INVERTING MONOTONE CONTINUOUS FUNCTIONS IN CONSTRUCTIVE ANALYSIS

HELMUT SCHWICHTENBERG

Abstract. We prove constructively (in the style of Bishop) that every monotone continuous function with a uniform modulus of increase has a continuous inverse. The proof is formalized, and a realizing term extracted. This term can be applied to concrete continuous functions and arguments, and then normalized to a rational approximation of say a zero of a given function. It turns out that even in the logical term language "normalization by evaluation" is reasonably efficient.

1. Introduction

There have been many attempts to formalize constructive analysis as presented in Bishop's classic [6]. One reason to do this is to uncover the computational content of constructive proofs, an aspect that has been one of the motivations of the FTA project [8], where Kneser's constructive proof of the fundamental theorem of algebra was formalized in Coq. This work has recently been extended to build a "Constructive Coq Repository (C-CoRN)" at Nijmegen (Barendregt, Geuvers, Wiedijk, Cruz-Filipe [7]). However, extraction of reasonable program from proofs in this setup turned out to be problematic. One reason is that witnesses were missing from compuational meaningful axioms (e.g., strong extensionality $\forall_{x,y}.f(x)\#f(y) \to x\#y$), another one that the `Set`, `Prop` distinction in Coq was found to be insufficient (cf. [7]). Here we desribe a different formalization of constructive analysis, from the point of view of later term extraction. In particular, we deal with the existence of a continuous inverse to a monotonically increasing continuous function. The proof uses the Intermediate Value Theorem IVT.

Some optimizations in definitions and proofs are necessary to produce extracted terms that can be evaluated efficiently. These are (a) addition of external code to the definitions of arithmetical operations, which is used (based on the corresponding function of the programming language) when the arguments are numerals; (b) introduction of the let-construct in extracted terms; (c) "non-computational" quantifiers [3].

The paper extends [15] by a formalization of and term extraction from the theorem on the existence of inverse functions. It turns out that in spite of the harder theorem (compared with IVT) one obtains even better extracted terms. – I have tried to make this paper readable independently of [15].

2. Inverse Functions

We prove that every continuous function with a uniform modulus of increase has a continuous inverse. A constructive proof of this fact has been given by Mandelkern [13]. More recently, J. Berger [2] introduced a concept he called "exact representation of continuous functions", and based on this gave a construction converting one such representation of an increasing function into another one of its inverse. The proof below is based on a particular concept of a continuous function, as a type-1 object (using separability of the real numbers).

The setup of constructive analysis is essentially the one of Bishop [6], and is only sketched here. More detailed elaborations can be found in [1, 15].

We view a *real* x as a Cauchy sequence $(a_n)_n$ of rationals with a separately given modulus M. When comparing two reals, $x < y$ needs a witness, but $x \leq y$ doesn't; in fact, we can prove $x \not< y \leftrightarrow y \leq x$. For reals $x = ((a_n)_n, M)$ and $y := ((b_n)_n, N)$, define $x <_k y$ to mean $1/2^k \leq b_p - a_p$, for $p := \max(M(k+2), N(k+2))$.

Constructively we cannot compare two reals, but we can compare a real with a proper interval:

Lemma (ApproxSplit)**.** *Let x, y, z be given and assume $x < y$. Then either $z \leq y$ or $x \leq z$.*

Proof. Let $x := ((a_n)_n, M)$, $y := ((b_n)_n, N)$, $z := ((c_n)_n, L)$. Assume $x <_k y$, and let $q := \max(p, L(k+2))$ and $d := (b_p - a_p)/4$.

Case $c_q \leq \frac{a_p+b_p}{2}$. We show $z \leq y$. It suffices to prove $c_n \leq b_n$ for $n \geq q$. To see this, observe

$$c_n \leq c_q + \frac{1}{2^{k+2}} \leq \frac{a_p + b_p}{2} + \frac{b_p - a_p}{4} = b_p - \frac{b_p - a_p}{4} \leq b_p - \frac{1}{2^{k+2}} \leq b_n$$

Case $c_q \not\leq \frac{a_p+b_p}{2}$. We show $x \leq z$, via $a_n \leq c_n$ for $n \geq q$.

$$a_n \leq a_p + \frac{1}{2^{k+2}} \leq a_p + \frac{b_p - a_p}{4} \leq \frac{a_p + b_p}{2} - \frac{b_p - a_p}{4} \leq c_q - \frac{1}{2^{k+2}} \leq c_n.$$

This concludes the proof. □

A *continuous function* $f \colon I \to \mathbb{R}$ on a compact interval I with rational end points is given by

(a) an approximating map $h_f \colon (I \cap \mathbb{Q}) \times \mathbb{N} \to \mathbb{Q}$ and a map $\alpha_f \colon \mathbb{N} \to \mathbb{N}$ such that $(h_f(a,n))_n$ is a Cauchy sequence with (uniform) modulus α_f;
(b) a modulus $\omega_f \colon \mathbb{N} \to \mathbb{N}$ of (uniform) continuity, which satisfies

$$|a - b| \leq 2^{-\omega_f(k)+1} \to |h_f(a,n) - h_f(b,n)| \leq 2^{-k} \quad \text{for } n \geq \alpha_f(k);$$

α_f and ω_f are required to be weakly increasing. One may also add a lower bound N_f and an upper bound M_f for all $h_f(a,n)$.

Notice that a continuous function is given by objects of type level ≤ 1 only. This is due to the fact that it suffices to define its values on rational numbers.

To prove the Intermediate Value Theorem, we begin with an auxiliary lemma, which from a "correct" interval $c < d$ (that is, $f(c) \leq 0 \leq f(d)$ and $2^{-n} \leq d - c$) constructs a new one $c_1 < d_1$ with $d_1 - c_1 = \frac{2}{3}(d - c)$.

We say that $l \in \mathbb{N}$ is a *uniform modulus of increase* for $f \colon [a,b] \to \mathbb{R}$ if for all $c, d \in [a,b]$ and all $m \in \mathbb{N}$

$$2^{-m} \leq d - c \to f(c) <_{m+l} f(d).$$

Lemma (IVTAux)**.** *Let $f\colon [a,b] \to \mathbb{R}$ be continuous, and with a uniform modulus l of increase. Assume $a \le c < d \le b$, say $2^{-n} < d - c$, and $f(c) \le 0 \le f(d)$. Then we can construct c_1, d_1 with $d_1 - c_1 = \frac{2}{3}(d-c)$, such that again $a \le c \le c_1 < d_1 \le d \le b$ and $f(c_1) \le 0 \le f(d_1)$.*

Proof. Let $c_0 = c + \frac{d-c}{3} = \frac{2c+d}{3}$ and $d_0 = c + \frac{2(d-c)}{3} = \frac{c+2d}{3}$. From $2^{-n} < d-c$ we obtain $2^{-n-2} \le d_0 - c_0$, so $f(c_0) <_{n+2+l} f(d_0)$. Now compare 0 with this proper interval, using ApproxSplit. In the first case we have $0 \le f(d_0)$; then let $c_1 = c$ and $d_1 = d_0$. In the second case we have $f(c_0) \le 0$; then let $c_1 = c_0$ and $d_1 = d$. □

Theorem (IVT)**.** *If $f\colon [a,b] \to \mathbb{R}$ is continuous with $f(a) \le 0 \le f(b)$, and with a uniform modulus of increase, then we can find $x \in [a,b]$ such that $f(x) = 0$.*

Proof. Iterating the construction in the auxiliary lemma IVTAux above, we construct two sequences $(c_n)_n$ and $(d_n)_n$ of rationals such that for all n

$$\begin{aligned}&a = c_0 \le c_1 \le \dots \le c_n < d_n \le \dots \le d_1 \le d_0 = b,\\ &f(c_n) \le 0 \le f(d_n),\\ &d_n - c_n = \big(2/3\big)^n (b-a).\end{aligned}$$

Let x, y be given by the Cauchy sequences $(c_n)_n$ and $(d_n)_n$ with the obvious modulus. As f is continuous, $f(x) = 0 = f(y)$ for the real number $x = y$. □

From the Intermediate Value Theorem we obtain

Theorem (Inv)**.** *Let $f\colon [a,b] \to \mathbb{R}$ be continuous with a uniform modulus of increase, and assume $f(a) \le a' < b' \le f(b)$. We can find a continuous $g\colon [a',b'] \to \mathbb{R}$ such that $f(g(y)) = y$ for every $y \in [a',b']$ and $g(f(x)) = x$ for every $x \in [a,b]$ such that $a' \le f(x) \le b'$.*

Proof. Let $f\colon [a,b] \to \mathbb{R}$ be continuous with a uniform modulus of increase, that is, some $l \in \mathbb{N}$ such that for all $c, d \in [a,b]$ and all $m \in \mathbb{N}$

$$2^{-m} \le d - c \to f(c) <_{m+l} f(d).$$

Let $f(a) \le a' < b' \le f(b)$. We construct a continuous $g\colon [a',b'] \to \mathbb{R}$.

Let $u \in [a',b']$ be rational. Using $f(a) - u \le a' - u \le 0$ and $0 \le b' - u \le f(b) - u$, the IVT gives us an x such that $f(x) - u = 0$, as a Cauchy sequence (c_n). Let $h_g(u,n) := c_n$. Define the modulus α_g such that for $n \ge \alpha_g(k)$, $(2/3)^n(b-a) \le 2^{-\omega_f(k+l+2)}$. For the uniform modulus ω_g of continuity assume $a' \le u < v \le b'$ and $k \in \mathbb{N}$. We claim that with $\omega_g(k) := k + l + 2$ (l from the hypothesis on the slope) we can prove the required property

$$|u - v| \le 2^{-\omega_g(k)+1} \to |h_g(u,n) - h_g(v,n)| \le 2^{-k} \quad (n \ge \alpha_g(k)).$$

Let $a' \le u < v \le b'$ and $n \ge \alpha_g(k)$. For $c_n^{(u)} := h_g(u,n)$ and $c_n^{(v)} := h_g(v,n)$ assume that $|c_n^{(u)} - c_n^{(v)}| > 2^{-k}$; we must show $|u - v| > 2^{-\omega_g(k)+1}$.

By the proof of the Intermediate Value Theorem we have

$$d_n^{(u)} - c_n^{(u)} \le (2/3)^n(b-a) \le 2^{-\omega_f(k+l+2)} \quad \text{for } n \ge \alpha_g(k).$$

Using $f(c_n^{(u)}) - u \le 0 \le f(d_n^{(u)}) - u$, the fact that a continuous function f has ω_f as a modulus of uniform continuity gives us

$$|f(c_n^{(u)}) - u| \le |(f(d_n^{(u)}) - u) - (f(c_n^{(u)}) - u)| = |f(d_n^{(u)}) - f(c_n^{(u)})| \le 2^{-k-l-2}$$

and similarly $|f(c_n^{(v)}) - v| \le 2^{-k-l-2}$. Hence, using $|f(c_n^{(u)}) - f(c_n^{(v)})| \ge 2^{-k-l}$ (which follows from $|c_n^{(u)} - c_n^{(v)}| > 2^{-k}$ by the hypothesis on the slope),

$$|u - v| \ge |f(c_n^{(u)}) - f(c_n^{(v)})| - |f(c_n^{(u)}) - u| - |f(c_n^{(v)}) - v| \ge 2^{-k-l-1}.$$

Now $f(g(u)) = u$ follows from

$$|f(g(u)) - u| = |h_f(c_n, n) - u| \le |h_f(c_n, n) - h_f(c_n, m)| + |h_f(c_n, m) - u|,$$

which is $\le 2^{-k}$ for $n, m \ge \alpha_f(k+1)$. Since continuous functions are determined by their values on the rationals, we have $f(g(y)) = y$ for $y \in [a', b']$.

For every $x \in [a, b]$ with $a' \le f(x) \le b'$, from $g(f(x)) < x$ we obtain the contradiction $f(x) = f(g(f(x))) < f(x)$ by the hypothesis on the slope, and similarly for $>$. Using $u \not< v \leftrightarrow v \le u$ we obtain $g(f(x)) = x$. □

As an example, consider the squaring function $f\colon [1, 2] \to [1, 4]$, given by the approximating map $h_f(a, n) := a^2$, constant Cauchy modulus $\alpha_f(k) := 1$, and modulus $\omega_f(k) := k + 1$ of uniform continuity. The modulus of oncrease is $l := 0$, because for all $c, d \in [1, 2]$

$$2^{-m} \le d - c \to c^2 <_m d^2.$$

Then $h_g(u, n) := c_n^{(u)}$, as constructed in the IVT for $x^2 - u$, iterating IVTAux. The Cauchy modulus α_g is such that $(2/3)^n \le 2^{-k+3}$ for $n \ge \alpha_g(k)$, and the modulus of uniform continuity is $\omega_f(k) := k + 2$.

3. Formalization

We now aim at formalizing the proof above, with the planned extraction of realizing terms in mind. For this purpose it is clearly important to represent the underlying mathematical objects in an appropriate way.

It is tempting to start with groups, rings, fields etc. (as in [8, 7]). However, it turned out that in such a general approach it is hard to control the computational content of the proofs, and hence its extracted terms. This does not mean that an abstract approach is impossible for our task, but for the moment we prefer the more "concrete" setup, with explicit constructions of the objects.

- *Positive* natural numbers are written in binary; we take them as generated from 1 by two successors $n \mapsto 2n$ and $n \mapsto 2n + 1$. In the corresponding free algebra we have the constructors `One`, `SZero` and `SOne`.
- An *integer* is either a positive number, or zero, or a negative number.
- A *rational* is a pair of an integer and a positive, written `i#n`. Notice that equality of rationals is not the literal one, but given by the usual equivalence relation.
- A *real* is a pair of a Cauchy sequence of rationals and a modulus. We view the reals as a data type (i.e., no properties), with constructor `RealConstr as M`, whose components are written `x seq` and `x mod`.

Within this data type we inductively define the predicate `Real x`, meaning that `x` is a (proper) real.

- A *continuous function* is viewed as an element of a data type with constructor `ContConstr`, whose fields are written `f doml`, `f domr` (for the left and right end point of its domain), `f approx` (for the approximating function), `f uMod` (for the uniform Cauchy modulus) and `f uModCont` (for the modulus of uniform continuity). Within this data type we have an inductively defined predicate `Cont f`, meaning that `f` is a (proper) continuous function.

From this material we can now build typed lambda terms, as usual. They are terms in the sense of Gödel's T [9], that is, contain (structural) recursion operators for every data type (i.e., free algebra), with arbitrary value types. These terms are the basis of our logical (better: arithmetical) system, which contains an induction scheme (w.r.t. arbitrary formulas) for every data type.

The formalization itself is (tedious but) straightforward; proof scripts are available at `www.minlog-system.de`, in the directory `examples/analysis`.

4. Terms and their evaluation

4.1. **Computation rules.** Computable functionals are defined by "computation rules" [5, 4]; these rules are added to the standard conversion rules of typed λ-calculus. To simplify equational reasoning, terms with the same normal form are identified.

A *system of computation rules* for a defined constant D consists of finitely many equations $D\vec{P}_i = Q_i$ $(i = 1, \dots, n)$ with constructor patterns $\vec{P}_i$, such that $\vec{P}_i$ and $\vec{P}_j$ $(i \neq j)$ are non-unifiable. *Constructor patterns* are lists of applicative terms with distinct variables, defined inductively as follows (we write $\vec{P}(\vec{x}\,)$ to indicate all variables in $\vec{P}$; all expressions must be type-correct):

- $x(x)$ is a constructor pattern.
- If C is a constructor and $\vec{P}(\vec{x}\,)$ a constructor pattern, then $(C\vec{P})(\vec{x}\,)$ is a constructor pattern.
- If $\vec{P}(\vec{x}\,)$ and $Q(\vec{y}\,)$ are constructor patterns whose variables $\vec{x}$ and $\vec{y}$ are disjoint, then $(\vec{P}, Q)(\vec{x}, \vec{y}\,)$ is a constructor pattern.

One instance of such rules is the definition of the fixed point operator $\mathcal{Y}_\rho$ of type $(\rho \Rightarrow \rho) \Rightarrow \rho$, by $\mathcal{Y}_\rho f = f(\mathcal{Y}_\rho f)$, which clearly defines a partial functional. Another important example are the (Gödel) structural recursion operators.

However, in practice one wants to define computable functionals by recursion equations, and if possible consider total functionals only. This can be achieved if the patterns on the lhs are "complete" (as for the structural recusion operator) and moreover the rules terminate (as for Gödel's T [9]). Then every closed term of ground type reduces to a "numeral" (or a "canonical term"), that is, a term built from constructors only.

For example, addition for rational numbers is defined by the computation rule converting `(i1#k1)+(i2#k2)` into `i1*k2+i2*k1#k1*k2`.

4.2. **External code as part of arithmetical constants.** A problem when computing on rationals with the rule above is that the gcd is not cancelled out automatically. Therefore we add "external code" to the internal representation of the function. It works as follows: whenever addition for rationals is called with numerical arguments, these arguments are converted into Scheme rationals, then added with the rational addition function of Scheme, and the result is converted back into the internal representation (using the `#`-constructor) of a rational.

4.3. **Cleaning of Reals.** After some computations involving real numbers it is to be expected that the rational numbers occurring in the Cauchy sequences may become rather complex. Hence under computational aspects it is necessary to be able to *clean up* a real, as follows.

Lemma 4.1. *For every real $x = ((a_n)_n, M)$ we can construct an equivalent real $y = ((b_n)_n, N)$ where the rationals b_n are of the form $c_n/2^n$ with integers c_n, and with modulus $N(k) = k + 2$.*

Proof. Let $c_n := \lfloor a_{M(n)} \cdot 2^n \rfloor$ and $b_n := c_n \cdot 2^{-n}$, hence

$$\frac{c_n}{2^n} \leq a_{M(n)} < \frac{c_n}{2^n} + \frac{1}{2^n} \quad \text{with } c_n \in \mathbb{Z}.$$

Then for $m \leq n$

$$\begin{aligned} |b_m - b_n| &= |c_m \cdot 2^{-m} - c_n \cdot 2^{-n}| \\ &\leq |c_m \cdot 2^{-m} - a_{M(m)}| + |a_{M(m)} - a_{M(n)}| + |a_{M(n)} - c_n \cdot 2^{-n}| \\ &\leq 2^{-m} + 2^{-m} + 2^{-n} \\ &< 2^{-m+2}, \end{aligned}$$

hence $|b_m - b_n| \leq 2^{-k}$ for $n \geq m \geq k + 2 =: N(k)$, so $(b_n)_n$ is a Cauchy sequence with modulus N.

To prove that x is equivalent to $y := ((b_n)_n, N)$, observe

$$\begin{aligned} |a_n - b_n| &\leq |a_n - a_{M(n)}| + |a_{M(n)} - c_n \cdot 2^{-n}| \\ &\leq 2^{-k-1} + 2^{-n} \quad \text{for } n, M(n) \geq M(k+1) \\ &\leq 2^{-k} \quad \text{if in addition } n \geq k+1. \end{aligned}$$

Hence $|a_n - b_n| \leq 2^{-k}$ for $n \geq \max(k+1, M(k+1))$, and therefore $x = y$. □

5. Extracted terms

5.1. **Realizability.** We first describe some proof-theoretic background on term extraction, as it is implemented in the Minlog proof assistant (`www.minlog-system.de`). It is based on *modified realizability* as introduced by Kreisel [11]: from every constructive proof M (in natural deduction) of a formula A with computational content one extracts a term $[\![M]\!]$ "realizing" A. This term usually is much shorter than the proof it came from, because in the process all subproofs of formulas without computational content can be ignored. The extracted term has a type $\tau(A)$ which depends on the logical shape of the proven formula A only.

An important aspect of this "internal" term extraction (compared with say the extraction of OCaml programs in Coq [12]) is that one stays within the language of the logical theory, and hence – for a particular proof M

– can *prove* within the system that the extracted term indeed realizes the formula A (the "Soundness Theorem").

Of course, there is a good reason to extract programs rather than terms: running programs is much faster than evaluating (closed) terms. However, the point made in the previous paragraph is a strong argument for term extraction, particularly in safety critical applications. Moreover, as should become clear from what is done in the present paper, with some care one may well design proofs (and the underlying data types) in such a way that the extracted terms are short and easy to read and evaluate. One can then go on and (automatically) translate these terms into code of a functional programming language, for faster evaluation (cf. [15] for an example).

5.2. **Quantifiers without computational content.** Besides the usual quantifiers, $\forall$ and $\exists$, Minlog has so-called *non-computational quantifiers*, $\forall^{\mathsf{nc}}$ and $\exists^{\mathsf{nc}}$, which allow for the extraction of simpler terms. The nc-quantifiers, which were first introduced in [3], can be viewed as a refinement of the Set/Prop distinction in constructive type systems like Coq or Agda. Intuitively, a proof of $\forall^{\mathsf{nc}}_x A(x)$ ($A(x)$ non-Harrop, i.e., with a strictly positive occurrence of an existential quantifier) represents a procedure that assigns to every x a proof $M(x)$ of $A(x)$ where $M(x)$ does not make "computational use" of x, i.e., the extracted term $[\![M(x)]\!]$ does not depend on x. Dually, a proof of $\exists^{\mathsf{nc}}_x A(x)$ is a proof of $M(x)$ for some x where the witness x is "hidden", that is, not available for computational use. Consequently, the types of extracted terms for nc-quantifiers are $\tau(\forall^{\mathsf{nc}}_{x^\rho} A) = \tau(\exists^{\mathsf{nc}}_{x^\rho} A) = \tau(A)$ as opposed to $\tau(\forall_{x^\rho} A) = \rho \Rightarrow \tau(A)$ and $\tau(\exists_{x^\rho} A) = \rho \times \tau(A)$. The extraction rules are, for example in the case of $\forall^{\mathsf{nc}}$-introduction and -elimination, $[\![(\lambda x.M^{A(x)})^{\forall^{\mathsf{nc}}_x A(x)}]\!] = [\![M]\!]$ and $[\![(M^{\forall^{\mathsf{nc}}_x A(x)} t)^{A(t)}]\!] = [\![M]\!]$ as opposed to $[\![(\lambda x.M^{A(x)})^{\forall_x A(x)}]\!] = [\![\lambda x M]\!]$ and $[\![(M^{\forall_x A(x)} t)^{A(t)}]\!] = [\![Mt]\!]$. In order for the extracted terms to be correct the variable condition for $\forall^{\mathsf{nc}}$-introduction needs to be strengthened by requiring in addition the abstracted variable x not to occur in the extracted term $[\![M]\!]$. Note that for a Harrop formula A the formulas $\forall^{\mathsf{nc}}_x A$ and $\forall_x A$ are equivalent; similarly, $\exists^{\mathsf{nc}}_x A$ and $\exists_x A$ are equivalent.

5.3. **Animation.** Suppose a proof of a theorem uses a lemma. Then the proof term contains just the name of the lemma, say `L`. In the term extracted from this proof we want to preserve the structure of the original proof as much as possible, and hence we use a new constant `cL` at those places where the computational content of the lemma is needed. When we want to execute the program, we have to replace the constant `cL` corresponding to a lemma `L` by the extracted program of its proof. This can be achieved by adding computation rules for `cL` and `cGA`. We can be rather flexible here and enable/block rewriting by using `animate/deanimate` as desired.

5.4. **Removal of duplicated parts in terms.** In machine generated terms (e.g., those obtained by term extraction) it often happens that a subterm has many occurrences in a term, which leads to unwanted recomputations when evaluating it. A possible cure is to "optimize" the term after extraction, and replace for instance $M[x := N]$ with many occurrences of x in M by $(\lambda x M)N$ (or a corresponding "let"-expression). However, this can already be done at the proof level: When an object (value of a variable or realizer of a premise) might be used more than once, make sure (if necessary by a cut) that the goal has the form $A \to B$ or $\forall_x A$. Now use the "identity lemma" `Id`: $\hat{P} \to \hat{P}$, whose predicate variable $\hat{P}$ is then instantiated with $A \to B$ or $\forall_x A$; its realizer has the form $\lambda f, x.fx$. However, if `Id` is not animated, the extracted term has the form $\mathtt{cId}(\lambda x M)N$, which is printed as [`let` x N M].

5.5. **Extracted terms.** The term extracted from the proof of `ApproxSplit` is

```
(Rec real=>real=>real=>pos=>boole)
([as4,M5]
  (Rec real=>real=>pos=>boole)
  ([as9,M10]
    (Rec real=>pos=>boole)
    ([as13,M14,n15]
      as13(M5(S(S n15))max M10(S(S n15))max M14(S(S n15)))<=
      (as4(M5(S(S n15))max M10(S(S n15)))+
       as9(M5(S(S n15))max M10(S(S n15))))/2)))
```

of type `real=>real=>real=>pos=>boole`. It takes three reals x, y, z with moduli M, N, K (here given by their Cauchy sequences `as4`, `as9`, `as13` and moduli `M5`, `M10`, `M14`) and a positive number k (here `n15`), and computes $p := \max(M(k+2), N(k+2))$ and $q := \max(p, L(k+2))$. Then the choice whether to go right or left is by computing the boolean value $c_q \leq \frac{a_p+b_p}{2}$.

For the auxiliary lemma `IVTAux` we obtain the extracted term

```
[f0,n1,n2]
 (cId rat@@rat=>rat@@rat)
 ([cd4]
   [let cd5
     ((2#3)*left cd4+(1#3)*right cd4@
      (1#3)*left cd4+(2#3)*right cd4)
     [if (cApproxSplit(RealConstr(f0 approx left cd5)
                                 ([n6]f0 uMod(S(S n6))))
                      (RealConstr(f0 approx right cd5)
                                 ([n6]f0 uMod(S(S n6))))
          0
          (S(S(n2+n1))))
      (left cd4@right cd5)
      (left cd5@right cd4)]])
```

of type `cont=>pos=>pos=>rat@@rat=>rat@@rat`. As in the informal proof, it takes a continuous f (here `f0`), a uniform modulus l of increase (here `n1`), a positive number n (here `n2`) and two rationals c, d (here the pair `cd4`) such that $2^{-n} < d - c$. Let $c_0 := \frac{2c+d}{3}$ and $d_0 := \frac{c+2d}{3}$ (here the pair `cd5`, introduced via `let` because it is used four times). Then `ApproxSplit` is applied to $f(c_0)$, $f(d_0)$, 0 and the witness $n+2+l$ (here `S(S(n2+n1))`) for $f(c_0) < f(d_0)$. In the first case we go left, that is $c_1 := c$ and $d_1 := d_0$, and in the second case we go right, that is $c_1 := c_0$ and $d_1 := d$.

In the proof of the Intermediate Value Theorem, the construction step in `IVTAux` (from a pair c, d to the "better" pair c_0, d_0) had to be iterated, to produce two sequences $(c_n)_n$ and $(d_n)_n$ of rationals. This is the content of a separate lemma `IVTcds`, whose extracted term is

```
[f0,n1,n2](cDC rat@@rat)(f0 doml@f0 domr)
                        ([n4]cIVTAux f0 n1(n2+n4))
```

of type `cont=>pos=>pos=>pos=>rat@@rat`. It takes a continuous $f\colon [a, b] \to \mathbb{R}$ (here `f0`), a uniform modulus l of increase (here `n1`), and a positive number k_0 (here `n2`) such that $2^{-k_0} < b - a$. Then the axiom of dependent choice `DC` is used, to construct from an initial pair $(c_0, d_0) = (a, b)$ of rationals (here `f0 doml@f0 domr`) a sequence of pairs of rationals, by iterating the computational content `cIVTAux` of the lemma `IVTAux`.

The proof of the Inversion Theorem does not use the Intermediate Value theorem directly, but its essential ingredient `IVTcds`. Its extracted term is

```
[f0,n1,n2,n3,a4,a5]
 ContConstr a4 a5
 ([a6,n7]
   left((cACT rat pos=>rat@@rat)
        ([a8]
          (cIPT pos=>rat@@rat)
          ((cIPT pos=>rat@@rat)
           (cIVTcds
            (ContConstr f0 doml f0 domr
             ([a12,n13]f0 approx a12 n13-a8)
             f0 uMod
             f0 uModCont)
            n1
            n2)))
        a6
        n7))
 ([n6]n3+f0 uModCont(S(S(n6+n1))))
 ([n6]S(S(n6+n1)))
```

It takes a continuous function f (here `f0`), a uniform modulus l of increase (here `n1`), positive numbers k_0, k_1 (here `n2`, `n3`) such that $2^{-k_0-1} < b - a < 2^{k_1}$ and two rationals $a_1 < a_2$ (here `a4`, `a5`) in the range of f. Then the continuous inverse g is constructed (via `ContConstr`) from

- an approximating map,
- a uniform Cauchy modulus (involving the one from f), and
- an easy and explicit modulus of uniform continuity.

The approximating map takes a, u (here `a6`, `n7`). Ignoring the computational content `cACT`, `cIPT` of `ACT`, `IPT` (which are identities), it yields the left component (i.e., the Cauchy sequence) of the result of applying `cIVTcds` to a continuous function close to the original f.

To compute numerical approximations of values of an inverted function we need `RealApprox`, stating that every real can be approximated by a rational. Its extracted term is

```
(Rec real=>pos=>rat)([as2,M3,n4]as2(M3 n4))
```

of type `real=>pos=>rat`. It takes a real x (here given by the Cauchy sequence `as2` and modulus `M3`) and a positive number k (here `n4`), and computes a rational a such that $|x - a| \leq 2^{-k}$. Notice that the `Rec`-operator is somewhat trivial here: it just takes the given real apart. This is because the data type of the reals has no inductive constructor.

To compose `Inv` with `RealApprox`, we prove a proposition `InvApprox` stating that given an error bound, we can find a rational approximating the value of the inverted function g up to this bound. Clearly we need to refer to this value and hence the inverted function g in the statement of the theorem, but on the other hand we do not want to see a representation of g in the extracted term, but only the construction of the rational approximation from the error bound. Therefore in the statement of `InvApprox` we use the non-computational quantifier $\exists^{\mathsf{nc}}$ (see Section 5.2), for the inversion g of the given continuous f. The extracted term of `InvApprox` then simply is

```
[f0,n1,n2,n3,a4,a5,a6]
 cRealApprox
 (RealConstr((cInv f0 n1 n2 n3 a4 a5)approx a6)
  ([n8](cInv f0 n1 n2 n3 a4 a5)uMod(S(S n8))))
```

of type `cont=>pos=>pos=>pos=>rat=>rat=>rat=>pos=>rat`.

Now we "animate" the auxiliary lemmas, that is, add computation rules for all constants with "c" in front of name of the lemma. For `InvApprox` this gives

```
[f0,n1,n2,n3,a4,a5,a6,n7]
 left((cDC rat@@rat)(f0 doml@f0 domr)
      ([n8]
        (cId rat@@rat=>rat@@rat)
        ([cd10]
          [let cd11
            ((2#3)*left cd10+(1#3)*right cd10@
            (1#3)*left cd10+(2#3)*right cd10)
            [if (0<=(f0 approx left cd11
                     (f0 uMod(S(S(S(S(S(S(n2+n8+n1)))))))-
                     a6+
                     (f0 approx right cd11
                      (f0 uMod(S(S(S(S(S(S(n2+n8+n1)))))))-
                      a6))/2)
             (left cd10@right cd11)
             (left cd11@right cd10)]]))
      (n3+f0 uModCont(S(S(S(S(n7+n1))))))
```

Let us now use this term to compute numerical approximations of values of an inverted function. First we construct the continuous function $x \mapsto x^2$ on $[1, 2]$, with its (trivial) uniform Cauchy modulus and modulus of uniform continuity, and give it the name `sq`:

```
(define sq (pt "contConstr 1 2([a0,n1]a0*a0)([n0]1)S"))
```

We now apply the extracted term of theorem `InvApprox` to

- the continuous `sq` to be inverted,
- a uniform modulus l of increase,

- a positive number k_0 such that $2^{-k_0-1} < b-a$, and a positive number k_1 such that $b-a < 2^{k_1}$ (which all happen to be 1 in this case),
- two rational bounds a_1, b_1 for an interval in the range,

and normalize the result:

```
(define inv-sq-approx
  (normalize-term
    (apply mk-term-in-app-form
      (list (proof-to-extracted-term
              (theorem-name-to-proof "InvApprox"))
            sq ;continuous function to be inverted
            (pt "1") ;uniform modulus of increase
            (pt "1") (pt "1") ;bounds for b-a
            (pt "1") (pt "4") ;interval in range
            ))))
```

which prints as

```
[a0,n1]
 left((cDC rat@@rat)(1@2)
      ([n2]
        (cId rat@@rat=>rat@@rat)
        ([cd4]
          [let cd5
            ((2#3)*left cd4+(1#3)*right cd4@
             (1#3)*left cd4+(2#3)*right cd4)
            [if (0<=(left cd5*left cd5-a0+
                    (right cd5*right cd5-a0))/2)
             (left cd4@right cd5)
             (left cd5@right cd4)]]))
      (S(S(S(S(S(S(S n1))))))))
```

The term sqrt-two-approx has type rat=>pos=>rat, where the first argument is for the rational to be inverted and the second argument k is for the error bound 2^{-k}. We can now directly (that is, without first translating into a programming language) use it to compute an approximation of say $\sqrt{3}$ to 20 binary digits. To do this, we need to "animate" Id and then normalize the result of applying inv-sq-approx to 3 and 20 (we use normalization by evaluation here, for efficiency reasons):

```
(animate "Id")
(pp (nbe-normalize-term-without-eta
      (make-term-in-app-form sqrt-two-approx (pt "20"))))
```

The result (returned in .7 seconds) is the rational

```
4402608752054#2541865828329
```

or 1.7320382149943123, which differs from $\sqrt{3} = 1.7320508075688772$ at the fifth (decimal) digit.

5.6. **Translation into Scheme expressions.** For a further speed-up (beyond the use of external code; cf. Section 4.2), we can also translate this internal term (where "internal" means "in our underlying logical language", hence usable in formal proofs) into an expression of a programming language (Scheme in our case), by evaluating `(term-to-expr inv-sq-approx)`:

```
(lambda (a0)
  (lambda (n1)
    (car (((cdc (cons 1 2))
           (lambda (n2)
             (lambda (cd4)
               (let ([cd5
                      (cons (+ (* 2/3 (car cd4))
                               (* 1/3 (cdr cd4)))
                            (+ (* 1/3 (car cd4))
                               (* 2/3 (cdr cd4))))])
                 (if (<= 0
                         (/ (+ (- (* (car cd5) (car cd5)) a0)
                               (- (* (cdr cd5) (cdr cd5)) a0))
                            2))
                     (cons (car cd4) (cdr cd5))
                     (cons (car cd5) (cdr cd4)))))))
          (+ (+ (+ (+ (+ (+ (+ n1 1) 1) 1) 1) 1) 1) 1)))))
```

This Scheme program is very close to the internal term displayed above; we have replaced the internal constant `cDC` (computational content of the axiom of dependent choice) by the corresponding Scheme function (a curried form of iteration):

```
(define cdc
  (lambda (init)
    (lambda (step)
      (lambda (n)
        (if (= 1 n)
            init
            ((step n) (((cdc init) step) (- n 1)))))))),
```

the internal arithmetical functions `+`, `*`, `/`, `<=` by the ones from the programming language and the internal pairing and unpairing functions by `cons`, `car` and `cdr`. – It turns out that this code is reasonably fast: evaluating

```
(((ev (term-to-expr inv-sq-approx)) 3) 200)
```

gives the result in .5 seconds, with an accuracy of 200 binary digits.

6. Conclusion, future work

The present case study shows that it is possible – albeit after some formalization effort – to machine extract reasonable terms from proofs in constructive analysis, and that ordinary evaluation of these terms can be used to numerically compute approximations to say reals whose existence is claimed by the theorems, with a prescribed precision.

As for future work, an obvious canditate is to do the same for the Cauchy-Euler construction of approximate solutions to ordinary differential equations. A particularly promising candiate is the treatment of ordinary differential equations in Chapter 1 of Hurewicz's textbook [10], which can easily be adapted to our constructive setting. It should also be possible to compare estimates for solutions of ordinary differential equations with the treatment of the same problem in the interval analysis setting of Moore [14].

References

[1] Patrik Andersson. Exact real arithmetic with automatic error estimates in a computer algebra system. Master's thesis, Mathematics department, Uppsala University, 2001.

[2] Josef Berger. Exact calculation of inverse functions. *Math. Log. Quart.*, 51(2):201–205, 2005.

[3] Ulrich Berger. Program extraction from normalization proofs. In M. Bezem and J.F. Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *LNCS*, pages 91–106. Springer Verlag, Berlin, Heidelberg, New York, 1993.

[4] Ulrich Berger. Uniform Heyting Arithmetic. *Annals Pure Applied Logic*, 133:125–148, 2005.

[5] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Term rewriting for normalization by evaluation. *Information and Computation*, 183:19–42, 2003.

[6] Errett Bishop. *Foundations of Constructive Analysis.* McGraw-Hill, New York, 1967.

[7] Luis Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications.* PhD thesis, Nijmegen University, 2004.

[8] Herman Geuvers, Freek Wiedijk, and Jan Zwanenburg. A constructive proof of the fundamental theorem of algebra without using the rationals. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Proc. Types 2000*, volume 2277 of *LNCS*, pages 96–111. Springer Verlag, Berlin, Heidelberg, New York, 2000.

[9] Kurt Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunkts. *Dialectica*, 12:280–287, 1958.

[10] Witold Hurewicz. *Lectures on Ordinary Differential Equations.* MIT Press, Cambridge, Mass., 1958.

[11] Georg Kreisel. Interpretation of analysis by means of constructive functionals of finite types. In A. Heyting, editor, *Constructivity in Mathematics*, pages 101–128. North–Holland, Amsterdam, 1959.

[12] Pierre Letouzey. A New Extraction for Coq. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002*, volume 2646 of *Lecture Notes in Computer Science.* Springer-Verlag, 2003.

[13] Mark Mandelkern. Continuity of monotone functions. *Pacific J. of Math.*, 99(2):413–418, 1982.

[14] Ramon E. Moore. *Interval Analysis.* Prentice-Hall, 1966.

[15] Helmut Schwichtenberg. Program extraction in constructive analysis. Submitted to: Logicism, Intuitionism, and Formalism – What has become of them? (eds. S. Lindström, E.Palmgren, K. Segerberg, V. Stoltenberg-Hansen), 2006.