Prawf: An Interactive Proof System for Program Extraction

Ulrich Berger[1(✉)], Olga Petrovska[1], and Hideki Tsuiki[2]

[1] Swansea University, Swansea, UK
{u.berger,olga.petrovska}@swansea.ac.uk
[2] Kyoto University, Kyoto, Japan
tsuiki.hideki.8e@kyoto-u.ac.jp

Abstract. We present an interactive proof system dedicated to program extraction from proofs. In a previous paper [5] the underlying theory IFP (Intuitionistic Fixed Point Logic) was presented and its soundness proven. The present contribution describes a prototype implementation and explains its use through several case studies. The system benefits from an improvement of the theory which makes it possible to extract programs from proofs using unrestricted strictly positive inductive and coinductive definitions, thus removing the previous admissibility restrictions.

1 Introduction

One of the salient features of constructive proofs is the fact that they carry computational content which can be extracted by a simple automatic procedure. Examples of formal systems providing constructive proofs are intuitionistic (Heyting) arithmetic or (varieties of) constructive type theory. There exist several computer implementations of these systems, which support program extraction based on Curry-Howard correspondence (e.g. Minlog [4], Nuprl [8,10], Coq [7,9], Isabelle [1], Agda [2]). However, none of them has program extraction as their main raison d'être.

In [5] the system IFP (Intuitionistic Fixed Point Logic) was introduced whose primary goal is program extraction. IFP is first-order logic extended with least and greatest fixed points of strictly positive predicate transformers. Program extraction in IFP is based on a refined realizability interpretation that permits arbitrary classically true disjunction-free formulas as axioms and ignores the (trivial) computational content of proofs of Harrop formulas thus leading to programs without formal garbage. The main purpose of [5] was to show soundness of this realizability interpretation, that is, the correctness of extracted programs.

In the present paper we present Prawf[1], the first prototype of an implementation of IFP as an interactive proof system with program extraction feature. Prawf is based on a (compared with [5]) simplified notion of program and

[1] 'Prawf' (pronounced /prau̯v/) is Welsh for 'Proof'.

M. Anselmo et al. (Eds.): CiE 2020, LNCS 12098, pp. 137–148, 2020.
https://doi.org/10.1007/978-3-030-51466-2_12

an improved Soundness Theorem that admits least and greatest fixed points of arbitrary strictly positive predicate transformers, removing the admissibility restriction in [5].

The paper is structured as follows: In Sect. 2 we briefly recap IFP and program extraction in IFP, explaining in some detail the above mentioned changes and improvements. In Sect. 3 we describe Prawf and its basic use through some simple examples involving real and natural numbers. Section 4 contains an advanced case study about exact real number representations: The well-known signed digit representation and infinite Gray-code [12] are represented by coinductive predicates and inclusion between the predicates is proven in Prawf thus enabling the extraction of a program transforming the signed digit representation into infinite Gray-code. In the conclusion we reflect on what we achieved and compare our work with related approaches.

2 Program Extraction in IFP

We briefly summarize the system IFP and its associated program extraction procedure. For full details we refer to [5].

$$\frac{}{a : A \vdash a : A}\ \text{assumption}$$

$$\frac{p : A \qquad q : B}{\mathbf{Pair}(p, q) : A \wedge B}\ \wedge^+ \qquad \frac{p : A \wedge B}{\mathbf{proj}_L(p) : A}\ \wedge^-_L \qquad \frac{p : A \wedge B}{\mathbf{proj}_R(p) : B}\ \wedge^-_R$$

$$\frac{p : A}{\mathbf{Lt}(p) : A \vee B}\ \vee^+_L \qquad \frac{p : B}{\mathbf{Rt}(p) : A \vee B}\ \vee^+_R$$

$$\frac{p : A \vee B \qquad q : A \to C \qquad r : B \to C}{\mathbf{case}\, p\, \mathbf{of}\, \{\mathbf{Lt}(a) \to q\, a; \mathbf{Rt}(b) \to r\, b\} : C}\ \vee^-$$

$$\frac{a : A \vdash p : B}{\lambda a\, p : A \to B}\ \to^+ \qquad \frac{p : A \to B \qquad q : A}{p\, q : B}\ \to^-$$

$$\frac{p : A}{p : \forall x\, A}\ \forall^+ \qquad \frac{p : \forall x\, A}{p : A[t/x]}\ \forall^-$$

$$\frac{p : A[t/x]}{p : \exists x\, A}\ \exists^+ \qquad \frac{p : \exists x\, A \qquad q : \forall x\, (A \to B)}{p\, q : B}\ \exists^-$$

$$\frac{}{\lambda a\, a : \Phi(\Box(\Phi)) \subseteq \Box(\Phi)}\ cl \qquad \frac{}{\lambda a\, a : \Box(\Phi) \subseteq \Phi(\Box(\Phi))}\ cocl \qquad (\Box \in \{\mu, \nu\})$$

$$\frac{p : \Phi(\mathcal{P}) \subseteq \mathcal{P}}{\mathbf{rec}(\lambda f\, (p \circ m_\Phi\, f)) : \mu(\Phi) \subseteq \mathcal{P}}\ ind$$

$$\frac{p : \mathcal{P} \subseteq \Phi(\mathcal{P})}{\mathbf{rec}(\lambda f\, (m_\Phi\, f \circ p)) : \mathcal{P} \subseteq \nu(\Phi)}\ coind$$

Fig. 1. Proofs and their extracted programs

IFP *Syntax and Proofs.* The syntax of IFP has *terms*, *formulas*, *predicates* and *operators*, the latter describing strictly positive (s.p.) and hence monotone

predicate transformers. For every s.p. operator Φ there are predicates $\mu\Phi$ and $\nu\Phi$ denoting the predicates defined inductively resp. coinductively from Φ:

$$\begin{array}{lll}
\textit{Terms} \ni r, s, t & ::= & x \text{ (variables)} \mid f(t_1, \ldots, t_n) \ (f \text{ function constant}) \\
\textit{Formulas} \ni A, B & ::= & \mathcal{P}(\vec{t}) \ (\mathcal{P} \text{ not an abstraction, } \vec{t} \text{ arity}(\mathcal{P})\text{-many terms}) \\
 & & \mid A \wedge B \mid A \vee B \mid A \to B \mid \forall x\, A \mid \exists x\, A \\
\textit{Predicates} \ni \mathcal{P}, \mathcal{Q} & ::= & X \text{ (predicate variables)} \mid P \text{ (predicate constants)} \\
 & & \mid \lambda\vec{x}\, A \mid \mu\Phi \mid \nu\Phi \\
 & & (\text{arity}(\lambda\vec{x}\, A) = |\vec{x}|,\ \text{arity}(\mu\Phi) = \text{arity}(\nu\Phi) = \text{arity}(\Phi)) \\
\textit{Operators} \ni \Phi, \Psi & ::= & \lambda X\, \mathcal{P} \quad (P \text{ s.p. in } \mathcal{X},\ \text{arity}(\lambda X\, \mathcal{P}) = \text{arity}(X) = \text{arity}(\mathcal{P}))
\end{array}$$

where $\boldsymbol{P}$ *is s.p. in* $\boldsymbol{\mathcal{X}}$ if every free occurrence of $\boldsymbol{X}$ in $\boldsymbol{P}$ is at a *s.p. position*, i.e. not in the premise of an implication.

The proof rules of IFP are the usual rules of intuitionistic first-order logic with equality (regarding equality as binary predicate constant) augmented by rules stating that $\mu\Phi$ and $\nu\Phi$ are the least and greatest fixed points of Φ (see Fig. 1, ignoring for the moment the expressions to the left of the colon).

Programs. The programs extracted from proofs are terms in an untyped λ-calculus enriched by constructors for pattern matching and recursion.

$$\begin{array}{lll}
\textit{Programs} \ni p, q & ::= & a, b \ \text{(program variables)} \\
 & & \mid \mathbf{Nil} \mid \mathbf{Lt}(p) \mid \mathbf{Rt}(p) \mid \mathbf{Pair}(p, q) \\
 & & \mid \mathbf{case}\, p\, \mathbf{of}\, \{Cl_1; \ldots; Cl_n\} \\
 & & \mid \lambda a\, p \\
 & & \mid p\, q \\
 & & \mid \mathbf{rec}\, p
\end{array}$$

where in the case-construct each Cl_i is a *clause* of the form $C(a_1, \ldots, a_k) \to q$ in which C is a *constructor*, i.e. one of $\mathbf{Nil}, \mathbf{Lt}, \mathbf{Rt}, \mathbf{Pair}$, and the a_i are pairwise different program variables binding the free occurrences of the a_i in q. $\mathbf{rec}\, p$ computes the (least) fixed point of p, hence $p\, \mathbf{rec}(p) = \mathbf{rec}(p)$. It is well-known that an essentially equivalent calculus can be defined within the pure untyped λ-calculus, however, the enriched version is more convenient to work with. For the sake of readability we slightly simplify our notion of program compared to the one in [5] by no longer distinguishing between programs and functions.

Program Extraction. In its raw form the extracted program of a proof is simply obtained by replacing the proof rules by corresponding program constructs following the Curry-Howard correspondence. This is summarized in Fig. 1 where $p : A$ means that p is the program extracted from a proof of A. In the assumption rule and the rule $\to^+$, '$a : A \vdash$'indicates that the assumption A in the proof has been assigned the program variable a. In the rule $\wedge^-_L$, $\mathbf{proj}_L(p)$ stands for the program $\mathbf{case}\, p\, \mathbf{of}\, \{\mathbf{Pair}(a, b) \to a\}$. Similarly for $\wedge^-_R$. The rules $\forall^+$ and $\exists^-$

are subject to the usual provisos. In the rules ind and coind the program m_Φ realizes (in a sense explained below) the monotonicity of Φ, that is the formula $X \subseteq Y \to \Phi(X) \subseteq \Phi(Y)$ (with fresh predicate variables X, Y).

The correctness of programs is expressed through a realizability relation $p\,\mathbf{r}\,A$ between programs p and formulas A which is defined by recursion on formulas (see [5]). Formally, realizability is defined as a family of unary predicates $\mathbf{R}(A)$ on a Scott domain D of 'potential realizers'. $p\,\mathbf{r}\,A$ means that the denotation of p in D satisfies the predicate $\mathbf{R}(A)$. The Soundness Theorem [5] shows that if p is extracted from a proof of A, then p realizes A. The Soundness Theorem is formalised in a theory RIFP that extends IFP by a sort of realizers and axioms that describe the behaviour of programs. The denotational semantics of programs is linked to the operational one through the Adequacy Theorem, stating that programs with non-$\bot$ value terminate and reduce to that value [6].

Refinements. Program extraction in its raw form (as sketched above) produces correct programs which, however, contain a lot of garbage and are therefore practically useless. This is due to programs extracted from sub proofs of *Harrop formulas*, that is, formulas which do not contain a disjunction at a strictly positive position. These programs contain no useful information and should therefore be contracted to a trivial program, say **Nil**. In a refined realizability interpretation, which was presented in [5] and which is implemented in Prawf, this contraction is carried out. It is based on a refined notion of realizability and a refined program extraction procedure. The proof of the soundness theorem becomes considerably more complicated and could only be accomplished in [5] by subjecting induction and coinduction to a certain admissibility condition. In [6] a soundness proof without this restriction is given. It uses an intermediate system IFP′ whose induction and coinduction rules require as additional premise a proof of the monotonicity of Φ, e.g.,

$$\frac{p : \Phi(\mathcal{P}) \subseteq \mathcal{P} \qquad m : X \subseteq Y \to \Phi(X) \subseteq \Phi(Y)}{\mathbf{rec}(\lambda f\,(p \circ m\,f)) : \mu(\Phi) \subseteq \mathcal{P}}\; ind'$$

Soundness is then proven for IFP′ and transferred to IFP via an embedding of IFP into IFP′. Minlog [4] has a similar refined realizability interpretation but treats disjunction-free formulas and Harrop formulas in the same way. This simplifies program extraction but seems to restrict the validity of the Soundness Theorem to a constructive framework (see also the remarks in Sect. 5).

Axioms. For a proof with assumptions the soundness theorem states that the extracted program computes a realizer of the proven formula from realizers of the assumptions. If the assumptions contain no disjunctions at all - we call such assumptions *non-computational (nc)* - then they are Harrop formulas and hence their realizers are trivial but, even more, they are equivalent to their realizability interpretations. This fact is extremely useful since it implies that a program extracted from a proof that uses nc-assumptions (regarded as axioms specifying a class of structures) will not depend on realizers of these axioms and will be correct in any model of the axioms. For example, in a proof about real numbers

(see Sect. 3) the arithmetic operations may be given abstractly and specified by nc-axioms (e.g. $\forall x\,(x + 0 = x)$ and $\forall x\,(x \neq 0 \to \exists y\,(x * y = 1))$).

Computation vs. Equational Reasoning. In the systems Nuprl, Coq, Agda, and Minlog computation is built into the notion of proof by considering terms, formulas or types up to normal form with respect to certain rewrite rules. As a consequence, each of these systems has various (decidable or undecidable) notions of equality, which may make proof checking (deciding the correctness of a proof) algorithmically hard if not undecidable. The motivations for interweaving logic and computation are partly philosophical and partly practical since in this way a fair amount of (otherwise laborious) equational reasoning can be automatized. In contrast, the system IFP strictly separates computation from reasoning. Its proof calculus is free of computation and there is only one notion of equality obeying the usual rules of equational logic. This makes proof checking a nearly trivial task. Equational reasoning can be to a large extent (or even completely) externalised by stating the required equations (which are nc-formulas) as axioms which can be proven elsewhere (or believed). Computation is confined to programs and is given through rewrite rules which enjoy an Adequacy Theorem stating that the operational and the denotational semantics of programs match [3,6].

3 PRAWF

PRAWF [11] is a prototype implementation in Haskell, which allows users to write IFP proofs and extract executable programs from them. It follows pretty closely the theory of IFP sketched in the previous section but extends it in several respects:

- the logical language of PRAWF is many-sorted;
- names for predicates and operators can be introduced through declarations;
- induction and coinduction come in three variations, the original ones presented in Sect. 2, and two strengthenings (half-strong and strong (co)induction) which are explained and motivated below.

The software has two modes: a *prover mode* and an *execution mode*. The prover mode enables users to create a proof environment, consisting of a language, a context, declarations and axioms.

The proof rules in PRAWF correspond to those of IFP and include the usual natural deduction rules for predicate logic, rules for (co)induction, half-strong (co)induction and strong (co)induction, as well as the equality rules such as symmetry, reflexivity and congruence.

A theorem can be proven by applying these rules step by step or by using a tactic. A tactic consists of a sequence of proof commands that allows users to re-run a proof either partially or fully. Once proven, a theorem can be saved in a theory and used as a part of another proof.

The execution mode allows running extracted programs. In this mode a user can take advantage of the standard Prelude commands as well as special functions for running and showing programs.

An Introductory Example: Natural Numbers and Addition. We explain the working of PRAWF by means of a simple example based on the language of real numbers with the constants 0 and 1 and the operations + and − for addition and subtraction. We first give the idea in ordinary mathematical language and then show how to do it in PRAWF. We define the natural numbers as the least subset (predicate) of the reals that contains 0 and that contains x whenever it contains $x - 1$:

$$\mathbf{N} \stackrel{\text{Def}}{=} \mu(\Phi), \text{ where } \Phi \stackrel{\text{Def}}{=} \lambda X \lambda x (x = 0 \vee X(x - 1))$$

We prove that $\mathbf{N}$ is closed under addition:

$$\forall x(\mathbf{N}(x) \to \forall y(\mathbf{N}(y) \to \mathbf{N}(x + y)))$$

Hence assume $\mathbf{N}(x)$. We have to show $\forall y(\mathbf{N}(y) \to \mathbf{N}(x + y))$, that is $\mathbf{N} \subseteq \mathcal{P}$ where $\mathcal{P} \stackrel{\text{Def}}{=} \lambda y\, \mathbf{N}(x + y)$. By the induction rule it suffices to show $\Phi(\mathcal{P}) \subseteq \mathcal{P}$, that is,

$$\forall y\, ((y = 0 \vee \mathbf{N}(x + (y - 1)) \to \mathbf{N}(x + y))$$

If $y = 0$ then $\mathbf{N}(x + y)$ holds since $x + 0 = x$ (using an axiom) and $\mathbf{N}(x)$ holds by assumption. If $\mathbf{N}(x + (y - 1))$, then $\mathbf{N}((x + y) - 1)$ since $x + (y - 1) = x + (y - 1)$ (using an axiom) and hence $\mathbf{N}(x + y)$ by the closure rule.

In order to carry out this example in PRAWF one first needs to define the language. This can be done by creating in the directory `batches` a subdirectory `real` (the name can be freely chosen), and in that directory the text file `lang.txt` (the name is prescribed) with the contents

```
<sorts>
R
<end sorts>

<constants>
0,1:R
<end constants>

<functions>
+ : (R,R) -> R;
- : (R,R) -> R;
<end functions>

<predicates>
= : (R,R);
<end predicates>
```

Note that we do not need to give definitions of `+` and `-`. For the proofs it is sufficient to know their properties that are expressed through axioms.

In the same subdirectory one creates the file `decls.txt` (name prescribed) containing the definition of $\mathbf{N}$:

```
Phi:(R) = lambda Y:(R) lambda (z:R) (z=0 v Y(z-1))
N:(R) = Mu(Phi)
```

Finally, one creates (again in the directory `real`) the file `axi.txt` (name prescribed) containing the axioms one wishes to use. The axioms must be nc-formulas that is, not contain any disjunctions (for example the predicate **N** must not occur since its definition contains $\vee$ as part of the definition of Φ).

```
ax1 . all x:R x+0 = x
ax2 . all x:R all y:R (x+y)-1 = x+(y-1)
```

Now we are set to start our proof. We load the Haskell file `Mode.hs`, execute `main`, load our batch by typing `real` (after which the contents of the files we created will be displayed) and type at the prompt our goal formula

```
Enter goal formula> all x:R (N(x) -> all y:R (N(y) -> N(x+y)))
```

Proving in PRAWF proceeds in the usual goal-directed backwards reasoning style. In our example the first two steps are easy: `alli` (for $\forall$-introduction backwards), then `impi v1` (for $\rightarrow$-introduction backwards creating the assumption `v1 : N(x)`). After these two steps one arrives at

```
Assumptions:
 v1 : N(x)
Context of the goal:
 variables: x: R
Current goal:
 |-  ?2 : all (y:R) (N(y) -> N(x+y))
```

at which point we use induction by typing `ind`. This brings us to the premise of induction

```
Assumptions:
 v1 : N(x)
Context of the goal:
 variables: x: R
Current goal:
 |-  ?3 : all (y:R) (Phi(lambda (y:R) N(x+y))(y) -> N(x+y))
```

The command `unfold Phi` yields

```
Assumptions:
 v1 : N(x)
Context of the goal:
 variables: x: R
Current goal:
 |-  ?3 : all (y:R) ((y=0 v N(x+(y-1))) -> N(x+y))
```

which easily follows from our axioms and some equality reasoning. The necessary steps in PRAWF (and much more) can be found in a tutorial on the PRAWF website.

Program Extraction. After completing the proof above one can extract a program by typing `extract addition` (`addition` is a name we chose). This will write the extracted program into the file `progs.txt`:

```
addition . ProgAbst "v1" (ProgRec (ProgAbst "f_mu"
(ProgAbst "a_comp" (ProgCase (ProgVar "a_comp")
[(Lt,["a_ore"],ProgVar "v1"),(Rt,["b_ore"],ProgCon
Rt [ProgApp (ProgVar "f_mu") (ProgVar "b_ore")])])))
```

The program transforms realizers of $\mathbf{N}(x)$ and $\mathbf{N}(y)$ into a realizer of $\mathbf{N}(x + y)$. By the inductive definition of the predicate $\mathbf{N}$ its elements are realized in unary notation where `Lt(_)` plays the role of 0 and `Rt` plays the role of the successor function. The extracted program can be rewritten in more readable form as follows (using Haskell notation):

```
addition v1 a_comp = case a_comp of
                       {
                         Lt _ -> v1 ;
                         Rt b_ore -> Rt (addition v1 b_ore)
                       }
```

which clearly is the usual algorithm for addition of unary natural numbers. How to run this program is described in detail in the tutorial.

Half-Strong and Strong Induction. The premise of the induction rule for the predicate $\mathbf{N}$ is logically equivalent to the conjunction of the *induction base*, $\mathcal{P}(0)$, and the *induction step*, $\forall x\,(\mathcal{P}(x-1) \to \mathcal{P}(x))$. The induction step slightly differs from the usual induction step since it lacks the additional assumption $\mathbf{N}(x)$. This discrepancy disappears in the following rule of *half-strong induction* and its associated extracted program

$$\frac{p : \mu(\Phi) \cap \Phi(\mathcal{P}) \subseteq \mathcal{P}}{\mathbf{rec}(\lambda f\,(p \circ \langle \mathbf{id}, m_\Phi\, f\rangle) : \mu(\Phi) \subseteq \mathcal{P}}\ \mathit{hsind}$$

where $\langle f, g\rangle \stackrel{\text{Def}}{=} \lambda a\,\mathbf{Pair}(f\,a, g\,a)$ and $\mathbf{id} \stackrel{\text{Def}}{=} \lambda a\,a$. In the following example, half-strong induction appears to be needed to extract a good program. We aim to prove that the distance of two natural numbers is a natural number

$$\forall x(\mathbf{N}(x) \to \forall y(\mathbf{N}(y) \to \mathbf{N}(x - y) \vee \mathbf{N}(y - x)))$$

It is possible to prove this by (ordinary) induction on $\mathbf{N}(x)$, however, the proof is complicated and the extracted program contrived and inefficient. On the other hand, with half-strong induction (command `hsind`) the goal reduces to proving $\forall y(\mathbf{N}(y) \to \mathbf{N}(x-y) \vee \mathbf{N}(y-x))$ from the assumptions $\mathbf{N}(x)$ and $x = 0 \vee \forall y(\mathbf{N}(y) \to \mathbf{N}((x-1)-y) \vee \mathbf{N}(y-(x-1)))$ which, because of the extra assumption $\mathbf{N}(x)$, is relatively straightforward. Moreover, the extracted program is the expected one which removes successors from (realizers of) x and y until one of the two becomes 0 after which what remains of the other one is returned as result. The interested reader is invited to try this example on their own.

Strong induction is similar to half-strong induction, however, the intersection with $\mu(\Phi)$ is taken 'inside' Φ:

$$\frac{p : \Phi(\mu(\Phi) \cap \mathcal{P}) \subseteq \mathcal{P}}{\mathbf{rec}(\lambda f\,(p \circ m_\Phi\,\langle \mathbf{id}, f\rangle)) : \mu(\Phi) \subseteq \mathcal{P}}\ \mathit{sind}$$

For the case of the natural numbers its effect is that the step becomes logically equivalent to $\forall x\,(\mathbf{N}(x) \wedge \mathcal{P}(x) \to \mathcal{P}(x+1))$, that is, precisely the step in Peano induction. The extracted program corresponds exactly to primitive recursion.

Half-strong and strong coinduction will be discussed in Sect. 4.

4 Case Study: Exact Real Number Representations

As a rather large example, we formalize the existence of various exact representations of real numbers and prove the existence of conversions between them in PRAWF.

We continue to work in the theory of real numbers implemented in PRAWF through the batch `real` introduced in the previous section, but extend language, declarations and axioms as needed.

The structure of real numbers represented by the sort `R` and various constants and functions does not support any kind of computation on real numbers. For computation, we need representations. These can be provided in IFP through suitable predicates and their realizability interpretation, in a similar style as we represented unary natural numbers through the predicate **N**. In general, a representation is provided by defining a predicate $\mathcal{P}$ such that a realizer of $\mathcal{P}(x)$ is a representation of x.

Exact representations of real numbers are typically infinite sequences or streams which are naturally expressed through coinductively defined predicates. For example, the predicate `S(x)` meaning the existence of the signed digit representation of `x`, which is one of the standard representations of real numbers for computation, is expressed as follows. To give an impression how this looks in PRAWF we use in the following machine notation where `v` stands for $\vee$, `ex` for $\exists$, `m` is a constant for -1, `*` is multiplication, and `<=` means 'less or equal'.

```
SD:(R) = lambda (x:R) ((x = m v x = 1) v x = 0)
PhiS:(R) = lambda X:(R) lambda (x:R) ex (d:R)
                (SD(d) and  (abs(2*x-d)<= 1) and X(2*x-d))
S:(R) = Nu(PhiS)
```

This defines `S` as the largest predicate on the reals satisfying `S = PhiS(S)`. A realizer of `S(x)` is an infinite stream of signed digits where a digit is a realizer of a formula of the form `SD(y)` that is either `Lt(Lt(Nil))` or `Lt(Rt(Nil))` or `R(Nil)` (representing $-1, 0, 1$). Streams are given as infinitely nested pairs, e.g. (writing `a:b` for `Pair(a,b)`) `Lt(Lt(Nil)) : Rt(Nil) : Rt(Nil) : ...` (that is, $-1 : 0 : 0 : \ldots$) which represents the real number -0.5.

Another representation, called infinite Gray-code [12], is defined through a coinductive predicate `G(x)` defined in PRAWF by

```
B:(R) = lambda (x:R) (x <= 0 v 0 <= x)
D:(R) = lambda (x:R) (not (x = 0)) -> B(x)
PhiG:(R) = lambda X:(R) lambda (x:R)
                (m <= x and x <= 1) and (D(x) and X(t(x)))
G:(R) = Nu(PhiG)
```

Here, `t:(R)->R` is the tent function defined as `t(x)=1-2*abs(x)`. An interesting and challenging aspects of the infinite Gray-code is the fact that it is partial, more precisely, a realizer of `G(x)` is a stream that may have one undefined element. This is due to the premise `not (x = 0)` in the definition of `D` which, if false (that is, `x=0`), will admit as realizer a program whose value is undefined (e.g. a program that loops infinitely).

Following [6], we proved in Prawf

```
Theorem . all (x:R) (S(x) -> G(x))
```

The proof is rather involved in that it consists of two coinductions, half-strong coinduction, and Archimedean induction, which is a special form of induction suitable for proving predicates with a premise $x \neq 0$ like `D` (see below). Due to space restrictions we can only highlight the most interesting aspects of the proof. The main parts of the proof are

```
Claim1 . all x:R (S(x) -> D(x))
Claim2 . all x:R (S(x) -> S(t(x)))
```

which immediately implies the `Theorem` by coinduction.

The proof of `Claim1` uses the inductive predicate `Accp` defined by

```
PhiAccp:(R) = lambda X:(R) lambda (x:R) (all y:R y << x -> X(y))
Accp:(R) = Mu(PhiAccp)
```

where `x << y` is defined as `2*abs(x) <= 1 and y = 2*x`. `Accp` is the accessible or wellfounded part of the relation `<<`. Using Brouwer's Thesis, which states that induction on a well-founded relation is valid, and the Archimedean property of the reals (see [6]) one can show that `Accp(x)` holds for all nonzero `x`. Therefore, induction on `Accp(x)` turns into an induction principle for nonzero real numbers which in [6] is dubbed *Archimedean induction*. It is logically equivalent to the rule

$$\frac{\forall x \neq 0\,((|x| \leq 1/2 \to \mathcal{P}(2x)) \to \mathcal{P}(x))}{\forall x \neq 0\,\mathcal{P}(x)}\ AI$$

Archimedean induction is used to prove `all x:R (S(x) -> B(x))` which is the essential step in the proof of `Claim1`.

The proof of `Claim2` uses *half-strong coinduction* which is the rule

$$\frac{p : \mathcal{P} \subseteq \nu(\Phi) \cup \Phi(\mathcal{P})}{\mathbf{rec}(\lambda f\,([\mathbf{id} + m_\Phi\, f] \circ p)) : \mathcal{P} \subseteq \nu(\Phi)}\ hscoind$$

where $[f + g] \stackrel{\text{Def}}{=} \lambda a\, \mathbf{case}\, a\, \mathbf{of}\, \{\mathbf{Lt}(b) \to f\, b;\ \mathbf{Rt}(c) \to g\, c\}$. Similarly, *strong coinduction* is the rule

$$\frac{p : \mathcal{P} \subseteq \Phi(\nu(\Phi) \cup \mathcal{P})}{\mathbf{rec}(\lambda f\,(m_\Phi\,[\mathbf{id} + f] \circ p)) : \mathcal{P} \subseteq \nu(\Phi)}\ \mathit{scoind}$$

This rule can be used to give a short proof of `G(-x) -> G(x)` and extract a simple program which negates the head of the input stream and leaves its tail untouched (instead of recursively reproducing the tail, which would happen with ordinary coinduction).

5 Conclusion

We presented PRAWF, a first prototype implementation of the logical system IFP and its associated program extraction procedure. The successful formalization in PRAWF of exact real number representations and formal proofs of their relationships guarantee the correctness of the proofs in [6]. This advanced case study also gives us evidence that our approach scales to substantial nontrivial problems.

The examples also demonstrate the enormous advantage gained from the possibility of describing different data representation in an abstract setting using only first-order logic, and postulating arbitrary true nc-axioms. In the formalization of infinite Gray-code it was also essential that our method is able to produce partial extracted programs.

We would like to point out that the Soundness Theorem, that is, the correctness proof for extracted programs, though constructive, is valid with respect to a classical semantics. This is in line with the attitude in constructive mathematics to produce only results that are constructively *and* classically valid, which is not necessary the case in other approaches to program extraction.

Despite its successful maiden voyage PRAWF has some loose ends that need to be tied up. The most urgent one is an implementation of the soundness proof, that is, the enhancement of program extraction so that not only extracted programs but also their correctness proofs are created automatically. Currently, correctness relies on soundness as a meta theorem that has not been formalized yet. Other necessary improvements concern support for schematic theorems (Π^1_1-theorems, essentially), advanced proof tactics and interpretations between different languages.

We also plan to extend PRAWF by sequent calculus rules and rules that permit the extraction of concurrent programs. The latter will be needed to prove, conversely, that `G` (infinite Gray-code) is included in `S` (signed digit representation). We know from [12] that the extracted translation program has to be concurrent and nondeterministic.

Acknowledgements. This work was supported by the International Research Staff Exchange Scheme (IRSES) No. 612638 CORCON and No. 294962 COMPUTAL of the European Commission, the JSPS Core-to-Core Program, A. Advanced research Networks and JSPS KAKENHI Grant Number 15K00015 as well as the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 731143.

References

1. Isabelle. https://isabelle.in.tum.de/
2. Agda official website. http://wiki.portal.chalmers.se/agda/
3. Berger, U.: Realisability for induction and coinduction with applications to constructive analysis. J. Univ. Comput. Sci. **16**(18), 2535–2555 (2010)
4. Berger, U., Miyamoto, K., Schwichtenberg, H., Seisenberger, M.: Minlog - a tool for program extraction supporting algebras and coalgebras. In: Corradini, A., Klin, B., Cîrstea, C. (eds.) CALCO 2011. LNCS, vol. 6859, pp. 393–399. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22944-2_29
5. Berger, U., Petrovska, O.: Optimized program extraction for induction and coinduction. In: Manea, F., Miller, R.G., Nowotka, D. (eds.) CiE 2018. LNCS, vol. 10936, pp. 70–80. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94418-0_7
6. Berger, U., Tsuiki, H.: Intuitionistic fixed point logic (2019). Unpublished manuscript available on ArXiv
7. Bertot, Y., Castéran, P.: Interactive theorem proving and program development (2004)
8. Constable, R.: Implementing Mathematics with the Nuprl Proof Development System. Prentice-Hall, Upper Saddle River (1986)
9. The Coq Proof Assistant. https://coq.inria.fr
10. Lockwood, J.: Nuprl: an open logical programming environment: a practical framework for sharing formal models and tools. Program extraction (1998). http://www.nuprl.org
11. Prawf official website. https://prawftree.wordpress.com/
12. Tsuiki, H.: Real number computation through gray code embedding. Theor. Comput. Sci. **284**(2), 467–485 (2002)