Extracting total Amb programs from proofs

Ulrich Berger[1]✉ and Hideki Tsuiki[2]✉

[1] Swansea University, Swansea, UK
u.berger@swansea.ac.uk
[2] Kyoto University, Kyoto, Japan
tsuiki@i.h.kyoto-u.ac.jp

Abstract. We present a logical system CFP (Concurrent Fixed Point Logic) that supports the extraction of nondeterministic and concurrent programs that are provably total and correct. CFP is an intuitionistic first-order logic with inductive and coinductive definitions extended by two propositional operators, $B|_A$ (restriction, a strengthening of implication) and $\downdownarrows(B)$ (total concurrency). The source of the extraction are formal CFP proofs, the target is a lambda calculus with constructors and recursion extended by a constructor Amb (for McCarthy's amb) which is interpreted operationally as globally angelic choice and is used to implement nondeterminism and concurrency. The correctness of extracted programs is proven via an intermediate domain-theoretic denotational semantics. We demonstrate the usefulness of our system by extracting a nondeterministic program that translates infinite Gray code into the signed digit representation. A noteworthy feature of our system is that the proof rules for restriction and concurrency involve variants of the classical law of excluded middle that would not be interpretable computationally without Amb.

1 Introduction

Nondeterministic bottom-avoiding choice is an important and useful idea. With the wide-spread use of hardware that supports parallel computation, it has the possibility to speed up practical computation and, at the same time, it is related to computation over mathematical structures like real numbers [20,42]. On the other hand, it is not easy to apply theoretical tools like denotational semantics to nondeterministic bottom-avoiding choice [24,29] and guaranteeing correctness and totality of such programs through logical systems is a difficult task.

To explain the subtleness of the problem, let us start with an example. Suppose that M and N are partial programs that, under the conditions A and $\neg A$, respectively, are guaranteed to terminate and produce values satisfying specification B. Then, by executing M and N in parallel and taking the result obtained first, we should always obtain a result satisfying B. This kind of bottom-avoiding nondeterministic program is known as *McCarthy's amb (ambiguous) operator* [32], and we denote such a program by $\mathbf{Amb}(M, N)$. **Amb** is called the angelic choice operator and is usually studied as one of the three nondeterministic choice

I. Sergey (Ed.): ESOP 2022, LNCS 13240, pp. 85–113, 2022.
https://doi.org/10.1007/978-3-030-99336-8_4

operators (the other two are erratic choice and demonic choice). On the other hand, we are interested in this operator not only from a theoretical point of view but also from the way it behaves as a concurrent program running on a parallel execution mechanism.

If one tries to formalize this idea naively, one will face some obstacles. Let $M\,\mathbf{r}\,B$ ("M realizes B") denote the fact that a program M satisfies a specification B and let $\downdownarrows(B)$ be the specification that can be satisfied by a concurrent program of the form $\mathbf{Amb}(M,N)$ that always terminates and produces a value satisfying B. Then, the above inference could be written as

$$\frac{A \to (M\,\mathbf{r}\,B) \qquad \neg A \to (N\,\mathbf{r}\,B)}{\mathbf{Amb}(M,N)\,\mathbf{r}\,\downdownarrows(B)}$$

However, this inference is not sound for the following reason. Suppose that A does not hold, that is, $\neg A$ holds. Then, the execution of N will produce a value satisfying B. But the execution of M may terminate as well, and with a data that does not satisfy B since there is no condition on M if A does not hold. Therefore, if M terminates first in the execution of $\mathbf{Amb}(M,N)$, then we obtain a result that may not satisfy B.

To amend this problem, we add a new operator $B|_A$ (pronounced "B restricted to A") and consider the rule

$$\frac{M\,\mathbf{r}\,(B|_A) \qquad N\,\mathbf{r}\,(B|_{\neg A})}{\mathbf{Amb}(M,N)\,\mathbf{r}\,\downdownarrows(B)} \tag{1}$$

Intuitively, $M\,\mathbf{r}\,(B|_A)$ means two things: (1) M terminates if A holds, and (2) if M terminates, then the result satisfies B even for the case A does not hold. As we will see in Sect. 5.2, the above rule is derivable in classical logic and can therefore be used to prove total correctness of Amb programs.

In this paper, we go a step further and introduce a logical system CFP whose formulas can be interpreted as specifications of nondeterministic programs although they do not talk about programs explicitly. CFP is defined by adding the two logical operators $B|_A$ and $\downdownarrows(B)$ to the system IFP, a logic for program extraction [12] (see also [4,9,7]). A related approach has been developed in the proof system Minlog [38,6,39]. IFP supports the extraction of lazy functional programs from inductive/coinductive proofs in intuitionistic first-order logic. It has a prototype implementation in Haskell, called Prawf [8].

We show that from a CFP-proof of a formula, both a program and a proof that the program satisfies the specification can be extracted (Soundness theorem, Theorem 3). For example, in CFP we have the rule

$$\frac{B|_A \qquad B|_{\neg A}}{\downdownarrows(B)}\ \text{(Conc-lem)} \tag{2}$$

which is realized by the program $\lambda a.\lambda b.\mathbf{Amb}(a,b)$, and whose correctness is expressed by the rule (1). Programs extracted from CFP proofs can be executed in Haskell, implementing **Amb** with the concurrent Haskell package.

Compared with program verification, the extraction approach has the benefit that (a) the proofs programs are extracted from take place in a formal system that is of a very high level of abstraction and therefore is simpler and easier to use than a logic that formalizes concurrent programs (in particular, programs do not have to be written manually at all); (b) not only the complete extracted program is proven correct but also all its sub-programs come with their specifications and correctness proofs since these correspond to sub-proofs. This makes it easier to locally modify programs without the danger of compromising overall correctness.

As an application, we extract a nondeterministic program that converts infinite Gray code to signed digit representation, where infinite Gray code is a coding of real numbers by partial digit streams that are allowed to contain a $\bot$, that is, a digit whose computation does not terminate [18,42]. Partiality and multi-valuedness are common phenomena in computable analysis and exact real number computation [46,30]. This case study connects these two aspects through a nondeterministic and concurrent program whose correctness is guaranteed by a CFP-proof. The extracted Haskell programs are available in the repository [3].

Organization of the paper: In Sects. 2 and 3 we present the denotational and operational semantics of a functional language with **Amb** and prove that they match (Thms. 1 and 2). Sects. 4 and 5 describe the formal system CFP and its realizability interpretation which our program extraction method is based on (Thms. 3 and 5). In Sect. 6 we extract a concurrent program that converts representation of real numbers and study its behaviour in Sect. 7. Most proofs, unless very short, are omitted do to space limitation. Full proofs of the main results can be found in the extended version [11].

2 Denotational semantics of globally angelic choice

In [32], McCarthy defined the ambiguity operator **amb** as

$$\mathbf{amb}(x,y) = \begin{cases} x & (x \neq \bot) \\ y & (y \neq \bot) \\ \bot & (x = y = \bot) \end{cases}$$

where $\bot$ means 'undefined' and x and y are taken nondeterministically when both x and y are not $\bot$. This is called *locally* angelic nondeterministic choice since convergence is chosen over divergence for each local call for the computation of $\mathbf{amb}(x,y)$. It can be implemented by executing both of the arguments in parallel and taking the result obtained first. Despite being a simple construction, **amb** is known to have a lot of expressive power, and many constructions of nondeterministic and parallel computation such as erratic choice, countable choice (random assignment), and 'parallel or' can be encoded through it [28]. These multifarious aspects of the operator **amb** are reflected by the difficulty of its mathematical treatment in denotational semantics. For example, **amb** is not monotonic when interpreted over powerdomains with the Egli-Milner order [14].

On the other hand, one can consider an interpretation of **amb** as *globally* angelic choice, where an argument of **amb** is chosen so that the whole ambient

computation converges, if convergence is possible at all [17,40]. Since globally angelic choice is not defined compositionally, it is not easy to integrate it into a design of a programming language with clear denotational semantics. However, it can be easily implemented by running the whole computation for both of the arguments of **amb** in parallel and taking the result obtained first. Denotationally, globally angelic choice can be modelled by the Hoare powerdomain construction. However, this would not be suitable for analyzing total correctness because the ordering of the Hoare powerdomain does not discriminate X and $X \cup \{\bot\}$ [23,24]. Instead, we consider a two-staged approach (see Sect. 2.2).

The difference between the locally and the globally angelic interpretation of **amb** is highlighted by the fact that the former does not commute with function application. For example, if $f(0) = 0$ but $f(1)$ diverges, then $\mathbf{amb}(f(0), f(1))$ will always terminate with the value 0, whereas $f(\mathbf{amb}(0,1))$ may return 0 or diverge. On the other hand, the latter term will always return 0 if **amb** is implemented with a globally angelic semantics. As suggested in [17], we use this commutation property to realize the globally angelic semantics.

2.1 Programs and types

Our target language for program extraction is an untyped lambda calculus with recursion operator and constructors as in [12], but extended by an additional constructor **Amb** that corresponds to globally angelic version of McCarthy's **amb**. This could be easily generalized to an **Amb** operator of any arity ≥ 2.

$$
\begin{aligned}
\mathit{Programs} \ni M, N, L, P, Q, R ::= {} & a, b, \ldots, f, g \quad \text{(program variables)} \\
& \mid \lambda a.\, M \mid M\, N \mid M{\downarrow}N \mid \mathbf{rec}\, M \mid \bot \\
& \mid \mathbf{Nil} \mid \mathbf{Left}(M) \mid \mathbf{Right}(M) \mid \mathbf{Pair}(M, N) \mid \mathbf{Amb}(M, N) \\
& \mid \mathbf{case}\, M\, \mathbf{of}\, \{\mathbf{Left}(a) \to L; \mathbf{Right}(b) \to R\} \\
& \mid \mathbf{case}\, M\, \mathbf{of}\, \{\mathbf{Pair}(a, b) \to N\} \\
& \mid \mathbf{case}\, M\, \mathbf{of}\, \{\mathbf{Amb}(a, b) \to N\}
\end{aligned}
$$

Denotationally, **Amb** is just another pairing operator. Its interpretation as globally angelic choice will come to effect only through its operational semantics. Though essentially a call-by-name language, it also has strict application $M{\downarrow}N$, needed for realizing the rules for restriction and the concurrency operator.

We use $a, \ldots, g$ for program variables to distinguish them from the variables x, y, z of the logical system CFP (Sect. 4). **Nil**, **Left**, **Right**, **Pair**, **Amb** are called *constructors*. Constructors different from **Amb** are called *data constructors*. $\mathrm{C_d}$ denotes the set of data constructors. $\mathbf{Left}{\downarrow}M$ stands for $(\lambda a.\mathbf{Left}(a)){\downarrow}M$, etc., and we sometimes write **Left** and **Right** for $\mathbf{Left}(\mathbf{Nil})$ and $\mathbf{Right}(\mathbf{Nil})$. Natural numbers are encoded as $0 \stackrel{\text{Def}}{=} \mathbf{Left}$, $1 \stackrel{\text{Def}}{=} \mathbf{Right}(\mathbf{Left})$, and so on.

Although programs are untyped, programs extracted from proofs will be typable by the following system of simple recursive types:

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

Here, $\mathbf{A}(\rho)$ is the type of programs which, if they terminate (see Sect. 3), reduce to a form $\mathbf{Amb}(M, N)$ with $M, N : \rho$. The formation of $\mathbf{fix}\,\alpha\,.\,\rho$ has the side conditions that α occurs freely in ρ, ρ is strictly positive in α (that is, there is no free occurrence of α in ρ which is in the left part of a function type), and not of the form α or $\mathbf{A}(\alpha)$. These conditions ensure, among other things, that the type transformer $\alpha \mapsto \rho$ has a unique fixed point, which is taken as the semantics of $\mathbf{fix}\,\alpha\,.\,\rho$ (see below). We require in $\mathbf{A}(\rho)$ that ρ is neither a variable nor of the form $\mathbf{fix}\,\alpha_1\,.\,\ldots\,\mathbf{fix}\,\alpha_n\,.\,\mathbf{A}(\rho')$ $(n \geq 0)$. This enables the interpretation of **Amb** as a bottom-avoiding choice operator (see the explanation below Corollary 1). We call types that satisfy all these conditions *regular*. An example of a regular type is the type of lazy (partial) natural numbers, $\mathbf{nat} \stackrel{\text{Def}}{=} \mathbf{fix}\,\alpha\,.\,\mathbf{1} + \alpha$.

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

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

$$\frac{\Gamma \vdash M : \rho \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash \mathbf{Pair}(M, N) : \rho \times \sigma} \qquad \frac{\Gamma \vdash M : \rho \qquad \Gamma \vdash N : \rho}{\Gamma \vdash \mathbf{Amb}(M, N) : \mathbf{A}(\rho)}$$

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

$$\frac{\Gamma \vdash M : \rho \Rightarrow \sigma \qquad \Gamma \vdash N : \rho}{\Gamma \vdash M\, N : \sigma} \qquad \frac{\Gamma \vdash M : \rho \Rightarrow \sigma \qquad \Gamma \vdash N : \rho}{\Gamma \vdash M{\downarrow}N : \sigma}$$

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

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

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

Fig. 1. Typing rules

The typing rules are listed in Fig. 1. They are valid w.r.t. the denotational semantics given in Sect. 2.2 and extend the rules given in [12]. Recursive types are equirecursive [35] in that $M : \mathbf{fix}\,\alpha\,.\,\rho$ iff $M : \rho[\mathbf{fix}\,\alpha\,.\,\rho/\alpha]$.

As an example of a program consider

$$f \stackrel{\text{Def}}{=} \lambda a.\mathbf{case}\, a\, \mathbf{of}\, \{\mathbf{Left}(\_) \to \mathbf{Left}; \mathbf{Right}(\_) \to \bot\} \tag{3}$$

which implements the function f discussed earlier, i.e., $f\,0 = 0$ and $f\,1 = \bot$. f has type $\mathbf{nat} \Rightarrow \mathbf{nat}$. Since $\mathbf{Amb}(0, 1)$ has type $\mathbf{A}(\mathbf{nat})$, the application $f\,\mathbf{Amb}(0, 1)$ is not well-typed. Instead, we consider $\mathsf{mapamb}\ f\ \mathbf{Amb}(0, 1)$ where $\mathsf{mapamb} : (\rho \to \sigma) \to \mathbf{A}(\rho) \to \mathbf{A}(\sigma)$ is defined as

$$\mathsf{mapamb} \stackrel{\text{Def}}{=} \lambda f.\, \lambda c.\ \mathbf{case}\, c\, \mathbf{of}\ \{\mathbf{Amb}(a, b) \to \mathbf{Amb}(f{\downarrow}a, f{\downarrow}b)\}$$

This operator realizes the globally angelic semantics: mapamb f **Amb**$(0,1)$ is reduced to **Amb**$(f{\downarrow}0, f{\downarrow}1)$, and $f{\downarrow}0$ and $f{\downarrow}1$ (which are the same as $f\ 0$ and $f\ 1$ since 0 and 1 are defined) are computed concurrently and the whole expression is reduced to 0, using the operational semantics in Section 3. In Sect. 5, we will introduce a concurrent (or nondeterministic) version of Modus Ponens, (Concmp), which will automatically generate an application of mapamb.

2.2 Denotational semantics

The denotational semantics has two phases: *Phase I* interprets programs in a Scott domain D defined by the following recursive domain equation

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

where $+$ and $\times$ denote separated sum and cartesian product, and the operation $\cdot_\perp$ adds a least element $\perp$ ([21] is a recommended reference for domain theory and the solution of domain equations). A closed program M denotes an element $[\![M]\!] \in D$ as defined in Fig. 2. Note that **Amb** is interpreted (like **Pair**) as a simple pairing operator.

A type is interpreted as a subdomain, which is a subset of D that is downward closed and closed under suprema of bounded subsets. We use the following operations on subdomains:

$$\begin{aligned}
(X+Y)_\perp &\stackrel{\text{Def}}{=} \{\mathbf{Left}(a) \mid a \in X\} \cup \{\mathbf{Right}(b) \mid b \in Y\} \cup \{\perp\}\\
(X\times Y)_\perp &\stackrel{\text{Def}}{=} \{\mathbf{Pair}(a,b) \mid a \in X, b \in Y\} \cup \{\perp\}\\
(X \Rightarrow Y)_\perp &\stackrel{\text{Def}}{=} \{\mathbf{Fun}(f) \mid f : D \to D \text{ continuous}, \forall a \in X(f(a) \in Y)\} \cup \{\perp\}.
\end{aligned}$$

Through the semantics in Fig. 2, closed programs denote elements of D and closed types denote subdomains of D such that the typing rules (Fig. 1) are sound.

In *Phase II* we assign to every $a \in D$ a set $\mathrm{data}(a) \subseteq D$ that reveals the role of **Amb** as a choice operator. The relation '$d \in \mathrm{data}(a)$' is defined (coinductively) as the largest relation satisfying

$$\begin{aligned}
d \in \mathrm{data}(a) \quad \stackrel{\nu}{=} \quad & (a = \mathbf{Amb}(a', b') \wedge a' \neq \perp \wedge d \in \mathrm{data}(a')) \vee\\
& (a = \mathbf{Amb}(a', b') \wedge b' \neq \perp \wedge d \in \mathrm{data}(b')) \vee\\
& (a = \mathbf{Amb}(\perp, \perp) \wedge d = \perp) \vee\\
& \bigvee_{C \in \mathrm{C_d}} \left(a = C(\vec{a'}) \wedge d = C(\vec{d'}) \wedge \bigwedge_i d'_i \in \mathrm{data}(a'_i) \right) \vee\\
& (a = \mathbf{Fun}(f) \wedge d = a) \ \vee (a = d = \perp)\,.
\end{aligned}$$

Now, every closed program M denotes the set $\mathrm{data}([\![M]\!]) \subseteq D$ containing all possible globally angelic choices derived form its denotation in D. For example, $\mathrm{data}(\mathbf{Amb}(0,1)) = \{0,1\}$ and, for f as defined in (3), we have, as expected,

$$
\begin{aligned}
[\![a]\!]\eta &= \eta(a)\\
[\![\lambda a.\, M]\!]\eta &= \mathbf{Fun}(f) \quad \text{where } f(d) = [\![M]\!]\eta[a \mapsto d]\\
[\![M\ N]\!]\eta &= f([\![N]\!]\eta) \quad \text{if } [\![M]\!]\eta = \mathbf{Fun}(f)\\
[\![M{\downarrow}N]\!]\eta &= f([\![N]\!]\eta) \quad \text{if } [\![M]\!]\eta = \mathbf{Fun}(f) \text{ and } [\![N]\!] \neq \bot\\
[\![\mathbf{rec}\, M]\!]\eta &= \text{the least fixed point of } f \text{ if } [\![M]\!]\eta = \mathbf{Fun}(f)\\
[\![C(M_1, \ldots, M_k)]\!]\eta &= C([\![M_1]\!]\eta, \ldots, [\![M_k]\!]\eta) \quad (C \text{ a constructor (including } \mathbf{Amb}))\\
[\![\mathbf{case}\, M\, \mathbf{of}\, \vec{C}l\}]\!]\eta &= [\![K]\!]\eta[\vec{a} \mapsto \vec{d}] \quad \text{if } [\![M]\!]\eta = C(\vec{d}) \text{ and } C(\vec{a}) \to K \in \vec{C}l\\
[\![M]\!]\eta &= \bot \quad \text{in all other cases, in particular } [\![\bot]\!]\eta = \bot
\end{aligned}
$$

η is an environment that assigns elements of D to variables.

$$
\begin{aligned}
D^{\zeta}_{\alpha} &= \zeta(\alpha), \qquad D^{\zeta}_{\mathbf{1}} = \{\mathbf{Nil}, \bot\},\\
D^{\zeta}_{\mathbf{fix}\,\alpha\,.\,\rho} &= \bigcap\{X \lhd D \mid D^{\zeta[\alpha \mapsto X]}_{\rho} \subseteq X\} \qquad (X \lhd D \text{ means } X \text{ is a subdomain of } D)\\
D^{\zeta}_{\mathbf{A}(\rho)} &= \{\mathbf{Amb}(a, b) \mid a, b \in D^{\zeta}_{\rho}\} \cup \{\bot\}\\
D^{\zeta}_{\rho \diamond \sigma} &= (D^{\zeta}_{\rho} \diamond D^{\zeta}_{\sigma})_{\bot} \quad (\diamond \in \{+, \times, \Rightarrow\})
\end{aligned}
$$

ζ is a type environment that assigns subdomains D to type variables.

Fig. 2. Denotational semantics of programs (Phase I) and types

$\mathrm{data}(\mathsf{mapamb}\ f\ \mathbf{Amb}(0,1)) = \mathrm{data}(\mathbf{Amb}(0,\bot)) = \{0\}$. In Sect. 3 we will define an operational semantics whose fair execution sequences starting with a regular-typed program M compute exactly the elements in $\mathrm{data}([\![M]\!])$.

Example 1. Let $M = \mathbf{rec}\ \lambda a.\mathbf{Amb}(\mathbf{Left}(\mathbf{Nil}), \mathbf{Right}(a))$. M is a closed program of type $\mathbf{fix}\,\alpha\,.\,\mathbf{A}(\mathbf{1} + \alpha)$. We have $\mathrm{data}(M) = \{0, 1, 2, \ldots\}$. Thus, we can express countable choice (random assignment) with **Amb**.

Lemma 1. *If $a \in D$ belongs to a regular type, then the following are equivalent: (1) $a \in \{\bot, \mathbf{Amb}(\bot, \bot)\}$; (2) $\{\bot\} = \mathrm{data}(a)$; (3) $\bot \in \mathrm{data}(a)$.*

3 Operational semantics

We define a small-step operational semantics that, in the limit, reduces each closed program M nondeterministically to an element in $\mathrm{data}([\![M]\!])$ (Thm. 1). If M has a regular type, the converse holds as well: For every $d \in \mathrm{data}([\![M]\!])$ there exists a reduction sequence for M computing d in the limit (Thm. 2). If M denotes a compact data, then the limit is obtained after finitely many reductions. In the following, all programs are assumed to be closed.

3.1 Reduction to weak head normal form

A program is called a *weak head normal form (w.h.n.f.)* if it begins with a constructor (including **Amb**), or has the form $\lambda a.M$. We define inductively a

small-step leftmost-outermost reduction relation $\rightsquigarrow$ on programs where C ranges over constructors.

(s-i) $(\lambda a.\, M)\ N \rightsquigarrow M[N/a]$

(s-ii) $\dfrac{M \rightsquigarrow M'}{M\,N \rightsquigarrow M'\,N}$

(s-iii) $(\lambda a.\, M)\downarrow N \rightsquigarrow M[N/a]$ if N is a w.h.n.f.

(s-iv) $\dfrac{M \rightsquigarrow M'}{M\downarrow N \rightsquigarrow M'\downarrow N}$ if N is a w.h.n.f.

(s-v) $\dfrac{N \rightsquigarrow N'}{M\downarrow N \rightsquigarrow M\downarrow N'}$

(s-vi) $\mathbf{rec}\,M \rightsquigarrow M\,(\mathbf{rec}\,M)$

(s-vii) $\mathbf{case}\,C(\vec{M})\,\mathbf{of}\,\{\ldots; C(\vec{b}) \to N; \ldots\} \rightsquigarrow N[\vec{M}/\vec{b}]$

(s-viii) $\dfrac{M \rightsquigarrow M'}{\mathbf{case}\,M\,\mathbf{of}\,\{\vec{C}\vec{l}\} \rightsquigarrow \mathbf{case}\,M'\,\mathbf{of}\,\{\vec{C}\vec{l}\}}$

(s-ix) $M \rightsquigarrow \bot$ if M is $\bot$-like (see below)

$\bot$-like programs are such that their syntactic forms immediately imply that they denote $\bot$, more precisely they are of the form $\bot$, $C(\vec{M})\,N$, $C(\vec{M})\downarrow N$, and $\mathbf{case}\,M\,\mathbf{of}\,\{\ldots\}$ where M is a lambda-abstraction or of the form $C(\vec{M})$ such that there is no clause in $\{\ldots\}$ which is of the form $C(\vec{a}) \to N$. W.h.n.f.s are never $\bot$-like, and the only typeable $\bot$-like program is $\bot$.

Lemma 2. *(1) $\rightsquigarrow$ is deterministic (i.e., $M \rightsquigarrow M'$ for at most one M').*
(2) $\rightsquigarrow$ preserves the denotational semantics (i.e., $[\![M]\!] = [\![M']\!]$ if $M \rightsquigarrow M'$).
(3) M is a $\rightsquigarrow$-normal form iff M is a w.h.n.f.
(4) [Adequacy Lemma] If $[\![M]\!] \neq \bot$, then there is a w.h.n.f. V s.t. $M \rightsquigarrow^* V$.

3.2 Making choices

Next, we define the reduction relation $\overset{c}{\rightsquigarrow}$ ('c' for 'choice') that reduces arguments of **Amb** in parallel.

(c-i) $\dfrac{M \rightsquigarrow M'}{M \overset{c}{\rightsquigarrow} M'}$

(c-ii) $\dfrac{M_1 \rightsquigarrow M_1'}{\mathbf{Amb}(M_1, M_2) \overset{c}{\rightsquigarrow} \mathbf{Amb}(M_1', M_2)}$

(c-ii') $\dfrac{M_2 \rightsquigarrow M_2'}{\mathbf{Amb}(M_1, M_2) \overset{c}{\rightsquigarrow} \mathbf{Amb}(M_1, M_2')}$

(c-iii) $\mathbf{Amb}(M_1, M_2) \overset{c}{\rightsquigarrow} M_1$ if M_1 is a w.h.n.f.

(c-iii') $\mathbf{Amb}(M_1, M_2) \overset{c}{\rightsquigarrow} M_2$ if M_2 is a w.h.n.f.

From this definition and Lemma 2, it is immediate that M is a $\overset{\mathrm{c}}{\rightsquigarrow}$-normal form iff M is a *deterministic weak head normal form (d.w.h.n.f.)*, that is, a w.h.n.f. that does not begin with **Amb**. Finally, we define a reduction relation $\overset{\mathrm{p}}{\rightsquigarrow}$ that reduces arguments of data constructors in parallel.

$$\text{(p-i)}\quad \frac{M \overset{\mathrm{c}}{\rightsquigarrow} M'}{M \overset{\mathrm{p}}{\rightsquigarrow} M'}$$

$$\text{(p-ii)}\quad \frac{M_i \overset{\mathrm{p}}{\rightsquigarrow} M_i' \ (i = 1, \ldots, k)}{C(M_1, \ldots, M_k) \overset{\mathrm{p}}{\rightsquigarrow} C(M_1', \ldots, , M_k')}\ (C \in \mathrm{C_d})$$

$$\text{(p-iii)}\quad \lambda a.\, M \overset{\mathrm{p}}{\rightsquigarrow} \lambda a.\, M$$

Every (closed) program reduces under $\overset{\mathrm{p}}{\rightsquigarrow}$ (easy proof by structural induction). For example, **Nil** $\overset{\mathrm{p}}{\rightsquigarrow}$ **Nil** by (p-ii). In the following, all $\overset{\mathrm{p}}{\rightsquigarrow}$-reduction sequences are assumed to be infinite.

We call a $\overset{\mathrm{p}}{\rightsquigarrow}$-reduction sequence *unfair* if, intuitively, from some point on, one side of an **Amb** term is permanently reduced but not the other. More precisely, we inductively define $M_1 \overset{\mathrm{p}}{\rightsquigarrow} M_2 \overset{\mathrm{p}}{\rightsquigarrow} \ldots$ to be unfair if

- each M_i is of the form $\mathbf{Amb}(L_i, R)$ (with fixed R) and $L_i \rightsquigarrow L_{i+1}$, or
- each M_i is of the form $\mathbf{Amb}(L, R_i)$ (with fixed L) and $R_i \rightsquigarrow R_{i+1}$, or
- each M_i is of the form $C(N_{i,1}, \ldots, N_{i,n})$ (with a fixed n-ary constructor C) and $N_{1,k} \overset{\mathrm{p}}{\rightsquigarrow} N_{2,k} \overset{\mathrm{p}}{\rightsquigarrow} \ldots$ is unfair for some k, or
- the tail of the sequence, $M_2 \overset{\mathrm{p}}{\rightsquigarrow} M_3 \ldots$, is unfair.

A $\overset{\mathrm{p}}{\rightsquigarrow}$-reduction sequence is *fair* if it is not unfair.

Intuitively, reduction by $\overset{\mathrm{p}}{\rightsquigarrow}$ proceeds as follows: A program L is head reduced by $\rightsquigarrow$ to a w.h.n.f. L', and if L' is a data constructor term, all arguments are reduced in parallel by (p-ii). If L' has the form $\mathbf{Amb}(M, N)$, two concurrent threads are invoked for the reductions of M and N in parallel, and the one reduced to a w.h.n.f. first is used. Fairness corresponds to the fact that the 'speed' of each thread is positive which means, in particular, that no thread can block another. Note that $\overset{\mathrm{c}}{\rightsquigarrow}$ is not used for the reductions of M and N in (s-ii), (s-iv), (s-v) and (s-viii). This means that $\overset{\mathrm{c}}{\rightsquigarrow}$ is applied only to the outermost redex. Also, (c-ii) is defined through $\rightsquigarrow$, not $\overset{\mathrm{c}}{\rightsquigarrow}$, and thus no thread creates new threads. This ability to limit the bound of threads was not available in an earlier version of this language [5] (see also the discussion in Sect. 8.1).

3.3 Computational adequacy: Matching denotational and operational semantics

We define $M_D \in D$ by structural induction on programs:

$$\begin{aligned} C(M_1, \ldots, M_k)_D &= C(M_{1\,D}, \ldots, M_{k\,D}) && (C \in \mathrm{C_d}) \\ (\lambda a.M)_D &= [\![\lambda a.M]\!] \\ M_D &= \bot && \text{otherwise} \end{aligned}$$

Since clearly $M \overset{\mathrm{p}}{\rightsquigarrow} N$ implies $M_D \sqsubseteq_D N_D$, for every computation sequence $M_0 \overset{\mathrm{p}}{\rightsquigarrow} M_1 \overset{\mathrm{p}}{\rightsquigarrow} \ldots$, the sequence $((M_i)_D)_{i\in\mathbf{N}}$ is increasing and therefore has a least upper bound in D. Intuitively, M_D is the part of M that has been fully evaluated to a data.

A *computation* of M is an infinite fair sequence $M = M_0 \overset{\mathrm{p}}{\rightsquigarrow} M_1 \overset{\mathrm{p}}{\rightsquigarrow} \ldots$.

Theorem 1 (Computational Adequacy: Soundness). *For every computation* $M = M_0 \overset{\mathrm{p}}{\rightsquigarrow} M_1 \overset{\mathrm{p}}{\rightsquigarrow} \ldots$, $\sqcup_{i\in\mathbf{N}}(M_i)_D \in \mathrm{data}([\![M]\!])$.

The converse does not hold in general, i.e. $d \in \mathrm{data}([\![M]\!])$ does not necessarily imply $d = \sqcup_{i\in\mathbf{N}}((M_i)_D)$ for some computation of M. For example, for $M \overset{\mathrm{Def}}{=} \mathbf{rec}\,\lambda a.\,\mathbf{Amb}(a,\bot)$ (for which $[\![M]\!] = [\![\mathbf{Amb}(M,\bot)]\!]$) one sees that $d \in \mathrm{data}([\![M]\!])$ for every $d \in D$ while $M \overset{\mathrm{p}}{\rightsquigarrow}{}^{*} M$ and $M_D = \bot$. But M has the type $\mathbf{fix}\,\alpha\,.\,\mathbf{A}(\alpha)$ which is not regular (see Sect. 2.1). For programs of a regular type, the converse of Thm. 1 holds.

Theorem 2 (Computational Adequacy: Completeness). *If M has a regular type, then for every $d \in \mathrm{data}([\![M]\!])$, there is a computation $M = M_0 \overset{\mathrm{p}}{\rightsquigarrow} M_1 \overset{\mathrm{p}}{\rightsquigarrow} \ldots$ with $d = \sqcup_{i\in\mathbf{N}}((M_i)_D)$.*

A computation $M = M_0 \overset{\mathrm{p}}{\rightsquigarrow} M_1 \overset{\mathrm{p}}{\rightsquigarrow} \ldots$ is *productive* if some M_i is a deterministic w.h.n.f. Clearly, this is the case iff $\sqcup_{i\in\mathbf{N}}((M_i)_D) \neq \bot$. Therefore, by the Adequacy Theorem and Lemma 1:

Corollary 1. *For a program M of regular type, the following are equivalent.*

(1) One of the computations of M is productive.
(2) All computations of M are productive.
(3) $[\![M]\!]$ is neither $\bot$ nor $\mathbf{Amb}(\bot,\bot)$.

The corollary does not hold without the regularity condition. For example, $M = \mathbf{Amb}(\mathbf{Amb}(\mathbf{Nil},\mathbf{Nil}),\mathbf{Amb}(\bot,\bot))$ can be reduced to $M_1 = \mathbf{Amb}(\bot,\bot)$ and then repeats M_1 forever, whereas it can also be reduced to **Nil**. McCarthy's **amb** operator is bottom-avoiding in that when it can terminate, it always terminates. Corollary 1 guarantees a similar property for our globally angelic choice operator **Amb**.

4 CFP (Concurrent Fixed Point Logic)

In [12], the system IFP (Intuitionistic Fixed Point Logic) was introduced. IFP is an intuitionistic first-order logic with strictly positive inductive and coinductive definitions, from the proofs of which programs can be extracted. CFP is obtained by adding to IFP two propositional operators, $B|_A$ and $\downarrow\!\downarrow(B)$, that facilitate the extraction of nondeterministic and concurrent programs.

4.1 Syntax

CFP is defined relative to a many-sorted first-order language. CFP-formulas have the form $A \wedge B$, $A \vee B$, $A \to B$, $\forall x\, A$, $\exists x\, A$, $s = t$ (s, t terms of the same sort), $P(\vec{t})$ (for a predicate P and terms $\vec{t}$ of fitting arities), as well as $B|_A$ (restriction) and $\downdownarrows(B)$ (concurrency). Predicates are either predicate constants (as given by the first-order language), or predicate variables (denoted $X, Y, \ldots$), or comprehensions $\lambda\vec{x}\, A$ (where A is a formula and $\vec{x}$ is a tuple of first-order variables), or fixed points $\mu(\Phi)$ and $\nu(\Phi)$ (least fixed point aka inductive predicate and greatest fixed point aka coinductive predicate) where Φ is a strictly positive (s.p.) operator. Operators are of the form $\lambda X\, Q$ where X is a predicate variable and Q is a predicate of the same arity as X. $\lambda X\, Q$ is s.p. if every free occurrence of X in Q is at a strictly positive position, that is, at a position that is not in the left part of an implication. We identify $(\lambda\vec{x}\, A)(\vec{t})$ with $A[\vec{t}/\vec{x}]$ where $[\vec{t}/\vec{x}]$ means capture avoiding substitution.

The following syntactic properties of expressions (i.e., formulas, predicates and operators) will be important. A *Harrop* expression is one that contains at strictly positive positions neither free predicate variables nor disjunctions ($\vee$) nor restrictions ($|$) nor concurrency ($\downdownarrows$). An expression is *non-Harrop* if it is not Harrop; it is *non-computational (nc)* if it contains neither disjunctions, nor restrictions nor concurrency nor free predicate variables. Every nc-formula is Harrop but not conversely. Finally, we define, recursively, when a formula is *strict*: Harrop formulas and disjunctions are strict. A non-Harrop conjunction is strict if either both conjuncts are non-Harrop or it is a conjunction of a Harrop formula and a strict formula. A non-Harrop implication is strict if the premise is non-Harrop. Formulas of the form $\diamond x\, A$ ($\diamond \in \{\forall, \exists\}$) or $\Box(\lambda X \lambda\vec{x}\, A)$ ($\Box \in \{\mu, \nu\}$) are *strict* if A is strict. Formulas of other forms (e.g., $B|_A$, $\downdownarrows(A)$, $X(\vec{t})$) are not strict. The significance of these definitions is that Harropness ensures that (a proof of) the formula will have no computational content. Strictness ensures, among other things, that $\bot$ is not a realizer (see Sect. 5).

As an additional requirement for formulas to be wellformed we demand that in formulas of the form $B|_A$ or $\downdownarrows(B)$, B must be strict.

Notation: $P(\vec{t})$ will also be written $\vec{t} \in P$, and if Φ is $\lambda X\, Q$, then $\Phi(P)$ stands for $Q[P/X]$. Definitions (on the meta level) of the form $P \stackrel{\text{Def}}{=} \Box(\Phi)$ ($\Box \in \{\mu, \nu\}$) where $\Phi = \lambda X\, \lambda\vec{x}\, A$, will usually be written $P(\vec{x}) \stackrel{\Box}{=} A[P/X]$. We write $P \subseteq Q$ for $\forall\vec{x}\ (P(\vec{x}) \to Q(\vec{x}))$, $\forall x \in P\ A$ for $\forall x\ (P(x) \to A)$, and $\exists x \in P\ A$ for $\exists x\ (P(x) \wedge A)$. $\neg A \stackrel{\text{Def}}{=} A \to \mathbf{False}$ where $\mathbf{False} \stackrel{\text{Def}}{=} \mu(\lambda X\, X)$.

4.2 Proof rules

The proof rules of CFP contain those of IFP, which are the usual natural deduction rules for intuitionistic first-order logic with equality (see e.g. [53]), plus the following rules for induction and coinduction, where Φ is a s.p. operator:

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

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

The rules for restriction and concurrency are (with the earlier mentioned condition that in formulas of the form $B|_A$ or $\downdownarrows(B)$, B must be strict):

$$\frac{A \to (B_0 \vee B_1) \quad \neg A \to B_0 \wedge B_1}{(B_0 \vee B_1)|_A}\ \text{Rest-intro } (A, B_0, B_1 \text{ Harrop})$$

$$\frac{B|_A \quad B \to (B'|_A)}{B'|_A}\ \text{Rest-bind} \qquad \frac{B}{B|_A}\ \text{Rest-return}$$

$$\frac{A' \to A \quad B|_A}{B|_{A'}}\ \text{Rest-antimon} \qquad \frac{B|_A \quad A}{B}\ \text{Rest-mp}$$

$$\frac{}{B|_{\mathbf{False}}}\ \text{Rest-efq} \qquad \frac{B|_A}{B|_{\neg\neg A}}\ \text{Rest-stab}$$

$$\frac{B|_A \quad B|_{\neg A}}{\downdownarrows(B)}\ \text{Conc-lem} \qquad \frac{A}{\downdownarrows(A)}\ \text{Conc-return}$$

$$\frac{A \to B \quad \downdownarrows(A)}{\downdownarrows(B)}\ \text{Conc-mp}$$

In Sect. 5 we will prove that each of these rules is realized by a program from our programming language in Sect. 2.

4.3 Tarskian semantics, axioms and classical logic

Although we are mainly interested in the realizability interpretation of CFP, it is important that all proof rules of CFP are also valid w.r.t. a standard Tarskian semantics, provided we identify $B|_A$ with $A \to B$ and $\downdownarrows(B)$ with B.

Like IFP, CFP is parametric in a set $\mathcal{A}$ of *axioms*, which have to be closed nc-formulas. The significance of the restriction to nc-formulas is that these are identical to their (formalized) realizability interpretation (see Sect. 5), in particular, Tarskian and realizability semantics coincide for them. Axioms should be chosen such that they are true in an intended Tarskian model. Since Tarskian semantics admits classical logic, this means that a fair amount of classical logic is available through axioms. For example, for each closed nc-formula $A(\vec{x})$, stability, $\forall \vec{x}\,(\neg\neg A(\vec{x}) \to A(\vec{x}))$ can be postulated as axiom. In addition, the rule (Conc-lem) is a variant of the classical law of excluded middle and (Rest-stab) permits stability for arbitrary right arguments of restriction.

In our examples and case studies we will use an instance of CFP with a sort for real numbers and some standard axiomatization of real closed fields formulated as a set of nc-formulas. In particular, we will freely use constants, operations and relations such as $0, 1, +, -, *, <, |\cdot|, /$ and assume their expected properties as axioms (expressed as nc-formulas).

5 Program extraction

We define a realizability interpretation of CFP that will enable us to extract concurrent programs from proofs. Since the interpretation extends the one in IFP [12], it suffices to define realizability for the restriction and concurrency operators and prove that their proof rules are realizable (Sects. 5.2). All definitions and proofs of this section can be carried out in a formal system RCFP (realizability logic for CFP) which is CFP without $|$ and $\downdownarrows$ but with classical logic and an extended first-order language that contains the earlier introduced programs and types as terms and the typing relation ':' as a predicate constant, and describes their semantics through suitable axioms. In particular, RCFP proves the correctness of extracted programs (Soundness Theorem 3). Since it only matters that RCFP is classically correct (since no realizability interpretation is applied to it), details of RCFP do not matter and are therefore omitted.

5.1 Realizability

Realizability for CFP is formalized in RCFP and follows the pattern in [12]. For every non-Harrop CFP-formula A a type $\tau(A)$ and a RCFP-predicate $\mathbf{R}(A)$ are defined such that $\mathbf{R}(A)$ is a subset of $\tau(A)$ (more precisely, RCFP proves $\forall a(\mathbf{R}(A)(a) \to a : \tau(A))$ hence the interpretation of $\mathbf{R}(A)$ is a subset of $D_{\tau(A)}$). We often write $a\,\mathbf{r}\,A$ for $\mathbf{R}(A)(a)$ ('a realizes A') and $\mathbf{r}\,A$ for $\exists a\,\mathbf{R}(A)(a)$ ('A is realizable').

Since Harrop formulas (see Sect. 4.1) have trivial computational content, it only matters whether they are realizable or not. Therefore, we define for a Harrop formula A, a RCFP-formula $\mathbf{H}(A)$ that represents the realizability interpretation of A, but with suppressed realizer. Formally, we define by simultaneous recursion, for every Harrop CFP-expression E an RCFP-expressions $\mathbf{H}(E)$, and for every non-Harrop CFP-expressions E an RCFP-expressions $\mathbf{R}(E)$. It is convenient to set, in addition, for Harrop formulas $\tau(A) \stackrel{\text{Def}}{=} \mathbf{1}$ and $\mathbf{R}(A) \stackrel{\text{Def}}{=} \lambda a\,(a = \mathbf{Nil} \wedge \mathbf{H}(A))$, so that $\tau(A)$ and $\mathbf{R}(A)$ are defined for *all* CFP-formulas.

The complete definition, which is shown in Fig. 3, assumes that to each CFP predicate variable X there are assigned a fresh type variable α_X and a fresh RCFP predicate variable $\hat{X}$ with one extra argument for domain elements. Furthermore, to define realizability for the fixed points of a Harrop operator $\lambda X\,P$, we use the notation

$$\mathbf{H}_X(P) \stackrel{\text{Def}}{=} \mathbf{H}(P[\hat{X}/X])[X/\hat{X}]$$

where $\hat{X}$ is a fresh predicate constant assigned to the (non-Harrop) predicate variable X. This is motivated by the fact that $\lambda X\,P$ is Harrop iff $P[\hat{X}/X]$ is. The idea is that $\mathbf{H}_X(P)$ is the same as $\mathbf{H}(P)$ but considering X as a (Harrop) predicate constant.

To see that the definitions make sense, note that a formula $P(\vec{t})$ is Harrop iff P is, predicate variables and disjunctions are always non-Harrop, a conjunction is Harrop iff both conjuncts are, an implication $A \to B$ is Harrop iff B is, and

For Harrop formulas A: $\tau(A) = \mathbf{1}$ and $\mathbf{R}(A) = \lambda a\,(a = \mathbf{Nil} \wedge \mathbf{H}(A))$.

$\tau(E)$ for non-Harrop expressions E:

$$\tau(P(\vec{t})) = \tau(P) \qquad \tau(A \vee B) = \tau(A) + \tau(B)$$

$$\tau(A \wedge B) = \begin{cases} \tau(A) \times \tau(B) & (A, B \text{ non-Harrop}) \\ \tau(A) & (B \text{ Harrop}) \\ \tau(B) & (A \text{ Harrop}) \end{cases}$$

$$\tau(A \to B) = \begin{cases} \tau(A) \Rightarrow \tau(B) & (A \text{ non-Harrop}) \\ \tau(B) & (A \text{ Harrop}) \end{cases}$$

$$\tau(B|_A) = \tau(B) \qquad \tau(\Downarrow\!(B)) = \mathbf{A}(\tau(B))$$

$$\tau(\diamond x\, A) = \tau(A) \qquad (\diamond \in \{\forall, \exists\})$$

$$\tau(X) = \alpha_X \qquad \tau(P) = \mathbf{1} \quad (P \text{ a predicate constant})$$

$$\tau(\lambda \vec{x}\, A) = \tau(A) \qquad \tau(\Box(\lambda X\, P)) = \mathbf{fix}\,\alpha_X\,.\,\tau(P) \qquad (\Box \in \{\mu, \nu\})$$

$\mathbf{R}(E)$ for non-Harrop expressions E:

$$\mathbf{R}(P(\vec{t})) = \lambda a\,(\mathbf{R}(P)(\vec{t}, a))$$

$$\mathbf{R}(A \vee B) = \lambda c\,(\exists a\,(c = \mathbf{Left}(a) \wedge a\,\mathbf{r}\,A) \vee \exists b\,(c = \mathbf{Right}(b) \wedge b\,\mathbf{r}\,B))$$

$$\mathbf{R}(A \wedge B) = \begin{cases} \lambda c\,(\exists a, b\,(c = \mathbf{Pair}(a, b) \wedge a\,\mathbf{r}\,A \wedge b\,\mathbf{r}\,B)) & (A, B \text{ non-Harrop}) \\ \lambda a\,(a\,\mathbf{r}\,A \wedge \mathbf{H}(B)) & (B \text{ Harrop}) \\ \lambda b\,(\mathbf{H}(A) \wedge b\,\mathbf{r}\,B) & (A \text{ Harrop}) \end{cases}$$

$$\mathbf{R}(A \to B) = \begin{cases} \lambda c\,(c : \tau(A) \Rightarrow \tau(B) \wedge \forall a\,(a\,\mathbf{r}\,A \to (c\,a)\,\mathbf{r}\,B)) & (A \text{ non-Harrop}) \\ \lambda b\,(b : \tau(B) \wedge (\mathbf{H}(A) \to b\,\mathbf{r}\,B)) & (A \text{ Harrop}) \end{cases}$$

$$\mathbf{R}(B|_A) = \lambda b\,(b{:}\tau(B) \wedge (\mathbf{r}\,A \to b \neq \bot) \wedge (b \neq \bot \to b\,\mathbf{r}\,B))$$

$$\mathbf{R}(\Downarrow\!(B)) = \lambda c\,\exists a, b\ (c = \mathbf{Amb}(a, b) \wedge a, b : \tau(B) \wedge (a \neq \bot \vee b \neq \bot) \wedge (a \neq \bot \to a\,\mathbf{r}\,B) \wedge (b \neq \bot \to b\,\mathbf{r}\,B))$$

$$\mathbf{R}(\diamond x\, A) = \lambda a\,(\diamond x\,(a\,\mathbf{r}\,A)) \qquad (\diamond \in \{\forall, \exists\})$$

$$\mathbf{R}(X) = \tilde{X} \qquad \mathbf{R}(\lambda \vec{x}\, A) = \lambda(\vec{x}, a)\,(a\,\mathbf{r}\,A)$$

$$\mathbf{R}(\Box(\lambda X\, P)) = \Box(\lambda \tilde{X}\,\mathbf{R}(P)) \qquad (\Box \in \{\mu, \nu\})$$

$\mathbf{H}(E)$ for Harrop expressions E:

$$\mathbf{H}(P(\vec{t})) = \mathbf{H}(P)(\vec{t}) \qquad \mathbf{H}(A \wedge B) = \mathbf{H}(A) \wedge \mathbf{H}(B)$$

$$\mathbf{H}(A \to B) = \begin{cases} \mathbf{r}\,A \to \mathbf{H}(B) & (A \text{ non-Harrop}) \\ \mathbf{H}(A) \to \mathbf{H}(B) & (A \text{ Harrop}) \end{cases}$$

$$\mathbf{H}(\diamond x\, A) = \diamond x\,\mathbf{H}(A) \quad (\diamond \in \{\forall, \exists\})$$

$$\mathbf{H}(P) = P \quad (P \text{ a predicate constant}) \qquad \mathbf{H}(\lambda \vec{x}\, A) = \lambda \vec{x}\,\mathbf{H}(A)$$

$$\mathbf{H}(\Box(\lambda X\, P)) = \Box(\lambda X\,\mathbf{H}_X(P)) \qquad (\Box \in \{\mu, \nu\})$$

Fig. 3. Realizability interpretation of CFP

$\forall x\, A$, $\exists x\, A$, $\lambda \vec{x}\, A$ are Harrop iff A is. The rationale and correctness of realizability for restriction and concurrency are discussed in Sect. 5.2.

If a formula A is nc, then it is Harrop (see Sect. 4.1 for definitions) but in addition A and $\mathbf{H}(A)$ are syntactically identical. In contrast, in general, a Harrop formula A neither implies nor is implied by $\mathbf{H}(A)$.

Lemma 3. *For every* CFP*-formula A:*

(1) $\tau(A)$ *is a regular type.*
(2) If A is strict, then $\bot$ *does not realize A, provably in* RCFP*.*
(3) $\mathbf{Amb}(\bot, \bot)$ *is not a realizer of A.*
(4) For a program M that realizes A, t.f.a.e.: (i) M has some productive computation; (ii) all computations of M are productive; (iii) $[\![M]\!] \neq \bot$*.*

Proof. (1) and (2) are easily proved by structural induction on formulas. (3) follows from the fact that if A is of the form $\mathbf{Amb}(B)$, then B must be strict. (4) is proved by (3) and Corollary 1 (3).

Remarks and examples. The main difference of our interpretation to the usual realizability interpretation of intuitionistic number theory lies in the interpretation of quantifiers. While in number theory variables range over natural numbers, which have concrete computationally meaningful representations, we make no general assumption of this kind, since it is our goal to extract programs from proofs in abstract mathematics. This is the reason why we interpret quantifiers *uniformly*, that is, a realizer of a universal statement must be independent of the quantified variable and a realizer of an existential statement does not contain a witness. A similar uniform interpretation of quantifiers can be found in the Minlog system. The usual definition of realizability of quantifiers in intuitionistic number theory can be recovered by relativization to an inductively defined predicate $\mathbf{N}$ describing natural numbers in unary representation:

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

which is shorthand for $\mathbf{N} \stackrel{\text{Def}}{=} \mu(\lambda X\, \lambda x\, (x = 0 \vee X(x-1)))$. The type $\tau(\mathbf{N})$ assigned to $\mathbf{N}$ is the recursive type of unary natural numbers

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

Realizability for $\mathbf{N}$ works out as

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

Thus, $\mathbf{N}(0)$, $\mathbf{N}(1)$, $\mathbf{N}(2)$ are realized by $\mathbf{Left}$ (i.e., $\mathbf{Left}(\mathbf{Nil})$), $\mathbf{Right}(\mathbf{Left})$, $\mathbf{Right}(\mathbf{Right}(\mathbf{Left}))$, and so on. Therefore, the (unique) realizer of $\mathbf{N}(n)$ is the unary representation of n. Other ways of defining natural numbers may induce different representations. An example of a formula with interesting realizers is the formula expressing that the sum of two natural number is a natural number,

$$\forall x, y\ (\mathbf{N}(x) \to \mathbf{N}(y) \to \mathbf{N}(x+y)). \tag{4}$$

It has type $\mathbf{nat} \to \mathbf{nat} \to \mathbf{nat}$ and is realized by a function f that, given realizers of $\mathbf{N}(x)$ and $\mathbf{N}(y)$, returns a realizer of $\mathbf{N}(x+y)$, hence f performs addition of unary numbers.

Example 2 (Non-terminating realizer). Let

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

Then $\tau(\mathbf{D}) = \mathbf{2}$ where $\mathbf{2} = \mathbf{1} + \mathbf{1}$, and $a\,\mathbf{r}\,\mathbf{D}(x)$ unfolds to

$$a : \tau(\mathbf{2}) \wedge (x \neq 0 \to (a = \mathbf{Left} \wedge x \leq 0) \vee (a = \mathbf{Right} \wedge x \geq 0)).$$

Therefore, $\mathbf{D}(x)$ is realized by **Left** if $x < 0$ and by **Right** if $x > 0$. If $x = 0$, any element of $\tau(\mathbf{2})$ realizes $D(x)$, in particular $\bot$. Hence, nonterminating programs, which, by Lemma 3 (4), denote $\bot$, realize $D(x)$. In contrast, *strict* formulas are never realized by a nonterminating program, as shown in Lemma 3 (2).

5.2 Partial correctness and concurrency

We explain realizability for $B|_A$ and $\downdownarrows(B)$ and show that the associated proof rules are sound.

As we have seen in Example 2, a realizer of an implication $A \to B$ where A is a Harrop formula is realized by a 'conditionally correct' program M, that is, if $\mathbf{H}(A)$, then M realizes B, but otherwise no condition is imposed on M, in particular M may be non-terminating. However, M may terminate but fail to realize B. This means that termination of a realizer of $A \to B$ is not a sufficient condition for correctness (correctness meaning to realize B). But, as explained in the Introduction, this is what we need to concurrently realize a formula. The definition of realizability for the new logical operator $|$ (shown in Fig. 3) achieves exactly this: A realizer of a restriction $B|_A$ is 'partially correct' in the sense that it is correct iff it terminates. By Lemma 3 (4), for a program M to realize $B|_A$ means that M has type $\tau(B)$, and if A is realizable then all the computations of M are productive, and conversely, if M has a productive computation then M always (that is, independently of the realizability of A) realizes B.

To highlight the difference between restriction and implication in a more concrete situation, consider $(A \vee B)|_A$ vs. $A \to (A \vee B)$ where A is Harrop. Clearly **Left** realizes $A \to (A \vee B)$, but in general $(A \vee B)|_A$ is not realizable. Note that **Left** even realizes $A \stackrel{\mathrm{u}}{\to} (A \vee B)$ where $\stackrel{\mathrm{u}}{\to}$ is Schwichtenberg's uniform implication [39], hence restriction is also different from uniform implication.

The intuition of $\mathbf{Amb}(a,b)$ realizing $\downdownarrows(A)$ is that it is a pair of candidate realizers at least one of which is productive, and each productive one is a realizer.

Lemma 4. *The rules for restriction and concurrency are realizable.*

Proof. The table below shows the realizers of each rule for the (most interesting) case where the conclusion is non-Harrop, using the definitions

$$\mathsf{leftright} \stackrel{\text{Def}}{=} \lambda b.\mathbf{case}\,b\,\mathbf{of}\,\{\mathbf{Left}(\_) \to \mathbf{Left}; \mathbf{Right}(\_) \to \mathbf{Right}\}\,,$$
$$\mathsf{mapamb} \stackrel{\text{Def}}{=} \lambda f.\,\lambda c.\ \mathbf{case}\,c\,\mathbf{of}\ \{\mathbf{Amb}(a,b) \to \mathbf{Amb}(f{\downarrow}a, f{\downarrow}b)\}\,.$$

Proofs of their correctness are in [11]. For (Rest-intro), (Rest-stab), and (Conc-lem), classical logic is needed. Here, we set $a\,\mathbf{seq}\,b \stackrel{\text{Def}}{=} (\lambda c.\ b)\downarrow a$.

$$\frac{b\,\mathbf{r}\,(A \to (B_0 \vee B_1)) \quad \mathbf{H}(\neg A \to B_0 \wedge B_1)}{(\mathsf{leftright}\ b)\,\mathbf{r}\,(B_0 \vee B_1)|_A}\ \text{Rest-intro } (A, B_0, B_1 \text{ Harrop})$$

$$\frac{a\,\mathbf{r}\,B|_A \quad f\,\mathbf{r}\,(B \to (B'|_A))}{(f\downarrow a)\,\mathbf{r}\,B'|_A}\ \begin{array}{l}\text{Rest-bind } (B \text{ non-Harrop})\\ ((a\,\mathbf{seq}\,f)\,\mathbf{r}\,B'|_A\ (B \text{ Harrop}))\end{array} \qquad \frac{a\,\mathbf{r}\,B}{a\,\mathbf{r}\,B|_A}\ \text{Rest-return}$$

$$\frac{\mathbf{r}\,(A' \to A) \quad a\,\mathbf{r}\,B|_A}{a\,\mathbf{r}\,B|_{A'}}\ \text{Rest-antimon} \qquad \frac{b\,\mathbf{r}\,B|_A \quad \mathbf{r}\,A}{b\,\mathbf{r}\,B}\ \text{Rest-mp}$$

$$\frac{}{\bot\,\mathbf{r}\,B|_{\mathbf{False}}}\ \text{Rest-efq} \qquad \frac{b\,\mathbf{r}\,B|_A}{b\,\mathbf{r}\,B|_{\neg\neg A}}\ \text{Rest-stab}$$

$$\frac{a\,\mathbf{r}\,B|_A \quad b\,\mathbf{r}\,B|_{\neg A}}{\mathbf{Amb}(a,b)\,\mathbf{r}\,\Downarrow\!\!\downarrow(B)}\ \text{Conc-lem} \qquad \frac{a\,\mathbf{r}\,A}{\mathbf{Amb}(a,\bot)\,\mathbf{r}\,\Downarrow\!\!\downarrow(A)}\ \text{Conc-return}$$

$$\frac{f\,\mathbf{r}\,(A \to B) \quad c\,\mathbf{r}\,\Downarrow\!\!\downarrow(A)}{(\mathsf{mapamb}\ f\ c)\,\mathbf{r}\,\Downarrow\!\!\downarrow(B)}\ \begin{array}{l}\text{Conc-mp } (A \text{ non-Harrop})\\ (\mathbf{Amb}(f,\bot)\,\mathbf{r}\,\Downarrow\!\!\downarrow(B)\ (A \text{ Harrop}))\end{array}$$

Lemma 5. CFP *derives the following rules. The rules are displayed together with their extracted realizers.*

$$(1)\quad \frac{a\,\mathbf{r}\,B_0|_{A_0} \quad b\,\mathbf{r}\,B_1|_{A_1} \quad \mathbf{H}(\neg\neg(A_0 \vee A_1))}{\mathbf{Amb}(\mathbf{Left}\downarrow a, \mathbf{Right}\downarrow b)\,\mathbf{r}\,\Downarrow\!\!\downarrow(B_0 \vee B_1)}$$

$$(2)\quad \frac{a\,\mathbf{r}\,(B \vee C)|_D}{\mathbf{case}\,a\,\mathbf{of}\,\{\mathbf{Left}(\_) \to \bot; \mathbf{Right}(b) \to b\}\,\mathbf{r}\,C|_{D \wedge \neg B}} \qquad (C \text{ strict})$$

Example 3. Continuing Example 2, we modify $\mathbf{D}(x)$ to

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

A realizer of $\mathbf{D}'(x)$, which has type $\mathbf{2}$, may or may not terminate (non-termination occurs when $x = 0$). However, in case of termination, the result is guaranteed to realize $x \le 0 \vee x \ge 0$. Note that, a realizer of $\mathbf{D}(x)$ also has type $\mathbf{2}$ and may or may not terminate, but there is no guarantee that it realizes $x \le 0 \vee x \ge 0$ when it does terminate. Nevertheless, $\mathbf{D} \subseteq \mathbf{D}'$ follows from (Rest-intro) (since $\neg x \neq 0$ implies $x \le 0 \wedge x \ge 0$) and is realized by $\mathsf{leftright}$. $\mathbf{D}' \subseteq \mathbf{D}$ holds trivially.

Example 4. This builds on the examples 2 and 3 and will be used in Sect. 6. Let $\mathbf{t}(x) = 1 - 2|x|$ and consider the predicates $\mathbf{E}(x) \stackrel{\text{Def}}{=} \mathbf{D}(x) \wedge \mathbf{D}(\mathbf{t}(x))$ and

$$\mathbf{ConSD}(x) \stackrel{\text{Def}}{=} \Downarrow\!\!\downarrow((x \le 0 \vee x \ge 0) \vee |x| \le 1/2).$$

We show $\mathbf{E} \subseteq \mathbf{ConSD}$: From $\mathbf{E}(x)$ and Example 3 we get $\mathbf{D}'(x)$ and $\mathbf{D}'(\mathbf{t}(x))$ which unfolds to $(x \leq 0 \vee x \geq 0)|_{x\neq 0}$ and $(|x| \geq 1/2 \vee |x| \leq 1/2)|_{|x|\neq 1/2}$. By Lemma 5 (2), $(|x| \leq 1/2)|_{|x|<1/2}$. Since $\neg\neg((x \neq 0) \vee |x| < 1/2)$, we have $\mathbf{ConSD}(x)$ by Lemma 5 (1). Moreover, $\tau(\mathbf{E}) = \mathbf{2} \times \mathbf{2}$ and $\tau(\mathbf{ConSD}) = \mathbf{A}(\mathbf{3})$ where $\mathbf{3} \stackrel{\text{Def}}{=} \mathbf{2} + \mathbf{1}$. The extracted realizer of $\mathbf{E} \subseteq \mathbf{ConSD}$ is

$$\mathsf{conSD} \stackrel{\text{Def}}{=} \lambda c.\mathbf{case}\, c\, \mathbf{of}\, \{\mathbf{Pair}(a,b) \to \mathbf{Amb}(\mathbf{Left}{\downarrow}(\mathsf{leftright}\ a), \mathbf{Right}{\downarrow}(\mathbf{case}\, b\, \mathbf{of}\, \{\mathbf{Left}(\_) \to \bot; \mathbf{Right}(\_) \to \mathbf{Nil}\}))\}$$

of type $\tau(\mathbf{E} \subseteq \mathbf{ConSD}) = \mathbf{2} \times \mathbf{2} \to \mathbf{A}(\mathbf{3})$. Explanation of this program: a is **Left** or **Right** depending on whether $x \leq 0$ or $x \geq 0$ but may also be $\bot$ if $x = 0$. b is **Left** or **Right** depending on whether $|x| \leq 1/2$ or $|x| \geq 1/2$ but may also be $\bot$ if $|x| = 1/2$. Since $x = 0$ and $x = 1/2$ do not happen simultaneously, by evaluating a and b concurrently, we obtain one of them from which we can determine one of the cases $x \leq 0$, $x \geq 0$, or $|x| \leq 1/2$.

5.3 Soundness and program extraction

As we did in the above example, one can extract from any CFP-proof of a formula a program that realizes it. This property is called the Soundness Theorem of realizability. Its proof is the same as for IFP [12] but extended by the rules for the new logical operators whose realizability we proved in Sects. 5.2.

Theorem 3 (Soundness Theorem I). *From a* CFP*-proof of a formula A from a set of axioms one can extract a program M of type $\tau(A)$ (which is a regular type) such that* RIFP *proves $M\,\mathbf{r}\,A$ from the same axioms.*

In CFP, we have a second Soundness Theorem which ensures the correctness of all results of fair computation paths of an extracted program M. More precisely, correctness of M means that all $d \in \mathrm{data}([\![M]\!])$ realize the formula A^- obtained from A by deleting all concurrency operators $\downdownarrows$. Since A^- is an IFP formula, the Theorem relates the realizability interpretations of CFP and IFP.

However, such a correctness result only holds for formulas whose realizers do not contain **Amb** in the scope of a lambda-abstraction. This restriction is enforced by the following syntactic admissibility condition: An expression is called *admissible* if it contains neither free predicate variables nor restrictions ($|$), and all occurrences of concurrency ($\downdownarrows$) are strictly positive and at non-F-position. Here, the notion of a *subexpression at F-position* in a CFP expression is defined inductively by three rules: (i) A subexpression of the form $A \to B$ where A and B are both non-Harrop is at F-position. (ii) A subexpression $\Box\, \lambda X\, Q$ ($\Box \in \{\mu, \nu\}$) is at F-position if Q has a free occurrence of X at F-position. (iii) A subexpression within a subexpression at F-Position is at F-position.

For example, $\downdownarrows(\mu(\lambda X\, \lambda x\, (x = 0 \vee \forall y\, (\mathbf{N}(y) \to X(f(x,y))))))$ is admissible, whereas $\mu(\lambda X\, \lambda x\, \downdownarrows(x = 0 \vee \forall y\, (\mathbf{N}(y) \to X(f(x,y)))))$ is not. The predicate **ConSD** in Example 4 is admissible.

Theorem 4 (Faithfulness). *If $a \in D$ realizes an admissible formula A, then all $d \in \mathrm{data}(a)$ realize A^-.*

Theorems 3 and 4 imply:

Theorem 5 (Soundness Theorem II). *From a* CFP *proof of an admissible formula A from a set of axioms one can extract a program $M : \tau(A)$ such that* RCFP *proves* $\forall d \in \mathrm{data}([\![M]\!])\, d\,\mathbf{r}\, A^-$ *from the same set of axioms.*

Thms. 5 and 1, together with and classical soundness (see Sect. 4.3), yield:

Theorem 6 (Program Extraction). *From a* CFP *proof of an admissible formula A from a set of axioms one can extract a program $M : \tau(A)$ such that for any computation $M = M_0 \overset{\mathrm{p}}{\rightsquigarrow} M_1 \overset{\mathrm{p}}{\rightsquigarrow} \ldots$, $\sqcup_{i\in\mathbf{N}}(M_i)_D$ realizes A^- in every model of the axioms.*

6 Application

As our main case study, we extract a concurrent conversion program between two representations of real numbers in [-1, 1], the signed digit representation and infinite Gray code. In the following, we also write $d : p$ for $\mathbf{Pair}(d, p)$.

The signed digit representation is an extension of the usual binary expansion that uses the set $\mathbf{SD} \overset{\mathrm{Def}}{=} \{-1, 0, 1\}$ of *signed digits*. The following predicate $\mathbf{S}(x)$ expresses coinductively that x has a signed digit representation.

$$\mathbf{S}(x) \overset{\nu}{=} |x| \leq 1 \wedge \exists\, d \in \mathbf{SD}\, \mathbf{S}(2x - d)\,,$$

with $\mathbf{SD}(d) \overset{\mathrm{Def}}{=} (d = -1 \vee d = 1) \vee d = 0$. The type of $\mathbf{S}$ is $\tau(\mathbf{S}) = \mathbf{3}^\omega$ where $\mathbf{3} \overset{\mathrm{Def}}{=} (\mathbf{1} + \mathbf{1}) + \mathbf{1}$ and $\delta^\omega \overset{\mathrm{Def}}{=} \mathbf{fix}\, \alpha\,.\, \delta \times \alpha$, and its realizability interpretation is

$$p\,\mathbf{r}\,\mathbf{S}(x) \overset{\nu}{=} |x| \leq 1 \wedge \exists\, d \in \mathbf{SD}\, \exists p'\ (p = d : p' \ \wedge p'\,\mathbf{r}\,\mathbf{S}(2x - d))$$

which expresses indeed that p is a signed digit representation of x, that is, $p = d_0 : d_1 : \ldots$ with $d_i \in \mathbf{SD}$ and $x = \sum_i d_i 2^{-(i+1)}$. Here, we identified the three digits $d = -1, 1, 0$ with their realizers $\mathbf{Left}(\mathbf{Left}), \mathbf{Left}(\mathbf{Right}), \mathbf{Right}$.

Infinite Gray code ([18,42]) is an almost redundancy free representation of real numbers in [-1, 1] using the partial digits $\{-1, 1, \bot\}$. A stream $p = d_0 : d_1 : \ldots$ of such digits is an infinite Gray code of x iff $d_i = \mathrm{sgb}(\mathbf{t}^i(x))$ where $\mathbf{t}$ is the tent function $\mathbf{t}(x) = 1 - |2x|$ and sgb is a multi-valued version of the sign function for which sgb(0) is any element of $\{-1, 1, \bot\}$ (see also Example 4). One easily sees that $\mathbf{t}^i(x) = 0$ for at most one i. Therefore, this coding has little redundancy in that the code is uniquely determined and total except for at most one digit which may be undefined. Hence, infinite Gray code is accessible through concurrent computation with two threads. The coinductive predicate

$$\mathbf{G}(x) \overset{\nu}{=} |x| \leq 1 \wedge \mathbf{D}(x) \wedge \mathbf{G}(\mathbf{t}(x))\,,$$

where $\mathbf{D}$ is the predicate $\mathbf{D}(x) \overset{\mathrm{Def}}{=} x \neq 0 \to (x \leq 0 \vee x \geq 0)$ from Example 2, expresses that x has an infinite Gray code (identifying $-1, 1, \bot$ with $\mathbf{Left}, \mathbf{Right}, \bot$). Indeed, $\tau(\mathbf{G}) = \mathbf{2}^\omega$ and

$$p\,\mathbf{r}\,\mathbf{G}(x) \overset{\nu}{=} |x| \leq 1 \wedge \exists d, p' (p = d : p' \wedge (x \neq 0 \to d\,\mathbf{r}\,(x \leq 0 \vee x \geq 0)) \wedge p'\,\mathbf{r}\,\mathbf{G}(\mathbf{t}(x)))\,.$$

In [12], the inclusion $\mathbf{S} \subseteq \mathbf{G}$ was proved in IFP and a sequential conversion function from signed digit representation to infinite Gray code extracted. On the other hand, a program producing a signed digit representation from an infinite Gray code cannot access its input sequentially from left to right since it will diverge when it accesses $\bot$. Therefore, the program needs to evaluate two consecutive digits concurrently to obtain at least one of them. With this idea in mind, we define a concurrent version of $\mathbf{S}$ as

$$\mathbf{S}_2(x) \overset{\nu}{=} |x| \leq 1 \wedge \Downarrow\!\Downarrow(\exists\, d \in \mathbf{SD}\, \mathbf{S}_2(2x - d))$$

with $\tau(\mathbf{S}_2) = \mathbf{fix}\,\alpha\,.\,\mathbf{A}(\mathbf{3} \times \alpha)$ and prove $\mathbf{G} \subseteq \mathbf{S}_2$ in CFP (Thm. 7). Then we can extract from the proof a concurrent algorithm that converts infinite Gray code to signed digit representation. Note that, while the formula $\mathbf{G} \subseteq \mathbf{S}_2$ is *not* admissible (it contains $\Downarrow\!\Downarrow$ at an F-position), the formula $\mathbf{S}_2(x)$ *is*. Therefore, if for some real number x we can prove $\mathbf{G}(x)$, the proof of $\mathbf{G} \subseteq \mathbf{S}_2$ will give us a proof of $\mathbf{S}_2(x)$ to which Theorem 6 applies. Since $\mathbf{S}_2(x)^-$ is $\mathbf{S}(x)$, this means that we have a nondeterministic program all whose fair computation paths will result in a (deterministic) signed digit representation of x.

Now we carry out the proof of $\mathbf{G} \subseteq \mathbf{S}_2$. For simplicity, we use pattern matching on constructor expressions for defining functions. For example, we write $\mathsf{f}\ (a : t) \overset{\text{Def}}{=} M$ for $\mathsf{f} \overset{\text{Def}}{=} \lambda x.\, \mathbf{case}\, x\, \mathbf{of}\ \{\mathbf{Pair}(a, t) \to M\}$.

The crucial step in the proof is accomplished by Example 4, since it yields nondeterministic information about the first digit of the signed digit representation of x, as expressed by the predicate

$$\mathbf{ConSD}(x) \overset{\text{Def}}{=} \Downarrow\!\Downarrow((x \leq 0 \vee x \geq 0) \vee |x| \leq 1/2).$$

Lemma 6. $\mathbf{G} \subseteq \mathbf{ConSD}$.

Proof. $\mathbf{G}(x)$ implies $\mathbf{D}(x)$ and $\mathbf{D}(\mathbf{t}(x))$, and hence $\mathbf{ConSD}$, by Example 4.

The extracted program $\mathsf{gscomp} : \mathbf{2}^\omega \Rightarrow \mathbf{A}(\mathbf{3})$ uses the program conSD defined in Example 4:

$$\mathsf{gscomp}\ (a : b : p) \overset{\text{Def}}{=} \mathsf{conSD}\ (\mathbf{Pair}(a, b))\,.$$

We also need the following closure properties of $\mathbf{G}$:

Lemma 7. *Assume* $\mathbf{G}(x)$. *Then:*

(1) $\mathbf{G}(\mathbf{t}(x))$, $\mathbf{G}(|x|)$, *and* $\mathbf{G}(-x)$;
(2) if $x \geq 0$, *then* $\mathbf{G}(2x - 1)$ *and* $\mathbf{G}(1 - x)$;
(3) if $|x| \leq 1/2$, *then* $\mathbf{G}(2x)$.

Proof. This follows directly from the definition of $\mathbf{G}$ and elementary properties of the tent function $\mathbf{t}$. The extracted programs consist of simple manipulations of the given digit stream realizing $\mathbf{G}(x)$, concerning only its tail and first two digits. No nondeterminism is involved. A detailed proof is in [11].

Theorem 7. $\mathbf{G} \subseteq \mathbf{S}_2$.

Proof. By coinduction. Setting $A(x) \stackrel{\text{Def}}{=} \exists d \in \mathbf{SD}\, \mathbf{G}(2x - d)$, we have to show

$$\mathbf{G}(x) \to |x| \le 1 \wedge \Downarrow\!\Downarrow(A(x))\,. \tag{5}$$

Assume $\mathbf{G}(x)$. Then $\mathbf{ConSD}(x)$, by Lemma 6. Therefore, it suffices to show

$$\mathbf{ConSD}(x) \to \Downarrow\!\Downarrow(A(x)) \tag{6}$$

which, with the help of the rule (Conc-mp), can be reduced to

$$(x \le 0 \vee x \ge 0 \vee |x| \le 1/2) \to A(x). \tag{7}$$

(7) can be easily shown using Lemma 7: If $x \le 0$, then $\mathbf{t}(x) = 2x + 1$. Since $\mathbf{G}(\mathbf{t}(x))$, we have $\mathbf{G}(2x - d)$ for $d = -1$. If $x \ge 0$, then $\mathbf{G}(2x - d)$ for $d = 1$ by (2). If $|x| \le 1/2$, then $\mathbf{G}(2x - d)$ for $d = 0$ by (3).

The program $\mathsf{onedigit} : \mathbf{2}^\omega \Rightarrow \mathbf{3} \Rightarrow \mathbf{3} \times \mathbf{2}^\omega$ extracted from the proof of (7) from the assumption $\mathbf{G}(x)$ is

$$\begin{aligned}
\mathsf{onedigit}\ (a : b : p)\ c \stackrel{\text{Def}}{=} \mathbf{case}\, c\, \mathbf{of}\, \{&\mathbf{Left}(d) \to \mathbf{case}\, d\, \mathbf{of}\, \{\\
&\mathbf{Left}(_) \to \mathbf{Pair}(-1, b : p);\\
&\mathbf{Right}(_) \to \mathbf{Pair}(1, (\mathsf{not}\ b) : p)\};\\
&\mathbf{Right}(_) \to \mathbf{Pair}(0, a : (\mathsf{nh}\ p))\}\\
\mathsf{not}\ a \stackrel{\text{Def}}{=} \mathbf{case}\, a\, \mathbf{of}\, \{&\mathbf{Left}(_) \to \mathbf{Right};\\
&\mathbf{Right}(_) \to \mathbf{Left}\}\\
\mathsf{nh}\ (a : p) \stackrel{\text{Def}}{=} &(\mathsf{not}\ a) : p
\end{aligned}$$

This is lifted to a proof of (6) using mapamb (the realizer of (Conc-mp)). Hence the extracted realizer $\mathsf{s} : \mathbf{2}^\omega \Rightarrow \mathbf{A}(\mathbf{3} \times \mathbf{2}^\omega)$ of (5) is

$$\mathsf{s}\ p \stackrel{\text{Def}}{=} \mathsf{mapamb}\ (\mathsf{onedigit}\ p)\ (\mathsf{gscomp}\ p)$$

The main program extracted from the proof of Theorem 7 is obtained from the step function s by a special form of recursion, commonly known as *coiteration*. Formally, we use the realizer of the coinduction rule $\mathbf{COIND}(\Phi_{\mathbf{S}_2}, \mathbf{G})$ where $\Phi_{\mathbf{S}_2}$ is the operator used to define $\mathbf{G}$ as largest fixed point, i.e.

$$\Phi_{\mathbf{S}_2} \stackrel{\text{Def}}{=} \lambda X\, \lambda x\, |x| \le 1 \wedge \Downarrow\!\Downarrow(\exists d \in \mathbf{SD}\, X(2x - d)).$$

The realizer of coinduction (whose correctness is shown in [12]) also uses a program $\mathsf{mon} : (\alpha_X \Rightarrow \alpha_Y) \Rightarrow \mathbf{A}(\mathbf{3} \times \alpha_X) \Rightarrow \mathbf{A}(\mathbf{3} \times \alpha_Y)$ extracted from the canonical proof of the monotonicity of $\Phi_{\mathbf{S}_2}$:

$$\begin{aligned}
&\mathsf{mon}\ f\ p = \mathsf{mapamb}\ (\mathsf{mon}'\ f)\ p\\
&\quad \text{where} \quad \mathsf{mon}'\ f\ (a : t) = a : f\ t
\end{aligned}$$

Putting everything together, we obtain the *infinite Gray code to signed digit representation conversion program* $\mathsf{gtos} : \mathbf{2}^\omega \Rightarrow \mathbf{fix}\,\alpha\,.\,\mathbf{A}(\mathbf{3} \times \alpha)$

$$\mathsf{gtos} \stackrel{\mathrm{rec}}{=} (\mathsf{mon}\ \mathsf{gtos}) \circ \mathsf{s}$$

Using the equational theory of RIFP, one can simplify gtos to the following program. The soundness of RIFP axioms with respect to the denotational semantics and the adequacy property of our language guarantees that these two programs are equivalent.

$$\begin{aligned}
\mathsf{gtos}\ (a : b : t) = \mathbf{Amb}(&\\
&(\mathbf{case}\, a\, \mathbf{of}\, \{\mathbf{Left}(_) \to -1 : \mathsf{gtos}\ (b : t);\\
&\qquad\qquad \mathbf{Right}(_) \to 1 : \mathsf{gtos}((\mathsf{not}\ b) : t)\}),\\
&(\mathbf{case}\, b\, \mathbf{of}\, \{\mathbf{Right}(_) \to 0 : \mathsf{gtos}(a : (\mathsf{nh}\ t))\})).\\
&\qquad\qquad \mathbf{Left}(_) \to \bot\})).
\end{aligned}$$

In [43], a Gray-code to signed digit conversion program was written with the locally angelic **Amb** operator that evaluates the first two cells a and b in parallel and continues the computation based on the value obtained first. In that program, if the value of b is first obtained and it is **Left**, then it has to evaluate a again. With globally angelic choice, as the above program shows, one can simply neglect the value to use the value of the other thread. Globally angelic choice also has the possibility to speed up the computation if the two threads of **Amb** are computed in parallel and the *whole* computation based on the secondly-obtained value of **Amb** terminates first.

7 Implementation

Since our programming language can be viewed as a fragment of Haskell, we can execute the extracted program in Haskell by implementing the Amb operator with the Haskell concurrency module. We comment on the essential points of the implementation. The full code is available from [3].

First, we define the domain D as a Haskell data type:

```
data D = Nil | Le D | Ri D | Pair(D, D) | Fun(D -> D) | Amb(D, D)
```

The $\rightsquigarrow$-reduction, which preserves the Phase I denotational semantics and reduces a program to a w.h.n.f. with the leftmost outermost reduction strategy, coincides with reduction in Haskell. Thus, we can identify extracted programs with programs of type D that compute that phase.

The $\stackrel{\mathrm{c}}{\rightsquigarrow}$ reduction that concurrently calculates the arguments of **Amb** can be implemented with the Haskell concurrency module. In [19], the (locally angelic) amb operator was implemented in Glasgow Distributed Haskell (GDH). Here, we implemented it with the Haskell libraries `Control.Concurrent` and `Control.Exception` as a simple function `ambL :: [b] -> IO b` that concurrently evaluates the elements of a list and writes the result first obtained in a mutable variable.

Finally, the function `ed :: D -> IO D` produces an element of data(a) from $a \in D$ by activating `ambL` for the case of $\mathbf{Amb}(a, b)$. It corresponds to $\overset{\mathrm{p}}{\rightsquigarrow}$-reduction though it computes arguments of a pair sequentially. This function is nondeterministic since the result of executing `ed (Amb a b)` depends on which of the arguments `a,b` delivers a result first. The set of all possible results of `ed a` corresponds to the set data(a).

We executed the program extracted in Section 6 with `ed`. As we have noted, the number 0 has three Gray-codes (i.e., realizers of $\mathbf{G}(0)$): $a = \bot : 1 : (-1)^\omega$, $b = 1 : 1 : (-1)^\omega$, and $c = -1 : 1 : (-1)^\omega$. On the other hand, the set of signed digit representations of 0 is $A \cup B \cup C$ where $A = \{0^\omega\}$, $B = \{0^k : 1 : (-1)^\omega \mid k \geq 0\}$, and $C = \{0^k : (-1) : 1^\omega \mid k \geq 0\}$, i.e., $A \cup B \cup C$ is the set of realizers of $\mathbf{S}(0)$. One can calculate

$$\mathsf{gtos}(a) = \mathbf{Amb}(\bot, 0 : \mathbf{Amb}(\bot, 0 : \ldots))$$

and $\mathrm{data}(\mathsf{gtos}(a)) = A$. Thus $\mathsf{gtos}(a)$ is reduced uniquely to $0 : 0 : \ldots$ by the operational semantics. On the other hand, one can calculate $\mathrm{data}(\mathsf{gtos}(b)) = A \cup B$ and $\mathrm{data}(\mathsf{gtos}(c)) = A \cup C$. They are subsets of the set of realizers of $\mathbf{S}(0)$ as Theorem 5 says, and $\mathsf{gtos}(b)$ is reduced to an element of $A \cup B$ as Theorem 6 says.

We wrote a program that produces a $\{-1, 1, \bot\}$-sequence with the speed of computation of each digit (-1 and 1) be controlled. Then, apply it to `gtos` and then to `ed` to obtain expected results.

8 Conclusion

We introduced the logical system CFP by extending IFP [12] with two propositional operators $B|_A$ and $\Downarrow\!\Downarrow(A)$, and developed a method for extracting nondeterministic and concurrent programs that are provably total and satisfy their specifications.

While IFP already imports classical logic through nc-axioms that need only be true classically, in CFP the access to classical logic is considerably widened through the rule (Conc-lem) which, when interpreting $B|_A$ as $A \to B$ and identifying $\Downarrow\!\Downarrow(A)$ with A, is constructively invalid but has nontrivial nondeterministic computational content.

We applied our system to extract a concurrent translation from infinite Gray code to the signed digit representation, thus demonstrating that this approach not only is about program extraction 'in principle' but can be used to solve nontrivial concurrent computation problems through program extraction.

After an overview of related work, we conclude with some ideas for follow-up research.

8.1 Related work

The CSL 2016 paper [5] is an early attempt to capture concurrency via program extraction and can be seen as the starting point of our work. Our main

advances, compared to that paper, are that it is formalized as a logic for concurrent execution of partial programs by a globally angelic choice operator which is formalized by introducing a new connective $B|_A$, and that we are able to express bounded nondeterminism with complete control of the number of threads while [5] modelled nondeterminism with countably infinite branching, which is unsuitable or an overkill for most applications. Furthermore, our approach has a typing discipline, a sound and complete small-step reduction, and has the ability to switch between global and local nondeterminism (see Sect. 8.2 below).

As for the study of angelic nondeterminism, it is not easy to develop a denotational semantics as we noted in Section 2, and it has been mainly studied from the operational point of view, e.g., notions of equivalence or refinement of processes and associated proof methods, which are all fundamental for correctness and termination [28,33,27,37,16,29]. Regarding imperative languages, Hoare logic and its extensions have been applied to nondeterminism and proving totality from the very beginning ([2] is a good survey on this subject). [31] studies angelic nondeterminism with an extension of Hoare Logic.

There are many logical approaches to concurrency. An example is an approach based on extensions of Reynolds' separation logic [36] to the concurrent and higher-order setting [34,13,25]. Logics for session types and process calculi [45,15,26] form another approach that is oriented more towards the formulae-as-types/proofs-as-programs [22,44] or rather proofs-as-processes paradigm [1]. All these approaches provide highly specialized logics and expression languages that are able to model and reason about concurrent programs with a fine control of memory and access management and complex communication patterns.

8.2 Modelling locally angelic choice

We remarked earlier that our interpretation of **Amb** corresponds to *globally* angelic choice. Surprisingly, *locally* angelic choice can be modelled by a slight modification of the restriction and the total concurrency operators: We simply replace A by the logically equivalent formula $A \vee \mathbf{False}$, more precisely, we set $B|'_A \stackrel{\text{Def}}{=} (B \vee \mathbf{False})|_A$ and $\Downarrow'(A) \stackrel{\text{Def}}{=} \Downarrow(A \vee \mathbf{False})$. Then the proof rules in Sect. 4 with $|$ and $\Downarrow$ replaced by $|'$ and $\Downarrow'$, respectively but without the strictness condition, are theorems of CFP. To see that the operator $\Downarrow'$ indeed corresponds to locally angelic choice it is best to compare the realizers of the rule (Conc-mp) for $\Downarrow$ and $\Downarrow'$. Assume A, B are non-Harrop and f is a realizer of $A \to B$. Then, if $\mathbf{Amb}(a, b)$ realizes $\Downarrow(A)$, then $\mathbf{Amb}(f{\downarrow}a, f{\downarrow}b)$ realizes $\Downarrow(B)$. This means that to choose, say, the left argument of **Amb** as a result, a must terminate and so must the ambient (global) computation $f{\downarrow}a$. On the other hand, the program extracted from the proof of (Conc-mp) for $\Downarrow'$ takes a realizer $\mathbf{Amb}(a, b)$ of $\Downarrow'(A)$ and returns $\mathbf{Amb}((\mathsf{up} \circ f \circ \mathsf{down}){\downarrow}a, (\mathsf{up} \circ f \circ \mathsf{down}){\downarrow}b)$ as realizer of $\Downarrow'(B)$, where up and down are the realizers of $B \to (B \vee \mathbf{False})$ and $(A \vee \mathbf{False}) \to A$, namely, $\mathsf{up} \stackrel{\text{Def}}{=} \lambda a.\, \mathbf{Left}(a)$ and $\mathsf{down} \stackrel{\text{Def}}{=} \lambda c.\, \mathbf{case}\, c\, \mathbf{of}\, \{\mathbf{Left}(a) \to a\}$. Now, to choose the left argument of **Amb**, it is enough for a to terminate since the non-strict operation up will immediately produce a w.h.n.f. without invoking the ambient

computation. By redefining realizers of $B|_A$ and $\downdownarrows(A)$ as realizers of $B|'_A$ and $\downdownarrows'(A)$ and the realizers of the rules of CFP as those extracted from the proofs of the corresponding rules for $|'$ and $\downdownarrows'$, we have another realizability interpretation of CFP that models locally angelic choice.

8.3 Markov's principle with restriction

So far, (Rest-intro) is the only rule that derives a restriction in a non-trivial way. However, there are other such rules, for example

$$\frac{\forall x \in \mathbf{N}(P(x) \vee \neg P(x))}{\exists x \in \mathbf{N}\, P(x)|_{\exists x \in \mathbf{N}\, P(x)}} \text{ Rest-Markov}$$

If $P(x)$ is Harrop, then (Rest-Markov) is realized by minimization. More precisely, if f realizes $\forall x \in \mathbf{N}(P(x) \vee \neg P(x))$, then $\min(f)$ realizes the formula $\exists x \in \mathbf{N}\, P(x)|_{\exists x \in \mathbf{N}\, P(x)}$, where $\min(f)$ computes the least $k \in \mathbf{N}$ such that $f\, k = \mathbf{Left}$ if such k exists, and does not terminate, otherwise. One might expect as conclusion of (Rest-Markov) the formula $\exists x \in \mathbf{N}\, P(x)|_{(\neg\neg\exists x \in \mathbf{N}\, P(x))}$. However, because of (Rest-stab) (which is realized by the identity), this wouldn't make a difference. The rule (Rest-Markov) can be used, for example, to prove that Harrop predicates that are recursively enumerable (re) and have re complements are decidable. From the proof one can extract a program that concurrently searches for evidence of membership in the predicate and its complement.

8.4 Further directions for research

The undecidability of equality of real numbers, which is at the heart of our case study on infinite Gray code, is also a critical point in Gaussian elimination where one needs to find a non-zero entry in a non-singular matrix. As shown in [10], our approach makes it possible to search for such 'pivot elements' in a concurrent way. A further promising research direction is to extend the work on coinductive presentations of compact sets in [41] to the concurrent setting.

Acknowledgements This work was supported by IRSES Nr. 612638 CORCON and Nr. 294962 COMPUTAL of the European Commission, the JSPS Core-to-Core Program, A. Advanced research Networks and JSPS KAKENHI 15K00015 as well as the Marie Curie RISE project CID (H2020-MSCA-RISE-2016-731143).

References

1. Abramsky, S.: Proofs as processes. Theoretical Computer Science **135**(1), 5–9 (Apr 1992). https://doi.org/10.1016/0304-3975(94)00103-0
2. Apt, K., Olderog, E.: Fifty years of Hoare's logic. Formal Aspects of Computing **31**, 751 – 807 (2019). https://doi.org/10.1007/s00165-019-00501-3
3. Berger, U.: CFP (concurrent fixed point logic) repository, https://github.com/ujberger/cfp

4. Berger, U.: From coinductive proofs to exact real arithmetic: theory and applications. Logical Methods in Comput. Sci. **7**(1), 1–24 (2011). https://doi.org/10.2168/LMCS-7(1:8)2011
5. Berger, U.: Extracting Non-Deterministic Concurrent Programs. In: Talbot, J.M., Regnier, L. (eds.) 25th EACSL Annual Conference on Computer Science Logic (CSL 2016). Leibniz International Proceedings in Informatics (LIPIcs), vol. 62, pp. 26:1–26:21. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2016). https://doi.org/10.4230/LIPIcs.CSL.2016.26
6. Berger, U., Miyamoto, K., Schwichtenberg, H., Seisenberger, M.: Minlog - a tool for program extraction for supporting algebra and coalgebra. In: CALCO-Tools. Lecture Notes in Computer Science, vol. 6859, pp. 393–399. Springer (2011). https://doi.org/10.1007/978-3-642-22944-2_29
7. Berger, U., Petrovska, O.: Optimized program extraction for induction and coinduction. In: CiE 2018: Sailing Routes in the World of Computation. LNCS, vol. 10936, pp. 70–80. Springer Verlag, Berlin, Heidelberg, New York (2018). https://doi.org/10.1007/978-3-319-94418-0_7
8. Berger, U., Petrovska, O., Tsuiki, H.: Prawf: An interactive proof system for program extraction. In: Anselmo, M., Vedova, G., Manea, F., Pauly, A. (eds.) Beyond the Horizon of Computability - 16th Conference on Computability in Europe, CiE 2020. Lecture Notes in Computer Science, vol. 12098, pp. 137–148. Springer (2020). https://doi.org/10.1007/978-3-030-51466-2_12
9. Berger, U., Seisenberger, M.: Proofs, programs, processes. Theory of Computing Systems **51**(3), 213–329 (2012). https://doi.org/10.1007/s00224-011-9325-8
10. Berger, U., Seisenberger, M., Spreen, D., Tsuiki, H.: Concurrent Gaussian elimination. To appear (2022)
11. Berger, U., Tsuiki, H.: Extracting total amb programs from proofs (2021), https://arxiv.org/abs/2104.14669
12. Berger, U., Tsuiki, H.: Intuitionistic fixed point logic. Annals of Pure and Applied Logic **172**(3), 102903 (2021). https://doi.org/10.1016/j.apal.2020.102903
13. Brookes, S.: A semantics for concurrent separation logic. Theoretical Computer Science **375**, 227–370 (2007). https://doi.org/10.1016/j.tcs.2006.12.034
14. Broy, M.: A theory for nondeterminism, parallelism, communication, and concurrency. Theoretical Computer Science **45**, 1 – 61 (1986). https://doi.org/10.1016/0304-3975(86)90040-X
15. Caires, L., Pfenning, F., Toninho, B.: Linear logic propositions as session types. Mathematical Structures in Computer Science **26**, 367–423 (2016). https://doi.org/10.1017/S0960129514000218
16. Carayol, A., Hirschkoff, D., Sangiorgi, D.: On the representation of mccarthy's amb in the π-calculus. Theoretical Computer Science **330**(3), 439 – 473 (2005). https://doi.org/10.1016/j.tcs.2004.10.005, expressiveness in Concurrency
17. Clinger, W., Halpern, C.: Alternative semantics for McCarthy's amb. In: Brookes S.D., Roscoe A.W., W.G. (ed.) Seminar on Concurrency. CONCURRENCY 1984. Lecture Notes in Computer Science, vol. 197. Springer (1985). https://doi.org/10.1007/3-540-15670-4_22
18. Di Gianantonio, P.: An abstract data type for real numbers. Theoretical Computer Science **221**(1-2), 295–326 (1999). https://doi.org/10.1016/S0304-3975(99)00036-5
19. Du Bois, A., Pointon, R., Loidl, H.W., Trinder, P.: Implementing declarative parallel bottom-avoiding choice. In: 14th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2002), 28-30 October 2002, Vitoria, Espirito Santo, Brazil. pp. 82–92. IEEE Computer Society (2002). https://doi.org/10.1109/CAHPC.2002.1180763

20. Escardo, M.H.: PCF extended with real numbers. Theoretical Computer Science **162**, 79–115 (1996). https://doi.org/10.1016/0304-3975(95)00250-2
21. Gierz, G., Hofmann, K.H., Keimel, K., Lawson, J.D., Mislove, M., Scott, D.S.: Continuous Lattices and Domains, Encyclopedia of Mathematics and its Applications, vol. 93. Cambridge University Press (2003)
22. Howard, W.A.: The formulae-as-types notion of construction. In: Seldin, J.P., Hindley, J.R. (eds.) To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, pp. 479–490. Academic Press (1980)
23. Hughes, J., Moran, A.: A semantics for locally bottom-avoiding choice. In: Launchbury, J., Sansom, P.M. (eds.) Functional Programming, Glasgow 1992, Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland, UK, 6-8 July 1992. pp. 102–112. Workshops in Computing, Springer (1992). https://doi.org/10.1007/978-1-4471-3215-8_9
24. Hughes, J., O'Donnell, J.: Expressing and reasoning about non-deterministic functional programs. In: Davis, K., Hughes, J. (eds.) Functional Programming, Proceedings of the 1989 Glasgow Workshop, 21-23 August 1989, Fraserburgh, Scotland, UK. pp. 308–328. Workshops in Computing, Springer (1989)
25. Jung, R., Krebbers, R., Jourdan, J.H., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up. Journal of Functional Programming **28**, 1–73 (2018). https://doi.org/10.1017/S0956796818000151
26. Kouzapas, D., Nobuko, Y., Hu, R., Honda, K.: On asynchronous eventful session semantics. Mathematical Structures in Computer Science **26**, 303–364 (2016). https://doi.org/10.1017/S096012951400019X
27. Lassen, S.B.: Normal Form Simulation for McCarthy's Amb. Electronic Notes in Theoretical Computer Science **155**, 445 – 465 (2006). https://doi.org/10.1016/j.entcs.2005.11.068, proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXI)
28. Lassen, S.B., Moran, A.: Unique Fixed Point Induction for McCarthy's Amb. In: Kutylowski, M., Pacholski, L., Wierzbicki, T. (eds.) Mathematical Foundations of Computer Science 1999, 24th International Symposium, MFCS'99, Szklarska Poreba, Poland, September 6-10, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1672, pp. 198–208. Springer (1999). https://doi.org/10.1007/3-540-48340-3_18
29. Levy, P.B.: Amb breaks Well-Pointedness, Ground Amb doesn't. Electronic Notes in Theoretical Computer Science **173**, 221 – 239 (2007). https://doi.org/10.1016/j.entcs.2007.02.036, proceedings of the 23rd Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXIII)
30. Luckhardt, H.: A fundamental effect in computations on real numbers. Theoretical Computer Science **5**(3), 321–324 (1977). https://doi.org/10.1016/0304-3975(77)90048-2
31. Mamouras, K.: Synthesis of strategies and the hoare logic of angelic nondeterminism. In: Pitts, A.M. (ed.) Foundations of Software Science and Computation Structures - 18th International Conference, FoSSaCS 2015. Lecture Notes in Computer Science, vol. 9034, pp. 25–40. Springer (2015). https://doi.org/10.1007/978-3-662-46678-0_2
32. McCarthy, J.: A basis for a mathematical theory of computation. In: Braffort, P., Hirschberg, D. (eds.) Computer Programming and Formal Systems, Studies in Logic and the Foundations of Mathematics, vol. 35, pp. 33 – 70. Elsevier (1963). https://doi.org/10.1016/S0049-237X(08)72018-4

33. Moran, A., Sands, D., Carlsson, M.: Erratic fudgets: a semantic theory for an embedded coordination language. Science of Computer Programming **46**(1), 99 – 135 (2003). https://doi.org/10.1016/S0167-6423(02)00088-6, special Issue on Coordination Languages and Architectures
34. O'Hearn, P.: Resources, concurrency, and local reasoning. Theoretical Computer Science **375**(1), 271–307 (2007). https://doi.org/10.1016/j.tcs.2006.12.035
35. Pierce, B.C.: Types and Programming Languages. The MIT Press (2002)
36. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science. pp. 55–74. LICS '02, IEEE Computer Society, Washington, DC, USA (2002). https://doi.org/10.1109/LICS.2002.1029817
37. Sabel, D., Schmidt-Schauss, M.: A call-by-need lambda calculus with locally bottom-avoiding choice: context lemma and correctness of transformations. Mathematical Structures in Computer Science **18**(3), 501–553 (2008). https://doi.org/10.1017/S0960129508006774
38. Schwichtenberg, H.: Minlog. In: Wiedijk, F. (ed.) The Seventeen Provers of the World. pp. 151–157. No. 3600 in Lecture Notes in Artificial Intell. (2006). https://doi.org/10.1016/j.jlap.2004.07.005
39. Schwichtenberg, H., Wainer, S.S.: Proofs and Computations. Cambridge University Press (2012)
40. Sondergard, H., Sestoft, P.: Non-determinism in Functional Languages. The Computer Journal **35**(5), 514–523 (1992). https://doi.org/10.1093/comjnl/35.5.514
41. Spreen, D.: Computing with continuous objects: a uniform co-inductive approach. Mathematical Structures in Computer Science **31**(2), 144–192 (2021). https://doi.org/10.1017/S0960129521000116
42. Tsuiki, H.: Real number computation through Gray code embedding. Theoretical Computer Science **284**(2), 467–485 (2002). https://doi.org/10.1016/S0304-3975(01)00104-9
43. Tsuiki, H.: Real number computation with committed choice logic programming languages. J. Log. Algebr. Program. **64**(1), 61–84 (2005). https://doi.org/10.1016/j.jlap.2004.07.005
44. Wadler, P.: Propositions as sessions. Journal of Functional Programming **24**, 384–418 (2014). https://doi.org/10.1017/S095679681400001X
45. Wadler, P.: Propositions as types. Communications of the ACM **58**(12), 75–84 (2014). https://doi.org/10.1145/2699407
46. Weihrauch, K.: Computable Analysis. Springer (2000)