

PROGRAM EXTRACTION IN EXACT REAL ARITHMETIC

KENJI MIYAMOTO AND HELMUT SCHWICHTENBERG

ABSTRACT. The importance of an abstract approach to a computation theory over general data types has been stressed by John Tucker in many of his papers. Ulrich Berger and Monika Seisenberger recently elaborated the idea for extraction out of proofs involving (only) abstract reals. They considered a proof involving coinduction of the proposition that any two reals in $[-1, 1]$ have their average in the same interval, and informally extract a Haskell program from this proof, which works with stream representations of reals. Here we formalize the proof, and machine-extract its computational content using the Minlog proof assistant. This required an extension of this system to also take coinduction into account.

Keywords: Program extraction, exact real numbers, coinduction, corecursion, streams.

Extraction of programs from constructive proofs has received extensive interest in recent years [11, 18]. Here we concentrate on an application area of particular importance, namely exact real arithmetic, seen as a subfield of constructive analysis. A standard way to approach the subject is to base the study on a concrete representation of (constructive) real numbers, like Cauchy sequences with a modulus [17, 14]. However, an attractive alternative to such an approach has been proposed by [5]: one might start out with an *abstract* theory of reals instead. But then the immediate question is: how could one associate computational content with a formula $\forall_x \dots$ where x ranges over abstract reals? By the very idea of abstractness we do (and should) not know the type of x ; in fact, it may be a type variable. Therefore the quantifier \forall_x should have no computational significance. Following [3] we use a “non-computational” universal quantifier \forall_x^{nc} instead, and move the computational content of our assumption into a predicate, i.e., we consider $\forall_x^{\text{nc}}(Px \rightarrow \dots)$ instead. This leaves the exact form of realizers for Px and hence the data type we use for representing real numbers open. Particularly relevant for concrete computations with exact real numbers is a stream representation based on signed digits, say $-1, 0, 1$. This has been realized very early: the thesis of [20, 21] (supervised by Engeler) is an example, and it is noted there that already Cauchy saw the usefulness of such a representation. Now how can we achieve that Px has a stream of signed digits as realizer? An obvious way is to define the predicate P coinductively.

Date: February 13, 2012.

Kenji Miyamoto is supported by the Marie Curie Initial Training Network in Mathematical Logic – MALOA – From MAtheMatical LOGic to Applications, PITN-GA-2009-238381.

More precisely, we start out with a (free) algebra \mathbf{I} of “lists of signed digits”, given by a nullary constructor \mathbb{I} of type \mathbf{I} and a binary constructor C of type $\mathbf{SD} \rightarrow \mathbf{I} \rightarrow \mathbf{I}$. Here \mathbf{SD} are the “signed digits” $-1, 0, 1$ (or L, M, R for left, middle, right); we write $C_d v$ for $C d v$. The intuition is that each signed digit indicates the subinterval of the half length. For example, \mathbb{I} means $[-1, 1]$, $C_0 \mathbb{I}$ means $[-\frac{1}{2}, \frac{1}{2}]$ and $C_1 C_0 \mathbb{I}$ means $[\frac{1}{4}, \frac{3}{4}]$. Now define inductively a set I of (abstract) reals, by the clauses

$$(1) \quad I0, \quad \forall_x^{\text{nc}} \forall_d (Ix \rightarrow I \frac{x+d}{2}).$$

Clearly, a witness for a proposition Ir according to the two clauses above can be seen as a constructor expression in our algebra \mathbf{I} , or alternatively as a “total ideal” in the intended model (the base domain of the Scott-Ershov model of partial continuous functionals over free algebras; cf. Section 1.1 or [18] for details). For I we can (as for every inductively defined predicate) define its “companion”, the coinductively defined predicate ${}^{\text{co}}I$, by the (single) clause

$$(2) \quad \forall_x^{\text{nc}} ({}^{\text{co}}Ix \rightarrow x = 0 \vee \exists_y^{\text{r}} \exists_d (x = \frac{y+d}{2} \wedge {}^{\text{co}}Iy)).$$

Here \exists_y^{r} indicates that the existentially quantified variable y is disregarded in the realizability interpretation; cf. Section 1.3. A witness for a proposition ${}^{\text{co}}Ir$ according to this clause can be seen as a finite or infinite stream of signed digits, indicating which signed digit d has been chosen in the second disjunct, and stopping if the first disjunct is taken. Such objects can be seen as “cototal ideals” for the algebra \mathbf{I} (see loc. cit. for details). Now we can formulate the proposition to be proved:

$$\forall_{x,y}^{\text{nc}} ({}^{\text{co}}Ix \rightarrow {}^{\text{co}}Iy \rightarrow {}^{\text{co}}I \frac{x+y}{2}),$$

where addition $+$ and the division by 2 are performed on the abstract level. By the above the computational content of this proposition will be a stream transformer from two streams realizing ${}^{\text{co}}Ix$ and ${}^{\text{co}}Iy$ into another stream realizing ${}^{\text{co}}I \frac{x+y}{2}$, and this is exactly what we want.

Now what will be special in the proof of our proposition? First of all, to reasonably work with the predicate ${}^{\text{co}}I$ we will need coinduction, or more precisely, the greatest-fixed-point axiom for ${}^{\text{co}}I$. On the level of extracted terms we will have to provide the corecursion operator, as the computational content of coinduction. The main contribution of the work reported in this paper is an appropriate extension of the general extraction mechanism (based on realizability) as implemented in the proof assistant Minlog¹, and its application in the present example.

Comparison with the literature. The importance of an abstract approach to a computation theory over general data types has been stressed by John Tucker in many of his papers (for instance [19]). In Berger and Seisenberger [5] the idea for extraction out of proofs involving (only) abstract reals is presented. Moreover, they give a proof of the proposition above about the average, and informally extract a Haskell program from

¹See <http://www.minlog-system.de>

this proof. Here we formalize the proof, and machine-extract its computational content using the Minlog proof assistant. This required an extension of this system to also take coinduction into account.

The average function has been studied from an exact real number perspective by Plume [15] and Ciaffaglione and Gianantonio in [8]. The latter paper starts with explicit definitions of stream transformers; correctness is verified in the proof assistant Coq [9]. Work involving coinduction is also prominent in the proof assistant Agda [1]; see the recent PhD thesis of Chi Ming Chuang [7]. However, in Agda whole proofs are taken as programs (not only their computationally relevant parts), and this causes difficulties.

1. THE FORMAL SYSTEM

We discuss particular features of the underlying formal system TCF (“theory of computable functionals”, cf. [18]), which are relevant for the example at hand.

1.1. Algebras and their total and cototal ideals. Rather than working with algebras and coalgebras in a categorical setting (as for instance done in [5]), we just use (free) algebras to generate the basic domains of the Scott-Ersov model of partial continuous functionals, and let the ground type variables range over the ideals (defined as deductively closed consistent sets of tokens, cf. [18]) in this domain. As an example, consider the algebra \mathbf{I} of standard intervals introduced above, and let \mathbf{C} range over its constructors.

- (a) The *tokens* a are the type correct constructor expressions $Ca_1^* \dots a_n^*$ where a_i^* is an *extended token*, i.e., a token or the special symbol $*$ which carries no information.
- (b) A finite set U of tokens is *consistent* if all its elements start with the same constructor \mathbf{C} , say of arity $\tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \mathbf{I}$, and all U_i are consistent ($i = 1, \dots, n$), where U_i consists of all (proper) tokens at the i -th argument position of some token in $U = \{Ca_1^*, \dots, Ca_m^*\}$.
- (c) $\{Ca_1^*, \dots, Ca_m^*\} \vdash C'a^*$ (read “entails”) is defined to mean $\mathbf{C} = \mathbf{C}'$, $m \geq 1$ and $U_i \vdash a_i^*$, with U_i as in (b) above (and $U \vdash *$ taken to be true).

To define total and cototal ideals, consider a constructor tree $P(*)$ with a distinguished occurrence of $*$. Then an arbitrary $P(Ca^*)$ is called *one-step extension* of $P(*)$, written $P(Ca^*) \succ_1 P(*)$. An ideal x is called *cototal* if every constructor tree $P(*) \in x$ has a \succ_1 -predecessor $P(C^*) \in x$; it is called *total* if it is cototal and the relation \succ_1 on x is well-founded. Every total ideal then can be seen as a standard interval

$$\mathbb{I}_{i-2^k, k} := \left[\frac{i}{2^k} - \frac{1}{2^k}, \frac{i}{2^k} + \frac{1}{2^k} \right] \quad \text{for } -2^k < i < 2^k.$$

However, the cototal ideals include $\{C_{-1}^n * \mid n \geq 0\}$, a “stream” representation of the real -1 , and also $\{C_{-1}C_1^n * \mid n \geq 0\}$ and $\{C_1C_{-1}^n * \mid n \geq 0\}$, which both represent the real 0 . Generally, the cototal ideals give us all reals in $[-1, 1]$, in the (non-unique) stream representation via signed digits $-1, 0, 1$.

1.2. Corecursion. Streams are infinite objects, and require a special treatment when computing with them. It is well-known that an arbitrary “reduction sequence” beginning with a term in Gödel’s T terminates. For this to hold it is essential that the constants allowed in T are restricted to constructors \mathbf{C} and (structural) recursion operators \mathcal{R} . A consequence will be that every closed term of a base type denotes a total ideal. The conversion rules for \mathcal{R} work from the leaves towards the root, and terminate because total ideals are well-founded. If, however, we deal with cototal ideals (streams, for example), then a similar operator is available to define functions with cototal ideals as values, namely “corecursion”. The corecursion operator ${}^{\text{co}}\mathcal{R}_{\mathbf{I}}^{\tau}$ is used to construct a mapping from τ to \mathbf{I} by corecursion on the structure of \mathbf{I} . Its type and conversion relation is as follows.

$$\begin{aligned} {}^{\text{co}}\mathcal{R}_{\mathbf{I}}^{\tau} &: \tau \rightarrow (\tau \rightarrow \mathbf{U} + \mathbf{SD} \times (\mathbf{I} + \tau)) \rightarrow \mathbf{I}, \\ {}^{\text{co}}\mathcal{R}_{\mathbf{I}}^{\tau} NM &\mapsto [\lambda_{\perp} \mathbb{I}, \lambda_{d,z} (C_d([\lambda_{\perp} \mathbb{I}, \lambda_u ({}^{\text{co}}\mathcal{R}_{\mathbf{I}}^{\tau} uM])z))](MN) \end{aligned}$$

or in a more readable notation

$$\begin{aligned} (3) \quad {}^{\text{co}}\mathcal{R}_{\mathbf{I}}^{\tau} NM &\mapsto [\mathbf{case} (MN)^{\mathbf{U} + \mathbf{SD} \times (\mathbf{I} + \tau)} \mathbf{of} \\ &\quad \text{inl } _ \mapsto \mathbb{I} \mid \\ &\quad \text{inr } \langle d, z \rangle \mapsto C_d[\mathbf{case} z^{\mathbf{U} + \tau} \mathbf{of} \\ &\quad \quad \text{inl } _ \mapsto \mathbb{I} \mid \\ &\quad \quad \text{inr } u^{\tau} \mapsto {}^{\text{co}}\mathcal{R}_{\mathbf{I}}^{\tau} uM]]. \end{aligned}$$

Note that \mathbf{U} is the unit type, and for $f: \rho \rightarrow \tau$ and $g: \sigma \rightarrow \tau$, $[f, g]$ of type $\rho + \sigma \rightarrow \tau$ is recursively defined on the sum type.

$$\begin{aligned} [f, g](\text{inl } x^{\rho}) &= fx, \\ [f, g](\text{inr } y^{\sigma}) &= gy. \end{aligned}$$

As an example of a function defined by corecursion consider the transformation of an abstract real in the interval $[-1, 1]$ into a stream representation using signed digits. Assume that we work in an abstract (axiomatic) theory of reals, having an unspecified type ρ , and that we have a type \mathbf{Q} for rationals as well. Assume that the abstract theory provides us with a function $g: \rho \rightarrow \mathbf{Q} \rightarrow \mathbf{Q} \rightarrow \mathbf{B}$ (\mathbf{B} is the type of booleans) comparing a real x with a proper rational interval $p < q$:

$$\begin{aligned} g(x, p, q) &= \mathbf{tt} \rightarrow x \leq q, \\ g(x, p, q) &= \mathbf{ff} \rightarrow p \leq x. \end{aligned}$$

From g we define a function $h: \rho \rightarrow \mathbf{U} + \mathbf{SD} \times (\mathbf{I} + \rho)$ by

$$h(x) := \begin{cases} \text{inr} \langle -1, \text{inr}(2x + 1) \rangle & \text{if } g(x, -\frac{1}{2}, 0) = \mathbf{tt}, \\ \text{inr} \langle 0, \text{inr}(2x) \rangle & \text{if } g(x, -\frac{1}{2}, 0) = \mathbf{ff}, g(x, 0, \frac{1}{2}) = \mathbf{tt}, \\ \text{inr} \langle 1, \text{inr}(2x - 1) \rangle & \text{if } g(x, 0, \frac{1}{2}) = \mathbf{ff}. \end{cases}$$

h is definable by a closed term M in Gödel’s T. Then the desired function $f: \rho \rightarrow \mathbf{I}$ transforming an abstract real x into a cototal ideal (i.e., a stream) in \mathbf{I} can be defined by

$$f(x) := {}^{\text{co}}\mathcal{R}_{\mathbf{I}}^{\rho} xM.$$

This $f(x)$ will thus be a stream of signed digits $-1, 0, 1$.

1.3. Realizability. We now address the issue of extracting computational content from proofs. The method of program extraction is based on *modified realizability* as introduced by Kreisel [12] and described in detail in [18]. In short, from every constructive proof M of a non-Harrop formula A (in natural deduction) one extracts a program $\text{et}(M)$ “realizing” A , essentially by removing computationally irrelevant parts from the proof (proofs of Harrop formulas have no computational content). The extracted program has some simple type $\tau(A)$ which depends solely on the logical shape of the proven formula A . In its original form the extraction process is fairly straightforward, but often leads to unnecessarily complex programs. In order to obtain better programs, proof assistants (for instance Coq, Isabelle/HOL, Agda, Nuprl, Minlog) offer various optimizations of program extraction. Below we describe optimizations implemented in Minlog [16], which are relevant for our present case study.

Quantifiers without computational content. Besides the usual quantifiers, \forall and \exists , Minlog has so-called *non-computational quantifiers*, \forall^{nc} and \exists^{r} , which allow for the extraction of simpler programs. These quantifiers, which were first introduced by Berger in [2], can be viewed as a refinement of the Set/Prop distinction in constructive type systems like Coq or Agda. Intuitively, a proof of $\forall_x^{\text{nc}} A(x)$ ($A(x)$ non-Harrop) represents a procedure that assigns to any x a proof $M(x)$ of $A(x)$ where $M(x)$ does not make “computational use” of x , i.e., the extracted program $\text{et}(M(x))$ does not depend on x . Dually, a proof of $\exists_x^{\text{r}} A(x)$ is a proof of $M(x)$ for some x where the witness x is “hidden”, that is, not available for computational use (the “r” stands for “right”); in fact, \exists^{r} can be seen as inductively defined by the clause $\forall_x^{\text{nc}}(A \rightarrow \exists_x^{\text{r}} A)$. The types of extracted programs for non-computational quantifiers are $\tau(\forall_x^{\text{nc}} A) = \tau(\exists_x^{\text{r}} A) = \tau(A)$ 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^{nc} -introduction and -elimination, $\text{et}((\lambda_x M^{A(x)})^{\forall_x^{\text{nc}} A(x)}) = \text{et}(M)$ and $\text{et}((M^{\forall_x^{\text{nc}} A(x)} t)^{A(t)}) = \text{et}(M)$ as opposed to $\text{et}((\lambda_x M^{A(x)})^{\forall_x A(x)}) = \text{et}(\lambda_x M)$ and $\text{et}((M^{\forall_x A(x)} t)^{A(t)}) = \text{et}(Mt)$. For the extracted programs to be correct the variable condition at \forall^{nc} -introduction must be strengthened by requiring in addition the abstracted variable x not to occur in the extracted program $\text{et}(M)$, and similarly for \exists^{r} . Note that for a Harrop formula A the formulas $\forall_x^{\text{nc}} A$, $\forall_x A$ and also $\exists_x^{\text{r}} A$, $\exists_x A$ are equivalent.

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 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 . We can be rather flexible here and enable/block rewriting by using `animate/deanimate` as desired.

1.4. Inductive and coinductive definitions. To be able to work from a computational point of view with our abstract reals, we have to inductively define what we need to know in order to be able to view an abstract real as

a computational object. This is done by means of inductive and coinductive definitions.

As an example, consider the inductive definition of I by the clauses (1) and the coinductive definition of ${}^{\text{co}}I$ by (2). We have already seen above that a witness for a proposition Ir can be seen as a total ideal in our algebra \mathbf{I} , and that a witness for a proposition ${}^{\text{co}}Ir$ can be seen as a cototal ideal. We still need to express that I is the least predicate satisfying the two clauses (1). This is done by means of the *least-fixed-point* axiom

$$(4) \quad \forall_x^{\text{nc}}(Ix \rightarrow P0 \rightarrow \forall_x^{\text{nc}}\forall_d(Ix \rightarrow Px \rightarrow P\frac{x+d}{2}) \rightarrow Px).$$

Dually we need to express that ${}^{\text{co}}I$ is the greatest predicate satisfying (2). This is done by means of the *greatest-fixed-point* axiom

$$(5) \quad \forall_x^{\text{nc}}(Px \rightarrow \forall_x^{\text{nc}}(Px \rightarrow x = 0 \vee \exists_d\exists_y^{\text{r}}(x = \frac{y+d}{2} \wedge ({}^{\text{co}}Iy \vee Py))) \rightarrow {}^{\text{co}}Ix).$$

Both can be understood as dealing with a “competitor” predicate P satisfying the same clauses/clause as $I/{}^{\text{co}}I$. Then (4) says that P is a superset of I , and (5) that P is a subset of ${}^{\text{co}}I$.

The computational content of these axioms depends on the type $\tau := \tau(Pr)$ of P . Then the term extracted from the least-fixed-point axiom (4) is the recursion operator \mathcal{R}_1^{τ} , and the term extracted from the greatest-fixed-point axiom (5) is the corecursion operator ${}^{\text{co}}\mathcal{R}_1^{\tau}$. Moreover, the terms extracted from the clauses (1) for I are the constructors of \mathbf{I} , and the term extracted from the clause (2) for ${}^{\text{co}}I$ is the *destructor* D for \mathbf{I} of type $\mathbf{I} \rightarrow \mathbf{U} + \mathbf{SD} \times \mathbf{I}$, defined by

$$D0 := \text{inl } \mathbf{u}, \quad D(C_d v) := \text{inr} \langle d, v \rangle.$$

2. THE INFORMAL PROOF

By our convention on variable names, d, e are for \mathbf{SD} , and i, j are for \mathbf{SD}_2 . Here \mathbf{SD}_2 are the “extended signed digits” $\{-2, -1, 0, 1, 2\}$ (or LL, LT, MT, RT, RR).

Theorem (Average).

$$\forall_{x,y}^{\text{nc}}({}^{\text{co}}Ix \rightarrow {}^{\text{co}}Iy \rightarrow {}^{\text{co}}I\frac{x+y}{2}).$$

Proof. We essentially follow the argument in [5]. Let

$$X := \left\{ \frac{x+y}{2} \mid x, y \in {}^{\text{co}}I \right\}, \quad Y := \left\{ \frac{x+y+i}{4} \mid x, y \in {}^{\text{co}}I, i \in \mathbf{SD}_2 \right\}.$$

Below we will show $X \subseteq Y$ and that Y satisfies the clause coinductively defining ${}^{\text{co}}I$. Therefore by the greatest-fixed-point for ${}^{\text{co}}I$ we have $Y \subseteq {}^{\text{co}}I$. Hence $X \subseteq {}^{\text{co}}I$, which is our claim. \square

Lemma (XSubY).

$$\forall_{x,y \in {}^{\text{co}}I} \forall_z^{\text{nc}} \left(z = \frac{x+y}{2} \rightarrow \exists_i \exists_{x',y' \in {}^{\text{co}}I} z = \frac{x'+y'+i}{4} \right).$$

Proof. Let $x, y \in {}^{\text{co}}I$ and $z := \frac{x+y}{2}$. Assume for instance $x = \frac{x'+d}{2}$ and $y = \frac{y'+e}{2}$ for some $x', y' \in {}^{\text{co}}I$ and $d, e \in \mathbf{SD}$. Then $z = \frac{x+y}{2} = \frac{x'+y'+d+e}{4}$. In case $x = 0$ and $y = \frac{y'+e}{2}$ we have $z = \frac{x+y}{2} = \frac{y'+e}{4}$. The other cases are similar. \square

Lemma (YSatClause).

$$\forall_i \forall_{x,y \in {}^{\text{co}}I} \forall_z^{\text{nc}} \left(z = \frac{x+y+i}{4} \rightarrow \right. \\ \left. z = 0 \vee \exists_{j,d} \exists_{x',y' \in {}^{\text{co}}I} \exists_{z'}^{\text{r}} \left(z' = \frac{x'+y'+j}{4} \wedge z = \frac{z'+d}{2} \right) \right).$$

Proof. Let $i \in \mathbf{SD}_2$ and $x, y \in {}^{\text{co}}I$. We show that $z := \frac{x+y+i}{4}$ satisfies the right hand side of the disjunction. In case $x = \frac{x'+d'}{2}$ and $y = \frac{y'+e'}{2}$ we have $z = \frac{x'+y'+d'+e'+2i}{8}$. Solve $d' + e' + 2i = j + 4d$. Then for $z' := \frac{x'+y'+j}{4}$

$$\frac{z'+d}{2} = \frac{4z'+4d}{8} = \frac{x'+y'+j+4d}{8} = \frac{x'+y'+d'+e'+2i}{8} = z.$$

The other cases are simpler. \square

3. FORMALIZATION AND EXTRACTION

Since the formal proof follows rather closely the informal one above, we do not comment on how it is generated interactively, but only on some of the more interesting points, and the extracted terms.

3.1. Formalization.

Abstract reals. We use a type variable ρ to denote an abstract type of reals. To formulate their properties, we need functions **P** (plus) of type $\rho \rightarrow \rho \rightarrow \rho$ for addition, and **H** (half) of type $\rho \rightarrow \rho$ for division by 2. Moreover we need auxiliary functions connecting the concrete data types **Z** (integers), **SD** (signed digits $\{-1, 0, 1\}$) and **SD**₂ (extended signed digits $\{-2, -1, 0, 1, 2\}$) with our abstract reals and also among themselves. These are

$$\mathbf{SDToInt}: \mathbf{SD} \rightarrow \mathbf{Z}, \quad \mathbf{SDtwoToInt}: \mathbf{SD}_2 \rightarrow \mathbf{Z}, \quad \mathbf{IntToR}: \mathbf{Z} \rightarrow \rho,$$

The properties we need to assume for our abstract reals are

$$\begin{aligned} (x+k)/2 + l &= (x + (k + \mathbf{z} 2l))/2, \\ (x+k)/4 + l &= (x + (k + \mathbf{z} 4l))/4, \\ (x+k)/2 + (y+l)/2 &= ((x+y) + (k + \mathbf{z} l))/2, \\ x+0 &= x, \quad 0+y = y, \\ 0/2 &= 0, \quad 2k/2 = k, \quad k+l = k + \mathbf{z} l, \end{aligned}$$

where $+$ is the addition on ρ and $+\mathbf{z}$ is one on **Z**. It is crucial to treat everything connected with ρ (a type variable) as non-computational. This is a point where the non-computational quantifiers are essential.

The functions J and D. In the proof of lemma YSatClause we had to solve $d' + e' + 2i = j + 4d$ for given $d', e' \in \mathbf{SD}$ and $i \in \mathbf{SD}_2$. This is a finite problem and hence can be solved by defining $J: \mathbf{SD} \rightarrow \mathbf{SD} \rightarrow \mathbf{SD}_2 \rightarrow \mathbf{SD}_2$ and $D: \mathbf{SD} \rightarrow \mathbf{SD} \rightarrow \mathbf{SD}_2 \rightarrow \mathbf{SD}$ explicitly. The validity of $d' + e' + 2i = J(d', e', i) + 4D(d', e', i)$ is then verified by means of case distinctions.

3.2. Extraction. We present terms extracted from the formalized proofs (literal output of Minlog), and give some explanations.

Extraction from lemma XSubY. The term extracted from the proof is

```
[v0,v1]
[if (des v0)
  [if (des v1)
    (MT@v0@v1)
    ([dv2]JOne M left dv2@v0@right dv2)]
  ([dv2]
    [if (des v1)
      (JOne left dv2 M@right dv2@v1)
      ([dv3]JOne left dv2 left dv3@right dv2@right dv3)]]]
```

Here v is a name for variables ranging over \mathbf{I} , and dv for variables ranging over $\mathbf{SD} \times \mathbf{I}$. The constant `des` denotes the destructor for \mathbf{I} of type $\mathbf{I} \rightarrow \mathbf{U} + \mathbf{SD} \times \mathbf{I}$, and `JOne`: $\mathbf{SD} \rightarrow \mathbf{SD} \rightarrow \mathbf{SD}_2$ adds the two integers. Clearly `left` and `right` are (prefix) operators for the components of a pair. The constant `cXSubY` of type $\mathbf{I} \rightarrow \mathbf{I} \rightarrow \mathbf{SD}_2 \times \mathbf{I} \times \mathbf{I}$ is defined to be the extracted term above. It satisfies the equations

$$\begin{aligned} \text{cXSubY}(\mathbb{I}, \mathbb{I}) &= \langle 0, \mathbb{I}, \mathbb{I} \rangle, \\ \text{cXSubY}(\mathbb{I}, C_e w) &= \langle e, \mathbb{I}, w \rangle, \\ \text{cXSubY}(C_d v, \mathbb{I}) &= \langle d, v, \mathbb{I} \rangle, \\ \text{cXSubY}(C_d v, C_e w) &= \langle d + e, v, w \rangle. \end{aligned}$$

For the given two streams with regarding \mathbb{I} as $C_M \mathbb{I}$, `cXSubY` computes the sum of the first two digits of them and the tails of the given two streams. This sum of digits of type \mathbf{SD}_2 is a “carry” which contains intermediate information to compute the average.

Extraction from lemma YSatClause. The term extracted from the proof is

```
[i0,v1,v2]
[if (des v1)
  [if (des v2)
    (J M M i0@D M M i0@v1@v2)
    ([dv3]J M left dv3 i0@D M left dv3 i0@v1@right dv3)]
  ([dv3]
    [if (des v2)
      (J left dv3 M i0@D left dv3 M i0@right dv3@v2)
      ([dv4]J left dv3 left dv4 i0@D left dv3 left dv4 i0
        @right dv3@right dv4)]]]
```

The constant `cYSatClause` of type $\mathbf{SD}_2 \rightarrow \mathbf{I} \rightarrow \mathbf{I} \rightarrow \mathbf{SD}_2 \times \mathbf{SD} \times \mathbf{I} \times \mathbf{I}$ is defined to be the extracted term above. It satisfies the equations

$$\begin{aligned} \text{cYSatClause}(i, \mathbb{I}, \mathbb{I}) &= \langle J(0, 0, i), D(0, 0, i), \mathbb{I}, \mathbb{I} \rangle, \\ \text{cYSatClause}(i, \mathbb{I}, C_e w) &= \langle J(0, e, i), D(0, e, i), \mathbb{I}, w \rangle, \\ \text{cYSatClause}(i, C_d v, \mathbb{I}) &= \langle J(d, 0, i), D(d, 0, i), v, \mathbb{I} \rangle, \\ \text{cYSatClause}(i, C_d v, C_e w) &= \langle J(d, e, i), D(d, e, i), v, w \rangle. \end{aligned}$$

For the given carry and two signed digit streams, `cYSatClause` computes the carry for the next step, the first signed digit of the average of the streams, and the tails of the streams.

Extraction from theorem Average. The term extracted from the proof is

```
[v0, v1]
(CoRec sdtwo@@iv@@iv=>iv)(cXSubY v0 v1)
([ivw2]
  Inr
  [let jdvw3
    (cYSatClause left ivw2 left right ivw2 right right ivw2)
    (left right jdvw3@
      (InR sdtwo@@iv@@iv iv)(left jdvw3@right right jdvw3))])]
```

of type $\mathbf{I} \rightarrow \mathbf{I} \rightarrow \mathbf{I}$. It calls `cXSubY` to compute the first carry and the tails of the inputs. Then `CoRec` repeatedly calls `cYSatClause` in order to compute the average step by step. Here `ivw` is a name for variables ranging over $\mathbf{SD}_2 \times \mathbf{I} \times \mathbf{I}$, and `jdvw` for variables ranging over $\mathbf{SD}_2 \times \mathbf{SD} \times \mathbf{I} \times \mathbf{I}$, and `[let x M N]` is for $N[M/x]$. The second argument of the corecursion operator, say $M: \mathbf{SD}_2 \times \mathbf{I} \times \mathbf{I} \rightarrow \mathbf{U} + \mathbf{SD} \times (\mathbf{I} + \mathbf{SD}_2 \times \mathbf{I} \times \mathbf{I})$, operates on an argument $\langle i, v, w \rangle$ as follows. Let $\text{cYSatClause}(i, v, w) = \langle j, d, v', w' \rangle$. Then $M\langle i, v, w \rangle = \text{inr}\langle d, \text{inr}\langle j, v', w' \rangle \rangle$. Given v and w , let $\text{cXSubY}(v, w) = \langle i, v, w \rangle$ for which N stands. Then MN is $\text{inr}\langle d, \text{inr}\langle j, v', w' \rangle \rangle$. Therefore by the conversion rule (3) for the corecursion operator the result is $C_d(\text{co}\mathcal{R}\langle j, v', w' \rangle M)$.

The extracted term can be applicable for concrete cases. An experiment is given in Section. 3.4.

3.3. Normalization. Recall that the term t extracted from a proof of a formula A is a realizer of A , in particular a term of the underlying formal theory TCF. This is in contrast to what is extracted in the proof assistants Coq [13] and Isabelle [6], which (for efficiency reasons) are programs in a programming language like OCaml, ML, Haskell or Scheme. We prefer to stay within TCF, for then the proposition $t \mathbf{r} A$ (“ t realizes A ”) can be proved formally, as a special case of the soundness theorem. This gives a higher degree of security, since the formal proof of $t \mathbf{r} A$ can easily be machine-checked, largely independent of the full proof assistant. When efficiency is an issue, one can – in a second step – translate the term t into an expression of a programming language and employ standard optimization techniques in the process. In Minlog the command is called `term-to-expr`; it has Scheme as the target language.

However, we can use the extracted (closed higher-order) term t directly as a program. For this to work we need to apply it to (closed) input or argument terms, and calculate (i.e., normalize) the resulting applicative term. For efficiency reasons we use normalization-by-evaluation [4] here. In case proofs involve coinduction the extracted term may contain corecursion operators. This creates a well-known problem for normalization, since the conversion rule (3) for the corecursion operator does not terminate. However, we can view this fact as a feature rather than a bug, since we *want* to compute with streams. In particular, we want to see the result only up to a given accuracy, which controls the number of unfoldings of the corecursion operator.

In the present case a simple approach suffices. First normalize the extracted term, treating the corecursion operator as a constant (i.e., without conversion rules), and then provide an external bound for the number of unfoldings of the corecursion operator. This could be refined to a more demand driven device, as Haskell's `take` command.

3.4. An experiment. Let `eterm` be the term above, extracted from the Average theorem. We want to use this term as a program to compute the average of $\frac{5}{8}$ and $\frac{3}{4}$. To this end we proceed as follows.

- (1) Normalize `eterm` to `neterm`, treating the corecursion operator as a constant.
- (2) Represent the two arguments as terms of type `I`. Recall that the constructors of the algebra `I` are `ℓ: I` and `C: SD → I → I`, and that the three (nullary) constructors of `SD` are `-1, 0, 1` (or internally `L, M, R` for left, middle, right); we write $C_d v$ for $C d v$. Then $\frac{5}{8} = \frac{1}{2} + \frac{1}{8}$ appears as $C_R(C_M(C_R\mathbb{I}))$ and $\frac{3}{4} = \frac{1}{2} + \frac{1}{4}$ as $C_R(C_R\mathbb{I})$.
- (3) Let `test` be the result of applying `neterm` to the two arguments, and `ntest` its normal form. This term still contains the corecursion operator; it denotes a cototal ideal in `I` (i.e., a stream).
- (4) Normalize `ntest` again, but this time allowing say 10 unfoldings of the corecursion operator. The commands are

```
(define eterm10 (undelay-delayed-corec ntest 10))
(define neterm10 (nt eterm10))
(pp neterm10)
```

The final result is

```
C R (C R (C M (C L (C M
(C M (C M (C M (C M (C M
((CoRec sdtwo@@iv@@iv=>iv) ... ))))))))
```

which is effectively computed in 30 msec. of time consumption. This is the correct result, since

$$\frac{\frac{5}{8} + \frac{3}{4}}{2} = \frac{11}{16} = \frac{1}{2} + \frac{1}{4} - \frac{1}{16}.$$

4. CONCLUSION

We presented a formal proof of the existence of the average of two real numbers in $[-1, 1]$, as a case study of constructive exact real arithmetic. From the formal proof involving coinduction we extracted a term containing the corecursion operator. The evaluation method of the corecursion operator allows to approximate the computation of cototal ideals or streams, the standard representation of non-well-founded data as real numbers.

As for the future work, general uniformly continuous functions can be studied with the same motivation as in the present paper. A promising treatment is in [3] (or [18]) which involves nested algebras in TCF and their cototal-total ideals.

REFERENCES

- [1] Agda. <http://wiki.portal.chalmers.se/agda/>.

- [2] U. Berger. Program extraction from normalization proofs. In M. Bezem and J. Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *LNCS*, pages 91–106. Springer Verlag, Berlin, Heidelberg, New York, 1993.
- [3] U. Berger. From coinductive proofs to exact real arithmetic. In E. Grädel and R. Kahle, editors, *Computer Science Logic*, LNCS, pages 132–146. Springer Verlag, Berlin, Heidelberg, New York, 2009.
- [4] U. Berger, M. Eberl, and H. Schwichtenberg. Term rewriting for normalization by evaluation. *Information and Computation*, 183:19–42, 2003.
- [5] U. Berger and M. Seisenberger. Proofs, programs, processes. In F. Ferreira et al., editors, *Proceedings CiE 2010*, volume 6158 of *LNCS*, pages 39–48. Springer Verlag, Berlin, Heidelberg, New York, 2010.
- [6] S. Berghofer. *Proofs, Programs and Executable Specifications in Higher Order Logic*. PhD thesis, Institut für Informatik, TU München, 2003.
- [7] C. M. Chuang. *Extraction of Programs for Exact Real Number Computation Using Agda*. PhD thesis, Swansea University, Wales, UK, 2011.
- [8] A. Ciaffaglione and P. D. Gianantonio. A certified, corecursive implementation of exact real numbers. *Theoretical Computer Science*, 351:39–51, 2006.
- [9] Coq Development Team. *The Coq Proof Assistant Reference Manual – Version 8.2*. Inria, 2009.
- [10] A. Heyting, editor. *Constructivity in Mathematics*. North-Holland, Amsterdam, 1959.
- [11] U. Kohlenbach. *Applied Proof Theory: Proof Interpretations and Their Use in Mathematics*. Springer Verlag, Berlin, Heidelberg, 2008.
- [12] G. Kreisel. Interpretation of analysis by means of constructive functionals of finite types. In Heyting [10], pages 101–128.
- [13] P. 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.
- [14] R. O’Connor. *Incompleteness & Completeness. Formalizing Logic and Analysis in Type Theory*. PhD thesis, Nijmegen University, 2009.
- [15] D. Plume. *A Calculator for Exact Real Number Computation*. PhD thesis, University of Edinburgh, 1998.
- [16] H. Schwichtenberg. Minlog. In F. Wiedijk, editor, *The Seventeen Provers of the World*, volume 3600 of *LNAI*, pages 151–157. Springer Verlag, Berlin, Heidelberg, New York, 2006.
- [17] H. Schwichtenberg. Realizability interpretation of proofs in constructive analysis. *Theory of Computing Systems*, 43(3):583–602, 2008.
- [18] H. Schwichtenberg and S. S. Wainer. *Proofs and Computations*. Perspectives in Logic. Association for Symbolic Logic and Cambridge University Press, to appear 2011.
- [19] J. V. Tucker and J. I. Zucker. Theory of computation over stream algebras, and its applications. In I. M. Havel and V. Koubek, editors, *MFCs*, volume 629 of *Lecture Notes in Computer Science*, pages 62–80. Springer, 1992.
- [20] E. Wiedmer. *Exaktes Rechnen mit reellen Zahlen und anderen unendlichen Objekten*. PhD thesis, ETH Zürich, 1977.
- [21] E. Wiedmer. Computing with infinite objects. *Theoretical Computer Science*, 10:133–155, 1980.