A Nominal Exploration of Intuitionism

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

Mark Bickford
Cornell University, USA
markb@cs.cornell.edu

Abstract

This papers extends the Nuprl proof assistant (a system representative of the class of extensional type theories à la Martin-Löf) with *named exceptions* and *handlers*, as well as a nominal *fresh* operator. Using these new features, we prove a version of Brouwer's Continuity Principle for numbers. We also provide a simpler proof of a weaker version of this principle that only uses diverging terms. We prove these two principles in Nuprl's meta-theory using our formalization of Nuprl in Coq and show how we can reflect these meta-theoretical results in the Nuprl theory as derivation rules. We also show that these additions preserve Nuprl's key meta-theoretical properties, in particular consistency and the congruence of Howe's computational equivalence relation. Using continuity and the fan theorem we prove important results of Intuitionistic Mathematics: Brouwer's continuity theorem and bar induction on monotone bars.

Categories and Subject Descriptors D.3.1 [*PROGRAMMING LANGUAGES*]: Formal Definitions and Theory; F.4.1 [*MATHEMATICAL LOGIC AND FORMAL LANGUAGES*]: Mathematical Logic

Keywords Intuitionistic Type Theory, Nuprl, Coq, Continuity, Nominal Type Theory, Exceptions, Squashing, Truncation

1. Introduction

Continuity. There are two principles that distinguish Brouwer's mathematics from other constructive mathematics, namely *bar induction* and a *continuity principle* [11, 19, 37, 53, 74, 81, 82, 84, 85]. In this document we consider the following weak and strong continuity principles on the Baire space $\mathcal{B} = \mathbb{N}^{\mathbb{N}} = \mathbb{N} \to \mathbb{N}$ ($\mathbf{\Pi}$,$\mathbf{\Sigma}$ are the logical $\forall$,$\exists$; as explained below, $\underline{\mathbf{\Sigma}}$ is a truncated/squashed existential quantifier; $+$ in the context of types is the disjoint union type; `inl` is the left injection constructor; `isl` checks whether a term is a left injection; $\mathbb{N}_n$ is the type of natural numbers strictly less than n; and $t =_T u$ expresses that t and u are equal in type T):

$$\begin{aligned}\texttt{WCP} = {} & \mathbf{\Pi}F{:}\mathcal{B} \to \mathbb{N}.\\ & \quad \mathbf{\Pi}f{:}\mathcal{B}.\\ & \quad\quad \underline{\mathbf{\Sigma}}n{:}\mathbb{N}.\ \mathbf{\Pi}g{:}\mathcal{B}.f =_{\mathbb{N}^{\mathbb{N}_n}} g \to F(f) =_{\mathbb{N}} F(g)\end{aligned}$$

$$\begin{aligned}\texttt{SCP} = {} & \mathbf{\Pi}F{:}\mathcal{B} \to \mathbb{N}.\\ & \quad \underline{\mathbf{\Sigma}}M{:}(\mathbf{\Pi}n{:}\mathbb{N}.\mathbb{N}^{\mathbb{N}_n} \to (\mathbb{N}+\texttt{Unit})).\\ & \quad\quad \mathbf{\Pi}f{:}\mathcal{B}.\\ & \quad\quad\quad \underline{\mathbf{\Sigma}}n{:}\mathbb{N}.\\ & \quad\quad\quad\quad M\ n\ f =_{\mathbb{N}+\texttt{Unit}} \texttt{inl}(F(f))\\ & \quad\quad\quad\quad \wedge\ \mathbf{\Pi}m{:}\mathbb{N}.\texttt{isl}(M\ m\ f) \to m =_{\mathbb{N}} n\end{aligned}$$

`WCP` is the usual *pointwise continuity principle* on natural numbers, sometimes called *weak continuity principle*. It says that given a function F of type $\mathcal{B} \to \mathbb{N}$ and a function f of type $\mathcal{B}$, $F(f)$ can only depend on an initial segment of f. The length of the smallest such segment is called the *modulus of continuity* of F at f. Kleene used some version of the *strong continuity principle* `SCP`[1] to prove bar induction on monotone bars from bar induction on decidable bars [53, pp.78]. `SCP` says that there is a uniform way (called M in the formula) to decide whether n is the modulus of continuity of F at f, and if so returns the value $F(f)$ [53, pp.70–71].

Truncation/Squashing. Escardó and Xu [39] proved in Agda [2, 17] that `WCP` is false when $\underline{\mathbf{\Sigma}}$ is the sum type $\mathbf{\Sigma}$ of Martin-Löf's type theory. They also mention that this principle is consistent when $\mathbf{\Sigma}$ is *truncated at the propositional level* [83, pp.117]. In Nuprl [6, 26], propositional truncation corresponds to *squashing* a type down to a single equivalence class (i.e., all inhabitants are equal) using quotient types [28]: $\lfloor T = T//\texttt{True}$. $\lfloor T$ is a proof-irrelevant type. Its members are the members of T, and they are all equal to each other because if $x, y \in T$ then $(x =_T y \iff \texttt{True})$. In Nuprl we often squash types in a much stronger sense by throwing away the evidence that a type is inhabited and squashing it down to a single inhabitant using, e.g., set types: $\downarrow T = \{\texttt{Unit} \mid T\}$ (this is the same definition as in [26, pp.60]). The only member of this type is $\star$[2], which is the single inhabitant of `Unit`, and $\star$ inhabits $\downarrow T$ if T is true/inhabited, but we do not keep the proof that it is true. Note that $\lfloor T \to \downarrow T$ is true because it is inhabited by $\lambda x.\star$, but we cannot prove the converse because to prove $\lfloor T$ we have to exhibit an inhabitant of T, which $\downarrow T$ does not give us because we have thrown away the evidence that T is inhabited (only $\star$ inhabits $\downarrow T$). Appendix F of [72] discusses derivable inference rules that one can use to reason about these two squashing operators.

In this paper we prove that versions of `WCP` and `SCP` are true facts about Nuprl's functions. We carry out these proofs in Nuprl's meta-theory [4, 5] using our formalization of Nuprl in Coq [13, 33], which contains among other things: (1) an implementation

[*] This work was partially supported by the SnT and the National Research Fund Luxembourg (FNR), through PEARL grant FNR/P14/8149128.

[1] Rathjen calls it "Strong Continuity for Numbers" [74] and names it C-N (as in [82]). Dummett refers to it as "a stronger version of the Continuity Principle", names it $\text{CP}_{\exists n}$, and later calls it "the Continuity Principle" [37, pp.59–60]. Troelstra [81, pp.1006] calls it CONT_0. This is Kleene's *27.2 principle [53, pp.73], which he calls "Brouwer's principle (for numbers)". Bridges and Richman [19, pp.119] mention that `SCP` is equivalent to a "principle of continuous choice", which they divide into a continuous part, namely `WCP`, and a choice part, namely the axiom of choice $\text{AC}_{1,0}$ (see Sec 5.3). Note that $\text{AC}_{1,0}$ is also sometimes used to refer to `SCP` [45, 86].

[2] The term $\star$ can be thought of as `()` in, e.g., OCaml, Haskell or SML.

of Nuprl's computation system; (2) an implementation of Howe's computational equivalence relation [50] and a proof that it is a congruence; (3) a definition of Nuprl's *Constructive Type Theory* (CTT), where types are defined as *Partial Equivalence Relations* (PERs) on closed terms following Allen's PER semantics [4, 5]; (4) definitions of Nuprl's derivation rules and proofs that these rules are valid w.r.t. Allen's PER semantics; (5) and a proof of Nuprl's consistency [7, 8].

In Sec. 3 we prove WCP where $\underline{\mathbf{\Sigma}}x{:}T.\ P$ is defined as $\downarrow\mathbf{\Sigma}x{:}T.P$, and refer to this principle as $\texttt{WCP}_{\downarrow}$. We call $\texttt{WCP}_{\lfloor}$ the version of WCP where $\underline{\mathbf{\Sigma}}$ is $\lfloor\mathbf{\Sigma}$. In Sec. 4 we prove SCP where the first (outer) $\underline{\mathbf{\Sigma}}$ is $\lfloor\mathbf{\Sigma}$ and the second (inner) is $\downarrow\mathbf{\Sigma}$, and refer to this principle as $\texttt{SCP}_{\lfloor}$. There proofs are carried out using our Coq formalization of Nuprl. We make these results available in Nuprl as inference rules, and show how we can derive directly in Nuprl a proof of $\texttt{SCP}_{\lfloor}$ without the second (inner) squashing operator. Sec. 5 shows that $\texttt{SCP}_{\lfloor}$ and $\texttt{WCP}_{\lfloor}$ are equivalent. Even though the implication $\texttt{WCP}_{\lfloor} \rightarrow \texttt{WCP}_{\downarrow}$ is trivial, we believe our proof of $\texttt{WCP}_{\downarrow}$ in Sec. 3 is still valuable because of its simplicity and because $\downarrow$ is often enough.

An important point is that we add operations sufficient to prove these principles to the Nuprl proof assistant without breaking any property of its type theory such as Nuprl's consistency or the congruence of Howe's computational equivalence relation.

Effectful computations. Following Longley's method [59], we use computational effects [10], namely named exceptions, to derive $\texttt{SCP}_{\lfloor}$. The basic method to find the n such that $F(f)$ depends only on the first n elements of f is a program $P(F, f)$ that works as follows: P tests whether F applies its argument f to a number n by running the sub-routine (written in an ML-like language):

```
let exception e in
(F (fun x => if x < n then f x else raise e);
 true) handle e => false
```

Then by testing F on increasingly larger n's, if the continuity principle is true, P eventually finds an n such that the test returns true[3]. However, for extensionally equal F and G, $P(F, f)$ and $P(G, f)$ could return different numbers. For example, if $P(F, f) = m$ and G is constructed from F by replacing an expression t occurring in F with ($\texttt{let}\ \_ := f(m+1)\ \texttt{in}\ t$), that first evaluates $f(m+1)$ and then evaluates t, then $P(G, f)$ is not guaranteed to be m. This is why we only realize squashed versions of the above mentioned continuity principles.

As Longley mentions, if F can catch the exception e then $P(F, f)$ will not necessarily compute F's modulus of continuity at f. Therefore, we have extended Nuprl with exception handlers that can only catch exceptions with a specific name, and we have added the ability to generate fresh names (Sec. 7 discusses related nominal systems).

Related proofs of continuity. This is not the first (formal) proof that a type theory satisfies Brouwer's continuity principle. Coquand and Jaber [30, 31] proved the *uniform* continuity of a Martin-Löf-like intensional type theory using *forcing* [9, 11, 24, 25, 61]. Their method consists in adding a generic element f as a constant to the language that stands for a Cohen real of type $2^{\mathbb{N}}$, and defining the forcing conditions as approximations of f, i.e., finite sub-graphs of f. They then define a suitable *computability* predicate that expresses when a term is a computable term of some type up to approximations given by the forcing conditions. The key steps are to (1) first prove that f is computable and then (2) prove that well-typed terms are computable, from which they derive uniform continuity (the uniform modulus of continuity is given by the approximations). The uniform continuity principle is, where F is now a function on the Cantor space $\mathcal{C} = 2^{\mathbb{N}}$ instead of the Baire space: $\texttt{UCP} = \mathbf{\Pi}F{:}\mathcal{C} \rightarrow \mathbb{N}.\underline{\mathbf{\Sigma}}n{:}\mathbb{N}.\ \mathbf{\Pi}f, g{:}\mathcal{C}.f =_{2^{\mathbb{N}_n}} g \rightarrow F(f) =_{\mathbb{N}} F(g)$. Escardó and Xu [39] showed that in the case of uniform continuity $\underline{\mathbf{\Sigma}}$ can equivalently be $\mathbf{\Sigma}$ or $\lfloor\mathbf{\Sigma}$. In [31], Coquand and Jaber provide a Haskell realizer that computes the uniform modulus of continuity of a functional on the Cantor space[4].

Similarly, Escardó and Xu [89] proved that the definable functionals of Gödel's system T [46] are uniformly continuous on the Cantor space (without assuming classical logic or the Fan Theorem). For that, they developed a constructive continuous model, the C-**Space** category, of Gödel's system T, and proved that C-**Space** has a *Fan functional* that given a function F in $\mathcal{C} \rightarrow \mathbb{N}$ can compute the modulus of uniform continuity of F. Relating C-**Space** and the standard set-theoretical model of system T, they show that all T-definable functions on the Cantor space are uniformly continuous. Finally, using this model, they show how to extract computational content from proofs in HA^{ω} extended with a uniform continuity axiom UC, which is realized by the Fan functional.

In [42] Escardó provides a simple and elegant proof that all T-definable functions are continuous on the Baire space and uniformly continuous on the Cantor space using a *generic element* as in [30] but without using forcing. His method consists in providing an alternative interpretation of system T, where a number is interpreted by a dialogue tree that "describes the computation of a natural number relative to an unspecified oracle $\alpha : \mathbb{N}^{\mathbb{N}}$" [42]. Such a computation is called a *dialogue*, which is a function that given a dialogue tree, returns a function of type $\mathcal{B} \rightarrow \mathbb{N}$. Escardó first proves that dialogues are continuous. This means that a function is continuous if it is extensionally equal to a dialogue. The key steps are to (1) define a suitable logical relation between the standard interpretation and the alternative one that relates numbers and dialogues w.r.t. a given oracle; and (2) prove that all system T terms are related under the two interpretations. It then follows that for all system T term t of type $(\iota \Rightarrow \iota) \Rightarrow \iota$ (where ι is the type of numbers), there is a dialogue tree d such that the standard interpretation of t and the dialogue on d are extensionally equal functions, from which he derives uniform continuity. The dialogue d is built using a *generic* sequence that allows dialogue trees to call the oracle.

Results. Our proof method differs from the ones discussed above in the sense that it is "mostly" computational. In Sec. 3 we use diverging terms to prove $\texttt{WCP}_{\downarrow}$, and in Sec. 4 we use computational effects (named exceptions) to probe terms and derive $\texttt{SCP}_{\lfloor}$ using (non-strict) lock-step simulations of these effectful computations. Sec. 5 shows that $\texttt{SCP}_{\lfloor}$ and $\texttt{WCP}_{\lfloor}$ are equivalent. To prove $\texttt{SCP}_{\lfloor}$, we added named exceptions as well as a *fresh* operator to Nuprl's computation system, and showed that these additions preserve Nuprl's key meta-theoretical properties, such as consistency (see Sec. 2.3 and 4.3) and the congruence of Howe's computational equivalence relation (see Sec. 2.2 and 4.2). As mentioned in Sec. 4.9, $\texttt{SCP}_{\lfloor}$ justifies a corresponding inference rule that we added to Nuprl. Sec. 5 discusses the relation between WCP and SCP and the connection with the axiom of choice, as well as the status of the (squashed) axiom of choice in Nuprl. Using those continuity rules, as explained in Sec. 6, we have proved in Nuprl (1) a fully unsquashed version of UCP using Escardó and Xu's method [39]; (2) that all real functions defined on the unit interval are uniformly continuous [37, pp.87]; and (3) that bar induction on monotone bars follows from bar induction on decidable bars following Kleene's proof [53, pp.69–73].

[3] See Bauer's blog for more details: http://math.andrej.com/2006/03/27/sometimes-all-functions-are-continuous/.

[4] See also Escardó's tutorial http://www.cs.bham.ac.uk/~mhe/.talks/popl2012/ for examples on how to search the Cantor space, as well as [40, 41, 62], which point to citations and constructions by, among others, Gandy and Berger.

The results presented in this paper have either been formalized in Coq and are available both at https://github.com/vrahli/NuprlInCoq and http://www.nuprl.org/html/Nuprl2Coq/; or they have been formalized in Nuprl and are available at http://www.nuprl.org/LibrarySnapshots/Published/Version1/Standard/continuity/index.html for results related to continuity, at http://www.nuprl.org/LibrarySnapshots/Published/Version1/Standard/int_2/index.html for results related to the fan theorem, and at http://www.nuprl.org/LibrarySnapshots/Published/Version1/Standard2/reals/index.html for results related to real analysis.

2. Nuprl

Nuprl is an interactive theorem prover that implements a type theory called Constructive Type Theory (CTT) [6, 26]. Nuprl's CTT "mostly" differs from other similar constructive type theories such as the ones implemented by Agda [2, 17], Coq [13, 33], or Idris [18, 51], in the sense that CTT is an *extensional* type theory (i.e., propositional and definitional equality are identified [49]) with types of partial functions [29, 34, 80]. This section presents some key aspects of Nuprl that will be used in the rest of this paper.

2.1 Nuprl's Computation System

Fig. 1 presents a subset of Nuprl's syntax and small-step operational semantics [6, 8]. Nuprl's programming language is an untyped (à la Curry), lazy and applied (with pairs, injections, a fixpoint operator,. . .) λ-calculus. For efficiency, integers are primitive and Nuprl provides operations on integers such as addition, subtraction,. . . , a test for equality and a "less than" operator. Nuprl also has what we call *canonical form tests* [73] such as `ifint`, which are used to distinguish between our different kinds of values. These canonical form tests are especially useful when working with (non-disjoint) union types, which are sometimes easier to work with than disjoint unions because one does not need injections.

A term is either a variable, a value (or canonical term), or a non-canonical term. Non-canonical terms have one or two *principal arguments* (marked using boxes in Fig. 1). A principal argument of a term t is a term that has to be evaluated to a canonical form before checking whether t can be reduced further. For example the application $f(a)$ diverges if f diverges, and the canonical form test $\texttt{ifaxiom}(t, a, b)$ diverges if t diverges.

Nuprl uses a uniform syntax for terms [7, 8], and the terms in Fig. 1 are "display forms" for some specific Nuprl terms. An advantage of having a uniform syntax is that operations that work uniformly on terms are easier to define—they do not have repetitive cases as when using one constructor per operator. In Nuprl a term is of the form $\theta(\overline{b})$, where θ is its operator, and $\overline{b}$ is a list of *bound terms*. A bound term b is of the form $\overline{l}.t$, where $\overline{l}$ is a variable list. For example, the underlying representation of a λ-abstraction is $\{\text{lambda}\}(x.t)$. For convenience, we use the uniform syntax to, e.g., define Howe's computational equivalence relation below.

Fig. 1 also shows part of Nuprl's small-step operational semantics. We omit the rules that reduce principal arguments such as: if $t_1 \mapsto t_2$ then $t_1\ u \mapsto t_2\ u$. As usual, $\mapsto^*$ is the reflexive and transitive closure of $\mapsto$, and $t_1 \mapsto^k t_2$ is defined inductively on k: $t \mapsto^0 t$ and $t_1 \mapsto^{k+1} t_2$ if there exists a t such that $t_1 \mapsto t$ and $t \mapsto^k t_2$.

We now define a few useful abstractions:

$$\begin{array}{rcl@{\qquad}lcl}
\bot & = & \texttt{fix}(\lambda x.x) & \mathbb{N}_? & = & \mathbb{N} \cup \texttt{Unit} \\
\texttt{tt} & = & \texttt{inl}(\star) & \texttt{isint}(t) & = & \texttt{ifint}(t, \texttt{tt}, \texttt{ff}) \\
\texttt{ff} & = & \texttt{inr}(\star) & \texttt{isl}(t) & = & \texttt{if}\ t\ \texttt{then}\ \texttt{tt}\ \texttt{else}\ \texttt{ff}
\end{array}$$
$$\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 = \texttt{case}\ t_1\ \texttt{of}\ \texttt{inl}(x) \Rightarrow t_1 \mid \texttt{inr}(x) \Rightarrow t_2$$

We sometimes write $a =_T b$ for the type $a = b \in T$. Also, we sometimes write b for ($\texttt{if}\ b\ \texttt{then Unit else Void}$), where `Unit` and `Void` can, e.g., be defined as $0 =_{\mathbb{Z}} 0$ and $0 =_{\mathbb{Z}} 1$ respectively. We define `True` as `Unit` and `False` as `Void`.

2.2 Howe's Computational Equivalence

It turns out that Nuprl's type system is not only closed under computation but more generally under Howe's computational equivalence $\sim$, which he proved to be a congruence [50]. In Nuprl, in any context C, when $t \sim t'$ we can rewrite t into t' without having to prove anything about types. We rely on this relation to prove equalities between programs (bisimulations) without concern for typing [73]. Howe's computational equivalence is defined on closed terms as follows: $t \sim u$ if $t \preccurlyeq u \wedge u \preccurlyeq t$. Howe coinductively defines the approximation (or simulation) relation $\preccurlyeq$ as the largest relation R on closed terms such that $R \subset [R]$, where $[\cdot]$ is the following closure operator (also defined on closed terms): $t\ [R]\ u$ if whenever t computes to a value $\theta(\overline{b})$, then u also computes to a value $\theta(\overline{b'})$ such that $\overline{b}\ R\ \overline{b'}$. To make that precise we have to extend R to open and bound terms: see [7, 8, 50] for details. By definition, one can derive, e.g., that $\bot \preccurlyeq t$ for all closed term t.

2.3 Nuprl's Type System

Following Allen's PER semantics, Nuprl's types are defined as partial equivalence relations (PERs) on closed terms [4, 5]. Allen's PER semantics can be seen as an inductive-recursive definition of: (1) an inductive relation $T_1 \equiv T_2$ that expresses type equality; and (2) a recursive function $a \equiv b \in T$ that expresses equality in a type. We write $\texttt{type}(T)$ for $T \equiv T$, and $t \in T$ for $t \equiv t \in T$. Among other things, it follows that the (theoretical) proposition $a = b \in T$ is true (inhabited by $\star$) iff $a \equiv b \in T$ holds in the meta-theory. See [7, 8] for more details.

Nuprl's type system includes Martin-Löf dependent types, identity (or equality) types, a hierarchy of universes, W types, union and intersection types, quotient types [28], set types, and partial types [34]. The top part of Fig. 1 lists some of Nuprl's types. Among these, `Base` is the type of all closed terms of the computation system with $\sim$ as its equality. The type $t_1 \preceq t_2$ is true if the meta-theoretical statement $t_1 \preccurlyeq t_2$ is true, and $t_1 \preceq t_2$ and $t_3 \preceq t_4$ are equal types if $(t_1 \preccurlyeq t_2 \Longleftrightarrow t_3 \preccurlyeq t_4)$. Similarly the type $t_1 \simeq t_2$ is true if $t_1 \sim t_2$ is true, and $t_1 \simeq t_2$ and $t_3 \simeq t_4$ are equal types if $(t_1 \sim t_2 \Longleftrightarrow t_3 \sim t_4)$ (see [73] and [72, Appendices A&B] for more details). For example, it is enough to prove that t_1 and t_2 are members of `Base` to prove that $t_1 \simeq t_2$ is a type. Also, it turns out that $t \simeq t$ is a true type in any context. These types allow us, to some extent, to reason about Nuprl's computation system directly in the theory. Nuprl has a rich type theory that makes type checking undecidable. In practice this is mitigated by type inference and type checking heuristics implemented as tactics.

We have implemented Nuprl's term language, its computation system, Howe's $\sim$ relation, and Allen's PER semantics in Coq [7, 8]. We have also showed that Nuprl is consistent by (1) proving that Nuprl's inference rules are valid w.r.t. Allen's PER semantics, and (2) proving that `False` is not inhabited. Using these two facts, we derive that there cannot be a proof derivation of `False`, i.e., Nuprl is consistent. (In addition to [7, 8], see also [72, Appendix A] for more details regarding Nuprl's consistency.)

We are using our Coq formalization to prove all the inference rules of Nuprl, and have already verified a large number of them. This paper presents extensions we made to Nuprl in order to prove $\texttt{SCP}_{\downarrow}$. It includes adding some *nominal features* such as a *fresh* operator, *named exceptions*, and *exception handlers*. Using our Coq formalization, we provide in Sec. 3 a simple proof that $\texttt{WCP}_{\downarrow}$ is true w.r.t. Nuprl's PER semantics using the fact that $\bot$ diverges, and in Sec. 4 we prove $\texttt{SCP}_{\downarrow}$ using the nominal features mentioned above.

3. Weak Continuity Principle

Our proof of $\texttt{WCP}_{\downarrow}$ uses $\bot$ and the fact that it diverges. For further details regarding this proof conducted in our implementation of

$$
\begin{array}{llllllllllll}
v \in \texttt{Value} ::= vt & \text{(type)} & \mid \texttt{inl}(t) & \text{(left injection)} & \mid \star & \text{(axiom)} & \mid \lambda x.t & \text{(lambda)} \\
\quad\quad\quad\mid i & \text{(integer)} & \mid \texttt{inr}(t) & \text{(right injection)} & \mid \langle t_1, t_2\rangle & \text{(pair)} & & \\
vt \in \texttt{Type} ::= \mathbb{Z} & \text{(integer)} & \mid \mathbf{\Pi} x{:}t_1.t_2 & \text{(product)} & \mid \mathbf{\Sigma} x{:}t_1.t_2 & \text{(sum)} & \mid \texttt{Base} & \text{(base)} \\
\quad\quad\quad\mid t_1 = t_2 \in t & \text{(equality)} & \mid \cup x{:}t_1.t_2 & \text{(union)} & \mid \cap x{:}t_1.t_2 & \text{(intersection)} & \mid t_1 \preceq t_2 & \text{(simulation)} \\
\quad\quad\quad\mid t_1//t_2 & \text{(quotient)} & \mid t_1{+}t_2 & \text{(disjoint union)} & \mid \{x : t_1 \mid t_2\} & \text{(set)} & \mid t_1 \simeq t_2 & \text{(bisimulation)} \\
\quad\quad\quad\mid \mathbb{U}_i & \text{(universe)} & \mid \overline{t} & \text{(partial)} & \mid \texttt{W}(x{:}t_1.t_2) & \text{(W)} & & \\
t \in \texttt{Term} ::= x & \text{(variable)} & \mid \texttt{let}\ x := \boxed{t_1}\ \texttt{in}\ t_2 & \text{(call-by-value)} & \mid \texttt{if}\ \boxed{t_1}{<}\boxed{t_2}\ \texttt{then}\ t_3\ \texttt{else}\ t_4 & \text{(less than)} & & \\
\quad\quad\quad\mid v & \text{(value)} & \mid \texttt{let}\ x, y = \boxed{t_1}\ \texttt{in}\ t_2 & \text{(spread)} & \mid \texttt{ifint}(\boxed{t_1}, t_2, t_3) & \text{(integer test)} & & \\
\quad\quad\quad\mid \boxed{t_1}\ t_2 & \text{(application)} & \mid \texttt{if}\ \boxed{t_1} =_{\mathbb{Z}} \boxed{t_2}\ \texttt{then}\ t_3\ \texttt{else}\ t_4 & \text{(integer equality)} & \mid \texttt{ifaxiom}(\boxed{t_1}, t_2, t_3) & \text{(axiom test)} & & \\
\quad\quad\quad\mid \texttt{fix}(\boxed{t}) & \text{(fixpoint)} & \mid \texttt{case}\ \boxed{t_1}\ \texttt{of}\ \texttt{inl}(x) \Rightarrow t_2 \mid \texttt{inr}(y) \Rightarrow t_3 & \text{(decide)} & & & &
\end{array}
$$

$$
\begin{array}{lllllll}
(\lambda x.F)\ a & \mapsto & F[x\backslash a] & \quad & \texttt{fix}(v) & \mapsto & v\ \texttt{fix}(v) \\
\texttt{let}\ x, y = \langle t_1, t_2\rangle\ \texttt{in}\ F & \mapsto & F[x\backslash t_1; y\backslash t_2] & & \texttt{let}\ x := v\ \texttt{in}\ t & \mapsto & t[x\backslash v] \\
\texttt{if}\ i_1 =_{\mathbb{Z}} i_2\ \texttt{then}\ t_1\ \texttt{else}\ t_2 & \mapsto & t_1, \text{if } i_1 = i_2 & & \texttt{ifint}(i, t_1, t_2) & \mapsto & t_1 \\
\texttt{if}\ i_1 =_{\mathbb{Z}} i_2\ \texttt{then}\ t_1\ \texttt{else}\ t_2 & \mapsto & t_2, \text{if } i_1 \neq i_2 & & \texttt{ifint}(v, t_1, t_2) & \mapsto & t_2, \text{if } v \text{ is not an integer} \\
\texttt{if}\ i_1{<}i_2\ \texttt{then}\ t_1\ \texttt{else}\ t_2 & \mapsto & t_1, \text{if } i_1 < i_2 & & \texttt{ifaxiom}(\star, t_1, t_2) & \mapsto & t_1 \\
\texttt{if}\ i_1{<}i_2\ \texttt{then}\ t_1\ \texttt{else}\ t_2 & \mapsto & t_2, \text{if } i_1 \not< i_2 & & \texttt{ifaxiom}(v, t_1, t_2) & \mapsto & t_2, \text{if } v \text{ is not } \star \\
\texttt{case inl}(t)\ \texttt{of inl}(x) \Rightarrow F \mid \texttt{inr}(y) \Rightarrow G & \mapsto & F[x\backslash t] & & \texttt{case inr}(t)\ \texttt{of inl}(x) \Rightarrow F \mid \texttt{inr}(y) \Rightarrow G & \mapsto & G[y\backslash t]
\end{array}
$$

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

Nuprl in Coq, the interested reader is invited to look at `https://github.com/vrahli/NuprlInCoq/blob/master/continuity/continuity_roadmap.v`. The same proof would not work for $\downharpoonleft$, because using $\downarrow$ we can compute the modulus of continuity of a function in the meta-theory (this computation does not have to be expressible in the theory because only $\star$ inhabits $\downarrow$-squashed types), while using $\downharpoonleft$ we would have to come up with a Nuprl term t that does the computation (Sec. 4 shows how to do that), i.e., such that $t{\in}\texttt{WCP}_{\downharpoonleft}$.

Let $F \in \mathbb{Z}^{\mathbb{Z}} \to \mathbb{Z}$ and $f \in \mathbb{Z}^{\mathbb{Z}}$ (we use $\mathbb{Z}$ here instead of $\mathbb{N}$, but we proved a slightly more general result for functions of type $T^{\mathbb{Z}} \to \mathbb{Z}$ where T is a non-empty subtype of $\mathbb{Z}$, such as $\mathbb{N}$ or $\mathbb{N}_2$).

Step 1. By typing, this means that $F(f) \in \mathbb{Z}$, i.e., $F(f)$ computes to an integer i.

Step 2. It might seem that in that computation f only gets applied to integers, however, this is not necessarily true in an untyped language such as Nuprl. To remedy this issue, let $\texttt{force}(f) = \lambda x.\texttt{let}\ x := x + 0\ \texttt{in}\ f\ x$. Because $f = \texttt{force}(f) \in \mathbb{Z}^{\mathbb{Z}}$, by typing again we get $F(\texttt{force}(f)) \mapsto^* i$. Let us call that computation C_1. We use `force` to ensure that f's arguments are integers. If $\texttt{force}(f)$ was to be applied to a term that is not an integer then the computation would either get stuck or diverge. We know that this cannot happen because $F(\texttt{force}(f)) \mapsto^* i$.

Step 3. Let $\texttt{bound}(t, b) = \texttt{let}\ x := t\ \texttt{in}\ \texttt{if}\ |x|{<}b\ \texttt{then}\ x\ \texttt{else}\ \bot$. By computation we prove that there exists a number b such that $F(\lambda x.\texttt{let}\ x := \texttt{bound}(x, b)\ \texttt{in}\ f\ x) \mapsto^* i$. We can get such a b in the meta-theory by computing the largest number occurring in the computation C_1, i.e., if $t_1 = F(\texttt{force}(f)) \mapsto t_2 \mapsto \cdots \mapsto i = t_n$, then let b be the largest number occurring in one of the t_i. We have to squash `WCP`'s existential quantifier using $\downarrow$ because this meta-theoretical computation of b is not a Nuprl term. We prove this step using a *simulation* technique that we will reuse over and over again in this paper. We prove that given a context G, if $G[x{+}0]$ computes to a value v then $G[\texttt{bound}(x, b)]$ also computes to v, assuming that b is greater that any number occurring in the computation $G[x + 0] \mapsto^* v$. Note that we have not yet used the fact that $\bot$ diverges. This will be used in step 5. Let us call C_2 the computation $F(\lambda x.\texttt{let}\ x := \texttt{bound}(x, b)\ \texttt{in}\ f\ x) \mapsto^* i$.

Step 4. We can now instantiate our conclusion using b. It remains to prove that $\mathbf{\Pi} g{:}\mathbb{Z}^{\mathbb{Z}}.f =_{\mathbb{Z}^{\mathbb{Z}_b}} g \to F(f) =_{\mathbb{Z}} F(g)$. Because f and g agree up to b, and because the computation C_2 converges, by computation we know that $F(\lambda x.\texttt{let}\ x := \texttt{bound}(x, b)\ \texttt{in}\ g\ x) \mapsto^* i$. We prove this by showing that given a context G, if $G[\texttt{let}\ x := \texttt{bound}(x, b)\ \texttt{in}\ f\ x]$ computes to a value, then $G[\texttt{let}\ x := \texttt{bound}(x, b)\ \texttt{in}\ g\ x]$ computes to the same value. We still have not used the fact that $\bot$ diverges, because we could use any number in `bound`'s definition instead of $\bot$, such as 0, and make sure that $0 < b$.

Step 5. Again by computation: $F(\texttt{force}(g)) \mapsto^* i$. We prove this by showing that given a context G, if $G[\texttt{bound}(x, b)]$ computes to a value then $G[x + 0]$ computes to the same value because the "less than" operator in `bound`'s definition ensures that x is an integer, and because we know that $G[\texttt{bound}(x, b)]$ *does not diverge*.

Step 6. Finally, by typing, $F(g) \mapsto^* i$, i.e., $F(f) =_{\mathbb{Z}} F(g)$.

4. Strong Continuity Principle

We now prove $\texttt{SCP}_{\downharpoonleft}$ [53, pp.69–73] (Sec. 5 shows that $\texttt{SCP}_{\downharpoonleft}$ and $\texttt{WCP}_{\downharpoonleft}$ are equivalent). We need to come up with a Nuprl term of type $\mathbf{\Pi} n{:}\mathbb{N}.\mathbb{N}^{\mathbb{N}_n} \to \mathbb{N}{+}\texttt{Unit}$ that checks whether we have reached the modulus of continuity of a function. For that, we now use exceptions as a probing mechanism to compute the modulus of continuity of a function. Instead of $\texttt{SCP}_{\downharpoonleft}$, we prove the following equivalent but slightly simpler statement [53, pp.71-72] (where the `T` in `SCPT` is for Test—see below):

$$
\begin{array}{lll}
\texttt{SCPF}(F) & = & \downharpoonleft \mathbf{\Sigma} M{:}(\mathbf{\Pi} n{:}\mathbb{N}.\mathbb{N}^{\mathbb{N}_n} \to \mathbb{N}_?). \\
 & & \quad \mathbf{\Pi} f{:}\mathcal{B}. \\
 & & \quad\quad \downharpoonleft \mathbf{\Sigma} n{:}\mathbb{N}.M\ n\ f =_{\mathbb{N}} F(f) \\
 & & \quad\quad \wedge\ \mathbf{\Pi} n{:}\mathbb{N}. \\
 & & \quad\quad\quad \texttt{isint}(M\ n\ f) \to M\ n\ f =_{\mathbb{N}} F(f) \\
\texttt{SCPT} & = & \mathbf{\Pi} F{:}\mathcal{B} \to \mathbb{N}.\ \texttt{SCPF}(F)
\end{array}
$$

Using our Coq formalization and making use of computations on terms that are only possible in the meta-theory, we proved that `SCPT` is true w.r.t. the PER semantics of Nuprl extended with the nominal features mentioned above. We then proved directly in Nuprl that `SCPT` and $\texttt{SCP}_{\downharpoonleft}$ are equivalent. We prove `SCPT` rather than $\texttt{SCP}_{\downharpoonleft}$ mainly because its realizer is simpler. Intuitively, the M part of `SCPT`'s realizer is a simple test function (top), while the one for $\texttt{SCP}_{\downharpoonleft}$ is a recursive search function of the form (bottom):[5]

```
fun test n f =
  let exception e in
  (let v = F (fun x => if x < n then f x
                       else raise e)
   in Some v) handle e => None
```

[5] In both functions, None means that n is less than the modulus of continuity of F at f. In the test function, Some v means that v is F(f) and n is greater than or equal to the modulus of continuity of F at f, while the search function returns F(f) only when n is the modulus of continuity of F at f (and not when n is past the modulus as in the test function).

```
let fun search n m f =
      if m <= 0 then test n f
      else case test m f of
             | Some k => None
             | None => search n (m - 1) f
             end
in search n (n - 1) f
```

4.1 Extension of Nuprl's Computation System

4.1.1 Syntax

We extend Nuprl with *names* (or unguessable atoms [14]), *named exceptions*, *exception handlers*, and a *fresh* operator as follows:

$$\begin{array}{lll}
v & ::= \cdots \mid \boldsymbol{a} & \text{(name value)}\\
vt & ::= \dots & \\
 & \mid \texttt{Name} & \text{(name type)}\\
 & \mid \texttt{Exc}(t_1,t_2) & \text{(exception type)}\\
e & ::= \texttt{exc}(t_1,t_2) & \text{(exception)}\\
t & ::= \dots & \\
 & \mid e & \text{(exception)}\\
 & \mid \texttt{if}\ \boxed{t_1}{=}\boxed{t_2}\ \texttt{then}\ t_3\ \texttt{else}\ t_4 & \text{(name equaliy)}\\
 & \mid \boldsymbol{\nu}x.\boxed{t} & \text{(fresh)}\\
 & \mid \texttt{try}_n\ \boxed{t}\ \texttt{with}\ x.c & \text{(try/catch)}
\end{array}$$

Name is a type of names (constants) and $\boldsymbol{a}$ stands for a name. Names were introduced in Nuprl to reason about logical foundations for security [14]. To account for names, Allen generalized his PER semantics [5] to a so-called *supervaluation* semantics that quantifies over all possible implementations of the Name type [3]. Names come with two meta-theoretical operations: a fresh operator to generate a fresh name w.r.t. a list of names, and a test for equality. As in Pitts and Stark's ν-calculus [68] or Odersky's $\lambda\nu$-calculus [63], we add two corresponding operators to Nuprl.

Our exceptions and handlers are similar to Lebresne's [57]. In Nuprl, an exception e has two subterms: the first one is e's name and the second one is some piece of data that can be used if e is caught. The type $\texttt{Exc}(t_1,t_2)$ is the type of exceptions with names of type t_1 and data of type t_2. In general exceptions can be named with terms other than names. For example, if $\boldsymbol{a}$ and $\boldsymbol{b}$ are names, both $\texttt{exc}(\boldsymbol{a},0)$ and $\texttt{exc}(\boldsymbol{b},0)$ have type $\texttt{Exc}(\texttt{Name},\mathbb{Z})$ (among others); and $\texttt{exc}(1,0)$ has type $\texttt{Exc}(\mathbb{Z},\mathbb{Z})$. We also add exception handlers of the form $\texttt{try}_n\ t\ \texttt{with}\ x.c$, where t is the term we try to evaluate, and $c[x\backslash d]$ is the code we run if we catch an exception with name n and data d. Therefore, a handler cannot catch all exceptions. A canonical operator is now either a value or an exception.

Let us define a few useful abstractions/abbreviations:

$$\begin{array}{ll}
\texttt{Name}_n & = \{x : \texttt{Name} \mid x \simeq n\}\\
\texttt{Exc}_n(T) & = \texttt{Exc}(\texttt{Name}_n, T)\\
\texttt{Exc}_n & = \texttt{Exc}_n(\texttt{Unit})\\
T?_n & = T \cup \texttt{Exc}_n\\
\texttt{exc}_n & = \texttt{exc}(n,\star)\\
\texttt{try}_n\ t & = \texttt{try}_n\ t\ \texttt{with}\ x.\star
\end{array}$$

If T is not an exception type, $T?_n$ is the type of terms that either compute to elements of type T or that compute to exceptions with name n and data $\star$.

The type $T?_n$ is similar to Lebresne's type $A \uplus \{\epsilon\}$, where A is a type and ϵ is an exception [56, 57]. In addition Lebresne also introduces *corruption* types of the form $A^{\{\epsilon\}}$. A term in $A^{\{\epsilon\}}$ is a term in A where some part has been replaced by the exception ϵ. As he mentions, an expression of that type does not necessarily evaluates to an exception. For example, if Nuprl had such a type, $\texttt{inl}(\texttt{exc}_{\boldsymbol{a}})$ could be of type $(\mathbb{N}{+}\texttt{Unit})^{\{\boldsymbol{a}\}}$. We leave adding corruption types to Nuprl for future work (Sacchini [75] shows that corruption in the presence of dependent types has interesting consequences).

Exceptions are a standard programming language feature. In the interactive theorem proving realm they are "well-adapted to programming strategies which may be (in fact usually are) inapplicable to certain goals" [48, pp.11]. However, exceptions are often not accounted for in types. As mentioned above, Lebresne's Fx system [57] provides type constructors to express two different levels of *corruption*. Lebresne [57] mentions that to get exceptions one could either directly encode them in the language (e.g., using monads) or add them as primitive. We decided to add them as primitives for the same reasons (e.g., compositionality) stated in his paper. David and Mounier [36] introduced EX_2 as an extension of Krivine's FA_2 system [55] with exceptions. As in Nuprl, both Fx and in EX_2 implement call-by-name exceptions. Also, in all three systems exceptions and handlers are named, and handlers can only catch exceptions with the correct name.

4.1.2 Operational Semantics of $\boldsymbol{\nu}$

Let us now precisely define how fresh and handlers compute. A fresh expression of the form $\boldsymbol{\nu}x.t$ computes differently depending on whether t is a variable, a canonical term, or a non-canonical term. Let us consider each of these cases.

Variable. If t is the variable x then $\boldsymbol{\nu}x.t$ reduces to itself and therefore diverges. Therefore, one can prove that $\boldsymbol{\nu}x.x \sim \bot$. This differs both from Odersky's [63] approach where $\boldsymbol{\nu}x.x$ is stuck and from Pitts' approach [66] where $\boldsymbol{\nu}x.x$ is a normal form. If t reduces to another variable than x then the computation gets stuck because the term is open.

Non-canonical. If t is non-canonical then

$$\boldsymbol{\nu}x.t \mapsto \boldsymbol{\nu}x.u[\boldsymbol{a}\backslash x] \qquad \text{if} \qquad t[x\backslash \boldsymbol{a}] \mapsto u$$

where $\boldsymbol{a}$ is a fresh name w.r.t. t (written $\boldsymbol{a}\#t$), and $t[\boldsymbol{a}\backslash u]$ is a capture avoiding substitution function on names (similar to the usual substitution operation on variables). This ensures that fresh names do not escape the scope of $\boldsymbol{\nu}$ expressions. As expected (if $x \neq y$):

$$\boldsymbol{\nu}x.\boldsymbol{\nu}y.\texttt{if}\ x{=}y\ \texttt{then}\ \texttt{tt}\ \texttt{else}\ \texttt{ff} \mapsto^* \texttt{ff}$$

We cannot simply reduce $\boldsymbol{\nu}$ as follows: $\boldsymbol{\nu}x.t \mapsto t[x\backslash \boldsymbol{a}]$, because Howe's computational equivalence would not be a congruence. For example, $\boldsymbol{\nu}x.\texttt{inl}(x) \mapsto \texttt{inl}(\boldsymbol{a})$ and $(\texttt{let}\ y := \boldsymbol{a}\ \texttt{in}\ x) \sim x$ but $\boldsymbol{\nu}x.\texttt{inl}(\texttt{let}\ y := \boldsymbol{a}\ \texttt{in}\ x) \not\mapsto^* \texttt{inl}(\boldsymbol{a})$.

Canonical. If t is a canonical form (a value or an exception), then we "push" $\boldsymbol{\nu}$ "down" the expression as in Odersky's $\lambda\nu$-calculus [60, 63] (as opposed to using, e.g., stateful dynamic allocation [60] or the notion of *prevalues* [52], which are values prefixed with a list of "fresh name" binders):

$$\boldsymbol{\nu}x.t \ \mapsto\ \Downarrow_x t$$

where $\Downarrow$ computes as follows on terms:

$$\Downarrow_x \theta(b_1;\cdots;b_n) = \theta(\Downarrow_x b_1;\cdots;\Downarrow_x b_1)$$

and as follows on bound terms:

$$\Downarrow_x (\bar{l}.t) = \bar{l}.\boldsymbol{\nu}x'.t$$

where, in order to avoid variable capture, x' is x if $x \notin \bar{l}$, and a fresh variable w.r.t. t otherwise. For example $\boldsymbol{\nu}x.\langle 1,x\rangle \mapsto \langle \boldsymbol{\nu}x.1, \boldsymbol{\nu}x.x\rangle$ and $\boldsymbol{\nu}x.\lambda y.t \mapsto \lambda y.\boldsymbol{\nu}x.t$ if $x \neq y$. Note that when $x \in \bar{l}$, we could have defined $\Downarrow_x (\bar{l}.t)$ to be $\bar{l}.t$. However, this would make the definition less uniform and therefore harder to reason about. To this effect, we proved $\boldsymbol{\nu}x.t \sim t$ if t is closed.

4.1.3 Operational Semantics of try

Handlers of the form $\texttt{try}_n\ e\ \texttt{with}\ x.c$ catch exceptions of the form $\texttt{exc}(n,d)$. For example,

$$\texttt{try}_{\boldsymbol{a}}\ (1 + \texttt{exc}(\boldsymbol{a}, \lambda x.x+1))\ \texttt{with}\ f.f(2) \mapsto^* 3$$

When its principal argument is non-canonical or a variable, a handler computes exactly like the other non-canonical operators (except $\boldsymbol{\nu}$). Let us consider the exception and value cases.

Exception. If t is an exception of the form $\mathtt{exc}(n,d)$ then we have to check whether the handler has the right name as follows:

$$\begin{array}{l}\mathtt{try}_m\ \mathtt{exc}(n,d)\ \mathtt{with}\ x.c\\ \mapsto \mathtt{if}\ m{=}n\ \mathtt{then}\ c[x\backslash d]\ \mathtt{else}\ \mathtt{exc}(n,d)\end{array}$$

This computational rule also has the following effect that if m computes to an exception e, then $\mathtt{try}_m\ \mathtt{exc}(n,d)\ \mathtt{with}\ x.c \mapsto^* e$. Also, if m is a name and n computes to an exception e then $\mathtt{try}_m\ \mathtt{exc}(n,d)\ \mathtt{with}\ x.c \mapsto^* e$.

Value. A naive way or reducing a handler when its principal argument is a value would be to simply return the value as follows:

$$\mathtt{try}_n\ v\ \mathtt{with}\ x.c \mapsto v$$

However, note that in the case where the principal argument of a handler is an exception, we have to evaluate the "name" part of the handler to check whether the exception has the correct name. This means that if we were to simply return the value here and if the "name" part was $\bot$ for example, raising an exception in an expression that is "well-behaved" could cause the expression to diverge. For example, using the above rule: $\mathtt{try}_\bot\ 1 \mapsto 1$, and if we replace 1 by $\mathtt{exc}_a$, then $\mathtt{try}_\bot\ \mathtt{exc}_a$ diverges. This is undesirable, especially in the context of using exceptions to probe a function, e.g., to compute its modulus of continuity. Therefore, instead of simply returning the value, we first check that n is something that we can compare:

$$\mathtt{try}_n\ v\ \mathtt{with}\ x.c \mapsto \mathtt{if}\ n{=}n\ \mathtt{then}\ v\ \mathtt{else}\ \bot$$

4.2 Howe's Computational Equivalence in the Presence of $\boldsymbol{\nu}$

To prove that $\sim$ is a congruence, Howe first proves that $\preccurlyeq$ is a congruence [50]. Unfortunately, this is not easy to prove directly. Howe's "trick" was to define another relation $\preccurlyeq^*$, which is a congruence and contains $\preccurlyeq$ by definition.

Howe's definition of $\preccurlyeq^*$ does not use types, but to account for the fact that the binders of $\boldsymbol{\nu}$ expressions are only meant to be names (as opposed to the binders of, e.g., λ-abstractions, which can be substituted by any term when applied), rather than turn Nuprl into a typed language, we added "simple" type information to the definition of $\preccurlyeq^*$. We define a function $\mathtt{BT}$ that, for a given operator, returns the types of the binders of its bound terms. The type of a binder can either be $\mathtt{NAME}$ or $\mathtt{ANY}$. $\mathtt{BT}(\boldsymbol{\nu}) = [[\mathtt{NAME}]]$ because $\boldsymbol{\nu}$ has one subterm (the outer brackets) that has one binder (the inner brackets). The type of all the other binders is $\mathtt{ANY}$. For example, $\mathtt{BT}(\lambda) = [[\mathtt{ANY}]]$ because a λ-abstraction has one subterm which has one binder. When extending the definition of $\preccurlyeq^*$ from terms to bound terms, $\mathtt{BT}$ is used to restrict what terms can be substituted for free variables. This modification of $\preccurlyeq^*$'s definition was inspired by Gordon's [47] and Jeffrey and Rathke's [52] adaptations of Howe's method to typed λ-calculi. It is interesting to note that until we added the $\boldsymbol{\nu}$ operator to Nuprl, there was no need to use type information in the proof that $\sim$ is a congruence.

Howe defines $t \preccurlyeq^* u$ by induction on t: if t is a variable then $t \preccurlyeq^* u$ if $t \preccurlyeq u$; and if t is of the form $\tau(\overline{b})$ then $t \preccurlyeq^* u$ if there exists $\overline{b'}$ such that $\overline{b} \preccurlyeq^* \overline{b'}$ and $\tau(\overline{b'}) \preccurlyeq u$. To prove that $\preccurlyeq^*$ and $\preccurlyeq$ are equivalent and therefore that $\preccurlyeq$ and $\sim$ are congruences, it suffices to prove that $\preccurlyeq^*$ respects computation, i.e., given that $t \preccurlyeq^* u$, if t computes to a value of the form $\theta(\overline{b})$ then u also computes to a value $\theta(\overline{b'})$ such that $\overline{b} \preccurlyeq^* \overline{b'}$. Howe's Lemma 2 in [50] shows that this is true when t is a value.

Howe then defines a condition called *extensionality* that non-canonical operators of lazy computation systems have to satisfy for $\preccurlyeq^*$ to imply $\preccurlyeq$, and therefore for $\preccurlyeq$ and $\sim$ to be congruences. First, we extended all these definitions to deal with the fact that canonical forms can either be values or exceptions. Then, using our new definition of $\preccurlyeq^*$ we were able to prove that $\boldsymbol{\nu}$ is extensional (see [72, Appendix D] for more details).

4.3 Consistency

As mentioned above, Nuprl's consistency follows from the fact that all its inference rules are valid w.r.t. Allen's PER semantics and from the fact that False is not inhabited. In addition to extending Nuprl's computation system, and fixing its properties including Howe's computational equivalence relation, we had to re-run all the proofs that Nuprl's inference rules are valid. Most of these rules and proofs did not have to change. The only one that had to change is discussed in details in [72, Appendix C] (see [72, Appendices A&B] for details regarding the validity of rules). Let us summarize this discussion here.

First, note that because exceptions are canonical forms as mentioned in Sec. 4.1.1 above, if $a \mapsto^* \mathtt{exc}(t_1,t_2)$ then $a \preccurlyeq b$ if there exists u_1 and u_2 such that $b \mapsto^* \mathtt{exc}(u_1,u_2)$, $t_1 \preccurlyeq u_1$, and $t_2 \preccurlyeq u_2$. Therefore, even though we cannot have a canonical form test (such as $\mathtt{ifint}$ or $\mathtt{ifaxiom}$) for exceptions that would check whether a term computes to an exception because we cannot catch an exception without having its name (i.e., we have no way of catching all exceptions), we can define a proposition $\mathtt{isexc}(t)$ that asserts that a term t computes to an exception as follows: $\mathtt{isexc}(t) = \mathtt{exc}_\bot \preceq t$, where $\mathtt{exc}_\bot = \mathtt{exc}(\bot,\bot)$. Similarly, the following proposition $\mathtt{halts}(t)$ asserts that t computes to a value: $\mathtt{halts}(t) = \star \preceq (\mathtt{let}\ x := t\ \mathtt{in}\ \star)$. By definition of Howe's approximation relation, before adding exceptions, when proving a proposition of the form $t_1 \preceq t_2$ we could assume $\mathtt{halts}(t_1)$. This was captured by our old [convergence] inference rule described in [72, Appendix C]. This is no longer true because we also have to consider the case where t_2 is an exception. To that effect our new [convergence] inference rule generates (among others) two subgoals: one that assumes $\mathtt{halts}(t_1)$ and one that assumes $\mathtt{isexc}(t_1)$. Alternatively, we could capture that a term t computes to either a value or an exception using the type: $\mathtt{exc}_\bot \preceq (\mathtt{let}\ x := t\ \mathtt{in}\ \mathtt{exc}_\bot)$. We have not yet investigated the usefulness of such a type.

4.4 Computing the Modulus of Continuity

We now have the tools in hand to compute the modulus of continuity of a functional using exceptions as described above:

$$\begin{array}{ll}\mathtt{force}(k,t) & = \mathtt{if}\ k{<}0\ \mathtt{then}\ \bot\ \mathtt{else}\ t\\ \mathtt{bound}(n,f,e,k) & = \mathtt{force}(k, \mathtt{if}\ k{<}n\ \mathtt{then}\ f(k)\ \mathtt{else}\ \mathtt{exc}_e)\\ \mathtt{bound}(n,f,e) & = \lambda x.\mathtt{bound}(n,f,e,x)\\ \mathtt{test}(F,n,f) & = \boldsymbol{\nu}x.\mathtt{try}_x\ F(\mathtt{bound}(n,f,x))\\ \mathtt{M}(F) & = \lambda n.\lambda f.\mathtt{test}(F,n,f)\end{array}$$

i.e., unfolding the definitions, $\mathtt{M}(F)$ is

$$\lambda n.\lambda f.\boldsymbol{\nu}x.\mathtt{try}_x\ F\left(\lambda y.\begin{array}{l}\mathtt{if}\ y{<}0\ \mathtt{then}\ \bot\\ \mathtt{else}\ \left(\begin{array}{l}\mathtt{if}\ y{<}n\ \mathtt{then}\ f(y)\\ \mathtt{else}\ \mathtt{exc}_x\end{array}\right)\end{array}\right)$$

Also, let $\mathtt{force}(f) = \lambda x.\mathtt{force}(x, f(x))$. As in our proof of $\mathtt{WCP}_\downarrow$, we will partly use typing, partly use computation to prove that $\mathtt{M}(F)$ is indeed our witness for $\mathtt{SCPT}$. This is why $\mathtt{bound}$ starts off by checking whether its argument x is an integer less than 0. If a computation that uses $\mathtt{bound}$ converges and along the way applies f to some term k, we will be guaranteed that k is a natural number.

4.5 Well-Typedness

To prove $\mathtt{SCPT}$, we first prove that $\mathtt{M}(F)$ has type $\boldsymbol{\Pi}n{:}\mathbb{N}.\mathbb{N}^{\mathbb{N}_n} \to \mathbb{N}_?$. As mentioned above, this term does not respect computation, it is not functional over $F{\in}\mathcal{B} \to \mathbb{N}$. However, given a term F, we can still prove that $\mathtt{M}(F)$ has the right type. For that, we have to prove that for all closed terms n and m such that $n{\equiv}m{\in}\mathbb{N}$, and for all closed terms f and g such that $f{\equiv}g{\in}\mathbb{N}^{\mathbb{N}_n}$, we have $\mathtt{test}(F,n,f){\equiv}\mathtt{test}(F,m,g){\in}\mathbb{N}_?$. By definition, $n{\equiv}m{\in}\mathbb{N}$ means that there exists a natural number k such that both n and m compute to k. Therefore, let us assume $f{\equiv}g{\in}\mathbb{N}^{\mathbb{N}_k}$ and let us prove $\mathtt{test}(F,k,f){\equiv}\mathtt{test}(F,k,g){\in}\mathbb{N}_?$. Unfolding $\mathtt{test}$'s definition, we have to prove

$$\begin{array}{l}\boldsymbol{\nu}x.\mathtt{try}_x\ F(\mathtt{bound}(k,f,x))\\ {\equiv}\boldsymbol{\nu}x.\mathtt{try}_x\ F(\mathtt{bound}(k,g,x))\\ {\in}\mathbb{N}_?\end{array}$$

As we show below in Sec. 4.6, to prove that it is enough to prove

$$\mathtt{try}_{\boldsymbol{a}}\ F(\mathtt{bound}(k,f,\boldsymbol{a})){\equiv}\mathtt{try}_{\boldsymbol{a}}\ F(\mathtt{bound}(k,g,\boldsymbol{a})){\in}\mathbb{N}_?$$

where $\boldsymbol{a}$ is such that $\boldsymbol{a}\#F$, $\boldsymbol{a}\#f$, and $\boldsymbol{a}\#g$. Again, it is enough to prove

$$F(\mathtt{bound}(k,f,\boldsymbol{a})){\equiv}F(\mathtt{bound}(k,g,\boldsymbol{a})){\in}\mathbb{N}_{?\boldsymbol{a}} \quad (1)$$

By typing again, from $f{\equiv}g{\in}\mathbb{N}^{\mathbb{N}_k}$, we deduce that

$$\mathtt{bound}(k,f,\boldsymbol{a}){\equiv}\mathtt{bound}(k,g,\boldsymbol{a}){\in}(\mathbb{N}_{?\boldsymbol{a}})^{\mathbb{N}} \quad (2)$$

A general fact about exceptions is: if $F{\in}\mathcal{B}\rightarrow\mathbb{N}$ and $\boldsymbol{a}\#F$ then

$$\mathtt{Force}(F){\in}(\mathbb{N}_{?\boldsymbol{a}})^{\mathbb{N}}\rightarrow\mathbb{N}_{?\boldsymbol{a}} \quad (3)$$

where $\mathtt{Force}(F)=\lambda f.F(\mathtt{force}(f))$. Is $\mathtt{Force}$ necessary? Can't we simply prove $F{\in}(\mathbb{N}_{?\boldsymbol{a}})^{\mathbb{N}}\rightarrow\mathbb{N}_{?\boldsymbol{a}}$? In other words, can we find a F in $\mathcal{B}\rightarrow\mathbb{N}$, such that $\boldsymbol{a}\#F$, and an f in $(\mathbb{N}_{?\boldsymbol{a}})^{\mathbb{N}}$ such that $F(f)$ is not in $\mathbb{N}_{?\boldsymbol{a}}$? Yes we can: take

$$\begin{array}{l}F\ =\ \lambda f.f(f(0))\\ f\ \ =\ \lambda x.\mathtt{let}\ z:=(\mathtt{try}_{\boldsymbol{a}}\ x\ \mathtt{with}\ z.\bot)\ \mathtt{in}\ \mathtt{exc}_{\boldsymbol{a}}\end{array}$$

(Note that in f's definition $\bot$ could be any term not in $\mathbb{N}$.) These expressions are of the right type, but $F(f)$ computes to $f(f(0))$, which computes to $f(\mathtt{exc}_{\boldsymbol{a}})$, which computes to $\mathtt{let}\ z:=\bot\ \mathtt{in}\ \mathtt{exc}_{\boldsymbol{a}}$, which diverges and is therefore not of type $\mathbb{N}_{?\boldsymbol{a}}$. Proving 3 is the crux of proving that $\mathtt{M}(F)$ is well-typed. To prove that we use the same technique as in $\mathtt{WCP}_{\downarrow}$'s proof. Given 3, it is trivial to deduce that the equality 1 is true using equality 2.

Let us now prove 3. We have to prove that for all F in $\mathcal{B}\rightarrow\mathbb{N}$, and f and g such that $f{\equiv}g{\in}(\mathbb{N}_{?\boldsymbol{a}})^{\mathbb{N}}$,

$$F(\mathtt{force}(f)){\equiv}F(\mathtt{force}(g)){\in}\mathbb{N}_{?\boldsymbol{a}}$$

First, we define a function $\mathtt{force0}$ so that the function $f'=\lambda x.\mathtt{force0}(x,f)$ computes as the function $f_0=\mathtt{force}(f)$, except that on natural numbers, when f_0 returns $\mathtt{exc}_{\boldsymbol{a}}$, f' returns 0 (this 0 could be any natural number):

$$\mathtt{force0}(x,f)=\mathtt{if}\ x{<}0\ \mathtt{then}\ \bot\ \mathtt{else}\ \mathtt{try}_{\boldsymbol{a}}\ f(x)\ \mathtt{with}\ z.0$$

Let $g_0=\mathtt{force}(g)$. Note that $f_0{\equiv}g_0{\in}(\mathbb{N}_{?\boldsymbol{a}})^{\mathbb{N}}$. Because f' is in $\mathcal{B}$, we get that $F(f')$ computes to a natural number. Let us now use again the same simulation technique as before. Let us prove that in any context C with no occurrence of $\boldsymbol{a}$, if $C[f']$ computes to a natural number j, then either both $C[f_0]$ and $C[g_0]$ also compute to j or both $C[f_0]$ and $C[g_0]$ compute to $\mathtt{exc}_{\boldsymbol{a}}$. We prove that by induction on the length of the reduction $C[f']\mapsto^* j$. For that we prove that if $C[f']\mapsto u$ and u computes to a canonical expression, then there exists a context C' such that $u\mapsto^* C'[f']$ and either both $C[f_0]\mapsto^* C'[f_0]$ and $C[g_0]\mapsto^* C'[g_0]$ or both $C[f_0]\mapsto^*\mathtt{exc}_{\boldsymbol{a}}$ and $C[g_0]\mapsto^*\mathtt{exc}_{\boldsymbol{a}}$. This gives us that $F(f_0){\equiv}F(g_o){\in}\mathbb{N}_{?\boldsymbol{a}}$

4.6 Interlude: Reasoning About $\boldsymbol{\nu}$

In Sec. 4.1.2 we saw how $\boldsymbol{\nu}$ computes. We show here how to reason about $\boldsymbol{\nu}$. One can prove that $\boldsymbol{\nu}x.t_1{\equiv}\boldsymbol{\nu}x.t_2{\in}T$ by proving that $t_1[x\backslash\boldsymbol{a}]{\equiv}t_2[x\backslash\boldsymbol{a}]{\in}T$, assuming $\boldsymbol{a}\#t_1$ and $\boldsymbol{a}\#t_2$, and that T is *flat*, meaning that its inhabitants compute to terms that have no subterms and that are not names, such as integers or $\star$. This follows from the way $\boldsymbol{\nu}$ computes. If $t[x\backslash\boldsymbol{a}]\mapsto^* u$ such that $\boldsymbol{a}\#t$ then $\boldsymbol{\nu}v.t\mapsto^*\boldsymbol{\nu}x.u[\boldsymbol{a}\backslash x]$. In the integer case, if $t_1[x\backslash\boldsymbol{a}]\mapsto^* i$ then $\boldsymbol{\nu}x.t_1\mapsto^*\boldsymbol{\nu}x.i$ and $\boldsymbol{\nu}x.i\mapsto i$. We get that $\boldsymbol{\nu}x.t_1\sim t_1[x\backslash\boldsymbol{a}]$. Because the union of flat types is flat, $\mathbb{N}_?$ is flat.

We can prove similar rules for the other types. For example, one can prove that $\boldsymbol{\nu}x.f_1{\equiv}\boldsymbol{\nu}x.f_2{\in}\boldsymbol{\Pi}a{:}A.B$ by proving that $\boldsymbol{\Pi}a{:}A.B$ is a type, and that for all a_1 and a_2 such that $a_1{\equiv}a_2{\in}A$, $\boldsymbol{\nu}x.f_1(a_1){\equiv}\boldsymbol{\nu}x.f_2(a_2){\in}B[a\backslash a_1]$ (see lemma `fresh_in_function` in https://github.com/vrahli/NuprlInCoq/blob/master/continuity/stronger_continuity_props1.v).

4.7 1st Condition

The first property we prove about the function $\mathtt{M}$ defined above in Sec. 4.4 is that for all f in $\mathcal{B}$, $\downarrow\boldsymbol{\Sigma}n{:}\mathbb{N}.\mathtt{M}(F)\ n\ f=_{\mathbb{N}}F(f)$. This condition says that for all f there exists a n such that $\mathtt{M}$ only requires an "initial sequence" of length n of f to compute the same result as $F(f)$. This n is therefore at least the modulus of continuity of F at f.

As before, by typing we get that $F(f){\equiv}F(\mathtt{force}(f)){\in}\mathbb{N}$. Hence, there exists a natural number k such that $F(\mathtt{force}(f))\mapsto^* k$. As in the proof of $\mathtt{WCP}_{\downarrow}$, we first compute the maximum of all the numbers occurring in that computation, and we instantiate our conclusion with b a number which is strictly greater than this maximum. We now have to prove: $\mathtt{M}(F)\ b\ f=_{\mathbb{N}}F(f)$, or equivalently $\mathtt{test}(F,b,f)=_{\mathbb{N}}F(\mathtt{force}(f))$. Unfolding $\mathtt{test}$'s definition, we have to prove:

$$\boldsymbol{\nu}x.\mathtt{try}_x\ F(\mathtt{bound}(b,f,x)){\equiv}F(\mathtt{force}(f)){\in}\mathbb{N}$$

As in Sec. 4.5, because we're trying to prove that this $\boldsymbol{\nu}$ is in $\mathbb{N}$ and because $\mathbb{N}$ is flat, it is enough to prove for some name $\boldsymbol{a}$ such that $\boldsymbol{a}\#F$ and $\boldsymbol{a}\#f$:

$$\mathtt{try}_{\boldsymbol{a}}\ F(\mathtt{bound}(b,f,\boldsymbol{a})){\equiv}F(\mathtt{force}(f)){\in}\mathbb{N}$$

Again, let us use the same simulation technique as before to prove that in any context C with no occurrence of $\boldsymbol{a}$, if $C[\mathtt{force}(f)]\mapsto^* k$ then $C[\mathtt{bound}(b,f,\boldsymbol{a})]\mapsto^* k$. We prove that by induction on the length of the reduction $C[\mathtt{force}(f)]\mapsto^* k$. For that we prove that if $C[\mathtt{force}(f)]\mapsto u$ such that u computes to a canonical expression, and all the numbers occurring in $C[\mathtt{force}(f)]$ are strictly less than b, then there exists a context C' such that $u\mapsto^* C'[\mathtt{force}(f)]$ and $C[\mathtt{bound}(b,f,\boldsymbol{a})]\mapsto^* C'[\mathtt{bound}(b,f,\boldsymbol{a})]$.

Using this result, we get that $F(\mathtt{bound}(b,f,\boldsymbol{a}))\mapsto^* k$, from which we deduce that $\mathtt{try}_{\boldsymbol{a}}\ F(\mathtt{bound}(b,f,\boldsymbol{a}))\mapsto^* k$, and finally $\mathtt{try}_{\boldsymbol{a}}\ F(\mathtt{bound}(b,f,\boldsymbol{a}))=_{\mathbb{N}}F(\mathtt{force}(f))$.

4.8 2nd Condition

The second property we prove about the function $\mathtt{M}$ defined above in Sec. 4.4 is that for all f in $\mathcal{B}$ and n in $\mathbb{N}$, if $\mathtt{M}(F)\ n\ f$ computes to a number then $\mathtt{M}(F)\ n\ f=_{\mathbb{N}}F(f)$. In order to implement our search function to realize $\mathtt{SCP}_{\downarrow}$, we need to return the smallest n, say m, such that $\mathtt{M}(F)\ n\ f$ computes to a number. However, if $\mathtt{M}(F)$ could return different answers for different n's, we would not know whether $\mathtt{M}(F)\ m\ f$ returns $F(f)$ or some other value.

Let us prove that if $\mathtt{M}(F)\ n\ f\sim k$ for some $k\in\mathbb{N}$ then $F(f)\sim k$. As before, we can assume that $\mathtt{try}_{\boldsymbol{a}}\ F(\mathtt{bound}(n,f,\boldsymbol{a}))\sim k$, for some $\boldsymbol{a}$ such that $\boldsymbol{a}\#F$ and $\boldsymbol{a}\#f$. By typing we get that $F(f)$ computes to a natural number k'. Because $\mathtt{try}_{\boldsymbol{a}}\ F(\mathtt{bound}(n,f,\boldsymbol{a}))$ computes to a canonical form (the natural number k), we deduce that $F(\mathtt{bound}(n,f,\boldsymbol{a}))$ also computes to a canonical form. This canonical form is either (1) an exception or (2) a value.

(1) If $F(\mathtt{bound}(n,f,\boldsymbol{a}))$ computes to an exception then we get a contradiction: either the term computes to $\mathtt{exc}_{\boldsymbol{a}}$ and then we obtain that $\mathtt{try}_{\boldsymbol{a}}\ F(\mathtt{bound}(n,f,\boldsymbol{a}))\mapsto^*\star$ and $\star\neq k$; or it

computes to an exception e with some other name than $\boldsymbol{a}$ and then $\mathtt{try}_{\boldsymbol{a}}\ F(\mathtt{bound}(n, f, \boldsymbol{a})) \mapsto^* e$ and $e \neq k$. In both cases we get a contradiction.

(2) We now assume that $F(\mathtt{bound}(n, f, \boldsymbol{a}))$ computes to a value. If so, it has to compute to k. We now prove that $k = k'$. As before, because $F(f){\equiv}F(\mathtt{force}(f)){\in}\mathbb{N}$, we get that $F(\mathtt{force}(f)) \mapsto^* k'$. The rest of this proof closely follows the one in Sec. 4.7. We prove that in any context C with no occurrence of $\boldsymbol{a}$, if $C[\mathtt{force}(f)]$ computes to the natural number k' then $C[\mathtt{bound}(n, f, \boldsymbol{a})]$ computes to either k' or $\mathtt{exc}_{\boldsymbol{a}}$. We prove that by induction on the length of the reduction $C[\mathtt{force}(f)] \mapsto^* k'$. For that we prove that if $C[\mathtt{force}(f)] \mapsto u$ such that u computes to a canonical expression then there exists a context C' such that $u \mapsto^* C'[\mathtt{force}(f)]$ and $C[\mathtt{bound}(b, f, \boldsymbol{a})] \mapsto^* C'[\mathtt{bound}(b, f, \boldsymbol{a})]$ or $C[\mathtt{bound}(b, f, \boldsymbol{a})] \mapsto^* \mathtt{exc}_{\boldsymbol{a}}$. We get that either: (1) $F(\mathtt{bound}(b, f, \boldsymbol{a})) \mapsto^* k'$ and therefore $k = k'$; or (2) $F(\mathtt{bound}(b, f, \boldsymbol{a})) \mapsto^* \mathtt{exc}_{\boldsymbol{a}}$ and we get a contradiction because $k \neq \mathtt{exc}_{\boldsymbol{a}}$.

4.9 Nuprl's Strong Continuity Inference Rule

Using the fact that $\mathtt{SCPT}$ (defined at the beginning of Sec. 4) is true in Nuprl's meta-theory, we proved that the following inference rule, called [StrongContinuity], is true w.r.t. Allen's PER semantics (for further details regarding this proof conducted in our implementation of Nuprl in Coq, the interested reader is invited to look at https://github.com/vrahli/NuprlInCoq/blob/master/continuity/continuity_roadmap.v):

$$\frac{H \vdash F \in (\mathbb{N} \to T) \to \mathbb{N} \qquad H \vdash {\downarrow}T \qquad H \vdash T \sqsubseteq \mathbb{N}}{H \vdash \mathtt{M}(F) \in \mathtt{SCPF}(F)}$$

Using this inference rule, we proved a version of $\mathtt{SCPT}$ in Nuprl, where the first (outer) existential quantifier is $\downharpoonleft$-squashed and the second (inner) one is not squashed (this lemma can be accessed by clicking the following hyperlink: strong-continuity2-no-inner-squash).[6] We get rid of the second squash operator using the usual unbounded search μ operator. As expected the extract of that lemma is (we use colored parentheses for visual convenience):

$$\lambda F.\langle \mathtt{M'}(F), \lambda f.\langle \mu(\lambda n.\mathtt{isl}(\mathtt{test'}(F, n, f))), \langle \star, \lambda m.\lambda i.\star\rangle\rangle\rangle$$

where

$$\begin{aligned}\mathtt{test'}(F, n, f) = {} & \mathtt{let}\ x := \mathtt{test}(F, n, f)\ \mathtt{in} \\ & \mathtt{ifint}(x, \mathtt{inl}(x), \mathtt{inr}(\star))\end{aligned}$$

$$\mathtt{M'}(F) = \lambda n.\lambda f.\mathtt{test'}(F, n, f)$$

$$\mu(f) = \mathtt{fix}\left(\begin{aligned}\lambda F.\lambda n.\ & \mathtt{if}\ f(n)\ \mathtt{then}\ n \\ & \quad \mathtt{else}\ \mathtt{let}\ m := n + 1\ \mathtt{in}\ F(m)\end{aligned}\right)\ 0$$

We then derived a version of $\mathtt{SCP}$ where, as mentioned in the introduction, the first (outer) $\underline{\boldsymbol{\Sigma}}$ is $\downharpoonleft\boldsymbol{\Sigma}$ and the second (inner) one is $\boldsymbol{\Sigma}$ (see/click Nuprl lemma strong-continuity2-no-inner-squash-unique). Therefore, because these versions of $\mathtt{SCP}$ are equivalent of $\mathtt{SCP}_{\downharpoonleft}$, we also refer to them as $\mathtt{SCP}_{\downharpoonleft}$.

5. Relations Between $\mathtt{WCP}$ and $\mathtt{SCP}$

As mentioned in Sec. 1, Bridges and Richman [19, pp.119] state that $\mathtt{SCP}$ is equivalent to $\mathtt{WCP}$ plus some form of the axiom of choice—some version of $\mathrm{AC}_{1,0}$. As we saw above the existential quantifiers in these statements have to be truncated using $\downharpoonleft$ (see Escardó and Xu [39]). Therefore, for that equivalence to be true, we probably need the $\downharpoonleft$-truncated version of $\mathrm{AC}_{1,0}$, which is true in Nuprl as discussed below in Sec. 5.3. Therefore, we only need to prove that $\mathtt{SCP}_{\downharpoonleft}$ and $\mathtt{WCP}_{\downharpoonleft}$ are equivalent. We prove below that $\mathtt{WCP}_{\downharpoonleft}$ is a trivial consequence of $\mathtt{SCP}_{\downharpoonleft}$, and that $\mathtt{SCP}_{\downharpoonleft}$ is a consequence of $\mathtt{WCP}_{\downharpoonleft}$ using the same "trick" as the one used by Bridges and Richman to prove that $\mathtt{UCP}$ follows from the Fan Theorem and $\mathtt{WCP}$ [19, pp.113] (see Sec. 6.1 below on uniform continuity).

5.1 $\mathtt{SCP}_{\downharpoonleft}$ Implies $\mathtt{WCP}_{\downharpoonleft}$

Let us sketch the proof that $\mathtt{SCP}_{\downharpoonleft} \to \mathtt{WCP}_{\downharpoonleft}$. For convenience, we will write $\mathbb{N}_{\mathtt{U}}$ for $\mathbb{N}{+}\mathtt{Unit}$. Let $F \in \mathcal{B} \to \mathbb{N}$ and $f \in \mathcal{B}$. We prove the formula $C = {\downharpoonleft}\boldsymbol{\Sigma} n{:}\mathbb{N}.\ \boldsymbol{\Pi} g{:}\mathcal{B}.f =_{\mathbb{N}^{\mathbb{N}_n}} g \to F(f) =_{\mathbb{N}} F(g)$. From $\mathtt{SCP}_{\downharpoonleft}$, we get a $M \in \boldsymbol{\Pi} n{:}\mathbb{N}.\mathbb{N}^{\mathbb{N}_n} \to \mathbb{N}_{\mathtt{U}}$ (we can unsquash $\mathtt{SCP}_{\downharpoonleft}$'s outer $\boldsymbol{\Sigma}$ because our conclusion C is squashed) and a function

$$\begin{aligned}A \in \boldsymbol{\Pi} f{:}\mathcal{B}.\boldsymbol{\Sigma} n{:}\mathbb{N}.\quad & M\ n\ f =_{\mathbb{N}_{\mathtt{U}}} \mathtt{inl}(F(f)) \\ & \wedge \boldsymbol{\Pi} m{:}\mathbb{N}.\mathtt{isl}(M\ m\ f) \to m =_{\mathbb{N}} n\end{aligned}$$

By applying A to f we get a $n \in \mathbb{N}$ such that:

- $M\ n\ f =_{\mathbb{N}_{\mathtt{U}}} \mathtt{inl}(F(f))$
- and $B \in \boldsymbol{\Pi} m{:}\mathbb{N}.\mathtt{isl}(M\ m\ f) \to m =_{\mathbb{N}} n$.

We unsquash and instantiate with n our conclusion C, and we now get to assume that there is a $g \in \mathcal{B}$ such that $f =_{\mathbb{N}^{\mathbb{N}_n}} g$. It remains to prove that $F(f) =_{\mathbb{N}} F(g)$. By applying A to g we get a $n' \in \mathbb{N}$ such that:

- $M\ n'\ g =_{\mathbb{N}_{\mathtt{U}}} \mathtt{inl}(F(g))$
- and $B' \in \boldsymbol{\Pi} m{:}\mathbb{N}.\mathtt{isl}(M\ m\ g) \to m =_{\mathbb{N}} n'$.

Because $f =_{\mathbb{N}^{\mathbb{N}_n}} g$, we get that $M\ n\ f =_{\mathbb{N}_{\mathtt{U}}} M\ n\ g$. Because $M\ n\ f =_{\mathbb{N}_{\mathtt{U}}} \mathtt{inl}(F(f))$, we get $\mathtt{isl}(M\ n\ f)$ and therefore also $\mathtt{isl}(M\ n\ g)$. Then, by applying B' to n we get $n =_{\mathbb{N}} n'$. Therefore, $\mathtt{inl}(F(f)) =_{\mathbb{N}_{\mathtt{U}}} \mathtt{inl}(F(g))$, and finally we get that $F(f) =_{\mathbb{N}} F(g)$.

5.2 $\mathtt{WCP}_{\downharpoonleft}$ Implies $\mathtt{SCP}_{\downharpoonleft}$

In this section we prove that $\mathtt{skWCP}$ implies $\mathtt{SCP}_{\downharpoonleft}$ (see Nuprl lemma weak-continuity-implies-strong1), where $\mathtt{skWCP}$ is the following "skolemized" version of $\mathtt{WCP}_{\downharpoonleft}$:

$$\begin{aligned}\mathtt{skWCP} = {} & \boldsymbol{\Pi} F{:}\mathcal{B} \to \mathbb{N}. \\ & \quad {\downharpoonleft}\boldsymbol{\Sigma} M{:}\mathcal{B} \to \mathbb{N}. \\ & \qquad \boldsymbol{\Pi} f, g{:}\mathcal{B}.(f =_{\mathbb{N}_{M(f)} \to \mathbb{N}} g) \to (F(f) =_{\mathbb{N}} F(g))\end{aligned}$$

It is easy to prove that $\mathtt{skWCP}$ and $\mathtt{WCP}_{\downharpoonleft}$ are equivalent using our proof that the $\downharpoonleft$-truncated axiom of choice $\mathrm{AC}_{1,\mathrm{X}}$ discussed below in Sec. 5.3 is true (see Nuprl lemma axiom-choice-1X-quot).

Let us assume $\mathtt{skWCP}$ and let $F \in \mathcal{B} \to \mathbb{N}$. We have to prove the following formula C:

$$\begin{aligned}& {\downharpoonleft}\boldsymbol{\Sigma} M{:}\boldsymbol{\Pi} n{:}\mathbb{N}.\mathbb{N}^{\mathbb{N}_n} \to \mathbb{N}_{\mathtt{U}}. \\ & \quad \boldsymbol{\Pi} f{:}\mathcal{B}.\quad \boldsymbol{\Sigma} n{:}\mathbb{N}.M\ n\ f =_{\mathbb{N}_{\mathtt{U}}} \mathtt{inl}(F(f)) \\ & \qquad\qquad \wedge \boldsymbol{\Pi} n{:}\mathbb{N}.\mathtt{isl}(M\ m\ f) \to M\ n\ f =_{\mathbb{N}_{\mathtt{U}}} \mathtt{inl}(F(f))\end{aligned}$$

Instantiating $\mathtt{skWCP}$ with F we get (because our conclusion C is $\downharpoonleft$-truncated, we can unsquash our hypothesis):

- a $M \in \mathcal{B} \to \mathbb{N}$ (F' modulus of continuity)
- and a $G \in \boldsymbol{\Pi} f, g{:}\mathcal{B}.(f =_{\mathbb{N}_{M(f)} \to \mathbb{N}} g) \to (F(f) =_{\mathbb{N}} F(g))$

Bridges and Richman's trick is to also use the modulus of continuity of M. Therefore, instantiating $\mathtt{skWCP}$ with M we get:

- a $X \in \mathcal{B} \to \mathbb{N}$ (M's modulus of continuity)
- and a $K \in \boldsymbol{\Pi} f, g{:}\mathcal{B}.(f =_{\mathbb{N}_{X(f)} \to \mathbb{N}} g) \to (M(f) =_{\mathbb{N}} M(g))$

Let us now unsquash our conclusion C and instantiate it with

$$B = \lambda n.\lambda a.\mathtt{if}\ M(a_{n,0}) \leq n\ \mathtt{then}\ \mathtt{inl}(F(a_{n,0}))\ \mathtt{else}\ \mathtt{inr}(\star)$$

[6] Alternatively, the reader can search for the lemma with that name available here: http://www.nuprl.org/LibrarySnapshots/Published/Version1/Standard/continuity/index.html, and similarly for the other Nuprl lemmas mentioned below and highlighted in green (or gray).

where $a_{n,k} = \lambda x.\texttt{if}\ x{<}n\ \texttt{then}\ a(x)\ \texttt{else}\ k$ (this is the infinite sequence consisting of the first n values of a followed by k's—$a_{n,0}$ is sometimes denoted $\overline{a,n}$, e.g., in [12, 38]). If $a \in \mathbb{N}^{\mathbb{N}_n}$ and $k \in \mathbb{N}$ then $a_{n,k} \in \mathbb{N} \to \mathbb{N}$. We now have to prove that assuming that $f \in \mathcal{B}$ then:

$$\mathbf{\Sigma} n{:}\mathbb{N}.B\ n\ f =_{\mathbb{N}_\mathtt{U}} \texttt{inl}(F(f)) \tag{4}$$

$$\mathbf{\Pi} n{:}\mathbb{N}.\texttt{isl}(B\ n\ f) \to (B\ n\ f =_{\mathbb{N}_\mathtt{U}} \texttt{inl}(F(f))) \tag{5}$$

Equality 5 follows from G. We now prove 4 by instantiating it with $m = \texttt{max}(M(f), X(f))$. We have to prove $B\ m\ f =_{\mathbb{N}_\mathtt{U}} \texttt{inl}(F(f))$, i.e.,

$$\begin{aligned}&\texttt{if}\ M(a_{m,0}) \le m\ \texttt{then}\ \texttt{inl}(F(a_{m,0}))\ \texttt{else}\ \texttt{inr}(\star) \\ &=_{\mathbb{N}_\mathtt{U}} \texttt{inl}(F(f))\end{aligned} \tag{6}$$

Because $f =_{\mathbb{N}_{M(f)} \to \mathbb{N}} a_{m,0}$, then using G we obtain: $F(f) =_\mathbb{N} F(a_{m,0})$. Therefore, to prove equality 6, it remains to prove that its conditional is true, i.e., $M(a_{m,0}) \le m = \texttt{max}(M(f), X(f))$. If we instantiate K with f and $a_{m,0}$, we have to prove $f =_{\mathbb{N}_{X(f)} \to \mathbb{N}} a_{m,0}$, which is true by definition of m, and get to assume $M(f) =_\mathbb{N} M(a_{m,0})$ which gives us that $M(a_{m,0}) \le m$.

5.3 Axiom of Choice

The following axiom of choice is usually trivial in constructive type theories such as Nuprl when $\underline{\mathbf{\Sigma}}$ is $\mathbf{\Sigma}$ (where A and B are types):

$$AC = \mathbf{\Pi} a{:}A.\underline{\mathbf{\Sigma}} b{:}B.\ P\ a\ b\ \Rightarrow\ \underline{\mathbf{\Sigma}} f{:}B^A.\ \mathbf{\Pi} a{:}A.P\ a\ f(a)$$

It follows from the usual rules of the universal and existential quantifiers. We can prove that these rules are true in our predicative Coq model [7, 8] without assuming any axiom. In that predicative model we can model n Nuprl universes using $n+1$ Coq universes. However, in our impredicative model we have to assume some axiom of choice, namely FunctionalChoice_on (see http://coq.inria.fr/cocorico/CoqAndAxioms), to prove some of these rules.

However, the non-squashed version of AC is not always enough because as we saw above existential quantifiers cannot always be interpreted as $\mathbf{\Sigma}$ but sometimes as truncated $\mathbf{\Sigma}$'s. Therefore, we sometimes need instances of AC where $\underline{\mathbf{\Sigma}}$ is either $\downharpoonleft\mathbf{\Sigma}$ or $\downarrow\mathbf{\Sigma}$. In that case it is not obvious anymore which instances of AC are consistent or provable in Nuprl.

Some versions of AC for particular choices of types A and B are of particular interest. One can often find in the literature the name AC$_{n,m}$, where $n, m \in \{0,1\}$ [82, pp.238]: $n = 0$ means that $A = \mathbb{N}$ and $n = 1$ means that $A = \mathcal{B}$; similarity $m = 0$ means that $B = \mathbb{N}$ and $m = 1$ means that $B = \mathcal{B}$.

Using a technique similar to the one discussed in [71], we proved the $\downarrow$-squashed version of AC$_{0,0}$, where $\underline{\mathbf{\Sigma}}$ is $\downarrow\mathbf{\Sigma}$, once again conducting the proof first in the meta-theory, and then reflecting the meta-theoretical result in the Nuprl theory as an inference rule: see lemma `rule_AC00_true` in https://github.com/vrahli/NuprlInCoq/blob/master/axiom_choice/axiom_choice.v. We also proved directly in Nuprl the $\downharpoonleft$-squashed versions of AC, where $\underline{\mathbf{\Sigma}}$ is $\downharpoonleft\mathbf{\Sigma}$, namely AC$_{0,\mathrm{X}}$ (see Nuprl lemma axiom-choice-0X-quot) and AC$_{1,\mathrm{X}}$ (see Nuprl lemma axiom-choice-1X-quot)—X here indicates that the type B above could be anything. These two lemmas are instances of the more general Nuprl lemma: axiom-choice-quot.

6. Applications

Using SCP$_\downharpoonleft$ we proved in Nuprl a $\downharpoonleft$-truncated version of UCP (see Sec. 1) from the fan theorem, and then a fully unsquashed version of this principle using Escardó and Xu's method [39] (see Sec. 6.1). We write UCP$_\downharpoonleft$ for the version of UCP where $\underline{\mathbf{\Sigma}}$ is $\downharpoonleft\mathbf{\Sigma}$. Using UCP$_\downharpoonleft$ we then proved that real functions defined on the unit interval are uniformly continuous (see Sec. 6.2).

We have recently proved that Bar Induction on Decidable bars (BID) is consistent with Nuprl using our Coq model of Nuprl [71], and this for free choice sequences of natural numbers. Following Kleene, given that SCP$_\downharpoonleft$ is true we can now prove Bar Induction on Monotone bars (BIM) [53, pp.78] (see also Dummett's Thm 3.8 [37, pp.64]): see Nuprl lemma monotone-bar-induction1.

We have also recently used exceptions to implement the constructive content of the completeness result of intuitionistic first-order logic proved in [27]. As in the present paper, exceptions are used to probe how a computation uses its arguments. Given a uniform evidence for a proposition, we construct a proof of that proposition by using this probing mechanism to determine the next step of the proof.

6.1 Uniform Continuity

6.1.1 UCP$_\downharpoonleft$ Follows From FT and SCP$_\downharpoonleft$.

Using BID we prove that the Fan Theorem (FT) is true: see Nuprl lemma fan_theorem. We then derive UCP$_\downharpoonleft$ from FT and SCP$_\downharpoonleft$: see Nuprl lemma strong-continuity2-implies-uniform-continuity. Let us sketch that proof. The version of FT that we have proved in Nuprl is (if X is a bar and is decidable then it is uniform):

$$\begin{aligned}\texttt{FT} = {} & \mathbf{\Pi} X{:}(\mathbf{\Pi} n{:}\mathbb{N}.2^{\mathbb{N}_n} \to \mathbb{P}).\\ & \quad \mathbf{\Pi} f{:}\mathcal{C}.{\downarrow}\mathbf{\Sigma} n{:}\mathbb{N}.X\ n\ f \\ & \quad \to\ \mathbf{\Pi} n{:}\mathbb{N}.\mathbf{\Pi} f{:}2^{\mathbb{N}_n}.\texttt{Dec}(X\ n\ f) \\ & \quad \to\ \mathbf{\Sigma} k{:}\mathbb{N}.\mathbf{\Pi} f{:}\mathcal{C}.\mathbf{\Sigma} n{:}\mathbb{N}_k.X\ n\ f\end{aligned}$$

We also use the following corollary of SCP$_\downharpoonleft$ for functions on the Cantor space instead of the Baire space:

$$\begin{aligned}&\texttt{SCPB}\\ &= \mathbf{\Pi} F{:}\mathcal{C} \to \mathbb{N}.\\ &\qquad \downharpoonleft \mathbf{\Sigma} M{:}(\mathbf{\Pi} n{:}\mathbb{N}.2^{\mathbb{N}_n} \to \mathbb{N}_\mathtt{U}).\\ &\qquad\quad \mathbf{\Pi} f{:}\mathcal{C}.\\ &\qquad\qquad \mathbf{\Sigma} n{:}\mathbb{N}.M\ n\ f =_{\mathbb{N}_\mathtt{U}} \texttt{inl}(F(f))\\ &\qquad\qquad \wedge\ \mathbf{\Pi} n{:}\mathbb{N}.\texttt{isl}(M\ n\ f) \to M\ n\ f =_{\mathbb{N}_\mathtt{U}} \texttt{inl}(F(f))\end{aligned}$$

Let us start proving UCP$_\downharpoonleft$. Let F be in $\mathcal{C} \to \mathbb{N}$. We have to prove

$$\downharpoonleft\mathbf{\Sigma} n{:}\mathbb{N}.\ \mathbf{\Pi} f, g{:}\mathcal{C}.f =_{2^{\mathbb{N}_n}} g \to F(f) =_\mathbb{N} F(g) \tag{7}$$

We start by instantiating SCPB with F and we unsquash the resulting formula (which we can do because our conclusion is squashed), i.e., we get to assume:

- $M \in \mathbf{\Pi} n{:}\mathbb{N}.2^{\mathbb{N}_n} \to \mathbb{N}_\mathtt{U}$ and
- $G \in \mathbf{\Pi} f{:}\mathcal{C}.\quad \mathbf{\Sigma} n{:}\mathbb{N}.M\ n\ f =_{\mathbb{N}_\mathtt{U}} \texttt{inl}(F(f))$
  $\wedge\ \mathbf{\Pi} n{:}\mathbb{N}.\texttt{isl}(M\ n\ f) \to M\ n\ f =_{\mathbb{N}_\mathtt{U}} \texttt{inl}(F(f))$

We now instantiate FT using $X = \lambda n.\lambda f.\texttt{isl}(M\ n\ f)$. We now have to prove (1) $\mathbf{\Pi} f{:}\mathcal{C}.{\downarrow}\mathbf{\Sigma} n{:}\mathbb{N}.\texttt{isl}(M\ n\ f)$, which follows from G; and (2) $\mathbf{\Pi} n{:}\mathbb{N}.\mathbf{\Pi} f{:}2^{\mathbb{N}_n}.\texttt{Dec}(\texttt{isl}(M\ n\ f))$, which is trivial; and we get a $k \in \mathbb{N}$ and $A \in \mathbf{\Pi} f{:}\mathcal{C}.\mathbf{\Sigma} n{:}\mathbb{N}_k.\texttt{isl}(M\ n\ f)$. We unsquash and instantiate our conclusion 7 using $k-1$. We have to prove that $F(f) =_\mathbb{N} F(g)$ assuming that $f =_{2^{\mathbb{N}_k}} g$ for f and g in $\mathcal{C}$. From G we get:

- $G(f) \in \mathbf{\Pi} n{:}\mathbb{N}.\texttt{isl}(M\ n\ f) \to M\ n\ f =_{\mathbb{N}_\mathtt{U}} \texttt{inl}(F(f))$
- $G(g) \in \mathbf{\Pi} n{:}\mathbb{N}.\texttt{isl}(M\ n\ g) \to M\ n\ g =_{\mathbb{N}_\mathtt{U}} \texttt{inl}(F(g))$

and from A (applying A to f) we get a $i \in \mathbb{N}_k$ and that $\texttt{isl}(M\ i\ f)$. Therefore, because $f =_{2^{\mathbb{N}_k}} g$, we get $M\ i\ f =_{\mathbb{N}_\mathtt{U}} M\ i\ g$ and $\texttt{isl}(M\ i\ g)$. Which means that (from $G(f)$ and $G(g)$):

- $M\ i\ f =_{\mathbb{N}_\mathtt{U}} \texttt{inl}(F(f))$
- $M\ i\ g =_{\mathbb{N}_\mathtt{U}} \texttt{inl}(F(g))$

We conclude that $F(f) =_\mathbb{N} F(g)$.

6.1.2 UCP$_\downharpoonleft$ Follows From FT and skWCP.

Following Bridges and Richman's proof of their Theorem 3.2 [19, pp.113], the following Nuprl lemma proves that UCP$_\downharpoonleft$ follows from FT and `skWCP`: fan+weak-continuity-implies-uniform-continuity (as mentioned in Sec. 5, `skWCP` and $\texttt{WCP}_{\downarrow}$ are equivalent). Their proof is slightly more involved than the one presented above in Sec. 6.1.1 and uses the "trick" of building a bar that uses both the skolemized modulus of continuity M of type $\mathcal{C} \to \mathbb{N}$ of a functional F of type $\mathcal{C} \to \mathbb{N}$ and the (skolemized) modulus of continuity of M. As mentioned above `skWCP` is a trivial consequence of $\texttt{SCP}_{\downarrow}$: see Nuprl lemma strong-continuity2-implies-weak-skolem-cantor-nat.

6.1.3 Unsquashed `UCP` Follows From $\texttt{UCP}_{\downarrow}$.

We can then get rid of the truncation operator $\downarrow$ in $\texttt{UCP}_{\downarrow}$ following exactly Escardó and Xu's proof [39, Sec.4]: see Nuprl lemma strong-continuity2-implies-uniform-continuity2. Their proof consists in proving that (1) the existence of a uniform modulus of continuity is equivalent to (2) the existence of the smallest uniform modulus of continuity. Because (2) is a proposition in HoTT's sense [83], we can "untruncate" it. Their proof goes in three steps: Nuprl lemma uniform-continuity-pi-dec corresponds to their Lemma 4; Nuprl lemma prop-truncation-implies corresponds to their Lemma 5; and Nuprl lemma uniform-continuity-pi-pi-prop2 corresponds to their Lemma 6.

6.2 Brouwer's Theorem on Uniform Continuity

In Nuprl, a real number $\alpha : \mathbb{R}$ is a regular sequence of integers. This means that $\alpha : \mathbb{N}^+ \to \mathbb{Z}$ and $\forall n, m. |n * \alpha(m) - m * \alpha(n)| \leq 2(n+m)$. This differs from, but is equivalent to, Bishop's definition of real numbers as regular sequences of rationals [15]. Two regular sequences α and β represent the same real number if $\forall n.\ |\alpha(n) - \beta(n)| \leq 4$, and this is an equivalence relation, $\alpha =_r \beta$, on regular sequences. If $\alpha(n) + 4 < \beta(n)$ for some n, then $\alpha < \beta$, and $\alpha \# \beta$ (α is apart from β) if $\alpha < \beta \vee \beta < \alpha$. If $\forall n. \alpha(n) \leq \beta(n) + 4$, then $\alpha \leq \beta$.

The closed interval $[\alpha, \beta]$ is the type $\{x : \mathbb{R} \mid \alpha \leq x \leq \beta\}$. Bishop calls a member f of the type $[\alpha, \beta] \to \mathbb{R}$ an *operation* on the interval $[\alpha, \beta]$, and reserves the word *function* for those operations that satisfy

$$\text{FUN}(f, \alpha, \beta) = \forall x, y : [\alpha, \beta].\ x =_r y \Rightarrow f(x) =_r f(y)$$

A stronger condition—called strong extensionality in Coq's CoRN library [54]—is

$$\text{SFUN}(f, \alpha, \beta) = \forall x, y : [\alpha, \beta].\ f(x) \# f(y) \Rightarrow x \# y$$

An operation is uniformly continuous on $[\alpha, \beta]$ if

$$\begin{aligned} &\text{CONT}(f, \alpha, \beta) \\ &= \forall \epsilon > 0. \\ &\quad \exists \delta > 0.\ \forall x, y : [\alpha, \beta].\ |x - y| \leq \delta \to |f(x) - f(y)| \leq \epsilon \end{aligned}$$

(The Nuprl lemmas mentioned below are available at the following addess: `http://www.nuprl.org/LibrarySnapshots/Published/Version1/Standard2/reals/index.html`.) Using the fact from Sec. 6.1 that functionals of type $\mathcal{C} \to \mathbb{Z}$ are uniformly continuous, we proved in Nuprl that for *proper* intervals $[\alpha, \beta]$ (where $\alpha < \beta$), we have (see Nuprl lemma real-continuity4):

$$\text{CONT}(f, \alpha, \beta) \Leftrightarrow \text{FUN}(f, \alpha, \beta)$$

In the proof, we construct the usual map from $\mathcal{C}$ onto $[\alpha, \beta]$, using a tree of nested, decreasing intervals. However, we can not show that this is an onto map without using the condition $\alpha < \beta$. Using the fact that we can decide whether a functional of type $\mathcal{C} \to \mathbb{Z}$ is constant, we could extend the proof to the case of intervals that are not necessarily proper—for which only $\alpha \leq \beta$—to show that (see Nuprl lemma real-continuity3):

$$\text{CONT}(f, \alpha, \beta) \Leftrightarrow \text{SFUN}(f, \alpha, \beta)$$

Thus, for Bishop's definition of real function, it is correct to say that all functions on the unit interval $[0, 1]$ are uniformly continuous (Brouwer's theorem). But it is not correct to say that functions are uniformly continuous on all closed intervals $[\alpha, \beta]$—only on proper closed intervals—unless the strongly extensional definition of real function is used. The two definitions are equivalent only when Markov's principle holds [15].

7. Related Work on Nominal Systems

Nominal systems. There has been a tremendous amount of work on nominal approaches to logic and programming starting from Gabbay and Pitts' work on using Fraenkel-Mostowski's permutation model of set theory to formally reason about abstract syntax in the presence of α-equivalence and variable binding [44]. This work then led to the design of the so-called Nominal Logic [65], which provides primitives and axioms to reason about names, name-swapping, freshness, and name-binding. These ideas were then later used and implemented in programming languages and type theories [16, 20–23, 35, 43, 64, 66, 67, 69, 70, 77–79, 88] (to cite only a few). We know describe some of these systems.

FreshML. For example, FreshML [67, 79] is an extension of ML with constructs for declaring and manipulating data with binding structure that provide support for object-level α-equivalence, such as constructs for binding names, declaring new types of bindable names, and generating fresh names. Nuprl does not yet have such a name-abstraction construct. Also, our paper does not try to tackle the issue of reasoning about α-equivalence classes of terms using names. This provides an alternative approach to, e.g., using de Bruijn indices or HOAS in order to deal with names and binders. These ideas were then also ported to OCaml [78]. Following this line of work, Pure FreshML [69] is a pure (in the sense that name generation is not an observable side effect) version of FreshML [79] that ensures fresh atoms do not escape their scopes.

Nominal type theories. Schopp and Stark [76, 77] developed a *bunched* dependent type theory for programming and reasoning with names, based on a categorical axiomatization of names, and taking freshness as the central primitive instead of swapping. Bunches are typing contexts that have a tree-like shape instead of a list-like shape, where branching is used to model the disjointness of names spaces. In their theory, α-equivalence classes can be either modeled as "fresh functions" or as pairs, which are members of "non-standard fresh" $\mathbf{\Pi}^*$ and $\mathbf{\Sigma}^*$ types. It turns out that these types are isomorphic when indexed by names, giving rise to a *hidden-name* type constructor $\mathbf{H}$ which can either be interpreted as a sum or a product, and which corresponds to Gabbay and Pitts freshness quantifier И [44]. In our paper, we only focus on computational aspects of freshness.

Cheney designed SNTT [20], which is a nominal simply-typed λ-calculus with names as well as name-abstraction and name-concretion operators (but no name-generation operator such as $\boldsymbol{\nu}$). SNTT contexts are expressive enough so that one can state the freshness constraint on name-concretions. It is also designed with decidable typechecking in mind. Cheney extended SNTT to a dependent type theory, called $\lambda^{\mathbf{\Pi}И}$ [21], with dependent products ($\mathbf{\Pi}$) and dependent name-abstraction types (И). As for SNTT, one of his main focus was to provide a strongly normalizing theory with decidable type checking.

Westbrook's CNIC calculus (the Calculus of Nominal Inductive Constructions) [87, 88] can also be seen as an extension of SNTT with inductive constructions, or similarly as an extension of CIC [32] with nominal features such as name abstraction and concretion operators, and pattern matching operators for names and name abstractions.

Pitts' Nominal System T [66] extends System T with nominal features such as a fresh operator $\boldsymbol{\nu}$ à la Odersky and a name-swapping operator. As opposed to the systems mentioned above, this system has ordinary non-bunched contexts. FreshMLTT [64] is a dependent type theory that has name-abstraction and name-concretion operators, as well as a name-swapping operator and a fresh operator $\boldsymbol{\nu}$ à la Odersky, which allow them to derive expressive name-concretion rules.

8. Conclusion and Future Work

This paper provides "mostly" computational proofs of two Brouwerian continuity principles that use (1) diverging terms to prove that the modulus of continuity of a function on the Baire space exists in the meta-theory (Sec. 3), and (2) named exceptions to exhibit it in the theory (Sec. 4). We proved that all functions of type $T^{\mathbb{N}} \to \mathbb{N}$ are continuous, where T is a subtype of $\mathbb{N}$. It is not clear how to adapt our proof for other types than subtypes of $\mathbb{N}$. This is left for future work.

In Sec. 4.8 we used the fact that the exception $\mathtt{exc}_{\boldsymbol{a}}$ cannot be caught by F if $\boldsymbol{a}\#F$. This would not longer be true if our computation system was non-deterministic or if we allowed parallel computations. For example, let $t_1 \mid\mid t_2$ be an operator that dovetails the computations of t_1 and t_2. If t_1 computes to $\mathtt{exc}_{\boldsymbol{a}}$ then this exception might get "caught" if t_2 computes to a canonical expression "before" t_1. Once we add non-determinism to Nuprl, we might be able to use non-deterministic computations to compute the modulus of continuity of functions in a similar fashion as done by Coquand and Jaber [31]. This is left for future work.

Finally, many more inference rules can be derived (and verified in our Coq model) from the definitions of our new computations and types than the ones discussed in Sec. 4. Investigating these rules is left for future work.

Acknowledgements

We would like to thank our colleagues Robert L. Constable, Ross Tate, Rich Eaton, Abhishek Anand, and Evan Moran for their helpful criticism.

References

[1] *LICS 2007*. IEEE Computer Society, 2007.

[2] The Agda wiki. `http://wiki.portal.chalmers.se/agda/pmwiki.php`.

[3] Stuart Allen. An abstract semantics for atoms in Nuprl. Technical report, Cornell University, 2006.

[4] Stuart F. Allen. A non-type-theoretic definition of Martin-Löf's types. In *LICS*, pages 215–221. IEEE Computer Society, 1987.

[5] Stuart F. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language*. PhD thesis, Cornell University, 1987.

[6] Stuart F. Allen, Mark Bickford, Robert L. Constable, Richard Eaton, Christoph Kreitz, Lori Lorigo, and Evan Moran. Innovations in computational type theory using Nuprl. *J. Applied Logic*, 4(4):428–469, 2006. `http://www.nuprl.org/`.

[7] Abhishek Anand and Vincent Rahli. Towards a formally verified proof assistant. In *ITP 2014*, volume 8558 of *LNCS*, pages 27–44. Springer, 2014.

[8] Abhishek Anand and Vincent Rahli. Towards a formally verified proof assistant. Technical report, Cornell University, 2014. `http://www.nuprl.org/html/Nuprl2Coq/`.

[9] Jeremy Avigad. Forcing in proof theory. *Bulletin of Symbolic Logic*, 10(3):305–333, 2004.

[10] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *J. Log. Algebr. Meth. Program.*, 84(1):108–123, 2015.

[11] Michael J. Beeson. *Foundations of Constructive Mathematics*. Springer, 1985.

[12] Ulrich Berger and Paulo Oliva. Modified bar recursion. *Mathematical Structures in Computer Science*, 16(2):163–183, 2006.

[13] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development*. SpringerVerlag, 2004. `http://www.labri.fr/perso/casteran/CoqArt`.

[14] Mark Bickford. Unguessable atoms: A logical foundation for security. In *Verified Software: Theories, Tools, Experiments, Second Int'l Conf.*, volume 5295 of *LNCS*, pages 30–53. Springer, 2008.

[15] E. Bishop and D. Bridges. *Constructive Analysis*. Springer, 1985.

[16] Mikolaj Bojanczyk, Laurent Braud, Bartek Klin, and Slawomir Lasota. Towards nominal computation. In *POPL'12*, pages 401–412. ACM, 2012.

[17] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda - a functional language with dependent types. In *TPHOLs 2009*, volume 5674 of *LNCS*, pages 73–78. Springer, 2009. `http://wiki.portal.chalmers.se/agda/pmwiki.php`.

[18] Edwin Brady. Idris —: systems programming meets full dependent types. In *5th ACM Workshop Programming Languages meets Program Verification, PLPV 2011*, pages 43–54. ACM, 2011.

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

[20] James Cheney. A simple nominal type theory. *Electr. Notes Theor. Comput. Sci.*, 228:37–52, 2009.

[21] James Cheney. A dependent nominal type theory. *Logical Methods in Computer Science*, 8(1), 2012.

[22] James Cheney and Christian Urban. alpha-prolog: A logic programming language with names, binding and a-equivalence. In *ICLP 2004*, volume 3132 of *LNCS*, pages 269–283. Springer, 2004.

[23] James Cheney and Christian Urban. Nominal logic programming. *ACM Trans. Program. Lang. Syst.*, 30(5), 2008.

[24] Paul J. Cohen. The independence of the continuum hypothesis. *the National Academy of Sciences of the United States of America*, 50(6):1143–1148, December 1963.

[25] Paul J. Cohen. The independence of the continuum hypothesis ii. *the National Academy of Sciences of the United States of America*, 51(1):105–110, January 1964.

[26] R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P.Panangaden, J.T. Sasaki, and S.F. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[27] Robert Constable and Mark Bickford. Intuitionistic completeness of first-order logic. *Annals of Pure and Applied Logic*, 165(1):164–198, January 2014.

[28] Robert L. Constable. Constructive mathematics as a programming logic I: some principles of theory. In *Fundamentals of Computation Theory, Proceedings of the 1983 International*, volume 158 of *LNCS*, pages 64–77. Springer, 1983.

[29] Robert L. Constable and Scott F. Smith. Computational foundations of basic recursive function theory. *Theoretical Computer Science*, 121(1&2):89–112, 1993.

[30] Thierry Coquand and Guilhem Jaber. A note on forcing and type theory. *Fundam. Inform.*, 100(1-4):43–52, 2010.

[31] Thierry Coquand and Guilhem Jaber. A computational interpretation of forcing in type theory. In *Epistemology versus Ontology*, volume 27 of *Logic, Epistemology, and the Unity of Science*, pages 203–213. Springer, 2012.

[32] Thierry Coquand and Christine Paulin. Inductively defined types. In *COLOG-88, Int'l Conf. on Computer Logic*, volume 417 of *LNCS*, pages 50–66. Springer, 1988.

[33] The Coq Proof Assistant. `http://coq.inria.fr/`.

[34] Karl Crary. *Type-Theoretic Methodology for Practical Programming Languages*. PhD thesis, Cornell University, Ithaca, NY, August 1998.

[35] Roy L. Crole and Frank Nebel. Nominal lambda calculus: An internal language for fm-cartesian closed categories. *Electr. Notes Theor. Comput. Sci.*, 298:93–117, 2013.

[36] R. David and G. Mounier. An intuitionistic λ-calculus with exceptions. *J. Funct. Program.*, 15(1):33–52, January 2005.

[37] Michael A. E. Dummett. *Elements of Intuitionism*. Clarendon Press, second edition, 2000.

[38] Martín Escardó and Paulo Oliva. Bar recursion and products of selection functions. *J. Symb. Log.*, 80(1):1–28, 2015.

[39] Martín Escardó and Chuangjie Xu. The inconsistency of a Brouwerian continuity principle with the Curry-Howard interpretation. TLCA'2015, available at `http://www.cs.bham.ac.uk/~mhe/papers/escardo-xu-inconsistency-continuity.pdf`, 2015.

[40] Martín Hötzel Escardó. Infinite sets that admit fast exhaustive search. In *LICS 2007* [1], pages 443–452.

[41] Martín Hötzel Escardó. Exhaustible sets in higher-type computation. *Logical Methods in Computer Science*, 4(3), 2008.

[42] Martín Hötzel Escardó. Continuity of Gödel's system T definable functionals via effectful forcing. *Electr. Notes Theor. Comput. Sci.*, 298:119–141, 2013.

[43] Elliot Fairweather, Maribel Fernández, Nora Szasz, and Alvaro Tasistro. Dependent types for nominal terms with atom substitutions. In *TLCA 2015*, volume 38 of *LIPIcs*, pages 180–195. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[44] Murdoch Gabbay and Andrew M. Pitts. A new approach to abstract syntax involving binders. In LICS'1999 [58], pages 214–224.

[45] W. Gielen, Harrie C. M. de Swart, and Wim Veldman. The continuum hypothesis in intuitionism. *J. Symb. Log.*, 46(1):121–136, 1981.

[46] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, 1989.

[47] Andrew D. Gordon. Bisimilarity as a theory of functional programming. *Electr. Notes Theor. Comput. Sci.*, 1:232–252, 1995.

[48] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation.*, volume 78 of *LNCS*. Springer-Verlag, 1979.

[49] Martin Hofmann. *Extensional concepts in intensional type theory*. PhD thesis, University of Edinburgh, 1995.

[50] Douglas J. Howe. Equality in lazy computation systems. In *LICS 1989*, pages 198–203. IEEE Computer Society, 1989.

[51] Idris. `http://www.idris-lang.org/`.

[52] Alan Jeffrey and Julian Rathke. Towards a theory of bisimulation for local names. In LICS'1999 [58], pages 56–66.

[53] S.C. Kleene and R.E. Vesley. *The Foundations of Intuitionistic Mathematics, especially in relation to recursive functions*. North-Holland Publishing Company, 1965.

[54] Robbert Krebbers and Bas Spitters. Type classes for efficient exact real arithmetic in Coq. *Logical Methods in Computer Science*, 9(1), 2011.

[55] Jean-Louis Krivine. *Lambda-calculus, types and models*. Ellis Horwood series in computers and their applications. Masson, 1993.

[56] Sylvain Lebresne. A system F with call-by-name exceptions. In *ICALP 2008*, volume 5126 of *LNCS*, pages 323–335. Springer, 2008.

[57] Sylvain Lebresne. A type system for call-by-name exceptions. *Logical Methods in Computer Science*, 5(4), 2009.

[58] *LICS 1999*. IEEE Computer Society, 1999.

[59] John Longley. When is a functional program not a functional program? In *ICFP'99*, pages 1–7. ACM, 1999.

[60] Steffen Lösch and Andrew M. Pitts. Relating two semantics of locally scoped names. In *CSL 2011*, volume 12 of *LIPIcs*, pages 396–411. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

[61] Gregory H. Moore. The origins of forcing. In *Logic Colloquium '86*, pages 143–173. Elsevier Science Publishers B.V. (North-Holland), 1988.

[62] Dag Normann. Computing with functionals - computability theory or computer science? *Bulletin of Symbolic Logic*, 12(1):43–59, 2006.

[63] Martin Odersky. A functional theory of local names. In *POPL'94*, pages 48–59. ACM Press, 1994.

[64] A. M. Pitts, J. Matthiesen, and J. Derikx. A dependent type theory with abstractable names. In I. Mackie and M. Ayala-Rincon, editors, *Proceedings of the LSFA 2014 Workshop*, volume 312 of *Electronic Notes in Theoretical Computer Science*, pages 19–50. Elsevier, 2015.

[65] Andrew M. Pitts. Nominal logic: A first order theory of names and binding. In *TACS 2001*, volume 2215 of *LNCS*, pages 219–242. Springer, 2001.

[66] Andrew M. Pitts. Nominal system T. In *POPL'10*, pages 159–170. ACM, 2010.

[67] Andrew M. Pitts and Murdoch Gabbay. A metalanguage for programming with bound names modulo renaming. In *MPC 2000*, volume 1837 of *LNCS*, pages 230–255. Springer, 2000.

[68] Andrew M. Pitts and Ian D. B. Stark. Observable properties of higher order functions that dynamically create local names, or what's new? In *MFCS'93*, volume 711 of *LNCS*, pages 122–141. Springer, 1993.

[69] François Pottier. Static name control for FreshML. In *LICS 2007* [1], pages 356–365.

[70] Nicolas Pouillard and François Pottier. A fresh look at programming with names and binders. In *ICFP 2010*, pages 217–228. ACM, 2010.

[71] Vincent Rahli and Mark Bickford. Coq as a metatheory for Nuprl with bar induction. Presented at CCC 2015, available at `http://www.nuprl.org/html/Nuprl2Coq/barind.pdf`, 2015.

[72] Vincent Rahli and Mark Bickford. A nominal exploration of intuitionism. Extended version avaible at `http://www.nuprl.org/html/Nuprl2Coq/continuity-long.pdf`, 2015.

[73] Vincent Rahli, Mark Bickford, and Abhishek Anand. Formal program optimization in Nuprl using computational equivalence and partial types. In *ITP'13*, volume 7998 of *LNCS*, pages 261–278. Springer, 2013.

[74] Michael Rathjen. Constructive set theory and brouwerian principles. *J. UCS*, 11(12):2008–2033, 2005.

[75] Jorge Luis Sacchini. Exceptions in dependent type theory. Presented at TYPES'14 (`http://www.pps.univ-paris-diderot.fr/types2014/abstract-18.pdf`), 2014.

[76] Ulrich Schöpp. *Names and Binding in Type Theory*. PhD thesis, University of Edinburgh, 2006.

[77] Ulrich Schöpp and Ian Stark. A dependent type theory with names and binding. In *CSL 2004*, volume 3210 of *LNCS*, pages 235–249. Springer, 2004.

[78] Mark R. Shinwell. Fresh O'Caml: Nominal abstract syntax for the masses. *Electr. Notes Theor. Comput. Sci.*, 148(2):53–77, 2006.

[79] Mark R. Shinwell, Andrew M. Pitts, and Murdoch James Gabbay. FreshML: programming with binders made simple. *SIGPLAN Notices*, 38(9):263–274, 2003.

[80] Scott F. Smith. *Partial Objects in Type Theory*. PhD thesis, Cornell University, Ithaca, NY, 1989.

[81] A.S. Troelstra. Aspects of constructive mathematics. In *Handbook of Mathematical Logic*, pages 973–1052. North-Holland Publishing Company, 1977.

[82] A.S. Troelstra and D. van Dalen. *Constructivism in Mathematics An Introduction*, volume 121 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 1988.

[83] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `http://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

[84] Mark van Atten and Dirk van Dalen. Arguments for the continuity principle. *Bulletin of Symbolic Logic*, 8(3):329–347, 2002.

[85] Wim Veldman. Understanding and using Brouwers continuity principle. In *Reuniting the Antipodes Constructive and Nonstandard Views of the Continuum*, volume 306 of *Synthese Library*, pages 285–302. Springer Netherlands, 2001.

[86] Frank Waaldijk. On the foundations of constructive mathematics – especially in relation to the theory of continuous functions. *Foundations of Science*, 10(3):249–324, 2005.

[87] Edwin M. Westbrook. *Higher-Order Encodings with Constructors*. PhD thesis, Washington University, Saint Louis, Missouri, 2008.

[88] Edwin M. Westbrook, Aaron Stump, and Evan Austin. The calculus of nominal inductive constructions: an intensional approach to encoding name-bindings. In *LFMTP '09*, pages 74–83. ACM, 2009.

[89] Chuangjie Xu and Martín Hötzel Escardó. A constructive model of uniform continuity. In *TLCA 2013*, volume 7941 of *LNCS*, pages 236–249. Springer, 2013.