Annals of Pure and Applied Logic 172 (2021) 102903

Contents lists available at ScienceDirect

Annals of Pure and Applied Logic

www.elsevier.com/locate/apal

Intuitionistic fixed point logic ☆

Ulrich Berger [a,*], Hideki Tsuiki [b]

[a] *Department of Computer Science, Swansea University, Swansea, United Kingdom*
[b] *Graduate School of Human and Environmental Studies, Kyoto University, Yoshida-Nihonmatsu, Kyoto, Japan*

ARTICLE INFO

Article history:
Received 6 October 2019
Received in revised form 22 September 2020
Accepted 4 October 2020
Available online 9 October 2020

MSC:
03B70
03D70
03D78
03F03
03F60
06B35

Keywords:
Proof theory
Realizability
Program extraction
Induction
Coinduction
Exact real number computation

ABSTRACT

We study the system IFP of intuitionistic fixed point logic, an extension of intuitionistic first-order logic by strictly positive inductive and coinductive definitions. We define a realizability interpretation of IFP and use it to extract computational content from proofs about abstract structures specified by arbitrary classically true disjunction free formulas. The interpretation is shown to be sound with respect to a domain-theoretic denotational semantics and a corresponding lazy operational semantics of a functional language for extracted programs. We also show how extracted programs can be translated into Haskell. As an application we extract a program converting the signed digit representation of real numbers to infinite Gray code from a proof of inclusion of the corresponding coinductive predicates.

Contents

☆ 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 Sklodowska-Curie grant agreement No. 731143.

* Corresponding author.
E-mail addresses: u.berger@swansea.ac.uk (U. Berger), tsuiki.hideki.8e@kyoto-u.ac.jp (H. Tsuiki).

https://doi.org/10.1016/j.apal.2020.102903
0168-0072/

1. Introduction

According to the Brouwer-Heyting-Kolmogorov interpretation of constructive logic, formulas correspond to data types and proofs to constructions of objects of these data types [69,50,23,71,70,64]. Moreover, by the Curry-Howard correspondence constructive proofs can be directly represented in a typed λ-calculus such that proof normalization is modelled by β-reduction. This tight connection between logic and computation has led to a number of implementations of proof systems that support the extraction of programs from constructive proofs, e.g. PX [38], Nuprl [23], Coq [24], Minlog [63,15], Isabelle/HOL [18], Agda [4]. In general, program extraction is restricted to proofs about structures that are constructively given. This can be considered a drawback since it excludes abstract mathematics done on a purely axiomatic basis. This paper introduces the formal system IFP of *Intuitionistic Fixed Point Logic* as a basis for program extraction from proofs that does not suffer from this limitation. Preliminary versions of the system were presented in [7–9,12,11].

IFP is an extension of first-order logic by inductive and coinductive definitions, i.e., predicates defined as least and greatest fixed points of strictly positive operators. Program extraction is performed via a 'uniform' realizability interpretation. Uniformity concerns the interpretation of quantifiers: A formula $\forall x\, A(x)$ is realized uniformly by one object a that realizes $A(x)$ for all x, so a may not depend on x. Dually, a formula $\exists x\, A(x)$ is realized uniformly by one object a that realizes $A(x)$ for some x, so a does not contain a witness for x. The usual interpretations of quantifiers may be recovered by relativization, $\forall x\,(D(x) \to A(x))$ and $\exists x\,(D(x) \wedge A(x))$, for a predicate D that specifies that x has some concrete representation. The uniform interpretation of quantifiers makes IFP classically inconsistent with the scheme 'realizability implies truth' (see the remark after Lemma 15 in Sect. 3). The Minlog system [63], which also supports program extraction based on realizability, does permit a uniform interpretation of quantifiers as well but differs from IFP in other respects, for example the treatment of inductive and coinductive definitions.

Besides the support of proofs about abstract structures on an axiomatic basis, IFP has further features that distinguishes it from other approaches to program extraction. *Classical logic*: Although IFP is based on intuitionistic logic a fair amount of classical logic is available. For example, soundness of realizability holds in the presence of any disjunction-free axioms that are classically true. Typical example is stability of equality, $\forall x, y\,(\neg\neg x = y \to x = y)$. *Partial computation*: Like the majority of programming languages, IFP's language of extracted programs admits general recursion and therefore partial, i.e., nonterminating computation. This makes it possible to extract data representations that are inherently partial, such as infinite Gray code [28,72] (see also Sect. 5). *Infinite computation*: Infinite data, as they naturally occur in exact real number computation, can be represented by infinite computations. This is achieved by an operational semantics where computations may continue forever outputting arbitrary close approximations to the complete (infinite) result at their finite stages (Sect. 6). *Haskell output*: Extracted programs are typable and can be translated into executable Haskell code in a straightforward way.

Related work: Minlog. The motivation for this article mainly stems from recent developments in the Minlog proof system [63]. Minlog implements a formal system which, from its very conception, is a constructive theory of computable objects and functionals with an effective domain-theoretic semantics [64]. In order to increase the expressiveness of the logic and the flexibility of program extraction this system has been extended by an elaborate 'decoration' mechanism for the logical operations that allows for a fine control of computational content (this extension is also described in [64]). For example, an existential quantifier can be decorated as 'computational' or 'non-computational' which causes the extracted program to include the witnessing term or not. Since in the non-computational case no witness is required, the range of the quantified variable no longer needs to be effectively (i.e. domain-theoretically) given but may be an abstract mathematical structure. This new possibility of including abstract structures in Minlog formalizations triggered the present article which studies the implications and the potential of a computationally meaningful theory of abstract structures in isolation. Minlog's 'non-computational' decoration corresponds to the uniform realizability interpretation of IFP mentioned earlier. There are some differences between Minlog and IFP though. For example, regarding the logical system, in Minlog all logical operations except implication and universal quantification are defined in terms of clausal inductive definitions while in IFP they are primitive and inductive definitions are not in the format of clauses. Regarding computational content, Minlog's realizers are typed and realizability is defined in the style of Kreisel's modified realizability [44] whereas in IFP realizers are untyped and realizability is closer to Kleene [40] (albeit IFP realizers are not numbers but domain elements denoted by functional programs).

PX. Another related system is PX [38] which is based on Feferman's system T_0 of explicit mathematics [32] and uses a version of realizability with truth to extract untyped programs from proofs. The main differences to IFP are that PX has a fixed, constructively given, model similar to LISP expressions and treats quantifiers in the usual 'non-uniform' way. PX supports positive inductive definitions, however, restricted to operators without computational content.

Further related work. Theories of inductive and coinductive definitions have been studied extensively in the past. The proof-theoretic strength of classical iterated inductive definitions has been determined in [21]. A proof-theoretic analysis of a stronger system that is close to IFP, but based on classical logic, has been given in [54]. In [74] it was shown that the proof-theoretic strength does not change if the base system is changed to intuitionistic logic. Inductive definitions have also been studied in the context of constructive set theory [3,61], type theory [29,55] and explicit mathematics [36]. In [5] and [78], Inductive definitions are related to theories of finite type in the framework of Gödel's Functional Interpretation. Propositional logics for inductive and coinductive definitions interpreted on (finite) labelled transition are known as *modal μ-calculi* [43,20]. These systems are based on classical logic and are mainly concerned with determining the computational complexity of definable properties aiming at applications in automatic program verification systems. Computational aspects of induction and coinduction (coiteration and corecursion), in particular questions regarding termination, are studied widely in the context of inductive and coinductive types. The

strongest and most far reaching normalization can be found in [52] and [51]. A programming language for real numbers extending PCF has been studied in [31]. It has a small step operational semantics that permits the incremental computation of digits, similar to our semantics in Sect. 6. Logical, computational, semantical and category-theoretical aspects of coinduction are studied in the context of coalgebra [39,47]. The representation of coinductive types in dependent type theories and the associated problems are an intensive object of study [34,25,37,1,13]. The computational complexity of corecursion has been studied in [60]. Realizability interpretation related to the one for IFP were also studied in [38,68,53,6] (see the introduction of [8] for a discussion of similarities and differences). In Constructive Analysis [19] and Computable Analysis [76] one works with represented structures and explicitly manipulates and reasons about these representations. In contrast, in IFP representations remain implicit and are made explicit only through realizability. Proof Mining [42] treats real numbers as a represented space but one can extract effective bounds from ineffective proofs about abstract spaces without a constructive representation (see e.g. [41,33]).

Overview of the paper. *Section 2* introduces the system IFP. Among other things, the usual principle of wellfounded induction is exhibited as an instance of strictly positive induction and shown to be strengthened by an abstract form of Brouwer's Thesis. The definitions are illustrated by an axiomatic specification of the real numbers and a definition of the natural numbers as an inductively defined subset of the reals. Special attention is paid to a formulation of the Archimedean property as an induction principle.

Section 3 begins with a definition of a Scott domain D that serves as the semantic domain of simple untyped functional programming language with constructors and unrestricted recursion. Then we introduce simple recursive types denoting sub domains of D that serve as spaces of potential realizers of formulas and show that the expected typing rules are valid. We extend IFP to a system RIFP that contains new sorts δ and Δ for elements and subdomains of D as well as new terms, called programs and types, for denoting them. This is followed by a formal realizability interpretation of IFP in RIFP. The interpretation is optimized by exploiting the fact that Harrop formulas, which are formulas that do not contain a disjunction at a strictly positive position, have trivial realizers (similar optimizations are available in the Minlog system).

In Section 4 we prove the Soundness Theorem (Theorem 2) which shows that from an IFP proof of a formula A from nc axioms one can extract a program provably realizing A. For the proof we use an intermediate system IFP$'$ which in the rules for induction and coinduction for the least and greatest fixed point of an operator Φ requires in addition a proof of monotonicity of Φ. We provide a recursive definition of the program extracted from an IFP derivation and give explicit constructions of realizers for derived principles such as wellfounded induction and its variants introduced in Sect. 2.

Section 5 is devoted to a case study on exact real number computation that utilizes all the concepts introduced so far. It is shown that the well-known signed digit representation and also the infinite (and partial!) Gray code representation can be obtained through realizability from simple coinductively defined predicates $\mathbf{S}$ and $\mathbf{G}$. A detailed IFP proof that $\mathbf{S}$ is contained in $\mathbf{G}$ is given and from it a program is extracted that converts the signed digit representation into infinite Gray code. The equivalence of the extracted program with the one given in [72] is also proved, which guarantees the correctness of the original program.

Section 6 introduces an operational semantics of programs that is able to capture infinite computation. While the First Adequacy Theorem (Theorem 5) states that an inductively defined bigstep reduction relation $\stackrel{\mu}{\Longrightarrow}$ captures the semantics of programs M with a finite total denotation, i.e., $M \stackrel{\mu}{\Longrightarrow} a$ iff $a = [\![M]\!]$, the Second Adequacy Theorem (Theorem 6) establishes an equivalence of programs that have a possibly *infinite* and *partial* denotational semantics with a *small step* reduction relation. This means that it is possible to incrementally compute arbitrary close approximations to a program that has an infinite value. Sect. 6 closes with a concrete example of infinite computation using a concrete instance of the results of Sect. 5.

Section 7 concludes the paper with a summary and a discussion of open problems and directions for further work.

2. Intuitionistic fixed point logic

We introduce the logical system IFP of *intuitionistic fixed point logic* as a basis for the formalization of proofs which can be subject to program extraction. IFP can be viewed as a subsystem of second-order logic with its standard classical set-theoretic semantics. We first define the language and the proof rules of IFP and then draw some simple consequences demonstrating that IFP includes common principles such as wellfounded induction and permits a natural formalization of real numbers as a real closed Archimedean field.

2.1. The formal system IFP

IFP is an extension of intuitionistic first-order predicate logic by least and greatest fixed points of strictly positive operators. Rather than a fixed system IFP is a *schema* for a family of systems suitable to formalize different mathematical fields. An *instance* of IFP is given by a *many-sorted first-order language* $\mathcal{L}$ and a set of *axioms* $\mathcal{A}$ described below. Hence $\mathcal{L}$ consists of

(1) *Sorts* $\iota, \iota_1, \dots$ as names for spaces of abstract mathematical objects.
(2) *Terms* $s, t, \dots$ with a notion of free variables and a notion of substitution. First order terms are the main example but we will also consider term languages with binding mechanism (Sect. 3).
(3) *Predicate constants*, each of fixed arity $(\vec{\iota})$.

Relative to a language $\mathcal{L}$ we define simultaneously

Formulas A, B: *Equations* $s = t$ (s, t terms of the same sort), $P(\vec{t})$ (P a predicate which is not an abstraction, $\vec{t}$ a tuple of terms whose sorts fit the arity of P), *conjunction* $A \wedge B$, *disjunction* $A \vee B$, *implication* $A \to B$, *universal and existential quantification* $\forall x\, A$, $\exists x\, A$.

Predicates P, Q: *Predicate variables* $X, Y, \dots$ (each of fixed arity), *predicate constants*, *abstraction* $\lambda \vec{x}\, A$ (arity given by the sorts of the variable tuple $\vec{x}$), $\mu(\Phi)$, $\nu(\Phi)$ (arities = arity of Φ).

Operators Φ: $\lambda X\, P$ where P must be strictly positive in X (see below) and the arities of X and P must coincide. The arity of $\lambda X\, P$ is this common arity.

Falsity is defined as $\mathbf{False} \stackrel{\text{Def}}{=} \mu(\lambda X\, X)()$ where X is a predicate variable of arity $()$.

By an *expression* we mean a formula, predicate, or operator. When considering an expression it is tacitly assumed that the arity of a predicate and the sorts of terms it is applied to fit. The set of free object variables and the set of free predicate variables of an expression are defined as expected.

An occurrence of an expression E is *strictly positive (s.p.)* in an expression F if that occurrence is not within the premise of an implication. A predicate P is strictly positive in a predicate variable X if every occurrence of X in P is strictly positive. The requirement of strict positivity could be easily relaxed to mere positivity. However, since non-strict positivity will not be required at any point, but would come at the cost of more complicated proofs, we refrain from this generalization. A similar remark applies to the strict positivity condition for fixed point types in Sect. 3.3.

We adopt the following *notational conventions*. Application of an abstraction to terms, $(\lambda \vec{x}\, A)(\vec{t})$, is defined as $A[\vec{t}/\vec{x}]$ (therefore $P(\vec{t})$ is now defined for all predicates P and terms $\vec{t}$ of fitting arity). Application of an operator $\Phi = \lambda X\, P$ to a predicate Q, $\Phi(Q)$, is defined as $P[Q/X]$. Instead of $P(\vec{t})$ we also write $\vec{t} \in P$ and a definition $P \stackrel{\text{Def}}{=} \mu(\Phi)$ will also be written $P \stackrel{\mu}{=} \Phi(P)$. The notation $P \stackrel{\nu}{=} \Phi(P)$ has a similar meaning. If $\Phi = \lambda X \lambda \vec{x}\, A$, then we also write $P(\vec{x}) \stackrel{\mu}{=} A[P/X]$ and $P(\vec{x}) \stackrel{\nu}{=} A[P/X]$ instead of $P \stackrel{\text{Def}}{=} \mu(\Phi)$ and $P \stackrel{\text{Def}}{=} \nu(\Phi)$. Inclusion of predicates (of the same arity), $P \subseteq Q$, is defined as $\forall \vec{x}\, (P(\vec{x}) \to Q(\vec{x}))$, intersection,

$P \cap Q$, as $\lambda\vec{x}\,(P(\vec{x}) \wedge Q(\vec{x}))$, and union, $P \cup Q$, as $\lambda\vec{x}\,(P(\vec{x}) \vee Q(\vec{x}))$. Pointwise implication, $P \Rightarrow Q$, is defined as $\lambda\vec{x}\,(P(\vec{x}) \to Q(\vec{x}))$. Hence $P \subseteq Q$ is the same as $\forall\vec{x}\,(P \Rightarrow Q)(\vec{x})$. Equivalence, $A \leftrightarrow B$, is defined as $(A \to B) \wedge (B \to A)$, and extensional equality of predicates, $P \equiv Q$, as $P \subseteq Q \wedge Q \subseteq P$.

Negation, $\neg A$, is defined as $A \to \mathbf{False}$ and inequality, $t \neq s$, as $\neg(t = s)$. Bounded quantification, $\forall x \in A\, B(x)$ and $\exists x \in A\, B(x)$, is defined, as usual, as $\forall x\,(A(x) \to B(x))$ and $\exists x\,(A(x) \wedge B(x))$. Exclusive 'or' and unique existence are defined as $A \oplus B \stackrel{\text{Def}}{=} (A \vee B) \wedge \neg(A \wedge B)$, $\exists_1 x\, A(x) \stackrel{\text{Def}}{=} \exists x \forall y\,(A(y) \leftrightarrow x = y)$. If we write $E = E'$ for expressions that are not terms, we mean that E and E' are syntactically equal up to renaming of bound variables.

An expression is called *non-computational (nc)* if it is disjunction-free and contains no free predicate variables. Our realizability interpretation (Sect. 3) will be defined such that nc formulas do not carry computational content and are interpreted by themselves. The reader may wonder why existential quantifiers aren't banned from nc formulas as well. The reason is that quantifiers are interpreted uniformly, in particular, existential quantifiers are not witnessed. The existential quantifier in intuitionistic arithmetic, which *is* witnessed, can be expressed in IFP by $\exists x\,(\mathbf{N}(x) \wedge A(x))$, where $\mathbf{N}(x)$ means 'x is a natural number' and the predicate $\mathbf{N}$ is defined using disjunction (see Sect. 2.3.3).

The set of *axioms* $\mathcal{A}$ of an $\mathcal{L}$-instance of IFP can be any set of closed $\mathcal{L}$-formulas. We denote that instance by IFP($\mathcal{A}$) leaving the language implicit since it is usually determined by the axioms. In order for IFP($\mathcal{A}$) to admit a sound realizability interpretation (Sects. 3 and 4), the axioms in $\mathcal{A}$ are required to be non-computational. The reason is that this guarantees that they have trivial computational content and are equivalent to their realizability interpretations, as will be explained in Sect. 3. Therefore, it suffices that the axioms are true in the intended structure where "true" can be interpreted in the sense of classical logic. For example, if one is willing to accept a certain amount of classical logic (as we do in this paper) one may include in $\mathcal{A}$ the *stability axiom*

$$\forall\vec{x}(\neg\neg A \to A)$$

for every nc formula A with free variables $\vec{x}$.

The *proof rules* of IFP include the usual natural deduction rules for intuitionistic first-order logic with equality (see below or e.g. [64]). In addition there are the following rules for strictly positive induction and coinduction:

$$\frac{}{\Phi(\mu(\Phi)) \subseteq \mu(\Phi)}\ \mathbf{CL}(\Phi) \qquad \frac{\Phi(P) \subseteq P}{\mu(\Phi) \subseteq P}\ \mathbf{IND}(\Phi, P)$$

$$\frac{}{\nu(\Phi) \subseteq \Phi(\nu(\Phi))}\ \mathbf{COCL}(\Phi) \qquad \frac{P \subseteq \Phi(P)}{P \subseteq \nu(\Phi)}\ \mathbf{COIND}(\Phi, P)$$

These rules can be applied in any context, that is, in the presence of free object and predicate variables as well as assumptions.

Intuitively, $\mu(\Phi)$ is the predicate defined inductively by the rules encoded by the operator Φ. For example natural numbers (viewed as a subset of the real numbers) can be defined as $\mathbf{N} \stackrel{\text{Def}}{=} \mu(\lambda X\, \lambda x\,(x = 0 \vee X(x-1)))$ corresponding to the rules '$\mathbf{N}(0)$' and 'if $\mathbf{N}(x-1)$, then $\mathbf{N}(x)$'. The closure axiom $\mathbf{CL}(\Phi)$ expresses that $\mu(\Phi)$ is closed under the rules, the induction rule $\mathbf{IND}(\Phi, P)$ says that $\mu(\Phi)$ is the smallest predicate closed under the rules (see also Sect. 2.3.3). Dually, $\nu(\Phi)$ is a coinductive predicate defined by 'co-rules'. For example, the elements of a partial order which start an infinite descending path can be characterized by the predicate $\mathbf{Path} = \nu(\lambda X\, \lambda x\, \exists y\,(y < x \wedge X(y)))$ (see also Sect. 2.2). Hence if $\mathbf{Path}(x)$, then $\mathbf{Path}(y)$ for some $y < x$ ($\mathbf{COCL}(\Phi)$), and $\mathbf{Path}$ is the largest predicate with that property ($\mathbf{COIND}(\Phi, P)$).

The existence of $\mu(\Phi)$ and $\nu(\Phi)$ is guaranteed, essentially, by Tarski's fixed point theorem applied to the complete lattice of predicates (of appropriate arity) ordered by inclusion and the operator Φ, which

is monotone due to its strict positivity. A simple but important observation is that $\mu(\Phi)$ and $\nu(\Phi)$ are (provably in IFP) least and greatest fixed points of Φ, respectively. For example, $\mu(\Phi) \subseteq \Phi(\mu(\Phi))$ follows by induction: One has to show $\Phi(\Phi(\mu(\Phi))) \subseteq \Phi(\mu(\Phi))$, which, by monotonicity, follows from the closure axiom $\Phi(\mu(\Phi)) \subseteq \mu(\Phi)$.

Induction and coinduction can be strengthened to the derivable principles of *strong and half-strong induction and coinduction* (which will be used in Sect. 5).

$$\frac{\Phi(P \cap \mu(\Phi)) \subseteq P}{\mu(\Phi) \subseteq P}\ \mathbf{SI}(\Phi, P) \qquad \frac{P \subseteq \Phi(P \cup \nu(\Phi))}{P \subseteq \nu(\Phi)}\ \mathbf{SCI}(\Phi, P)$$

$$\frac{\Phi(P) \cap \mu(\Phi) \subseteq P}{\mu(\Phi) \subseteq P}\ \mathbf{HSI}(\Phi, P) \qquad \frac{P \subseteq \Phi(P) \cup \nu(\Phi)}{P \subseteq \nu(\Phi)}\ \mathbf{HSCI}(\Phi, P)$$

It is clear that these proof rules are indeed strengthenings of ordinary s.p. induction since their premises are weaker due to the inclusions

$$\Phi(P \cap \mu(\Phi)) \subseteq \Phi(P) \cap \mu(\Phi) \subseteq \Phi(P)$$
$$\Phi(P \cup \nu(\Phi)) \supseteq \Phi(P) \cup \nu(\Phi) \supseteq \Phi(P)$$

which follow from the monotonicity of Φ. The derivations of these principles in IFP are straightforward. For example, assuming the premise of $\mathbf{SI}(\Phi, P)$, $\Phi(P \cap \mu(\Phi)) \subseteq P$, one defines another operator $\Psi = \lambda X\ \Phi(X \cap \mu(\Phi))$ so that $\Psi(P) \subseteq P$. Then, $\mu(\Psi) \subseteq P$ by induction. Hence it suffices to show $\mu(\Phi) \subseteq \mu(\Psi)$. The converse inclusion, $\mu(\Psi) \subseteq \mu(\Phi)$, follows by induction since $\Psi(\mu(\Phi)) \equiv \Phi(\mu(\Phi)) \subseteq \mu(\Phi)$. But now $\mu(\Phi) \subseteq \mu(\Psi)$ follows by induction since $\Phi(\mu(\Psi)) \subseteq \Phi(\mu(\Psi) \cap \mu(\Phi)) = \Psi(\mu(\Psi)) \subseteq \mu(\Psi)$. The other proofs are similar. Despite their derivability we will adopt these strengthenings of induction and coinduction as genuine rules of IFP since we can realize them by programs that are simpler than those that would be extracted from their derivations (see Theorem 2).

When defining the syntax of IFP we deliberately left open the exact structure of terms. This will give us greater flexibility regarding different instantiations of IFP. All we need to require of terms, in order to guarantee that the theorems of IFP are true with respect to the usual Tarskian semantics, is that their semantics satisfies a 'substitution lemma', that is

$$[\![t[r/x]]\!]\eta = [\![t]\!]\eta[x \mapsto [\![r]\!]\eta].$$

The rest follows from the Tarskian soundness of the rules of intuitionistic predicate logic and the existence of least and greatest fixed points of monotone predicate transformers as explained above.

Note that, since **False** is defined as $\mu(\lambda X\, X)()$, the schema *ex-falso-quodlibet*, $\mathbf{False} \to A$, follows from $A \to A$ by induction.

For the proof of the Soundness Theorem and the description of the program extraction procedure (Sect. 4) it will be convenient to denote IFP derivations by derivation terms and describe the proof calculus through an inductive definition of a set of derivation judgements $\Gamma \vdash d : A$ where Γ is a context of assumptions, d is a derivation term in that context, and A is the formula proved by the derivation. Derivations are defined relative to a given set $\mathcal{A}$ of *axioms* consisting of pairs (o, A) where A is any closed formula and o is the name of the axiom, though the soundness theorem holds only under nc axioms.

$$\Gamma, u : A \vdash u : A \qquad \Gamma \vdash o : A \quad ((o, A) \in \mathcal{A})$$

$$\Gamma \vdash \mathbf{Refl}_t : t = t \qquad \frac{\Gamma \vdash d : A[s/x] \qquad \Gamma \vdash e : s = t}{\Gamma \vdash \mathbf{Cong}_{\lambda x\, A}(d, e) : A[t/x]}$$

$$\frac{\Gamma \vdash d : A \qquad \Gamma \vdash e : B}{\Gamma \vdash \wedge^+(d, e) : A \wedge B} \qquad \frac{\Gamma \vdash d : A \wedge B}{\Gamma \vdash \wedge_l^-(d) : A} \qquad \frac{\Gamma \vdash d : A \wedge B}{\Gamma \vdash \wedge_r^-(d) : B}$$

$$\frac{\Gamma \vdash d : A}{\Gamma \vdash \vee^+_{l,B}(d) : A \vee B} \qquad \frac{\Gamma \vdash d : B}{\Gamma \vdash \vee^+_{r,A}(d) : A \vee B}$$

$$\frac{\Gamma \vdash d : A \vee B \qquad \Gamma \vdash e : A \to C \qquad \Gamma \vdash f : B \to C}{\Gamma \vdash \vee^-(d, e, f) : C}$$

$$\frac{\Gamma, u : A \vdash d : B}{\Gamma \vdash \to^+_{u:A}(d) : A \to B} \qquad \frac{\Gamma \vdash d : A \to B \qquad \Gamma \vdash e : A}{\Gamma \vdash \to^- (d, e) : B}$$

$$\frac{\Gamma \vdash d : A}{\Gamma \vdash \forall^+_x(d) : \forall x\, A}\ (x \text{ not free in } \Gamma) \qquad \frac{\Gamma \vdash d : \forall x\, A}{\Gamma \vdash \forall^-_t(d) : A[t/x]}$$

$$\frac{\Gamma \vdash d : A[t/x]}{\Gamma \vdash \exists^+_{\lambda x\, A, t}(d) : \exists x\, A} \qquad \frac{\Gamma \vdash d : \exists x\, A \qquad \Gamma \vdash e : \forall x\, (A \to B)}{\Gamma \vdash \exists^-(d, e) : B}\ (x \text{ not free in } B)$$

$$\Gamma \vdash \mathbf{Cl}_\Phi : \Phi(\mu(\Phi)) \subseteq \mu(\Phi) \qquad \frac{\Gamma \vdash d : \Phi(P) \subseteq P}{\Gamma \vdash \mathbf{Ind}_{\Phi,P}(d) : \mu(\Phi) \subseteq P}$$

$$\Gamma \vdash \mathbf{CoCl}_\Phi : \nu(\Phi) \subseteq \Phi(\nu(\Phi)) \qquad \frac{\Gamma \vdash d : P \subseteq \Phi(P)}{\Gamma \vdash \mathbf{CoInd}_{\Phi,P}(d) : P \subseteq \nu(\Phi)}$$

$$\frac{\Gamma \vdash d : \Phi(P) \cap \mu(\Phi) \subseteq P}{\Gamma \vdash \mathbf{HSInd}_{\Phi,P}(d) : \mu(\Phi) \subseteq P} \qquad \frac{\Gamma \vdash d : \Phi(P \cap \mu(\Phi)) \subseteq P}{\Gamma \vdash \mathbf{SInd}_{\Phi,P}(d) : \mu(\Phi) \subseteq P}$$

$$\frac{\Gamma \vdash d : P \subseteq \Phi(P) \cup \nu(\Phi)}{\Gamma \vdash \mathbf{HSCoInd}_{\Phi,P}(d) : P \subseteq \nu(\Phi)} \qquad \frac{\Gamma \vdash d : P \subseteq \Phi(P \cup \nu(\Phi))}{\Gamma \vdash \mathbf{SCoInd}_{\Phi,P}(d) : P \subseteq \nu(\Phi)}$$

Note that symmetry and transitivity of equality can be derived from reflexivity and the congruence rule.

2.2. Wellfounded induction and Brouwer's thesis

The principle of *wellfounded induction* is an induction principle for elements in the accessible or wellfounded part of a binary relation $\prec$ (definable in the language of the given instance of IFP). We show that it is an instance of strictly positive induction: The *accessible part* of $\prec$ is defined inductively by

$$\mathbf{Acc}_\prec(x) \stackrel{\mu}{=} \forall y \prec x\, \mathbf{Acc}_\prec(y)$$

that is, $\mathbf{Acc}_\prec = \mu(\Phi)$ where $\Phi \stackrel{\text{Def}}{=} \lambda X\, \lambda x\, \forall y \prec x\, X(y)$. A predicate P is called *progressive* if $\Phi(P) \subseteq P$, that is, $\mathbf{Prog}_\prec(P)$ holds where

$$\mathbf{Prog}_\prec(P) \stackrel{\text{Def}}{=} \forall x\, (\forall y \prec x P(y) \to P(x))\,.$$

Therefore, the principle of wellfounded induction, which states that a progressive predicate holds on the accessible part of $\prec$, is a direct instance of the rule of strictly positive induction:

$$\frac{\mathbf{Prog}_\prec(P)}{\mathbf{Acc}_\prec \subseteq P}\ \mathbf{WfI}_\prec(P)$$

In most applications P is of the form $A \Rightarrow P$. The progressivity of $A \Rightarrow P$ can be equivalently written as progressivity of P relativized to A,

$$\mathbf{Prog}_{\prec,A}(P) \stackrel{\text{Def}}{=} \forall x \in A\, (\forall y \in A\, (y \prec x \to P(y)) \to P(x))$$

and the conclusion becomes $\mathbf{Acc}_\prec \subseteq A \Rightarrow P$, equivalently, $\mathbf{Acc}_\prec \cap A \subseteq P$.

$$\frac{\mathbf{Prog}_{\prec,A}(P)}{\mathbf{Acc}_\prec \cap A \subseteq P}\ \mathbf{WfI}_{\prec,A}(P)$$

Dually to the accessibility predicate one can define for a binary relation a path predicate

$$\mathbf{Path}_\prec(x) \overset{\nu}{=} \exists y \prec x\, \mathbf{Path}_\prec(y)$$

that is, $\mathbf{Path}_\prec = \nu(\Phi)$ where $\Phi \overset{\text{Def}}{=} \lambda X\, \lambda x\, \exists y \prec x\, X(y)$. Intuitively, $\mathbf{Path}_\prec(x)$ states that there is an infinite descending path $\dots x_2 \prec x_1 \prec x$.

With the axiom of choice and classical logic one can show that $\neg\mathbf{Path}_\prec(x)$ implies $\mathbf{Acc}_\prec(x)$ (the converse holds even intuitionistically), which can be viewed as an abstract form of Brouwer's Thesis:

$$\mathbf{BT}_\prec \qquad \forall x\, (\neg\mathbf{Path}_\prec(x) \to \mathbf{Acc}_\prec(x))$$

In conjunction with wellfounded induction, $\mathbf{BT}_\prec$ says that $\prec$-induction is valid for all elements without infinite $\prec$-descending path.

If $\prec$ is defined in a disjunction-free way, then $\mathbf{BT}_\prec$ is a true nc formula which can be postulated as an nc axiom. By $\mathbf{BT_{nc}}$ we denote the schema $\mathbf{BT}_\prec$ for any binary nc predicate $\prec$. In Sect. 2.3.5 we will use $\mathbf{BT_{nc}}$ to justify a principle called 'Archimedean Induction' which in turn will be needed in Sect. 5.

Remark. Brouwer's original thesis which he used to justify Bar Induction is obtained from $\mathbf{BT}_\prec$ by defining for a 'bar predicate' P on finite sequences of natural numbers the relation $y \prec x \overset{\text{Def}}{=} \neg P(x) \wedge \exists a\, (y = ax)$, where ax denotes the sequence x prefixed with a (see e.g. [75]). Bar Induction on a predicate Q is then equivalent to $\mathbf{WfI}_\prec(Q)$.

2.3. Example: real numbers

We illustrate the concepts introduced so far by an instance of IFP providing an abstract specification of real numbers. This will be the basis for the program extraction case study in Sect. 5. Hence we will take care to postulate only non-computational axioms.

2.3.1. The language of real numbers

The language of the real numbers is given by

(1) *Sorts*: One sort ι as a name for the set of real numbers.
(2) *Terms*: First-order terms built from the constants and function symbols $0, 1, +, -, *, /,$ $2^{(\cdot)}$(exponentiation), max. Further function symbols may be added on demand. We set $|x| \overset{\text{Def}}{=} \max(x, -x)$.
(3) *Predicate constants*: $<, \leq$.

2.3.2. The axioms of real numbers

As axioms we may choose any disjunction-free formulas that are true in the real numbers. As such, we define $\mathcal{A}_R$ that consists of a disjunction-free formulation of the axioms of real-closed fields, equations for exponentiation, the defining axiom for max

$$\max(x, y) \leq z \leftrightarrow y \leq z \wedge x \leq z,$$

stability of $=, \leq, <$, as well as **AP** (Archimedean property) that will be defined in Sect. 2.3.4 and Brouwer's Thesis for nc predicates ($\mathbf{BT_{nc}}$) introduced in Sect. 2.2. In the remainder of Sect. 2 and also in Sect. 5 all proofs take place in IFP($\mathcal{A}_R$).

2.3.3. Natural numbers

The natural numbers, considered as a subset of the real numbers, can be defined inductively by

$$\mathbf{N}(x) \stackrel{\mu}{=} x = 0 \vee \mathbf{N}(x-1)$$

which is shorthand for $\mathbf{N} \stackrel{\text{Def}}{=} \mu(\Phi_{\mathbf{N}})$ where $\Phi_{\mathbf{N}} \stackrel{\text{Def}}{=} \lambda X\, \lambda x\, (x = 0 \vee X(x-1))$. Equivalently, one could define $\mathbf{N}(x) \stackrel{\mu}{=} x = 0 \vee \exists y\, (\mathbf{N}(y) \wedge x = y + 1)$. The closure and induction rules for $\mathbf{N}$ are literally

$$\frac{}{\forall x\, ((x = 0 \vee \mathbf{N}(x-1)) \to \mathbf{N}(x))} \qquad \frac{\forall x\, ((x = 0 \vee P(x-1)) \to P(x))}{\forall x \in \mathbf{N}\, P(x)}$$

equivalently (using equality reasoning and axioms for real numbers),

$$\frac{}{0 \in \mathbf{N}} \qquad \frac{}{\forall x \in \mathbf{N}\, (x + 1 \in \mathbf{N})} \qquad \frac{P(0) \wedge \forall x\, (P(x) \to P(x+1))}{\forall x \in \mathbf{N}\, P(x)}$$

The missing Peano axiom, $\forall x\, (\mathbf{N}(x) \to x + 1 \neq 0)$, follows from the formula $\forall x\, (\mathbf{N}(x) \to 0 \leq x)$ which can be proven by induction.

Strong induction on natural numbers is equivalent to

$$\frac{P(0) \wedge \forall x \in \mathbf{N}\, (P(x) \to P(x+1))}{\forall x \in \mathbf{N}\, P(x)}$$

The rational numbers $\mathbf{Q}$ can be defined from the natural numbers as usual, for example $\mathbf{Q}(q) \stackrel{\text{Def}}{=} \exists x, y, z \in \mathbf{N}\, (z \neq 0 \wedge q \cdot z = x - y)$.

Example 1. We prove that the sum of two natural numbers is a natural number, which is expressed as $\forall x, y\, (\mathbf{N}(x) \to \mathbf{N}(y) \to \mathbf{N}(x + y))$. An addition program for natural numbers will be extracted from this proof in Example 2. Suppose that x satisfies $\mathbf{N}(x)$. We prove $\forall y(\mathbf{N}(y) \to \mathbf{N}(x + y))$ by induction. Thus, we need to prove $\forall y\ (y = 0 \vee \mathbf{N}(x + (y-1)) \to \mathbf{N}(x + y))$. If $y = 0$, then $\mathbf{N}(x + y)$ holds by the assumption and $x + 0 = x$. If $\mathbf{N}(x + (y-1))$, then $y = 0 \vee \mathbf{N}((x + y) - 1)$ since $x + (y-1) = (x + y) - 1$. Therefore, $\mathbf{N}(x + y)$ holds by the closure rule.

2.3.4. Infinite numbers and the Archimedean property

As an example of a coinductive definition we define infinite numbers by

$$\infty(x) \stackrel{\nu}{=} x \geq 0 \wedge \infty(x-1)\,.$$

Hence a real number is infinite iff by repeatedly subtracting 1 one always stays non-negative (and hence positive).

The Archimedean property of real numbers can be expressed by stating that there are no infinite numbers:

$$\mathbf{AP} \qquad \forall x\, \neg\infty(x)$$

Since this is a true nc formula we include it as an axiom for the real numbers.

To give simple examples of proofs by induction and coinduction we show

Lemma 1. $\forall x\, (\infty(x) \leftrightarrow \forall y \in \mathbf{N}\, y \leq x)$.

Proof. For the implication from left to right we show

$$\forall y \in \mathbf{N}\, \forall x\, (\infty(x) \to y \le x)$$

by induction on $y \in \mathbf{N}$. $\forall x\, (\infty(x) \to 0 \le x)$ holds by the coclosure axiom for ∞. In the step, the induction hypothesis is $\forall x\, (\infty(x) \to y \le x)$. We have to show $\forall x\, (\infty(x) \to y + 1 \le x)$. Hence assume $\infty(x)$. By coclosure, $\infty(x-1)$. Therefore $y \le x - 1$, by the induction hypothesis. It follows $y + 1 \le x$.

The implication from right to left can be shown by coinduction. Setting $P(x) \stackrel{\text{Def}}{=} \forall y \in \mathbf{N}\, y \le x$ we have to show that $P(x)$ implies $x \ge 0$ and $P(x-1)$. Hence assume $P(x)$. $x \ge 0$ holds since $0 \in \mathbf{N}$. To show $P(x-1)$ let $y \in \mathbf{N}$. Then $y + 1 \in \mathbf{N}$ and therefore, since $P(x)$, $y + 1 \le x$. It follows $y \le x - 1$. □

2.3.5. Archimedean induction

Now we study a formulation of the Archimedean property as an induction principle. This principle will be needed to prove the conversion of the signed digit representation into Gray code (Theorem 4).

If we set $y \prec x \stackrel{\text{Def}}{=} x \ge 0 \wedge y = x - 1$, then clearly $\infty(x)$ is equivalent to $\mathbf{Path}_{\prec}(x)$. Therefore, by the Archimedean property, $\mathbf{Path}_{\prec}$ is empty, and by Brouwer's Thesis, **BT** (Sect. 2.2), it follows that $\mathbf{Acc}_{\prec}(x)$ holds for all x. Hence, wellfounded induction on $\prec$, $\mathbf{WfI}_{\prec}(P)$, is equivalent to the rule

$$\frac{\forall x\, ((x \ge 0 \to P(x-1)) \to P(x))}{\forall x\, P(x)}\ \mathbf{AI}(P)$$

since clearly its premise is equivalent to $\mathbf{Prog}_{\prec}(P)$.

A useful variant of **AI** is obtained by defining

$$y \prec_q x \stackrel{\text{Def}}{=} |x| \le q \wedge y = 2x$$

where here and in the following we assume $q > 0$. Then, as we will prove in Lemma 3, $\mathbf{Acc}_{\prec_q}(x)$ is equivalent to $x \ne 0$. Therefore, half strong induction, $\mathbf{HSI}(\Phi, P)$, for $\Phi = \lambda X\, \lambda x\, (\forall y \prec_q x\, X(y))$, yields the rule

$$\frac{\forall x \ne 0\ ((|x| \le q \to P(2x)) \to P(x))}{\forall x \ne 0\ P(x)}\ \mathbf{AI}_q(P)$$

since its premise is equivalent to $\mathbf{Prog}_{\prec_q}(P)$. We call the principles $\mathbf{AI}(P)$ and $\mathbf{AI}_q(P)$ *Archimedean induction*. Therefore, we have shown:

Lemma 2. *Archimedean induction is derivable in* $\mathrm{IFP}(\mathcal{A}_R)$.

Lemma 3. $\mathbf{Acc}_{\prec_q}(x)$ *iff* $x \ne 0$.

Proof. The 'only if' part follows by induction on $\mathbf{Acc}_{\prec_q}(x)$: Since $\mathbf{Acc}_{\prec_q}(x) \stackrel{\mu}{=} \forall y\, (|x| \le q \wedge y = 2x \to \mathbf{Acc}_{\prec_q}(y))$ is equivalent to $\mathbf{Acc}_{\prec_q}(x) \stackrel{\mu}{=} (|x| \le q \to \mathbf{Acc}_{\prec_q}(2x))$ it suffices to show that $(|x| \le q \to 2x \ne 0)$ implies $x \ne 0$, which is immediate (using $2 \cdot 0 = 0$).

The 'if' part reduces, by $\mathbf{BT_{nc}}$, to the implication $x \ne 0 \to \neg\mathbf{Path}_{\prec_q}(x)$. Therefore, we assume $x \ne 0$ and $\mathbf{Path}_{\prec_q}(x)$ with the aim to arrive at a contradiction. Recall that $\mathbf{Path}_{\prec_q}(x) \stackrel{\nu}{=} \exists y\, (|x| \le q \wedge y = 2x \wedge \mathbf{Path}_{\prec_q}(y))$, which is equivalent to $\mathbf{Path}_{\prec_q}(x) \stackrel{\nu}{=} (|x| \le q \wedge \mathbf{Path}_{\prec_q}(2x))$. By induction on $\mathbf{N}$ we can prove

$$\forall n \in \mathbf{N}\ \forall x(\mathbf{Path}_{\prec_q}(x) \to |x| \le q2^{-n})\,.$$

Therefore, if $\mathbf{Path}_{\prec_q}(x)$, then for all $n \in \mathbf{N}$, $q/|x| \ge 2^n \ge n$, which, by Lemma 1 and **AP**, is impossible. □

In most applications Archimedean induction is used with a predicate of the form $B \Rightarrow P$, and its premise is stated in an intuitionistically slightly stronger (though classically equivalent) form.

$$\frac{\forall x \in B\,(P(x) \vee (x \geq 0 \wedge B(x-1) \wedge (P(x-1) \to P(x))))}{\forall x \in B\,P(x)}\ \mathbf{AIB}(B,P)$$

$$\frac{\forall x \in B \setminus \{0\}\,(P(x) \vee (|x| \leq q \wedge B(2x) \wedge (P(2x) \to P(x))))}{\forall x \in B \setminus \{0\}\,P(x)}\ \mathbf{AIB}_q(B,P)$$

Lemma 4. **AI** *implies* **AIB**. $\mathbf{AI}_q$ *implies* $\mathbf{AIB}_q$.

Proof. The premise of $\mathbf{AIB}(B,P)$ implies the premise of $\mathbf{AI}(B \Rightarrow P)$. The premise of $\mathbf{AIB}_q(B,P)$ implies the premise of $\mathbf{AI}_q(B \Rightarrow P)$. □

3. Realizability

In this section we define a realizability interpretation of IFP. The interpretation will be formalized in a system RIFP defined in Sect. 3.4 which is another instance of IFP with extra sorts and terms for extracted programs and their types (Sect. 3.2 and Sect. 3.3) as well as axioms describing them (Sect. 3.4).

Our programming language is an untyped language with a type assignment system. It is similar to the language studied in [49], but simpler in that recursive types are restricted to strictly positive ones.

Programs will be interpreted in a Scott domain D satisfying a recursive domain equation, types will be interpreted as subdomains of D (Sect. 3.1).

A lazy operational semantics of this language will be studied in Sect. 6 and shown to be equivalent to the denotational semantics. Our domain-theoretic model of untyped programs originates in work by Scott [65]. An overview and comparison of different models of untyped λ-calculi can be found in [59]

To define the realizability interpretation, we first assign types to IFP expressions (Sect. 3.6) and then define the set of realizers of an expression as a subset of the subdomain defined by its type (Sect. 3.7). We also show that typable RIFP programs can be translated into Haskell programs (Sect. 3.5) and explain how Haskell programs can be directly extracted from IFP proofs in Section 4.

3.1. The domain of realizers and its subdomains

Extracted programs will be interpreted in a Scott domain D defined by the recursive domain equation

$$D = (\mathbf{Nil} + \mathbf{Left}(D) + \mathbf{Right}(D) + \mathbf{Pair}(D \times D) + \mathbf{Fun}(D \to D))_\perp$$

where $D \to D$ is the domain of continuous functions from D to D, $+$ denotes the disjoint sum of partial orders and $(\cdot)_\perp$ adds a new bottom element. $\mathbf{Nil}, \mathbf{Left}, \mathbf{Right}, \mathbf{Pair}, \mathbf{Fun}$ denote the injections of the various components of the sum into D. $\mathbf{Nil}, \mathbf{Left}, \mathbf{Right}, \mathbf{Pair}$ (but not $\mathbf{Fun}$) are called *constructors*. D carries a natural partial order $\sqsubseteq$ with respect to which D is a countably based *Scott domain* (*domain* for short), that is, a bounded complete algebraic dcpo with least element $\perp$ and countably many compact elements. The theory of Scott domains and recursive domain equations is standard and can be found e.g. in [2,35].

Since domains are closed under suprema of increasing chains, D contains not only finite but also infinite combinations of the constructors. For example, writing $a : b$ for $\mathbf{Pair}(a,b)$, an infinite sequence of domain elements $(d_i)_{i \in \mathbf{N}}$ is represented in D as the stream

$$d_0 : d_1 : \ldots \stackrel{\text{Def}}{=} \sup_{n \in \mathbf{N}} \mathbf{Pair}(d_0, \mathbf{Pair}(d_1, \ldots \mathbf{Pair}(d_n, \perp)\ldots))\,.$$

Since Scott domains and continuous functions form a cartesian closed category, D can be equipped with the structure of a partial combinatory algebra (PCA, [35]) by defining a continuous application operation $a\,b$ such that $\mathbf{Fun}(f)\,b \stackrel{\text{Def}}{=} f(b)$ and otherwise $a\,b \stackrel{\text{Def}}{=} \bot$, as well as combinators K and S satisfying $K\,a\,b = b$ and $S\,a\,b\,c = a\,c\,(b\,c)$ (where application associates to the left). In particular D has a continuous least fixed point operator which can be defined by Curry's Y-combinator or as the mapping $(D \to D) \ni f \mapsto \sup_n f^n(\bot) \in D$.

Besides the PCA structure we will use the algebraicity of D, that is, the fact that every element of D is the directed supremum of compact elements. Compact elements have a strongly finite character which will be exploited in the proof of uniqueness of certain fixed points (Sect. 3.3) and in the proof of the Computational Adequacy Theorem (Theorem 5). The finiteness of compact element is captured by their defining property ($d \in D$ is compact iff for every directed set $A \subseteq D$, if $d \sqsubseteq \bigsqcup A$, then $d \sqsubseteq a$ for some $a \in A$) and the existence of a function assigning to every compact element a a *rank*, $\mathbf{rk}(a) \in \mathbf{N}$, satisfying

rk1 If a has the form $C(a_1, \ldots, a_k)$ for a constructor C, then $a_1, \ldots, a_k$ are compact and $\mathbf{rk}(a) > \mathbf{rk}(a_i)$ $(i \leq k)$.

rk2 If a has the form $\mathbf{Fun}(f)$, then for every $b \in D$, $f(b)$ is compact with $\mathbf{rk}(a) > \mathbf{rk}(f(b))$ and there exists a compact $b_0 \sqsubseteq b$ such that $\mathbf{rk}(a) > \mathbf{rk}(b_0)$ and $f(b_0) = f(b)$. Moreover, there are finitely many compacts $b_1, \ldots b_n$ with $\mathbf{rk}(b_i) < \mathbf{rk}(a)$ such that $f(b) = \bigsqcup\{f(b_i) | i = 1, \ldots n, b_i \sqsubseteq b\}$.

In Sect. 3.3 we will model types as *subdomains* of D, that is, subsets of D that are downwards closed and closed under suprema of bounded subsets. We write $X \lhd D$ if X is a subdomain of D and denote by $\lhd D$ the set of all subdomains of D. It is easy to see that a subdomain X is a domain with respect to the partial order inherited from D and the notions of supremum and compact element in X are the same as taken with respect to D. The following is easy to see.

Lemma 5.

(a) $\lhd D$ *is a complete lattice. The meet operation is intersection.*
(b) $\lhd D$ *is closed under the following operations.*
$(X + Y)_\bot \stackrel{\text{Def}}{=} \{\mathbf{Left}(a) \mid a \in X\} \cup \{\mathbf{Right}(b) \mid b \in Y\} \cup \{\bot\}$,
$(X \times Y)_\bot \stackrel{\text{Def}}{=} \{\mathbf{Pair}(a, b) \mid a \in X, b \in Y\} \cup \{\bot\}$,
$(X \Rightarrow Y)_\bot \stackrel{\text{Def}}{=} \{\mathbf{Fun}(f) \mid f : D \to D \ \textit{cont.}, \forall a \in X(f(a) \in Y)\} \cup \{\bot\}$.

By Lemma 5 (a), for every set $S \subseteq D$ there is a smallest subdomain X containing S, called the subdomain *generated* by S. Hence for any subdomain Y, $S \subseteq Y$ iff $X \subseteq Y$. Furthermore, any subdomain is generated by the set of its compact elements.

3.2. Programs

In order to formally denote elements of D we introduce terms $M, N, \ldots$ of a new sort δ, called *programs*.

$$
\begin{aligned}
\textit{Programs} \ni M, N ::= {} & a, b \ \ \text{(program variables, i.e. variables of sort } \delta) \\
\mid {} & \mathbf{Nil} \mid \mathbf{Left}(M) \mid \mathbf{Right}(M) \mid \mathbf{Pair}(M, N) \\
\mid {} & \mathbf{case}\, M\, \mathbf{of}\, \{Cl_1; \ldots; Cl_n\} \\
\mid {} & \lambda a.\, M \\
\mid {} & M\, N \\
\mid {} & \mathbf{rec}\, M
\end{aligned}
$$

$$\mid \ \bot$$

In the case-construct each Cl_i is a *clause* of the form $C(\vec{a}) \to N$ where C is a constructor and $\vec{a}$ is a tuple of different variables whose free occurrences in N are bound by the clause. Furthermore, for different clauses $C(\vec{a}) \to M$ and $C'(\vec{a}') \to M'$, the constructors C and C' must be different. The intuitive meaning of a case-expression, say $\mathbf{case}\, M\, \mathbf{of}\, \{\ldots; \mathbf{Left}(a) \to L; \ldots\}$, is that if M evaluates to a term matching the pattern $\mathbf{Left}(a)$, say $\mathbf{Left}(M')$, then the whole case-expression evaluates to $L[M'/a]$. The recursion construct $\mathbf{rec}\, M$ defines the least fixed point of M. It could be defined as $Y\, M$ where Y is the well-known combinator $\lambda f\,.\,(\lambda a\,.\,f\,(a\,a))\,(\lambda a\,.\,f\,(a\,a))$, however, we prefer an explicit construct for general recursion since it better matches programming practice (Sect. 3.5) and it can be naturally assigned a type (Sect. 3.3) while Y involves self-application which is not typable in our system (see the remark after Lemma 13). The constant $\bot$ represents the 'undefined' domain element $\bot$. It could be defined as a non-terminating recursion but it is more convenient to have it as a constant. Overall, our goal is to have a programming language that enables us to naturally express the computational contents of IFP expressions and proofs.

Substitution of programs, $M[N/a]$, is defined as usual in term languages with binders so that a substitution lemma holds (Lemma 6). We also identify α-equal programs, that is, programs that are equal up to renaming of bound variables. Composition, sum, pairing, and projections are defined as

$$\begin{aligned}
M \circ N &\stackrel{\text{Def}}{=} \lambda a.\, M(N\, a)\\
[M + N] &\stackrel{\text{Def}}{=} \lambda c.\, \mathbf{case}\, c\, \mathbf{of}\, \{\mathbf{Left}(a) \to M\, a; \mathbf{Right}(b) \to N\, b\}\\
\langle M, N\rangle &\stackrel{\text{Def}}{=} \lambda c.\, \mathbf{Pair}(M\, c, N\, c)\\
\pi_{\mathbf{Left}}\, M &\stackrel{\text{Def}}{=} \mathbf{case}\, M\, \mathbf{of}\, \{\mathbf{Pair}(a, b) \to a\}\\
\pi_{\mathbf{Right}}\, M &\stackrel{\text{Def}}{=} \mathbf{case}\, M\, \mathbf{of}\, \{\mathbf{Pair}(a, b) \to b\}
\end{aligned}$$

We write $a \stackrel{\text{rec}}{=} M$ for $a \stackrel{\text{Def}}{=} \mathbf{rec}(\lambda a.\, M)$, and $a\, b \stackrel{\text{rec}}{=} M$ for $a \stackrel{\text{rec}}{=} \lambda b.\, M$. Occasionally we will use generalized clauses such as $\mathbf{Right}(\mathbf{Pair}(a, b)) \to M$ as an abbreviation for $\mathbf{Right}(c) \to \mathbf{case}\ c\ \mathbf{of}\ \{\mathbf{Pair}(a, b) \to M\}$.

Since D is a combinatory algebra every program M denotes an element $[\![M]\!]\eta \in D$ depending continuously (w.r.t. the Scott topology) on the environment η that maps program variables to elements of D.

$$\begin{aligned}
[\![a]\!]\eta &= \eta(a)\\
[\![C(M_1, \ldots, M_k)]\!]\eta &= C([\![M_1]\!]\eta, \ldots, [\![M_k]\!]\eta)\\
[\![\mathbf{case}\, M\, \mathbf{of}\, \{Cl_1; \ldots; Cl_n\}]\!]\eta &= [\![K]\!]\eta[\vec{a} \mapsto \vec{d}] \quad \text{if } [\![M]\!]\eta = C(\vec{d})\\
&\qquad \text{and some } Cl_i \text{ is of the form } C(\vec{a}) \to K\\
[\![\lambda a.\, M]\!]\eta &= \mathbf{Fun}(f) \quad \text{where } f(d) = [\![M]\!]\eta[a \mapsto d]\\
[\![M\, N]\!]\eta &= f([\![N]\!]\eta) \quad \text{if } [\![M]\!]\eta = \mathbf{Fun}(f)\\
[\![\mathbf{rec}\, M]\!]\eta &= \text{the least fixed point of } f\\
&\qquad \text{if } [\![M]\!]\eta = \mathbf{Fun}(f)\\
[\![M]\!]\eta &= \bot \ \text{ in all other cases, in particular } [\![\bot]\!]\eta = \bot
\end{aligned}$$

For closed terms the environment is redundant and may therefore be omitted. The following lemma is standard.

Lemma 6 *(Substitution).* $[\![M[N/a]]\!]\eta = [\![M]\!]\eta[a \mapsto [\![N]\!]\eta]$.

3.3. Types

We introduce simple recursive types which are interpreted as subdomains of the domain D defined in Sect. 3.1.

Types are expressions defined by the grammar

$$\mathit{Types} \ni \rho, \sigma ::= \alpha \text{ (type variables)} \mid \mathbf{1} \mid \rho + \sigma \mid \rho \times \sigma \mid \rho \Rightarrow \sigma \mid \mathbf{fix}\,\alpha\,.\,\rho.$$

where in $\mathbf{fix}\,\alpha\,.\,\rho$ the type ρ must be strictly positive in α.

Given an environment ζ that assigns to each type variable a subdomain of D, every type ρ denotes a subdomain D^ζ_ρ of D:

$$\begin{aligned} D^\zeta_\alpha &= \zeta(\alpha), \\ D^\zeta_{\mathbf{1}} &= \{\mathbf{Nil}, \bot\}, \\ D^\zeta_{\rho \diamond \sigma} &= (D^\zeta_\rho \diamond D^\zeta_\sigma)_\bot \quad (\diamond \in \{+, \times, \Rightarrow\}), \\ D^\zeta_{\mathbf{fix}\,\alpha\,.\,\rho} &= \bigcap\{X \lhd D \mid D^{\zeta[\alpha \mapsto X]}_\rho \subseteq X\} \end{aligned}$$

Lemma 7.

$$D^\zeta_{\mathbf{fix}\,\alpha\,.\,\rho} = D^{\zeta[\alpha \mapsto D^\zeta_{\mathbf{fix}\,\alpha\,.\,\rho}]}_\rho = D^\zeta_{\rho[\mathbf{fix}\,\alpha\,.\,\rho/\alpha]}.$$

Proof. By strict positivity, $D^{\zeta[\alpha \mapsto X]}_\rho$ is monotone in X. Therefore, the left equation holds by Tarski's fixed point theorem. The right equation is an instance of the usual substitution lemma. □

As an example, we consider the type of natural numbers,

$$\mathbf{nat} \stackrel{\text{Def}}{=} \mathbf{fix}\,\alpha\,.\,1 + \alpha.$$

By Lemma 7, $D_{\mathbf{nat}} = (D_{\mathbf{1}} + D_{\mathbf{nat}})_\bot$. It is easy to see that

$$D_{\mathbf{nat}} = \{\mathbf{Right}^n(d) \mid n \in \mathbf{N}, d \in \{\bot, \mathbf{Left}(\bot), \mathbf{Left}(\mathbf{Nil})\}\} \cup \{\sqcup_{n \in \mathbf{N}} \mathbf{Right}^n(\bot)\}.$$

By identifying $\mathbf{Left}(\bot)$ with $\mathbf{Left}(\mathbf{Nil})$, one obtains an isomorphic copy of the domain of lazy natural numbers where $\mathbf{Left}(\mathbf{Nil})$ represents 0 and **Right** represents the successor operation. See Remark 1 at the end of this section for a discussion on the relation between these domains.

Lemma 7 says that $D^\zeta_{\mathbf{fix}\,\alpha\,.\,\rho}$ is a fixed point of the type operator $\alpha \mapsto \rho$. We show that it is the *unique* fixed point under a regularity condition. The regularity conditions exclude type operators of the form $\alpha \mapsto \mathbf{fix}\,\beta_1\,.\,\ldots\,\mathbf{fix}\,\beta_n\,.\,\alpha$ (where the β_i are all different from α) which, obviously, have every subdomain of D as fixed point. It turns out that if fixed points of such type operators are excluded then uniqueness of fixed points holds. Therefore, we call a type *regular* if it contains no sub-expression of the form $\mathbf{fix}\,\alpha\,.\,\mathbf{fix}\,\beta_1\,.\,\ldots\,\mathbf{fix}\,\beta_n\,.\,\alpha$.

Lemma 8.

(a) *Regular types are closed under substitutions.*
(b) *Every regular type is semantically equal to a non-fixed-point type, that is, a type which is not of the form* $\mathbf{fix}\,\alpha\,.\,\rho$.

Proof. (a) is easy.

(b) can be proved by induction on the *fixed point height* of a type which is the unique number n such that the type is of the form $\mathbf{fix}\,\alpha_1 . \ldots \mathbf{fix}\,\alpha_n . \rho_0$ and ρ_0 is not a fixed point type. Let ρ be a regular type. If the fixed point height of ρ is 0 we are done. If the fixed point height of ρ is $n+1$, then ρ is of the form $\mathbf{fix}\,\alpha . \sigma$ where σ has fixed point height n. By Lemma 7, ρ is semantically equal to $\sigma[\rho/\alpha]$ which has fixed point height n as well since ρ is regular. Moreover, by (a), $\sigma[\rho/\alpha]$ is regular. Hence the induction hypothesis can be applied. □

Let X, Y range over $\lhd D$ and set $X \restriction n \stackrel{\text{Def}}{=} \{a \in X \mid a \text{ compact}, \mathbf{rk}(a) \leq n\}$.

Lemma 9. *If $X \restriction n \subseteq Y$ for all n, then $X \subseteq Y$.*

Proof. This is clear since a domain is the completion of the subset of its compact elements, and with increasing n, $X \restriction n$ exhausts all compact elements of X. □

Define $\mathbf{depth}_\alpha(\rho) \in \mathbf{N} \cup \{\infty\}$ by recursion on ρ as follows. $\mathbf{depth}_\alpha(\rho) = \infty$ if α is not free in ρ. Otherwise (using the expected order on $\mathbf{N} \cup \{\infty\}$ and setting $1 + \infty = \infty$)

$$\begin{aligned}
\mathbf{depth}_\alpha(\alpha) &= 0 \\
\mathbf{depth}_\alpha(\rho_1 \diamond \rho_2) &= 1 + \min_i \mathbf{depth}_\alpha(\rho_i) \quad \diamond \in \{+, \times\} \\
\mathbf{depth}_\alpha(\rho_1 \Rightarrow \rho_2) &= \mathbf{depth}_\alpha(\rho_2) \\
\mathbf{depth}_\alpha(\mathbf{fix}\,\beta . \rho) &= \mathbf{depth}_\alpha(\rho)
\end{aligned}$$

The following lemma exploits regularity in an essential way and is key to proving uniqueness of fixed points (Theorem 1).

Lemma 10. *Let ρ be regular and s.p. in α.*
If $X \restriction n \subseteq Y$, then $D_\rho^{\zeta[\alpha \mapsto X]} \restriction (n + \mathbf{depth}_\alpha(\rho)) \subseteq D_\rho^{\zeta[\alpha \mapsto Y]}$.

Proof. Suppose that $X \restriction n \subseteq Y$. We write $\rho(X)$ for $D_\rho^{\zeta[\alpha \mapsto X]}$ and show that for all compact elements $a \in \rho(X) \restriction (n + \mathbf{depth}_\alpha(\rho))$, we have $a \in \rho(Y)$. The proof is by induction on $\mathbf{rk}(a)$. We do a case analysis on ρ. Thanks to Lemma 8 (b) we may skip fixed point types.

Let $a \in \rho(X) \restriction (n + \mathbf{depth}_\alpha(\rho))$. If $a = \bot$ then the assertion holds since all subdomains contain $\bot$. Therefore in the following we assume $a \neq \bot$.

Case α is not free in ρ. Then $\rho(X) = \rho(Y)$ and therefore the assertion holds trivially.

Case $\rho = \alpha$. Then $\rho(X) = X$, $\rho(Y) = Y$ and $\mathbf{depth}_\alpha(\rho) = 0$. Therefore, the assertion is again trivial.

Case $\rho = \rho_1 + \rho_2$. W.l.o.g. $a = \mathbf{Left}(b)$ with $b \in \rho_1(X)$. Since $\mathbf{rk}(a) \leq n + \mathbf{depth}_\alpha(\rho) \leq n + 1 + \mathbf{depth}_\alpha(\rho_1)$ and $\mathbf{rk}(a) = 1 + \mathbf{rk}(b)$ it follows $\mathbf{rk}(b) \leq n + \mathbf{depth}_\alpha(\rho_1)$, that is, $b \in \rho_1(X) \restriction (n + \mathbf{depth}_\alpha(\rho_1))$. By induction hypothesis $b \in \rho_1(Y)$, hence $a \in \rho(Y)$.

Case $\rho = \rho_1 \times \rho_2$. Then $a = \mathbf{Pair}(a_1, a_2)$ with $a_i \in \rho_i(X)$ $(i = 1, 2)$. Since $\mathbf{rk}(a) \leq n + \mathbf{depth}_\alpha(\rho) \leq 1 + n + \mathbf{depth}_\alpha(\rho_i)$ and $\mathbf{rk}(a) \geq 1 + \mathbf{rk}(a_i)$ it follows $\mathbf{rk}(a_i) \leq n + \mathbf{depth}_\alpha(\rho_i)$, that is, $a_i \in \rho_i(X) \restriction (n + \mathbf{depth}_\alpha(\rho_i))$. By induction hypothesis $a_i \in \rho_i(Y)$, hence $a \in \rho(Y)$.

Case $\rho = \rho_1 \Rightarrow \rho_2$. Then $a = \mathbf{Fun}(f)$ with $f \in D \to D$ such that $f[\rho_1(X)] \subseteq \rho_2(X)$ and $\mathbf{rk}(f(a_1)) < \mathbf{rk}(a)$ for all $a_1 \in D$. We have to show $a \in \rho(Y)$, that is $f[\rho_1(Y)] \subseteq \rho_2(Y)$. Hence we assume $a_1 \in \rho_1(Y)$ and show $f(a_1) \in \rho_2(Y)$. Since ρ is s.p. in α, α is not free in ρ_1. Therefore $\rho_1(X) = \rho_1(Y)$ and we have $a_1 \in \rho_1(X)$. Since $\mathbf{rk}(f(a_1)) < \mathbf{rk}(a) \leq n + \mathbf{depth}_\alpha(\rho) = n + \mathbf{depth}_\alpha(\rho_2)$ it follows $\mathbf{rk}(f(a_1)) \leq n + \mathbf{depth}_\alpha(\rho_2)$, i.e. $f(a_1) \in \rho_2(X) \restriction (n + \mathbf{depth}_\alpha(\rho_2))$. By induction hypothesis $f(a_1) \in \rho_2(Y)$. □

Lemma 11. *Let ρ be regular and s.p. in α with $\mathbf{depth}_\alpha(\rho) > 0$. Assume $X \subseteq D_\rho^{\zeta[\alpha\mapsto X]}$ and $D_\rho^{\zeta[\alpha\mapsto Y]} \subseteq Y$. Then $X \subseteq Y$.*

Proof. By Lemma 9 it suffices to show that $X \restriction n \subseteq Y$ for all $n \in \mathbf{N}$. We induct on n.

$n = 0$: $X \restriction 0 = \{\bot\} \subseteq Y$.

$n + 1$: By induction hypothesis, $X \restriction n \subseteq Y$. Since $\mathbf{depth}_\alpha(\rho) > 0$ it follows with Lemma 10 that $D_\rho^{\zeta[\alpha\mapsto X]} \restriction (n+1) \subseteq D_\rho^{\zeta[\alpha\mapsto Y]}$. Therefore,

$$X \restriction (n+1) \subseteq D_\rho^{\zeta[\alpha\mapsto X]} \restriction (n+1) \subseteq D_\rho^{\zeta[\alpha\mapsto Y]} \subseteq Y \quad \square$$

Theorem 1 *(Uniqueness of fixed points). Let $\mathbf{fix}\,\alpha\,.\,\rho$ be regular. If $X \subseteq D_\rho^{\zeta[\alpha\mapsto X]}$ then $X \subseteq D^\zeta_{\mathbf{fix}\,\alpha\,.\,\rho}$, and if $X \supseteq D_\rho^{\zeta[\alpha\mapsto X]}$ then $X \supseteq D^\zeta_{\mathbf{fix}\,\alpha\,.\,\rho}$. Hence $X = D_\rho^{\zeta[\alpha\mapsto X]}$ iff $X = D^\zeta_{\mathbf{fix}\,\alpha\,.\,\rho}$,*

Proof. For the first implication use Lemma 11 with $Y \stackrel{\text{Def}}{=} D^\zeta_{\mathbf{fix}\,\alpha\,.\,\rho}$, noting that $D_\rho^{\zeta[\alpha\mapsto Y]} = Y$ by Lemma 7, and $\mathbf{depth}_\alpha(\rho) > 0$ since $\mathbf{fix}\,\alpha\,.\,\rho$ is regular. For the second implication the argument is similar. $\square$

Remark. In [49] a similar result is obtained for a larger type system that includes universal and existential type quantification as well as union and intersection types, and permitting fixed point types without positivity condition. Types are interpreted as ideals, which are similar to subdomains but are only closed under directed suprema. Subdomains are called strong ideals in [49]. The existence of fixed points is proven using the Banach Fixed Point Theorem w.r.t. a metric d such that for $X \neq Y$ as $d(X,Y) \stackrel{\text{Def}}{=} \min\{2^{-n} \mid X \restriction n \neq Y \restriction n\}$. We added strict positivity since this provides stronger information about extracted programs (see e.g. Lemma 37 and Theorem 7, and the remark after Lemma 13) and the definition of fixed points is more direct.

3.4. The formal system RIFP

We introduce an extension RIFP of IFP suitable for a formal definition of realizability and a formal proof of its soundness. In addition to the sorts of IFP, RIFP contains the sorts δ denoting the domain D, and Δ denoting the set of subdomains of D. Programs are terms of sort δ, types are terms of sort Δ. We also add a new relation symbol : of arity (δ, Δ) where $a : \alpha$ means that a is an element of the subdomain α. We write $\forall a : \rho\, A$ for $\forall a\,(a : \rho \to A)$ and $\exists a : \rho\, A$ for $\exists a\,(a : \rho \wedge A)$. We identify a type ρ with the predicate $\lambda a.\, a : \rho$, so that $\rho(a)$ stands for $a : \rho$ and, for example, $\rho \subseteq \sigma$ means $\forall a\,(a : \rho \to a : \sigma)$.

In addition to the axioms and rules of IFP, which are extended to the language of RIFP in the obvious way (and which include stability of equations), RIFP contains (universally quantified) axioms that reflect the denotational semantics of programs and types as well as those that express injectivity, range disjointness and compactness of constructors. Since we will not apply a realizability interpretation to RIFP we do not need to restrict axioms to nc formulas. We use the abbreviation $\mathbf{IsFun}(a) \stackrel{\text{Def}}{=} \exists b\,(a = \lambda c\,.\,(b\,c))$.

Axioms for programs

(i) $\mathbf{case}\, C_i(\vec{b})\, \mathbf{of}\{C_1(\vec{a}_1) \to M_1; \ldots; C_n(\vec{a}_n) \to M_n\} = M_i[\vec{b}/\vec{a}_i]$

(ii) $\bigwedge_i \forall \vec{b}\, a \neq C_i(\vec{b}) \;\to\; \mathbf{case}\, a\, \mathbf{of}\{C_1(\vec{a}_1) \to M_1; \ldots; C_n(\vec{a}_n) \to M_n\} = \bot$

(iii) $(\lambda b.\, M)\, a = M[a/b]$

(iv) $\neg\mathbf{IsFun}(a) \;\to\; a\, b = \bot$

(v) $\mathbf{IsFun}(a) \wedge \mathbf{IsFun}(b) \wedge \forall c\,(a\,c = b\,c) \;\to\; a = b$

(vi) $\bigoplus_{C \text{ constructor}} \exists_1 \vec{b}\,(a = C(\vec{b})) \;\oplus\; \mathbf{IsFun}(a) \;\oplus\; a = \bot$

(vii) $\mathbf{rec}\,a = a\,(\mathbf{rec}\,a)$

(viii) $P(\bot) \wedge \forall b\,(P(b) \;\to\; P(a\,b)) \to P(\mathbf{rec}\,a)$ $\quad (P \text{ admissible})$

where an RIFP predicate of arity (δ) is called *admissible* if it contains neither free predicate variables nor existential quantifiers nor inductive definitions.

Axioms for types

(ix) $\bot : \alpha$

(x) $\rho[\mathbf{fix}\,\alpha\,.\,\rho/\alpha] \equiv \mathbf{fix}\,\alpha\,.\,\rho$

(xi) $\beta \diamond \rho[\beta/\alpha] \;\to\; \beta \diamond \mathbf{fix}\,\alpha\,.\,\rho$ $\quad (\diamond \in \{\subseteq, \supseteq\},\ \mathbf{fix}\,\alpha\,.\,\rho \text{ regular})$

(xii) $c : \mathbf{1} \;\leftrightarrow\; (c = \mathbf{Nil} \vee c = \bot)$

(xiii) $c : \alpha \times \beta \;\leftrightarrow\; (\exists a : \alpha, b : \beta\,(c = \mathbf{Pair}(a, b)) \vee c = \bot)$

(xiv) $c : \alpha + \beta \;\leftrightarrow\; (\exists a : \alpha\,(c = \mathbf{Left}(a)) \vee \exists b : \beta\,(c = \mathbf{Right}(b)) \vee c = \bot)$

(xv) $c : \alpha \Rightarrow \beta \;\leftrightarrow\; ((\mathbf{IsFun}(c) \wedge \forall a : \alpha\,(c\,a : \beta)) \vee c = \bot)$

(xvi) $\exists \alpha \forall \beta\,(P \subseteq \beta \leftrightarrow \alpha \subseteq \beta)$ $\quad (P \text{ an RIFP predicate of arity } (\Delta))$

Clearly, axioms (i-vii) and (xii-xv) are correct in D respectively in $\triangleleft D$. Axiom (viii) is a restricted form of Scott-induction, a.k.a. fixed point induction. It is a way of expressing that $\mathbf{rec}\,a$ is the *least* fixed point of a, that is, the supremum of the chain $\bot \sqsubseteq a\,\bot \sqsubseteq a\,(a\,\bot) \sqsubseteq \ldots$. Scott-induction holds more generally for predicates that are closed under suprema of chains (such predicates are called *inclusive* in [77]). It is easy to see that admissible predicates have this property. As an example of Scott-induction, we show that every type is closed under least fixed points of endofunctions, that is,

$$a : \alpha \Rightarrow \alpha \;\to\; \mathbf{rec}\,a : \alpha.$$

Indeed, assuming $a : \alpha \Rightarrow \alpha$, the admissible predicate $P \stackrel{\text{Def}}{=} (\lambda b\,.\,b : \alpha)$ satisfies the premises of (viii) since $\bot : \alpha$ by axiom (ix) (which is valid since all subdomains of D contain $\bot$). Obviously, Scott-induction is also valid for admissible predicates of more than one argument, e.g.

$$P(\bot, \bot) \wedge \forall b_1, b_2\,(P(b_1, b_2) \;\to\; P(a_1\,b_1, a_2\,b_2)) \to P(\mathbf{rec}\,a_1, \mathbf{rec}\,a_2)$$

and Axiom (viii) should be understood in this more general form. Axioms (x) and (xi) hold by Lemmas 7 and 1. Axiom (xvi) expresses the existence of the subdomain generated by P and can be viewed as a form of comprehension.

We set $\mathrm{RIFP}(\mathcal{A}) \stackrel{\text{Def}}{=} \mathrm{IFP}(\mathcal{A} \cup \mathcal{A}')$ where $\mathcal{A}'$ consist of the axioms (i-xvi) for programs and types above. We write RIFP for $\mathrm{RIFP}(\mathcal{A})$ if the set of axioms is not important.

The following lemma will be used later to simplify extracted programs.

Lemma 12. $\mathrm{RIFP}(\emptyset)$ *proves: If f is strict, that is, $f\,\bot = \bot$, then*

$$f(\mathbf{case}\,M\,\mathbf{of}\,\{C_1(\vec{a}_1) \to L_1; \ldots; C_n(\vec{a}_n) \to L_n\}) \\ = \mathbf{case}\,M\,\mathbf{of}\,\{C_1(\vec{a}_1) \to f\,L_1; \ldots; C_n(\vec{a}_n) \to f\,L_n\}\,.$$

Proof. Let f be strict. We have to prove the equation $f\,K = K'$ where $K \stackrel{\text{Def}}{=} \mathbf{case}\,M\,\mathbf{of}\,\{C_1(\vec{a}_1) \to L_1; \ldots; C_n(\vec{a}_n) \to L_n\}$ and $K' \stackrel{\text{Def}}{=} \mathbf{case}\,M\,\mathbf{of}\,\{C_1(\vec{a}_1) \to f\,L_1; \ldots; C_n(\vec{a}_n) \to f\,L_n\}$. Since we have to prove an equation and we assume equations to be $\neg\neg$-stable, we may use classical logic. If $M = C_i(\vec{b})$ for some i and $\vec{b}$, then $K = L_i[\vec{b}/\vec{a}_i]$ and $K' = f\,L_i[\vec{b}/\vec{a}_i]$, by axiom (i), and the equation holds. Otherwise, $K = K' = \bot$ by axiom (ii), and the equation holds since f is strict. □

Lemma 13. *The following typing rules are derivable in* RIFP($\emptyset$) *(where* Γ *is a typing context, that is, a list of assumptions* $a_1 : \rho_1, \ldots a_n : \rho_n$*).*

$$\Gamma, a : \rho \vdash a : \rho \qquad \Gamma \vdash \mathbf{Nil} : \mathbf{1} \qquad \Gamma \vdash \bot : \rho$$

$$\frac{\Gamma \vdash M : \rho}{\Gamma \vdash \mathbf{Left}(M) : \rho + \sigma} \qquad \frac{\Gamma \vdash M : \sigma}{\Gamma \vdash \mathbf{Right}(M) : \rho + \sigma}$$

$$\frac{\Gamma \vdash M : \rho \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash \mathbf{Pair}(M, N) : \rho \times \sigma} \qquad \frac{\Gamma \vdash M : \rho \times \sigma \qquad \Gamma, a : \rho, b : \sigma \vdash N : \tau}{\Gamma \vdash \mathbf{case}\,M\,\mathbf{of}\,\{\mathbf{Pair}(a, b) \to N\} : \tau}$$

$$\frac{\Gamma \vdash M : \rho + \sigma \qquad \Gamma, a : \rho \vdash L : \tau \qquad \Gamma, b : \sigma \vdash R : \tau}{\Gamma \vdash \mathbf{case}\,M\,\mathbf{of}\,\{\mathbf{Left}(a) \to L\,;\,\mathbf{Right}(b) \to R\} : \tau}$$

$$\frac{\Gamma, a : \rho \vdash M : \sigma}{\Gamma \vdash \lambda a.\,M : \rho \Rightarrow \sigma} \qquad \frac{\Gamma \vdash M : \rho \Rightarrow \sigma \qquad \Gamma \vdash N : \rho}{\Gamma \vdash M\,N : \sigma}$$

$$\frac{\Gamma \vdash M : \rho[\mathbf{fix}\,\alpha\,.\,\rho/\alpha]}{\Gamma \vdash M : \mathbf{fix}\,\alpha\,.\,\rho}\ \mathbf{ROLL} \qquad \frac{\Gamma \vdash M : \mathbf{fix}\,\alpha\,.\,\rho}{\Gamma \vdash M : \rho[\mathbf{fix}\,\alpha\,.\,\rho/\alpha]}\ \mathbf{UNROLL}$$

$$\frac{\Gamma, a : \rho \vdash M\,a : \rho}{\Gamma \vdash \mathbf{rec}\,M : \rho}\ (a \text{ not free in } M)$$

Proof. Immediate from the axioms for types. □

Remark. Only terms typable with these rules will be extracted in Sect. 4.2. Note that the Y-combinator (Sect. 3.2) is not typable by these rules since its type must be of the form $(\rho \Rightarrow \rho) \Rightarrow \rho$, and in order to type the self application $(a\,a)$ occurring in Y one needs a type σ satisfying $\sigma \equiv \sigma \Rightarrow \rho$, that is, a fixed point of a non-positive type operator.

3.5. Translation to Haskell

We sketch how to translate typable RIFP programs into Haskell. First we define a Haskell type $\mathsf{H}(\rho)$ for each type ρ and a sequence of Haskell algebraic data type declarations. We begin with the declaration

$$\mathsf{data\ One} = \mathsf{Nil},$$

and then define

(i) $\mathsf{H}(\mathbf{1}) = \mathsf{One}$
(ii) $\mathsf{H}(\alpha) = \alpha$
(iii) $\mathsf{H}(\rho + \sigma) = \mathsf{Either}\ \mathsf{H}(\rho)\ \mathsf{H}(\sigma)$
(iv) $\mathsf{H}(\rho \times \sigma) = (\mathsf{H}(\rho), \mathsf{H}(\sigma))$
(v) $\mathsf{H}(\rho \Rightarrow \sigma) = \mathsf{H}(\rho) \to \mathsf{H}(\sigma)$
(vi) $\mathsf{H}(\mathbf{fix}\,\alpha\,.\,\rho) = \mathsf{C}_{\alpha,\rho}\,\vec{\beta}$

In case (v), $\to$ is Haskell's function type constructor, in case (vi), $\mathsf{C}_{\alpha,\rho}$ is a new name generated from α and ρ, and $\vec{\beta}$ is a list of the free type variables in $\mathbf{fix}\,\alpha\,.\,\rho$. The list of Haskell data type declarations is extended

by the following recursive and possibly polymorphic data type $\mathsf{C}_{\alpha,\rho}$ with one constructor which we again call $\mathsf{C}_{\alpha,\rho}$.

$$\mathsf{data}\ \mathsf{C}_{\alpha,\rho}\,\vec{\beta} = \mathsf{C}_{\alpha,\rho}\,\mathsf{H}(\rho)[\mathsf{C}_{\alpha,\rho}\,\vec{\beta}/\alpha]$$

To accommodate the typing rules **ROLL** and **UNROLL** we need Haskell programs

$$\begin{aligned} \mathsf{roll}_{\mathsf{C}_{\alpha,\rho}} &:: \mathsf{H}(\rho)[\mathsf{C}_{\alpha,\rho}\,\vec{\beta}/\alpha] \to \mathsf{C}_{\alpha,\rho}\,\vec{\beta} \qquad & \mathsf{roll}_{\mathsf{C}_{\alpha,\rho}}\,\mathsf{x} &= \mathsf{C}_{\alpha,\rho}\,\mathsf{x} \\ \mathsf{unroll}_{\mathsf{C}_{\alpha,\rho}} &:: \mathsf{C}_{\alpha,\rho}\,\vec{\beta} \to \mathsf{H}(\rho)[\mathsf{C}_{\alpha,\rho}\,\vec{\beta}/\alpha] \qquad & \mathsf{unroll}_{\mathsf{C}_{\alpha,\rho}}(\mathsf{C}_{\alpha,\rho}\,\mathsf{x}) &= \mathsf{x} \end{aligned}$$

and for recursive programs a fixed point combinator

$$\mathsf{rec} :: (\alpha \to \alpha) \to \alpha \qquad \mathsf{rec}\,\mathsf{f} = \mathsf{f}\,(\mathsf{rec}\,\mathsf{f})$$

Now suppose that d is a type derivation of $M : \rho$ built from the typing rules in Lemma 13. We define a Haskell program $\mathsf{H}(d)$ of type $\mathsf{H}(\rho)$ as follows. By considering $\mathbf{Pair}(M, N)$ as the Haskell term (M, N), our program is an untyped Haskell program. $\mathsf{H}(d)$ is obtained by inserting appropriate $\mathsf{roll\_}$ and $\mathsf{unroll\_}$ to M following the type derivation d. We do not modify M for rules other than **ROLL** and **UNROLL**. If d ends with **ROLL** with $\rho = \mathbf{fix}\,\alpha\,.\,\rho'$, we define $\mathsf{H}(d) = \mathsf{roll}_{\mathsf{C}_{\alpha,\rho'}}\mathsf{H}(d')$. If d ends with **UNROLL** and $\rho = \rho[\mathbf{fix}\,\alpha\,.\,\rho'/\alpha]$, we define $\mathsf{H}(d) = \mathsf{unroll}_{\mathsf{C}_{\alpha,\rho'}}\mathsf{H}(d')$. Here, d' is the derivation of the premise of **ROLL** and **UNROLL**. With the Haskell program $\mathsf{H}(d)$ obtained in this way, we have a sound derivation of the typing $\mathsf{H}(d) :: \mathsf{H}(\rho)$ in Haskell.

One can optimize this translation in several ways. For example, one can treat a type of the form $\mathbf{fix}\,\alpha\,.\,\rho_1 + \ldots + \rho_k$ so that it is translated to a data type with k constructors. One can also use Haskell's list type for $\mathbf{fix}\,\alpha\,.\,(\tau \times \alpha + \mathbf{1})$ (i.e., finite/infinite list type) and $\mathbf{fix}\,\alpha\,.\,(\tau \times \alpha)$ (i.e., infinite list type).

3.6. Types of IFP *expressions*

We inductively assign to every IFP-expression (i.e., formula or predicate) E a type $\tau(E)$. The idea is that $\tau(A)$, for a formula A, is the type of potential realizers of A. We call an expression *Harrop* if it contains neither disjunctions nor free predicate variables at strictly positive positions. This deviates from the usual definition of the Harrop property [70] since existential quantifiers at strictly positive positions are permitted. The reason for this is that quantifiers are interpreted uniformly, that is, not witnessed by realizers. Like nc formulas, Harrop formulas have no computational content, however, they differ from nc formulas in that they need not coincide with their own realizability interpretation (see Remark 3 at the end of this section).

We define $\tau(E)$ so that the type $\mathbf{1}$ is assigned to an expression iff it is Harrop. In the following definition, we say that a predicate P is *X-Harrop* if $\lambda X\,P$ is Harrop, that is, if P is strictly positive in X and $P[\hat{X}/X]$ is Harrop where $\hat{X}$ is a predicate constant associated with X.

$$\begin{aligned} \tau(P(\vec{t})) &= \tau(P) && \\ \tau(A \vee B) &= \tau(A) + \tau(B) && \\ \tau(A \wedge B) &= \tau(A) \times \tau(B) && (A, B \text{ non-Harrop}) \\ &= \tau(A) && (B \text{ Harrop}, A \text{ non-Harrop}) \\ &= \tau(B) && (A \text{ Harrop}, B \text{ non-Harrop}) \\ &= \mathbf{1} && (A, B \text{ Harrop}) \\ \tau(A \to B) &= \tau(A) \Rightarrow \tau(B) && (A, B \text{ non-Harrop}) \end{aligned}$$

$$
\begin{aligned}
&= \tau(B) && \text{(otherwise)}\\
\tau(\diamond x\, A) &= \tau(A) && (\diamond \in \{\forall, \exists\})\\
\tau(X) &= \alpha_X && (X \text{ a predicate variable, } \alpha_X \text{ a fresh type variable})\\
\tau(P) &= \mathbf{1} && (P \text{ a predicate constant})\\
\tau(\lambda \vec{x}\, A) &= \tau(A)\\
\tau(\diamond(\lambda X\, P)) &= \mathbf{fix}\, \alpha_X\, .\, \tau(P) && (\diamond \in \{\mu, \nu\},\ P \text{ not } X\text{-Harrop})\\
&= \mathbf{1} && (\diamond \in \{\mu, \nu\},\ P\ X\text{-Harrop})
\end{aligned}
$$

Remark. Though the semantics $D^{\zeta}_{\mathbf{1}}$ of the type $\mathbf{1}$ is $\{\mathbf{Nil}, \bot\}$, we will stipulate in Section 3.7 that only **Nil** is a possible realizer of a Harrop expression. We will also define simplified realizers for products and implications if some of their components are Harrop and therefore have corresponding simplified definitions of $\tau(A)$ for these cases. Note that we define the type of a (co)inductively defined Harrop predicate $\diamond(\lambda X\, P)$ to be $\mathbf{1}$. Without this simplified type assignment a non-regular type may be assigned to a predicate, for example, $\tau(\mathbf{False}) = \tau(\mu(\lambda X\, X))$ would become $\mathbf{fix}\, \alpha_X\, .\, \alpha_X$.

Lemma 14. *For every expression E (formula or predicate) and predicate P,*

(a) *E is Harrop if and only if $\tau(E) = \mathbf{1}$,*
(b) *$\tau(E)$ is regular,*
(c) *if P is non-Harrop, then $\tau(E[P/X]) = \tau(E)[\tau(P)/\alpha_X]$,*
(d) *If P is Harrop, then $\tau(E[P/X]) = \tau(E[\hat{X}/X])$.*

Proof. Straightforward structural induction on E. □

3.7. Realizers of expressions

In this section, we define the notion that $a : \tau(A)$ is a *realizer* of a formula A, written $a\, \mathbf{r}\, A$. This intuitively means that a is a computational content of the formula A. In intuitionistic logic, a proof of $A \vee B$ provides evidence that A is true or B is true, together with an indicator of which of the two cases holds. We construct our notion of realizer by treating this as the primitive source of computational content. Therefore, we defined an expression *non-computational* (nc) if it contains neither disjunctions nor free predicate variables (Sect. 2.1). A more general notion of an expression with trivial computational content is provided by the Harrop property which forbids the occurrence of disjunctions and free predicate variables only at strictly positive positions (Sect. 3.6).

In order to formalize realizability in RIFP we define for every IFP formula A an RIFP predicate $\mathbf{R}(A)$ of arity (δ) that specifies the set of domain elements a such that $a\, \mathbf{r}\, A$ holds. For defining $\mathbf{R}(A)$, we simultaneously define $\mathbf{H}(B)$ for Harrop formulas B which expresses that B is realizable, however with trivial computational content **Nil**. We define for every IFP-expression an RIFP-expression, more precisely we define for a

formula A a predicate $\mathbf{R}(A)$ of arity (δ);
non-Harrop predicate P of arity $(\vec{\iota})$ a predicate $\mathbf{R}(P)$ of arity $(\vec{\iota}, \delta)$;
non-Harrop operator Φ of arity $(\vec{\iota})$ an operator $\mathbf{R}(\Phi)$ of arity $(\vec{\iota}, \delta)$;
Harrop formula A a formula $\mathbf{H}(A)$;
Harrop predicate P a predicate $\mathbf{H}(P)$ of the same arity;
Harrop operator Φ an operator $\mathbf{H}(\Phi)$ of the same arity.

In the definition of realizability below we assume that to every IFP predicate variable X of arity $(\vec{\iota})$ there are assigned, in a one-to-one fashion, an RIFP predicate variable $\tilde{X}$ of arity $(\vec{\iota}, \delta)$ and a type variable α_X. Furthermore, we write $a\,\mathbf{r}\,A$ for $\mathbf{R}(A)(a)$ and $\mathbf{r}A$ for $\exists a\, a\,\mathbf{r}\,A$. Recall that a predicate P is *X-Harrop* if it is strictly positive in X and $P[\hat{X}/X]$ is Harrop where $\hat{X}$ is a fresh predicate constant associated with X. In this situation we write $\mathbf{H}_X(P)$ for $\mathbf{H}(P[\hat{X}/X])[X/\hat{X}]$. The idea is that $\mathbf{H}_X(P)$ is the same as $\mathbf{H}(P)$ but considering X as a (non-computational) predicate constant.

$$
\begin{aligned}
a\,\mathbf{r}\,A &= (a = \mathbf{Nil} \wedge \mathbf{H}(A)) && (A \text{ Harrop})\\
a\,\mathbf{r}\,P(\vec{t}) &= \mathbf{R}(P)(\vec{t}, a) && (P \text{ non-H.})\\
c\,\mathbf{r}\,(A \vee B) &= \exists a\,(c = \mathbf{Left}(a) \wedge a\,\mathbf{r}\,A) \vee \exists b\,(c = \mathbf{Right}(b) \wedge b\,\mathbf{r}\,B)\\
c\,\mathbf{r}\,(A \wedge B) &= \exists a, b\,(c = \mathbf{Pair}(a, b) \wedge a\,\mathbf{r}\,A \wedge b\,\mathbf{r}\,B) && (A, B \text{ non-H.})\\
a\,\mathbf{r}\,(A \wedge B) &= a\,\mathbf{r}\,A \wedge \mathbf{H}(B) && (B \text{ Harrop}, A \text{ non-H.})\\
b\,\mathbf{r}\,(A \wedge B) &= \mathbf{H}(A) \wedge b\,\mathbf{r}\,B && (A \text{ Harrop}, B \text{ non-H.})\\
c\,\mathbf{r}\,(A \to B) &= c : \tau(A) \Rightarrow \tau(B) \wedge \forall a\,(a\,\mathbf{r}\,A \to (c\,a)\,\mathbf{r}\,B) && (A, B \text{ non-H.})\\
b\,\mathbf{r}\,(A \to B) &= b : \tau(B) \wedge (\mathbf{H}(A) \to b\,\mathbf{r}\,B) && (A \text{ Harrop}, B \text{ non-H.})\\
a\,\mathbf{r}\,\diamond x\,A &= \diamond x\,(a\,\mathbf{r}\,A) && (\diamond \in \{\forall, \exists\}, A \text{ non-H.})\\
\mathbf{R}(X) &= \tilde{X}\\
\mathbf{R}(\lambda \vec{x}\,A) &= \lambda(\vec{x}, a)\,(a\,\mathbf{r}\,A) && (A \text{ non-H.})\\
\mathbf{R}(\Box(\Phi)) &= \Box(\mathbf{R}(\Phi)) && (\Box \in \{\mu, \nu\}, \Phi \text{ non-H.})\\
\mathbf{R}(\lambda X\,P) &= \lambda \tilde{X}\,\mathbf{R}(P) && (P \text{ non-H.})
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{H}(P(\vec{t})) &= \mathbf{H}(P)(\vec{t}) && (P \text{ Harrop})\\
\mathbf{H}(A \wedge B) &= \mathbf{H}(A) \wedge \mathbf{H}(B) && (A, B \text{ Harrop})\\
\mathbf{H}(A \to B) &= \mathbf{r}A \to \mathbf{H}(B) && (B \text{ Harrop})\\
\mathbf{H}(\diamond x\,A) &= \diamond x\,\mathbf{H}(A) && (\diamond \in \{\forall, \exists\}, A \text{ Harrop})
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{H}(P) &= P && (P \text{ a predicate constant})\\
\mathbf{H}(\lambda \vec{x}\,A) &= \lambda \vec{x}\,\mathbf{H}(A) && (A \text{ Harrop})\\
\mathbf{H}(\Box(\Phi)) &= \Box(\mathbf{H}(\Phi)) && (\Box \in \{\mu, \nu\}, \Phi \text{ Harrop})\\
\mathbf{H}(\lambda X\,P) &= \lambda X\,\mathbf{H}_X(P) && (P\ X\text{-Harrop})
\end{aligned}
$$

In order to see that $\mathbf{R}(\Box(\Phi))$ and $\mathbf{H}(\Box(\Phi))$ are wellformed one needs to prove simultaneously that if an expression E is s.p. in X, then $\mathbf{R}(E)$ is s.p. in $\tilde{X}$, and if E is X-Harrop, then $\mathbf{H}_X(E)$ $(= \mathbf{H}(E[\hat{X}/X]))$ is s.p. in $\hat{X}$.

Lemma 15.

(a) *If P is non-Harrop, then* $\mathbf{R}(A[P/X]) = \mathbf{R}(A)[\mathbf{R}(P)/\tilde{X}][\tau(P)/\alpha_X]$.
(b) *If P is Harrop, then* $\mathbf{R}(A[P/X]) = \mathbf{R}(A[\hat{X}/X])[\mathbf{H}(P)/\hat{X}]$.
(c) *If A is Harrop, then* $\mathbf{H}(A) \leftrightarrow \mathbf{r}A$.
(d) *If E is an nc expression, then* $\mathbf{H}(E) = E$, *in particular,* $\mathbf{H}(\mathbf{False}) = \mathbf{False}$.

(e) $\mathbf{R}(A) \subseteq \tau(A)$ *under the assumptions* $\forall \vec{x}\,(\mathbf{R}(\tilde{X}(\vec{x})) \subseteq \alpha_X)$, *that is,* $\forall \vec{x}, b\,(\tilde{X}(\vec{x}, b) \to b : \alpha_X)$, *for all free predicate variables* X *in* A.

Proof. The statements are proven by induction on the size of expressions suitably generalizing the statements to formulas or predicates. Parts (a-d) are easy.

For (e) one proves, simultaneously with the statement for formulas, that for predicates P, $a\,\mathbf{r}\,P(\vec{x})$ implies $a : \tau(P)$ assuming $\forall \vec{x}, b\,(\tilde{Y}(\vec{x}, b) \to b : \alpha_Y)$ for all free predicate variables Y in P. The only difficult case is a non-Harrop predicate P of the form $\Box(\lambda X\, Q)$ ($\Box \in \{\mu, \nu\}$). In that case $\tau(P) = \mathbf{fix}\,\alpha_X\,.\,\tau(Q)$ and by Lemma 14 (b) this is a regular type. Furthermore, $\mathbf{R}(P) = \Box(\lambda \tilde{X}\,\mathbf{R}(Q)) \equiv \mathbf{R}(Q)[\mathbf{R}(P)/\tilde{X}]$. By the induction hypothesis, $\forall a, \vec{x}\,(a\,\mathbf{r}\,Q(\vec{x}) \to a : \tau(Q))$ under the extra assumption $\forall \vec{x}, b\,(\tilde{X}(\vec{x}, b) \to b : \alpha_X)$. Let α be the subdomain generated by the set $\{a \in D \mid \exists \vec{x}\,(a\,\mathbf{r}\,P(\vec{x}))\}$ whose existence is guaranteed by Axiom (xvi). It suffices to show $\alpha \subseteq \tau(Q)[\alpha/\alpha_X]$ since then, by Axiom (xi), $\alpha \subseteq \mathbf{fix}\,\alpha_X\,.\,\tau(Q) = \tau(P)$ and consequently if $a\,\mathbf{r}\,P(\vec{x})$, then $a : \alpha$ and therefore $a : \tau(P)$. For the proof of $\alpha \subseteq \tau(Q)[\alpha/\alpha_X]$ it suffices to show that if $a\,\mathbf{r}\,P(\vec{x})$, then $a : \tau(Q)[\alpha/\alpha_X]$. Assume $a\,\mathbf{r}\,P(\vec{x})$. Then $(a\,\mathbf{r}\,Q(\vec{x}))[\mathbf{R}(P)/\tilde{X}]$ since $\mathbf{R}(P) \equiv \mathbf{R}(Q)[\mathbf{R}(P)/\tilde{X}]$. Using the induction hypothesis with $\tilde{X} \stackrel{\text{Def}}{=} \mathbf{R}(P)$ and $\alpha_X \stackrel{\text{Def}}{=} \alpha$ and we get $a : \tau(Q)[\alpha/\alpha_X]$ as required. □

Remarks. 1. Since **Nil** is the only possible realizer of a Harrop formula, one could as well define **1** as $\{\bot\}$ and use $\bot$ as the realizer of a realizable Harrop formula. Then, the domain $D_{\mathbf{nat}}$ for $\mathbf{nat} \stackrel{\text{Def}}{=} \tau(\mathbf{N})$ (see Sect. 4.3) would be isomorphic to the domain of lazy natural numbers, and the domain $D_{\mathbf{2}}$ for $\mathbf{2} \stackrel{\text{Def}}{=} \mathbf{1} + \mathbf{1}$ would be isomorphic to the domain of truth values $\{\mathbf{true}, \mathbf{false}, \bot\}$ (see Sect. 5.3). However, using $\bot$ as a realizer of Harrop formulas contradicts our intuitive understanding that $\bot$ means non-termination. One could as well obtain these isomorphisms without modifying the realizer of a Harrop formula by adding nullary constructors $\mathbf{Left}_0$ (representing $\mathbf{Left}(\mathbf{Nil})$) and $\mathbf{Right}_0$ (representing $\mathbf{Right}(\mathbf{Nil})$) to D and corresponding constructors to type expressions. However, we refrain from these additions since their comparably small benefits would not match the considerable complications they would create.

2. While $a\,\mathbf{r}\,(\forall x\, A) \equiv \forall x\,(a\,\mathbf{r}\,A)$ holds, $\mathbf{r}(\forall x\, A) \equiv \forall x\,\mathbf{r}\,A$ does not hold in general since $\mathbf{r}(\forall x\, A) = \exists a\, a\,\mathbf{r}\,(\forall x\, A) = \exists a\, \forall x\, a\,\mathbf{r}\,A$ whereas $\forall x\,\mathbf{r}\,A = \forall x\, \exists a\, a\,\mathbf{r}\,A$.

3. Regarding (c) vs. (d) we note that for Harrop formulas A, $\mathbf{H}(A)$ need not be equivalent to A. In fact, A and $\mathbf{H}(A)$ may even contradict each other. For example, if A is the Harrop formula $\neg\forall x\,(x = 0 \vee x \neq 0)$, then $\mathbf{H}(A)$ is $\neg\exists a\, \forall x\,(a = \mathbf{Left}(\mathbf{Nil}) \wedge x = 0 \vee a = \mathbf{Right}(\mathbf{Nil}) \wedge x \neq 0)$ which is intuitionistically provable from $0 \neq 1$. On the other hand $\neg A$ is provable in classical logic. Hence, $\mathbf{r}A \to A$ is classically contradictory and therefore unprovable in RIFP. The reason for this difference between A and $\mathbf{r}A$ is *logical*, more precisely it lies in the uniform interpretation of the universal quantifier which forbids a realizer of a formula $\forall x\, B$ to depend on x. In contrast, in Kleene realizability the main source of discrepancy between realizability and truth is *computational* and follows from the existence of undecidable predicates. For example, the formula $C \stackrel{\text{Def}}{=} \forall x \in \mathbf{N}(\mathrm{Halt}(x) \vee \neg\mathrm{Halt}(x))$ is classically true but not realizable since any realizer, which in Kleene realizability has to be computable, would solve the halting problem (and hence $\neg C$ is classically false but realizable). In our setting C is realizable since the domain D admits non-computable functions.

4. A crucial property of our realizability interpretation is that $\bot$ can be a realizer of a formula. For example, $a\,\mathbf{r}\,(\mathbf{False} \to A)$ for any $a : \tau(A)$. In particular, $\bot\,\mathbf{r}\,(\mathbf{False} \to A)$ for any non-Harrop formula A. This enables us to manipulate non-terminating computation in logic and extract non-terminating programs from logical proofs. On the other hand, $\bot\,\mathbf{r}\,A$ does not hold if A is a Harrop formula.

5. Although, by Lemma 15 (e), realizers are typable, they may be partial as remarked above. Therefore our realizability is closer to Kleene's realizability by (codes of) partial recursive functions [40], rather than Kreisel's modified realizability [44] whose characteristic feature is that realizers are typed and *total*. For example, it is easy to see that the schema *Independence of Premise*, $(A \to \exists x \in \mathbf{N}\, B) \to \exists x \in \mathbf{N}\,(A \to B)$ where A is a Harrop formula, which is realizable in modified realizability, is not realizable in our system.

6. Despite the availability of classical logic through disjunction-free axioms our interpretation is very different from Krivine's classical realizability [45,46]. While our interpretation fundamentally rests on the intuitionistic interpretation of disjunction as a problem whose solution requires a decision between two alternatives, Krivine's classical realizability is formulated in the negative fragment of logic given with implication, conjunction and universal quantification as the only logical connectives. In [56] it is shown that Krivine's realizability (roughly) corresponds to Gödel's negative translation followed by intuitionistic realizability.

4. Soundness

The Soundness Theorem stating that provable formulas are realizable is the theoretical foundation for program extraction.

Theorem 2 *(Soundness). Let $\mathcal{A}$ be a set of nc axioms. From an* $\mathrm{IFP}(\mathcal{A})$ *proof of a formula A one can extract a closed program $M : \tau(A)$ such that $M\,\mathbf{r}\,A$ is provable in* $\mathrm{RIFP}(\mathcal{A})$.

More generally, let Γ be a set of Harrop formulas and Δ a set of non-Harrop formulas. Then, from an $\mathrm{IFP}(\mathcal{A})$ *proof of a formula A from the assumptions Γ, Δ one can extract a program M with $\mathrm{FV}(M) \subseteq \vec{u}$ such that $\vec{u} : \tau(\Delta) \vdash M : \tau(A)$ and $M\,\mathbf{r}\,A$ are provable in* $\mathrm{RIFP}(\mathcal{A})$ *from the assumptions $\mathbf{H}(\Gamma)$ and $\vec{u}\,\mathbf{r}\,\Delta$.*

Moreover, all typing judgements above are derivable by the rules of Lemma 13.

In this Section we prove this theorem (Sect. 4.1) and read off from it a program extraction procedure for IFP-proofs (Sect. 4.2). We also study the realizers of natural numbers (Sect. 4.3) and wellfounded induction (Sect. 4.4).

Remarks. 1. From the general version of the Soundness Theorem one sees that Harrop formulas B can be freely used as assumptions (or axioms) as long as their Harrop interpretations $\mathbf{H}(B)$ are true. For example, $\mathbf{BT}_\prec$ (Brouwer's Thesis, defined in Sect. 2.2) is a Harrop formula (for an arbitrary relation $\prec$) and one can show that $\mathbf{H}(\mathbf{BT}_\prec)$ is equivalent to $\mathbf{BT}_{\mathbf{r}\prec}$ and hence true. Therefore, $\mathbf{BT}_\prec$ (without restriction on the relation $\prec$) can be used as an axiom in a proof without spoiling program extraction.

2. Since $\mathrm{RIFP}(\mathcal{A})$ is an instance of IFP it follows from the Tarskian soundness of IFP (see Sect. 2.1) that the statements $M : \tau(A)$ and $M\,\mathbf{r}\,A$ in the Soundness Theorem are true, in particular M denotes indeed a realizer of A.

4.1. Proof of the soundness theorem

The expected proof of the Soundness Theorem by structural induction on IFP derivations faces the obstacle that in order to prove realizability of s.p. induction and coinduction one needs realizers for the monotonicity of the operators in question, and this, in turn, requires the realizability of s.p. induction and coinduction. We escape this circularity by introducing an equivalent system IFP' for which soundness can be proven by induction on the length of derivations. The only difference between the two systems is that IFP' requires a monotonicity proof for the operator as an additional premise of s.p. induction and coinduction.

Let $\mathrm{Mon}(\Phi)$ be the following formula expressing the monotonicity of Φ:

$$\mathrm{Mon}(\Phi) \stackrel{\mathrm{Def}}{=} X \subseteq Y \to \Phi(X) \subseteq \Phi(Y)$$

where X and Y are fresh predicate variables. The system IFP' is obtained from IFP by replacing the rules $\mathbf{IND}(\Phi, P)$ and $\mathbf{COIND}(\Phi, P)$ by

$$\frac{\Phi(P) \subseteq P \quad \mathrm{Mon}(\Phi)}{\mu(\Phi) \subseteq P}\ \mathbf{IND}'(\Phi, P) \quad (*)$$

$$\frac{P \subseteq \Phi(P) \quad \mathrm{Mon}(\Phi)}{P \subseteq \nu(\Phi)}\ \mathbf{COIND}'(\Phi, P) \quad (*)$$

$(*)$ is the side condition that the free assumptions in the proof of $\mathrm{Mon}(\Phi)$ must not contain X or Y free.

The modified rules $\mathbf{SI}'(\Phi, P)$, $\mathbf{HSI}'(\Phi, P)$, $\mathbf{SCI}'(\Phi, P)$, $\mathbf{HSCI}'(\Phi, P)$, are defined similarly.

By the *length of a derivation* we mean the number of occurrences of derivation rules.

Lemma 16. *If* IFP, IFP′, *or* RIFP *proves* $\Gamma \vdash A$, *then the same system proves* $\Gamma[P/X] \vdash A[P/X]$, $\Gamma[P/\hat{X}] \vdash A[P/\hat{X}]$ *and, if applicable,* $\Gamma[\rho/\alpha] \vdash A[\rho/\alpha]$, *with the same derivation length, where* A, P, X, ρ, α *are arbitrary formulas, predicates, predicate variables, types, type variables respectively, and* $\hat{X}$ *is an arbitrary predicate constant that does not appear in any axiom.*

Proof. Easy structural induction on derivations. □

Remark. Important instances of Lemma 16 are derivations of $\mathrm{Mon}(\Phi)$, which occur as premises of the rules $\mathbf{IND}'$ and $\mathbf{COIND}'$. If we replace in $\mathrm{Mon}(\Phi)$ one or both of the predicate variables X and Y by different fresh predicate constants, say $\hat{X}$ and $\hat{Y}$, then, by Lemma 16, the resulting formulas have derivations of the same length. This fact will be used in the soundness proof for IFP′ (Theorem 3).

Lemma 17.

(a) *If* RIFP *proves* $a\,\mathbf{r}\,A$ *from assumptions that do not contain the predicate variable* X *and if* P *is a non-Harrop predicate of the same arity as* X, *then* RIFP *proves* $a\,\mathbf{r}\,(A[P/X])$ *from the same assumptions.*

(b) *If* RIFP *proves* $a\,\mathbf{r}\,(A[\hat{X}/X])$ *from assumptions that do not contain the predicate constant* $\hat{X}$ *and if* P *is a Harrop predicate of the same arity as* X, *then* RIFP *proves* $a\,\mathbf{r}\,(A[P/X])$ *from the same assumptions.*

Proof. From $a\,\mathbf{r}\,A$ we get, by Lemma 16, $(a\,\mathbf{r}\,A)[\mathbf{R}(P)/\tilde{X}][\tau(P)/\alpha_X]$ which, by Lemma 15 (a), is the same as $a\,\mathbf{r}\,(A[P/X])$ provided P is non-Harrop.

From $a\,\mathbf{r}\,(A[\hat{X}/X])$ we get, by Lemma 16, $(a\,\mathbf{r}\,(A[\hat{X}/X]))[\mathbf{H}(P)/\hat{X}]$ which, by Lemma 15 (b), is the same as $a\,\mathbf{r}\,(A[P/X])$ provided P is Harrop. □

In Theorem 3 we will use the following monotone predicate transformers:

$$\begin{array}{llll}
(f^{-1} \circ Q)(\vec{x}, a) \stackrel{\mathrm{Def}}{=} Q(\vec{x}, f\,a) & (f \circ Q)(\vec{x}, b) \stackrel{\mathrm{Def}}{=} \exists a\,(f\,a = b \wedge Q(\vec{x}, a)) \\
(a^{-1} * Q)(\vec{x}) \stackrel{\mathrm{Def}}{=} Q(\vec{x}, a) & (a * P)(\vec{x}, b) \stackrel{\mathrm{Def}}{=} a = b \wedge P(\vec{x}) \\
\Delta(P)(\vec{x}, b) \stackrel{\mathrm{Def}}{=} P(\vec{x}) & \exists(Q)(\vec{x}) \stackrel{\mathrm{Def}}{=} \exists a\, Q(\vec{x}, a)
\end{array}$$

The next lemma states their relevant properties. We omit the easy proofs.

Lemma 18. Equivalences.

$$\begin{array}{ll}
f^{-1} \circ (g^{-1} \circ Q) \equiv (g \circ f)^{-} \circ Q & f \circ (g \circ Q) \equiv (f \circ g) \circ Q \\
a^{-1} * (f^{-1} \circ Q) \equiv (f\,a)^{-1} * Q & f \circ (a * P) \equiv (f\,a) * P \\
f^{-1} \circ \Delta(P) \equiv \Delta(P) & \exists(f \circ Q) \equiv \exists(Q)
\end{array}$$

$$f^{-1} \circ P \cap g^{-1} \circ Q \equiv \langle f, g\rangle^{-1} \circ ({\pi_{\mathbf{Left}}}^{-1} \circ P \cap {\pi_{\mathbf{Right}}}^{-1} \circ Q)$$

$$f \circ P \cup g \circ Q \equiv [f + g] \circ (\mathbf{Left} \circ P \cup \mathbf{Right} \circ Q)$$

Adjunctions.

$$\begin{array}{lcl} Q \subseteq f^{-1} \circ Q' & \leftrightarrow & f \circ Q \subseteq Q' \\ P \subseteq a^{-1} * Q & \leftrightarrow & a * P \subseteq Q \\ Q \subseteq \Delta(P) & \leftrightarrow & \exists(Q) \subseteq P \end{array}$$

Realizability. *Below let Q, Q' be non-Harrop predicates, P, P' Harrop predicates, $f : \tau(Q) \Rightarrow \tau(Q')$ and $a : \tau(Q)$:*

$$\begin{array}{lcllcl} f\,\mathbf{r}\,(Q \subseteq Q') & \leftrightarrow & \mathbf{R}(Q) \subseteq f^{-1} \circ \mathbf{R}(Q') & \leftrightarrow & f \circ \mathbf{R}(Q) \subseteq \mathbf{R}(Q') \\ a\,\mathbf{r}\,(P \subseteq Q) & \leftrightarrow & \mathbf{H}(P) \subseteq a^{-1} * \mathbf{R}(Q) & \leftrightarrow & a * \mathbf{H}(P) \subseteq \mathbf{R}(Q) \\ \mathbf{H}(Q \subseteq P) & \leftrightarrow & \mathbf{R}(Q) \subseteq \Delta(\mathbf{H}(P)) & \leftrightarrow & \exists(\mathbf{R}(Q)) \subseteq \mathbf{H}(P) \\ \mathbf{H}(P \subseteq P') & \leftrightarrow & \mathbf{H}(P) \subseteq \mathbf{H}(P') & & \end{array}$$

$$\mathbf{R}(Q \cap Q') \equiv {\pi_{\mathbf{Left}}}^{-1} \circ \mathbf{R}(Q) \cap {\pi_{\mathbf{Right}}}^{-1} \circ \mathbf{R}(Q')$$

$$\mathbf{R}(Q \cup Q') \equiv \mathbf{Left} \circ \mathbf{R}(Q) \cup \mathbf{Right} \circ \mathbf{R}(Q')$$

Theorem 3 *(IFP′ version of Soundness). Let $\mathcal{A}$ be a set of nc axioms. From an $\mathrm{IFP}'(\mathcal{A})$ proof of a formula A one can extract a closed program $M : \tau(A)$ such that $M\,\mathbf{r}\,A$ is provable in $\mathrm{RIFP}(\mathcal{A})$.*

More generally, let Γ be a set of Harrop formulas and Δ a set of non-Harrop formulas. Then, from an $\mathrm{IFP}'(\mathcal{A})$ proof of a formula A from the assumptions Γ, Δ one can extract a program M with $\mathrm{FV}(M) \subseteq \vec{u}$ such that $\vec{u} : \tau(\Delta) \vdash M : \tau(A)$ and $M\,\mathbf{r}\,A$ are provable in $\mathrm{RIFP}(\mathcal{A})$ from the assumptions $\mathbf{H}(\Gamma)$ and $\vec{u}\,\mathbf{r}\,\Delta$.

Moreover, all typing judgements above are derivable by the rules of Lemma 13.

Proof. By induction on the length of IFP′ derivations.

In the following we mean by 'induction hypothesis' always an induction hypothesis of the induction on the length of derivations. In order to avoid confusion with IFP′ induction on a strictly positive inductive predicate $\mu(\Phi)$, we will refer to the latter always as 's.p. induction'.

The logical rules are straightforward. We only look at one case, to highlight the difference to other forms of realizability. In Sect. 4.2 the extracted programs for all logical rules are shown.

Existence elimination. Assume we have derivations of $\exists x\,A$ and $\forall x\,(A \to B)$ where x is not free in B. We need a realizer of B. By the first induction hypothesis we have realizers d of $\exists x\,A$, that is, d realizes $A[y/x]$ for some y (in the case that A is Harrop, this means that $d = \mathbf{Nil}$ and $\mathbf{H}(A[y/x])$ holds). Consider the case that B is non-Harrop. Then the second induction hypothesis yields a realizer e of $\forall x\,(A \to B)$, that is, e realizes $A[z/x] \to B$ for all z. If A is Harrop, the latter means that e realizes B provided $\mathbf{H}(A[z/x])$ holds for some z. But $\mathbf{H}(A[y/x])$ holds. Hence e realizes B. If A is non-Harrop then $e\,f$ realizes B for all f that realize $A[z/x]$ for some z. Since d realizes $A[y/x]$, it follows that $e\,d$ realizes B. If B is Harrop, then the second induction hypothesis means that $\mathbf{H}(B)$ holds provided $A[z/x]$ is realizable for some z. Since $A[y/x]$ is realizable, it follows $\mathbf{H}(B)$ which means that $\mathbf{Nil}$ realizes B.

For s.p. induction and s.p. coinduction we consider an operator $\Phi = \lambda X\,Q$, hence $\Phi(P) = Q[P/X]$.

$\mathbf{IND}'(\Phi, P)$. Assume we have derived $\mu(\Phi) \subseteq P$ by s.p. induction from $\Phi(P) \subseteq P$, that is, $Q[P/X] \subseteq P$, and $\mathrm{Mon}(\Phi)$, that is, $X \subseteq Y \to Q \subseteq Q[Y/X]$. It is our goal to find a realizer of $\mu(\Phi) \subseteq P$.

(a) Case Φ and P are both non-Harrop. We want $\tilde{f} : (\mathbf{fix}\,\alpha_X\,.\,\tau(Q)) \Rightarrow \tau(P)$ realizing $\mu(\Phi) \subseteq P$, that is, by Lemma 18, $\mu(\lambda\tilde{X}\,\mathbf{R}(Q)) \subseteq \tilde{f}^{-1} \circ \mathbf{R}(P)$. We attempt to prove this by s.p. induction, so our goal is to prove

$$\mathbf{R}(Q)[\tilde{f}^{-1} \circ \mathbf{R}(P)/\tilde{X}] \subseteq \tilde{f}^{-1} \circ \mathbf{R}(P)\,.$$

By the induction hypothesis we have $s : \tau(\Phi(P)) \Rightarrow \tau(P)$ such that $s\,\mathbf{r}\,(\Phi(P) \subseteq P)$, that is, by Lemma 18,

$$\mathbf{R}(Q[P/X]) \subseteq s^{-1} \circ \mathbf{R}(P), \tag{1}$$

and also some $m : (\alpha_X \Rightarrow \alpha_Y) \Rightarrow \tau(Q) \Rightarrow \tau(Q)[\alpha_Y/\alpha_X]$ such that $m\,\mathbf{r}\,(\mathrm{Mon}(\Phi))$ and hence also $m\,\mathbf{r}\,(\mathrm{Mon}(\Phi)[P/Y])$, by Lemma 17 (a). Therefore, $\forall f\,(\tilde{X} \subseteq f^{-1} \circ \mathbf{R}(P) \to \mathbf{R}(Q) \subseteq (m\,f)^{-1} \circ \mathbf{R}(Q[P/X]))$. Using this with $f \stackrel{\mathrm{Def}}{=} \tilde{f}$, where $\tilde{f}$ is yet unknown, and $\tilde{X} \stackrel{\mathrm{Def}}{=} \tilde{f}^{-1} \circ \mathbf{R}(P)$ we obtain, using Lemma 16 for RIFP′,

$$\mathbf{R}(Q)[\tilde{f}^{-1} \circ \mathbf{R}(P)/\tilde{X}] \subseteq (m\,\tilde{f})^{-1} \circ \mathbf{R}(Q[P/X]) \tag{2}$$

Now,

$$\begin{aligned}\mathbf{R}(Q)[\tilde{f}^{-1} \circ \mathbf{R}(P)/\tilde{X}] &\stackrel{(2)}{\subseteq} (m\,\tilde{f})^{-1} \circ \mathbf{R}(Q[P/X]) \\ &\stackrel{(1)}{\subseteq} (m\,\tilde{f})^{-1} \circ (s^{-1} \circ \mathbf{R}(P)) \\ &= (s \circ m\,\tilde{f})^{-1} \circ \mathbf{R}(P)\end{aligned}$$

Therefore we define recursively $\tilde{f} \stackrel{\mathrm{rec}}{=} s \circ m\,\tilde{f}$ and are done. Clearly, $\tilde{f}$ has the right type.

(b) Case Φ and P are both Harrop (then $\mu(\Phi)$ and $Q[P/X]$ are Harrop). We aim to prove $\mathbf{H}(\mu(\Phi) \subseteq P)$, that is, $\mu(\lambda X\,\mathbf{H}_X(Q)) \subseteq \mathbf{H}(P)$. We try s.p. induction, so our goal is to prove

$$\mathbf{H}_X(Q)[\mathbf{H}(P)/X] \subseteq \mathbf{H}(P).$$

By the induction hypothesis we have $\mathbf{H}(\Phi(P) \subseteq P)$, that is,

$$\mathbf{H}(Q[P/X]) \subseteq \mathbf{H}(P), \tag{3}$$

and $\mathbf{H}(\mathrm{Mon}(\Phi)[\hat{X}/X][\hat{Y}/Y])$ (see the remark after Lemma 16). The latter yields, by Lemma 17 (b), the derivability of $\mathbf{H}(\mathrm{Mon}(\Phi)[\hat{X}/X][P/Y])$, that is, $\hat{X} \subseteq \mathbf{H}(P) \to \mathbf{H}(Q[\hat{X}/X]) \subseteq \mathbf{H}(Q[P/X])$. Using Lemma 16 for RIFP with $\hat{X} \stackrel{\mathrm{Def}}{=} \mathbf{H}(P)$ we obtain $\mathbf{H}(Q[\hat{X}/X])[\mathbf{H}(P)/\hat{X}] \subseteq \mathbf{H}(Q[P/X])$ which is the same as

$$\mathbf{H}_X(Q)[\mathbf{H}(P)/X] \subseteq \mathbf{H}(Q[P/X]) \tag{4}$$

since $\mathbf{H}(Q[\hat{X}/X])[\mathbf{H}(P)/\hat{X}] = \mathbf{H}_X(Q)[\mathbf{H}(P)/X]$. (4) and (3) yield the desired inclusion $\mathbf{H}_X(Q)[\mathbf{H}(P)/X] \subseteq \mathbf{H}(P)$.

(c) Case Φ is non-Harrop, P is Harrop (then $\mu(\Phi)$ and $Q[P/X]$ are non-Harrop). We aim to prove $\mathbf{H}(\mu(\Phi) \subseteq P)$, which, by Lemma 18, is equivalent to $\mu(\lambda\tilde{X}\,\mathbf{R}(Q)) \subseteq \Delta(\mathbf{H}(P))$. Trying s.p. induction, our goal is to prove

$$\mathbf{R}(Q)[\Delta(\mathbf{H}(P))/\tilde{X}] \subseteq \Delta(\mathbf{H}(P)).$$

By the induction hypothesis we have $\mathbf{H}(\Phi(P) \subseteq P)$, that is,

$$\mathbf{R}(Q[P/X]) \subseteq \Delta(\mathbf{H}(P))), \tag{5}$$

and $m : \tau(Q) \Rightarrow \tau(Q[\hat{Y}/Y])$ s.t. $m\,\mathbf{r}\,(\mathrm{Mon}(\Phi)[\hat{Y}/Y])$. Hence by Lemma 17 (b), $m\,\mathbf{r}\,(\mathrm{Mon}(\Phi)[P/Y])$ that is, $\tilde{X} \subseteq \Delta(\mathbf{H}(P)) \to \mathbf{R}(Q) \subseteq m^{-1} \circ \mathbf{R}(Q[P/X])$. Using this with $\tilde{X} \stackrel{\mathrm{Def}}{=} \Delta(\mathbf{H}(P))$ we obtain

$$\mathbf{R}(Q)[\Delta(\mathbf{H}(P))/\tilde{X}] \subseteq m^{-1} \circ \mathbf{R}(Q[P/X]). \tag{6}$$

Now,

$$\begin{aligned}\mathbf{R}(Q)[\Delta(\mathbf{H}(P))/\tilde{X}] &\overset{(6)}{\subseteq} m^{-1} \circ \mathbf{R}(Q[P/X]) \\ &\overset{(5)}{\subseteq} m^{-1} \circ \Delta(\mathbf{H}(P)) \\ &= \Delta(\mathbf{H}(P))\end{aligned}$$

(d) Case Φ is Harrop, P is non-Harrop.

Subcase X is not free in Q. The goal to find a realizer $\tilde{a} : \tau(P)$ of $\mu(\Phi) \subseteq P$ can be written as $\mu(\lambda X\, \mathbf{H}(Q)) \subseteq \tilde{a}^{-1} \circ \mathbf{R}(P)$ whose s.p. inductive proof, in this case, boils down to proving $\mathbf{H}(Q) \subseteq \tilde{a}^{-1} \circ \mathbf{R}(P)$. But such an $\tilde{a}$ is provided by the induction hypothesis as a realizer of $\Phi(P) \subseteq P$.

Subcase X is free in Q (then Q, $Q[P/X]$ and $\mathrm{Mon}(\Phi)[\hat{X}/X][P/Y]$ are non-Harrop). We need to find $\tilde{a} : \tau(P)$ such that $\tilde{a}\,\mathbf{r}\,(\mu(\Phi) \subseteq P)$, which is equivalent to $\mu(\lambda X\, \mathbf{H}_X(Q)) \subseteq \tilde{a}^{-1} * \mathbf{R}(P)$. A proof attempt by s.p. induction leads to the goal

$$\mathbf{H}_X(Q)[\tilde{a}^{-1} * \mathbf{R}(P)/X] \subseteq \tilde{a}^{-1} * \mathbf{R}(P).$$

By the induction hypothesis we have $s : \tau(\Phi(P)) \Rightarrow \tau(P)$ such that $s\,\mathbf{r}\,(\Phi(P) \subseteq P)$, equivalently,

$$\mathbf{R}(Q[P/X]) \subseteq s^{-1} \circ \mathbf{R}(P), \tag{7}$$

and some $m : \alpha_Y \Rightarrow \tau(Q[\hat{X}/X])$ such that $m\,\mathbf{r}\,(\mathrm{Mon}(\Phi)[\hat{X}/X])$ and hence, by Lemma 17 (a), also $m\,\mathbf{r}\,(\mathrm{Mon}(\Phi)[\hat{X}/X][P/Y])$, that is, by Lemma 18,

$$\forall a\,(\hat{X} \subseteq a^{-1} * \mathbf{R}(P) \to \mathbf{H}_X(Q)[\hat{X}/X] \subseteq (m\,a)^{-1} * \mathbf{R}(Q[P/X]).$$

Using this with $a \overset{\mathrm{Def}}{=} \tilde{a}$ (yet unknown) and $\hat{X} \overset{\mathrm{Def}}{=} \tilde{a}^{-1} * \mathbf{R}(P)$ we obtain

$$\mathbf{H}_X(Q)[\tilde{a}^{-1} * \mathbf{R}(P)/X] \subseteq (m\,a)^{-1} * \mathbf{R}(Q[P/X]) \tag{8}$$

Now,

$$\begin{aligned}\mathbf{H}_X(Q)[\tilde{a}^{-1} * \mathbf{R}(P)/X] &\overset{(8)}{\subseteq} (m\,\tilde{a})^{-1} * \mathbf{R}(Q[P/X]) \\ &\overset{(7)}{\subseteq} (m\,\tilde{a})^{-1} * (s^{-1} \circ \mathbf{R}(P)) \\ &= (s\,(m\,\tilde{a}))^{-1} * \mathbf{R}(P)\end{aligned}$$

Hence, the recursive definition $\tilde{a} \overset{\mathrm{rec}}{=} s\,(m\,\tilde{a})$ provides a solution. Clearly, $\tilde{a}$ has the right type $\tau(P)$.

COIND′(Φ, P). Assume we have derived $P \subseteq \nu(\Phi)$ by s.p. coinduction from $P \subseteq \Phi(P)$, i.e. $P \subseteq Q[P/X]$, and $\mathrm{Mon}(\Phi)$. It is our goal to find a realizer of $P \subseteq \nu(\Phi)$.

(a) Case Φ and P are both non-Harrop. Dual to case (a) for **IND′**.

(b) Case Φ and P are both Harrop. Dual to case (b) for **IND′**.

(c) Case Φ is non-Harrop, P is Harrop (then $\nu(\Phi)$, $Q[P/X]$ and $Q[Y/X]$ are non-Harrop). We aim to prove $\tilde{a}\,\mathbf{r}\,(P \subseteq \nu(\Phi))$, for suitable $\tilde{a} : \mathbf{fix}\,\alpha_X\,.\,\tau(Q)$, which is equivalent to $\tilde{a} * \mathbf{H}(P) \subseteq \nu(\lambda\tilde{X}\, \mathbf{R}(Q))$. Thanks to s.p. coinduction, this reduces to the goal

$$\tilde{a} * \mathbf{H}(P) \subseteq \mathbf{R}(Q)[\tilde{a} * \mathbf{H}(P)/\tilde{X}].$$

The induction hypothesis yields $s : \tau(\Phi(P))$ such that $s\,\mathbf{r}\,(P \subseteq \Phi(P))$, that is,

$$s * \mathbf{H}(P) \subseteq \mathbf{R}(Q[P/X]), \tag{9}$$

and some $m : \alpha_Y \Rightarrow \tau(Q[\hat{X}/X]) \Rightarrow \tau(Q[Y/X])$ such that $m\,\mathbf{r}\,(\mathrm{Mon}(\Phi)[\hat{X}/X])$. By Lemma 17 (b), this entails that $m\,\mathbf{r}\,(\mathrm{Mon}(\Phi)[P/X])$, that is, by Lemma 18, $\forall a\,(a * \mathbf{H}(P) \subseteq \tilde{Y} \to (m\,a) \circ \mathbf{R}(Q[P/X]) \subseteq \mathbf{R}(Q[Y/X]))$. Using this with $a \overset{\mathrm{Def}}{=} \tilde{a}$ and $\tilde{Y} \overset{\mathrm{Def}}{=} \tilde{a} * \mathbf{H}(P)$ we arrive at

$$(m\,\tilde{a}) \circ \mathbf{R}(Q[P/X]) \subseteq \mathbf{R}(Q)[\tilde{a} * \mathbf{H}(P)/\tilde{X}]. \tag{10}$$

Now,

$$\begin{aligned} \mathbf{R}(Q)[\tilde{a} * \mathbf{H}(P)/\tilde{X}] &\overset{(10)}{\supseteq} (m\,\tilde{a}) \circ \mathbf{R}(Q[P/X]) \\ &\overset{(9)}{\supseteq} (m\,\tilde{a}) \circ (s * \mathbf{H}(P)) \\ &= (m\,\tilde{a}\,s) * \mathbf{H}(P) \end{aligned}$$

Therefore, we set $\tilde{a} \overset{\mathrm{rec}}{=} m\,\tilde{a}\,s$ which clearly is of the right type $\mathbf{fix}\,\alpha_X\,.\,\tau(Q)$.

(d) Case Φ is Harrop, P is non-Harrop.

Subcase X is not free in Q. We have to show $\mathbf{H}(P \subseteq \nu(\Phi))$, equivalently, $\exists(\mathbf{R}(P)) \subseteq \nu(\lambda X\,\mathbf{H}(Q))$. By s.p. coinduction, this reduces to $\exists(\mathbf{R}(P)) \subseteq \mathbf{H}(Q)$ which is equivalent to the induction hypothesis, $\mathbf{H}(P \subseteq Q)$.

Subcase X is free in Q (then Q, $Q[P/X]$ and $\mathrm{Mon}(\Phi)[P/X][\hat{X}/Y]$ are non-Harrop). We need to prove $\mathbf{H}(P \subseteq \nu(\Phi))$, that is, $\exists(\mathbf{R}(P)) \subseteq \nu(\lambda X\,\mathbf{H}_X(Q))$. S.p. coinduction reduces this to the goal

$$\exists(\mathbf{R}(P)) \subseteq \mathbf{H}_X(Q)[\exists(\mathbf{R}(P))/X].$$

By the induction hypothesis we have $s\,\mathbf{r}\,(P \subseteq \Phi(P))$, equivalently,

$$s \circ \mathbf{R}(P) \subseteq \mathbf{R}(Q[P/X]), \tag{11}$$

and $\mathbf{H}(\mathrm{Mon}(\Phi)[P/X][\hat{X}/Y])$, that is,

$$\exists(\mathbf{R}(P)) \subseteq \hat{X} \to \exists(\mathbf{R}(Q[P/X])) \subseteq \mathbf{H}(Q[\hat{X}/X]).$$

Using Lemma 16 for RIFP′ with $\hat{X} \overset{\mathrm{Def}}{=} \exists(\mathbf{R}(P))$ yields

$$\exists(\mathbf{R}(Q[P/X])) \subseteq \mathbf{H}(Q[\hat{X}/X])[\exists(\mathbf{R}(P))/\hat{X}] = \mathbf{H}_X(Q)[\exists(\mathbf{R}(P))/X]. \tag{12}$$

Now,

$$\begin{aligned} \mathbf{H}_X(Q)[\exists(\mathbf{R}(P))/X] &\overset{(12)}{\supseteq} \exists(\mathbf{R}(Q[P/X])) \\ &\overset{(11)}{\supseteq} \exists(s \circ \mathbf{R}(P)) \\ &\equiv \exists(\mathbf{R}(P)). \end{aligned}$$

We conclude the proof with the strong and half strong variants of s.p. induction and coinduction. Since, as remarked in Sect. 2.1, these variants are derivable from ordinary s.p. induction and coinduction, they do

not need to be treated separately. We will do this nevertheless in order to obtain simpler realizers. We only derive these simplified realizers for those instances that will be used later although simplified realizers can be given in all cases where the conclusion of a rule is a non-Harrop formula.

HSCI′(Φ, P). Assume we have derived $P \subseteq \Phi(P) \cup \nu(\Phi)$, that is, $P \subseteq Q[P/X] \cup \nu(\Phi)$, as well as $\mathrm{Mon}(\Phi)$, that is, $X \subseteq Y \to Q \subseteq Q[Y/X]$.

Case Φ and P are both non-Harrop. We are looking for $\tilde{f} : \tau(P) \Rightarrow \mathbf{fix}\,\alpha_X\,.\,\tau(Q)$, realizing $P \subseteq \nu(\Phi)$, that is, $\tilde{f} \circ \mathbf{R}(P) \subseteq \nu(\mathbf{R}(\Phi))$. We will attempt to prove this by half strong coinduction, so our goal is to prove (since $\mathbf{R}(\Phi) = \lambda\tilde{X}\,\mathbf{R}(Q)$)

$$\tilde{f} \circ \mathbf{R}(P) \subseteq \mathbf{R}(Q)[\tilde{f} \circ \mathbf{R}(P)/\tilde{X}] \cup \nu(\mathbf{R}(\Phi))\,.$$

By the induction hypothesis we have $s : \tau(P) \Rightarrow (\tau(Q)[\tau(P)/\alpha_X] + \mathbf{fix}\,\alpha_X\,.\,\tau(Q))$ such that $s\,\mathbf{r}\,(P \subseteq \Phi(P) \cup \nu(\Phi))$, that is, by Lemma 18 for IFP′,

$$s \circ \mathbf{R}(P) \subseteq \mathbf{Left} \circ \mathbf{R}(Q[P/X]) \cup \mathbf{Right} \circ \nu(\mathbf{R}(\Phi)) \tag{13}$$

and, using Lemma 14 (c), some $m : (\alpha_X \Rightarrow \alpha_Y) \Rightarrow \tau(Q) \Rightarrow \tau(Q)[\alpha_Y/\alpha_X])$ such that $m\,\mathbf{r}\,(\mathrm{Mon}(\Phi))$ and hence also $m\,\mathbf{r}\,(\mathrm{Mon}(\Phi[P/X]))$, by Lemma 17 (a). Therefore, $\forall f\,(f \circ \mathbf{R}(P) \subseteq \tilde{Y} \to m\,f \circ \mathbf{R}(Q[P/X]) \subseteq \mathbf{R}(Q[Y/X]))$. Using this with $f \stackrel{\mathrm{Def}}{=} \tilde{f}$, where $\tilde{f}$ is yet unknown, and $\tilde{Y} \stackrel{\mathrm{Def}}{=} \tilde{f} \circ \mathbf{R}(P)$ we obtain, using again Lemma 16

$$m\,\tilde{f} \circ \mathbf{R}(Q[P/X]) \subseteq \mathbf{R}(Q)[\tilde{f} \circ \mathbf{R}(P)/\tilde{X}]\,. \tag{14}$$

Now,

$$\begin{aligned}
& \mathbf{R}(Q)[\tilde{f} \circ \mathbf{R}(P)/\tilde{X}] \cup \nu(\mathbf{R}(\Phi)) \\
\overset{(14)}{\supseteq}\ & (m\,\tilde{f} \circ \mathbf{R}(Q[P/X])) \cup \nu(\mathbf{R}(\Phi)) \\
\overset{\text{Lemma 18}}{\equiv}\ & [(m\,\tilde{f}) + \mathbf{id}] \circ (\mathbf{Left} \circ \mathbf{R}(Q[P/X]) \cup \mathbf{Right} \circ \nu(\mathbf{R}(\Phi))) \\
\overset{(13)}{\supseteq}\ & [(m\,\tilde{f}) + \mathbf{id}] \circ (s \circ \mathbf{R}(P)) \\
\overset{\text{Lemma 18}}{\equiv}\ & ([(m\,\tilde{f}) + \mathbf{id}] \circ s) \circ \mathbf{R}(P)
\end{aligned}$$

Therefore we define recursively $\tilde{f} \stackrel{\mathrm{rec}}{=} [(m\,\tilde{f}) + \mathbf{id}] \circ s$ and are done. Clearly, $\tilde{f}$ has the right type.

SCI′(Φ, P), *case Φ and P are both non-Harrop.* Using the induction hypothesis with realizers s of $P \subseteq \Phi(P \cup \nu(\Phi))$, and m of $\mathrm{Mon}(\Phi[P/X])$, one sees, with a similar reasoning as above, that the recursive definition $\tilde{f} \stackrel{\mathrm{rec}}{=} (m\,[\tilde{f} + \mathbf{id}]) \circ s$ provides a realizer of $P \subseteq \nu(\Phi)$.

HSI′(Φ, P), *case Φ is Harrop but not constant, P is non-Harrop.* Using the induction hypothesis with realizers s of $\Phi(P) \cap \mu(\Phi) \subseteq P$, and m of $\mathrm{Mon}(\Phi[P/X])$, one sees that the recursive definition $\tilde{a} \stackrel{\mathrm{rec}}{=} s\,(m\,\tilde{a})$ provides a realizer of $\mu(\Phi) \subseteq P$ (which is the same as the realizer for the corresponding instance of s.p. induction). □

Lemma 19. $\mathrm{Mon}(\Phi)$ *is provable in IFP′.*

Proof. We define $\mathrm{Mon}_X(P) \stackrel{\mathrm{Def}}{=} X \subseteq X' \to P \subseteq P[X'/X]$ where X' is a fresh variable accompanied with X. Then, for $\Phi = \lambda X\,P$, $\mathrm{Mon}(\Phi)$ is equivalent to $\mathrm{Mon}_X(P)$. Therefore, we prove $\mathrm{Mon}_X(P)$ by induction on P. That is, prove $\mathrm{Mon}_X(P)$ assuming that $\mathrm{Mon}_Y(Q)$ holds for every operator $\lambda Y\,Q$ such that Q is a subexpression of P.

For the case that P has the form $\mu(\lambda Y\, Q)$ we assume $X \subseteq X'$ and show $\mu(\lambda Y\, Q) \subseteq \mu(\lambda Y\, Q[X'/X])$. Here, we may assume that $Y \notin \{X, X'\}$. We use IFP'-induction on $\mu(\lambda Y\, Q)$ and hence have to show

$$Q[\mu(\lambda Y\, Q[X'/X])/Y] \subseteq \mu(\lambda Y\, Q[X'/X]) \tag{15}$$

and $\mathrm{Mon}(\lambda Y\, Q)$, that is, $\mathrm{Mon}_Y(Q)$. The latter holds by the induction hypothesis. But $\mathrm{Mon}_X(Q)$ also holds. Therefore, $Q \subseteq Q[X'/X]$. Thus, by Lemma 16, $Q[\mu(\lambda Y\, Q[X'/X])/Y] \subseteq Q[X'/X][\mu(\lambda Y\, Q[X'/X])/Y]$ holds. Furthermore, by closure, $Q[X'/X][\mu(\lambda Y\, Q[X'/X])/Y] \subseteq \mu(\lambda Y\, Q[X'/X])$. Thus, we have (15).

For the case that P has the form $\nu(\lambda Y\, Q)$, the argument is completely dual if we replace $\mathrm{Mon}_X(P)$ by the equivalent formula $X' \subseteq X \to P[X'/X] \subseteq P$.

The remaining cases are easy using the extracted programs in Sect. 4.2 as a guide. □

Proof of the Soundness Theorem for IFP **(Theorem 2).** From an IFP proof one can obtain an IFP' proof of the same formula by Lemma 19. Therefore we obtain the result by Theorem 3. □

4.2. Program extraction

The proof of the Soundness Theorem contains an algorithm for computing the realizing program M which we now describe. We also note how to produce a Haskell program at the end of this section. For brevity we write a derivation judgement $\Gamma \vdash d : A$ as d^A, suppressing the context.

For an IFP derivation d^A the extracted program $\mathbf{ep}(d^A)$ is defined as

$$\mathbf{ep}(d^A) \stackrel{\mathrm{Def}}{=} \mathbf{ep}'(\mathbf{pt}(d^A))$$

where $\mathbf{pt}(\cdot)$ is the transformation of IFP proofs into IFP$'$ proofs based on Lemma 19, and $\mathbf{ep}'(\cdot)$ is the program extraction procedure based on Theorem 3.

The transformation $\mathbf{pt}(d^A)$ simply replaces recursively every subderivation of the form $\mathbf{Ind}(e^{\Phi(P)\subseteq P})$ by $\mathbf{Ind}'(\mathbf{pt}(e^{\Phi(P)\subseteq P}), \mathbf{Mon}_\Phi^{\mathrm{Mon}(\Phi)})$ where $\mathbf{Mon}_\Phi^{\mathrm{Mon}(\Phi)}$ is the proof described in Lemma 19. Similarly, $\mathbf{CoInd}(e^{\Phi(P)\subseteq P})$ is replaced by $\mathbf{CoInd}'(\mathbf{pt}(e^{\Phi(P)\subseteq P}), \mathbf{Mon}_\Phi^{\mathrm{Mon}(\Phi)})$, and so on.

The extraction procedure $\mathbf{ep}'(d^A)$ is defined by recursion on derivations as follows:

If A is Harrop then $\mathbf{ep}'(d^A) \stackrel{\mathrm{Def}}{=} \mathbf{Nil}$. Hence, in the following we assume that the proven formula is non-Harrop.

Closure and coclosure are realized by the identity:

$$\mathbf{ep}'(\mathbf{Cl}_\Phi^{\Phi(\mu(\Phi))\subseteq\mu(\Phi)}) = \mathbf{ep}'(\mathbf{CoCl}_\Phi^{\nu(\Phi)\subseteq\Phi(\nu(\Phi))}) = \lambda a\,.\,a \tag{16}$$

For induction, in the case where P is non-Harrop, the extracted program is

$$\mathbf{ep}'(\mathbf{Ind}'(d^{\Phi(P)\subseteq P}, e^{\mathrm{Mon}(\Phi)})^{\mu(\Phi)\subseteq P}) =$$

$$\begin{cases} \mathbf{rec}\,(\lambda a\,.\,\mathbf{ep}'(d) \circ \mathbf{ep}'(e)\,a) & \text{if } \Phi \text{ is non-Harrop} \\ \mathbf{rec}\,(\lambda a\,.\,\mathbf{ep}'(d)\,(\mathbf{ep}'(e[\hat{X}/X])\,a)) & \text{otherwise.} \end{cases} \tag{17}$$

For coinduction in the case where Φ is non-Harrop, the extracted program is

$$\mathbf{ep}'(\mathbf{CoInd}'(d^{P\subseteq\Phi(P)}, e^{\mathrm{Mon}(\Phi)})^{P\subseteq\nu(\Phi)}) =$$

$$\begin{cases} \mathbf{rec}\,(\lambda a\,.\,\mathbf{ep}'(e)\,a \circ \mathbf{ep}'(d)) & \text{if } P \text{ is non-Harrop} \\ \mathbf{rec}\,(\lambda a\,.\,(\mathbf{ep}'(e[\hat{X}/X])\,a\,\mathbf{ep}'(d))) & \text{otherwise.} \end{cases} \tag{18}$$

For the strong and half-strong versions of induction and coinduction we only present a few cases that will be used later. For half strong induction in the case where Φ is Harrop but not constant and P is non-Harrop, the extracted program is the same as for induction, namely

$$\mathbf{ep}'(\mathbf{HSInd}(d^{\Phi(P)\cap\mu(\Phi)\subseteq P}, e^{\mathrm{Mon}(\Phi)})^{\mu(\Phi)\subseteq P}) = \mathbf{rec}\,(\lambda a\,.\,\mathbf{ep}'(d)(\mathbf{ep}'(e[\hat{X}/X])\,a)).$$

For half strong coinduction in the case where both Φ and P are non-Harrop, the extracted program is

$$\mathbf{ep}'(\mathbf{HSCoInd}(d^{P\subseteq\Phi(P)\cup\nu(\Phi)}, e^{\mathrm{Mon}(\Phi)})^{P\subseteq\nu(\Phi)}) = \mathbf{rec}\,(\lambda a\,.\,[\mathbf{ep}'(e)\,a + \mathbf{id}]\circ\mathbf{ep}'(d)). \tag{19}$$

For strong coinduction in the case where both Φ and P are non-Harrop, the extracted program is

$$\mathbf{ep}'(\mathbf{SCoInd}(d^{P\subseteq\Phi(P\cup\nu(\Phi))}, e^{\mathrm{Mon}(\Phi)})^{P\subseteq\nu(\Phi)}) = \mathbf{rec}\,(\lambda a\,.\,(\mathbf{ep}'(e)[a + \mathbf{id}])\circ\mathbf{ep}'(d)).$$

Assumptions are realized by variables, and the congruence rule does not change the realizer:

$$\begin{aligned}\mathbf{ep}'(u_i^{A_i}) &= u_i\\ \mathbf{ep}'(\mathbf{Cong}_P(d^{P(s)}, e^{s=t})^{P(t)}) &= \mathbf{ep}'(d)\end{aligned}$$

The logical rules are realized as follows:

$$\begin{aligned}\mathbf{ep}'(\vee^+_{l,B}(d^A)^{A\vee B}) &= \mathbf{Left}(\mathbf{ep}'(d))\\ \mathbf{ep}'(\vee^+_{r,A}(d^B)^{A\vee B}) &= \mathbf{Right}(\mathbf{ep}'(d))\\ \mathbf{ep}'(\vee^-(d^{A\vee B}, e^{A\to C}, f^{B\to C})^C) &= \mathbf{case}\,\mathbf{ep}'(d)\,\mathbf{of}\\ &\quad\{\mathbf{Left}(a)\to\mathbf{ep}'(e)*a\,;\\ &\quad\ \mathbf{Right}(b)\to\mathbf{ep}'(f)*b\}\end{aligned}$$

where $\mathbf{ep}'(e)*a$ means $\mathbf{ep}'(e)\,a$ if A in non-Harrop and $\mathbf{ep}'(e)$ if A is Harrop. Similarly for $\mathbf{ep}'(f)*b$.

$$\mathbf{ep}'(\wedge^+(d^A, e^B)^{A\wedge B}) = \begin{cases}\mathbf{ep}'(d) & \text{if } B \text{ is Harrop}\\ \mathbf{ep}'(e) & \text{if } A \text{ is Harrop}\\ \mathbf{Pair}(\mathbf{ep}'(d), \mathbf{ep}'(e)) & \text{otherwise}\end{cases}$$

$$\mathbf{ep}'(\wedge^-_l(d^{A\wedge B})^A) = \begin{cases}\mathbf{ep}'(d) & \text{if } B \text{ is Harrop}\\ \pi_{\mathbf{Left}}(\mathbf{ep}'(d)) & \text{otherwise}\end{cases}$$

$$\mathbf{ep}'(\wedge^-_r(d^{A\wedge B})^B) = \begin{cases}\mathbf{ep}'(d) & \text{if } A \text{ is Harrop}\\ \pi_{\mathbf{Right}}(\mathbf{ep}'(d)) & \text{otherwise}\end{cases}$$

$$\mathbf{ep}'((\to^+_{u^A}(d^B))^{A\to B}) = \begin{cases}\mathbf{ep}'(d) & \text{if } A \text{ is Harrop}\\ \lambda u.\,\mathbf{ep}'(d) & \text{otherwise}\end{cases}$$

$$\mathbf{ep}'((\to^-(d^{A\to B}, e^A))^B) = \begin{cases}\mathbf{ep}'(d) & \text{if } A \text{ is Harrop}\\ \mathbf{ep}'(d)\,\mathbf{ep}'(e) & \text{otherwise}\end{cases}$$

$$\begin{aligned}\mathbf{ep}'(\forall^+_x(d^A)^{\forall x\,A}) &= \mathbf{ep}'(d)\\ \mathbf{ep}'(\forall^-_t(d^{\forall x\,A})^{A[t/x]}) &= \mathbf{ep}'(d)\\ \mathbf{ep}'(\exists^+_{\lambda x A,t}(d^{A[t/x]})^{\exists x\,A}) &= \mathbf{ep}'(d)\end{aligned}$$

$$\mathbf{ep}'(\exists^-(d^{\exists x\,A}, e^{\forall x\,(A\to B)})^B) = \begin{cases} \mathbf{ep}'(e) & \text{if } A \text{ is Harrop} \\ \mathbf{ep}'(e)\,\mathbf{ep}'(d) & \text{otherwise} \end{cases}$$

Extraction into Haskell By the Soundness Theorem (Theorem 2) one can extract from a proof of a formula A a realizing program M such that the typing $M : \tau(A)$ can be derived using the rules given in Lemma 13. The extraction procedure $\mathbf{ep}'(\cdot)$ implicitly computes not only M but a typing derivation for $M : \tau(A)$. Composing this with the translation of RIFP programs into Haskell one obtains an extraction procedure directly into Haskell. It is easy to see that the composed procedure can be obtained by the following small modifications of $\mathbf{ep}'(\cdot)$ which we call $\mathbf{eph}'(\cdot)$. In addition to replacing $\mathbf{Pair}(M, N)$ by (M, N), the definition of $\mathbf{ep}'(\cdot)$ is changed for closure and coclosure derivation rules with $\Phi = \lambda XP$, $\alpha = \alpha_X$, and $\rho = \tau(P)$ from (16) to

$$\mathbf{eph}'(\mathbf{Cl}_\Phi^{\Phi(\mu(\Phi))\subseteq\mu(\Phi)}) = \mathsf{roll}_{\mathsf{C}_{\alpha,\rho}}$$
$$\mathbf{eph}'(\mathbf{CoCl}_\Phi^{\nu(\Phi)\subseteq\Phi(\nu(\Phi))}) = \mathsf{unroll}_{\mathsf{C}_{\alpha,\rho}}$$

For induction and coinduction with $\Phi = \lambda XQ$, $\alpha = \alpha_X$ and $\rho = \tau(Q)$, we use the following definitions instead of (17) and (18).

$$\mathbf{eph}'(\mathbf{Ind}'(d^{\Phi(P)\subseteq P}, e^{\mathrm{Mon}(\Phi)})^{\mu(\Phi)\subseteq P}) =$$

$$\begin{cases} \mathbf{rec}\,(\lambda a\,.\,\mathbf{eph}'(d)\circ(\mathbf{eph}'(e)\,a)\circ\mathsf{unroll}_{\mathsf{C}_{\alpha,\rho}}) & \text{if } \Phi \text{ is non-Harrop} \\ \mathbf{rec}\,(\lambda a\,.\,\mathbf{eph}'(d)\,(\mathbf{eph}'(e[\hat{X}/X])\,a)) & \text{otherwise.} \end{cases}$$

$$\mathbf{eph}'(\mathbf{CoInd}'(d^{P\subseteq\Phi(P)}, e^{\mathrm{Mon}(\Phi)})^{P\subseteq\nu(\Phi)}) =$$

$$\begin{cases} \mathbf{rec}\,(\lambda a.\ \mathsf{roll}_{\mathsf{C}_{\alpha,\rho}}\circ(\mathbf{eph}'(e)\,a)\circ\mathbf{eph}'(d)) & \text{if } P \text{ is non-Harrop} \\ \mathbf{rec}\,(\lambda a\,.\,(\mathbf{eph}'(e[\hat{X}/X])\,a\ \mathbf{eph}'(d))) & \text{otherwise.} \end{cases}$$

Similar modifications need to be carried out for the other induction and coinduction schemes.

4.3. Realizing natural numbers

In Sect. 2.3.3 we defined natural numbers as a subset of the real numbers through the inductive predicate $\mathbf{N}(x) \stackrel{\mu}{=} x = 0 \vee \mathbf{N}(x-1)$. This view of natural numbers is abstract since no concrete representation is associated with it. A concrete representation of natural numbers is provided through the realizability interpretation of the predicate $\mathbf{N}$. Note that the formula $\mathbf{N}(x)$ is not Harrop since it contains a disjunction at a strictly positive position. We have $\tau(\mathbf{N}) = \mathbf{nat} = \mathbf{fix}\,\alpha\,.\,1{+}\alpha$, the type of natural numbers (see Sect. 3.3). Realizability for $\mathbf{N}$ works out as

$$a\,\mathbf{r}\,\mathbf{N}(x) \stackrel{\mu}{=} a = \mathbf{Left}(\mathbf{Nil}) \wedge x = 0 \vee \exists b\,(a = \mathbf{Right}(b) \wedge b\,\mathbf{r}\,\mathbf{N}(x-1))\,.$$

Therefore, $a\,\mathbf{r}\,\mathbf{N}(x)$ means that a is the unary representation of the natural number x.

Lemma 20.

(a) $(\mathbf{r}\,\mathbf{N}(x)) \leftrightarrow \mathbf{N}(x)$
(b) $(\mathbf{r}\,\exists x \in \mathbf{N}\,A(x)) \leftrightarrow (\exists x \in \mathbf{N}\,\mathbf{r}\,A(x))$.
(c) $\mathbf{H}(\forall x \in \mathbf{N}\,A(x)) \leftrightarrow \forall x \in \mathbf{N}\,\mathbf{H}(A(x))$ *if* $A(x)$ *is a Harrop formula.*

(d) $a\,\mathbf{r}\,\mathbf{N}(x) \wedge b\,\mathbf{r}\,\mathbf{N}(y) \to (a = b \leftrightarrow x = y)$.

Proof. Both implications of part (a) are easily proven by induction.

Parts (b) and (c) follow immediately from (a).

To prove part (d) one can use that natural numbers are non-negative and subtraction is an injective function in its first argument. □

Remark. In the parts (a-c) of Lemma 20, $\mathbf{N}$ may be replaced by any predicate that contains neither implications nor universal quantifiers nor free predicate variables. However, (d) depends on the concrete definition of $\mathbf{N}$ and specific properties of the theory of real numbers.

By Lemma 20 it is safe to identify natural numbers with their realizers. Henceforth we will use the variables $n, m, k, l, \ldots$ for both. Hence, in an IFP proof a natural number is a special real number while in an extracted program it is a special domain element. Recall from Sect. 2.3.3 that rational numbers are defined by the predicate $\mathbf{Q}(q) \stackrel{\text{Def}}{=} \exists x, y, z \in \mathbf{N}\,(z \neq 0 \wedge q \cdot z = x - y)$ which corresponds to a representation of rational numbers by triples of natural numbers (n, m, k) $(k \neq 0)$ denoting $(n - m)/k$. Although the corresponding statement of Lemma 20 (d) (i.e. uniqueness of realizers) does not hold for $\mathbf{Q}$, the generalizations of Lemma 20 (a–c) do apply to $\mathbf{Q}$. Therefore, one can use realizers to express rational numbers.

Example 2. In Example 1, we proved $A \stackrel{\text{Def}}{=} \forall x, y\ (\mathbf{N}(x) \to \mathbf{N}(y) \to \mathbf{N}(x + y))$. We have $\tau(A) = \mathbf{nat} \Rightarrow \mathbf{nat} \Rightarrow \mathbf{nat}$. According to Lemma 19, the formula $\mathrm{Mon}(\Phi_{\mathbf{N}}) \stackrel{\text{Def}}{=} X \subseteq Y \to \Phi_{\mathbf{N}}(X) \subseteq \Phi_{\mathbf{N}}(Y)$ expressing the monotonicity of the operator $\Phi_{\mathbf{N}}$ is provable in IFP' and the following program $\mathsf{mon}_{\mathbf{N}} : (\alpha_X \Rightarrow \alpha_Y) \Rightarrow \mathbf{1} + \alpha_X \Rightarrow \mathbf{1} + \alpha_Y$ is extracted from the proof.

$$\mathsf{mon}_{\mathbf{N}} = \lambda f.\lambda m.\,\mathbf{case}\ m\ of\{\mathbf{Left}(a) \to \mathbf{Left}(a); \mathbf{Right}(b) \to \mathbf{Right}(f(b))\}$$

Furthermore, from the proof of the induction premise we extract the following program of type $(\mathbf{1} + \mathbf{nat}) \Rightarrow \mathbf{nat}$.

$$s = (\lambda m.\mathbf{case}\, m\, \mathbf{of}\, \{\mathbf{Left}(c) \to n; \mathbf{Right}(c) \to \mathbf{Right}(c)\})$$

Here, n is the realizer of $\mathbf{N}(x)$. Therefore, by (17) of Sect. 4.2, the realizer extracted from the proof of A is the following program of type $\mathbf{nat} \Rightarrow \mathbf{nat} \Rightarrow \mathbf{nat}$

$$\begin{aligned} \mathsf{plus} &= \lambda n.\,\mathbf{rec}\ \lambda f.\, s \circ (\mathsf{mon}_{\mathbf{N}}\ f) \\ &= \lambda n.\,\mathbf{rec}\ \lambda f.\,\lambda m.\, s((\mathsf{mon}_{\mathbf{N}}\ f)m)\,. \end{aligned}$$

By program axiom (ii), s is strict. Therefore, by Lemma 12, we can rewrite

$$\begin{aligned} \mathsf{plus}\, n\, m &\stackrel{\text{rec}}{=} \mathbf{case}\ m\ \mathbf{of}\{\mathbf{Left}(a) \to s(\mathbf{Left}(a)); \mathbf{Right}(b) \to s(\mathbf{Right}(\mathsf{plus}\ n\ b))\} \\ &\stackrel{\text{rec}}{=} \mathbf{case}\ m\ \mathbf{of}\{\mathbf{Left}(a) \to n; \mathbf{Right}(b) \to \mathbf{Right}(\mathsf{plus}\ n\ b)\}. \end{aligned}$$

4.4. Realizing wellfounded induction

In this section we work out in detail the realizers of wellfounded induction and its specializations (Sect. 2.2) as provided by the Soundness Theorem (Theorem 2). This will be important for understanding the programs extracted in Sect. 5.

Lemma 21 *(Realizer of wellfounded induction). The schema of wellfounded induction,* $\mathbf{WfI}_{\prec,A}(P)$*, is realized as follows. If* s *realizes* $\mathbf{Prog}_{\prec,A}(P)$ *where* P *is non-Harrop, then* $\mathbf{Acc}_{\prec} \cap A \subseteq P$ *is realized by*

- $\tilde{f} \stackrel{\text{rec}}{=} \lambda a.\,(s\,a\,(\lambda a'.\,\lambda b.\,\tilde{f}\,a'))$ *if* $\prec$ *and* A *are both non-Harrop,*
- $\tilde{f} \stackrel{\text{rec}}{=} \lambda a.\,(s\,a\,\tilde{f})$ *if* $\prec$ *is Harrop and* A *is non-Harrop,*
- $\tilde{c} \stackrel{\text{rec}}{=} s\,(\lambda b.\,\tilde{c})$ *if* $\prec$ *is non-Harrop and* A *is Harrop,*
- $\mathbf{rec}\,s$ *if* $\prec$ *and* A *are both Harrop.*

Proof. Since $\mathbf{WfI}_{\prec,A}(P)$ follows from $\mathbf{WfI}_{\prec}(A \Rightarrow P)$ and the latter is an instance of induction, the extracted programs shown in the lemma can be obtained from Theorem 2. However, it is instructive to give some details of their derivations.

Recall that $\mathbf{Acc}_{\prec} = \mu(\Phi)$ where $\Phi(X) = \lambda x\,\forall y \prec x\,X(y)$ and $\mathbf{Prog}_{\prec}(Q) = \Phi(Q) \subseteq Q$. Since Φ is a Harrop operator, $\mathbf{Acc}_{\prec}$ is a Harrop predicate.

According to Theorem 2 and the program extraction procedure described in Sect. 4.2 the extracted realizer of $\mathbf{Acc}_{\prec} \subseteq A \Rightarrow P$ is

$$\tilde{f} \stackrel{\text{rec}}{=} s'\,(m\,\tilde{f})$$

provided $s'\,\mathbf{r}\,\mathbf{Prog}_{\prec}(A \Rightarrow P)$ and $m\,\mathbf{r}\,(\mathrm{Mon}(\Phi)[\hat{X}/X])$. Because s realizes $\mathbf{Prog}_{\prec,A}(P)$, that is,

$$\forall x\,(x \in A \to \forall y\,(y \in A \to y \prec x \to y \in P) \to x \in P)$$

and $\mathbf{Prog}_{\prec}(A \Rightarrow P)$ expands to

$$\forall x\,(\forall y(y \prec x \to y \in A \to y \in P) \to x \in A \to x \in P)$$

it is clear that we can define

- $s' \stackrel{\text{Def}}{=} \lambda g.\lambda a.\,(s\,a\,(\lambda a'.\,\lambda b.\,g\,b\,a'))$ if $\prec$ and A are both non-Harrop,
- $s' \stackrel{\text{Def}}{=} \lambda f.\lambda a.\,(s\,a\,f)$ if $\prec$ is Harrop and A is non-Harrop,
- $s' \stackrel{\text{Def}}{=} s$ if A is Harrop.

The realizer m of $\mathrm{Mon}(\Phi)[\hat{X}/X]$, which expands to

$$\hat{X} \subseteq Y \to \forall x\,(\forall y \prec x\,\hat{X}(y)) \to \forall y \prec x\,Y(y)$$

is easily extracted as

- $\lambda a.\,\lambda b.\,a$ if $\prec$ is non-Harrop,
- $\lambda a.\,a$ if $\prec$ is Harrop.

From this, one can easily see that the extracted realizer of $\mathbf{Acc}_{\prec} \subseteq A \Rightarrow P$ is as stated in the lemma. Since $\mathbf{Acc}_{\prec}$ is Harrop it follows that the same program realizes the inclusion $\mathbf{Acc}_{\prec} \cap A \subseteq P$. □

Finally, we exhibit the realizers of Archimedean induction. We only look at the forms $\mathbf{AI}_q$ and $\mathbf{AIB}_q$ since the principles **AI** and **AIB** have the same realizers and will not be used in the following.

Lemma 22 *(Realizers of Archimedean induction).*

$\mathbf{AI}_q$ *If* s *realizes* $\forall x \neq 0\,((|x| \leq q \to P(2x)) \to P(x))$*, where* P *is non-Harrop, then* $\mathbf{rec}\,s$ *realizes* $\forall x \neq 0\,P(x)$*.*

AIB$_q$ *If s realizes $\forall x \in B \setminus \{0\}\,(P(x) \vee (|x| \leq q \wedge B(2x) \wedge (P(2x) \to P(x))))$, where B and P are non-Harrop, then*

$$a\,b \stackrel{\text{rec}}{=} \mathbf{case}\,s\,b\,\mathbf{of}\,\{\mathbf{Left}(c) \to c; \mathbf{Right}(b', d) \to d\,(a\,b')\} \tag{20}$$

realizes $\forall x \in B \setminus \{0\}\,P(x)$.

Proof. $\mathbf{AI}_q$ is derived from half strong induction **HSI** as is shown in Lemma 2, and the realizer of the monotonicity of the operator in questions clearly is the identity. Therefore, as we studied in Section 4.2, it has the realizer $a \stackrel{\text{rec}}{=} s\,a$, that is, $\mathbf{rec}\,s$.

Clearly, the premise of $\mathbf{AIB}_q(B, P)$ implies the premise of $\mathbf{AI}_q(B \Rightarrow P)$. From a realizer s of the premise of the former one obtains the realizer

$$s' = \lambda a.\lambda b.\,\mathbf{case}\,s\,b\,\mathbf{of}\,\{\mathbf{Left}(c) \to c; \mathbf{Right}(b', d) \to d\,(a\,b')\}$$

of the premise of the latter. Therefore, $a \stackrel{\text{rec}}{=} s'\,a$, that is,

$$a\,b \stackrel{\text{rec}}{=} \mathbf{case}\,s\,b\,\mathbf{of}\,\{\mathbf{Left}(c) \to c; \mathbf{Right}(b', d) \to d\,(a\,b')\}$$

realizes the conclusion $\forall x \in P \setminus \{0\}\,B(x)$. □

5. Stream representations of real numbers

As a first serious application of IFP we present a case study about the specification and extraction of exact representations of real numbers. This will highlight many features of our system such as the use of classical axioms as well as partial and infinite realizers. We will continue the development of the system IFP in Sect. 6 with the operational semantics of programs.

We study three representations of real numbers as infinite streams of discrete data: Cauchy representation, signed digit representation, and infinite Gray code [28,72]. We first recall each representation informally in the style of computable analysis [76]. Then we show how it can be obtained as the realizability interpretation of a suitable predicate built on the formalization of real numbers in IFP in Sect. 2.3. Hence, in this section all formal definitions and proofs take place in IFP($\mathcal{A}_R$) where $\mathcal{A}_R$ is the non computational axiom system for the real numbers introduced in Sect. 2.3 which includes the Archimedean property (**AP**) and Brouwer's Thesis for nc relations ($\mathbf{BT_{nc}}$). In particular, in this instance of IFP the various versions of Archimedean Induction (Sect. 2.3.5) are valid.

For our purpose it is most convenient to work in the interval $[-1, 1]$. Everything could be easily transferred to the unit interval $[0, 1]$ which is used in [72].

We will use the notation $a_0 : a_1 : \ldots$ to denote infinite streams, mostly in an informal setting but occasionally also for elements of the domain D that represent streams (as we did in Sect. 3.1).

5.1. Cauchy representation

Informal definition An infinite sequence $a = (a_i)_{i \in \mathbf{N}}$ of rational numbers that converges quickly to a real number x is called a *Cauchy representation* of x:

$$\mathbf{C}(a, x) \stackrel{\text{Def}}{=} \forall n \in \mathbf{N}\,|x - a_n| \leq 2^{-n}\,.$$

We consider the Cauchy representation as the standard representation and call any other representation R of real numbers (in $[-1, 1]$) *computable* if it is computably equivalent to the Cauchy representation restricted

to $[-1,1]$, i.e. R-representations can be effectively transformed into Cauchy representations and vice-versa. More precisely, if $R(r,x)$ expresses that r is a R-representation of x we say that R is computable if there exist (possibly partial) computable functions φ,ψ such that for all a,r and $x\in[-1,1]$

$$R(r,x)\to\mathbf{C}(\varphi(r),x)\qquad\text{and}\qquad\mathbf{C}(a,x)\to R(\psi(a),x)\,.$$

Note that all representations we will consider are functions or infinite sequences of discrete objects (rational numbers or digits) possibly extended with undefinedness. There exist natural notions of computable functions between such representations (see [62], [76], [72]).

Formalization in IFP The Cauchy representation can be obtained through the realizability interpretation of the predicate

$$\mathbf{C}(x)\stackrel{\text{Def}}{=}\forall n\in\mathbf{N}\,\exists q\in\mathbf{Q}\,|x-q|\le 2^{-n}\,.$$

By unfolding the definition of realizability one obtains $a\,\mathbf{r}\,\mathbf{C}(x)\leftrightarrow$

$$\forall n\,(a:\mathbf{nat}\Rightarrow\mathbf{rat}\wedge\forall b\;(b\,\mathbf{r}\,\mathbf{N}(n)\to\exists q\,((a\,b)\,\mathbf{r}\,\mathbf{Q}(q)\wedge|x-q|\le 2^{-n})))$$

where $\mathbf{rat}\stackrel{\text{Def}}{=}\tau(\mathbf{Q})=\mathbf{nat}\times\mathbf{nat}\times\mathbf{nat}$. By identifying natural numbers with their realizers, this simplifies to

$$a\,\mathbf{r}\,\mathbf{C}(x)\leftrightarrow a:\mathbf{nat}\Rightarrow\mathbf{rat}\wedge\forall n\in\mathbf{N}\,\exists q\,((a\,n)\,\mathbf{r}\,\mathbf{Q}(q)\wedge|x-q|\le 2^{-n})$$

and by further expressing rational numbers through their realizers, it becomes

$$a\,\mathbf{r}\,\mathbf{C}(x)\leftrightarrow a:\mathbf{nat}\Rightarrow\mathbf{rat}\wedge\forall n\in\mathbf{N}\,|x-a\,n|\le 2^{-n}\,.$$

Therefore $a\,\mathbf{r}\,\mathbf{C}(x)\leftrightarrow\mathbf{C}(a,x)$ where the infinite sequence a is given as a function on the natural numbers. Alternatively, one can formalize the Cauchy representation coinductively by

$$\mathbf{C}'(x)\stackrel{\nu}{=}\exists n\in\mathbf{N}\,(|x-n|\le 1\wedge\mathbf{C}'(2x)).$$

Defining the type of streams of type ρ as

$$\rho^{\omega}\stackrel{\text{Def}}{=}\mathbf{fix}\,\alpha\,.\,\rho\times\alpha$$

the predicate $\mathbf{C}'$ has the type $\tau(\mathbf{C}')=\mathbf{nat}^{\omega}$ and we obtain the realizability interpretation

$$a\,\mathbf{r}\,\mathbf{C}'(x)\stackrel{\nu}{=}\exists n\ \in\mathbf{N},a'\,(a=\mathbf{Pair}(n,a')\wedge|x-n|\le 1\wedge a'\,\mathbf{r}\,\mathbf{C}'(2x))$$

by identifying natural numbers with their realizers. Therefore, the two formalizations lead to different 'implementations' of the Cauchy representation. However, they are equivalent in the sense that one can prove $\mathbf{C}(x)\leftrightarrow\mathbf{C}'(x)$ and extract from the proof mutually inverse translations between the representations. The stream representation has the advantage that it permits 'memoized' computation due to a lazy operational semantics (see Sect. 6).

5.2. Signed digit representation

Informal definition For an infinite sequence $p = (p_i)_{i<\omega}$ of signed digits $p_i \in \{-1, 0, 1\}$ set

$$[\![p]\!] \stackrel{\text{Def}}{=} \sum_{i<\omega} p_i 2^{-i} \in [-1, 1]\,. \tag{21}$$

If $x = [\![p]\!]$, then p is called a *signed digit representation* of $x \in [-1, 1]$. We set

$$\mathbf{S}(p, x) \stackrel{\text{Def}}{=} [\![p]\!] = x\,.$$

The digit 0 is redundant since every $x \in [-1, 1]$ has a *binary representation*, that is, a signed digit representation $p \in \{-1, 1\}^\omega$. However, the redundancy is needed to render the signed digit representation computable, in particular to be able to compute from a Cauchy representation of x a signed digit representation of x.

One easily sees that for $d \in \{-1, 0, 1\}$, $p \in \{-1, 0, 1\}^\omega$ and $x \in [-1, 1]$

$$\mathbf{S}(d : p, x) \leftrightarrow |2x - d| \le 1 \wedge \mathbf{S}(p, 2x - d) \tag{22}$$

where $d : p$ denotes the sequence beginning with d and continuing with p.

Formalization in IFP We define a predicate $\mathbf{S}(x)$ expressing that x has a signed digit representation. First, we define the property of being a signed digit,

$$\mathbf{SD}(x) \stackrel{\text{Def}}{=} (x = -1 \vee x = 1) \vee x = 0\,.$$

We define $\mathbf{3} \stackrel{\text{Def}}{=} (\mathbf{1} + \mathbf{1}) + \mathbf{1}$. Then, $\tau(\mathbf{SD}) = \mathbf{3}$ and

$$\begin{aligned} d\,\mathbf{r}\,\mathbf{SD}(x) = {} & (d = \mathbf{Left}(\mathbf{Left}(\mathbf{Nil})) \wedge x = -1) \vee \\ & (d = \mathbf{Left}(\mathbf{Right}(\mathbf{Nil})) \wedge x = 1) \vee \\ & (d = \mathbf{Right}(\mathbf{Nil}) \wedge x = 0)\,. \end{aligned}$$

Thus, the three digits $-1, 1, 0$ are realized by the three elements $\mathbf{Left}(\mathbf{Left}(\mathbf{Nil}))$, $\mathbf{Left}(\mathbf{Right}(\mathbf{Nil}))$, $\mathbf{Right}(\mathbf{Nil})$ of $\mathbf{3}$. We identify these natural numbers and their realizers and use variables d, e for both of them.

Next we define a predicate expressing that $d \in \{-1, 0, 1\}$ is the first digit of a signed digit representation of x

$$\mathbb{I}(d, x) \stackrel{\text{Def}}{=} |2x - d| \le 1\,.$$

Finally, in view of (22), we set

$$\mathbf{S}(x) \stackrel{\nu}{=} \exists d \in \mathbf{SD}\,(\mathbb{I}(d, x) \wedge \mathbf{S}(2x - d))\,.$$

We have $\tau(\mathbf{S}) = \mathbf{3}^\omega$ and

$$p\,\mathbf{r}\,\mathbf{S}(x) \stackrel{\nu}{=} \exists d\ \in \mathbf{SD}, p'\ (p = \mathbf{Pair}(d, p')\ \wedge \mathbb{I}(d, x) \wedge p'\,\mathbf{r}\,\mathbf{S}(2x - d))\,.$$

Because of (22) one easily sees that $p\,\mathbf{r}\,\mathbf{S}(x)$ holds iff p is an infinite stream of signed digits that represents x, i.e. $\mathbf{S}(p, x)$ holds.

5.3. Infinite Gray code

Informal definition Gray code of a real number x in $[-1,1]$ is defined using the digits $\mathbf{L}$ and $\mathbf{R}$ and an 'undefined' digit, $\perp$. We first define *total Gray code* of x which is a variant of the binary representation and which does not use $\perp$. For an infinite sequence $q \in \{\mathbf{L},\mathbf{R}\}^\omega$, we say that q is a total Gray code of x if $x = [\![q]\!]_{\mathbf{G}}$ where, identifying $\mathbf{L}$ with -1 and $\mathbf{R}$ with 1,

$$[\![q]\!]_{\mathbf{G}} = \sum_{i<\omega}(-\prod_{j\le i}(-q_j))2^{-i} \in [-1,1]. \tag{23}$$

There are simple conversion algorithms between binary representation and total Gray code. Comparing (23) with (21), one can see that if $(q_i)_{i<\omega}$ is a Gray code, then $p = (p_i)_{i<\omega}$ is a binary representation of the same number for $p_i = -\prod_{j\le i}(-q_j)$. This equation means that p_i is 1 iff $q_0,\dots,q_j$ contains an odd number of $\mathbf{R}$. Conversely, if $p = (p_i)_{i<\omega}$ is a binary representation, then $(q_i)_{i<\omega}$ for

$$q_i = \begin{cases} \mathbf{L} & \text{if } p_{i-1} = p_i \\ \mathbf{R} & \text{if } p_{i-1} \neq p_i \end{cases}$$

is a total Gray code of the same number. Here, we temporarily define $p_{-1} = -1$. Defining the 'tent function' $\mathbf{t} : [-1,1] \to [-1,1]$ as

$$\mathbf{t}(x) = 1 - 2|x|,$$

one can show

$$[\![a : q]\!]_{\mathbf{G}} = x \leftrightarrow ((x \le 0 \wedge a = \mathbf{L}) \vee (x \ge 0 \wedge a = \mathbf{R})) \wedge [\![q]\!]_{\mathbf{G}} = \mathbf{t}(x)$$

for $a \in \{\mathbf{L},\mathbf{R}\}$, $q \in \{\mathbf{L},\mathbf{R}\}^\omega$ and $x \in [-1,1]$. This means that q is an itinerary of x along the tent function, i.e. q_n equals $\mathbf{L}$ or $\mathbf{R}$ depending on whether $\mathbf{t}^n(x)$ is negative or positive. If $\mathbf{t}^n(x) = 0$, then q_n may be either.

Total Gray code is non-unique for the dyadic rationals in $(-1,1)$, that is, numbers of the form $k/2^l$ where $l \in \mathbf{N}$ and $k \in \mathbf{Z}$ and with $|k| < 2^l$. Such numbers have two binary codes of the form $t(-1)1^\omega$ and $t1(-1)^\omega$ for some finite sequence $t \in \{-1,1\}^*$, and therefore have exactly two total Gray codes, namely,[1]

$$s\mathbf{LRL}^\omega \quad \text{and} \quad s\mathbf{RRL}^\omega$$

for some finite sequence $s \in \{\mathbf{L},\mathbf{R}\}^*$. These two codes only differ in the first digit after s, so it is natural to allow this digit to be $\perp$ since it carries no information. Therefore, we define the set of *Gray codes* as

$$\begin{aligned} \mathbf{GC} &\stackrel{\text{Def}}{=} \{\mathbf{L},\mathbf{R}\}^\omega \cup \{s\perp\mathbf{RL}^\omega \mid s \in \{\mathbf{L},\mathbf{R}\}^*\} \\ &= \{q \in \{\mathbf{L},\mathbf{R},\perp\}^\omega \mid \forall n\,(q_n = \perp \to (q_k)_{k>n} = \mathbf{RL}^\omega)\} \end{aligned}$$

and define $[\![\cdot]\!]_{\mathbf{G}} : \mathbf{GC} \to [-1,1]$ as the extension of total Gray code $[\![\cdot]\!]_{\mathbf{G}} : \{\mathbf{L},\mathbf{R}\}^\omega \to [-1,1]$ by setting

$$[\![s\perp\mathbf{RL}^\omega]\!]_{\mathbf{G}} \stackrel{\text{Def}}{=} [\![s\mathbf{LRL}^\omega]\!]_{\mathbf{G}} (= [\![s\mathbf{RRL}^\omega]\!]_{\mathbf{G}})\,.$$

For example $[\![\perp\mathbf{RL}^\omega]\!]_{\mathbf{G}} = 0$ and $[\![\mathbf{R}\perp\mathbf{RL}^\omega]\!]_{\mathbf{G}} = 1/2$. We set

[1] If $s = a_0,\dots,a_{n-1}$, then $s\mathbf{LRL}^\omega = a_0 : \dots : a_{n-1} : \mathbf{L} : \mathbf{R} : \mathbf{L} : \mathbf{L} : \dots$

$$\mathbf{G}(q,x) \stackrel{\text{Def}}{=} q \in \mathbf{GC} \wedge [\![q]\!]_{\mathbf{G}} = x\,.$$

One can see that

$$\begin{aligned}\mathbf{G}(a:q,x) \leftrightarrow ((x \le 0 \wedge a = \mathbf{L}) \vee (x \ge 0 \wedge a = \mathbf{R}) \vee\ (x = 0 \wedge a = \bot))\\ \wedge\, \mathbf{G}(q, \mathbf{t}(x))\end{aligned} \tag{24}$$

for $a \in \{\bot, \mathbf{L}, \mathbf{R}\}$, $q \in \{\bot, \mathbf{L}, \mathbf{R}\}^\omega$ and $x \in [-1,1]$. Note that $\mathbf{t}(x)$ in the right conjunction of (24) does not depend on the first digit a whereas for the signed digit case $2x - d$ in the right conjunction of (22) depends on d.

While it can be shown that total Gray code is *not* computable, Gray code *is*, thanks to the possibility of having an undefined digit. In [72] one finds programs translating between Gray code and the signed digit representation.

Formalization in IFP We define a predicate $\mathbf{G}(x)$ expressing that x has a Gray code. We first define a predicate for the digits of Gray code:

$$\mathbf{D}(x) \stackrel{\text{Def}}{=} x \neq 0 \to (x \le 0 \vee x \ge 0)\,.$$

We have $\tau(\mathbf{D}) = \mathbf{2}$ for $\mathbf{2} \stackrel{\text{Def}}{=} \mathbf{1} + \mathbf{1}$. Note that $D_{\mathbf{2}} = \{\mathbf{Left}(\mathbf{Nil}), \mathbf{Right}(\mathbf{Nil}), \bot, \mathbf{Left}(\bot), \mathbf{Right}(\bot)\}$ (see Remark 1 of Sect. 3.7). Setting $\mathbf{L} \stackrel{\text{Def}}{=} \mathbf{Left}(\mathbf{Nil})$ and $\mathbf{R} \stackrel{\text{Def}}{=} \mathbf{Right}(\mathbf{Nil})$, we have

$$a\,\mathbf{r}\,\mathbf{D}(x) = a:\mathbf{2} \wedge\ (x \neq 0 \to (a = \mathbf{L} \wedge x \le 0) \vee (a = \mathbf{R} \wedge x \ge 0))\,.$$

Thus, all elements of $\mathbf{2}$ realize $\mathbf{D}(0)$. By considering not only $\bot$ but also $\mathbf{Left}(\bot)$ and $\mathbf{Right}(\bot)$ as denotations of the Gray code digit $\bot$, $a\,\mathbf{r}\,\mathbf{D}(x)$ means that a is the first digit of a Gray code of x. Therefore, we define

$$\mathbf{G}(x) \stackrel{\nu}{=} (-1 \le x \le 1) \wedge \mathbf{D}(x) \wedge \mathbf{G}(\mathbf{t}(x))\,.$$

We have $\tau(\mathbf{G}) = \mathbf{2}^\omega$ and

$$q\,\mathbf{r}\,\mathbf{G}(x) \stackrel{\nu}{=} (-1 \le x \le 1) \wedge \exists a, q'\ (q = \mathbf{Pair}(a,q') \wedge a\,\mathbf{r}\,\mathbf{D}(x) \wedge q'\,\mathbf{r}\,\mathbf{G}(\mathbf{t}(x)))$$

and hence $q\,\mathbf{r}\,\mathbf{G}(x)$ means that q is a Gray code of x, i.e. $\mathbf{G}(q,x)$ by (24).

5.4. Extracting conversion from signed digit representation to Gray code

We show $\mathbf{S} \subseteq \mathbf{G}$ and extract from the proof a program that converts signed digit representation to Gray code. Proofs are presented in an informal style but are formalizable in the system IFP($\mathcal{A}_R$) and simplifications of programs are proven in RIFP($\emptyset$). We write $x \in \mathbb{I}_d$ for $\mathbb{I}(d,x)$ and allow combinations of patterns in case expressions. For example,

$$\begin{aligned}&\mathbf{case}\,M\,\mathbf{of}\,\{-1 \to N_1; 1 \to N_2; 0 \to N_3;\} \stackrel{\text{Def}}{=}\\ &\quad \mathbf{case}\,M\,\mathbf{of}\,\{\mathbf{Left}(a) \to (\mathbf{case}\,a\,\mathbf{of}\,\{\mathbf{Left}(b) \to N_1; \mathbf{Right}(b) \to N_2\});\\ &\qquad \mathbf{Right}(a) \to N_3\}\,.\end{aligned}$$

Recall that $\mathbf{S} = \nu(\Phi_{\mathbf{S}})$ and $\mathbf{G} = \nu(\Phi_{\mathbf{G}})$ for

$$\Phi_{\mathbf{S}} \stackrel{\text{Def}}{=} \lambda X\, \lambda x\, \exists d \in \mathbf{SD}\, (x \in \mathbb{I}_d \wedge X(2x-d))\,,$$
$$\Phi_{\mathbf{G}} \stackrel{\text{Def}}{=} \lambda X\, \lambda x\, (-1 \le x \le 1) \wedge \mathbf{D}(x) \wedge X(\mathbf{t}(x))\,.$$

According to Lemma 19, the formula $\mathrm{Mon}(\Phi_{\mathbf{S}}) \stackrel{\text{Def}}{=} X \subseteq Y \to \Phi_{\mathbf{S}}(X) \subseteq \Phi_{\mathbf{S}}(Y)$ expressing the monotonicity of the operator $\Phi_{\mathbf{S}}$ is proved in IFP' and the following program $\mathsf{mon} : (\alpha_X \Rightarrow \alpha_Y) \Rightarrow \mathbf{3} \times \alpha_X \Rightarrow \mathbf{3} \times \alpha_Y$ is extracted from the proof.

$$\mathsf{mon}\, f\, p \stackrel{\text{Def}}{=} \mathbf{Pair}(\pi_{\mathbf{Left}}\, p, f(\pi_{\mathbf{Right}}\, p)). \tag{25}$$

It is also the case for $\mathrm{Mon}(\Phi_{\mathbf{G}})$ and the same program mon with the type obtained by replacing $\mathbf{3}$ with $\mathbf{2}$ is realizing $\mathrm{Mon}(\Phi_{\mathbf{G}})$.

Lemma 23. $\forall x\, (\mathbf{S}(-x) \to \mathbf{S}(x))$.

Proof. By coinduction. Therefore, we show $P \subseteq \Phi_{\mathbf{S}}(P)$ for $P(x) \stackrel{\text{Def}}{=} \mathbf{S}(-x)$, that is,

$$\forall x\, (\mathbf{S}(-x) \to \exists d \in \mathbf{SD}\, (x \in \mathbb{I}_d \wedge \mathbf{S}(-(2x-d)))). \tag{26}$$

Suppose that $\mathbf{S}(-x)$ holds. By coclosure, for some $e \in \mathbf{SD}$, we have $-x \in \mathbb{I}_e \wedge \mathbf{S}(-2x-e)$. Since $-x \in \mathbb{I}_e$, we have $x \in \mathbb{I}_{-e}$. Since $\mathbf{S}(-2x-e)$, we have $\mathbf{S}(-(2x-d))$ for $d = -e$, and therefore $x \in \mathbb{I}_d \wedge \mathbf{S}(-(2x-d))$. □

The program $\mathsf{step1} : \mathbf{3}^\omega \Rightarrow \mathbf{3} \times \mathbf{3}^\omega$ extracted from the proof of (26) is

$$\mathsf{step1} \stackrel{\text{Def}}{=} \lambda p.\, \mathbf{Pair}(\mathbf{case}\, (\pi_{\mathbf{Left}}\, p)\, \mathbf{of}\, \{-1 \to 1; 0 \to 0; 1 \to -1\}, \pi_{\mathbf{Right}}\, p)\,.$$

Therefore, by (18) of Sect. 4.2, the realizer extracted from the proof of $P \subseteq \mathbf{S}$ is the following program $\mathsf{minus} : \mathbf{3}^\omega \Rightarrow \mathbf{3}^\omega$

$$\mathsf{minus} \stackrel{\text{rec}}{=} (\mathsf{mon}\ \mathsf{minus}) \circ \mathsf{step1}.$$

After some simplification using Lemma 12 we have

$$\mathsf{minus}\, p \stackrel{\text{rec}}{=} \mathbf{Pair}(\mathbf{case}\, (\pi_{\mathbf{Left}}\, p)\, \mathbf{of}\, \{-1 \to 1; 0 \to 0; 1 \to -1\}, \mathsf{minus}\ (\pi_{\mathbf{Right}}\, p)). \tag{27}$$

Theorem 4. $\mathbf{S} \subseteq \mathbf{G}$.

Proof. By coinduction. Hence we show $\forall x(\mathbf{S}(x) \to (-1 \le x \le 1) \wedge \mathbf{D}(x) \wedge \mathbf{S}(\mathbf{t}(x)))$. Since $\forall x(\mathbf{S}(x) \to -1 \le x \le 1)$ is immediate, we need to show the following two claims.
Claim 1. $\forall x\, (\mathbf{S}(x) \to \mathbf{D}(x))$, that is, $\forall x \in \mathbf{S} \setminus \{0\}\ \mathbf{B}(x)$ where $\mathbf{B}(x) \stackrel{\text{Def}}{=} x \le 0 \vee x \ge 0$. We use $\mathbf{AIB}_{1/2}(\mathbf{S}, \mathbf{B})$. Therefore, we show

$$\forall x \in \mathbf{S} \setminus \{0\}\ (\mathbf{B}(x) \vee (|x| \le 1/2 \wedge \mathbf{S}(2x) \wedge (\mathbf{B}(2x) \to \mathbf{B}(x)))). \tag{28}$$

Since $\mathbf{S}(x) \stackrel{\nu}{=} \exists d \in \mathbf{SD}\, (x \in \mathbb{I}_d \wedge \mathbf{S}(2x-d))$, we have the following cases.

Case $d = -1$. We have $-1 \le x \le 0 \wedge \mathbf{S}(2x+1)$ and thus $x \le 0$.

Case $d = 1$. We have $0 \le x \le 1 \wedge \mathbf{S}(2x-1)$ and thus $x \ge 0$.

Case $d = 0$. We have $|x| \le 1/2 \wedge \mathbf{S}(2x)$. In addition, we always have $\mathbf{B}(2x) \to \mathbf{B}(x)$ (realized by id). This completes the proof of *Claim 1*.

Claim 2. $\forall x(\mathbf{S}(x) \to \mathbf{S}(\mathbf{t}(x)))$.

We set $\mathbf{S}'(y) \stackrel{\text{Def}}{=} \exists x \in \mathbf{S}\ y = \mathbf{t}(x)$ and show $\mathbf{S}' \subseteq \mathbf{S}$ by half-strong coinduction. Therefore, we show

$$\mathbf{S}'(y) \to \exists d \in \mathbf{SD}(y \in \mathbb{I}_d \wedge \mathbf{S}'(2y - d)) \vee \mathbf{S}(y). \tag{29}$$

Assume $\mathbf{S}'(y)$, i.e., $y = \mathbf{t}(x)$ for an x that satisfies $\mathbf{S}(x)$.

Case $-1 \le x \le 0 \wedge \mathbf{S}(2x + 1)$. Then $2x + 1 = \mathbf{t}(x) = y$. Hence we have $\mathbf{S}(y)$.

Case $0 \le x \le 1 \wedge \mathbf{S}(2x - 1)$. Then $2x - 1 = -\mathbf{t}(x) = -y$ and hence $\mathbf{S}(-y)$. Therefore, by Lemma 23, we have $\mathbf{S}(y)$.

Case $|x| \le 1/2 \wedge \mathbf{S}(2x)$. Then $y = \mathbf{t}(x) \ge 0$ and thus $y \in \mathbb{I}_1$. Hence it suffices to show $\mathbf{S}'(2y-1)$. We have $2y - 1 = 1 - 4|x| = \mathbf{t}(2x)$ and therefore $\mathbf{S}'(2y - 1)$ holds. This completes the proof of *Claim 2* and hence the proof of the theorem. □

We extract a program from this proof.

Program from Claim 1. The program $\mathsf{step2} : \mathbf{3}^\omega \Rightarrow (\mathbf{2} + \mathbf{3}^\omega \times (\mathbf{2} \Rightarrow \mathbf{2}))$ extracted from the proof of (28) is

$$\begin{aligned}\mathsf{step2}\, p \stackrel{\text{Def}}{=} \mathbf{case}\,(\pi_{\mathbf{Left}}\, p)\,\mathbf{of}\,\{ &-1 \to \mathbf{Left}(\mathbf{Left}\,\mathbf{Nil});\\ &1 \to \mathbf{Left}(\mathbf{Right}\,\mathbf{Nil});\\ &0 \to \mathbf{Right}(\mathbf{Pair}(\pi_{\mathbf{Right}}\ p, \mathsf{id}))\}.\end{aligned}$$

Therefore, by (20) of Lemma 22, the extracted realizer of $\mathbf{S}(x) \to \mathbf{D}(x)$ is $\mathsf{sgh} : \mathbf{3}^\omega \Rightarrow \mathbf{2}$,

$$\mathsf{sgh}\, p \stackrel{\text{rec}}{=} \mathbf{case}\,(\mathsf{step2}\, p)\,\mathbf{of}\,\{\mathbf{Left}(b) \to b; \mathbf{Right}(q, g) \to g(\mathsf{sgh}\, q)\}\,.$$

By rewriting a nested case expression using Lemma 12, we have

$$\mathsf{sgh}\, p \stackrel{\text{rec}}{=} \mathbf{case}\,(\pi_{\mathbf{Left}}\, p)\,\mathbf{of}\,\{-1 \to \mathbf{L}; 1 \to \mathbf{R}; 0 \to \mathsf{sgh}(\pi_{\mathbf{Right}}\, p)\}\,. \tag{30}$$

Note that $\mathsf{sgh}(0 : 0 : \ldots) = \bot$. This can be seen by applying Scott induction (Axiom (viii)) to the predicate $P \stackrel{\text{Def}}{=} \lambda b\ (b\,(0 : 0 : \ldots) = \bot)$ and $a \stackrel{\text{Def}}{=} \lambda b.\, \lambda p.\, \mathbf{case}\,(\pi_{\mathbf{Left}}\, p)\,\mathbf{of}\,\{-1 \to \mathbf{L}; 1 \to \mathbf{R}; 0 \to b(\pi_{\mathbf{Right}}\, p)\}$.

Program from Claim 2. The program extracted from the proof of (29) is $\mathsf{step3} : \mathbf{3}^\omega \Rightarrow \mathbf{3} \times \mathbf{3}^\omega + \mathbf{3}^\omega$,

$$\begin{aligned}\mathsf{step3}\ p \stackrel{\text{Def}}{=} \mathbf{case}\,(\pi_{\mathbf{Left}}\, p)\,\mathbf{of}\,\{&-1 \to \mathbf{Right}(\pi_{\mathbf{Right}}\, p);\\ &1 \to \mathbf{Right}(\mathsf{minus}(\pi_{\mathbf{Right}}\, p));\\ &0 \to \mathbf{Left}\,(\mathbf{Pair}(1, \pi_{\mathbf{Right}}\, p))\}\,.\end{aligned}$$

Therefore, according to equation (19) of Sect. 4.2, the program extracted from the proof of $\mathbf{S}(x) \to \mathbf{S}(t(x))$ is $\mathsf{sgt} : \mathbf{3}^\omega \Rightarrow \mathbf{3}^\omega$,

$$\mathsf{sgt}\ p \stackrel{\text{rec}}{=} [(\mathsf{mon}\,\mathsf{sgt}) + \mathbf{id}](\mathsf{step3}\ p)\,.$$

This definition can be simplified to (using again Lemma 12),

$$\begin{aligned}\mathsf{sgt}\, p \stackrel{\text{rec}}{=} \mathbf{case}(\pi_{\mathbf{Left}}\, p)\mathbf{of}\{&-1 \to \pi_{\mathbf{Right}}\, p; 1 \to \mathsf{minus}(\pi_{\mathbf{Right}}\, p);\\ &0 \to \mathbf{Pair}(1, \mathsf{sgt}(\pi_{\mathbf{Right}}\, p))\}\,.\end{aligned} \tag{31}$$

Now, by equation (18) of Sect. 4.2, the extracted program $\mathsf{stog} : \mathbf{3}^\omega \Rightarrow \mathbf{2}^\omega$ from the proof of $\mathbf{S} \subseteq \mathbf{G}$ is $\mathsf{stog} \stackrel{\text{rec}}{=} (\mathsf{mon}\,\mathsf{stog}) \circ \mathsf{step4}$ with $\mathsf{step4} : \mathbf{3}^\omega \Rightarrow \mathbf{2}^\omega \times \mathbf{3}^\omega$, $\mathsf{step4}\ p = \mathbf{Pair}(\mathsf{sgh}\ p, \mathsf{sgt}\ p)$. This simplifies to

$$\mathsf{stog}\, p \stackrel{\text{rec}}{=} \mathbf{Pair}(\mathsf{sgh}\, p, \mathsf{stog}(\mathsf{sgt}\, p)). \tag{32}$$

Note that since $\mathsf{sgh}(0:0:\ldots) = \bot$ the first digit of $\mathsf{stog}(0:0:\ldots)$ is $\bot$ and therefore $\mathsf{stog}(0:0:\ldots)$ evaluates to $\bot : \mathbf{R} : \mathbf{L} : \mathbf{L} : \ldots$. We will study this evaluation in Example 3, at the end of this paper.

Thus, we have obtained a program that consists of four recursions. In the rest of this section, we transform this program into a program with one recursion. We use the list notation $a : p$ for $\mathbf{Pair}(a, p)$ and write head for $\pi_{\mathbf{Left}}$ and tail for $\pi_{\mathbf{Right}}$.

First, by Scott-induction it is easy to see the equivalence of (32) to the following program provided p is restricted to total elements of $\mathbf{3}^\omega$, that is, elements of $\mathbf{3}_t^\omega$ where $\mathbf{3}_t^\omega(a) \stackrel{\nu}{=} \mathsf{head}\, a \in \{-1, 0, 1\} \wedge \mathbf{3}_t^\omega(\mathsf{tail}\, a)$.

$$\begin{array}{lcl}
\mathsf{stog}\, p & \stackrel{\text{rec}}{=} & \mathbf{case}\ (\mathsf{head}\, p)\ \mathbf{of}\ \{ \\
-1 & \to & \mathbf{L} : \mathsf{stog}\, (\mathsf{tail}\, p)\,; \\
1 & \to & \mathbf{R} : \mathsf{stog}(\mathsf{minus}\, (\mathsf{tail}\, p))\,; \\
0 & \to & \mathsf{sgh}\, (\mathsf{tail}\, p) : \mathsf{stog}\, (1 : \mathsf{sgt}\, (\mathsf{tail}\, p)) \\
\} &&
\end{array}$$

Note that the two programs are not equal for $p = \bot$ since $\mathsf{stog}\, \bot$ is equal to $\bot : \mathsf{stog}\, \bot$ with the old definition (32) of stog, whereas $\mathsf{stog}\, \bot = \bot$ with the new definition of stog. However, since all realizers of $\mathbf{S}$ are total (easy proof by coinduction), both programs realize $\mathbf{S} \subseteq \mathbf{G}$. Therefore we use the same name stog for both.

We now show that the new definition of stog can be simplified.

By strong coinduction (Sect. 2) one can easily prove $G(-x) \to G(x)$. The extracted program $\mathsf{nh} : \mathbf{2}^\omega \Rightarrow \mathbf{2}^\omega$ inverts the first digit of a Gray code.

$$\mathsf{inv}\, a = \mathbf{case}\ a\ \mathbf{of}\ \{\, \mathbf{L} \to \mathbf{R}\,;\, \mathbf{R} \to \mathbf{L}\,\}$$

$$\mathsf{nh}\, q = (\mathsf{inv}\, (\mathsf{head}\, q)) : (\mathsf{tail}\, q)$$

One can also show, using Scott-induction, that $\mathsf{sgh}(\mathsf{minus}\, p) = \mathsf{inv}\, (\mathsf{sgh}\, p)$ and $\mathsf{sgt}(\mathsf{minus}\, p) = \mathsf{sgt}\, p$. Therefore, for total p,

$$\begin{aligned}
\mathsf{stog}\, (\mathsf{minus}\, p) &= \mathsf{sgh}\, (\mathsf{minus}\, p) : \mathsf{stog}\, (\mathsf{sgt}\, (\mathsf{minus}\, p)) \\
&= \mathsf{inv}\, (\mathsf{sgh}\, p) : \mathsf{stog}\, (\mathsf{sgt}\, p) \\
&= \mathsf{nh}\, (\mathsf{stog}\, p)\,.
\end{aligned}$$

With this equation, we can simplify stog as follows.

$$\begin{array}{lcl}
\mathsf{stog}\, p & = & \mathbf{case}\ (\mathsf{head}\, p)\ \mathbf{of}\ \{ \\
-1 & \to & \mathbf{L} : \mathsf{stog}\, (\mathsf{tail}\, p)\,; \\
1 & \to & \mathbf{R} : \mathsf{nh}\, (\mathsf{stog}\, (\mathsf{tail}\, p))\,; \\
0 & \to & \mathsf{sgh}\, (\mathsf{tail}\, p) : \mathsf{stog}\, (1 : \mathsf{sgt}\, (\mathsf{tail}\, p)) \\
\} &&
\end{array}$$

The last case further simplifies to $0 \to \mathsf{sgh}\, (\mathsf{tail}\, p) : \mathbf{R} : \mathsf{nh}\, (\mathsf{stog}\, (\mathsf{sgt}\, (\mathsf{tail}\, p)))$ by expanding stog. Since $\mathsf{stog}\, p = \mathsf{sgh}\, p : \mathsf{stog}\, (\mathsf{sgt}\, p)$, one can further rewrite the definition of stog using the let notation $\mathbf{let}\ q = M\ \mathbf{in}\ N$ for $(\lambda q.\, N)\, M$.

$$
\begin{array}{lcl}
\mathsf{stog}\, p & = & \mathbf{case}\ (\mathsf{head}\, p)\ \mathbf{of}\ \{ \\
-1 & \to & \mathbf{L} : \mathsf{stog}\, (\mathsf{tail}\, p)\, ; \\
1 & \to & \mathbf{R} : \mathsf{nh}\, (\mathsf{stog}\, (\mathsf{tail}\, p))\, ; \\
0 & \to & \mathbf{let}\ q = \mathsf{stog}\, (\mathsf{tail}\, p)\ \mathbf{in}\ (\mathsf{head}\, q) : \mathbf{R} : \mathsf{nh}\, (\mathsf{tail}\, q) \\
\} &&
\end{array}
$$

The above equation holds for total p. Viewing it as recursive definition (replacing '$=$' by '$\overset{\text{rec}}{=}$') one obtains a program which coincides with the previous one on total arguments (proof by Scott-induction) and hence realizes $\mathbf{S} \subseteq \mathbf{G}$. It is precisely the Haskell program of signed digit to Gray code conversion in [72] if we view $:$, head and tail as ordinary list operations.

6. Operational semantics

The Soundness Theorem (Theorem 2) shows that from an IFP-proof of a formula one can extract a program realizing it, provably in RIFP. Because the program axioms of RIFP are correct w.r.t. the domain-theoretic semantics, this theorem shows that the denotational semantics of a program extracted from an IFP proof is a correct realizer of the formula. However, so far we have no means to *run* the extracted programs in order to *compute* data that realize the formula. In this section we address this issue by defining an operational semantics and showing that it fits the denotational semantics through two Computational Adequacy Theorems (Theorems 5, 6). The first is essentially an untyped version of Plotkin's Adequacy Theorem for the simply typed language PCF [58]. Its proof uses compact elements of the untyped domain model as a replacement for types, a technique introduced by Coquand and Spiwack [26], and follows roughly the lines of [8]. The second Adequacy Theorem concerns the computation of infinite data. A related result for an extension of PCF by real numbers was obtained by Escardo [31]. While Escardo works in a typed setting and concerns incremental computation on the interval domain, our result is untyped and computes arbitrary infinite data built from constructors. There exists a rich literature on computational adequacy covering, for example, typed lambda calculi with various effects [57,48], denotational semantics based on games or categories [27,66], and axiomatic approaches [22,30].

In the following we work with our untyped programming language that includes programs not typable with our type system, and consider types only in Section 6.5. This shows that the operational properties of our programs are independent of the type system.

6.1. Inductive and coinductive definitions of data

First we make precise what we mean by data. Recall from Sect. 3.1 that programs are interpreted in the domain D defined by the recursive domain equation

$$D = (\mathbf{Nil} + \mathbf{Left}(D) + \mathbf{Right}(D) + \mathbf{Pair}(D \times D) + \mathbf{Fun}(D \to D))_\perp\, .$$

We consider the sub-domain E of D built from constructors only

$$E = (\mathbf{Nil} + \mathbf{Left}(E) + \mathbf{Right}(E) + \mathbf{Pair}(E \times E))_\perp$$

and call its elements *data*. We also define various predicates on D as least or greatest fixed points of the following operators Φ and $\Phi_\perp$ of arity (δ). The definitions and proofs below take place in informal mathematics although we take advantage of the notations and proof rules provided by the formal system IFP regarding inductive and coinductive definitions.

$$\Phi(X)(a) \stackrel{\text{Def}}{=} \bigvee_{\substack{C \\ \text{constructor}}} \left(\exists a_1, \ldots, a_k \;\; a = C(a_1, \ldots, a_k) \wedge \bigwedge_{i \leq k} X(a_i) \right)$$

and its variant $\Phi_\perp$ obtained by adding $\perp$ as an option

$$\Phi_\perp(X)(a) \stackrel{\text{Def}}{=} a = \perp \vee \Phi(X)(a)\,.$$

We have

$$E = \nu(\Phi_\perp) \qquad \text{(arbitrary data)}$$

and we define

$$\begin{aligned} E_{\mathrm{f}} &\stackrel{\text{Def}}{=} \mu(\Phi_\perp) && \text{(finite data)} \\ E_{\mathrm{t}} &\stackrel{\text{Def}}{=} \nu(\Phi) && \text{(total data)} \\ E_{\mathrm{ft}} &\stackrel{\text{Def}}{=} \mu(\Phi) && \text{(finite total data)} \end{aligned}$$

It is easy to see that E_{f} consists of the compact data, E_{t} of the data containing no $\perp$, and $E_{\mathrm{ft}} = E_{\mathrm{f}} \cap E_{\mathrm{t}}$, hence our choice of names.

Using binary versions of the operators Φ and $\Phi_\perp$,

$$\begin{aligned} \Phi^2(X)(a,b) &\stackrel{\text{Def}}{=} \bigvee_C \left(\begin{array}{r} \exists a_1, \ldots, a_k, b_1, \ldots, b_k \; a = C(a_1, \ldots, a_k) \wedge \\ b = C(b_1, \ldots, b_k) \wedge \bigwedge_{i \leq k} X(a_i, b_i) \end{array} \right) \\ \Phi^2_\perp(X)(a,b) &\stackrel{\text{Def}}{=} a = \perp \vee \Phi^2(X)(a,b) \\ \Phi^2_{\perp,\perp}(X)(a,b) &\stackrel{\text{Def}}{=} a = b = \perp \vee \Phi^2(X)(a,b) \end{aligned}$$

we define the relations

$$\begin{aligned} a \sqsubseteq_E b &\stackrel{\text{Def}}{=} \nu(\Phi^2_\perp)(a,b) && \text{(domain ordering on } E) \\ \mathrm{appr}(a,b) &\stackrel{\text{Def}}{=} \mu(\Phi^2_\perp)(a,b) && \text{(finite approximation)} \\ \mathrm{eq}(a,b) &\stackrel{\text{Def}}{=} \nu(\Phi^2_{\perp,\perp})(a,b) && \text{(bisimilarity)} \\ \mathrm{teq}(a,b) &\stackrel{\text{Def}}{=} \nu(\Phi^2)(a,b) && \text{(total bisimilarity)} \end{aligned}$$

Note that $\sqsubseteq_E$ coincides with the domain order $\sqsubseteq$ on E but not with that on D. $a \sqsubseteq_E b$ implies $a \in E$, by coinduction, therefore $\sqsubseteq_E$ is not reflexive on $D \setminus E$. Clearly, $\mathrm{appr}(a,b)$ holds iff $a \sqsubseteq_E b$ and $E_{\mathrm{f}}(a)$, and $\mathrm{teq}(a,b)$ holds iff $\mathrm{eq}(a,b)$ and $a, b \in E_{\mathrm{t}}$. If we replace in the definition of $\mathrm{eq}(a,b)$ the largest fixed point ν by the least fixed point μ, we obtain the relation $\mu(\Phi^2_{\perp,\perp})(a,b)$ which clearly implies that a and b are equal elements of E_{f} (easy inductive argument). However,

$$\forall a, b\, (\mathrm{eq}(a,b) \to a = b) \tag{33}$$

is a non-trivial assertion expressing that the elements of E are completely determined by their constructors, which we use in this section. From (33) one can derive the equivalence $(a = b \wedge a, b \in E) \leftrightarrow (a \sqsubseteq_E b \wedge b \sqsubseteq_E a)$ and the maximality of the elements in E_{t}, $(a \sqsubseteq_E b \wedge E_{\mathrm{t}}(a)) \to a = b$. We prove the following lemma to give typical examples of inductive and coinductive proofs on data.

Lemma 24.

(a) $\mathrm{appr}(a,b)$ *iff* $E_{\mathrm{f}}(a) \wedge a \sqsubseteq_E b$.
(b) $a \sqsubseteq_E b$ *iff* $E(a) \wedge \forall d(\mathrm{appr}(d,a) \to \mathrm{appr}(d,b))$.

Proof. (a) Left to right is by induction on $\mathrm{appr}(a,b)$. Right to left is induction on $E_{\mathrm{f}}(a)$ to prove that $E_{\mathrm{f}}(a) \to \forall b(a \sqsubseteq_E b \to \mathrm{appr}(a,b))$. We show $\forall a\,(\Phi_\perp(P)(a) \to P(a))$ for $P(a) = \forall b(a \sqsubseteq_E b \to \mathrm{appr}(a,b))$. Suppose that $\Phi_\perp(P)(a)$. If $a = \perp$, then $P(\perp)$. If $a = C(a_1,\dots,a_k) \wedge \bigwedge_{i\le k} P(a_i)$ and $a \sqsubseteq_E b$, let $b = C(b_1,\dots,b_k)$. We have $a_i \sqsubseteq_E b_i$ and thus $\mathrm{appr}(a_i,b_i)$ by $P(a_i)$.

(b) Left to right is immediate by (a). Right to left is by coinduction on $a \sqsubseteq_E b$. Let $P(a,b) \stackrel{\mathrm{Def}}{=} E(a) \wedge \forall d(\mathrm{appr}(d,a) \to \mathrm{appr}(d,b))$. We need to show $\forall a,b\,(P(a,b) \to \Phi^2_\perp(P)(a,b))$. Because $E(a)$, $a = \perp$ or a has the form $C(a_1,\dots,a_k)$ with $a_1,\dots,a_k \in E$. If $a = \perp$, then $\Phi^2_\perp(P)(a,b)$ holds. If a has the form $C(a_1,\dots,a_k)$, we have $\mathrm{appr}(C(\perp^k),a)$ and thus $\mathrm{appr}(C(\perp^k),b)$ by $P(a,b)$. Therefore, $b = C(b_1,\dots,b_k)$ for some b_i. We need to show that $P(a_i,b_i)$. If $\mathrm{appr}(d,a_i)$, then $\mathrm{appr}(C(\perp^i,d,\perp^{k-i-1}),a)$. Hence, $\mathrm{appr}(C(\perp^i,d,\perp^{k-i-1}),b)$, and thus $\mathrm{appr}(d,b_i)$. □

6.2. Inductively and coinductively defined reduction relations

We define four reduction relations between closed programs and data through induction and coinduction. These relations are related to computational procedures in Sect. 6.4. In order to treat programs as syntactic objects, we introduce a new sort π of programs and use $M,N,K,\dots$ for variables of sort π. When a program is considered as an element of π, we use $x,y,\dots$ as names for program variables while we use $a,b,\dots$ to denote elements of D.

A *value* is a closed program M that begins with a constructor or has the form $\lambda x.\,M$. Following [8], we first define inductively a *bigstep reduction relation* $M \Downarrow V$ between closed programs M and values V as follows:

(i) $V \Downarrow V$

(ii) $$\frac{M \Downarrow C(\vec{M}) \quad N[\vec{M}/\vec{y}] \Downarrow V}{\mathbf{case}\,M\,\mathbf{of}\,\{\dots;C(\vec{y}) \to N;\dots\} \Downarrow V}$$

(iii) $$\frac{M \Downarrow \lambda x.\,M' \qquad M'[N/x] \Downarrow V}{M\,N \Downarrow V}$$

(iv) $$\frac{M\,(\mathbf{rec}\,M) \Downarrow V}{\mathbf{rec}\,M \Downarrow V}$$

Lemma 25. *For a closed program M, there is at most one value V such that $M \Downarrow V$.*

Proof. There is at most one $\Downarrow$ reduction rule applicable to a closed program. □

Since bigstep reduction stops at constructors (due to rule (i)), in order to obtain a data, we need to continue computation under constructors. We define four reduction relations $M \stackrel{\mu}{\Longrightarrow} a$, $M \stackrel{\mu\perp}{\Longrightarrow} a$, $M \stackrel{\nu}{\Longrightarrow} a$, $M \stackrel{\nu\perp}{\Longrightarrow} a$, all of arity (π,δ), as least and greatest fixed points of the operators

$$\Phi^{\mathrm{op}}(X)(M,a) \stackrel{\mathrm{Def}}{=} \bigvee_C \left(\begin{array}{r} \exists M_1,\dots,M_k,a_1,\dots,a_k\ (M \Downarrow C(M_1,\dots,M_k) \\ \wedge\, a = C(a_1,\dots,a_k) \wedge \bigwedge_{i\le k} X(M_i,a_i)) \end{array} \right)$$

$$\Phi^{\mathrm{op}}_\perp(X)(M,a) \stackrel{\mathrm{Def}}{=} a = \perp \vee \Phi^{\mathrm{op}}(X)(M,a)\,.$$

Here again, C ranges over constructors. Now we define

$$\overset{\mu}{\Longrightarrow} \overset{\text{Def}}{=} \mu(\Phi^{\text{op}})$$
$$\overset{\nu}{\Longrightarrow} \overset{\text{Def}}{=} \nu(\Phi^{\text{op}})$$
$$\overset{\mu\bot}{\Longrightarrow} \overset{\text{Def}}{=} \mu(\Phi^{\text{op}}_{\bot})$$
$$\overset{\nu\bot}{\Longrightarrow} \overset{\text{Def}}{=} \nu(\Phi^{\text{op}}_{\bot}).$$

Note that the definition of $M \overset{\mu}{\Longrightarrow} a$ is equivalent to an inductive definition by the following reduction rules.

$$\frac{M \Downarrow \mathbf{Nil}}{M \overset{\mu}{\Longrightarrow} \mathbf{Nil}} \qquad \frac{M \Downarrow \mathbf{Pair}(M_1, M_2) \quad M_1 \overset{\mu}{\Longrightarrow} a_1 \quad M_2 \overset{\mu}{\Longrightarrow} a_2}{M \overset{\mu}{\Longrightarrow} \mathbf{Pair}(a_1, a_2)}$$

$$\frac{M \Downarrow \mathbf{Left}(M) \quad M \overset{\mu}{\Longrightarrow} a}{M \overset{\mu}{\Longrightarrow} \mathbf{Left}(a)} \qquad \frac{M \Downarrow \mathbf{Right}(M) \quad M \overset{\mu}{\Longrightarrow} a}{M \overset{\mu}{\Longrightarrow} \mathbf{Right}(a)}$$

$\overset{\nu}{\Longrightarrow}$ can be defined by replacing in the rules above μ with ν and interpreting the rules coinductively, that is, permitting infinite derivations. $\overset{\mu\bot}{\Longrightarrow}$ and $\overset{\nu\bot}{\Longrightarrow}$ are obtained by adding the axioms $M \overset{\mu\bot}{\Longrightarrow} \bot$ and $M \overset{\nu\bot}{\Longrightarrow} \bot$ respectively.

$M \overset{\mu}{\Longrightarrow} a$ is the finite reduction to a finite total data and $M \overset{\nu}{\Longrightarrow} a$ is the (possibly) infinite reduction to a (possibly) infinite total data. $M \overset{\mu\bot}{\Longrightarrow} a$ and $M \overset{\nu\bot}{\Longrightarrow} a$ are reductions that may leave some part unreduced by assigning $\bot$, and are used to obtain observations of infinite data through finite approximations. For example, for $M = \mathbf{rec}(\lambda x.\mathbf{Pair}(\mathbf{Nil}, x))$, no $a \in D$ satisfies $M \overset{\mu}{\Longrightarrow} a$ but

$$M \overset{\nu}{\Longrightarrow} \mathbf{Nil} : \mathbf{Nil} : \mathbf{Nil} : \ldots$$
$$M \overset{\nu\bot}{\Longrightarrow} \bot : \mathbf{Nil} : \mathbf{Nil} : \ldots$$
$$M \overset{\mu\bot}{\Longrightarrow} \bot : \mathbf{Nil} : \bot \qquad (= \mathbf{Pair}(\mathbf{Pair}(\bot, \mathbf{Nil}), \bot)).$$

Lemma 26.

(a) $M \overset{\mu}{\Longrightarrow} a$ *iff* $M \overset{\mu\bot}{\Longrightarrow} a \wedge E_{\text{ft}}(a)$.
(b) $M \overset{\nu}{\Longrightarrow} a$ *iff* $M \overset{\nu\bot}{\Longrightarrow} a \wedge E_{\text{t}}(a)$.
(c) $M \overset{\mu\bot}{\Longrightarrow} a$ *iff* $M \overset{\nu\bot}{\Longrightarrow} a \wedge E_{\text{f}}(a)$.
(d) $M \overset{\nu\bot}{\Longrightarrow} a$ *iff* $\forall d\,(\text{appr}(d, a) \to M \overset{\mu\bot}{\Longrightarrow} d) \wedge E(a)$.

Proof. (a) By induction on $\overset{\mu}{\Longrightarrow}$ and $\overset{\mu\bot}{\Longrightarrow}$.

(b) By coinduction on $\overset{\nu}{\Longrightarrow}$ and $\overset{\nu\bot}{\Longrightarrow}$.

(c) Left to right is immediate induction on $\overset{\mu\bot}{\Longrightarrow}$. Right to left is by induction on $E_{\text{f}}(a)$.

(d) Right to left by coinduction on $\overset{\nu\bot}{\Longrightarrow}$. For $P(M, a) \overset{\text{Def}}{=} \forall d\,(\text{appr}(d, a) \to M \overset{\mu\bot}{\Longrightarrow} d) \wedge E(a)$, we prove $P(M, a) \to \Phi^{\text{op}}_{\bot}(P)(M, a)$. Suppose that $P(M, a)$. Since $a \in E$, $a = \bot$ or a has the form $C(a_1, \ldots, a_k)$ for $a_i \in E$. If $a = \bot$, then we have $\Phi^{\text{op}}_{\bot}(P)(M, a)$. If $a = C(a_1, \ldots, a_k)$, then $\text{appr}(C(\bot^k), a)$ and therefore $M \overset{\mu\bot}{\Longrightarrow} C(\bot^k)$. Hence, $M \Downarrow C(M_1, \ldots, M_k)$ for some $M_1, \ldots, M_k$. We need to show $P(M_i, a_i)$ for each $i \leq k$. Suppose that $\text{appr}(d', a_i)$ and let $d = C(\bot^{i-1}, d', \bot^{k-i})$. Since $\text{appr}(d, a)$, we have $M \overset{\mu\bot}{\Longrightarrow} d$. Therefore, $M_i \overset{\mu\bot}{\Longrightarrow} d'$.

Left to right: Suppose $M \overset{\nu\perp}{\Longrightarrow} a$. We have $E(a)$ by coinduction on E. We show that $\mathrm{appr}(d,a)$ implies $M \overset{\mu\perp}{\Longrightarrow} d$. $\mathrm{appr}(d,a)$ implies $E_{\mathrm{f}}(d)$ by Lemma 24 (a). On the other hand, $M \overset{\nu\perp}{\Longrightarrow} a$ and $\mathrm{appr}(d,a)$ imply $M \overset{\nu\perp}{\Longrightarrow} d$ by coinduction. Therefore, by part (c), $M \overset{\mu\perp}{\Longrightarrow} d$. □

6.3. Computational adequacy theorem

Now we prove our first result linking the denotational with the operational semantics.

Theorem 5 *(Computational Adequacy I). Let M be a closed program.*

(a) $M \overset{\mu}{\Longrightarrow} a$ *iff* $a = [\![M]\!] \wedge E_{\mathrm{ft}}(a)$.
(b) $M \overset{\mu\perp}{\Longrightarrow} a$ *iff* $a \sqsubseteq_E [\![M]\!] \wedge E_{\mathrm{f}}(a)$.
(c) $M \overset{\nu}{\Longrightarrow} a$ *iff* $a = [\![M]\!] \wedge E_{\mathrm{t}}(a)$.
(d) $M \overset{\nu\perp}{\Longrightarrow} a$ *iff* $a \sqsubseteq_E [\![M]\!]$.

Note in (d) that $a \sqsubseteq_E [\![M]\!]$ implies $E(a)$. The proof of the theorem will be given through the following Lemmas 27-33. Computational adequacy usually means (a), and (c) is its generalization to infinite total data. As we will see in Lemma 27, (b) and (d) are proved as lemmas for (a) and (c). They are also foundations for the second adequacy theorem (Theorem 6).

Lemma 27. *In Theorem 5, part (b) implies part (a), and part (d) implies part (c).*

(b) implies (a). : $M \overset{\mu}{\Longrightarrow} a$ implies $E_{\mathrm{ft}}(a)$ by Lemma 26 (a). In addition, if $E_{\mathrm{ft}}(a)$ holds, then $M \overset{\mu}{\Longrightarrow} a$ and $M \overset{\mu\perp}{\Longrightarrow} a$ are equivalent by Lemma 26 (a), and $a \sqsubseteq_E b$ and $a = b$ are equivalent as we mentioned before Lemma 24.

[(d) implies (c)]: Similar. Note that, by (33), $=$ on E is the bisimulation relation. □

Due to this lemma, we only need to prove (b) and (d). The 'only if' parts of (b) and (d) are obtained by the following lemma.

Lemma 28 *(Correctness).*

(a) *If $M \Downarrow V$, then $[\![M]\!] = [\![V]\!]$.*
(b) *If $M \overset{\mu\perp}{\Longrightarrow} a$, then* $\mathrm{appr}(a, [\![M]\!])$.
(c) *If $M \overset{\nu\perp}{\Longrightarrow} a$, then $a \sqsubseteq_E [\![M]\!]$.*

Proof. (a) is proven by induction along the definition of $M \Downarrow V$.

(b) We define $P(M,a) \overset{\mathrm{Def}}{=} \mathrm{appr}(a, [\![M]\!])$ and prove $M \overset{\mu\perp}{\Longrightarrow} a \to P(M,a)$ by induction. Therefore, we prove $\Phi^{\mathrm{op}}_{\perp}(P)(M,a) \to P(M,a)$. Suppose that $\Phi^{\mathrm{op}}_{\perp}(P)(M,a)$. If $a = \perp$, then we have $P(M,a)$. If $\Phi^{\mathrm{op}}(P)(M,a)$, then $M \Downarrow C(M_1,\ldots,M_k)$, $a = C(a_1,\ldots,a_k)$, and $P(M_i,a_i)$ for every $i \leq k$. Hence, by (a), $[\![M]\!] = [\![C(M_1,\ldots,M_k)]\!] = C([\![M_1]\!],\ldots,[\![M_k]\!])$. Since $P(M_i,a_i)$, we have $\mathrm{appr}(a_i,[\![M_i]\!])$ and therefore $\mathrm{appr}(a,[\![M]\!])$).

(c) By Lemma 24 (b), we need to show that $M \overset{\nu\perp}{\Longrightarrow} a$ and $\mathrm{appr}(d,a)$ implies $\mathrm{appr}(d,[\![M]\!])$. First, we can easily show that $M \overset{\nu\perp}{\Longrightarrow} a$ and $\mathrm{appr}(d,a)$ implies $M \overset{\nu\perp}{\Longrightarrow} d$. Since $M \overset{\nu\perp}{\Longrightarrow} d$ and $E_{\mathrm{f}}(d)$, we have $M \overset{\mu\perp}{\Longrightarrow} d$ by Lemma 26 (a). Therefore, $\mathrm{appr}(d,[\![M]\!])$ by (b). □

We prove the 'if' part of Theorem 5 (b) following [8], which uses ideas from [58] and [26]. Let D_0 be the set of compact elements of D. To every $a \in D_0$ we assign a set of closed programs $\mathrm{Pr}(a)$ by induction on $\mathbf{rk}(a)$ (Sect. 3.1).

$$\begin{aligned}
\mathrm{Pr}(\bot) &= \text{the set of all closed programs}\\
\mathrm{Pr}(C(a_1,\dots,a_k)) &= \{M \mid \exists M_1,\dots,M_k,\ M \Downarrow C(M_1,\dots,M_k) \wedge \bigwedge_{i\le k} M_i \in \mathrm{Pr}(a_i))\}\\
\mathrm{Pr}(\mathbf{Fun}(f)) &= \{M \mid \exists x, M',\ (M \Downarrow \lambda x.\, M' \wedge \forall b \in D_0\,(\mathbf{rk}(b) < \mathbf{rk}(\mathbf{Fun}(f)) \to \\
&\qquad \forall N \in \mathrm{Pr}(b)\,(M'[N/x] \in \mathrm{Pr}(f(b)))))\}
\end{aligned}$$

Note that for $a \in D_0 \cap E$ $(= E_{\mathrm{f}}(a))$, $M \in \mathrm{Pr}(a)$ is equivalent to $M \overset{\mu\bot}{\Longrightarrow} a$.

Lemma 29. *For $a, b \in D_0$, if $a \sqsubseteq b$, then $\mathrm{Pr}(a) \supseteq \mathrm{Pr}(b)$.*

Proof. As the proof of Lemma 12 in [8]. □

Lemma 30. *Suppose that $a \in D_0 \setminus \{\bot\}$. $M \in \mathrm{Pr}(a)$ iff $M \Downarrow V$ for some $V \in \mathrm{Pr}(a)$.*

Proof. Immediate from the definition of $\mathrm{Pr}(a)$. □

Lemma 31. *If $M \in \mathrm{Pr}(\mathbf{Fun}(f))$, then $\mathbf{rec}\, M \in \mathrm{Pr}(f^n(\bot))$ for every $n \in \mathbf{N}$.*

Proof. Induction on n. It is trivial for $n = 0$ because $\mathrm{Pr}(\bot)$ contains every closed program. Suppose that $\mathbf{rec}\, M \in \mathrm{Pr}(f^n(\bot))$. According $\mathbf{rk}2$, for $b = f^n(\bot)$, $f(b) = f(b_0)$ for some compact $b_0 \sqsubseteq b$ with $\mathbf{rk}(\mathbf{Fun}(f)) > \mathbf{rk}(b_0)$. Since $\mathbf{rec}\, M \in \mathrm{Pr}(b)$, we have $\mathbf{rec}\, M \in \mathrm{Pr}(b_0)$ by Lemma 29. Since $M \in \mathrm{Pr}(\mathbf{Fun}(f))$, $M \Downarrow \lambda x.K$ for some x and K and $\forall c \in D_0(\mathbf{rk}(c) < \mathbf{rk}(\mathbf{Fun}(f)) \to \forall N \in \mathrm{Pr}(c)(K[N/x] \in \mathrm{Pr}(f(c)))$. We apply this to the case $c = b_0$ and $N = \mathbf{rec}\ M$ and get $K[\mathbf{rec}\ M/x] \in \mathrm{Pr}(f(b_0)) = \mathrm{Pr}(f^{n+1}(\bot))$. Therefore, $K[\mathbf{rec}\ M/x] \Downarrow V$ and $V \in \mathrm{Pr}(f^{n+1}(\bot))$. Thus, we also have $\mathbf{rec}\, M \Downarrow V$ and therefore $\mathbf{rec}\, M \in \mathrm{Pr}(f^{n+1}(\bot))$, by Lemma 30. □

Lemma 32 *(Approximation). For a closed program M and $a \in D_0$, if $a \sqsubseteq [\![M]\!]$, then $M \in \mathrm{Pr}(a)$.*

Proof. We show a more general statement about arbitrary programs involving substitutions and environments to take care of free variables. A substitution is a finite mapping from variables to the set of closed programs. An environment is a finite mapping from variables to D. For a substitution θ and an environment η, we write $\theta \in \mathrm{Pr}(\eta)$ if $\eta(x)$ is compact and $\theta(x) \in \mathrm{Pr}(\eta(x))$ for each $x \in \mathrm{dom}(\theta)$. We prove by induction on M:

For an environment η, a substitution θ such that $\theta \in \mathrm{Pr}(\eta)$, a program M such that $FV(M) \subseteq \mathrm{dom}(\theta)$ and $a \in D_0$, if $a \sqsubseteq [\![M]\!]\eta$ then $M\theta \in \mathrm{Pr}(a)$.

Since the statement is clear for $a = \bot$, we assume $a \neq \bot$. We may also assume $M \neq \bot$ since otherwise the condition $a \sqsubseteq [\![M]\!]\eta$ is not satisfied. The cases that M is x, $C(N_1,\dots,N_k)$, $\mathbf{case}\, M'\, \mathbf{of}\, \{\dots; C(\vec{y}) \to K; \dots\}$, $\lambda x.\, M'$, $M'\ N$ are similar to the corresponding cases of Lemma 15 in [8]. We only consider the case $M = \mathbf{rec}\, N$. Suppose that $a \sqsubseteq [\![M]\!]\eta$. Since $a \neq \bot$, $[\![N]\!]\eta = \mathbf{Fun}(g)$ for some continuous function $g : D \to D$

such that $[\![M]\!]\eta$ is the least fixed point of g. Therefore, $a \sqsubseteq g^n(\bot)$ for some n. By continuity, there is a compact $f \in D \to D$ such that $f \sqsubseteq g$ and $a \sqsubseteq f^n(\bot)$. Since $\mathbf{Fun}(f) \sqsubseteq [\![N]\!]\eta$, by induction hypothesis, $N\theta \in \mathrm{Pr}(\mathbf{Fun}(f))$. By Lemma 31, $\mathbf{rec}\,(N\theta) \in \mathrm{Pr}(f^n(\bot))$. By Lemma 29, $\mathrm{Pr}(a) \supseteq \mathrm{Pr}(f^n(\bot))$. Therefore, $M\theta = \mathbf{rec}\,(N\theta) \in \mathrm{Pr}(a)$. □

Proof of the if part of Theorem 5 (b). Suppose that $d \sqsubseteq_E [\![M]\!]$ for a finite data d. Then, $M \in \mathrm{Pr}(d)$ by the Approximation Lemma. Therefore, by the remark after the definition of $\mathrm{Pr}(a)$, we have $M \overset{\mu\bot}{\Longrightarrow} d$. □

Lemma 33. *If $[\![M]\!]$ has the form $C(a_1, \ldots, a_k)$, then $M \Downarrow C(M_1, \ldots, M_k)$ for some $M_1, \ldots, M_k$.*

Proof. Let $a = C(\bot, \ldots, \bot)$. If $[\![M]\!]$ has the form $C(a_1, \ldots, a_k)$, then $a \sqsubseteq_E [\![M]\!]$. By applying Theorem 5 (b), we obtain $M \overset{\mu\bot}{\Longrightarrow} a$. Thus, $M \Downarrow C(M_1, \ldots, M_k)$ for some $M_1, \ldots, M_k$. □

Completing the proof of the first Adequacy Theorem. Finally, we prove the 'if' part of (d) of Theorem 5. We prove by coinduction that $a \sqsubseteq_E [\![M]\!]$ implies $M \overset{\nu\bot}{\Longrightarrow} a$. Therefore, for $a \in D$ and a closed program M, we show

$$a \sqsubseteq_E [\![M]\!] \to a = \bot \vee \bigvee_C \left(\begin{array}{r} \exists M_1, \ldots, M_k, a_1, \ldots, a_k \; (M \Downarrow C(M_1, \ldots, M_k) \\ \wedge a = C(a_1, \ldots, a_k) \wedge \bigwedge_{i \le k} a_i \sqsubseteq_E [\![M_i]\!]) \end{array} \right).$$

Suppose that $a \sqsubseteq_E [\![M]\!]$. Since this implies $a \in E$, it follows that $a = \bot$ or a has the form $C(a_1, \ldots, a_k)$ for $a_i \in E$. If $a = \bot$ we are done. If $a = C(a_1, \ldots, a_k)$, then $[\![M]\!]$ also has the form $C(a'_1, \ldots, a'_k)$ for some $a'_i \sqsupseteq_E a_i$. Therefore, we can apply Lemma 33 and obtain $M \Downarrow C(M_1, \ldots, M_k)$ for some $M_1, \ldots, M_k$. By Lemma 28 (a), we have $[\![M]\!] = [\![C(M_1, \ldots, M_k)]\!] = C([\![M_1]\!], \ldots, [\![M_k]\!])$. Therefore, $a_i \sqsubseteq_E a'_i = [\![M_i]\!]$. □

6.4. Computation of infinite data

Theorem 5 (c) and (d) characterize the denotational semantics of a program M in terms of the relations $M \overset{\nu}{\Longrightarrow} a$ and $M \overset{\nu\bot}{\Longrightarrow} a$ which have a more proof-theoretic rather than operational character since they are defined by (possibly infinite) derivations. In this section we define a notion of possibly infinite step-by-step computation that continues under data constructor and prove a second adequacy theorem (Theorem 6) which provides a truly operational characterization of the denotational semantics of a program.

As one can see from Theorem 5 (d), the reduction relation $M \overset{\nu\bot}{\Longrightarrow} a$ is not functional and a program M is related to a set of data whose upper bound is the denotational semantics of M. To obtain a more precise operational notion, we use the following inductively defined smallstep leftmost-outermost reduction relation $\rightsquigarrow$ on closed programs that corresponds to bigstep reduction.

(i) $\mathbf{case}\, C(\vec{M})\, \mathbf{of}\, \{\ldots; C(\vec{y}) \to N; \ldots\} \rightsquigarrow N[\vec{M}/\vec{y}]$
(ii) $(\lambda x.\, M)\; N \rightsquigarrow M[N/x]$
(iii) $\mathbf{rec}\, M \rightsquigarrow M\, (\mathbf{rec}\, M)$
(iv) $$\frac{M \rightsquigarrow M'}{\mathbf{case}\, M\, \mathbf{of}\, \{\vec{C}\vec{l}\} \rightsquigarrow \mathbf{case}\, M'\, \mathbf{of}\, \{\vec{C}\vec{l}\}}$$
(v) $$\frac{M \rightsquigarrow M'}{M\, N \rightsquigarrow M'\, N}$$

Since we are only concerned with reducing closed terms the substitutions in (i) and (ii) do not need α-conversions.

Lemma 34. *If $M \Downarrow V$, then $M \rightsquigarrow^* V$.*

Proof. The proof is by induction on the definition of $M \Downarrow V$.

If $M = V$, then the assertion is trivial.

If $M = \mathbf{case}\, M' \,\mathbf{of}\, \{\ldots; C(\vec{y}) \to N; \ldots\}$, then $M' \Downarrow C(\vec{M})$ and $N[\vec{M}/\vec{y}]) \Downarrow V$. By the induction hypothesis, $M' \rightsquigarrow^* C(\vec{M})$ and $N[\vec{M}/\vec{y}] \rightsquigarrow^* V$. We have

$$\begin{aligned} M &= \mathbf{case}\, M' \,\mathbf{of}\, \{\ldots; C(\vec{y}) \to N; \ldots\} \\ &\rightsquigarrow^* \mathbf{case}\, C(\vec{M}) \,\mathbf{of}\, \{\ldots; C(\vec{y}) \to N; \ldots\} \\ &\rightsquigarrow N[\vec{M}/\vec{y}] \rightsquigarrow^* V\,. \end{aligned}$$

If $M = M_1\, N$, then $M_1 \Downarrow \lambda x.\, M'$ and $M'[N/x] \Downarrow V$. By the induction hypothesis, $M_1 \rightsquigarrow^* \lambda x.\, M'$ and $M'[N/x] \rightsquigarrow^* V$. Therefore, $M = M_1\, N \rightsquigarrow^* (\lambda x.\, M')\, N \rightsquigarrow M'[N/x] \rightsquigarrow^* V$.

If $M = \mathbf{rec}\, M'$, then $M'\, (\mathbf{rec}\, M') \Downarrow V$. We have

$$M = \mathbf{rec}\, M' \rightsquigarrow M'\, (\mathbf{rec}\, M') \rightsquigarrow^* V,$$

by the induction hypothesis. □

In order to approximate the denotational semantics operationally, we need to continue computation under constructors. Since a constructor may have more than one argument and some computations of arguments may diverge, we need to compute all the arguments in parallel. For this purpose, we extend the smallstep reduction $\rightsquigarrow$ to a relation $\overset{\mathrm{p}}{\rightsquigarrow}$ by the following inductive rules:

$$\frac{M \rightsquigarrow M'}{M \overset{\mathrm{p}}{\rightsquigarrow} M'} \qquad \frac{M_i \overset{\mathrm{p}}{\rightsquigarrow} M_i' \ (i = 1, \ldots, k)}{C(M_1, \ldots, M_k) \overset{\mathrm{p}}{\rightsquigarrow} C(M_1', \ldots, M_k')}$$

$$M \overset{\mathrm{p}}{\rightsquigarrow} M \qquad \text{otherwise.}$$

Clearly there is exactly one applicable rule for each closed program M. We denote by $M^{(n)}$ the unique program M' such that $M(\overset{\mathrm{p}}{\rightsquigarrow})^n M'$.

For a closed program M, we define $M_\perp \in E$ as follows.

$$\begin{aligned} C(M_1, \ldots, M_k)_\perp &= C(M_{1\perp}, \ldots, M_{k\perp}) \\ M_\perp &= \perp \quad \text{if } M \text{ is not a constructor term} \end{aligned}$$

Lemma 35 *(Accumulation). If $M \overset{\mathrm{p}}{\rightsquigarrow} M'$, then $M_\perp \sqsubseteq_E M'_\perp$. Therefore, $M^{(n)}{}_\perp \sqsubseteq_E M^{(m)}{}_\perp$ for $n \le m$.*

Proof. Immediate by the definition of $\overset{\mathrm{p}}{\rightsquigarrow}$. □

For a closed program M, $M^{(n)}{}_\perp$ can be viewed as the finite approximation of the value of M obtained after n consecutive parallel computation steps. The following lemma shows that this computation is complete, that is, *every* finite approximation is obtained eventually.

Lemma 36 *(Adequacy for finite values). If $M \overset{\mu\perp}{\Longrightarrow} a$, then $\exists n\, a \sqsubseteq_E M^{(n)}{}_\perp$.*

Proof. Let $P(M, a) \overset{\mathrm{Def}}{=} \exists n\, a \sqsubseteq_E M^{(n)}{}_\perp$. We prove by induction that $M \overset{\mu\perp}{\Longrightarrow} a$ implies $P(M, a)$. That is, we show

$$\Phi_{\perp}^{\mathrm{op}}(P)(M,a) \to P(M,a).$$

If $a = \perp$, then we have $P(M,a)$. If we have

$$(M \Downarrow C(M_1,\dots,M_k)) \wedge a = C(a_1,\dots,a_k) \wedge \bigwedge_{i\le k} (\exists n_i\ a_i \sqsubseteq_E {M_i}^{(n_i)}{}_{\perp})$$

for a constructor C, then, for n the maximum of n_i $(i \le k)$, $a = C(a_1,\dots,a_k) \sqsubseteq_E C({M_1}^{(n)}{}_{\perp},\dots,{M_k}^{(n)}{}_{\perp}) = C(M_1,\dots,M_k)^{(n)}{}_{\perp}$ by Lemma 35. On the other hand, by Lemma 34, we have $M^{(m)} = C(M_1,\dots,M_k)$ for some m. Therefore, $a \sqsubseteq_E M^{(m+n)}{}_{\perp}$. □

Since $M^{(n)}{}_{\perp}$ is an increasing sequence by Lemma 35, we can define

$$M^{(\infty)} = \bigsqcup_n M^{(n)}{}_{\perp}.$$

We say that the program M *infinitely computes* the data $M^{(\infty)}$.

For $d \in D$ we define the data-part $d_E \in E$ as follows.

$$\begin{aligned} \perp_E &= \perp \\ C(d_1,\dots,d_k)_E &= C((d_1)_E,\dots,(d_k)_E) \\ \mathbf{Fun}(f)_E &= \perp \end{aligned}$$

Clearly, the function $d \mapsto d_E$ is a projection of D onto E.

Theorem 6 *(Computational Adequacy II). $M^{(\infty)} = [\![M]\!]_E$ for every closed program M.*

Proof. It is easy to show the following.

(a) If $M \overset{\mathrm{p}}{\rightsquigarrow} M'$ then $[\![M]\!]_E = [\![M']\!]_E$.
(b) $M_{\perp} \sqsubseteq_E [\![M]\!]_E$.

Therefore, ${M^{(n)}}_{\perp} \sqsubseteq_E [\![M]\!]_E$. Since this holds for every n, we have $M^{(\infty)} \sqsubseteq_E [\![M]\!]_E$.

By Theorem 5 (d), $M \overset{\nu\perp}{\Longrightarrow} [\![M]\!]_E$ because $[\![M]\!]_E \sqsubseteq_E [\![M]\!]$. Therefore, by Lemma 26 (d), $\forall d\ (\mathrm{appr}(d,[\![M]\!]_E) \to M \overset{\mu\perp}{\Longrightarrow} d)$, and consequently, by Lemma 36, $\forall d\ (\mathrm{appr}(d,[\![M]\!]_E) \to \exists n\ d \sqsubseteq_E {M^{(n)}}_{\perp})$. Since $d \sqsubseteq_E {M^{(n)}}_{\perp} \to \mathrm{appr}(d, M^{(\infty)})$, we have $\forall d\ (\mathrm{appr}(d,[\![M]\!]_E) \to \mathrm{appr}(d, M^{(\infty)}))$. Therefore, $[\![M]\!]_E \sqsubseteq_E M^{(\infty)}$ by Lemma 24 (b). □

Note that if $[\![M]\!] \in E$, then we have $[\![M]\!]_E = [\![M]\!]$. Therefore, the second Adequacy Theorem says $M^{(\infty)} = [\![M]\!]$ in this case.

6.5. Data extraction

Using types we are able to identify criteria under which an extracted program denotes an observable data, i.e. an element of E.

Lemma 37. *If ρ is a type that contains no function type and ζ is a type environment such that $\zeta(\alpha) \subseteq E$ for all type variables α in the domain of ζ, then $D_{\rho}^{\zeta} \subseteq E$.*

Proof. Structural induction on ρ. The only non-obvious case is $\mathbf{fix}\,\alpha\,.\,\rho$. By the definition of $D^{\zeta}_{\mathbf{fix}\,\alpha\,.\,\rho}$, and since E is a subdomain of D, it suffices to show $D_{\rho}^{\zeta[\alpha\mapsto E]} \subseteq E$. But this holds by the induction hypothesis. □

We call an IFP-formula a *data formula* if it contains no free predicate variable and no strictly positive subformula of the form $A \to B$ where A and B are non-Harrop.

Theorem 7 *(Data Extraction). From a proof in* IFP *of a data formula A from Harrop assumptions Γ one can extract a closed program M realizing A, provably in* RIFP *from* $\mathbf{H}(\Gamma)$. *Moreover, M is a data that can hence be infinitely computed, that is, $M^{(\infty)} = [\![M]\!]$.*

Proof. By the Soundness Theorem (Theorem 2) we can extract a closed program $M : \tau(A)$ such that RIFP proves $\mathbf{H}(\Gamma) \vdash M\,\mathbf{r}\,A$. Clearly, since A is a data formula, $\tau(A)$ contains no function type. Therefore, by Lemma 37, M denotes a data. By the second Adequacy Theorem (Theorem 6), $M^{(\infty)} = [\![M]\!]$. □

Example 3. In Theorem 4, we proved $\mathbf{S} \subseteq \mathbf{G}$ and obtained a program stog as its realizer. On the other hand, one can prove $\mathbf{S}(1)$ by showing $\{1\} \subseteq \mathbf{S}$ by coinduction. From the proof, we can extract the realizer $a \overset{\mathrm{rec}}{=} \mathbf{Pair}(1, a)$ (i.e., $a = 1:1:\ldots$) of $\mathbf{S}(1)$. From $\mathbf{S} \subseteq \mathbf{G}$ and $\mathbf{S}(1)$, we can trivially prove $\mathbf{G}(1)$ and from these proofs we can extract a realizer $\mathsf{M}_1 = \mathsf{stog}\,(1:1:\ldots)$ of $\mathbf{G}(1)$. With the small-step reduction rule, one can compute

$$\mathsf{M}_1 \overset{\mathrm{p}}{\rightsquigarrow}{}^{*}\ \mathbf{R}{:}N_1 \overset{\mathrm{p}}{\rightsquigarrow}{}^{*}\ \mathbf{R}{:}\mathbf{L}{:}N_2 \overset{\mathrm{p}}{\rightsquigarrow}{}^{*}\ \mathbf{R}{:}\mathbf{L}{:}\mathbf{L}{:}N_3 \overset{\mathrm{p}}{\rightsquigarrow}{}^{*}\ \ldots$$

for some $N_i (i \geq 1)$. Taking $(\_)_{\perp}$ of these terms, we have an increasing sequence

$$\perp,\ \ \mathbf{R}{:}\perp,\ \ \mathbf{R}{:}\mathbf{L}{:}\perp,\ \ \mathbf{R}{:}\mathbf{L}{:}\mathbf{L}{:}\perp,\ \ \ldots$$

Taking the limit of these terms, one can see that M_1 infinitely computes the data ${\mathsf{M}_1}^{(\infty)} = \mathbf{R}:\mathbf{L}:\mathbf{L}:\ldots$, which is a realizer of $\mathbf{G}(1)$ by Theorem 7.

While for $\mathbf{S}(1)$ there was only one canonical proof and one realizer, we now look at $\mathbf{S}(1/2)$ which has more than one canonical proof and realizer and will give rise to three Gray codes, one with an undefined digit. By the coclosure axiom, $\mathbf{S}(1/2)$ unfolds to $\exists d \in \mathbf{SD}\,(1/2 \in \mathbb{I}_d \wedge \mathbf{S}(2 \cdot 1/2 - d))$. Therefore, we can choose $d = 0$ and use the above proof of $\mathbf{S}(1)$. This yields a realizer $0:1:1:\ldots$ of $\mathbf{S}(1/2)$, and $\mathsf{M}_{1/2} = \mathsf{stog}\,(0:1:1:\ldots)$ is a realizer of $\mathbf{G}(1/2)$. One can see that

$$\mathsf{M}_{1/2} \overset{\mathrm{p}}{\rightsquigarrow}{}^{*}\ N_1{:}\mathbf{R}{:}N_2 \overset{\mathrm{p}}{\rightsquigarrow}{}^{*}\ \mathbf{R}{:}\mathbf{R}{:}N_3 \overset{\mathrm{p}}{\rightsquigarrow}{}^{*}\ \mathbf{R}{:}\mathbf{R}{:}\mathbf{R}{:}N_4 \overset{\mathrm{p}}{\rightsquigarrow}{}^{*}$$

for some $N_i (i \geq 1)$. Therefore, the result of finite-time computation proceeds

$$\perp,\ \ \perp{:}\mathbf{R}{:}\perp,\ \ \mathbf{R}{:}\mathbf{R}{:}\perp,\ \ \mathbf{R}{:}\mathbf{R}{:}\mathbf{R}{:}\perp,\ \ \mathbf{R}{:}\mathbf{R}{:}\mathbf{R}{:}\mathbf{L}{:}\perp,\ \ \ldots$$

and in the limit, we have ${\mathsf{M}_{1/2}}^{(\infty)} = \mathbf{R}{:}\mathbf{R}{:}\mathbf{R}{:}\mathbf{L},\mathbf{L}{:}\ldots$.

Since $1:0:0:\ldots$ is another realizer of $\mathbf{S}(1/2)$, $\mathsf{M}'_{1/2} = \mathsf{stog}(1:0:0:\ldots)$ is also a realizer of $\mathbf{G}(1/2)$. One can see that

$$\mathsf{M}'_{1/2} \overset{\mathrm{p}}{\rightsquigarrow}{}^{*}\ \mathbf{R}{:}N_1 \overset{\mathrm{p}}{\rightsquigarrow}{}^{*}\ \mathbf{R}{:}N_2{:}\mathbf{R}{:}N_3 \overset{\mathrm{p}}{\rightsquigarrow}{}^{*}\ \mathbf{R}{:}N_4{:}\mathbf{R}{:}\mathbf{L}{:}N_5 \overset{\mathrm{p}}{\rightsquigarrow}{}^{*}\ \ldots$$

for some $N_i (i \geq 1)$. Therefore, one can observe the finite approximations

$$\perp,\ \ \mathbf{R}{:}\perp,\ \ \mathbf{R}{:}\perp{:}\mathbf{R}{:}\perp,\ \ \mathbf{R}{:}\perp{:}\mathbf{R}{:}\mathbf{L}{:}\perp,\ \ \ldots$$

hence $\mathsf{M}'_{1/2}$ computes the partial infinite data ${\mathsf{M}'_{1/2}}^{(\infty)} = \mathbf{R}{:}\perp{:}\mathbf{R}{:}\mathbf{L}{:}\mathbf{L}{:}\ldots$.

7. Conclusion

We presented IFP, a formal system supporting program extraction from proofs in abstract mathematics. IFP is plain many-sorted first-order logic extended with two extra constructs for strictly positive inductive and coinductive definitions that are dual to each other. Sorts in IFP represent abstract structures specified by (classically true) disjunction free closed axioms. Hence full classical logic is available. Computational content is extracted through a realizability interpretation that treats quantifiers uniformly in order to permit the interpretation of sorts as abstract spaces. The target language of the interpretation is a functional programming language in which extracted programs are typable and therefore easily translatable into Haskell and executed there. The exact fit of the denotational and operational semantics of the target language is proven by two computational adequacy theorems. The first (Theorem 5) states that all compact approximations of the denotational value of a program can be computed, the second (Theorem 6) states that the full (possibly infinite) denotation value can be computed through successive computation steps. It should be stressed that axioms used in a proof do not show up as non-executable constants in extracted programs and therefore do not spoil the computation of programs into canonical form. Besides the natural numbers as a primary example of a strictly positive inductive definition we studied wellfounded induction and useful variations thereof such as Archimedean induction.

In an extended case study we formalized in IFP the real numbers as an Archimedean real closed field and introduced various exact real number representations (Cauchy and signed digit representation as well as infinite Gray code) as the realizability interpretations of simple coinductive predicates (**C**, **S**, and **G**). From a proof that **S** is a subset of **G** we extracted a program converting the signed digit representation into infinite Gray code. There is an experimental Haskell implementation of IFP and its program extraction called Prawf [17] where this is carried out.

This case study highlights some crucial features of IFP:

- The real numbers are given axiomatically as an abstract structure;
- signed digit representation and infinite Gray code are obtained as realizers of coinductive predicates **S** and **G**;
- Archimedean induction is used to prove that the sign of non-zero reals in **S** can be decided (first part of the proof of Theorem 4);
- the definition of **G** permits partial realizers (which are inevitable for infinite Gray code);
- the second adequacy Theorem is applied to compute full infinite Gray code in the limit.

This case study not only puts to test the practical usability of IFP but also leads to the study of possible extensions of it. Having extracted a program realizing the inclusion $\mathbf{S} \subseteq \mathbf{G}$ it is natural to ask about the reverse inclusion. In [72] a parallel and nondeterministic program converting infinite Gray code into signed digit representation is given which is necessarily parallel and nondeterministic [73]. Since the programming language of RIFP doesn't have these features such conversion cannot be extracted. We leave it for further work to develop a suitable extension of our system improving and extending previous work in this direction [16,10]. A further interesting line of study will be the extraction of algorithms that operate on compact sets of real numbers as studied in [14,67].

References

[1] A. Abel, B. Pientka, A. Setzer, Copatterns: programming infinite structures by observations, in: 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'13), 2013, pp. 27–38.
[2] S. Abramsky, A. Jung, Domain theory, in: S. Abramsky, D.M. Gabbay, T.S.E. Maibaum (Eds.), Handbook of Logic in Computer Science, vol. 3, Clarendon Press, 1994, pp. 1–168.
[3] P. Aczel, An introduction to inductive definitions, in: J. Barwise (Ed.), Handbook of Mathematical Logic, vol. 2, North-Holland, Amsterdam, 1977, pp. 739–782.

[4] Agda, Agda official website, http://wiki.portal.chalmers.se/agda/.
[5] J. Avigad, H. Towsner, Functional interpretation and inductive definitions, J. Symb. Log. 74 (4) (2009) 1100–1120.
[6] A. Bauer, J. Blanck, Canonical effective subalgebras of classical algebras as constructive metric completions, in: A. Bauer, P. Hertling, K.I. Ko (Eds.), 6th Int'l Conf. on Computability and Complexity in Analysis, Dagstuhl, Germany, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany, 2009.
[7] U. Berger, From coinductive proofs to exact real arithmetic, in: E. Grädel, R. Kahle (Eds.), Computer Science Logic, in: Lecture Notes in Computer Science, vol. 5771, Springer, 2009, pp. 132–146.
[8] U. Berger, Realisability for induction and coinduction with applications to constructive analysis, J. Univers. Comput. Sci. 16 (18) (2010) 2535–2555.
[9] U. Berger, From coinductive proofs to exact real arithmetic: theory and applications, Log. Methods Comput. Sci. 7 (1) (2011) 1–24.
[10] U. Berger, Extracting non-deterministic concurrent programs, in: J.-M. Talbot, L. Regnier (Eds.), 25th EACSL Annual Conference on Computer Science Logic (CSL 2016), Dagstuhl, Germany, in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 62, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 26:1–26:21.
[11] U. Berger, O. Petrovska, Optimized program extraction for induction and coinduction, in: CiE 2018: Sailing Routes in the World of Computation, in: LNCS, vol. 10936, Springer Verlag, Berlin, Heidelberg, New York, 2018, pp. 70–80.
[12] U. Berger, M. Seisenberger, Proofs, programs, processes, Theory Comput. Syst. 51 (3) (2012) 213–329.
[13] U. Berger, A. Setzer, Undecidability of equality for codata types, in: Coalgebraic Methods in Computer Science, in: Lecture Notes in Computer Science, vol. 11202, Springer, 2018, pp. 34–55.
[14] U. Berger, D. Spreen, A coinductive approach to computing with compact sets, J. Log. Anal. 8 (2016).
[15] U. Berger, K. Miyamoto, H. Schwichtenberg, M. Seisenberger, Minlog - a tool for program extraction for supporting algebra and coalgebra, in: CALCO-Tools, in: Lecture Notes in Computer Science, vol. 6859, Springer, 2011, pp. 393–399.
[16] U. Berger, K. Miyamoto, H. Schwichtenberg, H. Tsuiki, Logic for Gray-code computation, in: Concepts of Proof in Mathematics, Philosophy, and Computer Science, in: Ontos Mathematical Logic, vol. 6, de Gruyter, 2016.
[17] U. Berger, O. Petrovska, H. Tsuiki Prawf, An interactive proof system for program extraction, in: CiE 2020: Beyond the Horizon of Computability, in: LNCS, vol. 12098, Springer Verlag, Berlin, Heidelberg, New York, 2020, pp. 137–148.
[18] S. Berghofer, Program extraction in simply-typed higher order logic, in: Types for Proofs and Programs (TYPES'02), in: Lecture Notes in Computer Science, vol. 2646, Springer, 2003, pp. 21–38.
[19] E. Bishop, D. Bridges, Constructive Analysis, Grundlehren der mathematischen Wissenschaften, vol. 279, Springer, 1985.
[20] J. Bradfield, C. Stirling, Modal mu-calculi, in: P. Blackburn, J. van Benthem, F. Wolter (Eds.), Handbook of Modal Logic, in: Studies in Logic and Practical Reasoning, vol. 3, Elsevier, 2007, pp. 721–756.
[21] W. Buchholz, F. Feferman, W. Pohlers, W. Sieg, Iterated Inductive Definitions and Subsystems of Analysis: Recent Proof–Theoretical Studies, Lecture Notes in Mathematics, vol. 897, Springer, Berlin, 1981.
[22] M.D. Campos, P.B. Levy, A syntactic view of computational adequacy, in: FOSSACS 2018, in: LNCS, vol. 10803, Springer Verlag, Berlin, Heidelberg, New York, 2018, pp. 71–87.
[23] R.L. Constable, Implementing Mathematics with the Nuprl Proof Development System, Prentice–Hall, New Jersey, 1986.
[24] Coq, The Coq Proof Assistant, https://coq.inria.fr.
[25] T. Coquand, Infinite objects in type theory, in: H. Barendregt, T. Nipkow (Eds.), Types for Proofs and Programs, in: Lecture Notes in Computer Science, vol. 806, 1994, pp. 62–78.
[26] T. Coquand, A. Spiwack, A proof of strong normalisation using domain theory, in: Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science (LICS'06), IEEE Computer Society Press, 2006, pp. 307–316.
[27] R. Crole, A. Pitts, New foundations for fixpoint computations: fix-hyperdoctrines and the fix-logic, Inf. Comput. 98 (1992) 171–210.
[28] P. Di Gianantonio, An abstract data type for real numbers, Theor. Comput. Sci. 221 (1–2) (1999) 295–326.
[29] P. Dybjer, A. Setzer, Induction-recursion and initial algebras, Ann. Pure Appl. Log. 124 (2003) 1–47.
[30] A. Edalat, J.P. Potts, M. Escardo, An axiomatisation of computationally adequate domain theoretic models of fpc, in: Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press, 1994.
[31] M.H. Escardo, PCF extended with real numbers, Theor. Comput. Sci. 162 (1996) 79–115.
[32] S. Feferman, Constructive theories of functions and classes, in: M. Boffa, D. van Dalen, K. McAloon (Eds.), Logic Colloquium '78, North–Holland, Amsterdam, 1979, pp. 159–224.
[33] P. Gerhardy, U. Kohlenbach, General logical metatheorems for functional analysis, Trans. Am. Math. Soc. 360 (2008) 2615–2660.
[34] H. Geuvers, Inductive and coinductive types with iteration and recursion, in: B. Nordström, K. Pettersson, G. Plotkin (Eds.), Informal Proceedings Workshop on Types for Proofs and Programs, Båstad, Sweden, 8–12 June 1992, Dept. of Computing Science, Chalmers Univ. of Technology and Göteborg Univ., 1992, pp. 193–217.
[35] G. Gierz, K.H. Hofmann, K. Keimel, J.D. Lawson, M. Mislove, D.S. Scott, Continuous Lattices and Domains, Encyclopedia of Mathematics and Its Applications, vol. 93, Cambridge University Press, 2003.
[36] T. Glaß, M. Rathjen, A. Schlüter, On the proof-theoretic strength of monotone induction in explicit mathematics, Ann. Pure Appl. Log. 85 (1) (1997) 1–46.
[37] P. Hancock, A. Setzer, Guarded induction and weakly final coalgebras in dependent type theory, in: L. Crosilla, P. Schuster (Eds.), From Sets and Types to Topology and Analysis. Towards Practicable Foundations for Constructive Mathematics, Clarendon Press, Oxford, 2005, pp. 115–134.
[38] S. Hayashi, H. Nakano, PX: A Computational Logic, MIT Press, Cambridge, MA, USA, 1988.
[39] B. Jacobs, J. Rutten, A tutorial on (co)algebras and (co)induction, Bull. Eur. Assoc. Theor. Comput. Sci. 62 (1997) 222–259.
[40] S.C. Kleene, On the interpretation of intuitionistic number theory, J. Symb. Log. 10 (1945) 109–124.
[41] U. Kohlenbach, Some logical metatheorems with applications in functional analysis, Trans. Am. Math. Soc. 357 (2005) 89–129.

[42] U. Kohlenbach, Proof Interpretations and Their Use in Mathematics, Springer Monographs in Mathematics, Springer, 2008.
[43] D. Kozen, Results on the propositional μ-calculus, Theor. Comput. Sci. 27 (1983) 333–354.
[44] G. Kreisel, Interpretation of analysis by means of constructive functionals of finite types, in: Constructivity in Mathematics, 1959, pp. 101–128.
[45] J-L. Krivine, Typed lambda-calculus in classical Zermelo-Fraenkel set theory, Ann. Math. Log. 40 (2001) 189–205.
[46] J-L. Krivine, Dependent choice, 'quote' and the clock, Theor. Comput. Sci. 308 (2003) 259–276.
[47] C. Kupke, A. Kurz, D. Pattinson, Algebraic semantics for coalgebraic logics, Electron. Notes Theor. Comput. Sci. 106 (2004) 35–47.
[48] J. Laird, G. Manzonetto, G. McCusker, M. Pagani, Weighted relational models of typed lambda-calculi, in: Proceedings of the 28th Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press, 2013, pp. 301–310.
[49] D. MacQueen, G. Plotkin, R. Sethi, An ideal model for recursive polymorphic types, Inf. Control 71 (1986) 95–130.
[50] P. Martin-Löf, Intuitionistic Type Theory, Bibliopolis, 1984.
[51] R. Matthes, Monotone inductive and coinductive constructors of rank 2, in: L. Fribourg (Ed.), Computer Science Logic (Proceedings of the Fifteenth CSL Conference), in: Lecture Notes in Computer Science, vol. number 2142, Springer, 2001, pp. 600–615.
[52] N.P. Mendler, Inductive types and type constraints in the second-order lambda calculus, Ann. Pure Appl. Log. 51 (1991) 159–172.
[53] F. Miranda-Perea, Realizability for monotone clausular (co)inductive definitions, Electron. Notes Theor. Comput. Sci. 123 (2005) 179–193.
[54] M. Möllerfeld, Generalized inductive definitions, PhD thesis, Westfälische Wilhelms-Universität Münster, 2003.
[55] F. Nordvall Forsberg, A. Setzer, Inductive-inductive definitions, in: D. Anuj, V. Helmut (Eds.), Computer Science Logic, in: Lecture Notes in Computer Science, vol. 6247, Springer, 2010, pp. 454–468.
[56] P. Oliva, T. Streicher, On Krivine's realizability interpretation of classical second-order arithmetic, Fundam. Inform. 84 (2) (2008) 207–220.
[57] G. Plotkin, J. Power, Adequacy for algebraic effects, in: M. Miculan, F. Honsell (Eds.), Foundations of Software Science and Computation Structures. FoSSaCS 2001, in: LNCS, vol. 2030, 2001.
[58] G.D. Plotkin, LCF considered as a programming language, Theor. Comput. Sci. 5 (1977) 223–255.
[59] G.D. Plotkin, Set-theoretical and other elementary models of the lambda-calculus, Theor. Comput. Sci. 121 (1993) 351–409.
[60] D. Ramyaa, R. Leivant, Ramified corecurrence and logspace, Electron. Notes Theor. Comput. Sci. 276 (2011) 247–261.
[61] M. Rathjen, Generalized inductive definitions in constructive set theory, in: From Sets and Types to Topology and Analysis, in: Oxford Logic Guides, vol. 48, Oxford University Press, Oxford, 2005, pp. 23–40.
[62] H. Rogers, Theory of Recursive Functions and Effective Computability, Mc Graw Hill, 1967.
[63] H. Schwichtenberg, Minlog, in: F. Wiedijk (Ed.), The Seventeen Provers of the World, in: Lecture Notes in Artificial Intell., vol. 3600, 2006, pp. 151–157.
[64] H. Schwichtenberg, S.S. Wainer, Proofs and Computations, Cambridge University Press, 2012.
[65] D.S. Scott, Domains for denotational semantics, in: Automata, Languages and Programming, 9th Colloquium, Aarhus, Denmark, 1982, pp. 577–610.
[66] A. Simpson, Computational adequacy for recursive types in models of intuitionistic set theory, Ann. Pure Appl. Log. 130 (2004) 207–275.
[67] D. Spreen, Computing with continuous objects: a uniform co-inductive approach, arXiv:2004.05392, 2020.
[68] M. Tatsuta, Realizability of monotone coinductive definitions and its application to program synthesis, in: R. Parikh (Ed.), Mathematics of Program Construction, in: Lecture Notes in Mathematics, vol. 1422, Springer, 1998, pp. 338–364.
[69] A.S. Troelstra, Metamathematical Investigation of Intuitionistic Arithmetic and Analysis, Lecture Notes in Mathematics, vol. 344, Springer, 1973.
[70] A.S. Troelstra, H. Schwichtenberg, Basic Proof Theory, Cambridge University Press, 1996.
[71] A.S. Troelstra, D. van Dalen, Constructivism in Mathematics. An Introduction, vol. 121, 123, North–Holland, Amsterdam, 1988.
[72] H. Tsuiki, Real number computation through Gray code embedding, Theor. Comput. Sci. 284 (2) (2002) 467–485.
[73] H. Tsuiki, Real number computation with committed choice logic programming languages, J. Log. Algebraic Program. 64 (1) (2005) 61–84.
[74] S. Tupailo, On the intuitionistic strength of monotone inductive definitions, J. Symb. Log. 69 (3) (2004) 790–798.
[75] V. Veldman, Brouwer's real thesis on bars, Philos. Sci. 6 (2001) 21–42.
[76] K. Weihrauch, Computable Analysis, Springer, 2000.
[77] G. Winskel, The Formal Semantics of Programming Languages, Foundations of Computing Series, The MIT Press, Cambridge, Massachusetts, 1993.
[78] J. Zucker, Iterated inductive definitions, trees and ordinals, in: A.S. Troelstra (Ed.), Metamathematical Investigation of Intuitionistic Arithmetic and Analysis, in: Lecture Notes in Mathematics, vol. 344, Springer Verlag, Berlin, Heidelberg, New York, 1973, pp. 392–461, chapter VI.