Computability Beyond Church-Turing via Choice Sequences

Mark Bickford
Cornell University
markb@cs.cornell.edu

Liron Cohen*†
Cornell University
lironcohen@cornell.edu

Robert L. Constable
Cornell University
rc@cs.cornell.edu

Vincent Rahli*‡
SnT, University of Luxembourg
vincent.rahli@gmail.com

Abstract

Church-Turing computability was extended by Brouwer who considered non-lawlike computability in the form of *free choice sequences*. Those are essentially unbounded sequences whose elements are chosen freely, i.e. not subject to any law. In this work we develop a new type theory BITT, which is an extension of the type theory of the Nuprl proof assistant, that embeds the notion of choice sequences. Supporting the evolving, non-deterministic nature of these objects required major modifications to the underlying type theory. Even though the construction of a choice sequence is non-deterministic, once certain choices were made, they must remain consistent. To ensure this, BITT uses the underlying library as *state* and store choices as they are created. Another salient feature of BITT is that it uses a Beth-like semantics to account for the dynamic nature of choice sequences. We formally define BITT and use it to interpret and validate essential axioms governing choice sequences. These results provide a foundation for a fully *intuitionistic* version of Nuprl.

ACM Reference format:
Mark Bickford, Liron Cohen, Robert L. Constable, and Vincent Rahli. 2018. Computability Beyond Church-Turing via Choice Sequences. In *Proceedings of , , (LICS 18),* 12 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 Introduction

Brouwer's broader notion of computability extends that of Church-Turing by the inclusion of non-lawlike computability based on *free choice sequences*. Those are fundamental objects introduced by Brouwer [10] that lay at the heart of intuitionistic mathematics. They are there described as "new mathematical entities...in the form of infinitely proceeding sequences, whose terms are chosen more or less freely from mathematical entities previously acquired". The first key feature of free choice sequences is the fact that they are infinitely proceeding. This is a non-platonic approach in which a free choice sequence comes into existence by a never ending process of picking elements from a previously well-defined collection, e.g. natural numbers. Therefore, a free choice sequence is never fully completed and can always be extended. The second component of free choice sequences is that the choices are made freely, that is, not governed by any law.

In this work we show that the constructive type theory implemented by the Nuprl proof assistant [15; 4] can be consistently extended to an intuitionistic type theory that supports Brouwer's broader sense of computability through the embedding of free choice sequences. Since the concept of non-lawlike computations, as well as the notion of spreads, Bar Induction, and the Continuity Principle, are the salient consequences of Brouwer's intuitions [11; 12], we call this new extended type theory BITT.[1] BITT then paves the way for turning Nuprl into a fully *intuitionistic* proof assistant.

The theory governing free choice sequences has been widely studied, but the various works on the subject take different interpretations of the basic notions.[2] This results in a variety of interpretations of free choice sequences (e.g., [26; 7; 43; 42; 30; 47; 35]). In this paper we aim to create a completely formal account of choice sequences, driven by the design constraints of their implementation in a theorem prover. That is, we offer an account that captures fundamental notions concerning free choice sequences, while being suitable for implementation.

In [39] the assumption of existence of choice sequences was exploited to establish Bar Induction, a key intuitionistic principle, in Nuprl. However, choice sequences were there used only as an instrumental tool in the metatheory, not embedded into the theory itself. Choice sequences were generated using Coq functions, including such that use non-computable axioms. As noted in [39]: "choice sequences do not have to be—and are not—part of the syntax of Nuprl definitions and proofs, i.e., the syntax visible to users". This approach had some undesired consequences. Mainly, it made Nuprl's syntax infinitary, which in turn had the side effect that many properties, such as syntactic equality or α-equality, became undecidable (in the metatheoretical syntax of Nuprl).

In this work we remedy this situation by implementing the concept of choice sequences in the theory itself as finite, unbounded sequences, as opposed to infinite sequences in [39]. The formalization presented here resolves both aforementioned issues. It incorporates choice sequences into the user syntax, while keeping it finitary, which entails that properties such as syntactic equality and α-equality remain decidable. One of our long-term goals for the implementation is to allow the derivation of the key Bar Induction and Continuity principles, which have been widely studied in the literature (e.g., [26; 30; 44; 47; 7; 41; 25; 22]) but only recentely in the context of a mechanized proof assistant [38; 39]. This

*Rahli and Cohen designed and have contributed equally to the design of the BITT theory presented in this paper. The Coq formalization is by Rahli.

†Cohen is supported by: Fulbright Post-doctoral Scholar program; Weizmann Institute of Science – National Postdoctoral Award program for Advancing Women in Science; Eric and Wendy Schmidt Postdoctoral Award program for Women in Mathematical and Computing Sciences.

‡Rahli was partially supported by the Fonds National de la Recherche Luxembourg (FNR) through PEARL grant FNR/P14/8149128.

LICS 18,
2018. 978-x-xxxx-xxxx-x/YY/MM...$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

[1] The term "intuitionistic type theory" had already been used by Per Martin-Löf [33; 36]. However, those type theories do not include Brouwer's fundamental idea of non-lawlike computation.

[2] In the literature free choice sequences are sometimes called "lawless sequences" [35].

presents an implementation challenge, as the two principles correspond to different properties of choice sequences. Bar Induction requires the existence of *some* form of non-recursive functions, i.e. free choice sequences. Continuity, on the other hand, entails a restriction on the behavior of *all* non-recursive functions, i.e. it puts a constraint on the topological space they induce.[3] Accordingly, our work is to carefully craft the design constraints of the theory of free choice sequences, balancing these two properties.

Implementing choice sequences entails incorporating certain non-deterministic features into BITT, as well as developing ways to handle them. One way to think about non-determinism is as a process that even for the same input, can exhibit different behaviors. The form of non-determinism required for reasoning in the presence of choice sequences is slightly different. While the process of picking the values of a choice sequence is non-deterministic, once certain values were chosen, they must remain unchanged. To capture that we rely on the digital library of facts and definitions underlying Nuprl to act as *state*, and store in it our already chosen values of the choice sequence. This required BITT to extend Nuprl's computation system, as is presented in Sec. 3.

To support this evolving nature of choice sequences, and thus libraries, the semantics of BITT also extends that of Nuprl. In BITT we invoke a Beth-style semantics [48; 21; 20, Sec.5.4], in which the possible worlds correspond to extensions of the library. Under this Beth model, types are interpreted as partial equivalence relations (PERs) on closed terms that need only exist in *bars* of the current library, i.e, collections of libraries covering all possible extensions of the current library (see Sec. 4). A Beth model is especially well-suited to model choice sequences because there expressions only need to "eventually" compute to values, which is compatible with the "eventual" nature of choice sequences that are only partially given at a given time, with the promise that they can always be extended in the future. We show that the resulting type system satisfies the standard properties of a type systems (such as transitivity and symmetry), as well as properties which are unique to possible-world semantics, such as monotonicity and locality.

After establishing the well-formedness of the resulting type system, we demonstrate its adequacy for the theory of choice sequences. We do so by validating inference rules governing choice sequences (see Sec. 6). The entire development and results presented in this paper have been fully formalized in Coq, and in the sequel we provide pointers to the Coq formalization in the appropriate places.

Exploring Brouwer's wider notion of computability in a formal setting has, in our opinion, the potential to provide a broader and deeper foundational theory for computer science. Nevertheless, the integration of choice sequences into a mechanized proof assistants is not only important from a foundational standpoint, but also seem to offer interesting consequences and possible practical applications. For instance , we believe that this formalization can be used to model complex systems. Computable functions could be used to model the processes of a distributed system, while the free choice sequences could be used to model sensors (or uncontrolled, unpredictable inputs from the environment).

Outline. The rest of the paper is organized as follows: Sec. 2 provides essential background on key features of Nuprl's type theory. Sec. 3 describes the integration of choice sequences into BITT, mainly the use of the underlying library. Sec. 4 provides a detailed account of how the semantics of Nuprl has been modified into a Beth-like one in BITT. This includes the formal treatment of bars, proving preservation of salient properties of the type system, as well as new properties that are distinctive to the new semantics. Sec. 5 describes the extension of the function type $\mathbb{N} \to \mathbb{N}$, which previously contained only computable functions, by choice sequences of numbers. Sec. 6 discusses the axioms of choice sequences in BITT. Finally, Sec. 7 concludes.

2 Background

Nuprl implements a dependent type theory called Constructive Type Theory (CTT). This section presents some key aspects of CTT.

Computation system. Nuprl's programming language is an untyped (à la Curry), lazy λ-calculus with pairs, injections, a fixpoint operator, etc. For efficiency, integers are primitive and Nuprl provides operations on integers as well as comparison operators.

Fig. 1 presents a subset of Nuprl's syntax and small-step operational semantics. We only show in it the part that is either mentioned or used in this paper. A term is either (1) a variable; (2) a canonical form, i.e., a value or an exception (see [38]); or (3) a non-canonical term. A non-canonical term t has one or two *principal arguments*—marked using boxes in Fig. 1—which are terms that have to be evaluated to canonical forms before t can be reduced further. For example, the application $f\ a$, often written as $f(a)$, diverges if f diverges. In Fig. 1 we omit rules that reduce principal arguments such as: if $t_1 \mapsto t_2$ then $t_1\ u \mapsto t_2\ u$.

We use the following abstractions in the sequel: $\bot = \mathtt{fix}(\lambda x.x)$, $\mathtt{tt} = \mathtt{inl}(\star)$, and $\mathtt{ff} = \mathtt{inr}(\star)$. Also, we write $a =_T b$ for the type $a = b \in T$, $\lambda x_1, \ldots, x_n.t$ for $\lambda x_1. \ldots \lambda x_n.t$, and $t_1 \to t_2$ for the non-dependent product type (i.e. the function type).[4]

Type system. Nuprl's types are interpreted as partial equivalence relations (PERs) on closed terms [2; 3; 18]. The PER semantics can be seen as an inductive-recursive definition of: (1) an inductive relation $T_1 \equiv T_2$ that expresses type equality; (2) a recursive function $a \equiv b \in T$ that expresses equality in a type. For example, one case in the definition of $T_1 \equiv T_2$ states that (i) T_1 computes to $\forall x_1 : A_1.\ B_1$; (ii) T_2 computes to $\forall x_2 : A_2.\ B_2$; (iii) $A_1 \equiv A_2$; and (iv) for all closed terms t_1, t_2 such that $t_1 \equiv t_2 \in A_1$, $B_1[x_1 \backslash t_1] \equiv B_2[x_2 \backslash t_2]$. We say that a term t *inhabits* or *realizes* a type T if t is equal to itself in the PER interpretation of T, i.e., $t \equiv t \in T$. It follows from the PER interpretation of types that an equality type of the form $a = b \in T$ is true (i.e. inhabited) iff $a \equiv b \in T$ holds. [5; 37]. Note that an equality type can only be inhabited by the constant $\star$, i.e., they do not have computational content, unlike in Homotopy type theory [46].

Computational equivalence relation. Nuprl is closed under Howe's computational equivalence $\sim$, which was proven to be a congruence [24]. In general, computing and reasoning about computation in Nuprl involves reasoning about Howe's computational equivalence relation. It is commonly used to reduce expressions by proving that terms are computationally equivalent and using the fact that $\sim$ is a congruence. For that, Nuprl provides the type $t_1 \simeq t_2$, which is the theoretical counterpart of the metatheoretical relation $t_1 \sim t_2$. To reason about any term in the computation system, Nuprl provides the `Base` type, which is the type of all closed terms of the computation system with $\sim$ as its equality.

[3] Note that *full* Bar Induction (i.e., where the bar is not constrained to be decidable or monotone) contradicts the Continuity Principle [26, Sec.7.14,Lem.*27.23].

[4] Note that BITT and its metatheory share similar connectors. For readability, we often omit type information in quantifiers for the metatheoretical ones.

Figure 1 Syntax (top) and operational semantics (bottom) of a subset of Nuprl

$$
\begin{array}{llllll}
v \in \texttt{Value} ::= vt & \text{(type)} & \mid \texttt{inl}(t) & \text{(left injection)} & \mid \star & \text{(axiom)} \quad \mid \langle t_1, t_2 \rangle \quad \text{(pair)} \\
\quad \mid \lambda x.t & \text{(lambda)} & \mid \texttt{inr}(t) & \text{(right injection)} & \mid i & \text{(integer)} \\
vt \in \texttt{Type} ::= \mathbb{Z} & \text{(integer type)} & \mid \forall x : t_1.\ t_2 & \text{(product)} & \mid t_1 = t_2 \in t & \text{(equality)} \quad \mid \{x : t_1 \mid t_2\} \quad \text{(set)} \\
\quad \mid \texttt{Base} & \text{(base)} & \mid \exists x : t_1.\ t_2 & \text{(sum)} & \mid t_1 + t_2 & \text{(disjoint union)} \quad \mid t_1 // t_2 \quad \text{(quotient)} \\
\quad \mid \mathbb{U}_i & \text{(universe)} & \mid t_1 \simeq t_2 & \text{(bisimulation)} & & \\
t \in \texttt{Term} ::= x & \text{(variable)} & \mid \texttt{let } x := \boxed{t_1} \texttt{ in } t_2 & \text{(call-by-value)} & \mid \texttt{case } \boxed{t_1} \texttt{ of inl}(x) \Rightarrow t_2 \mid \texttt{inr}(y) \Rightarrow t_3 & \text{(decide)} \\
\quad \mid v & \text{(value)} & \mid \texttt{let } x, y = \boxed{t_1} \texttt{ in } t_2 & \text{(spread)} & \mid \textbf{iflam}(\boxed{t_1}, t_2, t_3) & \text{(lambda test)} \\
\quad \mid \boxed{t_1}\ t_2 & \text{(application)} & \mid \texttt{fix}(\boxed{t}) & \text{(fixpoint)} & &
\end{array}
$$

$$
\begin{array}{ll}
(\lambda x.F)\ a \mapsto F[x\backslash a] & \texttt{fix}(v) \mapsto v\ \texttt{fix}(v) \\
\texttt{let } x := v \texttt{ in } t \mapsto t[x\backslash v] & \textbf{iflam}(\lambda x.t, t_1, t_2) \mapsto t_1 \\
\texttt{let } x, y = \langle t_1, t_2 \rangle \texttt{ in } F \mapsto F[x\backslash t_1; y\backslash t_2] & \textbf{iflam}(v, t_1, t_2) \mapsto t_2, \text{ if } v \text{ is not a } \lambda\text{-term} \\
\texttt{case inl}(t) \texttt{ of inl}(x) \Rightarrow F \mid \texttt{inr}(y) \Rightarrow G \mapsto F[x\backslash t] & \texttt{case inr}(t) \texttt{ of inl}(x) \Rightarrow F \mid \texttt{inr}(y) \Rightarrow G \mapsto G[y\backslash t]
\end{array}
$$

Squashing. Nuprl has a *squashing* mechanism, which we use in Sec. 6. It throws away the evidence that a type is inhabited and squashes it down to a single inhabitant using set types [15, pp.60]: $\downarrow T = \{\texttt{Unit} \mid T\}$. The only member of this type is the constant $\star$, which is `Unit`'s single inhabitant, and which is similar to `()` in languages such as OCaml, Haskell or SML. The constant $\star$ inhabits $\downarrow T$ if T is true/inhabited, but we do not keep the proof that it is true. See [38] for more details on squashing.

Sequents and rules. Sequents are of the form $h_1, \ldots, h_n \vdash T \lfloor \text{ext } t \rfloor$. The term t is a member of the type T, which in this context is called the *extract* or *evidence* of T. Extracts are programs that are computed by the system once a proof is complete. We will sometimes omit proof extracts when they are irrelevant to the discussion. An hypothesis h is of the form $x : A$, where the variable x is referred to as the name of the hypothesis and A its type. Such a sequent states, among other things, that T is a type and t is a member of T. A rule is a pair of a conclusion sequent S and a list of premise sequents, $S_1, \cdots, S_n$, which we write as:

$$\frac{S_1 \quad \cdots \quad S_n}{S}$$

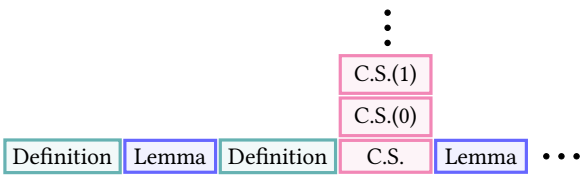
Several equivalent definitions for the validity of sequents appear in the Nuprl literature [15; 18; 27; 5]. Since our results are invariant to the specific semantics, we do not repeat them here. The sequent semantics standardly induces the notion of validity of a rule, i.e., the validity of the premises entails the validity of the conclusion.

Coq formalization. Recently, CTT has been formalized in Coq [5; 37; 38]. The implementation includes: (1) Nuprl's computation system; (2) Howe's computational equivalence relation, and a proof that it is a congruence; (3) a definition of the PER semantics of CTT; (4) definitions of Nuprl's derivation rules, and their soundness proofs w.r.t. the PER semantics (5) and a proof of Nuprl's consistency. This formalization allows for a safe and mechanical way to verify the soundness of existing as well as new rules.

3 Library as State

This section discusses the introduction of choice sequences into BITT's computation system, in which the library plays a major role.Basically, the library is used as a *state* in which we store the choices of values that have been made for a particular choice sequence at a given point in time. In mainstream programming languages such information can be stored using global variables. However, since proof assistants do not support global variables, the library is treated as one to enable stateful computations.

Figure 2 Library structure



3.1 Open-ended Libraries

Until now, a Nuprl library consisted of a list of definitions and lemmas. We here introduce a new kind of library entries – that of choice sequences. A choice sequence entry is again a list, this time of terms. Thus, a library can be extended in two orthogonal directions: by adding more entries to the library, or by adding more values to a choice sequence entry (see Fig. 2, where C.S. stands for a choice sequence entry). This can be seen as an interpretation of Brouwer's notion of a choice sequence progressing over *time*, as implemented by progressing over *library extensions*.

In [39] choice sequences were treated as infinite sequences. Here, because we now have a concrete implementation, instead of a priori assuming an infinite object, we take the infinite nature of choice sequences as potential. Choice sequences are: (1) essentially always finite, (2) but ever growing. Accordingly, we capture these two components in different layers of the implementation. Choice sequences are finite at any stage of the library since they are implemented simply as lists. However, they are infinitely proceeding because the library is open-ended and can always be extended (this is accounted for in the interpretation of types as explained in Sec. 4). The fact that a library is open-ended allows for arbitrary extensions of any particular choice sequence, but also for the introduction of arbitrarily many choice sequences entries.

3.2 Restrictions and Name Spaces

Choice sequences are implemented such that each choice sequence entry in the library comes equipped with a restriction. There is a vast discussion in the literature about the various types of restrictions: in [6; 7] there is a distinction between "definitive" restrictions and "provisional" restrictions: definitive ones are permanent, and provisional ones can be lifted at a later stage; [43] discusses choice sequences which are "hesitant" (start free, but at any stage

may be restricted to continue by a law); and there is also a classification of restrictions by their order (a restriction on future restriction is a second-order restriction). Restrictions can be made in advance or imposed at any stage of the construction of a sequence.

We implement a simple notion of restrictions, which are given in advance and can either be a law given as a Coq function f from numbers to terms, such that the nth entry of the choice sequence has to be $f(n)$; or some binary *restriction predicate* P on term/number pairs. In the latter case, $P(n,t)$ expresses whether t is the nth choice. Also, in the latter case one has to provide a function of default values d (from numbers—positions in the list of choices—to choices), and a proof of $P(n,d(n))$, to ensure that the sequence can be extended with legal values. This does not mean that any choice sequence is restricted. One can construct an unrestricted choice sequence (one in which the choices can be any term) by employing the empty restriction, i.e. a predicate that always returns true (formally, $\lambda n,t.\mathtt{True}$). When adding a new value to a choice sequence one has to prove that it satisfies the restriction of the sequence. For decidable restrictions, this can be done automatically by the system. Note that while we only support restrictions imposed a priori on a sequence, our restrictions are parameterized by the position in the sequence of choices. Therefore, one can define a restriction that only applies to values starting from a specific location. For example, the restriction $\lambda n,t.\mathtt{if}\ n < 10\ \mathtt{then}\ \mathtt{True}\ \mathtt{else}\ 2 \leq t$, enforces that the choices starting from position 10 must be greater than or equal to 2, but choices up to position 10 are unconstrained.

Two notable restrictions we shall use are the followings. The first restriction predicate enforces that the choices have to be natural numbers, i.e., the restriction predicate is $\lambda n,t.\exists i \in \mathbb{N}.\ t = i$. For this we supply the default value function $\lambda n.0$. The second restriction enforces that the choices have to be natural numbers, with the constraint that the first few values have to agree with a given list of numbers. Formally, given a list of natural numbers l, the restriction predicate is $\lambda n,t.\mathtt{if}\ n \leq |l|\ \mathtt{then}\ t = l[n]\ \mathtt{else}\ \exists i \in \mathbb{N}.\ t = i$.[5] The default value function is $\lambda n.\mathtt{if}\ n \leq |l|\ \mathtt{then}\ l[n]\ \mathtt{else}\ 0$.

To capture key properties of choice sequences, our formalization further provides a mechanism for enforcing certain restrictions through choice sequence name spaces. A choice sequence name is a pair consisting of a string and a constraint, where a constraint can either be a number or a list of numbers. Constraints are used to enforce some restrictions as follows. The constraint 0 enforces the choice sequence to be a choice sequence of *natural numbers*; any other number $n > 0$ does not constrain the type of values in the sequence (which leaves room for incorporating more specific constraints in future extensions). Finally, a list of numbers l is used to enforce that the choice sequence is a free choice sequence of numbers such that the $|l|$ first elements in it coincide with those given by the list l. For example, $\langle$"a", 0$\rangle$ states that a must be a free choice sequence of numbers; $\langle$"a", 1$\rangle$ states that a is a free choice sequence that can be filled with any values—not just numbers; and $\langle$"a", $[3,2,5]\rangle$ forces a to be a free choice sequence of numbers that starts with the choices 3, 2, and 5.

Definition 3.1. BITT's term syntax and operational semantics extends those of Nuprl (presented in Sec. 2) with choice sequences. The formal extensions are given in Fig. 3, where we use C.S. as an abbreviation for choice sequence, and $n, n_1, \ldots, n_k$ for variables ranging over natural numbers.[6]

Figure 3 Extended syntax and operational semantics

$csn \in \mathtt{CSName}$	$::= \langle s, space\rangle$	(C.S. name)
$s \in \mathtt{RawCSName}$		
$space \in \mathtt{Space}$	$::= n \mid [n_1;\ldots;n_k]$	(C.S. name space)
$v \in \mathtt{Value}$	$::= \cdots \mid \mathtt{seq}(csn)$	(C.S.)
$vt \in \mathtt{Type}$	$::= \cdots \mid \mathtt{Free}(n)$	(C.S. type)
$t \in \mathtt{Term}$	$::= \cdots \mid \mathtt{if}\ \boxed{t_1}{=}\boxed{t_2}\ \mathtt{then}\ t_3\ \mathtt{else}\ t_4$	(C.S. equality)
$\mathtt{if}\ \mathtt{seq}(csn_1){=}\mathtt{seq}(csn_2)\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2 \mapsto_{lib} t_1$		, if $csn_1 = csn_2$
$\mathtt{if}\ \mathtt{seq}(csn_1){=}\mathtt{seq}(csn_2)\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2 \mapsto_{lib} t_2$		, if $csn_1 \neq csn_2$
$\mathtt{seq}(csn)(i) \mapsto_{lib} cs[i]$		, if $cs[i]$ is defined in lib

Name spaces are introduced for choice sequences which can either be a number or a finite list of numbers. A choice sequence name is a pair which consists of a raw name (i.e. the name identifier for the entry in the library—simply a string in our formalization) and a space name. The type of choice sequences is $\mathtt{Free}(n)$, where, as for choice sequences, n is a name space. We use 0 for the type of free choice sequences of natural numbers, and any other number $n > 0$ does not constrain the inhabitants of $\mathtt{Free}(n)$ (this, again, allows for the addition of other constraints in the future). The choice sequences are incorporated as values of the form $\mathtt{seq}(csn)$, where csn is a choice sequence name; and a new term of the form $\mathtt{if}\ t_1{=}t_2\ \mathtt{then}\ t_3\ \mathtt{else}\ t_4$ is introduced for their equality judgment. Since choice sequences are identified with their names, it computes by simply comparing the corresponding names. Bottom of fig. 3 shows formally how it computes. Also, in addition to being able to apply λ-abstractions, we allow applying choice sequences of the form $\mathtt{seq}(csn)$ to numbers. The application $\mathtt{seq}(csn)(i)$ reduces to $cs[i]$ if $0 \leq i$ and if the i's choice for the choice sequence named cs is available in the current library, in which case $cs[i]$ returns that choice—otherwise, it is left undefined. Note that, unlike in the computation system described in Fig. 1, the extended computation rules explicitly depend on the current library.

Definition 3.2.

- A library is called *safe* if the values of its choice sequences satisfy the corresponding restrictions, and those restrictions respect the names of the sequences (as mentioned above).
- A library lib_2 *extends* a library lib_1, denoted by $\mathtt{extends}(lib_2, lib_1)$, if each entry accessible in lib_1 is also accessible in lib_2. For a definition or lemma, the two entries have to be the same, and for a choice sequence entry, the list of choices made so far in lib_1 has to be a sublist of the corresponding list in lib_2.

When defining an extension lib' of a library lib one must produce a proof that lib' is safe assuming that lib is safe. For simplicity, we assume all libraries are safe in the remainder of the paper.

3.3 Unbounded Objects

Since choice sequences are open-ended objects, it might be that to prove a theorem or carry on a computation one needs to know the value of a choice sequence at a certain point, say the 8th element in the sequence, but at that given stage it is yet undefined. How do we want our formal system to behave in such a situation? In the

[5] As usual, $|l|$ is the length of the list l, and $l[n]$ is the $(n+1)$th element of the list.

[6] See https://github.com/vrahli/NuprlInCoq/tree/beth/computation/library.v for a detailed account of the extension of Nuprl's computation system.

intuitionistic theory of free choice sequences, a reasonable answer will be 'wait until the creative subject picks enough values in the sequence'. This suggest one possible implementation: the system can print out a message to the user asking for more values until there is sufficient data. Another possibility is to have the system automatically fill in values up to the desired place in the sequence. This can be done by some random numbers generator that is being called in such situations. We can even use a computable function to complete the particular segment of the sequence. The free nature of the sequence is kept because the generator (random or not) is only applied to create the values, and then only the values are stored in the library, thus the intensional information concerning the generation process can not be accessed.

In our current implementation, when attempting to prove a statement that mentions a value in a choice sequence that is not yet registered in the library, applications of the basic computation rules for that object will fail. The user then has to fill in enough of the sequence into the library in order to be able to complete the proof. As discussed above, it is possible to build a way to generate such values automatically on top of the current implementation.

4 Beth-Style System

The BHK/realizability/Curry-Howard Isomorphism semantics are interpretations of intuitionistic logic that make explicit its computational power and its connection to programming languages. While intuitionistic logic clearly holds computational content, this is not as evident in other well-known interpretations of it, such as the possible-world semantics, either Kripke semantics [31] or Beth semantics [21]. These two types of semantics are mainly used for the theoretical foundational exploration of intuitionistic logic, but their relationship to programming concepts is not clear. We next combine these interpretations in a way that fleshes out the computational interpretation of the latter as well.

Since choice sequences evolve *dynamically* as more and more values are recorded in the library, supporting reasoning about them compels modifications to the semantics of Nuprl. Accordingly, this section describes the modified semantics of BITT, and its resulting type system. As choice sequences are implemented as entries in the library, the notion of a truth of a sequent must now also depend on the current state of the library, allowing our libraries to expend (under certain restrictions). Thus, the libraries essentially behave as the worlds in the possible-world semantics, where in any particular state of the library the semantic is induced by the realizability semantics. This provides a computational interpretation for the possible-world semantics in terms of libraries.

4.1 Modifying the Semantics

To support the evolving nature of the library, in [40] the semantics of sequents in Nuprl was modified into a Kripke-like semantics. There, the semantics of types and sequents was parametrized by a library, and then constrained so that a sequent is true in a library only if it is true in *all possible extensions* of the current library. Nevertheless, such generalization of the semantics is insufficient to support choice sequences. To demonstrate the problem, consider the claim "there is some value in a given place of a choice sequence" (e.g., formally, $\exists x.a(100) = x$). This should be a valid statement in the theory of choice sequences, based on their "infinitely proceeding" nature. However, if in the current stage of the library the choice sequence a has only three values, this will be false under the Kripke-like semantics, since there are extensions of the library in which the 100th value is yet to be filled in. Settling this requires further generalization of Nuprl's semantics, into a Beth-like one.

In both Kripke and Beth semantics, if a sentence is true on the basis of a given state of knowledge it will also be asserted to be true in later states of knowledge. The difference appears on how it is asserted to be true. For example, intuitively, in Kripke models an object is said to exist in a given state if it exists in *all* later states of knowledge, whereas in Beth models it exists if in any path through the states of knowledge starting from the given one there exists a point from which on the object exists. Normally in a constructive setting something exists only if it has been constructed. In contrast, using the Beth semantics, roughly speaking, one gets to assert the existence of objects by forcing them to "eventually" exist, without really constructing them. (The Beth semantics bears some resemblance to the notion of forcing [14].)

In [19] Beth models were used to validate the axioms of the theory of lawless sequences (and the axioms of the theory of the creative subject, Bar Induction and the Continuity Principle). Inspired by this work in BITT we have turned Nuprl's PER semantics into a Beth-like model. The fundamental component in Beth semantics is that of a bar. Roughly speaking, a bar b for a world w is a subset of the collection of worlds such that each path through w intersects it (see Def. 4.1 for a precise definition). The key difference from the more well-known concept of Kripke models lays in the definitions of the disjunction and the existential quantifier, which now depend on bars. Those are defined as follows:

- $\varphi \vee \psi$ holds in a world w if there is a bar b for w, such that for each $w' \in b$, φ holds in w' or ψ holds in w'.
- $\exists x.\varphi$ holds in a world w if there is a bar b for w, such that for each $w' \in b$, there exists an element d in the domain for which $\varphi(d)$ holds in w'.

Under this semantics the statement $\exists x.a(100) = x$ is valid because there is a *bar* b of the current library such that in every library in b, the 100th element of the sequence a is filled in, and from that point on it remains fixed (see Sec. 5 for more details regarding the validity of this statement). This demonstrates the imperativeness of the Beth semantics, and in particular of the notion of bars.

4.2 Bar Hopping

To define the key concept of a bar we first introduce the notion of infinite libraries. They are concretely implemented as functions from numbers to infinite library entries, where choice sequences have all their slots filled in. We also require that every infinite library has to have en entry for every named choice sequence.

Definition 4.1 (Bars)**.** A bar of a library lib is a collection of libraries b such that for each infinite library $ilib$ that extends lib, there exists a finite library lib' in b such that $\texttt{extends}(lib', lib)$ and $ilib$ extends lib'. We use $b \in \texttt{Bar}(lib)$ to denote that b is a bar of lib, in which case we also say that b *bars* lib.

Intuitively, being a bar of a library lib means that any way in which lib can be extended always hits the bar, i.e. there is always an intermediate extension of lib that is in the bar. For any library lib there exists a trivial bar containing only lib, i.e., $\{lib\}$ bars lib. Also, it follows from the definition that bars are non-empty.

We use the following abstractions below to interpret types:

$$\texttt{allExt}(lib, f) = \forall lib'.\ \texttt{extends}(lib', lib) \Rightarrow f(lib')$$
$$\texttt{allBar}(b, f) = \forall lib' \in b.\ \texttt{allExt}(lib', f)$$
$$\texttt{exBar}(lib, f) = \exists b \in \texttt{Bar}(lib).\ \texttt{allBar}(b, f)$$

$\texttt{allExt}(lib, f)$ states that all extensions of lib satisfy f; $\texttt{allBar}(b, f)$ states that all extensions of all libraries in b satisfy f; and finally $\texttt{exBar}(lib, f)$ states that there exists a bar of lib such that all extensions of libraries in it satisfy f, i.e. that f holds from that bar on.

We next introduce some useful operations on bars.

Lemma 4.2 (Intersection of bars)**.** *Let b_1 and b_2 be two bars of a library lib. Then, the following collection is also a bar of lib:*

$$b_1 \cap b_2 = \{lib' \mid \exists lib_1 \in b_1, lib_2 \in b_2.\texttt{extends}(lib', lib_1) \wedge \texttt{extends}(lib', lib_2)\}$$

Note that the intersection $b_1 \cap b_2$ builds a monotone bar, in the sense that if a library is in the bar, then all its finite extensions are also in the bar. Because types and PERs are interpreted in terms of existence of bars, we constantly need to intersect bars to prove properties about those. For example, to prove transitivity of the BAR operator defined in Sec. 4.3, because given two different bars we had to compute a third one that covers both of them.

Lemma 4.3 (Raising bars)**.** *Let b be a bar of a library lib, and lib_0 be another library. Then, the following collection is a bar of both lib_0 and b (in the sense that it bars every library in b):*

$$b \uparrow^{lib_0} = \left\{lib' \mid \exists lib'' \in b.\texttt{extends}(lib', lib'') \wedge \texttt{extends}(lib', lib_0)\right\}$$

$b \uparrow^{lib_0}$ is essentially a simple intersection, where one bar is b and the other one is the trivial bar $\{lib_0\}$. A prototypical example of the use of the bar raising operator arises in the proof of Thm. 4.14 below, where from a bar of a library lib and an extension lib' of that library, we need to construct a bar of lib'.

Lemma 4.4 (Families of bars)**.** *Let f be a family of bars of lib, i.e., a function from extensions of a library lib to corresponding bars. Then, the following collection is a bar of lib:*

$$\bigcup f = \left\{lib' \mid \exists lib''.\texttt{extends}(lib'', lib) \wedge lib' \in \texttt{Bar}(f(lib''))\right\}$$

$\bigcup f$ is an infinite intersection. It is useful, among other things, to collapse/expand bars.

Lemma 4.5 (Collapsing/expanding bars)**.** *For a given library lib,* $\texttt{exBar}(lib, \lambda lib'.\texttt{exBar}(lib', f))$ *is equivalent to* $\texttt{exBar}(lib, f)$.

The above lemma allows us to: (1) collapse two layers of bars, i.e. a bar b_1 and a bar b_2 that bars every library in b_1, into one (the $\Rightarrow$ direction); and (2) expand a bar to two layers of bars (the $\Leftarrow$ direction). Collapsing bars is used to simplify definitions that accumulate bars, while expanding bars gives us some leeway in proving the existence of bars in the context of several barred propositions.

4.3 Type Semantics

Let us now describe our Beth model, where types are interpreted as PERs on closed terms. This section culminate in the definition of the BITT type system, which in turn is used to formally define type equality $,T \equiv_{lib} T'$, and equality in a type $,a \equiv_{lib} b \in T$. Those are here parameterized by a library unlike the ones discussed in Sec. 2.

A type system τ is a 4-ary relation between a library lib, two closed terms T and T', and a binary relation ϕ on closed terms, which expresses when T and T' are equal types, and defines ϕ as the PER of those types in lib.[7] As is standard practice, to define BITT below we first define operators that interpret the various type constructor of the type theory. We then recursively define an hierarchy of universes by closing lower universes under the type constructors of the theory. Finally, BITT is the collection of all universes closed under the type constructors of the theory.

The PER model described in [2; 3; 18; 5] is modified so that expressions need only be defined in a bar of the current library.[8] For example, until now the integer type was interpreted as follows:

$$\texttt{INT}(\tau)(lib, T, T', \phi) = T \Downarrow_{lib} \mathbb{Z} \wedge T' \Downarrow_{lib} \mathbb{Z} \wedge (\phi \iff \texttt{INTper}(lib))$$

where τ is a type system; ϕ is a binary relation on closed terms; $\phi_1 \iff \phi_2$ stands for $\forall t, t'.\ t\ \phi_1\ t' \iff t\ \phi_2\ t'$; $a \Downarrow_{lib} b$ denotes that a computes to b in the library lib; and

$$\texttt{INTper}(lib) = \lambda t, t'.\exists i.\ t \Downarrow_{lib} i \wedge t' \Downarrow_{lib} i$$

This states that T and T' are equal types of the type system τ if they both compute to the integer type $\mathbb{Z}$, and ϕ is $\mathbb{Z}$'s PER, i.e., t and t' are equal members of $\mathbb{Z}$ if they both compute to some integer i.

In our Beth model, INT is defined similarely, using INTperb instead INTper, which incorporates bars into the definition:

Definition 4.6 (Integer type)**.**

$$\texttt{INT}(\tau)(lib, T, T', \phi) = T \Downarrow_{lib} \mathbb{Z} \wedge T' \Downarrow_{lib} \mathbb{Z} \wedge (\phi \iff \texttt{INTperb}(lib))$$

where $\texttt{INTperb}(lib) = \lambda t, t'.\texttt{exBar}(lib, \lambda lib'.t\ \texttt{INTper}(lib')\ t')$.

We applied similar changes to the other type constructors. For example, union types are now interpreted as follows:

Definition 4.7 (Union type)**.**

$$\begin{aligned}\texttt{UNION}(\tau)(lib, T, T', \phi) = \exists \psi_a, \psi_b, A, A', B, B'.\ & T \Downarrow_{lib} A{+}B \wedge T' \Downarrow_{lib} A'{+}B' \\ & \wedge \texttt{allExt}(lib, \lambda lib'.\tau(lib', A, A', \psi_a(lib'))) \\ & \wedge \texttt{allExt}(lib, \lambda lib'.\tau(lib', B, B', \psi_b(lib'))) \\ & \wedge (\phi \iff \texttt{UNIONperb}(lib, \psi_a, \psi_b))\end{aligned}$$

where:

$$\begin{aligned}&\texttt{UNIONperb}(lib, \psi_a, \psi_b) = \\ &\quad \lambda t, t'.\texttt{exBar}(lib, \lambda lib'.\texttt{INLper}(t, t', lib', \psi_a) \vee \texttt{INRper}(t, t', lib', \psi_b)) \\ &\texttt{INLper}(t, t', lib, \psi) = \exists x, y.\ t \Downarrow_{lib} \texttt{inl}(x) \wedge t' \Downarrow_{lib} \texttt{inl}(y) \wedge x\ \psi(lib)\ y \\ &\texttt{INRper}(t, t', lib, \psi) = \exists x, y.\ t \Downarrow_{lib} \texttt{inr}(x) \wedge t' \Downarrow_{lib} \texttt{inr}(y) \wedge x\ \psi(lib)\ y\end{aligned}$$

and ψ denotes a function that associates binary relations on closed terms with libraries.

Note that UNION requires A, A', B, B' to be types in all extensions lib' of lib such that $\psi_a(lib')$ is the PER of A and A', and $\psi_b(lib')$ is the PER of B and B'. This is so that ψ_a and ψ_b can be used to define the PER interpretation of union types in terms of a bar of the current library. They provide the PERs of A and B in all extensions of lib, so that we can define the PER of the union type $A{+}B$ in terms of the existence of objects in the PERs of A and B in a bar of lib, i.e., in extensions of lib, by applying ψ_a and ψ_b to those extensions. Knowing the PER of A and B only in the current library is insufficient for such a construction. In Sec. 4.4 we show that type interpretations are monotonic, therefore that if lib' extends lib then $\psi_a(lib)$ and $\psi_b(lib)$ are subsets of $\psi_a(lib')$ and $\psi_b(lib')$, respectively.

We also add a new constructor, BAR, that assigns meaning to types at a given library lib, provided they are defined in a bar of lib.

[7] Instead of using induction-recursion (not yet fully supported by Coq) to define $T \equiv_{lib} T'$ and $a \equiv_{lib} b \in T$, we use Allen's method [3], and define a 4-ary relation, BITT, between a library, two types and a PER, from which we derive $T \equiv_{lib} T'$ and $a \equiv_{lib} b \in T$.

[8] See https://github.com/vrahli/NuprlInCoq/tree/beth/per/per.v.

This is critical to obtain the locality property of the type system, discussed in Sec. 4.4, which is a salient feature of Beth models.

Definition 4.8. The BAR constructor is defined as follows:

$$\begin{aligned}&\mathtt{BAR}(\tau)(lib, T, T', \phi)\\ &= \exists b{\in}\mathtt{Bar}(lib).\ \exists\psi.\ \mathtt{allBar}(b, \lambda lib'.\tau\ lib'\ T\ T'\ (\psi(lib')))\\ &\qquad\qquad \wedge\ \phi\ \iff\ \mathtt{BARperb}(b,\psi)\end{aligned}$$

where:

$$\mathtt{BARperb}(b,\psi) = \lambda t, t'.\mathtt{allBar}(b, \lambda lib.\mathtt{exBar}(lib, \lambda lib'.t\ \psi(lib')\ t'))$$

We also add a new constructor FREE that assigns meaning to the new Free(n) types:

Definition 4.9. The FREE constructor is defined as follows:

$$\begin{aligned}\mathtt{FREE}(\tau)(lib, T, T', \phi) = \exists n.\ & T \Downarrow_{lib} \mathtt{Free}(n) \wedge T' \Downarrow_{lib} \mathtt{Free}(n)\\ &\wedge (\phi\ \iff\ \mathtt{FREEperb}(lib, n))\end{aligned}$$

where:

$$\begin{aligned}&\mathtt{FREEperb}(lib, n) = \lambda t, t'.\mathtt{exBar}(lib, \lambda lib'.\mathtt{FREEper}(lib', n))\\ &\mathtt{FREEper}(lib', n) = \exists csn.\ t \Downarrow_{lib} \mathtt{seq}(csn) \wedge t' \Downarrow_{lib} \mathtt{seq}(csn) \wedge n\#csn\end{aligned}$$

and where $n\#csn$ states that n is compatible with csn, i.e., $n = 0$ implies that csn's *space* part is either 0 or a list of numbers (in both cases, constraining the choice sequence to consist of numbers).

As in [2; 3; 18; 5], and as explained by Crary [18], we then define a closure operator CLOSE that, given a type system τ, builds a larger type system from the types in τ (e.g., INT and BAR) using any type constructors except universes.[9] Intuitively, if τ contains all universes up to some universe $\mathbb{U}_i$, then CLOSE(τ) contains all types built from those universes, i.e., all members of $\mathbb{U}_{i+1}$.

Definition 4.10. CLOSE is the smallest type system such that:

$$\begin{aligned}&\mathtt{CLOSE}(\tau)(lib, T, T', \phi)\ \iff\\ &\left(\begin{array}{l}\tau(lib, T, T', \phi) \vee \mathtt{INT}(\tau)(lib, T, T', \phi) \vee \mathtt{UNION}(\tau)(lib, T, T', \phi)\\ \vee\ \mathtt{FREE}(\tau)(lib, T, T', \phi) \vee \mathtt{BAR}(\tau)(lib, T, T', \phi) \vee \cdots\end{array}\right)\end{aligned}$$

where $\tau(lib, T, T', \phi)$ states that T and T' are equal types in the type system τ, with PER ϕ in the library lib; and the rest of the disjunction contains the other type constructors, excluding universes.

Next, we define for every natural number i the type system UNIVi(i) containing all universes up to some level i by induction on i, and then use those to define the type system UNIV containing all universes as follows:

Definition 4.11.

$$\begin{aligned}&\mathtt{UNIVex}(lib, T, T', \phi) = \exists i.\ \mathtt{UNIVi}(i)(lib, T, T', \phi)\\ &\mathtt{UNIV}(lib, T, T', \phi) = \mathtt{BAR}(\mathtt{UNIVex})(lib, T, T', \phi)\end{aligned}$$

Finally, we define the BITT type system, from which we derive the $T{\equiv}_{lib}T'$ and $a{\equiv}_{lib}b{\in}T$ relations:

Definition 4.12 (BITT type system)**.**

$$\begin{aligned}&\mathtt{BITT} = \mathtt{CLOSE}(\mathtt{UNIV})\\ &T{\equiv}_{lib}T' = \exists\phi.\ \mathtt{BITT}(lib, T, T', \phi)\\ &a{\equiv}_{lib}b{\in}T = \exists\phi.\ \mathtt{BITT}(lib, T, T, \phi) \wedge a\ \phi\ b\end{aligned}$$

The definitions presented in this section differ from the ones in [2; 3; 18; 5] in two ways: (1) they depend on a library; and (2) universes are closed using the BAR in order to guarantee that the type system satisfies the locality property discussed in Sec. 4.4.

[9] Our formalization currently includes sum types, pi types, equality types, choice sequence types, integer types, approximation and computational equivalence types, base types, name types, set types, image types, union types.

4.4 Type System Properties

We start by showing that BITT satisfies the key properties of a type system [2; 3; 18; 5].

Theorem 4.13 (Type system properties)**.** BITT *satisfies the following properties (free variables are universally quantified):*

Uniqueness: $\mathtt{BITT}(lib, T, T', \phi) \Rightarrow \mathtt{BITT}(lib, T, T', \phi') \Rightarrow (\phi\ \iff\ \phi')$
Extensionality: $\mathtt{BITT}(lib, T, T', \phi) \Rightarrow (\phi\ \iff\ \phi') \Rightarrow \mathtt{BITT}(lib, T, T', \phi')$
Type transitivity:
$\mathtt{BITT}(lib, T_1, T_2, \phi) \Rightarrow \mathtt{BITT}(lib, T_2, T_3, \phi) \Rightarrow \mathtt{BITT}(lib, T_1, T_3, \phi)$
Type symmetry: $\mathtt{BITT}(lib, T, T', \phi) \Rightarrow \mathtt{BITT}(lib, T', T, \phi)$
Type computation:
$\mathtt{BITT}(lib, T, T, \phi) \Rightarrow \mathtt{allExt}(lib, \lambda lib'.T \sim_{lib'} T') \Rightarrow \mathtt{BITT}(lib, T, T', \phi)$
Term transitivity: $\mathtt{BITT}(lib, T, T', \phi) \Rightarrow t_1\ \phi\ t_2 \Rightarrow t_2\ \phi\ t_3 \Rightarrow t_1\ \phi\ t_3$
Term symmetry: $\mathtt{BITT}(lib, T, T', \phi) \Rightarrow t\ \phi\ t' \Rightarrow t'\ \phi\ t$
Term computation:
$\mathtt{BITT}(lib, T, T', \phi) \Rightarrow t\ \phi\ t \Rightarrow \mathtt{allExt}(lib, \lambda lib'.t \sim_{lib'} t') \Rightarrow t\ \phi\ t'$

Uniqueness ensures that BITT uniquely defines PERs (up to extensional equality—see *Extensionality*). All four transitivity and symmetry properties ensure that the relations $T{\equiv}_{lib}T'$ and $a{\equiv}_{lib}b{\in}T$ derived from BITT are partial equivalence relations. Finally, the two computation properties ensure that $T{\equiv}_{lib}T'$ and $a{\equiv}_{lib}b{\in}T$ respect Howe's computational equivalence relation.

The above properties are similar to those presented in [2; 3; 18; 5], except here we use $\mathtt{allExt}(lib, \lambda lib'.t \sim_{lib'} t')$ instead of $t \sim_{lib} t'$ in the properties about computation. This is to enforce that the semantics is monotonic as discussed below. To see why this is necessary, consider a library lib that contains a choice sequence entry a whose 5th element has not yet been chosen. Then, $a(5)$ is computationally equivalent to $\bot$ w.r.t. lib (since both do not compute to values), but it is not computationally equivalent to $\bot$ w.r.t. some extension of lib that defines $a(5)$ to be, say, 0.

Proof outline. The main challenge in proving those properties is to show that the CLOSE operator preserves them, which we prove by induction on CLOSE. Since CLOSE is closed under bars using the BAR operator, it is helpful to use the locality property discussed below, and therefore we prove those properties by mutual induction.[10] □

The two unique properties of possible-world semantics, and thus of our new type system, are monotonicity and locality. While the former is a general feature of possible-world semantics, the second is a distinctive feature of Beth models. Monotonicity ensures that true facts remain true in the future, and locality allows one to deduce a fact about the current library if it is true in a bar of that library. Given the aforementioned interpretation of types, it is then straightforward to prove BITT's monotonicity w.r.t. libraries. As opposed to locality which is proven by mutual induction, monotonicity can be proved independently.[11]

Theorem 4.14 (Monotinicity)**.**

$$\begin{aligned}\mathtt{BITT}(lib, T, T', \phi) \Rightarrow \exists\psi.\ \forall lib'.\ &\mathtt{extends}(lib', lib)\\ &\Rightarrow \mathtt{BITT}(lib', T, T', \psi(lib'))\\ &\quad \wedge \phi \sqsubseteq \psi(lib) \wedge \mathtt{monPer}(lib', \psi)\end{aligned}$$

where $\phi_1 \sqsubseteq \phi_2$ *stands for* $\forall t, t'.\ t\ \phi_1\ t' \Rightarrow t\ \phi_2\ t'$, *and* $\mathtt{monPer}(lib', \psi)$ *stands for* $\forall lib'.\ \mathtt{extends}(lib', lib) \Rightarrow \psi(lib) \sqsubseteq \psi(lib')$.

[10] See https://github.com/vrahli/NuprlInCoq/tree/beth/per/nuprl_type_sys.v.
[11] See https://github.com/vrahli/NuprlInCoq/tree/beth/per/nuprl_mon_func2.v.

Thanks to the BAR constructor, it is also straightforward to prove BITT's locality:[12]

Theorem 4.15 (Locality).

$$\texttt{allBar}(b, \lambda lib'.\texttt{BITT}(lib', T, T', \psi(lib'))) \Rightarrow \texttt{BITT}(lib, T, T', \texttt{BARperb}(b, \psi))$$

4.5 Characterization Lemmas and Inference Rules

Once we have proved that BITT satisfies the above desired properties, we can provide characterization lemmas for each type constructor which describe when two given types are equal, and when two values are equal in a that type. We then use those characterization lemmas to validate BITT's inference rules. We here provide, as an example, the characterization lemmas for the union types.

Lemma 4.16. *The following two equivalences are provable:*

$$A{+}B \equiv_{lib} A'{+}B' \iff (A \equiv_{lib} A' \wedge B \equiv_{lib} B')$$
$$a \equiv_{lib} b \in A{+}B \iff \left(\begin{array}{l} A \equiv_{lib} A \wedge B \equiv_{lib} B \\ \wedge\ \texttt{exBar}(lib, \lambda lib'.\texttt{INLeq}(a, b, lib', A) \vee \texttt{INReq}(a, b, lib', B)) \end{array}\right)$$

where

$$\texttt{INLeq}(t, t', lib, T) = \exists x, y.\ a \Downarrow_{lib} \texttt{inl}(x) \wedge b \Downarrow_{lib} \texttt{inl}(y) \wedge x \equiv_{lib} y \in T$$
$$\texttt{INReq}(t, t', lib, T) = \exists x, y.\ a \Downarrow_{lib} \texttt{inr}(x) \wedge b \Downarrow_{lib} \texttt{inr}(y) \wedge x \equiv_{lib} y \in T$$

Proof outline. For the $\Rightarrow$ direction of the first equivalence, we have to show, among other things, that if $A{+}B \equiv_{lib} A'{+}B'$ then $A \equiv_{lib} A'$. Because the $T \equiv_{lib} T'$ relation is defined in terms of BITT which is, in turn, defined in terms of CLOSE, and because the CLOSE operator is closed under bars using BAR (which is necessary to obtain locality), from $A{+}B \equiv_{lib} A'{+}B'$ we obtain that $A{+}B$ and $A'{+}B'$ are equal in a bar of *lib*. This entails that A and A' are equal in a bar of *lib*, from which, using BITT's locality, we obtain $A \equiv_{lib} A'$.

To prove the $\Leftarrow$ direction of the first equivalence, we show that $A{+}B \equiv_{lib} A'{+}B'$ follows from $A \equiv_{lib} A'$ and $B \equiv_{lib} B'$. To prove this, we have to prove that A, A', B, and B' are types in all extensions *lib*, as required by UNION (defined in Sec. 4.3). We derive that from $A \equiv_{lib} A'$ and $B \equiv_{lib} B'$, and from the monotonicity of BITT.[13] □

We use these characterization lemmas to validate introduction and elimination rules for BITT's types, such as the following introduction rule for union types, which states that if a is a member of A (and B is a type) then $\texttt{inl}(a)$ is a member of $A{+}B$:[14]

$$\frac{H \vdash A\ \lfloor \texttt{ext}\ a \rfloor \qquad H \vdash B \in \mathbb{U}_i}{H \vdash A{+}B\ \lfloor \texttt{ext}\ \texttt{inl}(a) \rfloor}$$

In addition to proving the validity of such rules, we have also proved that BITT is weakly consistent w.r.t. Coq's consistency, in the sense that the proposition False is not derivable.[15]

5 Non-Computable Functions Type

Now that choice sequences are integrated into the system, in this section and the next one we demonstrate the adequacy of the implementation for the theory of choice sequences. Accordingly, we prove the validity of inference rules and axioms concerning choice sequences. We do so using the Coq formalization of BITT. Thus, a rule or an axiom is said to 'hold in BITT' if it was formally validated using the metatheory developed in this paper.

This section shows how BITT's function type (i.e. non-dependent product type) extends Nuprl's. To extend the notion of computability, the choice sequences of numbers are incorporated into the function type $\mathbb{N} \to \mathbb{N}$ (also called the Baire space, $\mathcal{B}$). This is possible since the Nuprl system (on which BITT is based) was never limited to assume that function types contain *only* computable (recursive) functions. We have validated the following rule in BITT, which asserts that all choice sequences are in the Baire space.

Proposition 5.1. *The following holds in BITT:*

$$\frac{H \vdash f \in \texttt{Free}(0)}{H \vdash f \in \mathcal{B}}$$

Proof outline. Let *lib* be the current library, f a free choice sequence of numbers (because of the use of 0 in $\texttt{Free}(0)$) with name *csn*, and n a natural number. We have to prove that $f(n)$ is in $\mathbb{N}$. To prove that, it is enough to pick a bar b of *lib* such that $f(n)$ computes to a number in b. This bar is simply the library *lib* extended so as to contain at least n values for the choice sequence named *csn*.[16] □

Considering choice sequences as functions might seem odd. Nevertheless, recalling the standard mathematical definition of a function as a single-valued relation demonstrates that there is no a priori assumption of a governing law. So the "free choice" principe is not in any contradiction to the abstract notion of a function. As for the "infinitely proceeding" property, a choice sequence might be thought of as a partial function with an undefined tail. This is also compatible with Nuprl which allows for partial functions, albeit in a somewhat different notion. The partialness of a choice sequence is a consequence of the fact that at any given stage of the library there is a tail of the sequence that is not yet defined. However, as opposed to the standard concept of partial functions, the partialness of a choice sequence is local. That is, a choice sequence has the guarantee that all of its values will get filled "eventually".

Proposition 5.1, in turn, allows us to prove simple facts about choice sequences of numbers. For example, we can prove a generalized version of the example given in Sec. 4.1:[17]

$$\forall \alpha : \texttt{Free}(0).\ \forall n : \mathbb{N}.\ \exists x : \mathbb{N}.\ a(n) =_{\mathbb{N}} x$$

Both choice sequences and recursive computable functions inhabit the $\mathcal{B}$ type because the meaning of a type is determined by its operations, and the only operation on a function type is *apply*. As mentioned in Sec. 3, we already modified the behavior of *apply* by changing the computation system to allow applying choice sequences to numbers to access already made choices. Because the computation system is "externalized" through inference rules which we use to reason about computation, such as [ApplyCases] presented below, those also need to be adjusted.

In [39] the inference rule [ApplyCases] already had to be modified to include the metatheoretical possibility that $f(a)$ might compute to a value also in case f is a choice sequence, not only if it computes to a λ-term. The modification of the rule was as follows:

$$\frac{H \vdash \texttt{halts}(f(a)) \qquad H \vdash f \in \texttt{Base}}{H \vdash f \simeq \lambda x.f(x)\ \vee\ \texttt{isChoiceSeq}(x, z, f)\ \lfloor \texttt{ext}\ \textbf{iflam}(f, \texttt{tt}, \texttt{ff}) \rfloor}$$

where $\texttt{isChoiceSeq}(x, z, f)$ only stated that f diverges on non-integer inputs. This modification was required so to *not exclude* choice sequences, even though in the theoretical syntax of Nuprl

[12] See https://github.com/vrahli/NuprlInCoq/tree/beth/per/nuprl_props.v.
[13] See https://github.com/vrahli/NuprlInCoq/tree/beth/per/per_props_union.v.
[14] See https://github.com/vrahli/NuprlInCoq/tree/beth/rules/rules_union.v.
[15] See https://github.com/vrahli/NuprlInCoq/tree/beth/per/weak_consistency.v.
[16] See https://github.com/vrahli/NuprlInCoq/tree/beth/rules/rules_choice.v.
[17] See https://github.com/vrahli/NuprlInCoq/tree/beth/rules/rules_choice3.v.

$f \simeq \lambda x.f(x)$ would always hold. This is not the case anymore. Because we *include* choice sequences in BITT's syntax, the right-disjunct of the conclusion is now plausible. Therefore we refine the predicate `isChoiceSeq` in a way that captures the computation of choice sequences in a more precise way. We do so by replacing $\texttt{isChoiceSeq}(x, z, f)$ with: $\texttt{isChoiceSeq}(f) = f \in \texttt{Free}(1)$, where 1 is used here to express that there are no restrictions on the choice sequence f—its choices do not have to be numbers.[18]

6 Validating the Axioms for Choice Sequences

In this section we validate in BITT key properties governing choice sequences that have been suggested in the literature. We focus on choice sequences of natural numbers, i.e. members of $\texttt{Free}(0)$. We adopt van Dalen's formalization of the three axioms for choice sequences given in [19], which are due to Kreisel [28]. We have found that these are essentially at the intersection of the various choice sequence theories.

6.1 Extending Initial Segments

The axiom named LS1 in [19], states that for any finite list of values l, there is a choice sequence that extends it, i.e. one that agrees with l on its first $|l|$ values. LS1 is a statement about the universe of choice sequences. Intuitively, it promises that there are *enough* choice sequences. This is the only existential axiom for choice sequences. We have validated a simple squashed (see Sec. 2) version of LS1 in BITT, which we present in Prop. 6.1. In addition, as discussed in Appx. C, we have validated a more involved non-squashed version of LS1, which we do not discuss here for space reasons, and which required extending BITT with computations to generate a choice sequence name $a \in \texttt{Free}(0)$ given a finite sequence of numbers, provided as a pair of a number $n \in \mathbb{N}$ and a function $f \in \mathcal{B}_n$ (see Prop. 6.1). Using the $\downarrow$ operator, however, allows us to compute this choice sequence in the metatheory.

Proposition 6.1 (Extending initial segments)**.** *The following holds in BITT (where $\mathcal{B}_n = \mathbb{N}_n \to \mathbb{N}$ for $\mathbb{N}_n = \{k : \mathbb{N} \mid k < n\}$):*

$$\forall n : \mathbb{N}.\ \forall f : \mathcal{B}_n.\ \downarrow\exists a : \texttt{Free}(0).\ f =_{\mathcal{B}_n} a$$

Because this proposition is squashed, its extract is simply $\lambda n, f.\star$, i.e., it does not have any computational content.

Proof outline. Let n be a Nuprl term that inhabits $\mathbb{N}$, and f be a Nuprl term that inhabits $\mathcal{B}_n$. In the metatheory we can build a Coq list l of Coq numbers such that for all $m < n$, $f(m)$ computes to the mth number in l. Using this list we build a choice sequence cs that includes l in its name so as to enforce that its n first values have to match the values in l (as explained in Sec. 3.2). Then, cs is added to the current library lib_0, and its n first values are filled. This forms a bar of lib_0 due to the restriction on libraries that enforces that an initial segment provided in a choice sequence name has to be respected in extensions. Finally, we use this bar to prove $\downarrow\exists a : \texttt{Free}(0).\ f =_{\mathcal{B}_n} a$. Note that we need to fill the n first values of the cs choice sequence to ensure that we can prove $f =_{\mathcal{B}_n} cs$. We can start using this bar either when proving the existential, or later when proving the equality.[19] □

Note the use of name spaces in the above proof. Those were introduced to Nuprl for exactly this purpose, i.e. in order to be able to guarantee the existence of a specific choice sequence *in principle*, without having to actually add it to the library.

6.2 Decidability of Equality

The axiom named LS2 in [19] states that intensional equality over choice sequences is decidable. As for LS1, this is an axiom about the universe of choice sequences. In general, since choice sequences are identified with their names in the library, any two different entries of choice sequences are computationally distinct. Accordingly, we have validated the following versions of LS2 in BITT (see Appx. D for more details):

Proposition 6.2 (Decidability of equality)**.** *The following intensional[20] and extensional[21] versions of LS2 hold in BITT:[22]*

$$\forall a, b : \texttt{Free}(0).\ a \simeq b \ \vee\ \neg a \simeq b$$
$$\forall a, b : \texttt{Free}(0).\ a =_{\mathcal{B}} b \ \vee\ \neg a =_{\mathcal{B}} b$$

and are both inhabited by the term: $\lambda a, b.$ **if** $a{=}b$ **then** `tt` **else** `ff`.

6.3 The Axiom of Open Data

As opposed to LS1 and LS2, which characterize the universe of choice sequences, the axiom named LS3(1) in [19] (also known as the "Axiom of Open Data" [44]) is concerned with ways in which they are to be reasoned about. It states that if a property φ (with certain side-conditions) holds for a choice sequence a, then there exists a finite initial segment of a such that φ holds for all choice sequences that agree with a on this initial segment. In other words, the validity of $\varphi(a)$ depends only on a finite initial segment of a.

LS3(1) is a generalized form of the Continuity Principle. Following [29, p.154; 45, Thm.IIA; 22], we have already shown that the non-squashed Continuity Principle is incompatible with Nuprl. (However, we have proved in [38] that squashed versions of the Continuity Principle are consistent with Nuprl.) Therefore, we will only be able to validate a squashed version of LS3(1). Using the $\downarrow$ squashing operator, this can be formulated as follows (where a is the only choice sequence in $\varphi(a)$):

$$\forall a : \texttt{Free}(0).\ \varphi(a) \implies \downarrow\exists n : \mathbb{N}.\ \forall b : \texttt{Free}(0).\ (a =_{\mathbb{N}_n} b \implies \varphi(b))$$

The informal justification for this claim in our implementation of choice sequences seems straightforward. In any concrete stage of the library, it only contains a finite initial segment of a. Thus, if at a certain state of the library we managed to deduce that a satisfies a certain property, the same conclusion should be inferred for any other choice sequence that shares that same finite information.

Formally validating it in BITT entails first internalizing the constraint on φ. This could be done using a nominal mechanism (such as in [1]) which checks for names appearing in φ. Assuming that, validating the axiom in our implementation of choice sequences turns out to amount to some key properties about the behavior of libraries, namely, monotonicity and name-invariance. Those were shown to hold for Nuprl in [40], however, proving they hold in BITT, and therefore also validating LS3(1), is left for future work.

[18] See https://github.com/vrahli/NuprlInCoq/tree/beth/rules/rules_apply.v.
[19] See https://github.com/vrahli/NuprlInCoq/tree/beth/rules/rules_choice.v.
[20] See https://github.com/vrahli/NuprlInCoq/tree/beth/rules/rules_choice2.v.
[21] See https://github.com/vrahli/NuprlInCoq/tree/beth/rules/rules_choice5.v.
[22] Recall that $\simeq$ denotes the theoretical counterpart of the metatheoretical relation $\sim$. Here, $a \simeq b$ means that a and b compute to the same choice sequence.

7 Conclusions

We have developed an extension of Nuprl's type theory, called BITT, which incorporates choice sequences. Next, we plan to update Nuprl accordingly, thereby turning it into a truly *intuitionistic* proof assistant. More so, it will be the only one, as far as we know, which supports any form of non-determinism. As such, we strongly believe this new version of Nuprl could be used to model large distributed systems. Investigating this application, as well as the exploration of others, is left for future work.

Another future research task is to investigate the foundational intuitionistic implications of this new type theory, namely : spreads, Bar Induction, and the Continuity Principle (CP). For example, CP was proven in Nuprl using exceptions [39]. In [13] the authors formalized choice sequences as monadic streams and internally proved CP for natural monadic stream functions. It has also been shown that one can use references to obtain CP [32]. We conjecture that our implementation of choice sequences can be used instead of the methods above to realize CP.

It is also interesting to determine the status of other well known principles in the new type theory, such as Markov's principle (MP) and Kripke's Scheme (KS). MP has recently been studied in the context of type theory [17; 16]. In particular, [16] established the independence of MP with Martin-Löf's type theory. MP was shown to be consistent with Nuprl (using a squashed form of excluded middle), however it was also shown in Nuprl, following [9, p.116; 44, Ch.4,Sec.9.5], that MP is inconsistent with KS [39, Appx.H]. Now, in [19] van Dalen used Beth models to validate KS. Given that we now invoke a Beth semantics, the status of these principles and their connection has to be settled. As discussed in Appx. B, we have so far proved that MP is false in BITT.

Another direction for future work is to determine whether BITT exhibits versions of the Axiom of Choice (AC). Berardi et al. [8] proposed an interpretation of classical analysis with AC, where the negative translation of AC is realized by a bar recursion-like operator. They achieve this by adding infinite sequences to their term language, which can be seen as lawlike choice sequences. Herbelin showed in [23] how to realize the Axiom of Countable Choice, the Axiom of Dependent Choice, and Bar Induction in a classical logic with strong (computational) existential called dPA^{ω}. Herbelin used steams, which can also be seen as some form of lawlike choice sequences, to compute the witnesses of the existential formulas in these axioms. Miquey [34] refined this work and proved the strong normalization (therefore also soundness) of a variant of dPA^{ω} presented as a sequent calculus. Using the current implementation, we aim to improve on previous results and derive squashed versions of the Axiom of Countable Choice directly in Nuprl, without using classical logic.

References

[1] Stuart Allen. *An Abstract Semantics for Atoms in Nuprl.* Tech. rep. Cornell University, 2006.
[2] Stuart F. Allen. "A Non-Type-Theoretic Definition of Martin-Löf's Types". In: *LICS*. IEEE Computer Society, 1987, pp. 215–221.
[3] Stuart F. Allen. "A Non-Type-Theoretic Semantics for Type-Theoretic Language". PhD thesis. Cornell University, 1987.
[4] Stuart F. Allen, Mark Bickford, Robert L. Constable, Richard Eaton, Christoph Kreitz, Lori Lorigo, and Evan Moran. "Innovations in computational type theory using Nuprl". In: *J. Applied Logic* 4.4 (2006). http://www.nuprl.org/, pp. 428–469.
[5] Abhishek Anand and Vincent Rahli. "Towards a Formally Verified Proof Assistant". In: *ITP 2014*. Vol. 8558. LNCS. Springer, 2014, pp. 27–44.
[6] Mark van Atten. *On Brouwer*. Wadsworth Philosophers. Cengage Learning, 2004.
[7] Mark van Atten and Dirk van Dalen. "Arguments for the continuity principle". In: *Bulletin of Symbolic Logic* 8.3 (2002), pp. 329–347.
[8] Stefano Berardi, Marc Bezem, and Thierry Coquand. "On the Computational Content of the Axiom of Choice". In: *J. Symb. Log.* 63.2 (1998), pp. 600–622.
[9] Douglas Bridges and Fred Richman. *Varieties of Constructive Mathematics*. London Mathematical Society Lecture Notes Series. Cambridge University Press, 1987.
[10] L. E. J Brouwer. "Begründung der mengenlehre unabhängig vom logischen satz vom ausgeschlossen dritten. zweiter teil: Theorie der punkmengen". In: *Koninklijke Nederlandse Akademie van Wetenschappen te Amsterdam* 12.7 (1919).
[11] L. E. J. Brouwer. *Brouwer's Cambridge Lectures on Intuitionism*. Edited by D. Van Dalen. Cambridge University Press, 1981, pp. 214–215.
[12] L. E. J. Brouwer. "Historical Background, Principles and Methods of Intuitionism". In: *Journal of Symbolic Logic* 19.2 (1954), pp. 125–125.
[13] Venanzio Capretta and Jonathan Fowler. "The continuity of monadic stream functions". In: *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. June 2017, pp. 1–12.
[14] Paul J. Cohen. "The independence of the continuum hypothesis". In: *the National Academy of Sciences of the United States of America* 50.6 (Dec. 1963), pp. 1143–1148.
[15] Robert L. Constable, Stuart F. Allen, Mark Bromley, Rance Cleaveland, J. F. Cremer, Robert W. Harper, Douglas J. Howe, Todd B. Knoblock, Nax P. Mendler, Prakash Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing mathematics with the Nuprl proof development system*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1986.
[16] Thierry Coquand and Bassel Mannaa. "The Independence of Markov's Principle in Type Theory". In: *FSCD 2016*. Vol. 52. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, 17:1–17:18.
[17] Thierry Coquand, Bassel Mannaa, and Fabian Ruch. "Stack semantics of type theory". In: *LICS 2017*. IEEE Computer Society, 2017, pp. 1–11.
[18] Karl Crary. "Type-Theoretic Methodology for Practical Programming Languages". PhD thesis. Ithaca, NY: Cornell University, Aug. 1998.
[19] Dirk van Dalen. "An interpretation of intuitionistic analysis". In: *Annals of mathematical logic* 13.1 (1978), pp. 1–43.
[20] Michael A. E. Dummett. *Elements of Intuitionism*. Second. Clarendon Press, 2000.
[21] VH Dyson and Georg Kreisel. *Analysis of Beth's semantic construction of intuitionistic logic*. Stanford University. Applied Mathematics and Statistics Laboratories, 1961.
[22] Martín H. Escardó and Chuangjie Xu. "The Inconsistency of a Brouwerian Continuity Principle with the Curry-Howard Interpretation". In: *TLCA 2015*. Vol. 38. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015, pp. 153–164.
[23] Hugo Herbelin. "A Constructive Proof of Dependent Choice, Compatible with Classical Logic". In: *LICS 2012*. IEEE, 2012, pp. 365–374.
[24] Douglas J. Howe. "Equality in Lazy Computation Systems". In: *LICS 1989*. IEEE Computer Society, 1989, pp. 198–203.
[25] Hajime Ishihara and Peter Schuster. "A continuity principle, a version of Baire's theorem and a boundedness principle". In: *J. Symb. Log.* 73.4 (2008), pp. 1354–1360.
[26] Stephen C. Kleene and Richard E. Vesley. *The Foundations of Intuitionistic Mathematics, especially in relation to recursive functions*. North-Holland Publishing Company, 1965.
[27] Alexei Kopylov. "Type Theoretical Foundations for Data Structures, Classes, and Objects". PhD thesis. Ithaca, NY: Cornell University, 2004.
[28] Georg Kreisel. "Lawless sequences of natural numbers". In: *Compositio Mathematica* 20 (1968), pp. 222–248.
[29] Georg Kreisel. "On weak completeness of intuitionistic predicate logic". In: *J. Symb. Log.* 27.2 (1962), pp. 139–158.
[30] Georg Kreisel and Anne S. Troelstra. "Formal systems for some branches of intuitionistic analysis". In: *Annals of Mathematical Logic* 1.3 (1970), pp. 229–387.
[31] Saul A. Kripke. "Semantical Considerations on Modal Logic". In: *Acta Philosophica Fennica* 16.1963 (1963), pp. 83–94.
[32] John Longley. "When is a Functional Program Not a Functional Program?" In: *ICFP'99*. ACM, 1999, pp. 1–7.
[33] Per Martin-Löf. *Intuitionistic Type Theory*. Studies in Proof Theory, Lecture Notes 1. Napoli: Bibliopolis, 1984.
[34] Etienne Miquey. "A sequent calculus with dependent types for classical arithmetic". In: *LICS 2018*. 2018.
[35] Joan R. Moschovakis. "An intuitionistic theory of lawlike, choice and lawless sequences". In: *Logic Colloquium'90: ASL Summer Meeting in Helsinki*. Association for Symbolic Logic. 1993, pp. 191–209.
[36] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's type theory: an introduction*. International Series of Monographs on Computer Science. Clarendon Press, 1990.
[37] *Nuprl in Coq*. https://github.com/vrahli/NuprlInCoq.
[38] Vincent Rahli and Mark Bickford. "A nominal exploration of intuitionism". In: *CPP 2016*. Extended version: http://www.nuprl.org/html/Nuprl2Coq/continuity-long.pdf. ACM, 2016, pp. 130–141.
[39] Vincent Rahli, Mark Bickford, and Robert L. Constable. "Bar induction: The good, the bad, and the ugly". In: *LICS 2017*. Extended version avaible at https://vrahli.github.io/articles/bar-induction-lics-long.pdf. IEEE Computer Society, 2017, pp. 1–12.
[40] Vincent Rahli, Liron Cohen, and Mark Bickford. "A Verified Theorem Prover Backend Supported by a Monotonic and Name-Invariant Library". Submitted. 2018.
[41] Michael Rathjen. "A note on Bar Induction in Constructive Set Theory". In: *Math. Log. Q.* 52.3 (2006), pp. 253–258.
[42] Anne S. Troelstra. *Choice sequences: a chapter of intuitionistic mathematics*. Clarendon Press Oxford, 1977.
[43] Anne S. Troelstra. "Choice Sequences and Informal Rigour". In: *Synthese* 62.2 (1985), pp. 217–227.
[44] Anne S. Troelstra and Dirk van Dalen. *Constructivism in Mathematics An Introduction*. Vol. 121. Studies in Logic and the Foundations of Mathematics. Elsevier, 1988.
[45] A.S. Troelstra. "A Note on Non-Extensional Operations in Connection With Continuity and Recursiveness". In: *Indagationes Mathematicae* 39.5 (1977), pp. 455–462.
[46] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: http://homotopytypetheory.org/book, 2013.
[47] Wim Veldman. "Understanding and Using Brouwer's Continuity Principle". In: *Reuniting the Antipodes — Constructive and Nonstandard Views of the Continuum*. Vol. 306. Synthese Library. Springer Netherlands, 2001, pp. 285–302.
[48] Beth E. W. "Semantic Construction of Intuitionistic Logic". In: *Journal of Symbolic Logic* 22.4 (1957), pp. 363–365.

A The Law of Excluded Middle

As explained in [1; 5; 6; 9, Appx.F; 8, Sec.6.3], the non-squashed law of excluded middle (LEM) is false in Nuprl. However, the following ↓-squashed LEM is consistent with Nuprl:

$$H \vdash \downarrow(P+\neg P) \quad \text{BY [LEM]} \quad H \vdash P \in \mathbb{U}_i$$

One can prove that this inference rule is consistent with Nuprl using the law of excluded middle in the metatheoretical proof of its validity w.r.t. Nuprl's PER semantics as shown in https://github.com/vrahli/NuprlInCoq/blob/master/rules/rules_classical.v. This seemingly classical principle is therefore computationally justified in the sense that the conclusion of the rule is inhabited by $\star$. As proved in [6, Thm.4.2], it implies Markov's principle, which is a principle of constructive recursive mathematics (CRM), also called Russian constructive mathematics [2, Ch.3]. Coquand and Mannaa [3] showed the independence of MP with Martin-Löf's type theory. Their method consists in providing a forcing extension of type theory, which contains a "generic" infinite sequence of Booleans, and where types are interpreted using a Beth model. They only need sequences of Booleans to ensure that sequences can always be extended with two distinct values. Kripke also proved a similar result for Kreisel's FC system of absolutely free choice sequences [7, p.104].

Similarly, we can directly prove that the ↓-squashed LEM is not consistent anymore with BITT because of a similar reason as for MP. To prove the validity of the ↓-squashed LEM, we would have to prove in the metatheory that for all propositions, there exists a bar of the current library such that either (1) the proposition is true at the bar, or (2) that it is false in all extensions of the bar. This is not possible anymore because choice sequences can always evolve differently when multiple choices are possible. We can now validate the following axiom in the metatheory:[23]

$$H \vdash \neg\forall P : \mathbb{U}_i.\ \downarrow(P+\neg P) \quad \text{BY [NOT-LEM]}$$

To prove this in the metatheory, we first pick a fresh name of an empty choice sequence of numbers, say *csn*. We then instantiate the universally quantified formula with $\exists n : \mathbb{N}.\ csn(n) =_{\mathbb{N}} 1$. We use 1 here instead of 0 as in [3] for convenience, because we use 0 as default value for choice sequences of numbers. We can then prove that $\exists n : \mathbb{N}.\ csn(n) =_{\mathbb{N}} 1$ is not inhabited in the current library because for all bars of the current library, there is always a library in the bar that can be extended so that the *csn* sequence does not contain 1, for example, filling up the sequence with the default value 0. Also, we can prove that $\neg\exists n : \mathbb{N}.\ csn(n) =_{\mathbb{N}} 1$ is false because we can prove that there exists an extension of the current library where the choice sequence contains the choice 1.

B Markov's Principle

As mentioned above Markov's Principle (MP) was shown to be consistent with Nuprl (using a certain squashed form of excluded middle). However, MP is not true anymore in BITT for similar reasons as in [4; 3]. We showed that MP is false using a similar method as in Appx. A, i.e., we proved:[24]

$$\forall P : \text{bool}^{\mathbb{N}}.\ \neg(\forall n : \mathbb{N}.\ \neg\uparrow(P(n))) \implies \exists n : \mathbb{N}.\ \uparrow(P(n))$$

To prove that MP is not true, i.e, not inhabited, we will derive `False` from MP in all extensions of the current library. Let lib_1 be such an extension. To prove this we will now reserve the namespace 2 for Booleans. Let *csn* be a fresh choice sequence name in that namespace—*csn* is fresh w.r.t. lib_1. We now instantiate MP with *csn*. It is easy to prove that *csn* has type $\text{bool}^{\mathbb{N}}$ because its namespace constrains its choices to be Booleans. Then, (1) we will prove that $\neg(\forall n : \mathbb{N}.\ \neg\uparrow(csn(n)))$ and finally (2) we will derive `False` from $\exists n : \mathbb{N}.\ \uparrow(csn(n))$.

(1) To prove $\neg(\forall n : \mathbb{N}.\ \neg\uparrow(csn(n)))$, we derive `False` from $\forall n : \mathbb{N}.\ \neg\uparrow(csn(n))$, and this for all extensions of lib_1. Let lib_2 be such an extension. The choice sequence *csn* might be filled with some Booleans in that library lib_2. Let us assume that it is filled with k Booleans. We then instantiate $\forall n : \mathbb{N}.\ \neg\uparrow(csn(n))$ with k, and we get to assume that in all extensions lib_3 of lib_2, $\uparrow(csn(n))$ is not inhabited. We create such an extension simply by adding the choice `tt` to the choice sequence *csn*, i.e., `tt` is the k's choice in that sequence. We now instantiate our assumption with this library, and we have to derive `False` from $\uparrow(csn(k))$, which is trivial because $csn(k)$ computes to `tt` in lib_3.

(2) Now let us turn to deriving `False` from $\exists n : \mathbb{N}.\ \uparrow(csn(n))$. This assumption says that there exists a bar of the current library in which n computes to a number k and $\uparrow(csn(k))$ is true. To prove that we can derive `False` from this, we simply use a library above the bar where *csn* is filled with `ff` up to k, which concludes our proof.

C Non-squashed LS1

In addition to the ↓-squashed version of LS1 presented in Sec. 6.1, we have also validated a non-squashed version of that axiom, namely we have validated:[25]

$$\forall n : \mathbb{N}.\ \forall f : \mathcal{B}_n.\ \exists a : \texttt{Free}(0).\ f =_{\mathcal{B}_n} a$$

To validate this non-squashed version of LS1 we have to provide some computation that generates a choice sequence name $a \in \texttt{Free}(0)$ given a finite sequence of numbers, provided as a pair of a number $n \in \mathbb{N}$ and a function $f \in \mathcal{B}_n$. To achieve that, we added the following expressions to BITT:

$$t \in \texttt{Term} ::= \cdots \mid \texttt{comp-cs-nat}(\boxed{t_1}, t_2) \mid \texttt{comp-cs-seq}_{l,i}(\boxed{t_1}, t_2)$$

where l and i are parameters: l is a list of Coq numbers, and i is a Coq number. These new expressions compute as follows:

$$\begin{array}{lll}
\texttt{comp-cs-nat}(0, t) & \mapsto_{lib} & \langle\text{""}, []\rangle \\
\texttt{comp-cs-nat}(i, t) & \mapsto_{lib} & \texttt{comp-cs-seq}_{[],i}(t(0), t) \quad \text{, if } 0 < i \\
\texttt{comp-cs-seq}_{l,i}(j, t) & \mapsto_{lib} & \langle\text{""}, l \mathbin{@} [j]\rangle \quad \text{, if } i = ||l|| + 1 \\
\texttt{comp-cs-seq}_{l,i}(j, t) & \mapsto_{lib} & \texttt{comp-cs-seq}_{l \mathbin{@} [j],i}(t(||l|| + 1), t) \quad \text{, if } ||l|| + 1 < i
\end{array}$$

Using these computations, we can then prove that the non-squashed version of LS1 is inhabited by $\lambda n, f.\langle\texttt{comp-cs-nat}(n, f), \star\rangle$. The proof is similar to the one of the ↓-squashed version of LS1.

D Extensional LS2

As mentioned in Sec. 6.2, both intensional and extensional equality over choice sequences are decidable. In addition to the standard intensional version of LS2, we have also validated the following extensional version of LS2:[26]

[23] See https://github.com/vrahli/NuprlInCoq/blob/master/rules/rules_not_classical.v.
[24] See https://github.com/vrahli/NuprlInCoq/blob/master/rules/rules_not_MP.v.
[25] See https://github.com/vrahli/NuprlInCoq/tree/beth/rules/rules_choice4.v.
[26] See https://github.com/vrahli/NuprlInCoq/tree/beth/rules/rules_choice5.v.

Proposition D.1 (Decidability of extensional equality)**.** *The following holds in BITT:*

$$\forall a, b : \texttt{Free}(0).\ a =_{\mathcal{B}} b \ \vee\ \neg a =_{\mathcal{B}} b$$

and it is inhabited by the term:

$$\lambda a, b.\texttt{if}\ a{=}b\ \texttt{then}\ \texttt{tt}\ \texttt{else}\ \texttt{ff}$$

The proof of this proposition slightly differs from the one for the intensional version of LS2. First of all, to prove that the proposition is well-formed, i.e., that it is indeed a type, one has to prove that $\texttt{Free}(0)$ is a subtype of $\mathcal{B}$, which is true as shown in Prop. 5.1. Next, we have to prove that if a and b have the same name (which is decidable) then they are extensionally equal, i.e., equal in the $\mathcal{B}$ type; and otherwise they are not extensionally equal. Proving that if a and b have the same name then they are extensionally equal is trivial because a and b are the same sequence. Proving that if a and b do not have the same name then they are not extensionally equal is slightly more complicated. According to the semantics of $\neg$, to prove this we essentially have to prove that there exists an extension of the current library where $a(k)$ and $b(k)$ compute to different numbers, for some number k. More precisely, we first assume that there exists an extension of the current library where $a =_{\mathcal{B}} b$ is true, and we prove a contradiction. Let us call lib' that extension. Then, thanks to monotonicity, we know that $a =_{\mathcal{B}} b$ must be true in all extensions of lib'. We will create an extension lib'' of lib' where a's kth choice is 0 for some number k, and b's kth choice is 1. One slight difficulty to create a valid extension of lib' is to make sure that k is greater than the length of any sequence in the "space" part of the choice sequence names a and b. Otherwise, we might not be able to ensure that the kth choices for a and b are 0 and 1 because choices have to respect the restrictions embedded in name spaces. Once we have filled a's kth slot with 0 and b's kth slot with 1, it is straightforward to prove that a and b are not extensionally equal because they compute to different values when applied to k.

References

[1] Abhishek Anand and Vincent Rahli. "Towards a Formally Verified Proof Assistant". In: *ITP 2014*. Vol. 8558. LNCS. Springer, 2014, pp. 27–44.

[2] Douglas Bridges and Fred Richman. *Varieties of Constructive Mathematics*. London Mathematical Society Lecture Notes Series. Cambridge University Press, 1987.

[3] Thierry Coquand and Bassel Mannaa. "The Independence of Markov's Principle in Type Theory". In: *FSCD 2016*. Vol. 52. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, 17:1–17:18.

[4] Thierry Coquand, Bassel Mannaa, and Fabian Ruch. "Stack semantics of type theory". In: *LICS 2017*. IEEE Computer Society, 2017, pp. 1–11.

[5] Karl Crary. "Type-Theoretic Methodology for Practical Programming Languages". PhD thesis. Ithaca, NY: Cornell University, Aug. 1998.

[6] Alexei Kopylov and Aleksey Nogin. "Markov's Principle for Propositional Type Theory". In: *CSL 2001*. Vol. 2142. LNCS. Springer, 2001, pp. 570–584.

[7] Saul A. Kripke. "Semantical Analysis of Intuitionistic Logic I". In: *Formal Systems and Recursive Functions*. Vol. 40. Studies in Logic and the Foundations of Mathematics. Elsevier, 1965, pp. 92–130.

[8] Vincent Rahli and Mark Bickford. "Validating Brouwer's continuity principle for numbers using named exceptions". In: *Mathematical Structures in Computer Science* (2017), pp. 1–49.

[9] Vincent Rahli, Mark Bickford, and Robert L. Constable. "Bar induction: The good, the bad, and the ugly". In: *LICS 2017*. Extended version avaible at https://vrahli.github.io/articles/bar-induction-lics-long.pdf. IEEE Computer Society, 2017, pp. 1–12.