op) → Fixpoint op
  outfix :: Fixpoint op → op (Fixpoint op)
class (Functor op, Fix op) ⇒ Ind op where
  it  :: (op a                  → a) → Fixpoint op → a
  rec :: (op (a, Fixpoint op) → a) → Fixpoint op → a
class (Functor op, Fix op) ⇒ Coind op where
  coit  :: (a → op a)                          → a → Fixpoint op
  corec :: (a → op (Either a (Fixpoint op))) → a → Fixpoint op
```

(*Ind* and *Coind* also have implementations of their methods, omitted here.) *it* and *rec* are 'induction' and 'complete induction', *coit* and *corec* 'corecursion' and 'complete coinduction'. There are instances for $((,)\ a)$, and variations on it (finite and infinite lists), as well as $((\to)\ a)$, which is used to express progressiveness of a function. 'Wellfounded' induction is defined both in terms of progressiveness and as just a general recursion operator; the proof of progressiveness is not actually used in the implementation of recursion, so the two operators are equivalent if they are used only in programs extracted from correctness proofs.

Note that in Haskell, there is only one fixpoint for a given operator. It can serve simultaneously as both the least fixpoint (and have an induction operator), and the greatest fixpoint (and have a coinduction operator).

5.2.2 `Extract`

In [Ber09a], the following predicates define coinductively uniformly continuous functions on $\mathbb{I}^n$:

$$\begin{aligned}
\mathcal{K} &: \mathcal{P}(\mathbb{R}^{\mathbb{I}^n}) \to \mathcal{P}(\mathbb{R}^{\mathbb{I}^n}) \to \mathcal{P}(\mathbb{R}^{\mathbb{I}^n}) \\
\mathcal{K}(X)(Y) &= \{f \mid \exists d(f[\mathbb{I}^n] \subseteq \mathbb{I}_d \wedge X(\mathrm{va}_d \circ f)) \vee \exists i \forall d Y(f \circ \mathrm{av}_{i,d})\} \\
\mathcal{J}_n &: \mathcal{P}(\mathbb{R}^{\mathbb{I}^n}) \to \mathcal{P}(\mathbb{R}^{\mathbb{I}^n}) \\
\mathcal{J}_n(X) &= \mu Y.\mathcal{K}_n(X)(Y) \\
C_n &\in \mathcal{P}(\mathbb{R}^{\mathbb{I}^n}) \\
C_n &= \nu X.\mathcal{J}_n(X)
\end{aligned}$$

Here $\mathrm{av}_d, \mathrm{va}_d : \mathbb{R} \to \mathbb{R}$ are the functions defining the semantics of digits $d \in D$, for some set of digits D; $\mathrm{av}_{i,d}$ applies av_d to the ith element of a list. D along with va_d and av_d form a digit system. For $D = SD := \{-1, 0, 1\}$ they are defined as follows:

$$\begin{aligned} \mathrm{va}_d(x) &= 2x - d \\ \mathrm{av}_d(x) &= \frac{x+d}{2} \\ \mathrm{av}_{i,d}(x_1, \ldots, x_n) &= \langle x_1, \ldots, \mathrm{av}_d(x_i), \ldots, x_n \rangle \end{aligned}$$

In this way a set of uniformly continuous functions is defined independently of the digit system used to represent them.

Remember that $d :\to: b$ (where d is an instance of the class *HasTrie*) is the type of d-indexed memo tries storing values of type b, representing functions $d \to b$. d is the type of digits, representing a digit system. This translates, via realisability, directly into Haskell as:

```
data Jop d e a b = W e a | R (d :->: b)
data J   d e a   = Jin {outJ :: Jop d e a (J d e a)}
data C   d e     = Cin {outC :: J d e (C d e)}
```

(Although e is not used in J and C, it needs to be mentioned to allow for composing functions. Without it, composite types would be ambiguous.) Originally the relationship between a digit type and the associated type of memo tries was expressed via functional dependencies, so that the trie types had to be added as extra parameters to these types:

```
data Jop d f e g a b = W e a | R (f b) | Dummy (g b)
data J   d f e g a   = Jin {outJ :: Jop d f e g a (J d f e g a)}
data C   d f e g     = Cin {outC :: J d f e g (C d f e g)}
```

The *Dummy* constructor is necessary to force the correct kind for g (although another extension, kind signatures, could eliminate the need for it). Expression with associated types makes it clearer that f and g are not used directly in J or C.

In *SDTypes* we set $d = SD$, where

```
data SD = N | Z | P deriving (Enum, Read, Show)
instance HasTrie SD where ...
```

so $C\ ()\ SD$ represents C_0, and $C\ SD\ SD$ represents C_1.

Chapter 6

Conclusion and further work

In this thesis I summarised the theory behind real number computation, including notions of computability (in terms of Type-2 Turing machines) and how they relate to both computability on finite data and continuity. I additionally defined a tentative notion of 'primitive corecursion' for functions on infinite data in terms of type-2 Turing machines, inspired by a similar notion for functions on finite data. I also explained the relationship between logical propositions and proofs and the functional programming concepts of type and program, and the use of modified realisability to find programs that realise propositions about real numbers. Finally, with the theory in mind, I examined Berger's Haskell code, which was extracted by hand using realisability from proofs of propositions in real analysis, to be clearer, better structured and more clearly reflect the propositions that it realises. In the process I discovered a connection between the realisability approach and direct implementations of number functions via functions on finite approximations of reals.

Real number computation is a deep subject, and this thesis suggests many further lines of research. On the theoretical level, my definition of 'primitive corecursive' functions bears further investigation. Several questions need to be asked: Is it non-trivial and interesting? Is there a hierarchy of primitive corecursive functions, in a similar way to the Grzegorczyk hierarchy of primitive recursive functions? Are the real number programs we have extracted so far primitive corecursive? And given the machine-oriented nature of the definition, is there a more natural characterisation of primitive corecursive functions at the proof level? If so, is it the one given by Leivant and Ramyaa [LR]?

The code could stand to be improved in numerous ways. Firstly, the programs are very slow, and use a lot of memory. It remains to be seen what techniques can be applied to real number programs to reduce the amount of memory allocation they do, not just to improve space complexity (such as introducing optimal strictness to remove any space leaks caused by excess laziness) but also to remove unnecessary overheads (such as using deforestation to eliminate redundant allocations).

Second, the specific technique of memoization used is inefficient. It leaves partially evaluated but potentially very large data structures sitting around

in memory for as long as they are needed; in many cases it is preferable to deallocate them to be re-evaluated later, or write them to temporary storage to free up physical memory. The very generic framework could also be specialised. For continuous functions f, for any input x that extends a given finite prefix w, the result $f(x)$ always extends the finite approximation $\tilde{f}(w)$; this fact could be harnessed by using a data structure that stores finite versions of intermediate results at given prefixes, instead of duplicating this data many times in streams that are identical in their first n elements. This may involve a 'little-endian' representation of finite portions of the streams, rather than the direct 'big-endian' representation of whole streams that is currently used.

Third, since the trie types have been removed as parameters to the type constructor C, the latter has the appropriate kind to be given an instance of the *Category* class, and composition is defined (as the function *compC*), so it should be possible to define an instance of *Category* (and possibly its subclass *Arrow*), which would allow it to be used in certain generic programs. This is made difficult by the *HasTrie* constraint on *compC* — Haskell does not allow extra class constraints on class methods — but it is possible in theory.

Finally, the ultimate goal of the project in the long term should be to produce a usable library for real number computation. This involves designing a programming interface and packaging the code in a standard package format (e.g. using Cabal [Cab]).

Programs currently have to be extracted via the realisability process by hand. Another intriguing option would be to write proofs in an automatic proof checker, and extract the programs automatically. For example, Coq [Coq] allows theorems to be defined using a dependently typed programming language including a type of non-computational propositions, and provides a mechanism for extraction of programs in various languages (including Haskell) from proofs, justified by the realisability interpretation.

Bibliography

[Abr87] Samson Abramsky. "Domain theory and the logic of observable properties". PhD thesis. Queen Mary College, University of London, 1987.

[Agd] *The Agda Wiki*. Chalmers University. 2010. URL: `http://wiki.portal.chalmers.se/agda/pmwiki.php` (visited on 31 March 2010).

[AHS90] Jiří Adámek, Horst Herrlich and George E. Strecker. *Abstract and concrete categories (The Joy of Cats)*. Wiley, 1990.

[AJ94] Samson Abramsky and Achim Jung. "Domain Theory". In: *Handbook of Logic in Computer Science*. Clarendon Press, 1994, pages 1–168.

[Alt01] Thorsten Altenkirch. "Representations of first order function types as terminal coalgebras". In: *Proceedings of the 5th International Conference on Typed Lambda Calculus and Applications*. Krakow, Poland. Edited by Samson Abramsky. Volume 2044. Lecture Notes in Computer Science. Berlin: Springer, 2001, pages 8–21. ISBN: 978-3-540-41960-0. URL: `http://citeseer.ist.psu.edu/471510.html`.

[Ber09a] Ulrich Berger. "From coinductive proofs to exact real arithmetic". In: *Proc. CSL 2009*. Edited by Erich Grädel and Reinhard Kahle. Volume 5771. Lecture Notes in Computer Science. Springer, 2009, pages 132–146. ISBN: 978-3-642-04026-9.

[Ber09b] Ulrich Berger. "Realisability and adequacy for (co)induction". In: *Proceedings of the Sixth International Conference on Computability and Complexity in Analysis, CCA 2009*. To appear, 2009.

[BH96] Vasco Brattka and Peter Hertling. "Feasible real random access machines". In: *SOFSEM'96: Theory and Practice of Informatics*. Edited by Keith Jeffery, Jaroslav Král and Miroslav Bartošek. Volume 1175. Lecture Notes in Computer Science. 10.1007/BFb0037415. Springer Berlin / Heidelberg, 1996, pages 335–342. URL: `http://dx.doi.org/10.1007/BFb0037415`.

[BS10] Ulrich Berger and Monika Seisenberger. "Proofs, programs, processes". To appear: CiE 2010, LNCS, 13 pages. 2010.

[Cab] *The Haskell Cabal*. haskell.org. 2010. URL: `http://www.haskell.org/cabal/` (visited on 31 March 2010).

[Cha+05] Manuel M. T. Chakravarty et al. “Associated types with class”. In: *POPL*. Edited by Jens Palsberg and Martín Abadi. ACM, 2005, pages 1–13. ISBN: 1-58113-830-X.

[CKP05] Manuel M. T. Chakravarty, Gabriele Keller and Simon L. Peyton Jones. “Associated type synonyms”. In: *Proc. ICFP 2005*. Edited by Olivier Danvy and Benjamin C. Pierce. ACM, 2005, pages 241–253. ISBN: 1-59593-064-7.

[Coq] *The Coq Proof Assistant*. ADT Coq. 2010. URL: `http://www.lix.polytechnique.fr/coq/` (visited on 31 March 2010).

[Cur34] Haskell B. Curry. “Functionality in Combinatory Logic.” In: *Proceedings of the National Academy of Sciences of the United States of America*. Volume 20. 11. 1934, pages 584–590. URL: `http://view.ncbi.nlm.nih.gov/pubmed/16577644`.

[EH02a] A. Edalat and R. Heckmann. “Computing with Real Numbers: I. The LFT Approach to Real Number Computation; II. A Domain Framework for Computational Geometry”. In: *Applied Semantics - Lecture Notes from the International Summer School, Caminha, Portugal*. Edited by G. Barthe et al. Springer-Verlag, 2002, pages 193–267.

[EH02b] Abbas Edalat and Reinhold Heckmann. “Computing with Real Numbers”. In: *Applied Semantics*. Edited by Gilles Barthe et al. Volume 2395. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2002, pages 953–984. URL: `http://dx.doi.org/10.1007/3-540-45699-6_5`.

[Hac] *HackageDB*. URL: `http://hackage.haskell.org/packages/hackage.html` (visited on 16 August 2010).

[Hin00a] Ralf Hinze. “Generalizing generalized tries”. In: *Journal of Functional Programming* 10.4 (2000).

[Hin00b] Ralf Hinze. “Memo functions, polytypically!” In: *Proceedings of the 2nd Workshop on Generic Programming*. Edited by Johan Jeuring. Ponte de Lima, Portugal 2000.

[How80] William A. Howard. “The formulae-as-types notion of construction”. In: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Edited by J. R. Hindley and J. P. Seldin. Academic Press, 1980, pages 479–490.

[HPG09] P. Hancock, D. Pattinson and N. Ghani. “Representations of Stream Processors using nested fixed points”. In: *Logical Methods in Computer Science* 5.3 (2009). URL: `http://www.doc.ic.ac.uk/~dirk/Publications/lmcs2009.pdf`.

[Jon00] Mark P. Jones. “Type classes with functional dependencies”. In: *Programming Languages and Systems*. Edited by Gert Smolka. Volume 1782. Lecture Notes in Computer Science. Springer, 2000, pages 230–244. ISBN: 3-540-67262-1.

[Kle45] S. C. Kleene. “On the interpretation of intuitionistic number theory”. In: *Journal of Symbolic Logic* 10.4 (December 1945). DOI: `10.2307/2269016`.

[Kre59] G. Kreisel. “Interpretation of analysis by means of constructive functionals of finite types”. Edited by A. Heyting. In: *Constructivity in Mathematics* (1959), pages 101–128.

[Lam07] Branimir Lambov. “RealLib: An efficient implementation of exact real arithmetic”. In: *Mathematical Structures in Computer Science* 17.01 (2007), pages 81–98. DOI: 10.1017/S0960129506005822. eprint: http://journals.cambridge.org/article_S0960129506005822. URL: http://journals.cambridge.org/action/displayAbstract?fromPage=online&aid=851588&fulltextType=RA&fileId=S0960129506005822.

[LR] Daniel Leivant and Ramyaa Ramyaa. “Implicit complexity for coinductive data: a proof-theoretic characterization of primitive corecursion”. Conference submission. Current draft, 2010. URL: http://www.cs.indiana.edu/~leivant/corecursion.pdf.

[ML80] Per Martin-Löf. *Intuitionistic Type Theory*. Notes by Giovanni Sambin on a series of lectures given in Padova. 1980. URL: http://www.cs.cmu.edu/~crary/819-f09/Martin-Lof80.pdf.

[MRE07] J. R. Marcial-Romero and M. H. Escardo. “Semantics of a sequential language for exact real-number computation”. In: *TCS* 379.1-2 (2007), pages 120–141.

[Mül01] Norbert Müller. “The iRRAM: Exact Arithmetic in C++”. In: *Computability and Complexity in Analysis*. Edited by Jens Blanck, Vasco Brattka and Peter Hertling. Volume 2064. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2001, pages 222–252. URL: http://dx.doi.org/10.1007/3-540-45335-0_14.

[O’C08] Russell O’Connor. “Certified Exact Transcendental Real Number Computation in Coq”. In: *Proc. TPHOLs 2008*. Edited by Otmane Mohamed, César Muñoz and Sofiène Tahar. Volume 5170. Lecture Notes in Computer Science. Springer, 2008. ISBN: 978-3-540-71065-3.

[Pip08] Dan Piponi. “How many functions are there from () to ()?” In: *A Neighbourhood of Infinity* (08 February 2008). URL: http://blog.sigfpe.com/2008/02/how-many-functions-are-there-from-to.html (visited on 18 November 2009).

[Sch+08] Tom Schrijvers et al. “Type checking with open type functions”. In: *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*. Edited by James Hook and Peter Thiemann. ACM, 2008, pages 51–62. ISBN: 978-1-59593-919-7.

[Tur36] Alan M. Turing. “On computable numbers, with an application to the Entscheidungsproblem”. In: *Proceedings of the London Mathematical Society* 2.42 (1936), pages 230–265.

[Vic89] Stephen Vickers. *Topology via logic*. Cambridge Tracts in Theoretical Computer Science 5. Cambridge University Press, 1989. ISBN: 0-521-57651-2.

[Wei00] Klaus Weihrauch. *Computable analysis: an introduction*. Texts in Theoretical Computer Science. Berlin: Springer, 2000. ISBN: 3-540-66817-9.

Appendix A

Code

This is a modified version of code originally written by Ulrich Berger.

A.0.3 Natural.hs

```
module Natural
    ( Natural
    , mkNatural
    ) where

newtype Natural = Natural { fromNatural :: Integer }
    deriving (Eq, Ord)
badNat = error "negative natural"
mkNatural :: Integer -> Natural
mkNatural n | n < 0     = badNat
            | otherwise = Natural n

instance Show Natural where
    show (Natural n) = show n -- makes sense thanks to Num instance

-- Unfortunately Num only allows for signed numbers - some of its methods don't
-- make sense for naturals.
instance Num Natural where
    Natural x + Natural y = Natural (x + y)
    Natural x * Natural y = Natural (x * y)
    Natural x - Natural y
        = if y >= x then Natural 0 else Natural (x - y) -- checked subtraction
    negate      _ = error "Num is annoying"
    abs         _ = error "Num is annoying"
    signum      _ = error "Num is annoying"
    fromInteger x
        | x >= 0    = Natural x
        | otherwise = error "Num is annoying"

instance Real Natural where
    toRational (Natural x) | x < 0     = badNat
```

```
                          | otherwise = toRational x

instance Enum Natural where
    succ (Natural n) | n < 0     = badNat
                     | otherwise = Natural (succ n)
    pred (Natural n) | n < 0     = badNat
                     | n == 0    = error "bad argument"
                     | otherwise = Natural (pred n)
    toEnum n     | n < 0     = badNat
                 | otherwise = Natural (fromIntegral n)
    fromEnum (Natural n) = fromInteger n

instance Integral Natural where
    Natural x `quotRem` Natural y
        | x < 0 || y < 0 = badNat
        | otherwise = (Natural q, Natural r) where (q,r) = x `quotRem` y
    Natural x `divMod`  Natural y
        | x < 0 || y < 0 = badNat
        | otherwise = (Natural d, Natural m) where (d,m) = x `quotRem` y
    toInteger (Natural x) = x
```

A.0.4 Expo.hs

```
{-# LANGUAGE TypeFamilies #-}
module Expo
    ( module Data.MemoTrie
    , module Natural
    ) where

import Data.MemoTrie
import Natural
import Data.List (genericIndex)

instance HasTrie Natural where
    data Natural :->: a = NaturalTrie [a]
    trie f = NaturalTrie (map f [0..])
    untrie (NaturalTrie l) n = l `genericIndex` n
    -- I don't see how this is useful, but it shuts up warnings
    enumerate (NaturalTrie xs) = zip [0..] xs
```

A.0.5 CoRec.hs

```
{-# LANGUAGE GeneralizedNewtypeDeriving, TypeFamilies #-}
module CoRec
    ( Fix (..)
    , Ind (..)
    , Coind (..)
    , Cart (..)
    , Cart1 (..)
    , wfrec, wfrecIt
```

```
    , Count
    -- examples - comment in for testing
    --, fibs, qsort, qsortIt
    ) where

import Control.Arrow ((&&&), (|||))

import Natural

-- General monotone (co)(iteration/recursion):

class Fix op where
   type Fixpoint op -- there is generally only one in Haskell,
                    -- but not in the underlying theory
   fixin  :: op (Fixpoint op) -> Fixpoint op
   outfix :: Fixpoint op -> op (Fixpoint op)

class (Functor op, Fix op) => Ind op where
   it :: (op a -> a) -> Fixpoint op -> a
   it step = step . fmap (it step) . outfix
   rec :: (op (a, Fixpoint op) -> a) -> Fixpoint op -> a
   rec step = step . fmap (rec step &&& id) . outfix

class (Functor op, Fix op) => Coind op where
   coit :: (a -> op a) -> a -> Fixpoint op
   coit step = fixin . fmap (coit step) . step
   corec :: (a -> op (Either a (Fixpoint op))) -> a -> Fixpoint op
   corec step = fixin . fmap (corec step ||| id) . step


-- Example: infinite lists

instance Fix ((,) a) where
  type Fixpoint ((,) a) = [a]
  fixin (x,xs) = x:xs
  outfix (x:xs) = (x,xs)

instance Coind ((,) a)

fibs :: [Natural]
fibs = coit costep (1,1)  where
        costep :: (Natural,Natural) -> (Natural , (Natural,Natural))
        costep (n,m) = (n, (m,n+m))
{-
Main> take 15 fibs
[1,1,2,3,5,8,13,21,34,55,89,144,233,377,610]
-}

newtype Cart a b = CartIn {outCart :: (a,b)}
    deriving (Functor)
```

```
instance Fix (Cart a) where
  type Fixpoint (Cart a) = [a]
  fixin (CartIn (x,xs)) = x:xs
  outfix (x:xs) = CartIn (x,xs)

instance Coind (Cart a)

{-
fibs :: [Nat]
fibs = coit costep (1,1)  where
        costep :: (Nat,Nat) -> Cart Nat (Nat,Nat)
        costep (n,m) = CartIn (n,(m,n+m))
-}


-- Example: finite and infinite lists

data Cart1 a b = NC | CC a b

instance Functor (Cart1 a) where
   fmap f NC       = NC
   fmap f (CC x y) = CC x (f y)

instance Fix (Cart1 a) where
  type Fixpoint (Cart1 a) = [a]
  fixin NC        = []
  fixin (CC x xs) = x:xs
  outfix []       = NC
  outfix (x:xs)   = (CC x xs)

instance Ind (Cart1 a)       -- finite list
instance Coind (Cart1 a)     -- finite and infinite lists

sumList :: [Int] -> Int
sumList = it step  where
  step :: Cart1 Int Int -> Int
  step NC       = 0
  step (CC h p) = h + p
{-
Main> sumList [1,2,3,4,5,6,7,8,9,10]
55
-}

-- Wellfounded recursion
wfrec :: (a -> (a -> b) -> b) -> a -> b
      -- (forall n. (forall m < n. A(m)) -> A(n)) -> forall n. A(n)
wfrec prog = h  where  h x = prog x h

-- Note that this is plain (general) recursion!
```

```
-- Only to be used if prog is extracted from the
-- proof of progressiveness of a pedicate w.r.t.
-- a wellfounded relation!

-- wfrec is obtained by optimising the following program
-- wfrecIT extracted from the inductive proof of
-- wellfounded induction:

data Count a = CountIn {outCount :: (a -> Count a)}

instance Fix ((->) a) where
  type Fixpoint ((->) a) = Count a
  fixin = CountIn
  outfix = outCount

instance Ind ((->) a)

wfrecIt :: (a -> Count a) -> (a -> (a -> b) -> b) -> a -> b
wfrecIt wfproof prog x = it step (wfproof x) x  where

-- step :: (a -> a -> b) -> a -> b
   step w p = prog p (\q -> w q q)

-- The optimisation is possible because the first
-- argument, wfproof, of wfrecIt isn't actually used.
-- It just serves as a witness for termination.
-- Since we know termination from outside
-- (because the relation we recur on is wellfounded)
-- it can be omitted.

-- Example: Quicksort
-- (wellfounded induction on the length of finite lists)

qProg :: Ord a => [a] -> ([a] -> [a]) -> [a]
qProg xs ih = case xs of
               []     -> []
               (x:ys) -> let low = [y | y <- ys, y <= x]
                             high = [y | y <- ys, y > x]
                         in ih low ++ [x] ++ ih high

qsort :: Ord a => [a] -> [a]
qsort = wfrec qProg

qsortIt :: Ord a => [a] -> [a]
qsortIt = wfrecIt (\_ -> error "Efq error") qProg
```

A.0.6 Digits.hs

```
{-# LANGUAGE TypeFamilies #-}
module Digits
```

```
    ( SD (..)
    , fromSD, signumSD
    , av, va
    , module Data.MemoTrie
    ) where

import Expo
import Data.MemoTrie -- already from Expo, but needs to be explicit

-- Signed digits
data SD = N | Z | P deriving (Enum, Read, Show)

instance HasTrie SD where
  data SD :->: a = TripleIn { outTriple :: (a,a,a) }
  trie g = TripleIn (g N, g Z, g P)
  untrie (TripleIn (x,y,z)) = \d -> case d of
                                N  -> x
                                Z  -> y
                                P  -> z
  -- I don't see how this is useful, but it shuts up warnings
  enumerate (TripleIn (n,z,p)) = [(N,n),(Z,z),(P,p)]

-- Without memoizations it's dramatically slower:
{-
newtype Triple a = TripleIn {outTriple :: SD -> a}

instance Fun SD Triple where
  abst = TripleIn
  appl = outTriple
-}

signumSD :: (Num a, Ord a) => a -> a -> SD
signumSD eps x | x < -eps  = N
               | x > eps   = P
               | otherwise = Z


fromSD :: Num a => SD -> a
fromSD N = -1
fromSD Z = 0
fromSD P = 1

toSD :: (Num a,Ord a) => a -> SD
toSD = signumSD 0

-- The interpretation of signed digits

av :: SD -> Rational -> Rational
av d p = (fromSD d + p)/2
```

```
va :: SD -> Rational -> Rational
va d p = 2*p - fromSD d
```

A.0.7 Extract.hs

```
{-# LANGUAGE TypeFamilies, TypeOperators, ScopedTypeVariables #-}
module Extract
    ( Jop (..)
    , C (..)
    , J (..)
    , compC
    , compCH
    , idC
    , module CoRec
    ) where

import CoRec
import Expo

-- Extracted programs

-- Inductive and coinductive data types

data Jop d e a b = W e a | R (d :->: b)

-- Inductive
data J d e a = Jin { outJ :: Jop d e a (J d e a) }


-- Coinductive
data C d e = Cin { outC :: J d e (C d e) }


-- Iteration and recursion for J

instance HasTrie d => Functor (Jop d e a) where
  fmap _ (W e c) = W e c
  fmap g (R as)  = R (fmap g as)

instance HasTrie d => Fix (Jop d e a) where
  type Fixpoint (Jop d e a) = J d e a
  fixin  = Jin
  outfix = outJ

instance HasTrie d => Ind (Jop d e a)


-- Coiteration and corecursion for C
```

```
instance HasTrie d => Functor (J d e) where
  fmap f =
   let mapJopW g (W e a)  = W e (g a)
       mapJopW _ (R cs)   = R cs
       myfixin = fixin
   in it (myfixin . mapJopW f)

instance HasTrie d => Fix (J d e) where
  type Fixpoint (J d e) = C d e
  fixin  = Cin
  outfix = outC

instance HasTrie d => Coind (J d e)


-- Lemma 16 a (identity)

idC :: HasTrie d => C d d
idC = coit (\_ -> fixin (R (trie (\d -> fixin (W d ())))) ()


-- Lemma 16 b (digits)

-- dC :: HasTrie d => d -> C d d
-- dC = corec (\d -> fixin (W d (Right idC)))

dC :: HasTrie d => d -> C d d
dC d = fixin (fixin (W d idC))


-- Lemma 18 (composition) (compare with Hancock/Pattinson/Ghani)

compC :: forall d1 d2 d3. (HasTrie d1, HasTrie d2) =>
            C d2 d3 -> C d1 d2 -> C d1 d3
compC c23 c12 = coit costep (c12, c23)  where

--  Type abreviations: Cij := C di fi dj fj
--                     Jij a := J di fi dj fj a
--                     Jopij a b := Jop di fi dj fj a b

-- costep :: (C12, C23) -> J13 (C12, C23)
   costep :: (C d1 d2, C d2 d3)
               -> J d1 d3 (C d1 d2, C d2 d3)
   costep (c12, c23) = aux (outC c23) c12

-- aux :: J23 C23 -> C12 -> J13 (C12, C23)
   aux :: J d2 d3 (C d2 d3) -> C d1 d2
            -> J d1 d3 (C d1 d2, C d2 d3)
   aux = it step
```

```
-- step :: Jop 23 C23 (C12 -> J13 (C12, C23)) -> C12 -> J13 (C12, C23)
   step :: Jop d2 d3 (C d2 d3)
                     (C d1 d2
                         -> J d1 d3 (C d1 d2, C d2 d3))
             -> C d1 d2 -> J d1 d3 (C d1 d2, C d2 d3)
   step (W d3 c23) c12 = fixin (W d3 (c12, c23))
   step (R fs)     c12 = subaux (outfix c12)  where

--       subaux :: J12 C12 -> J13 (C12, C23)
         subaux :: J d1 d2 (C d1 d2)
                     -> J d1 d3 (C d1 d2, C d2 d3)
         subaux = it substep

--       substep :: Jop12 C12 (J13 (C12, C23)) -> J13 (C12, C23)
         substep :: Jop d1 d2 (C d1 d2)
                              (J d1 d3 (C d1 d2, C d2 d3))
                     -> J d1 d3 (C d1 d2, C d2 d3)
         substep (W d2 c12) = untrie fs d2 c12
         substep (R j1223s) = fixin (R j1223s)

compCH :: (HasTrie d1, HasTrie d2) =>  -- honest version
             C d2 d3 -> C d1 d2 -> C d1 d3
compCH c23 c12 = coit costep (c12, c23)  where
   costep (c12, c23) = aux (outC c23) c12
   aux = it step
   step (W d3 c23) c12 = fixin (W d3 (c12, c23))
   step (R fs)     c12 = fixin (R (trie (\_ -> subaux (outfix c12))))  where
         subaux = it substep
         substep (W d2 c12) = untrie fs d2 c12
         substep (R j1223s) = fixin (R j1223s)



-- Lemma 19 a (terminal object)

oneC :: (HasTrie d) => C d ()
oneC = coit (fixin . W ()) ()


-- Lemma 19 b i (projections)

pr1C :: (HasTrie d, HasTrie e) => C (d, e) d
pr1C = coit (\_ ->
   fixin (R (trie (\de -> fixin (W (fst de) ())))))
   ()

pr2C :: (HasTrie d, HasTrie e) => C (d, e) e
pr2C = coit (\_ ->
   fixin (R (trie (\de -> fixin (W (snd de) ())))))
   ()
```

```
-- Lemma 19 b ii (pairing)

timesC :: forall d1 d2 e. (HasTrie d1, HasTrie d2, HasTrie e) =>
           C e d1 -> C e d2 -> C e (d1, d2)
timesC = curry (coit costep) where

--  Type abreviations: Ci := C e g di fi
--                     Jopi a b := Jop e g di fi a b
--                     J12 a := J e g (d1, d2) (CompF f1 f2) a

-- costep :: (C1, C2) -> J12 (C1, C2)
   costep :: (C e d1, C e d2)
               -> J e (d1, d2) (C e d1, C e d2)
   costep (c1, c2) = it step (outfix c1) c2

-- step :: Jop1 C1 (C2 -> J12 (C1, C2)) -> C2 -> J12 (C1, C2)
   step :: Jop e d1 (C e d1)
               (C e d2
                -> J e (d1, d2) (C e d1, C e d2))
            -> C e d2
            -> J e (d1, d2) (C e d1, C e d2)
   step (W d1 c1) c2 = it substepW (outfix c2) d1 c1
   step (R h)   c2 = it substepR (outfix c2) h

-- substepW :: Jop2 C2 (d1 -> C1 -> J12 (C1, C2)) -> d1 -> C1 -> J12 (C1, C2)
   substepW :: Jop e d2 (C e d2)
                    (d1
                     -> C e d1
                     -> J e (d1, d2) (C e d1, C e d2))
               -> d1
               -> C e d1
               -> J e (d1, d2) (C e d1, C e d2)
   substepW jop d1 c1 =
    fixin (case jop of
            W d2 c2 -> W (d1, d2) (c1, c2)
            R f1s   -> R (trie (\e -> untrie f1s e d1 (compC c1 (dC e)))))

-- substepR :: Jop2 C2 ((e :->: (C2 -> J12 (C1, C2))) -> J12 (C1, C2))
--                      -> (e :->: (C2 -> J12 (C1, C2))) -> J12 (C1, C2)
   substepR :: Jop e d2 (C e d2)
                   (e :->: (C e d2 -> J e (d1, d2) (C e d1, C e d2))
                    -> J e (d1, d2) (C e d1, C e d2))
               -> (e :->: (C e d2 -> J e (d1, d2) (C e d1, C e d2)))
               -> J e (d1, d2) (C e d1, C e d2)
   substepR jop h =
    fixin (R (trie (\e ->
                     case jop of
                      W d2 c2  -> untrie h e (dC d2 `compC` c2 `compC` dC e)
```

```
                    R fes    -> untrie fes e h)))
```

A.0.8 SDTypes.hs

```
{-# LANGUAGE
    ScopedTypeVariables
  , TypeOperators
  , MultiParamTypeClasses
  , TypeSynonymInstances
  #-}
module SDTypes
    ( CSD
    , II
    , UCII
    , C0
    , UC
    , qC
    , iterC
    , cRatM
    , uCtoCSD
    , ucSDS
    , defaultPrecision
    , wfC
    , piecewisemonotoneUC
    , uClasstoC
    , compUC
    , idUC
    , cool
    , module Digits
    , module Extract
    ) where

import Cool

import Digits
import Extract

-- This should be imported via Extract, but mysteriously isn't.
import Natural


-- pi4M 25

-- fromRational (defint (lmaC 2) 0.01)
-- fromRational (defint (lmaC 1.5) 0.01)
-- fromRational (defint (lmaC 1) 0.01)
-- fromRational (defint (lmaC 0.1) 0.0005)
```

```
-- Some hacks and verbose code are forced
-- by Hakell's inadequate type inference.
-- I would prefer a programming language
-- where, if desired, type inferenc can be switched
-- off and replaced by eplicit typing (in F omega).

-- Programs extracted from
-- "From coinductive proofs to exact real arithmetic"
-- Section: Digit Systems



-- C_D

type C0 d = C () d
type J0 d a = J () d a
type Jop0 d a b = Jop () d a b

-- Default number of digits to use
defaultPrecision :: Int
defaultPrecision = 60


instance Show d => Show (C0 d)  where
   show = take defaultPrecision . concat . (showC0' "-")      -- "-" means "wait"

showC0' :: forall d. Show d => String -> C0 d -> [String]
showC0' waitstring = coit costep  where

   costep :: C0 d -> (String, C0 d)
   costep = it step . outfix

   step :: Jop0 d (C0 d) (String, C0 d) -> (String, C0 d)
   step (W d c) = (show d, c)
   step (R h)   = let (str, c) = (untrie h ())
                              in (waitstring ++ str, c)
{-
-- costep :: C0 d -> Cart String (C0 d)
   costep = it step . outfix

-- step :: Jop0 (C0 d) (Cart String (C0 d)) -> Cart String (C0 d)
   step (W d c) = CartIn (show d, c)
   step (R h)   = let (str, c) = outCart (untrie h ())
                  in CartIn (waitstring ++ str, c)
-}


-- Lemma 20
```

```
{-
(X, D) is a digit system.
C_D [i.e. C0 d] is the largest subset A of X such that
    if x in A, then
        there exist x in D and x' in A
        such that x = d(x')
i.e. C_D is "closed under the inverses of the digits D".
 -}
toDigitStream :: C0 d -> [d]
toDigitStream = coit costep where

-- costep :: C0 d -> (d, C0 d)
   costep = it step . outfix

-- step :: Jop0 (C0 d) (Cart d, C0 d) -> (Cart d, C0 d)
   step (W d c) = (d, c)
   step (R h)   = untrie h ()

fromDigitStream :: forall d. [d] -> C0 d
fromDigitStream = coit costep where
   costep :: [d] -> J0 d [d]
   costep ds = fixin (W (head ds) (tail ds))


runC :: (HasTrie d, HasTrie e) => C d e -> [d] -> C0 e
runC c = compC c . fromDigitStream

runCH :: (HasTrie d, HasTrie e) => C d e -> [d] -> C0 e
runCH c = compCH c . fromDigitStream

runCS :: (HasTrie d, HasTrie e) => C d e -> [d] -> [e]
runCS c = toDigitStream . compC c . fromDigitStream


-- Lemma 25

wfC :: forall d e a. (HasTrie d, HasTrie e) =>
         (a -> Either (e, a) (d :->: a)) ->
         a -> C d e
wfC s = coit (wfrec prog)  where
   prog :: a -> (a -> J d e a) -> J d e a
   prog x ih =
     case s x of
      Left (e, y) -> fixin (W e y)
      Right ys   -> fixin (R (fmap ih ys))


-- Continuous functions w.r.t. signed digit reals

type CSD   = C SD SD
```

```
type JSD a = J SD SD a -- unused
type JopSD a b = Jop SD SD a b


sRealM :: Natural -> [SD] -> Float
sRealM m = aux m where
   aux 0 _          = 0
   aux (n+1) (d:ds) = (fromSD d + aux n ds)/2

sRatM :: Int -> [SD] -> Rational
sRatM m ds = foldr (.) id (map av (take m ds)) 0

sReal = sRealM 12

cRealM :: Natural -> C0 SD -> Float
cRealM m = sRealM m . toDigitStream

cRatM :: Int -> C0 SD -> Rational
cRatM m = sRatM m . toDigitStream

cReal = cRealM 12


-- Iterating a map

iterC :: HasTrie d => C d d -> Int -> C d d
iterC c n = iterate (compC c) idC !! n

iterCR :: HasTrie d => C d d -> Int -> C d d
iterCR c n = iterate (c `compC`) idC !! n

powiterC :: HasTrie d => C d d -> Int -> C d d
powiterC c n = iterate (\c -> compC c c) c !! n

-- cReal (runC (iterC lmC 3) (period [N, Z]))


-------------------------
-- Metric digit spaces

-- Uniform continuity

-- f m-cont. :<=> all delta, p ex eps in m(delta), q:
--                      f[B_delta(p)]\tm B_eps(q)

-- f u.c. :<=> f m-cont. for some modulus m

-- m modulus  :<=>
--   all b>0 ex a>0 all delta<=a all eps in m(delta) eps<=b
```

```
type Ball p = (Rational, p)

type UC p q = Ball p -> Ball q


-- Bounded metric spaces
class BMS p where
 bMS :: (Rational, p)
-- if bMS = (delta0, p0) then all x sigma(p0, x)<=delta0

-- Uniformly contracting metric digit spaces
class UContrMDS p d where
 ucontrMDS :: d -> UC p p
-- There must exist lambda < 1 such that
-- for every d the lambda-contractivity of d
-- is witnessed by ucontrMDS d

-- Uniformly covering metric digit spaces
class UCovMDS p d where
 ucovMDS :: Ball p -> Maybe d
-- there must exist a0>0 s.t.
-- all (delta, p): if ucovMDS(delta, p) = Just d  then B_delta(p) subseteq d[X];
--                 if ucovMDS(delta, p) = Nothing then delta > a0

-- Uniformly invertible metric digit spaces
class UInvMDS p d where
 uinvMDS :: d -> UC p p
-- if uinvMDS delta0 p0 d p = (q, h) then
-- all eps F_{h eps, eps}(d^{-1})

type II = Rational  -- = [-1, 1] intersect Rational


leftBall, rightBall :: Ball II -> II
leftBall (delta, p) = max (-1) (p - delta)
rightBall (delta, p) = min 1 (p + delta)

inBall :: Ball II -> II -> Bool
inBall ball q = leftBall ball <= q && q <= rightBall ball

truncateBall :: Ball II -> Ball II
truncateBall ball = (eps, q) where
    left  = leftBall ball
    right = rightBall ball
    eps   = (right-left)/2
    q     = (right+left)/2


-- subBall :: Ball II -> Ball II -> Bool
-- subBall (delta, p) (epsilon, q) = abs (p-q) + delta <= epsilon
```

```
subBall :: Ball II -> Ball II -> Bool
subBall ball1 ball2 = leftBall ball2 <= leftBall ball1 &&
                      rightBall ball1 <= rightBall ball2

ballSD :: SD -> Ball II
ballSD d = truncateBall (1/2, fromSD d)

-- the smallest ball containing a given list of points
hullBall :: [II] -> Ball II
hullBall ps = ((right-left)/2, (right+left)/2)
   where
     left  = max (-1) (minimum (1 : ps))
     right = min 1 (maximum ((-1) : ps))

instance BMS II where
  bMS = (1, 0)

type UCII = UC II II

sdUC :: SD -> UCII
sdUC d (delta, p) = truncateBall (delta/2, av d p)

instance UContrMDS II SD where
  ucontrMDS = sdUC
-- all 1/2 contracting

instance UCovMDS II SD where
  ucovMDS ball =
    case [ d | d <- [Z, P, N], subBall ball (ballSD d)] of
      (d:_) -> Just d
      _     -> Nothing
-- a0 = 1/4

instance UInvMDS II SD where
  uinvMDS d (delta, p) = truncateBall (2*delta, va d p)


-- Effective uniform continuity for u.c. piecewise monotone functions

piecewisemonotoneUC :: (II -> II) -> [II] -> UCII
piecewisemonotoneUC f peaks ball = hullBall (map f potentialExtrema)
   where
     potentialExtrema = filter (inBall ball) peaks
                        ++ [leftBall ball, rightBall ball]

-- Lemma 29

compUC :: Cool r => UC q r -> UC p q -> UC p r
compUC f g b = cool 3 (f (g b))
```

```
-- Lemma 31 (a)

{-
uCtoC :: (HasTrie d, BMS p, UContrMDS p d,
          HasTrie e, UInvMDS q e, UCovMDS q e)
          => UC p q  -> C d e
uCtoC = wfC s  where
   s uc = case ucovMDS (uc bMS) of
            Just e    -> Left (e, compUC (uinvMDS e) uc)
            Nothing   -> Right (trie (compUC uc . ucontrMDS))
-}

-- For family U of u.c. functions f closed under f . d and e^{-1} . f

uClasstoC :: (HasTrie d, BMS p, HasTrie e, UCovMDS q e) =>
               (u -> UC p q) ->
               (u -> d -> u) ->
               (u -> e -> u) ->
                           u -> C d e
uClasstoC uc r w = wfC s  where
   s u = case ucovMDS (uc u bMS) of
            Just e    -> Left (e, w u e)
            Nothing   -> Right (trie (r u))

-- old as instance of new
uCtoC :: (HasTrie d, BMS p, UContrMDS p d,
          HasTrie e, UInvMDS q e, UCovMDS q e,
          Cool q)
          => UC p q  -> C d e
uCtoC = uClasstoC id r w where
      r u d = compUC u (ucontrMDS d)
      w u e = compUC (uinvMDS e) u


qC :: Rational -> C0 SD
qC = coit costep where

  costep :: II -> J0 SD II
  costep q = let e = signumSD (1/4) q
             in fixin (W e (va e q))

uCtoCSD :: UCII -> CSD
uCtoCSD = uCtoC

runUC :: UCII -> [SD] -> C0 SD
runUC = runC . uCtoCSD

runUCS :: UCII -> [SD] -> [SD]
runUCS = runCS . uCtoCSD
```

```
avUC :: SD -> UCII
avUC d (delta, p) = truncateBall (delta/2, av d p)

-- Corollary 33

-- This doesn't type check for the usual
-- silly reasons (p does not occur after =>)

{-
changeDigits :: (BMS p, HasTrie d, HasTrie e,
                 UContrMDS p d, UInvMDS p e, UCovMDS p e)
                 => C0 d -> C0 e
changeDigits = compC (uCtoC (\eps -> (eps, id)))
-}

-- Version with p = II doesn't work either:
{-
cDII :: (HasTrie d, HasTrie e,
         UContrMDS II d, UInvMDS II e, UCovMDS II e)
                 => C0 d -> C0 e
cDII = compC (uCtoC (\eps -> (eps, id)))
-}

-- So, need to resign to monomorphism:

type PiDigits = Natural

piDig :: PiDigits -> II -> II
piDig n p = (1/2) + (np*p)/(2*np+1)  where
   np = fromIntegral n



instance UContrMDS II PiDigits where
  ucontrMDS n = piecewisemonotoneUC (piDig n) []


uCtoCpi = uCtoC :: UC II II -> C PiDigits SD

idUC :: UCII
idUC = id


cDpi4 :: C0 PiDigits -> C0 SD
cDpi4 = compC (uCtoCpi idUC)

pi4C0SD :: C0 SD
pi4C0SD = cDpi4 (fromDigitStream [1, 2..])
```

```
pi4M :: Natural -> Float
pi4M m = cRealM m pi4COSD

-- pi4M 25

pi4S ::[SD]
pi4S = toDigitStream pi4COSD

-- take 35 pi4S


-- Definite integral (Proposition 10)

defint :: CSD -> Rational -> Rational
defint c eps = wfrec prog eps c where

   prog :: Rational -> (Rational -> CSD -> Rational) -> CSD -> Rational
   prog eps ih = if eps >= 2
                 then \_-> 0
                 else let step :: JopSD CSD Rational -> Rational
                          step (W e c) = ih (2*eps) c / 2 + fromSD e
                          step (R ps)  = (untrie ps N + untrie ps P)/2
                      in it step . outfix


-- Continuity of identity:

ucId :: UC II II
ucId = id

ucSDS :: [SD] -> UC II II
ucSDS = foldr compUC ucId . map sdUC
```

A.0.9 Cool.hs

```
-- 20-10-08
-- Cooling down rational numbers (avoiding useless precision)

{-# LANGUAGE TypeSynonymInstances #-}
module Cool (Cool (..)) where

-- Given rationals delta,p, where delta>0,  define the ball
-- B(delta,p) := {x in QQ : |x - p| <= delta}

-- For every nonnegative integer k we map
--  (delta,p)  to  cool k (delta,p) = (delta',p')
-- such that

-- (1) B(delta,p) subset B(delta',p')
-- (2) bitsize (delta',p') <= k + 2 * lg (1/delta)
```

```
-- (3) delta' <= (1 + 2^(-k))*delta

-- where lg x = least integer n such that |x| <= 2^n

lg :: (RealFrac a, Integral b) => a -> b
lg x = (fromIntegral .
        length .
        takeWhile (\n -> fromIntegral (2^n) < abs x))
       [0,1..]

-- ceil q = least integer >= q cast into the rationals
ceil :: RealFrac a => a -> a
ceil q = fromInteger (ceiling q)

class Cool p where
    cool :: Integral a => a -> (Rational, p) -> (Rational, p)
instance Cool Rational where
    cool k (delta,q) = (ceil m / n , p / n)
       where  n  = fromIntegral (2^(k + (lg (1/delta))))
              p  = fromIntegral (round (n * q))
              m  = maximum [abs (p - n * (q + d)) | d <- [-delta,delta]]

test :: Integer -> (Rational,Rational) -> IO ()
test k (delta,q) =
   let (delta',q')   = cool k(delta,q)
       (deltaF,qF)   = (fromRational delta,fromRational q)
       (deltaF',qF') = (fromRational delta',fromRational q')
   in do putStrLn (-- show (delta,q) ++ "  " ++
                   show (deltaF,qF) ++ "  " ++
                   show (qF-deltaF,qF+deltaF))
         putStrLn (show (delta',q') ++ "  " ++
                   show (deltaF',qF') ++ "  " ++
                   show (qF'-deltaF',qF'+deltaF'))
         putStrLn (show deltaF' ++ "  " ++
                   show ((1 + 2^^(- fromIntegral k)) * deltaF))
```

A.0.10 Test.hs

```
{-# LANGUAGE TypeOperators, BangPatterns #-}
module Test where
-- 17-9-08

import System.Environment (getArgs, withArgs)
import System.IO (stdout, BufferMode (..), hSetBuffering)

import SDTypes
import UntilM


-- Example: linear affine maps f_(u, v)(x) = u*x+v
```

```
type Rat2 = (Rational, Rational)

linC :: Rat2 -> CSD
linC = wfC s  where
   s :: Rat2 -> Either (SD, Rat2) (SD :->: Rat2)
   s (u, v) = if u <= 1/4
                    then let w = if abs v <= 1/4 then 0 else signum v
                             e = signumSD 0 w
                         in Left (e, (2*u, 2*v-w))
                    else Right (trie (\d -> (u/2, u*fromSD d/2+v)))

-- runC (linC (1/15, 1/3)) (period [P, Z])


-- Example: quadratic maps f_(u, v, w)(x) = u*x^2+v*x+w

type Rat3 = (Rational, Rational, Rational)

quadC :: Rat3 -> CSD
quadC = wfC s where

   s :: Rat3 -> Either (SD, Rat3) (SD :->: Rat3)
   s !uvw = case (filter (quadTest uvw) [N, Z, P]) of
                      (e:_)  -> Left (e, quadWrite uvw e)
                      []     -> Right (trie (quadRead uvw))

   quadWrite :: Rat3 -> SD -> Rat3
   quadWrite (u, v, w) e = (2*u, 2*v, 2*w - e')
     where e' = fromSD e


   quadRead :: Rat3 -> SD -> Rat3
   quadRead (u, v, w) d = (u/4, (u*d'+v)/2, u*d'^2/4 + v*d'/2 + w)
     where d' = fromSD d

   quadTest :: Rat3 -> SD -> Bool
   quadTest (u, v, w) e = (e'-1)/2 <= low && high <= (e'+1)/2
     where
       e'        = fromSD e
       low       = minimum criticals                -- max (f_(u, v, w) I)
       high      = maximum criticals                -- min (f_(u, v, w) I)
       criticals = [ u+v+w                          -- f_(u, v, w) 1
                   , u-v+w                          -- f_(u, v, w) (-1)
                   ] ++
                   if u == 0
                    then []
                    else let x = -v/(2*u)           -- extremal point
                         in if -1 <= x && x <= 1
                               then [ u*x^2 + v*x + w ] -- f_(u, v, w) x
```

```
                                else []


-- Some logistic maps

lma :: (Num a) => a -> a -> a
lma a x = a*(1-x^2)-1          -- 0 <= a <= 2
--      = -a*x^2 + a - 1

lmaC :: Rational -> CSD
lmaC a = quadC (-a, 0, a-1)

-- new: unfortunately much slower
-- lmaC :: Rational -> CSD
-- lmaC a = quC (-a, 0, a-1)

lm12C, lm23C, lm34C, lm2C :: CSD
lm12C = lmaC (1/2)
lm23C = lmaC (2/3)
lm34C = lmaC (3/4)
lm2C = lmaC 2

lman :: Int -> (II -> II)
lman n = (iterate (. lma 2) id) !! n

-- Ball examples

-- Example: quadratic functions, absolute value

quUC :: Rat3 -> UCII
quUC (u, v, w) = piecewisemonotoneUC (\p -> u*p^2+v*p+w)
                                     (if u == 0 then [] else [-v/(2*u)])

ucabs :: UC II II
ucabs = piecewisemonotoneUC abs [0]

-- Example: quadratic functions:

quC :: Rat3 -> CSD
-- quC = uCtoC . quUC
quC = uClasstoC quUC quadRead quadWrite

 where

   quadRead :: Rat3 -> SD -> Rat3
   quadRead (u, v, w) d = (u/4, (u*d'+v)/2, u*d'^2/4 + v*d'/2 + w)
     where d' = fromSD d

   quadWrite :: Rat3 -> SD -> Rat3
```

```
   quadWrite (u, v, w) e = (2*u, 2*v, 2*w - e')
     where e' = fromSD e

lmanUC :: Int -> UCII
lmanUC n = (iterate (compUC (quUC (-2, 0, 1))) idUC) !! n

lmacoolnUC :: Int -> Int -> UCII
lmacoolnUC k n = (iterate (compUC (quUC (-2, 0, 1) . cool k)) idUC) !! n



-- NB: to test in ghci, use 'withArgs'.
main :: IO ()
main = titCnoninteractive lm2C (lma 2) (1/3) . map read =<< getArgs

-- Demo programs:

-- If fC is a tree representation of f:II->II
-- then testitC fC f = f^n p where p and n are
-- given interactively.

testitC :: CSD -> (II -> II) -> IO ()
testitC fC f =
    let getInput = do
            putStr "\nINPUT:"
            s <- getLine `catch` const (return "")
            return $ if null s
                        then Nothing
                        else Just (read s :: II)
    in  preUntilM_ getInput $ titC fC f

-- If fC is a tree representation of f:II->II
-- and p is in II (a rational number)
-- then titC fC f p = f^n p where n is given:

-- ... from a list.
titCnoninteractive :: CSD -> (II -> II) -> II -> [Int] -> IO ()
titCnoninteractive fC f p ns =
    let c0 = qC p
    in do titCpreamble c0 p
          mapM_ (\n -> putStrLn ("ITERATIONS: " ++ show n) >> titCstep c0 fC f p n) ns

-- ... interactively.
titC :: CSD -> (II -> II) -> Rational -> IO ()
titC fC f p =
   do titCpreamble c0 p
      preUntilM_ getInput $ titCstep c0 fC f p
    where
        c0 = qC p
        getInput = do putStr "ITERATIONS: "
```

```
                    strn <- getLine `catch` const (putStrLn "" >> return "")
                    if null strn
                      then return Nothing
                      else return (Just (read strn :: Int))

titCpreamble :: C0 SD -> II -> IO ()
titCpreamble c0 p = putStrLn $ concat
    [ show p, "\t = ", show c0
    , "\n\t = ", show (fromRational p :: Double), "\n"]

titCstep :: C0 SD -> CSD -> (II -> II) -> II -> Int -> IO ()
titCstep c0 fC f p n =
    let cn = compC (iterC fC n) c0
        pn = iterate f p !! n
        fR = fromRational . f . toRational :: Double -> Double
        x0 = (fromRational p :: Double)
    in  putStrLn $ concat
          ["OUTPUT"
          ,"\nExact SD:    \t", show cn
          ,"\n... as Float:\t", show (fromRational
                                (cRatM defaultPrecision cn) :: Double)
          ,"\nFloat:       \t", show (iterate fR x0 !! n)
          ,if n <= 0
            then concat ["\nE. R. as Float:\t",
                                show (fromRational pn :: Double)
                        ,"\nExact Rat:  \t",
                                show pn, "\n"]
            else "\n"]

-- I hate programming
{-
(Float = double precision)

*Main> titC lm2C (lma 2) (1/3)
1%3      = PNPNPNPNPNPNPNPNPNPNPNPNPNPNPNPNPNPNPNPNPNPNPNPNPNPNPNPNPNPNPNPNPN
         = 0.33333334

ITERATIONS: 1
OUTPUT
Exact SD:       PPZZPZZNZZPZZNZZPZZNZZPZZNZZPZZNZZPZZNZZPZZNZZPZZNZZPZZNZZPZ
... as Float:   0.7777777777777778
Float:          0.7777777777777778
E. R. as Float: 0.7777777777777778
Exact Rat:      7%9

ITERATIONS: 2
OUTPUT
Exact SD:       ZNZZPPNZZPZZPNPNPZZPNZZZZZNZPZNPNPZNPZNPPNPNZNPPZZZZZPZNZZPP
... as Float:   -0.20987654320987653
Float:          -0.2098765432098766
```

```
E. R. as Float: -0.20987654320987653
Exact Rat:      (-17)%81

ITERATIONS: 3
OUTPUT
Exact SD:       PPPZPZPNPZNPZPNZPZZZPNZPZZNZPNPNPZPNPNPZNPNZZZPPNPZZZZNPZZZN
... as Float:   0.9119036732205457
Float:          0.9119036732205457
E. R. as Float: 0.9119036732205457
Exact Rat:      5983%6561

ITERATIONS: 4
OUTPUT
Exact SD:       NNPNPZNZPNZZZNZPNPNPZNPZZNZNPZZNZZNZPZZZPNPNZZNZZNPZNZNZPZZZ
... as Float:   -0.6631366184662474
Float:          -0.6631366184662475
E. R. as Float: -0.6631366184662474
Exact Rat:      (-28545857)%43046721

ITERATIONS: 5
OUTPUT
Exact SD:       ZZPZZZZNZNPNPZZPZZPNZZPNNPNPNZPZZZZPZZPZZNPNZPZNPZZPNZZNPZZN
... as Float:   0.12049965049830123
Float:          0.120499650498301
E. R. as Float: 0.12049965049830123
Exact Rat:      223288285122943%1853020188851841

ITERATIONS: 10
OUTPUT
Exact SD:       PPZZZZZZZZNZPZPNZPNZZNPPNPZNZNPZNPNZZZPZZNZZZZZZNPZZZPZZZZZZP
... as Float:   0.7493017528354341
Float:          0.7493017528354383
E. R. as Float: 0.7493017528354341
Exact Rat:      279783166756109595520329154953886074722899063385902177740551732553042997330746

ITERATIONS: 30
OUTPUT
Exact SD:       PZZZZZPNPZPNPPNPNPZZZZZNZPNZZZZZZPNPZPNPNPNPZZNPZNZPZNZPZZZN
... as Float:   0.5062674954994535
Float:          0.5062674897488822

ITERATIONS: 40
OUTPUT
Exact SD:       PPPPPPPPZNPNZPNZPNZZZNPZNPNPZZZPNPNZPPZZZZPNZZPNZZPZZZPZNPNP
... as Float:   0.9953955227211435
Float:          0.9953961772115644

ITERATIONS: 50
OUTPUT
Exact SD:       PZPZNPZZZPNZPNPZZPNZPPZZZNPNPZNPNZPZZNPPNZZZPNNPZZZPPZPZNPNZ
```

```
... as Float:   0.609957446954024
Float:          0.615483351212418

ITERATIONS: 60
OUTPUT
Exact SD:       NNNZPNPZZNZNPZPNZPNZZPZZNZZNPZZPNPZPNZNZPZZNZNZZNZZPNZNZZZZN
... as Float:   -0.8526437597407311
Float:          -0.9469915748606237

ITERATIONS: 100
OUTPUT
Exact SD:       NNNNNNNNNNPZPPZZZZZPZPZNZZNNPNPPNPZZNZPZZZNZZZZPNZNPZNPZZZZN
... as Float:   -0.9983509279028324
Float:          0.9934928198528841

ITERATIONS: 300
OUTPUT
Exact SD:       PPNPZZNPZZZPNZZZNZNZZZPNZNZPZZPZZZZZZZZZPNZZZZZNZPNPNPNZPNZZZ
... as Float:   0.68370633246422
Float:          0.16185293297236963

ITERATIONS: 400
OUTPUT
Exact SD:       ZZZNPZZPZZPZPNZZPNPZNZPPZNZPZPNPNPZZNPZZPZZZZPZZPNPZPNZZNPNP
... as Float:   -2.67890201098297e-2
Float:          -0.6706012733485962

ITERATIONS: 500
OUTPUT
Exact SD:       NZZZZZNZZNPZNPZZZNNPNPZPZZZNPNZPNPZZNPPNZZPZNPZNPZZZZZZZZZNZ
... as Float:   -0.5083667662137419
Float:          0.9909988369437114

ITERATIONS: 600
OUTPUT
Exact SD:       NNZZZNPZZNZZZZZNPNZPZZPNPPNPNZZNPZZZPNZZZNZPNPZNPZNZZZZNPZNZ
... as Float:   -0.7587994543102519
Float:          0.33218504745590244

-- 600 brings the computer almost down
-}


-- Demo program:
-- If f:II->II and uc is a mod of u.c. for f,
-- then testit uc f = f^n(p) where p and n are
-- given interactively.

testit :: UCII -> (II -> II) -> IO ()
testit = testitC . uCtoCSD
```

```
testitd :: [SD] -> IO ()
testitd ds = testit (ucSDS ds) (foldr (.) id (map av ds))
```