A realizability interpretation of Church’s simple theory of types

ULRICH BERGER and TIE HOU

Department of Computer Science, Swansea University, UK

Received 2 March 2016

We give a realizability interpretation of an intuitionistic version of Church’s Simple Theory of Types (CST) which can be viewed as a formalization of intuitionistic higher-order logic. Although definable in CST we include operators for monotone induction and coinduction and provide simple realizers for them. Realizers are formally represented in an untyped lambda-calculus with pairing and case-construct. The purpose of this interpretation is to provide a foundation for the extraction of verified programs from formal proofs as an alternative to type-theoretic systems. The advantages of our approach are that (a) induction and coinduction are not restricted to the strictly positive case, (b) abstract mathematical structures and results may be imported, (c) the formalization is technically simpler than in other systems, for example, regarding the definition of realizability, which is a simple syntactical substitution, and the treatment of nested and simultaneous (co)inductive definitions.

1. Introduction

Building rigorous reasoning systems for hardware and software verification is a very demanding endeavor. Most systems provide means for modeling and specifying hardware or software and methods for proving compliance with the specification, but there are alternatives that can be summarized under the slogan “program extraction from proofs”. The first theorem provers that supported program extraction were Nuprl (Constable et al. 1986) which is based on Martin-Löf type theory, and PX (Hayashi and Nakano 1987) a system based on Feferman’s theory of functions and classes (Feferman 1979) which provides a method of extracting untyped LISP programs from proofs of specifications. Current systems with facilities for program extraction are Coq (Letouzey 2002), Isabelle/HOL (Berghofer 2003), and Minlog (Berger et al. 2011), which are respectively based on dependent type theory, higher-order logic, and intuitionistic arithmetic in finite types. Here we propose to use a variant of Church’s simple theory of types (Church 1940) for program extraction. Church’s theory, equipped with some modifications and enhancements, has been incorporated into theorem proving systems for specifying and

† This work was partly supported by the EU FP7 project COMPUTAL and the Japan Advanced Institute of Science and Technology (JAIST)
verifying the correctness of mathematical proofs. Notable developments in this direction include HOL (Gordon 1988; Gordon and melham 1993), IMPS (Farmer et al. 1993), Isabelle (Paulson 1994), LEO-II (Benzmüller et al. 2008), PVS (Owre et al. 1996), Satallax (Brown 2012) and TPS (Andrews et al. 1990; Miller 1987).

In this article we show that Church’s type theory admits a simple and elegant realizability interpretation that includes monotone inductive and coinductive definitions. Our interpretation is more general than related interpretations in Coq (Paulin-Mohring 1989 b; Paulin-Mohring 1989 a) and Isabelle/HOL (Berghofer 2003) in that it covers unrestricted intuitionistic higher-order logic and induction is not confined to the strictly positive case. Another important difference of our work to (Berghofer 2003) and other work on higher-order realizability (for example (Van Oosten 1994)) is the fact that instead of only interpreting formulas as sets of realizers we interpret any higher-order expression as another higher-order expressions in an extended language. More precisely, we define an extension RCST of CST that has an extra base type of realizers and constants representing the constructors of an untyped programming language, and interpret CST-terms in RCST by applying a syntactical substitution to base types and constants (see Sections 2 and 4).

A realizability interpretation for monotone coinduction has been given earlier by Tatsuta (Tatsuta 1998). Minor difference of his interpretation to ours are that Tatsuta uses realizability with truth ($q$-realizability) whereas we omit the ‘truth’ component, he works in second-order logic while we use higher order logic, and in his system the programming language is part of the ‘input system’, that is, the formal system that is to be interpreted, while we keep the input and output systems apart. The major difference is that we can avoid Tatsuta’s extra condition on monotone (co)induction namely that not only the operator has to be monotone, but also its realizability interpretation. Tatsuta shows that this extra condition is necessary in his system. The reason why it is not necessary in ours is that our realizing system RCST has rules for certain forms of non-monotone inductive and coinductive definitions similar to (Mendler 1991).

Our main motivation for using higher-order logic and monotone instead of strictly positive induction/coinduction is not that applications would require this greater expressive power and generality, but the fact that this allows to express problems more naturally and less technically. In addition, the formal theory of induction and coinduction is much simpler for monotone operators than for strictly positive ones (see e.g. (Miranda-Perea 2005; Miyamoto et al. 2013)), and in our system induction and coinduction are completely dual to each other. Regarding the problem of duality or symmetry of strictly positive inductive and coinductive types and how to recover it using copatterns, see (Abel et al. 2013).

Our system is mainly targeted at the formalization of abstract mathematics in an axiomatic setting with a traditional set-theoretic semantics, which is quite different from the view in constructive type theory. For example, we will consider the real numbers as an axiomatically specified real closed field with the natural numbers as an inductively defined subset (see Example 1). In this setting our realizability interpretation allows to extract computational content from constructive proofs, despite the fact that proofs
talk about classical objects. A similar view is taken in (Leivant 1989) regarding the computational content of constructive proofs in the setting of second-order logic.

2. Interpretations of Simply Typed Lambda Calculi

In this section, we consider instances of the simply typed lambda calculus (with products) à la Church (Barendregt 1992) and introduce the notion of an interpretation from one instance to another. The realizability interpretation in Sect. 4 will be such an interpretation.

Definition 1 (Simply Typed Lambda Calculus). An instance of the simply typed lambda calculus, \( \Lambda_{\rightarrow \times} (\mathcal{B}, \mathcal{C}) \), is given by a set \( \mathcal{B} \) of base types and a set \( \mathcal{C} \) of typed constants. The set \( \text{Ty} \) of types is given by the grammar

\[
\text{Ty} \ni \rho, \sigma ::= b \mid \rho \rightarrow \sigma \mid \rho \times \sigma
\]

where \( b \) ranges over \( \mathcal{B} \). The set \( \text{Te} \) of terms is defined by the grammar

\[
\text{Te} \ni M, N ::= x \mid c \mid \lambda x : \rho. M \mid MN \mid \langle M, N \rangle \mid \pi_0(M) \mid \pi_1(M)
\]

where \( x \) ranges over a set \( \text{Var} \) of variables, and \( c \) ranges over the set \( \{ c \mid c : \rho \in \mathcal{C} \text{ for some } \rho \in \text{Ty} \} \). A typing context is a finite set \( \Gamma \) of pairs \( x : \rho \) such that any two pairs in \( \Gamma \) have different left components. We write \( \Gamma, x : \rho \) for \( \Gamma \cup \{ x : \rho \} \).

The typing relation \( \Gamma \vdash M : \rho \) is inductively defined as follows:

\[
\begin{align*}
\Gamma, x : \rho & \vdash x : \rho & \Gamma \vdash c : \rho & \text{if } c : \rho \in \mathcal{C} \\
\Gamma \vdash \lambda x : \rho. M : \rho \rightarrow \sigma & \quad \Gamma \vdash M : \rho \rightarrow \sigma & \Gamma \vdash N : \rho & \Gamma \vdash MN : \sigma \\
\Gamma \vdash (M, N) : \rho \times \sigma & \quad \Gamma \vdash M : \rho \times \sigma & \Gamma \vdash \pi_0(M) : \rho & \Gamma \vdash \pi_1(M) : \sigma
\end{align*}
\]

The set \( \text{FV}(M) \subseteq \text{Var} \) of free variables of a term \( M \) and \( \beta \)-equivalence, \( M \equiv_\beta N \), are defined as usual. For a set of variables \( X \) we define \( \text{Te}(X) := \{ M \in \text{Te} \mid \text{FV}(M) \subseteq X \} \).

When writing \( M = N \) we mean that the terms \( M \) and \( N \) are the same up to renaming of bound variables.

Let \( S_i = \Lambda_{\rightarrow \times} (\mathcal{B}_i, \mathcal{C}_i) \) \( (i \in \{ 1, 2 \}) \) be two instances of the simply typed lambda calculus. Let \( \text{Ty}_i \) and \( \text{Te}_i \) be the corresponding sets of types and terms.

We assume that for every variable \( x \) in \( S_1 \) we have fixed in a one-to-one fashion a variable \( \tilde{x} \) in \( S_2 \). More precisely, let \( \tilde{\gamma} \) be an injective mapping from \( \text{Var} \) to \( \text{Var} \), and \( \tilde{X} := \{ \tilde{x} \mid x \in X \} \) where \( X \subseteq \text{Var} \).

Definition 2. A base type substitution from \( S_1 \) to \( S_2 \), written as \( \xi : S_1 \rightarrow S_2 \), is a function \( \xi : \mathcal{B}_1 \rightarrow \text{Ty}_2 \). Any base type substitution \( \xi \) can be extended to a function \( \xi : \text{Ty}_1 \rightarrow \text{Ty}_2 \) by setting

\[
\xi(\rho \rightarrow \sigma) := \xi(\rho) \rightarrow \xi(\sigma) \quad \xi(\rho \times \sigma) := \xi(\rho) \times \xi(\sigma)
\]
We also extend base type substitutions to contexts by setting \( \xi(\Gamma) := \{ \bar{x} : \xi(\rho) \mid x : \rho \in \Gamma \} \).

**Definition 3.** A *constant substitution* from \( S_1 \) to \( S_2 \), written as \( \theta : S_1 \to S_2 \), is a function \( \theta : C_1 \to \text{Te}_2(\emptyset) \), that is, \( \theta(c) \) is a closed \( S_2 \)-term for each \( S_1 \)-constant \( c \). Any constant substitution \( \theta \) from \( S_1 \) to \( S_2 \), together with a base type substitution \( \xi \) from \( S_1 \) to \( S_2 \), determines a function \( \theta_\xi : \text{Te}_1(X) \to \text{Te}_2(\bar{X}) \) as follows.

\[
\begin{align*}
\theta_\xi(x) &:= \bar{x} \\
\theta_\xi(\lambda x : \rho.M) &:= \lambda \bar{x} : \theta_\xi(\rho) \cdot \theta_\xi(M) \\
\theta_\xi([M,N]) &:= (\theta_\xi(M), \theta_\xi(N)) \\
\theta_\xi(\pi_0(M)) &:= \pi_0(\theta_\xi(M)) \\
\theta_\xi(\pi_1(M)) &:= \pi_1(\theta_\xi(M))
\end{align*}
\]

**Definition 4 (Interpretation).** An *interpretation* of \( S_1 \) in \( S_2 \), written \( (\xi, \theta) : S_1 \to S_2 \), consists of a base type substitution \( \xi : S_1 \to S_2 \) and a constant substitution \( \theta : S_1 \to S_2 \) such that whenever \( c : \rho \in C_1 \), then \( \vdash_2 \theta(c) : \xi(\rho) \).

**Theorem 1 (Interpretation).** Assume \( (\xi, \theta) \) is an interpretation of \( S_1 \) in \( S_2 \). If \( \Gamma \vdash_1 M : \rho \), then \( \xi(\Gamma) \vdash_2 \theta_\xi(M) : \xi(\rho) \).

*Proof.* By induction on the derivation of \( \Gamma \vdash_1 M : \rho \).

**Lemma 1.** \( \theta_\xi(M[N/x]) = \theta_\xi(M)[\theta_\xi(N)/\bar{x}] \).

*Proof.* Easy induction on \( M \).

**Lemma 2.** Every interpretation \( \theta_\xi : S_1 \to S_2 \) respects \( \beta \)-equality.

*Proof.* One first shows that \( \theta_\xi \) respects \( \beta \)-redexes, that is, \( \theta_\xi((\lambda x : \rho.M)N) \equiv_\beta \theta_\xi(M[N/x]) \). The left hand side is equal to \( (\lambda x : \xi(\rho)) \cdot \theta_\xi(M) \cdot \theta_\xi(N) \). The right hand side equals \( \theta_\xi(M)[\theta_\xi(N)/\bar{x}] \), by Lemma 1. Hence both sides are \( \beta \)-equal. One can now show that \( M \equiv_\beta N \) implies \( \theta_\xi(M) \equiv_\beta \theta_\xi(N) \) by induction on \( M \) and case analysis on how \( M \equiv_\beta N \) was derived.

**3. Church’s Simple Theory of Types (CST)**

CST is based on a particular instance of the simply typed lambda calculus \( \Lambda_{\rightarrow}x(I \cup \{ o \}, C_{\text{Church}}) \) where the set of base types contains a set \( I \) of base types for individuals and the type \( o \) of propositions, and the set \( C_{\text{Church}} \) consists of the constants

\[
\begin{align*}
\land, \lor, \rightarrow : o \rightarrow o \rightarrow o \\
\equiv_\rho : o \rightarrow o \\
\forall \rho, \exists \rho : (\rho \rightarrow o) \rightarrow o \\
\mu \rho, \nu \rho : (\rho \rightarrow o) \rightarrow o
\end{align*}
\]

In \( \forall \rho, \exists \rho \), the type \( \rho \) is arbitrary while for \( \mu \rho \) and \( \nu \rho \), \( \rho \) is restricted to *predicate types* (see Def. 5 below). Note the overloading of the symbol \( \rightarrow \) which we use at the same time as the function types constructor and the constant for implication. For the sake of readability we will write \( \forall x : \rho.A \) for \( \forall \rho(\lambda x : \rho.A) \) and \( \exists x : \rho.A \) for \( \exists \rho(\lambda x : \rho.A) \). Frequently we will also write \( M \bar{x} = E \) instead of \( M := \lambda \bar{x} : \bar{\rho}.E, \) and \( M \bar{x} = E[M/p] \) instead of \( M := \mu \bar{\rho}\lambda p : \bar{\rho} \rightarrow o \lambda \bar{x} : \bar{\rho}.E, \) etc., provided the types of the variables are clear from the context.
Although the system CST will be equipped with an intuitionistic proof calculus, its intended semantics is classical, i.e. we interpret CST in the cartesian closed category of sets where base types $\iota \in I$ are interpreted as arbitrary nonempty sets and $o$ is interpreted as the two-element set of classical Boolean values. The logical symbols $\land, \lor, \rightarrow, \forall, \exists, =_o$ are interpreted in the usual classical way, $=_o$ denotes set-theoretic equality, and the constants $\mu, \nu$ are interpreted, for monotone arguments, as least and greatest fixed point operators.

In order to formally express the semantics of $\mu, \nu$ we need higher-order versions of inclusion between predicates, which can be declared meaningfully only for predicate types, that is, types that are canonically (in any ccc) isomorphic to a finite product of types of the form $\rho \rightarrow o$. The definition is as follows.

**Definition 5.** We define *predicate types* inductively as follows.

— $o$ is a predicate type.
— if $\rho$ and $\sigma$ are predicate types, then $\rho \times \sigma$ is a predicate type.
— if $\rho$ is arbitrary and $\sigma$ is a predicate type, then $\rho \rightarrow \sigma$ is a predicate type.

**Definition 6.** For every predicate type $\rho$ we define an *inclusion predicate* $\subseteq_\rho: \rho \rightarrow \rho \rightarrow o$ by recursion on $\rho$.

\[
\begin{align*}
x \subseteq_o y &= x \rightarrow y \\
x \subseteq_{\rho \times \sigma} y &= \pi_0(x) \subseteq_\rho \pi_0(y) \land \pi_1(x) \subseteq_\sigma \pi_1(y) \\
x \subseteq_{\rho \rightarrow \sigma} y &= \forall u : \rho. x u \subseteq_\sigma y u
\end{align*}
\]

We also set $x \approx_\rho y = x \subseteq_\rho y \land y \subseteq_\rho x$.

**Definition 7.** For every predicate type $\rho$ we define a *monotonicity predicate*.

\[
\begin{align*}
\mathrm{mon}_\rho : (\rho \rightarrow \rho) &\rightarrow o \\
\mathrm{mon}_\rho \Phi &= \forall x, y : \rho. x \subseteq_\rho y \rightarrow \Phi x \subseteq_\rho \Phi y
\end{align*}
\]

The deductive system given in Table 1 derives sequents of the form $\Delta \vdash \Gamma A$ where $\Delta$ is a finite set of $\Gamma$-formulas and $A$ is a $\Gamma$-formula, where by a $\Gamma$-formula we mean a term $M$ such that $\Gamma \vdash M : o$.

**Remarks.** 1. In order to enable a realizability interpretation we had to deviate from Church’s calculus in several aspects: first our system is intuitionistic while Church’s is classical. Secondly, we dropped the choice operator since it does not admit a realizability interpretation (easy exercise). There are other realizability interpretations where versions of choice are realizable, e.g. Krivine’s countable choice in classical second-order logic (Krivine 2003), Raffalli and Ruyer’s choice axiom in higher-order logic (Raffalli and Ruyer 2008), Oliva and Streicher study the connection between Krivine’s work and modified realizability (Oliva and Streicher 2008).

2. From a logical point of view, the constants $\rightarrow$ and $\forall, \exists$ would suffice to define all other constants. For the logical constants including equality this was already observed by Church (Church 1940). Furthermore, $\mu, \nu$ can be defined as the infimum of all $x : \rho$...
such that $\Phi \subseteq \rho, x$, and $\nu_\rho \Phi$ can be defined similarly. The reason why we prefer to keep these definable constants as primitives is that they can be given simpler realizers than the ones extracted from the impredicative definitions above (see, for example, (Tatsuta 1998) for the realizer extracted from the impredicative definition of $\nu_\rho$).

3. In CST, the rules for $\mu_\rho$ and $\nu_\rho$ are restricted to monotone operators. In Sect. 4, we will consider an extension of CST that has rules for $\mu_\rho \Phi$ and $\nu_\rho \Phi$ for arbitrary operators $\Phi$.

**Example 1 (Inductive natural numbers).** Consider the type context $\Gamma_+ := \{0, 1 : t, + : t \rightarrow t \rightarrow t, - : t \rightarrow t\}$ and let $\Delta_+$ consist of the formulas stating that $(0, 1, +, -)$ is an abelian group. Formally, the terms $0, 1, +, -$ are variables, but we like to view them
as constants and the assumptions $\Delta_+$ as axioms. We will write $M - N$ as a shorthand for $M + (-N)$.

We define the set of natural numbers as an inductive predicate. Set $\Phi := \lambda p : \iota \rightarrow o . \lambda x : \iota . x = :_\iota 0 \lor p(x - 1)$, and define $\mathbb{N} := \mu_{\iota \rightarrow o} \Phi$. Using the more readable notation introduced earlier the definition of $\mathbb{N}$ is

$\mathbb{N} x \overset{\mu}{=} x = :_\iota 0 \lor \mathbb{N} (x - 1)$.

Note that the definition $\mathbb{N} x \overset{\mu}{=} x = :_\iota 0 \lor \exists y : \iota . \mathbb{N} y \land x = :_\iota y + 1$ would be equivalent under the assumptions $\Delta_+$. As an example of a proof by induction we show that the natural numbers are closed under addition:

$\Gamma \vdash \forall x,y : \iota . \mathbb{N} x \land \mathbb{N} y \rightarrow \mathbb{N} (x + y)$

Setting $P[x] := \lambda y : \iota . \mathbb{N}(x + y)$, the formula to be proven is equivalent to $\forall x : \iota . \mathbb{N} x \rightarrow \mathbb{N} \subseteq_{\iota \rightarrow o} P[x]$. Hence, it suffices to show $\Phi P[x] \subseteq_{\iota \rightarrow o} P[x]$ under the extra assumption $\Delta_+$.

Unfolding the definition of $\Phi$ and using proof by cases ($\lor$), this amounts to proving $\mathbb{N}(x + 0)$ and $\forall y : \iota . \mathbb{N}(x + (y - 1)) \rightarrow \mathbb{N}(x + y)$, which is easy, given the assumptions $\Delta_+$ and $\mathbb{N} x$.

**Example 2 (Coinductive Fibonacci numbers).** Continuing the previous example, we define a coinductive predicate $\text{FIB} := \nu_{\iota \rightarrow \iota \rightarrow o} \Psi$ where

$\Psi := \lambda q : \iota \rightarrow \iota \rightarrow o . \lambda x,y : \iota . \mathbb{N} x \land q y (x + y)$

or, using the more readable notation,

$\text{FIB} xy \overset{\nu}{=} \mathbb{N} x \land \text{FIB} y (x + y)$

Informally, $\text{FIB} x y$ states that the Fibonacci sequence starting with $x,y$ (i.e. $x_0, x_1, x_2, \ldots$ where $x_0 = x$, $x_1 = y$, and $x_{n+2} = x_n + x_{n+1}$) consists entirely of natural numbers. As an example of a proof that uses coinduction we show

$\Delta_+ \vdash_{\Gamma_+} \text{FIB} 1 1$.

We show more generally, that $\text{FIB}$ holds for any two natural numbers, i.e. $Q \subseteq_{\iota \rightarrow \iota \rightarrow o} \text{FIB}$ where $Q := \lambda x,y : \iota . \mathbb{N} x \land \mathbb{N} y$. By coinduction, it suffices to show $Q \subseteq_{\iota \rightarrow \iota \rightarrow o} \Psi Q$, which is easily done using the previously proven fact that natural numbers are closed under addition.

We will discuss the realizability interpretations of these examples in Sect. 4 and their extracted programs in Sect. 6.

### 4. Realizability Interpretation

We extend CST to RCST by adding an extra base type $\delta$ for realizers and extra constants

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nil</td>
<td>$\delta$</td>
<td>Realizer</td>
</tr>
<tr>
<td>inL</td>
<td>$\delta \rightarrow \delta$</td>
<td>Inl</td>
</tr>
<tr>
<td>inR</td>
<td>$\delta \rightarrow \delta$</td>
<td>Inr</td>
</tr>
<tr>
<td>prL</td>
<td>$\delta \rightarrow \delta$</td>
<td>ProjL</td>
</tr>
<tr>
<td>prR</td>
<td>$\delta \rightarrow \delta$</td>
<td>ProjR</td>
</tr>
<tr>
<td>fun</td>
<td>$(\delta \rightarrow \delta) \rightarrow \delta$</td>
<td>Function</td>
</tr>
<tr>
<td>case</td>
<td>$(\delta \rightarrow \delta) \rightarrow (\delta \rightarrow \delta) \rightarrow \delta$</td>
<td>Case</td>
</tr>
<tr>
<td>rec</td>
<td>$(\delta \rightarrow \delta) \rightarrow \delta$</td>
<td>Recursion</td>
</tr>
</tbody>
</table>

which we call *program constants*. We also extend the ranges of the parameters of the constants $\forall \rho$, $\exists \rho$, $\equiv \rho$, and $\mu \rho$, $\nu \rho$ to all types respectively predicate types $\rho$ of RCST.
We define $\equiv_\delta$ as the least congruence relation on terms such that
\[
\text{case } (\text{in}_L \ M \ M_1 \ M_2) \equiv_\delta \ M_1 \ M_2 \\
\text{pr}_L \ (\text{pair} \ M_1 \ M_2) \equiv_\delta \ M_1 \\
\text{app} \ (\text{fun} \ M) \ N \equiv_\delta \ M \ N \\
\text{pr}_R \ (\text{pair} \ M_1 \ M_2) \equiv_\delta \ M_2
\]
and define $\equiv$ as the least congruence relation containing $\equiv_\delta$ and $\equiv_\delta$. For better readability we use the notations
\[
\lambda_\delta x. \ M := \text{fun } (\lambda x : \delta. \ M) \\
M \cdot N := \text{app } M \ N
\]
Hence $(\lambda_\delta x. M) \cdot N \equiv M[N/x]$.

Remark. The new constants amount to modeling a type free lambda-calculus with pairing and definition by cases as terms of type $\delta$. The constant rec for general recursion is added for convenience and could be defined by the familiar fixed point combinator $Y := \lambda_\delta f. (\lambda x_\delta. f \cdot (x \cdot x)). (\lambda x_\delta. f \cdot (x \cdot x))$. The new constants have a straightforward interpretation in a domain-theoretic model which will be discussed in Sect. 6.

The logical rules of RCST are the same as those of CST (with $\rho$ ranging over RCST types of course), except for the rule $\beta$ and the rules for $\mu_\rho$ and $\nu_\rho$. The $\beta$-rule is replaced by the stronger rule
\[
\frac{\Delta \vdash_\Gamma A \quad \Gamma \vdash_\Delta B : \rho \quad \beta_\delta \ A \equiv B}{\Delta \vdash_\Gamma B : \rho}
\]
The rules for monotone induction, Cl$\rho$ and Ind$\rho$, are replaced by rules expressing that $\mu_\rho \Phi$ is the least fixed point of the monotone operator $\Phi^\cup : \rho \to \rho$ defined by
\[
\Phi^\cup X = \bigcup_{Y \subseteq_\rho X} \Phi Y
\]
where $\bigcup_{Y \subseteq_\rho X} \Phi Y := \bigcup_{\rho, \sigma} (\lambda Y : \rho. Y \subseteq_\rho X) \Phi$ and the big union $\bigcup_{\rho, \sigma} (\rho \to o) \to (\rho \to \sigma) \to \sigma$ is defined for arbitrary types $\rho$ and predicate types $\sigma$ by
\[
\bigcup_{\rho, o} f g = \exists x : \rho. f x \land g x \\
\bigcup_{\rho, \tau \to \sigma} f g = \lambda y : \tau. \bigcup_{\rho, \sigma} f (g y) \\
\bigcup_{\rho, \rho_0 \times \sigma_1} f g = \langle \bigcup_{\rho, \rho_0} f (\lambda x : \rho. g \pi_0(x)), \bigcup_{\rho, \sigma_1} f (\lambda x : \rho. g \pi_1(x)) \rangle
\]
It is easy to see that $\bigcup_{\rho, \sigma} f g$ is the least upper bound (w.r.t. $\subseteq_\sigma$) of the set $\{g x \mid x : \rho, f x = \top\}$ (where truth, $\top$, can be defined as $\nu_{o \to o} (\lambda x : o. x)$).

Dually, the rules for coinduction are replaced by rules expressing that $\nu_\rho \Phi$ is the greatest fixed point of the monotone operator $\Phi^\cap : \rho \to \rho$ defined by
\[
\Phi^\cap X := \bigcap_{Y \supseteq_\rho X} \Phi Y
\]
which can be defined in a similar way (replacing ∪, ∃, ∧ by ∩, ∀, → in the definition above). It can be easily seen that if Φ is monotone, then Φ, Φ⊔, and Φ∩ are equivalent, i.e. Φ ≈_{ρ→ρ} Φ⊔ ≈_{ρ→ρ} Φ∩ (see Def. 6). The definitions of the operators Φ⊔ and Φ∩ are for motivation only, they are not used in the rules below. Instead, μρ and νρ (acting on arbitrary operators) are taken as primitive because this leads to simpler realizers. It can be easily shown that being the least fixed point of Φ⊔ amounts to being the smallest X such that ∀Y : ρ. Y ⊆ρ X → Φ Y ⊆ρ X, and, dually, being the greatest fixed point of Φ∩ amounts to being the greatest X such that ∀Y : ρ. X ⊆ρ Y → X ⊆ρ Φ Y. This is expressed by the following rules.

\[
\Delta ⊢ Γ \vdash M \subseteq ρ \mu_ρ(Φ) \quad \text{Cig}_ρ
\]
\[
\Delta ⊢ Γ \vdash M \subseteq ρ \nu_ρ(Φ) \subseteq M \quad \text{Coi}_ρ \quad \text{Y fresh}
\]

Remark. The rules for general induction above are similar to “Mendler style” induction (Mendler 1991). While Mendler defines reductions for his calculus and proves them to be strongly normalizing, we will use general induction (and coinduction) to prove the soundness for our realizability interpretation of monotone induction and coinduction (Theorem 4). Mendler style induction and coinduction is also used and proved strongly normalizing in (Miranda-Perea and Gonzalez-Huesca 2012) in the context of intuitionistic second-order logic and in (Geuvers 1992) and (Abel et al. 2005) in extensions of system F respectively Fω.

**Definition 8 (Realizability).** We define an interpretation \( (r, R) \) of CST in RCST with the base type substitution

\[ r(α) := δ → α \quad r(τ) := τ \]

and the constant substitution

\[ R(→) := \lambda \tilde{A}, \tilde{B} : δ → o.\lambda d : δ. \forall a : δ. \tilde{A} a → \tilde{B}(d \cdot a) \]
\[ R(∧) := \lambda \tilde{A}, \tilde{B} : δ → o.\lambda d : δ. \exists a, b : δ. d =_δ a ∧ \tilde{A} a ∧ \tilde{B} b \]
\[ R(∨) := \lambda \tilde{A}, \tilde{B} : δ → o.\lambda d : δ. \exists a : δ. (d =_δ \text{in}_L a ∧ \tilde{A} a) ∨ (d =_δ \text{in}_R a ∧ \tilde{B} a) \]
\[ R(∀) := \lambda \tilde{A} : r(ρ) → δ → o.\lambda d : δ. \forall x : r(ρ). \tilde{A} x d \]
\[ R(∃) := \lambda \tilde{A} : r(ρ) → δ → o.\lambda d : δ. \exists x : r(ρ). \tilde{A} x d \]
\[ R(=_{ρ}) := \lambda \tilde{A}, \tilde{B} : r(ρ), λd : δ. \tilde{A} =_{r(ρ)} \tilde{B} \]
\[ R(μ_ρ) := μ r(ρ) \]
\[ R(ν_ρ) := ν r(ρ) \]

According to Def. 3, the type substitution \( r \) and the constant substitution \( R \) induce a mapping \( R_r \) from CST terms to RCST terms. For notational simplicity we denote this mapping again by \( R \).
Remark. The definition of $R(c)$ becomes more comprehensible when applying it to arguments so that a formula, i.e., a term of type $\sigma$, of the form $(\sigma \to \sigma)$ is obtained (note that if $M$ is a formula, then $R(M) d$ is usually written $d \in R(M)$):

\[
R(A \to B) d \equiv \forall x : \sigma. R(A) a \to R(B)(d \cdot a)
\]

\[
R(A \land B) d \equiv \exists a, b : \delta. d = \delta \land a b \land R(A) a \land R(B) b
\]

\[
R(A \lor B) d \equiv (\exists a : \delta. d = \delta \lor a \land R(A) a) \lor (\exists b : \delta. d = \delta \lor b \land R(B) b)
\]

\[
R(\forall x : \rho. A) d \equiv \forall \vec{x} : r(\rho). R(A) d
\]

\[
R(\exists x : \rho. A) d \equiv \exists \vec{x} : r(\rho). R(A) d
\]

\[
R(A \equiv \rho. B) d \equiv R(A) = r(\rho) R(B)
\]

\[
R(\mu_\rho \Phi \vec{x}) d \equiv \mu_{r(\rho)}(\Phi) \vec{x} d \quad \text{if } \rho = \sigma \to o
\]

\[
R(\nu_\rho \Phi \vec{x}) d \equiv \nu_{r(\rho)}(\Phi) \vec{x} d \quad \text{if } \rho = \sigma \to o
\]

However, note that $R(M)$ is defined for any term, not only for formulas. Moreover, in the last two equations the type $\rho = \sigma \to o$ is only a special case. For example, types $\rho$ of the form $(\sigma \to o) \times (\tau \to o)$ are possible as well. They correspond to simultaneous (co)inductive definitions. In these cases realizability cannot be described by defining a relation between formulas and realizers.

An important special case is the realizability of the quantifiers $\forall_\rho$ and $\exists_\rho$ where the type $\rho$ is an object type, i.e., does not contain the type $o$ of propositions. In that case $r(\rho) = \rho$ and in the realizability interpretation of $\forall x : \rho. A$ and $\exists x : \rho. A$ it is safe to assume that $\vec{x} = x$. Hence

\[
R(\forall x : \rho. A) d \equiv \forall x : \rho. R(A) d
\]

\[
R(\exists x : \rho. A) d \equiv \exists x : r(\rho). R(A) d
\]

This means that objects $x$ of type $\rho$ are treated uniformly, i.e., realizers do not depend on them. As a consequence, the realizability interpretation is not restricted to types that have computationally meaningful elements (such as natural numbers, trees, or higher types above them), but base types with an arbitrary set-theoretic interpretation may be admitted, for example a base type for some abstract (not necessarily computable) metric space. Combined with the (easy to prove) fact that for a large class of "non-computational" formulas $A$ (see Sect. 6, Def. 13) the formula $R(A) \text{nil}'$ (for some trivial term nil', see Sect. 6) is equivalent to $A$ this means that the non-computational fragment of any mathematical theory can be imported (see Theorems. 5 and 6).

**Example 3 (Realizers of natural numbers).** The realizability interpretation of natural numbers defined in Example 1 is $R(\mathbb{N}) = \mu_{A\to\delta\to o} R(\Phi)$ where

\[
R(\Phi) := \lambda \bar{p} : \iota \to \delta \to o. \lambda x : \iota. \lambda d : \delta. \\
\begin{equation}
\left( d =_\delta \text{in}_L \text{nil} \land x = 0 \right) \lor \left( \exists b : \delta. d =_\delta \text{in}_R b \land \bar{p} (x - 1) b \right)
\end{equation}
\]

In more readable notation

\[
R(\mathbb{N}) x d \overset{\mu_{A\to\delta\to o}}{=} \left( d =_\delta \text{in}_L \text{nil} \land x = 0 \right) \lor \left( \exists b : \delta. d =_\delta \text{in}_R b \land R(\mathbb{N}) (x - 1) b \right)
\]
Hence, a natural number \( n : \mathbb{N} \) is realized by the numeral \( n : \delta \), where \( \mathbb{N} := \text{in}_L \text{nil} \), \( \mathbb{N} + 1 := \text{in}_R \text{nil} \). An element \( d : \delta \) realizes closure of natural numbers under addition, i.e. the formula \( \forall x, y : \mathbb{N} \cdot N x \land N y \rightarrow N (x + y) \), if for all \( x, y : \mathbb{N} \) and all \( a, b : \delta \)

\[
R(N) \cdot x a \land R(N) \cdot y b \rightarrow R(N) \cdot (x + y) (d \cdot (\text{pair } a b)).
\]

which says that \( d \) adds natural numbers in unary notation.

**Example 4 (Realizability for Fibonacci numbers).**

The realizability interpretation of the Fibonacci predicate \( \text{FIB} \) defined in Example 2 is

\[
R(\text{FIB}) = \nu \tau \rightarrow \delta \rightarrow \sigma (\text{RCST})
\]

i.e.

\[
R(\text{FIB}) x y d \equiv \exists a, b : \delta \cdot d =_\delta \text{pair } a b \land R(N) x a \land R(\text{FIB}) y (x + y) b
\]

which says that \( d \) is a stream of natural numbers in unary notation where the head realizes \( N x \) and the tail realizes \( \text{FIB} y (x + y) \).

**Definition 9 (Realizability for sequents).** The intended interpretation of a sequent \( \Delta \vdash \Gamma A \), where \( \Delta = \{B_1, \ldots, B_n\} \) and \( \Gamma = \{x_1 : \rho_1, \ldots, x_k : \rho_k\} \), is the formula \( \forall x_1 : \rho_1 \ldots \forall x_k : \rho_k \cdot B_1 \land B_2 \land \ldots \land B_n \rightarrow A \). Hence we define

\[
R(\Delta \vdash \Gamma A) := R(\forall x_1 : \rho_1 \ldots \forall x_k : \rho_k \cdot B_1 \land B_2 \land \ldots \land B_n \rightarrow A)
\]

i.e. \( R(\Delta \vdash \Gamma A) a \equiv _\beta \forall \bar{x}_1 : r(\rho_1) \ldots \forall \bar{x}_k : r(\rho_k) \cdot \forall b : \delta \cdot R(B_1 \land \ldots \land B_n) b \rightarrow R(A)(a \cdot b) \)

Note that the statement that a term \( M \) realizes a sequent \( \Delta \vdash \Gamma A \) is equivalent to the derivability of \( R(\Delta) \vdash r(\Gamma) \vdash R(A)M \) and \( \Gamma' \vdash M : \delta \), where \( \Gamma' := \{a_1 : \delta, \ldots, a_n : \delta\} \), \( r(\Gamma) := \{\bar{x}_1 : r(\rho_1), \ldots, \bar{x}_k : r(\rho_k)\} \), and \( R(\Delta) := \{R(B_i) b_i \mid i \in \{1, \ldots, n\}\} \) with fresh variables \( b_i \).

The following Soundness Theorem for the realizability interpretation states that from a proof of a formula one can extract a program provably realizing it.

**Theorem 2 (Soundness).** If \( \Delta \vdash \Gamma A \) is provable in CST, then there exists a closed term \( M \) of type \( \rho \) such that \( R(\Delta \vdash \Gamma A) M \) is provable in RCST.

**Proof.** By induction on the derivation of \( \Delta \vdash \Gamma A \). It suffices to show that each rule is realizable, i.e., for each rule

\[
\frac{\Delta_1 \vdash \Gamma_1 A_1 \quad \ldots \quad \Delta_n \vdash \Gamma_n A_n}{\Delta \vdash \Gamma A}
\]

to show that there is an object \( e : \delta \) such that for all \( a_1, \ldots, a_n : \delta \) realizing the premises, \( e \cdot a_1 \ldots a_n \) realizes the conclusion.

Note that if the proof contexts \( \Delta \) and \( \Delta_i \), and the typing contexts \( \Gamma \) and \( \Gamma_i \) are all the same respectively, it suffices to find a realizer of the universal closure of the formula \( A_1 \land \ldots \land A_n \rightarrow A \).

The logical rules are easy and we skip them. The rules for equality have very simple
realizers. One easily checks that \((=^+)\) is realized by \(\lambda a, b : \delta \cdot \text{nil}, (=^-)\) is realized by 
\(\lambda a, b : \delta \cdot b, (\text{ext})\) is realized by \(\lambda a, b : \delta \cdot \text{nil}\). Finally, rule \((\beta)\) is realized by \(\lambda a : \delta \cdot a\), due to Lemma 2. Realizability of the rules for induction and coinduction will be proven separately in Section 5.

5. Realizability for Induction and Coinduction

The proof of soundness for induction and coinduction hinges on the fact that realizability of an inclusion can be expressed by an inclusion. To this end we introduce image and inverse-image operations. The realizability interpretation of higher type inclusion \(\subseteq_\rho\) can then be expressed as a composition of \(\subseteq_{\tau(\rho)}\) and these image and inverse-image operations.

**Definition 10.** For every predicate type \(\rho\) we define terms \(\text{Im}_{\rho}\), \(\text{Im}^-_{\rho} : \delta \rightarrow r(\rho) \rightarrow r(\rho)\) as follows.

\[
\begin{align*}
\text{Im}_\omega & := \lambda d : \delta \lambda p : \delta \rightarrow o \lambda b : \delta : \exists a : \delta \cdot p a \land d \cdot a =_\delta b \\
\text{Im}_{\rho \rightarrow \sigma} & := \lambda d : \delta \lambda p : r(\rho) \rightarrow r(\sigma) \lambda x : r(\rho). \text{Im}_\sigma d(p \ x) \\
\text{Im}_{\rho \times \sigma} & := \lambda d : \delta \lambda p : r(\rho) \times r(\sigma). (\text{Im}_\rho (pr_L d) \pi_0(p), \text{Im}_\sigma (pr_R d) \pi_1(p)) \\
\text{Im}_\neg & := \lambda d : \delta \lambda p : \delta \rightarrow o \lambda a : \delta \cdot p \ (d \cdot a) \\
\text{Im}^-_{\rho \rightarrow \sigma} & := \lambda d : \delta \lambda p : r(\rho) \rightarrow r(\sigma) \lambda x : r(\rho). \text{Im}^-_\sigma d(p \ x) \\
\text{Im}^-_{\rho \times \sigma} & := \lambda d : \delta \lambda p : r(\rho) \times r(\sigma). (\text{Im}^-_\rho (pr_L d) \pi_0(p), \text{Im}^-_\sigma (pr_R d) \pi_1(p))
\end{align*}
\]

**Lemma 3.** For every predicate type \(\rho\), RCST proves that the following are equivalent for all \(\tilde{A}, \tilde{B} : r(\rho)\) and \(d : \delta\)

\[
(i) \ R(A \subseteq_\rho B) \ d \quad (ii) \ \tilde{A} \subseteq_{\tau(\rho)} \text{Im}^-_{\rho} \ d \tilde{B} \quad (iii) \ \text{Im}_\rho \ d \tilde{A} \subseteq_{\tau(\rho)} \tilde{B}
\]

**Proof.** Easy induction on \(\rho\). For example, the formulas \(R(A \subseteq_\rho B) d\) and \(\tilde{A} \subseteq_{\tau(\rho)} \text{Im}^-_{\rho} \ d \tilde{B}\) are both \(\beta\)-equal (and hence equivalent) to \(\forall a : \delta. \tilde{A} a \rightarrow \tilde{B} (d \cdot a)\). The formula \(\text{Im}_\rho \ d \tilde{A} \subseteq_{\tau(\rho)} \tilde{B}\) is \(\beta\)-equal to \(\forall b : \delta. (\exists a : \delta. \tilde{A} a \land d \cdot a =_\delta b) \rightarrow \tilde{B} b\) which is again equivalent to \(\forall a : \delta. \tilde{A} a \rightarrow \tilde{B} (d \cdot a)\).

**Lemma 4.** For every predicate type \(\rho\), RCST proves that \(\text{Im}_\rho \ d\) and \(\text{Im}^-_{\rho} \ d\) are monotone for all \(d : \delta\), i.e., for all \(\tilde{A}, \tilde{B} : r(\rho)\),

\[
(a) \ \tilde{A} \subseteq_{\tau(\rho)} \tilde{B} \rightarrow \text{Im}_\rho \ d \tilde{A} \subseteq_{\tau(\rho)} \text{Im}_\rho \ d \tilde{B}, \\
(b) \ \tilde{A} \subseteq_{\tau(\rho)} \tilde{B} \rightarrow \text{Im}^-_{\rho} \ d \tilde{A} \subseteq_{\tau(\rho)} \text{Im}^-_{\rho} \ d \tilde{B}.
\]

**Proof.** Easy induction on \(\rho\).

**Definition 11.** For every predicate type \(\rho\) we define a closed term \(\text{id}_\rho\) of type \(\delta\).

\[
\begin{align*}
\text{id}_\delta & := \lambda \delta \cdot a \cdot a \\
\text{id}_{\rho \rightarrow \sigma} & := \text{id}_\sigma \\
\text{id}_{\rho \times \sigma} & := \text{pair} \text{id}_\rho \text{id}_\sigma
\end{align*}
\]

**Lemma 5.** For every predicate type \(\rho\), RCST proves that \(\text{Im}^-_{\rho} \text{id}_\rho p \equiv_{\tau(\rho)} p\) for all \(p : r(\rho)\).
Proof. By induction on $\rho$.

\[
\text{Im}_\rho \text{id}_\rho p = (r_\rho) \lambda a : \delta. p (\text{id}_\rho \cdot a) \quad \text{(by Def. 10)}
\]
\[
= (r_\rho) \lambda a : \delta. pa \quad \text{(by Def. 11 and } \equiv_\beta\text{)}
\]
\[
= (r_\rho) p \quad \text{(by extensionality)}
\]

\[
\text{Im}_{\rho \to \sigma} \text{id}_{\rho \to \sigma} p = (r_{\rho \to \sigma}) \lambda x : r(\rho). \text{Im}_\sigma \text{id}_{\rho \to \sigma} (px) \quad \text{(by Def. 10)}
\]
\[
= (r_{\rho \to \sigma}) \lambda x : r(\rho). \text{Im}_\sigma (px) \quad \text{(by Def. 11)}
\]
\[
= (r_{\rho \to \sigma}) \lambda x : r(\rho). px \quad \text{(By I.H.)}
\]
\[
= (r_{\rho \to \sigma}) p \quad \text{(by extensionality)}
\]

\[
\text{Im}_{\rho \times \sigma} \text{id}_{\rho \times \sigma} p = (r_{\rho \times \sigma} \langle \text{Im}_\rho (\text{pr}_L \text{id}_{\rho \times \sigma}) \pi_0 (p), \text{Im}_\sigma (\text{pr}_R \text{id}_{\rho \times \sigma}) \pi_1 (p) \rangle) \quad \text{(by Def. 10)}
\]
\[
= (r_{\rho \times \sigma} \langle \text{Im}_\rho (\pi_0 (p)), \text{Im}_\sigma (\pi_1 (p)) \rangle) \quad \text{(by Def. 11 and } \equiv_\beta\text{)}
\]
\[
= (r_{\rho \times \sigma} \langle \pi_0 (p), \pi_1 (p) \rangle) \quad \text{(By I.H.)}
\]
\[
= (r_{\rho \times \sigma}) p \quad \text{(by extensionality)}
\]

Lemma 6. If $\Gamma \vdash M, N : \rho$, then RCST proves

\[
\vdash_{r(\Gamma)} R(M \subseteq_\rho N) \text{id}_\rho \leftrightarrow R(M) \subseteq_{r(\rho)} R(N).
\]

Proof. We argue in RCST. By Lemmas 3 and 5, we have

\[
\vdash_{r(\Gamma)} \forall \tilde{A}, \tilde{B} : r(\rho). R(A \subseteq_\rho B) \text{id}_\rho \leftrightarrow \tilde{A} \subseteq_{r(\rho)} \tilde{B}.
\]

Since $\Gamma \vdash M, N : \rho$ it follows $r(\Gamma) \vdash R(M), R(N) : r(\rho)$, by Theorem 1. Hence

\[
\vdash_{r(\Gamma)} R(A \subseteq_\rho B)[R(M) / \tilde{A}][R(N) / \tilde{B}] \text{id}_\rho \leftrightarrow R(M) \subseteq_{r(\rho)} R(N),
\]

which, by Lemma 1, is the same as the sequent we have to prove. \hfill \Box

The following lemma is a part of the Soundness Theorem for induction and coinduction. It shows that the closure and coclosure rules are realizable.

Theorem 3. Closure and Coclosure at type $\rho$ are realized by $\text{id}_\rho$. More precisely, if $\Gamma \vdash \Phi : \rho \rightarrow \rho$ where $\rho$ is a predicate type, then RCST proves

(a) $\vdash_{r(\Gamma)} R(\Phi(\mu_\rho \Phi)) \subseteq_{r(\rho)} \mu_\rho \Phi \text{id}_\rho$

(b) $\vdash_{r(\Gamma)} R(\nu_\rho \Phi \subseteq_{r(\rho)} \Phi(\nu_\rho \Phi)) \text{id}_\rho$

Proof. Assume $\Gamma \vdash \Phi : \rho \rightarrow \rho$, where $\rho$ is a predicate type. Clearly $r(\rho)$ is a predicate type as well. Furthermore, by Theorem 1, $r(\Gamma) \vdash R(\Phi) : r(\rho) \rightarrow r(\rho)$. \hfill \Box
(a) By Lemma 6, it suffices to show \( R(\Phi)(\mu_{\rho}(\Phi)) \subseteq_{\rho} \mu_{\rho}(\Phi) R(\Phi) \), which follows by applying the closure rule \( Cl_{e(\rho)} \) to \( R(\Phi) \).

(b) Similar to (a),

\[ \]

\textbf{Definition 12.} We define composition operations \( o_\rho : \delta \rightarrow \delta \rightarrow \delta \) for every predicate type \( \rho \).

\[ o_\rho := \lambda d, e : \delta . \lambda a . d \cdot (e \cdot a) \]

\[ o_{\rho \rightarrow \sigma} := o_\sigma \]

\[ o_{\rho \times \sigma} := \lambda d, e : \delta . \text{pair}((pr_L d) o_\rho (pr_L e), (pr_R d) o_\sigma (pr_R e)) \]

\textbf{Lemma 7.} For every predicate type \( \rho \), RCST proves that for all \( e : \delta \) and \( x : r(\rho) \),

(a) \( Im_\rho d(Im_\rho e x) \approx_{\rho \rightarrow \rho} Im_\rho (d o_\rho e) x \)

(b) \( Im_\rho d(Im_\rho e x) =_{\rho \rightarrow \rho} Im_\rho (d o_\rho e) d x \)

\textit{Proof.} By induction on \( \rho \). We only give the proofs for \( o \) and \( \rho \rightarrow \sigma \), leaving the case \( \rho \times \sigma \) to the reader.

\[ \]

\[ Im_\rho d(Im_\rho e x) \]

\[ =_{\rho \rightarrow \rho} \lambda e : \delta . \exists b : \delta . (Im_\rho e x) b \cdot d \cdot b =_{\delta} c \quad \text{(by Def. 10)} \]

\[ =_{\rho \rightarrow \rho} \lambda e : \delta . \exists b : \delta . \exists a : \delta . x a \land e \cdot a =_{\delta} b \cdot d \cdot b =_{\delta} c \quad \text{(by Def. 10)} \]

\[ \approx_{\rho \rightarrow \rho} \lambda e : \delta . \exists a : \delta . x a \land (\lambda a . d \cdot (e \cdot a)) \cdot a =_{\delta} c \quad \text{(By \( \equiv_{\delta} \))} \]

\[ =_{\rho \rightarrow \rho} \lambda e : \delta . (d o_\rho e) x \quad \text{(by Def. 12)} \]

\[ Im_\rho d(Im_\rho e x) \]

\[ =_{\rho \rightarrow \rho} \lambda a : \delta . x \cdot (e \cdot (d \cdot a)) \quad \text{(by Def. 10)} \]

\[ =_{\rho \rightarrow \rho} \lambda a : \delta . x \cdot (\lambda a . e \cdot (d \cdot a)) \cdot a \quad \text{(By \( \equiv_{\delta} \) and \( \equiv_{\rho} \))} \]

\[ =_{\rho \rightarrow \rho} \lambda a : \delta . x \cdot (e o_\rho d) \cdot a \quad \text{(by Def. 12)} \]

\[ =_{\rho \rightarrow \rho} Im_\rho (e o_\rho d) x \quad \text{(by Def. 10)} \]

\[ Im_\rho d(Im_\rho e x) \]

\[ =_{\rho \rightarrow \rho} \lambda y : r(\rho). Im_\rho d(Im_\rho e (x y)) \quad \text{(by Def. 10)} \]

\[ \approx_{\rho \rightarrow \rho} \lambda y : r(\rho). Im_\rho (d o_\rho e) (x y) \quad \text{(By I.H.)} \]

\[ =_{\rho \rightarrow \rho} \lambda y : r(\rho). Im_\rho (d o_{\rho \rightarrow \sigma} e) (x y) \quad \text{(by Def. 12)} \]

\[ =_{\rho \rightarrow \rho} Im_\rho (d o_{\rho \rightarrow \sigma} e) x \quad \text{(by Def. 10)} \]

\[ Im_\rho d(Im_\rho e x) \]

\[ =_{\rho \rightarrow \rho} \lambda y : r(\rho). Im_\rho d(Im_\rho e (x y)) \quad \text{(by Def. 10)} \]
Realizability interpretation of CST

\[ \forall \tilde{\Phi}: \mathbf{r}(\rho) \rightarrow \mathbf{r}(\rho) \forall m: \delta . \]

Theorem 4. Monotone induction and coinduction are realized by the terms

\[ \mathbf{I}_\rho := \lambda m, s: \delta . \text{rec}(\lambda f: \delta . s \circ f (m \cdot f)) \]
\[ \mathbf{I}_\rho^{co} := \lambda m, s: \delta . \text{rec}(\lambda f: \delta . (m \cdot f) \circ o_s) \]
respectively, for every predicate type \( \rho \).

Proof. For induction it suffices to show the formula

\[ \forall \tilde{\Phi}: \mathbf{r}(\rho) \rightarrow \mathbf{r}(\rho) \forall m: \delta . \]

Letting \( f := \text{rec}(\lambda f: \delta . s \circ f (m \cdot f)) \), we have to show \( \mathbf{R}(\mu_{\rho}\Phi \subseteq \rho) A f \), which, by Lemma 3, is equivalent to \( \mu_{\rho}\Phi \subseteq \rho) \mathbf{I}_\rho f \tilde{A} \). We will use the general induction rule \( \text{IndG} \) to prove this. So we assume

\[ \forall \tilde{\Phi}: \mathbf{r}(\rho) \rightarrow \mathbf{r}(\rho) \forall m: \delta . \]

and have to show \( \tilde{\Phi} \tilde{Z} \subseteq \mathbf{I}_\rho f \tilde{A} \). From the assumptions (c) and (a) we have \( \tilde{\Phi} \tilde{Z} \subseteq \mathbf{I}_\rho f \tilde{A} \). By assumption (b) and Lemma 4 (b) we get \( \mathbf{I}_\rho f \tilde{A} \). By Lemma 7 (b) and the fact that \( f = \delta \circ o_s (m \cdot f) \) we obtain

\[ \mathbf{I}_\rho f \tilde{A} \]

Applying transitivity of \( \subseteq \) yields \( \tilde{\Phi} \tilde{Z} \subseteq \mathbf{I}_\rho f \tilde{A} \), as required.

The proof for coinduction can be obtained by dualization: replace \( \mathbf{I}_\rho \) by \( \mathbf{I}_\rho^{co} \) (hence define \( f := \text{rec}(\lambda f: \delta . (m \cdot f) \circ o_s) \), \( \mathbf{I}_\rho \) by \( \mathbf{I}_\rho^{co} \), \( \mu_{\rho} \) by \( \nu_{\rho} \), the rule \( \text{IndG} \) by \( \text{CoiG} \), and reverse all inclusions.

6. Program Extraction

In this Section we sketch how our realizability interpretation can be used to extract programs from proofs. Since most of the methods and results in this Section can be taken over from (Berger 2010), we will be rather brief and will only comment in detail on the changes and additions necessary.

To get a feel for the problems addressed in this Section, we first look at the extracted programs from Example 1.
Example 5 (Extracting a program for addition). Continuing Examples 1 and 3, let us calculate the realizer extracted from the proof that \( N \) is closed under addition. The core of the proof given in Example 1 was an inductive proof that \( N \subseteq \iota \rightarrow o P[x] \) (where \( P[x] := \lambda y : \iota . \iota N(x + y) \)) under the assumption \( N(x) \). According to Theorem 4, the extracted program of this proof is
\[
\text{ADD}[n] := \text{rec}(\lambda f : \delta . \lambda m . \text{case } m (\lambda a : \delta . n) (\lambda m' : \delta . \text{in}_R(f \cdot m'))) \]
where it is assumed that \( n \) realizes \( N(x) \). Thus, the base and step cases are \( \text{ADD}[n] \cdot (\text{in}_L \text{nil}) \equiv n \) and \( \text{ADD}[n] \cdot (\text{in}_R m) \equiv \text{in}_R(\text{ADD}[n] m) \) respectively. The addition program (the extracted realizer witnessing closure of \( N \) under addition) is then \( \text{Add} := \lambda \delta p . \text{ADD}[\text{pr}_L p] \cdot (\text{pr}_R p) \). Hence, if \( n \) and \( m \) witness (i.e. realize) the pre-conditions \( N(x) \) and \( N(y) \), then the term \( \text{Add}(\text{pair } n m) \) witnesses the post-condition \( N(x + y) \).

Example 5 bares the question how the program \( \text{Add} \) can be used for computing addition. The obvious answer is that one applies \( \text{Add} \) to two unary numbers \( n, m \) (extracted from proofs of the formulas \( N(x) \) and \( N(y) \)) and “runs” the term \( \text{Add} \cdot n \cdot m \). One expects that that run will terminate and output a unary number \( k \) representing \( x + y \) (i.e. realizing the formula \( N(x + y) \)). Theorems 5 and 6 below state exactly this for any situation where computation makes sense, namely if concrete observable data (without function components) are to be expected as realizers.

Definition 13 (Non-computational formula). A \( \Gamma \)-formula \( A \) is non-computational if it does not contain the constant \( \lor \), and for all occurrences of \( \forall \rho, \exists \rho \) in \( A \) and \( x : \rho \in \Gamma \), \( \rho \) is an object-type, that is, \( \rho \) contains only base types from \( I \).

Definition 14 (\( \Sigma \)-formula). A \( \Gamma \)-formula \( A \) is called \( \Sigma \)-formula if (a) it does not contain the constants \( \to \) and \( \nu \rho \), (b) for all occurrences of \( \forall \rho, \exists \rho \) in \( A \) and \( x : \rho \in \Gamma \), \( \rho \) is an object-type, and (c) for all occurrences of \( \mu \rho \), \( \rho \) is of the form \( \sigma \to o \) where \( \sigma \) is an object type.

The pathway to extracted programs is now as follows:

1. Given a proof of a sequent \( \Delta \vdash \Gamma A \) in CST we extract, using the Soundness Theorem (Theorem 2), a closed term \( M \) provably realizing the sequent.

2. If the formulas in \( \Delta \) are non-computational, then the formula \( \bigwedge \Delta \) is equivalent to the assertion that it is realized by some trivial term \( \text{nil}' \) built from \( \text{nil} \) by pairing (dummy lambda-abstraction). Hence we can derive \( \Delta \vdash \Gamma \text{R}(M \cdot \text{nil}') A \).

3. Let \( P \) be the \( \beta \)-normal form of \( M \cdot \text{nil}' \) (here we use the fact that simply typed lambda terms have a unique \( \beta \)-normal form (Barendregt 1992)). \( P \) will be closed and contain only program constants. Let us call such a term \( P \) a program.

4. If the assumptions \( \Delta \) are true in some standard classical model (see below), then in that model the value \( \llbracket P \rrbracket \) of \( P \) will realize \( A \).

5. If \( A \) is a \( \Sigma \)-formula, then \( \llbracket P \rrbracket \) will be a data, that is a finite combinatorial object built from \( \text{nil} \) by left and right injections and pairing (numerals are examples of data). We can identify a data with the canonical program built from \( \text{nil}, \text{in}_L, \text{in}_R, \text{pair} \) defining it.
Now we can employ a *Computational Adequacy Theorem* (Berger 2010 Theorem 11) according to which a program \( P \) denoting a data reduces to that data in a suitable lazy big-step semantics.

Since the big-step semantics is denotationally correct we know that the data \([P]\) realizes \( A \) and can be computed from \( P \).

The steps (1-7) amount to a proof of the following theorem:

**Theorem 5 (Program extraction for data).** Let \( \Gamma \vdash \Delta : o \) be a finite set of non-computational formulas. Then from a proof of \( \Delta \vdash A \), where \( A \) is a \( \Sigma \)-formula, one can extract a program \( P \) with the property that \( P \) reduces to some data \( d \) realizing \( A \). Furthermore, a proof that \( \Delta \) implies that \( d \) realizes \( A \) is extracted.

Theorem 5 can be easily generalized to the situation where the proven formula is an implication between \( \Sigma \)-formulas:

**Theorem 6 (Program extraction for data functions).** Let \( \Gamma \vdash \Delta : o \) be a finite set of non-computational formulas. Then from a proof of \( \Delta \vdash A \rightarrow B \), where \( A \) and \( B \) are \( \Sigma \)-formulas, one can extract a program \( P \) with the property that for any data \( d \) realizing \( A \), \( P \cdot d \) reduces to some data realizing \( B \). Furthermore, a proof of \( \Delta, \text{Data} d \vdash_r \delta \) \( R(A) d \rightarrow R(B) (P \cdot d) \) is extracted, where \( \text{Data} : \delta \rightarrow o \) is a closed term defining the property of being a data.

Theorems 5 and 6 cover most applications of program extraction, but they can be generalized to considerably larger classes of formulas in a similar style as in the Minlog system (Minlog). This will be the subject of further work. Example 1 is covered by Theorem 6 since \( \mathbb{N} x \land \mathbb{N} y \rightarrow \mathbb{N}(x + y) \) is an implication between \( \Sigma \)-formulas.

Regarding the classical model referred to in step (4) above we have to be a bit careful. On the one hand, we want types over the base types \( \iota \in \mathcal{I} \) and \( o \) and the constants of such types to be interpreted classically. In particular as interpretation of a base type \( \iota \in \mathcal{I} \) any nonempty sets should be admitted, \( o \) should be interpreted as the classical Booleans \( \{0, 1\} \), and a function type \( \rho \rightarrow \sigma \) should be interpreted as the set of all functions from \( \rho \rightarrow \sigma \). On the other hand, types over the base type \( \delta \) and the associated program constants do not admit such a set-theoretic interpretation: the program constants allow to encode the untyped lambda-calculus, therefore it cannot have a set-theoretic model where the type \( \delta \rightarrow \delta \) contains all functions from \( \delta \) to \( \delta \) (unless \( \delta \) is a singleton).

A simple way of joining the world of classical abstract mathematics with the constructive world of programs is to interpret types in the cartesian closed category of quasi-domains and continuous functions defined below.

**Definition 15 (Quasi-domain).** A *quasi-domain* is a topological space whose \( T_0 \)-collapse is a Scott-domain (Gierz et al. 2003) (called the domain associated with a quasi-domain).

Quasi-domains may also be defined order-theoretically as those quasi-orders \((D, \sqsubseteq)\), i.e. transitive and reflexive relations, whose quotients by the equivalence relation

\[
x \sim y :\iff x \sqsubseteq y \land y \sqsubseteq x
\]
are (Scott-)domains. A more direct definition is obtained by adapting the axioms of a domain to the setting of quasi-orders. To this end define for $X,Y \subseteq D$

$$X \sqsubseteq Y := \forall x \in X \exists y \in Y . x \sqsubseteq y$$

and identify elements of $D$ with their singleton sets, so that statements like "$x \sqsubseteq Y$" and "$Y \sqsubseteq x$" are meaningful. A set $X \subseteq D$ is bounded by $y \in D$ if $X \sqsubseteq y$, and directed if every finite subset of $X$ is bounded by some element in $X$. Let

$$\bigsqcup X := \{ y \in D \mid X \sqsubseteq y \wedge \forall z \in D . X \sqsubseteq z \rightarrow y \sqsubseteq z \}$$

An element $x \in D$ is compact if for all directed subsets $X$ of $D$, if $x \sqsubseteq \bigsqcup X$, then $x \sqsubseteq X$. Now, $(D, \sqsubseteq)$ is a quasi-domain iff the following properties hold:

- If $X \subseteq D$ is directed, then $\bigsqcup X$ is nonempty (cpo property).
- If $X \subseteq D$ is bounded, then $\bigsqcup X$ is nonempty (bounded completeness).
- For every $x \in D$, the set $\hat{x}$ of compact elements below $x$ is directed and $x \in \bigsqcup \hat{x}$ (algebraicity).
- The set of $\sim$-equivalence classes of compact elements is countable (countable base).

The Scott-topology can be defined for quasi-domains in the same way as for domains, namely a set $U \subseteq D$ is Scott-open if it is upwards closed and for every $x \in U$ there exists a compact $x_0 \in U$ below $x$. Some important properties of quasi-domains are listed in the theorem below. The straightforward proofs are omitted.

**Theorem 7.**

(a) Each continuous endo-map $f$ on a domain has a least fixed-point $x$ up to $\sim$, i.e. $f(x) \sim x$.

(b) Quasi-domains and continuous functions form a cartesian closed category.

(c) Domain-equations can be solved up to $\sim$-isomorphism.

For our purpose, the two extreme cases of quasi-domains are of primary interest, namely, plain non-empty sets with the trivial topology (which collapse to the trivial one-point domain) and (ordinary Scott-)domains. The interpretation of RCST in quasi-domains is now straightforward: the base type $\delta$ is interpreted as the domain $D$ solving the domain equation

$$D = 1 + D + D \times D + [D \rightarrow D],$$

(which is a slightly enriched version of Scott’s model $D_\infty$ (Scott 1970)), as required by the Adequacy Theorem used in step (6). Here $1$ is the one-point domain, and $+, \times, [\cdot \rightarrow \cdot]$ denote the separated sum, cartesian product and continuous function space. All other base types are interpreted classically, i.e. as plain non-empty sets. In particular the type $o$ is interpreted as the classical Booleans $\{0, 1\}$. It follows that every predicate type is interpreted as a plain set, and therefore all logical operators, including quantifiers and the least and greatest fixed-point operators can be interpreted classically.

Having settled the issue of interpreting RCST we return to our running example of Fibonacci numbers.

**Example 6 (Extracting the Fibonacci numbers).** The Fibonacci program realizing
Realizability interpretation of CST

the fact that FIB contains all pairs of natural numbers (see Examples 2 and 4) is

\[ \text{Fib} := \text{rec}(\lambda f : \delta . \lambda \delta . \text{pair}(\text{pr}_L p)(f \cdot (\text{pair}(\text{pr}_R p)(\text{Add} \cdot p)))) \].

It defines a function that computes for every pair of natural numbers \((n, m)\) the infinite stream of Fibonacci numbers starting with \(n, m\). In order to compute with it we need to extract finite data from these streams. For example, we can access their \(n\)-th elements. To this end we define inductively

\[ \text{FIB'} x y z = (z = \_ \land \text{N} x) \lor \text{FIB'} y (x + y) (z - 1) \]

Hence \(\text{FIB'} x y z\) means that the \(z\)th member of the Fibonacci sequence starting with \(x, y\) is a natural number. One can easily prove

\[ \forall z : \_ . \text{N} z \rightarrow \forall x, y : \_ . \text{FIB} x y \rightarrow \text{FIB'} x y z \]

by induction on \(\text{N}\). Combining this with the proof in Example 2 we obtain \(\forall z : \_ . \text{N} z \rightarrow \text{FIB'} 1 1 z\), which is covered by Theorem 6. Given a natural number \(n\) in unary notation, the extracted program will read off the \(n\)th element of the infinite stream of Fibonacci numbers. Note that memoization takes place here: if we compute first the 100th Fibonacci number, then 100 additions of natural numbers will be carried out, but if we later compute the 99th number, the (partially computed) stream will only be looked up without performing any additions. We could of course prove directly the formula \(\forall x, y, z : \_ . \text{N} x \land \text{N} y \land \text{N} z \rightarrow \text{FIB'} x y z\) (without detour through the coinductively defined predicate \(\text{FIB}\)), but then no memoization would take place.

7. Conclusion

We presented a realizability interpretation of an intuitionistic version of Church’s Simple Theory of Types that goes beyond previously known interpretations in that it interprets the full theory and gives direct interpretations of monotone induction and coinduction that are completely dual to each other.

As stated in the Introduction, the main motivation for moving from first- or second-order to full higher-order logic, and from strictly positive to monotone induction and coinduction is not the greater generality or expressive and proof-theoretic power, but rather ease of formalization and better usability. However, there are important cases where the greater generality allows for new applications. In (Berger 2010) a definition of uniformly continuous functions on a compact real interval is given that involves a coinductive definition with an inductive definition nested inside:

\[ C_1 := \nu F . \mu G . \{ g \mid \exists i \in \text{SD}, f \in \mathbb{I}^2(g = \text{av}_i \circ f \land F f) \lor \bigwedge_{i \in \text{SD}} G (g \circ \text{av}_i) \} \]

where \(\circ\) denotes function composition, the variables \(F, G\) range over sets of functions on the compact interval \(I := [-1, 1]\), \(\text{SD} := \{-1, 0, 1\}\) is the set of signed digits, and \(\text{av}_i x := (x + i)/2\). In CST the underlying space of real numbers can be modeled as a base type \(\_\) with suitable arithmetic operations with \(I : \_ \rightarrow o\), similar to our earlier Example 1. In (Berger 2011) and (Miyamoto et al. 2013) the same example is expressed
in versions of second-order logic with considerable more effort. In type-theoretic systems like Agda and Coq this example cannot be formalized directly since the formal strict positivity and guardedness requirements of these systems are not met. More precisely, Agda and Coq do not recognize the occurrence of a coinductive variable ($F$ in our case) in the scope of an inductive definition ($\mu G \ldots$ in our case) as guarded. An implementation in Coq is possible using Mendler style (co)induction, which however, prevents the use of Coq’s program extraction mechanism (see also (Nakata and Uustalu 2010) for a similar example of nested induction/coinduction and its implementation in Coq).

Further work. The extracted realizers of our systems are untyped. We see this as an advantage since it means that the target programming language and its associated Ade-

quacy Theorem do not have to be changed when the system CST is changed (for example, by adding new proof principles). Nevertheless the realizers can be typed in a suitable ex-
tension of $F_\omega$ and one could try to adapt the proof techniques in (Mendler 1991; Geuvers 1992; Miranda-Perea and Gonzalez-Huesca 2012; Abel et al. 2005) to obtain strong nor-
malization (of course after a suitable reformulation of the realizers of monotone induction
and coinduction avoiding the general recursion operator). It will also be interesting to see
whether the approach to monotonicity (Abel et al. 2005), in the style of logical relations,
can be adapted to our setting. Furthermore, for non-trivial applications it is crucial that
the rather elaborate optimizations of program extraction in the Minlog system (Minlog)
can at least partially be adapted to our system. On the logical side one should try to
define limited forms of choice principles that are realizable.

References

Abel, A., Matthes, R., Uustalu, T.: Iteration and coiteration schemes for higher-order and nested
Abel, A., Pientka, B., Setzer, A.: Copatterns: Programming infinite structures by observa-
tions. In 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages
(POPL’13), 2013.
Barendregt, H.: Lambda calculi with types. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum,
Andrews, P., Issar, S., Nesmith, D., Pfennig, F.: The TPS theorem proving system. 10th Inter-
national Conference on Automated Deduction, 641-642 (1990)
Berger, U.: Realisability for Induction and Coinduction with Applications to Constructive Anal-
Berger, U.: From coinductive proofs to exact real arithmetic: theory and applications. Logical
Methods in Computer Science 7(1), 1-24 (2011)
Berger, U., Miyamoto, K., Schwichtenberg, H., Seisenberger, M.: Minlog - a tool for program
extraction for supporting algebra and coalgebra. In CALCO-Tools, volume 6859 of LNCS,
Berger, U., Seisenberger, M.: Proofs, programs, processes. In F. Ferreira, B. Löwe, E. May-
ordomo, and L. M. Gomes, editors, Programs, Proofs, Processes, 6th Conference on Com-
Feferman, S. Constructive Theories of Functions and Classes. Logic Colloquium ’78, 97, 159-224 (1979)


Raffalli, C., Ruyer, F.: Realizability of the Axiom of Choice in HOL. (An Analysis of Krivine’s Work) Fundamenta Informaticae - Logic for Pragmatics 84(2), 241-258 (2008)