Math. Struct. in Comp. Science (2018), *vol.* 28, *pp.* 942–990. © Cambridge University Press 2017
doi:10.1017/S0960129517000172 First published online 2 November 2017

Validating Brouwer's continuity principle for numbers using named exceptions

VINCENT RAHLI[†] and MARK BICKFORD[‡]

[†]*SnT, University of Luxembourg, Esch-sur-Alzette, Luxembourg*
Email: vincent.rahli@gmail.com
[‡]*Cornell University, Computer Science Department, Ithaca, NY, USA*
Email: markb@cs.cornell.edu

Received 11 May 2016; revised 18 March 2017

This paper extends the Nuprl proof assistant (a system representative of the class of extensional type theories with dependent types) 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 metatheory using our formalization of Nuprl in Coq and reflect these metatheoretical results in the Nuprl theory as derivation rules. We also show that these additions preserve Nuprl's key metatheoretical 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, bar induction on monotone bars and the negation of the law of excluded middle.

1. Introduction

Continuity. There are two principles that distinguish Brouwer's mathematics from other constructive mathematics, namely *bar induction* and a *continuity principle* (Beeson 1985; Bridges and Richman 1987; Dummett 1977; Howard and Kreisel 1966; Kleene and Vesley 1965; Rathjen 2005, 2006; Troelstra 1977a; Troelstra and VanDalen 1988; van Atten and Dalen 2002; Veldman 2001, 2006, 2014). In this document, we consider the following weak and strong continuity principles on the Baire space $\mathcal{B} = \mathbb{N}^{\mathbb{N}} = \mathbb{N} \to \mathbb{N}$, i.e., the type of infinite sequences of natural numbers ($\boldsymbol{\Pi}$,$\boldsymbol{\Sigma}$ are the logical $\forall$,$\exists$ of constructive type theory (CTT); as explained below, $\underline{\boldsymbol{\Sigma}}$ is an existential quantifier – we shall consider several forms below; $+$ 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; $\mathcal{B}_n$ is $\mathbb{N}^{\mathbb{N}_n}$; and $t =_T u$ expresses that t and u are equal terms in the type T):

$$\mathtt{WCP} = \boldsymbol{\Pi} F{:}\mathcal{B} \to \mathbb{N}.\boldsymbol{\Pi} f{:}\mathcal{B}.\underline{\boldsymbol{\Sigma}} n{:}\mathbb{N}.\ \boldsymbol{\Pi} g{:}\mathcal{B}.f =_{\mathcal{B}_n} g \to F(f) =_{\mathbb{N}} F(g)$$

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

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

WCP is the *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[‡] to prove bar induction on monotone bars (BIM) from bar induction on decidable bars (BID) (Kleene and Vesley 1965, p. 78). SCP says that there is a uniform way, called M in the formula [such a function is often called a neighborhood function (Troelstra and VanDalen 1988, p. 212)], to decide whether n is the modulus of continuity of F at f, and if so returns the value $F(f)$ (Kleene and Vesley 1965, pp. 70–71).

Traditionally, Brouwer's continuity principle is stated using relations instead of functions as in WCP and SCP. We show in Section 6.4 that we can derive a more traditional continuity principle that uses relations.

Truncation/Squashing. As first shown by Kreisel in Kreisel (1962, p. 154), continuity is not an extensional property in the sense that it does not map equal arguments to equal values. Therefore, the existence of M in SCP's type has to be 'weakened' in some way. Troelstra later showed in Troelstra (1977b, Theorem IIA) that N-HA$^\omega$ [which is a 'neutral' version of HA$^\omega$ that 'permits extensional as well as intensional interpretations of equality at higher types' (Troelstra 1975)] extended with (1) Brouwer's continuity principle, (2) a function extensionality axiom and (3) a version of the unsquashed axiom of choice $\mathrm{AC}_{2,0}$, is inconsistent. Both Kreisel's and Troelstra's proofs rely on function extensionality. Escardo and Xu (2015) and Xu (2015) proved, without using function extensionality but using a rule that allows one to reduce under λs, that WCP is false in (both intensional and extensional) MLTT (MartinLof 1984), i.e., when $\underline{\boldsymbol{\Sigma}}$ is the sum type $\boldsymbol{\Sigma}$ of MLTT. They also formally proved this result using Agda (Agda Wiki, available at: http://wiki.portal.chalmers.se/agda/pmwiki.php; Bove *et al.* 2009). As pointed out by Escardó and Xu, what these results show is that 'although every function is continuous, there is no continuous way of finding a modulus of continuity of a give function f at a point α' (Escardo and Xu 2015). However, as they mention in their paper, Brouwer's continuity principle is consistent when $\boldsymbol{\Sigma}$ is *truncated at the propositional level* (The Univalent Foundations Program 2013, p. 117). In Nuprl's type theory (Allen *et al.* 2006; Constable *et al.* 1986), propositional truncation corresponds to *squashing* a type down to a single equivalence class (i.e., all inhabitants are equal) using quotient types (Constable

[†] In both WCP and SCP, f is declared to have type $\mathcal{B}$ and is used with type $\mathcal{B}_n$. This is possible in Nuprl without any coercion because $\mathcal{B}$ is a subtype of $\mathcal{B}_n$, i.e., the members of $\mathcal{B}$ are members of $\mathcal{B}_n$ too. This follows from the fact that $\mathbb{N}_n$ is a subtype of $\mathbb{N}$, which is made possible thanks to use of set types: $\mathbb{N}_n = \{x : \mathbb{N} \mid x < n\}$, and a member of a set type of the form $\{x : A \mid B\}$ is a member of A, and not a pair of an a in A and a b in $B[x\backslash a]$ – see for example Constable *et al.* (1986) for more details regarding set types.

[‡] Rathjen calls it 'Strong Continuity for Numbers' (Rathjen 2005) and names it C-N [as in Troelstra and VanDalen (1988)]. Dummett refers to it as 'a stronger version of the Continuity Principle,' names it $\mathrm{CP}_{\exists n}$ and later calls it 'the Continuity Principle' (Dummett 1977, pp. 59–60). Troelstra (1977a, p. 1006) calls it CONT_0. This is Kleene's *27.2 principle (Kleene and Vesley 1965, p. 73), which he calls 'Brouwer's principle (for numbers).' Bridges and Richman (1987, p. 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 $\mathrm{AC}_{1,0}$ (see Section 5.3). Note that $\mathrm{AC}_{1,0}$ is also sometimes used to refer to SCP (Gielen *et al.* 1981; Waaldijk 2005).

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

1983): $\downarrow T = T//\texttt{True}$. Because the members of a quotient type of the form $T//E$ are the same as the members of T, the members of $\downarrow T$ are the members of T. Also, $\downarrow T$ is a proof-irrelevant type, i.e., its members are all equal to each other because if $x, y \in T$, then $(x =_{\downarrow T} y \iff \texttt{True})$. 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 Constable *et al.* (1986, p. 60)]. The only member of this type is the constant $\star$, which can be thought of as `()` in, e.g., OCaml, Haskell or SML. It 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 $\downarrow T \rightarrow \downarrow T$ is true because it is inhabited by $\lambda x.\star$, but we cannot prove the converse because to prove $\downarrow T$ 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 $\downarrow T$). It turns out that the following $\downarrow$-version of the law of excluded middle $\boldsymbol{\Pi} P$:$\texttt{Type}.\downarrow(P+\neg P)$ is consistent with Nuprl (Anand and Rahli 2014a), while both the non-squashed version $\boldsymbol{\Pi} P$:$\texttt{Type}.P+\neg P$ and the $\downarrow$-squashed version $\boldsymbol{\Pi} P$:$\texttt{Type}.\downarrow(P+\neg P)$ are false, where as usual $\neg P$ is defined as $P \rightarrow \texttt{False}$. This means, that if $\downarrow T \rightarrow \downarrow T$ was consistent with Nuprl, then $\boldsymbol{\Pi} P$:$\texttt{Type}.\downarrow(P+\neg P)$ would not be anymore. Section 2.6 presents derivable inference rules that one can use to reason about these two squashing operators.

In this paper, we prove that squashed versions of `WCP` and `SCP` are true facts about Nuprl's functions. We carry out these proofs in Nuprl's metatheory (Allen 1987a,b) using our formalization of Nuprl in Coq (The Coq Proof Assistant, available at: http://coq.inria.fr/; Bertot and Casteran 2004), which contains among other things the following: (1) an implementation of Nuprl's computation system; (2) an implementation of Howe's computational equivalence relation (Howe 1989) and a proof that it is a congruence; (3) a definition of Nuprl's CTT, where types are interpreted as *Partial Equivalence Relations* (PERs) on closed terms following Allen's PER semantics (Allen 1987a,b); (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 (Anand and Rahli 2014a,b).

In Section 3, we prove `WCP` where $\underline{\boldsymbol{\Sigma}} x{:}T.\ P$ is defined as $\downarrow\boldsymbol{\Sigma} x{:}T.P$, and refer to this principle as $\texttt{WCP}_{\downarrow}$. We call $\texttt{WCP}_{\downarrow}$ the version of `WCP`, where $\underline{\boldsymbol{\Sigma}}$ is $\downarrow\boldsymbol{\Sigma}$. In Section 4, we prove `SCP` where the first (outer) $\underline{\boldsymbol{\Sigma}}$ is $\downarrow\boldsymbol{\Sigma}$ and the second (inner) is $\downarrow\boldsymbol{\Sigma}$, and refer to this principle as $\texttt{SCP}_{\downarrow}$. These 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 $\texttt{SCP}_{\downarrow}$ without the second (inner) squashing operator. Section 5 shows that $\texttt{SCP}_{\downarrow}$ and $\texttt{WCP}_{\downarrow}$ are equivalent. Even though the implication $\texttt{WCP}_{\downarrow} \rightarrow \texttt{WCP}_{\downarrow}$ is trivial, we believe our proof of $\texttt{WCP}_{\downarrow}$ in Section 3 is valuable because of its simplicity and because $\downarrow$ is often enough.

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

Effectful computations. Following Longley's method (Longley 1999), we use computational effects (Bauer and Pretnar 2015), namely named exceptions, to derive $\texttt{SCP}_{\downarrow}$. The

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

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 subroutine (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`.[§]

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 (`let _ :=` $f(m+1)$ `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 can 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 (Section 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. For example, Troelstra (1973, p. 158) proved that every closed term $t \in \mathbb{N}^{\mathcal{B}}$ of N-HA$^{\omega}$ possesses a provable modulus of continuity in N-HA$^{\omega}$ (the method used in Section 3 bears some resemblance with his) – see also Beeson (1985) for similar proofs of the consistency of continuity with various constructive theories. Coquand and Jaber (2010, 2012) proved the *uniform* continuity of a Martin-Löf-like intensional type theory using *forcing* (Avigad 2004; Beeson 1985; Cohen 1963, 1964; Moore 1988). 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 subgraphs 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 ($\mathcal{C}_n$ is $2^{\mathbb{N}_n}$):

$$\mathtt{UCP} = \boldsymbol{\Pi} F{:}\mathcal{C} \to \mathbb{N}.\underline{\boldsymbol{\Sigma}} n{:}\mathbb{N}.\ \boldsymbol{\Pi} f,g{:}\mathcal{C}.f =_{\mathcal{C}_n} g \to F(f) =_{\mathbb{N}} F(g).$$

Escardo and Xu (2015) and Xu (2015) showed that in the case of uniform continuity, $\underline{\boldsymbol{\Sigma}}$ can equivalently be $\boldsymbol{\Sigma}$ or $\downarrow\boldsymbol{\Sigma}$.

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

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

In Coquand and Jaber (2012), Coquand and Jaber provide a Haskell realizer that computes the uniform modulus of continuity of a functional on the Cantor space.[†]

Similarly, Xu and Escardo (2013) proved that the definable functionals of Gödel's system T (Girad *et al.* 1989) 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 $\mathcal{C} \to \mathbb{N}$ 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 Escardó (2013), 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 Coquand and Jaber (2010) 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 $\alpha : \mathbb{N}^{\mathbb{N}}$' (Escardó 2013). Such a computation is called a *dialogue*, which is a function that given a dialogue tree and an oracle of type $\mathcal{B}$, returns a number. 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 Section 3, we use diverging terms to prove $\mathtt{WCP}_{\downarrow}$, and in Section 4, we use computational effects (named exceptions) to probe terms and derive $\mathtt{SCP}_{\downarrow}$ using (non-strict) lock-step simulations of these effectful computations. Section 5 shows that $\mathtt{SCP}_{\downarrow}$ and $\mathtt{WCP}_{\downarrow}$ are equivalent. To prove $\mathtt{SCP}_{\downarrow}$, we added named exceptions as well as a *fresh* operator to Nuprl's computation system, and showed that these additions preserve Nuprl's key metatheoretical properties, such as consistency (see Sections 2.3 and 4.3) and the congruence of Howe's computational equivalence relation (see Sections 2.2 and 4.2). As mentioned in Section 4.9, $\mathtt{SCP}_{\downarrow}$ justifies a corresponding inference rule that we added to Nuprl. Section 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 Section 6, we have proved in Nuprl the

[†] 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 Escardo (2007, 2008) and Normann (2006), which point to citations and constructions by, among others, Gandy and Berger.

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

following: (1) a fully unsquashed version of UCP using Escardó and Xu's method (Escardo and Xu 2015); (2) that all real functions defined on the unit interval are uniformly continuous (Dummett 1977, p. 87); (3) that BIM follows from BID following Kleene's proof (Kleene and Vesley 1965, pp. 69–73); (4) the negation of the law of excluded middle (this is also provable in Nuprl without using continuity and using instead Nuprl's 'computation types' – see Section 6.3); and (5) a version of Brouwer's continuity principle stated using relations instead of functions.

The results presented in this paper have either been formalized in Coq and are available at https://github.com/vrahli/NuprlInCoq; or they have been formalized in Nuprl and are available at http://www.nuprl.org/LibrarySnapshots/Published/Version2/Standard/continuity/index.html for results related to continuity, at http://www.nuprl.org/LibrarySnapshots/Published/Version2/Standard/int_2/index.html for results related to the fan theorem and at http://www.nuprl.org/LibrarySnapshots/Published/Version2/Mathematics/reals/index.html for results related to real analysis.

Most of the results presented here were presented in the following conference paper: Rahli and Bickford (2016). In addition, we present here many more details regarding Nuprl and its semantics in Section 2. Some of these details were also presented in the following conference paper: Anand and Rahli (2014a). We also cover more related work regarding exception-based type/logical-systems in Section 4.1.2. We present many more details regarding the effects of adding nominal features to Nuprl in Sections 4.2 and 4.3. Finally, Section 6 discusses additional consequences of continuity, such as the law of excluded middle in Section 6.3, and alternative versions of the continuity principle in Sections 6.4 and 6.5.

2. Nuprl

Nuprl is an interactive theorem prover that implements a type theory called CTT (Allen *et al.* 2006; Constable *et al.* 1986). CTT 'mostly' differs from other similar constructive type theories such as the ones implemented by Agda (Agda Wiki, available at: http://wiki.portal.chalmers.se/agda/pmwiki.php; Bove *et al.* 2009), Coq (The Coq Proof Assistant, available at: http://coq.inria.fr/; Bertot and Casteran 2004) or Idris (Idris, available at: http://www.idris-lang.org/; Brady 2011), in the sense that CTT is an *extensional* type theory [i.e., propositional and definitional equality are identified (Hofmann 1995)] with types of partial functions (Constable and Smith 1993; Crary 1998; Smith 1989). This section presents some key aspects of Nuprl that will be used in the rest of this paper.

2.1. *Nuprl's computation system*

Figure 1 presents a subset of Nuprl's syntax, and Figure 2 presents a subset of Nuprl's small-step operational semantics (Allen *et al.* 2006; Anand and Rahli 2014b). We only show the part that is either used or mentioned in this paper. 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

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

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

Fig. 1. Syntax of a subset of Nuprl.

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

Fig. 2. Operational semantics of a subset of Nuprl.

such as addition, subtraction,. . . , a test for equality and a 'less than' operator. Nuprl also has what we call *canonical form tests* (Rahli *et al.* 2013) 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. A value is a weak-head normal form, i.e., the term is not a redex itself, but can contain subterms that could reduce further. For example, $\texttt{inl}(\texttt{fix}(\lambda x.x))$ is a value even though $\texttt{fix}(\lambda x.x)$ can reduce further. Note that types and terms belong to the same syntactic category: all types are values and all values are terms. Non-canonical terms have one or two *principal*

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

arguments (marked using boxes in Figure 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 (Anand and Rahli 2014a,b; Howe 1989), and the terms in Figure 1 are 'display forms' for some specific Nuprl terms. In Nuprl, all terms are of the form $\theta(\overline{b})$, where θ is called the *operator* of the term, and $\overline{b}$ is a list of *bound terms*. A bound term b is of the form $l.t$, where l is a list of variables. For example, the underlying representation of a λ-abstraction, displayed to users as $\lambda x.t$, is $\{\text{lambda}\}(x.t)$, i.e., its operator θ is $\{\text{lambda}\}$ and it has a single bound term, namely $x.t$. Note that $\{\text{lambda}\}(t)$ is also a term of the language, which we qualify as ill-formed because its subterm has no bound variables. In our Coq formalization, terms are defined follows:

```
Inductive NTerm : Set :=
| vterm (v : NVar)
| oterm (op : Opid) (bs : list BTerm)
with BTerm : Set :=
| bterm (l : list NVar) (t : NTerm).
```

where NVar is the type of variables (which are natural numbers in our implementation); Opid is type of Nuprl operators; NTerm is the type of Nuprl terms; and BTerm is the type of Nuprl bound terms. Note that we distinguish variables from the other terms because several operations, such as substitution, operate differently on variables from the other terms: vterm is the variable constructor, while oterm builds a term from an operator and a list of bound terms.

An advantage of using 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. For convenience, we use the uniform syntax to, e.g., define Howe's computational equivalence relation below.

Figure 2 shows part of Nuprl's small-step operational semantics. We omit the rules that reduce principal arguments such as the following: 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 abbreviations:

$$\begin{array}{llll} \bot & = \texttt{fix}(\lambda x.x) & \mathbb{N}_? & = \mathbb{N} \cup \texttt{Unit} & \texttt{isint}(t) & = \texttt{ifint}(t, \texttt{tt}, \texttt{ff}) \\ \texttt{tt} & = \texttt{inl}(\star) & \mathbb{N}_\cup & = \mathbb{N} + \texttt{Unit} & \texttt{isl}(t) & = \texttt{if } t \texttt{ then tt else ff} \\ \texttt{ff} & = \texttt{inr}(\star) & & & & \end{array}$$

$$\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 = \texttt{case } t_1 \texttt{ of inl}(x) \Rightarrow t_1 \mid \texttt{inr}(x) \Rightarrow t_2.$$

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. (Alternatively, Unit and Void could be defined as $\star \preceq \star$ and $\star \preceq \bot$, respectively.) We define True as Unit and False as Void.

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

2.2. *Howe's computational equivalence*

It turns out that CTT is not only closed under computation but more generally under Howe's computational equivalence $\sim$, which he proved to be a congruence (Howe 1989). 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 (Rahli *et al.* 2013).

Before we formally define Howe's computational equivalence relation, let us point out that a binary relation on closed terms can always be extended to a relation on open terms as follows: $t_1\ R\ t_2$ if for all closed substitutions s such that $t_1[s]$ and $t_1[s]$ are closed, $t_1[s]\ R\ t_2[s]$. A relation on terms can always be extended to a relation on bound terms as follows: $l_1.t_1\ R\ l_2.t_2$ if $t_1[l_1\backslash l]\ R\ t_2[l_2\backslash l]$. A relation on terms or bound terms can always be extended to a relation on lists of terms or bound terms as follows: $(t_1,\ldots,t_n)\ R\ (u_1,\ldots,u_n)$ if $t_i\ R\ u_i$ for all $i \in \{1,\ldots,n\}$.

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 $\overline{b}\ R\ \overline{b'}$. In Coq, this translates as follows:

```
CoInductive approx (t u : NTerm) : Type :=
| approx_fold : close_compute approx t u → approx t u,
```

where NTerm is the type of Nuprl terms; and where close_compute is defined as follows:

```
Definition close_compute (R : NTerm → NTerm → Type) (t u : NTerm) : Type :=
  program t
  ∧ program u
  ∧ ∀ (op : Opid) (bs : list BTerm),
       (t ⇓ oterm op bs)
       → exists bs', (u ⇓ oterm op bs') ∧ lblift (olift R) bs bs'.
```

where (program t) states that the term t is closed; $(t \Downarrow u)$ states that t computes to the value u; lblift lifts a binary relation on bound terms to a binary relation on lists of bound terms; and olift lifts a binary relation on closed terms to a binary relation on open terms.

By definition, one can derive, e.g., that $\bot \preccurlyeq t$ for all closed term t. This is because $t \preccurlyeq u$ is trivially true if t diverges, because $(t \Downarrow \text{oterm}\ op\ bs)$ is false in the above definition of close_compute. Moreover, it follows that one can prove that $t \sim u$ if both t and u diverge.

We show below in Sections 2.3 and 2.5 that Nuprl provides a way to reason about Howe's approximation and computational equivalence relations by providing types that are true/inhabited whenever the corresponding metatheoretical relations are true.

2.3. *Nuprl's type system*

In CTT, a type is a value of the computation system. Among other things, Allen's PER semantics associates PERs to these values, i.e., types are interpreted as PERs on closed terms (Allen 1987a,b). 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 $T \equiv T$,

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

and $t \in T$ for $t \equiv t \in T$. Among other things, it follows that the equality type $a = b \in T$ is a true theoretical proposition (inhabited by $\star$) iff $a \equiv b \in T$ holds in the metatheory. Also, if the equality type $T = U \in \mathbb{U}_i$ is true, then $T \equiv U$ holds in the metatheory. The converse is not necessarily true, because, for example, $\mathtt{type}(\boldsymbol{\Pi} n{:}\mathbb{N}.\mathbb{U}(n))$ and $\mathtt{type}(\cup n{:}\mathbb{N}.\mathbb{U}(n))$ hold in the metatheory, while they are not in any universe (if universe levels range over natural numbers). See Anand and Rahli (2014a,b) for more details.

Let us now explain how the two relations $T_1 \equiv T_2$ and $a \equiv b \in T$ are defined in our Coq formalization. See Allen (1987b), Anand and Rahli (2014a,b) and Crary (1998) for more details. Let CTerm be the type of closed terms. Let per, the type of PERs, be defined as CTerm→CTerm→Prop. Let TS, the type of type systems, be defined as CTerm→CTerm→per→Prop. First, we define the semantics of CTT's type constructors as follows. The interpretation of a CTT type constructor is a function of type TS→CTerm→CTerm→per→Prop. We only define here the semantics of product types:

```
Definition per_pi (ts : TS) (T1 T2 : CTerm) (eq : per) : Prop :=
  exists eqa eqb,
      type_family Pi ts T1 T2 eqa eqb
      ∧ (∀ t t', eq t t' ⇔(∀ a a' (e : eqa a a'), eqb a a' e (Apply t a) (Apply t' a'))).
```

where (1) Pi and Apply are functions that build product and application Nuprl terms, respectively; (2) the first conjunct defines when two product types are equal and the second one defines the PER of such a product type; and where (3) type_family is defined as follows:

```
Definition type_family TyCon ts T1 T2 eqa eqb : Prop :=
  exists A A' v v' B B',
     T1 ⇓ (TyCon A v B)
     ∧ T2 ⇓ (TyCon A' v' B')
     ∧ ts A A' eqa
     ∧ (∀ a a' (e : eqa a a'), ts (B[v\a]) (B'[v'\a']) (eqb a a' e)).
```

Next, we inductively define a close operator on type systems. Given a type system ts, this operator builds another candidate type system, which is closed w.r.t. the type constructors mentioned above:

```
Inductive close (ts : TS) (T T' : CTerm) (eq : per) : Prop :=
  | CL_init : ts T T' eq → close ts T T' eq
  | CL_pi : per_pi (close ts) T T' eq → close ts T T' eq
  ...
```

Then, we interpret the Nuprl term $\mathbb{U}(i)$, which is the universe type at level i, by recursion on $i \in \mathbb{N}$ as follows:

```
Fixpoint univi (i : nat) (T T' : CTerm) (eq : per) : Prop :=
  match i with
  | 0 ⇒ False
  | S n ⇒ (T ⇓ U(n)
           ∧ T' ⇓ U(n)
           ∧ ∀ A A', eq A A' ⇔ exists eqa, close (univi n) A A' eqa)
         ∨ univi n T T' eq
  end.
```

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

Note that cumulativity follows from the use of $\vee$ in the above definition, i.e., if two types are equal in $\mathbb{U}(n)$, then they will also be equal in $\mathbb{U}(m)$ if $n \leqslant m$. Next, we define univ, the collection of all universes, and the Nuprl type system as follows:

```
Definition univ (T T' : CTerm) (eq : per) := exists i , univi i T T' eq.
Definition nuprl := close univ.
```

Finally, we define $t_1 \equiv t_2 \in T$ and $T \equiv T'$, respectively, as follows:

$$\texttt{exists}\ eq\ ,\ \texttt{nuprl}\ T\ T\ eq \wedge eq\ t_1\ t_2$$
$$\texttt{exists}\ eq\ ,\ \texttt{nuprl}\ T\ T'\ eq$$

CTT includes dependent product and sum types, equality types, a hierarchy of universes, W types, union and (dependent) intersection types (Kopylov 2004), quotient types (Constable 1983), set types, image types (Nogin and Kopylov 2006), PER types (Anand *et al.* 2014b), approximation and computational equivalence types (Rahli *et al.* 2013) and partial types (Constable and Smith 1993; Crary 1998; Smith 1989). Figure 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, more precisely inhabited by $\star$, if and only if the metatheoretical statement $t_1 \preccurlyeq t_2$ is true; and $t_1 \preceq t_2$ and $t_3 \preceq t_4$ are equal types if and only if $(t_1 \preccurlyeq t_2 \iff t_3 \preccurlyeq t_4)$. Similarly, the type $t_1 \simeq t_2$ is true if the metatheoretical statement $t_1 \sim t_2$ is true in the metatheory, and $t_1 \simeq t_2$ and $t_3 \simeq t_4$ are equal types if $(t_1 \sim t_2 \iff t_3 \sim t_4)$ [for more details, see Sections 2.4, 2.5 and Rahli *et al.* (2013)]. 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 (Anand and Rahli 2014a,b). 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. [For more details regarding Nuprl's consistency, see Section 2.4, as well as Anand and Rahli (2014a,b).] 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 both Nuprl's computation and type systems in order to prove $\texttt{SCP}_{\downarrow}$. It includes adding some *nominal features* such as a *fresh* operator, *named exceptions*, *exception handlers* and an *exception type*. Using our Coq formalization, we provide in Section 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 Section 4, we prove $\texttt{SCP}_{\downarrow}$ using the nominal features mentioned above.

2.4. *Sequents and rules*

In Nuprl, one reasons about types using a sequent calculus, which is a collection of derivation rules that captures many properties of Nuprl's computation and type systems.

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

For example, for each type we have introduction and elimination rules. This calculus can be extended as required by adding more types and/or derivation rules. This section presents the syntax and semantics of Nuprl's sequents and rules. See Anand and Rahli (2014a,b) for more details. We have already verified a large number of Nuprl's inference rules. Our formalization of Nuprl in Coq provides a safe way to add new inference rules to Nuprl by mechanizing their semantics, and allowing one to formally prove their validity, which is a difficult task without the help of a proof assistant. Howe (1996) writes: 'Because of this complexity, many of the Nuprl rules have not been completely verified, and there is a strong barrier to extending and modifying the theory.'

Syntax of sequents and rules. Sequents are of the form $h_1, \ldots, h_n \vdash T \ \lfloor \text{ext } t \rfloor$. The term t is called the *extract* or *evidence* of the type T, An hypothesis h is either of the form $x : A$ (non-hidden) or of the form $[x : A]$ (hidden) – hidden hypotheses are discussed below – where x is referred to as the name of the hypothesis and A its type. A name cannot occur twice in a list of hypotheses. Such a sequent states, among other things, that T is a type and t is a member of T. We write $H \vdash T$ for $H \vdash T \ \lfloor \text{ext } \star \rfloor$ or whenever the extract is irrelevant. A rule is a pair of a sequent and a list of sequents, which we write either as (where *name* is the name of the rule):

$$(S_1 \wedge \cdots \wedge S_n) \Rightarrow S$$

$$\text{or} \quad \frac{S_1 \quad \cdots \quad S_n}{S} \ \textit{name}$$

$$\text{or} \quad \begin{array}{l} S \\ \quad \text{BY } \textit{name} \\ \quad S_1 \\ \quad \vdots \\ \quad S_n \end{array}$$

Hidden hypotheses. To understand the necessity of hidden hypotheses, let us consider the following intersection introduction rule:

$$\begin{array}{l} H \vdash \cap a{:}A.B[a] \ \lfloor \text{ext } e \rfloor \\ \quad \text{BY } \texttt{[isectMemberFormation]} \\ \quad H, [x : A] \vdash B[x] \ \lfloor \text{ext } e \rfloor \\ \quad H \vdash A = A \in \mathbb{U}_i \end{array}$$

This rule says that to prove that $\cap a{:}A.B[a]$ is true with extract e, one has to prove that $B[x]$ is true with extract e, assuming that x is of type A. The meaning of intersection types requires that the extract e be the same for all values of A, and is therefore called the *uniform evidence* of $\cap a{:}A.B[a]$. The fact that x is hidden means that it cannot occur free in e (but can occur free in B). The same mechanism is required to state the rules for, e.g., subset types or quotient types.

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

Hidden hypotheses can be unhidden when proving conclusions that do not have any computational content such as equalities. Therefore, the following rule is valid according to Allen's PER semantics explained below:

$$H, [x : T], J \vdash a = b \in C$$
BY `[Unhide]`
$$H, x : T, J \vdash a = b \in C$$

Digression on intersection types. As a side note, non-dependent intersections of type families and dependent intersection types were studied by Kopylov (2004) and added to the MetaPRL Hickey (2001) (a cousin of Nuprl mostly developed by Hickey) and Nuprl systems [intersections of type families were added to Nuprl around 2000 (Constable and Hickey 2000)]. Intersection types have recently been used by Constable and Bickford to prove a completeness result of intuitionistic first-order logic (Constable and Bickford 2014). Around the same time, Miquel (2000, 2001a,b) was also studying the addition of intersections of type families to a Curry-style version of the Calculus of Constructions. In Nuprl, we mostly use intersection types to avoid 'noise' in our extracts, e.g., to avoid having type parameters in extracts that have no role in the computational content of the extracts, and are only used for typing purposes. We believe that intersection types are central in our theory and we proposed in Anand *et al.* (2014b) an alternative definition of Nuprl that relies on PER types and intersection types as the main logical universal quantifier.

Semantics of sequents and rules. Several definitions for the truth of sequents occur in the Nuprl literature (Constable *et al.* 1986; Crary 1998; Kopylov 2004). Among these, Kopylov's definition was the simplest (Kopylov 2004) in the sense that it is more compactly defined than the others and also easier to work with. Our definition in Anand and Rahli (2014a,b) is a simplified version of Kopylov's definition, and we have proved that all these definitions are equivalent (Anand and Rahli 2014b, Section 5.1). The semantics we present uses a notion of *pointwise functionality* (Crary 1998, Section 4.2.1), which says that each type occurring in a true sequent must respect the equalities of the types on which it depends. This is captured by formula 1 below for the hypotheses of a sequent, and by formula 2 for its conclusion. For the purpose of this discussion, let us ignore the possibility that some hypotheses can be hidden.

Let H be a list of hypotheses of the form $x_1 : T_1, \ldots, x_n : T_n$, let s_1 be a substitution of the form $(x_1 \mapsto t_1, \ldots, x_n \mapsto t_n)$ and let s_2 be a substitution of the form $(x_1 \mapsto u_1, \ldots, x_n \mapsto u_n)$.

Similarity. The similarity relation lifts the notion of equality in a type (i.e., the relation $_\equiv_\in_$) to lists of hypotheses. We say that s_1 and s_2 are similar in H, and write $s_1 \equiv s_2 \in H$, if for all $i \in \{1, \ldots, n\}$, $t_i \equiv u_i \in T_i[x_1 \backslash t_1; \cdots; x_{i-1} \backslash t_{i-1}]$. Let $s \in H$ be $s \equiv s \in H$.

Equal hypotheses. The following notion of equality lifts the notion of equality between types (i.e., the relation $_=_$) to lists of hypotheses. We say that the hypotheses H are equal w.r.t. s_1 and s_2, and write $s_1(H) \equiv s_2(H)$, if for all $i \in \{1, \ldots, n\}$, $T_i[x_1 \backslash t_1; \cdots; x_{i-1} \backslash t_{i-1}] \equiv T_i[x_1 \backslash u_1; \cdots; x_{i-1} \backslash u_{i-1}]$.

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

Hypotheses functionality. We say that the hypotheses H are pointwise functional w.r.t. the substitution s, and write $H@s$ if

$$\forall s'.\ s \equiv s' \in H \Rightarrow s(H) \equiv s'(H). \tag{1}$$

Truth of sequents. We say that a sequent of the form $H \vdash T\ \lfloor \mathbf{ext}\ t \rfloor$ is true if

$$\begin{aligned} \forall s_1, s_2.\ \ & s_1 \equiv s_2 \in H \wedge H@s_1 \\ \Rightarrow\ \ & T[s_1] \equiv T[s_2] \wedge t[s_1] \equiv t[s_2] \in T[s_1]. \end{aligned} \tag{2}$$

In addition, the free variables of t have to be non-hidden in H.

Validity of rules. A rule of the form $(S_1 \wedge \cdots \wedge S_n) \Rightarrow S$ is valid if assuming that the sequents $S_1, \ldots, S_n$ are true then the sequent S is also true.

Consistency. Using our formalization of Nuprl in Coq, we have already verified a large number of Nuprl's inference rules, including the usual introduction and elimination rules to reason about the core type system presented in Anand and Rahli (2014a).

A Nuprl proof is a tree of sequents where each node corresponds to the application of a rule. Using the definition of the validity of a rule, and by induction on the size of the tree, this means that the sequent at the root of the tree is true w.r.t. Nuprl's PER semantics if the tree has no leaves. Hence, a proof of False, for any meaningful definition of False, i.e., a type with an empty PER such as $0 = 1 \in \mathbb{Z}$, $0 \preceq 1$, or $\star \preceq \bot$ (this one says that $\bot$ converges), would mean that its PER is in fact non-empty, which leads to a contradiction.

Extensionality. Nuprl is extensional in the sense that propositional and definitional equality are identified (Hofmann 1995), and functional extensionality holds from the PER semantics of $\boldsymbol{\Pi}$ types. It is also intensional in the sense that not all types with equivalent PERs are equal. This is also illustrated in Nogin's PhD thesis (Nogin 2002, p. 39). For example, equality types have trivial content. Both $0 =_{\mathbb{Z}} 0$ and $1 =_{\mathbb{Z}} 1$ are inhabited by $\star$. These two types have equivalent PERs but they are not equal as types. As it was recently reminded to us by our colleague Evan Moran, this intensionality is necessary to validate sequents such as $h : x \in \mathbb{Z} \vdash \mathbb{Z}\ \lfloor \mathbf{ext}\ x \rfloor$, where $h : x \in \mathbb{Z}$ is an hypothesis named h of type $x \in \mathbb{Z}$, and x is the *extract* of the sequent (i.e., the evidence that shows that $\mathbb{Z}$ is true, i.e., an inhabitant of that type). According to Nuprl's PER semantics [see Anand and Rahli (2014a,b) for the definition of equality types] and under an extensional definition of equality types, because $0 =_{\mathbb{Z}} 0$ and $1 =_{\mathbb{Z}} 1$ would be equal types, then we would have to prove that $0 \equiv 1 \in \mathbb{Z}$.

2.5. *Computational types*

This section discusses the types `Top`, `Base`, $\preceq$ and $\simeq$. Some of the results mentioned in this section were presented in Rahli *et al.* (2013).

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

Note that $t_1 \preccurlyeq t_2$ is Howe's approximation (or simulation) metarelation, while $t_1 \preceq t_2$ is a CTT type. This type is interpreted as follows:

$$\begin{array}{l}\texttt{per_approx}(\tau, T, T', \phi) = \\ \exists a, b, c, d. \\ \quad\quad T \Downarrow a \preceq b \\ \quad \wedge\ T' \Downarrow c \preceq d \\ \quad \wedge\ (a \preccurlyeq b \iff c \preccurlyeq d) \\ \quad \wedge\ (\forall t, t'.\ t\ \phi\ t' \iff a \preccurlyeq b \wedge t \Downarrow \star \wedge t' \Downarrow \star).\end{array}$$

The relation per_approx expresses when two types T and T' are equal approximation types and defines the equality ϕ of T. Note that the type system τ is not used in this definition because approximation types do not have types as subterms. However, it is used in, e.g., the interpretation of dependent products types (see Section 2.3) to state, e.g., that for two dependent product types to be equal, their domains have to be equal w.r.t. the type system. Approximation types are extensional in the sense that all true approximation types are equal to each other and all false approximation types are equal to each other (third clause in per_approx's definition).

Similarly, $t_1 \sim t_2$ is Howe's computational equivalence (or bisimulation) metarelation, while $t_1 \simeq t_2$ is a type. This type is interpreted as follows:

$$\begin{array}{l}\texttt{per_cequiv}(\tau, T, T', \phi) = \\ \exists a, b, c, d. \\ \quad\quad T \Downarrow a \simeq b \\ \quad \wedge\ T' \Downarrow c \simeq d \\ \quad \wedge\ (a \sim b \iff c \sim d) \\ \quad \wedge\ (\forall t, t'.\ t\ \phi\ t' \iff a \sim b \wedge t \Downarrow \star \wedge t' \Downarrow \star).\end{array}$$

Base is the type of all closed terms with $\sim$ as its equality. It is a primitive type. Top is the type of all closed terms with True as its equality. It is currently defined as follows: Top $= \cap x$:Void.Void. A more intuitive (equivalent) definition might be Base//True.

Here are a few rules that we can prove about $\simeq$ and $\preceq$:

$$\begin{array}{ll}H \vdash a \preceq a & H \vdash a \simeq a \\ \quad \text{BY [approx-refl]} & \quad \text{BY [cequiv-refl]} \\ \\ H \vdash a \simeq b & H \vdash a\ b \simeq c\ d \\ \quad \text{BY [cequiv-base]} & \quad \text{BY [cequiv-app-D]} \\ \quad H \vdash a = b \in \texttt{Base} & \quad H \vdash a \simeq c \\ & \quad H \vdash b \simeq d \\ \\ H \vdash \lambda x.a \simeq \lambda x.b & H \vdash a \simeq b \\ \quad \text{BY [cequiv-lam-D]} & \quad \text{BY [cequiv-approx]} \\ \quad H, x : \texttt{Base} \vdash a \simeq b & \quad H \vdash a \preceq b \\ & \quad H \vdash b \preceq a\end{array}$$

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

It turns out we can prove that `[approx-refl]` is a valid rule because $\preceq$ is an extensional type. If it was not, we would not be able to prove the 'functionality' of $a \preceq a$ in any context. We will freely use these rules below without mentioning them.

It turns out that when proving a proposition of the form $a \simeq b$ or of the form $a \preceq b$ in the context of an hypothesis $x : \mathtt{Top}$, then we can always change this instance of `Top` into `Base`, which is extremely useful because derivation rules that mention either of our two computation types $\simeq$ and $\preceq$ are often stated using `Base`. This 'trick' was discovered by our colleague David Guaspari. Let us assume that we are proving a sequent of the form

$$H, x : \mathtt{Top}, J \vdash a \simeq b.$$

Because $a \simeq (\lambda x.a)\ x$ and $b \simeq (\lambda x.b)\ x$ are true in any context (using computation and `[cequiv-refl]`), it is enough to prove

$$H, x : \mathtt{Top}, J \vdash (\lambda x.a)\ x \simeq (\lambda x.b)\ x.$$

Using `[cequiv-app-D]`, `[cequiv-refl]` and `[cequiv-lam-D]`, it is enough to prove

$$H, x : \mathtt{Top}, J, x' : \mathtt{Base} \vdash a[x \backslash x'] \simeq b[x \backslash x'],$$

which can easily be turned into

$$H, x : \mathtt{Base}, J \vdash a \simeq b.$$

2.6. *Squashing inference rules*

Let us now present a few derivable inference rules to reason about our two main squashing operators $\downarrow$ and $\lfloor$. Because most of them are trivial, we have proved them to be valid w.r.t. Allen's PER semantics (see Section 2.4) directly in Coq: https://github.com/vrahli/NuprlInCoq/blob/master/rules/rules_squash.v.

First, let us present the rules about our $\downarrow$ operator as they are simpler:

$$\begin{array}{ll}
H \vdash x = y \in \downarrow T & \\
\quad \text{BY } \mathtt{[SquashEqual]} & H \vdash \downarrow T \\
\quad H \vdash T & \quad \text{BY } \mathtt{[Squash]} \\
\quad H \vdash x \simeq \star & \quad H \vdash T \\
\quad H \vdash y \simeq \star & \\
 & \\
H, x : \downarrow T, J \vdash C & H \vdash x \simeq \star \\
\quad \text{BY } \mathtt{[SquashElim]} & \quad \text{BY } \mathtt{[SquashMember]} \\
\quad H, x : \downarrow T, J, [y : T] \vdash C & \quad H \vdash x \in \downarrow T
\end{array}$$

`[SquashEqual]` says that to prove that x and y are equal members in $\downarrow T$, it is enough to prove that T is true and that x and y both compute to $\star$. `[Squash]` says that to prove $\downarrow T$ it is enough to prove T. `[SquashMember]` says that $\star$ is the only inhabitant of $\downarrow T$. The only non-trivial rule is `[SquashElim]`, which introduces a hidden hypothesis. (Note that because y does not occur in $H, x : \downarrow T, J$ then it cannot occur in C either.) Therefore, we get to assume that T is true, but we do not get to know what term it is inhabited by. As mentioned above in Section 2.4, hidden hypotheses can be unhidden when proving

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

conclusions that do not have any computational content such as equalities. Using these rules, we can derive the following elimination rule:

$$H, x : \downarrow T, J \vdash a = b \in C$$
BY [SquashElim]′
$$H, x : T, J[x\backslash\star] \vdash a[x\backslash\star] = b[x\backslash\star] \in C[x\backslash\star].$$

We now present the rules about our $\downarrow$ operator:

$$H \vdash x = y \in \downarrow T$$
BY [TruncationEqual]
$$H \vdash x \in T$$
$$H \vdash y \in T$$

$$H \vdash \downarrow T$$
BY [Truncation]
$$H \vdash T$$

$$H, x : \downarrow T, J \vdash a = b \in C$$
BY [TruncationElim]
$$H, x : T, J, y : T \vdash a = b[x\backslash y] \in C$$
$$H, x : \downarrow T, J \vdash C \in \texttt{Type}$$
$$H \vdash T \in \texttt{Type}$$

Rule [TruncationEqual] is similar to rule [SquashEqual], and rule [Truncation] is similar to rule [Squash]. Again, the only non-trivial rule is [TruncationElim]. The third subgoal

$$H \vdash T \in \texttt{Type}$$

is necessary because $\downarrow$-types are extensional, i.e., $\downarrow T$ is equal to $\downarrow U$ iff T and U are types that have equivalent PERs. The second subgoal

$$H, x : \downarrow T, J \vdash C \in \texttt{Type}$$

is necessary to prove that C is functional w.r.t. the list of hypotheses $H, x : \downarrow T, J$. With the first subgoal, we only get that C is functional w.r.t. the list of hypotheses $H, x : T, J, y : T$. Intuitively, the first subgoal tells us that $C[x\backslash u]$ and $C[x\backslash v]$ are equal whenever u and v are equal in T, when what we want to prove is that $C[x\backslash u]$ and $C[x\backslash v]$ are equal whenever u and v are both (not necessarily equal) members of T, which is what the second subgoal tells us. Finally, the first subgoal

$$H, x : T, J, y : T \vdash a = b[x\backslash y] \in C$$

says that we get to unsquash x's type because we are proving an equality and because equality types do not have computational content. In addition, we also have to prove that our equality is functional w.r.t. the hypothesis $x : \downarrow T$, i.e., given another element y in T not necessarily equal to x, we have to prove that a is equal to $b[x\backslash y]$ in C.

We now have presented enough background on Nuprl to start proving new metatheoretical and theoretical results. We now turn to the validity of Brouwer's weak continuity principle for numbers.

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 Nuprl in Coq, the interested

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

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

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. For example, f could be $\lambda x.x$ which is a member of $\mathbb{Z}^{\mathbb{Z}}$, and just knowing that $F(f)$ computes to the integer i, it could still happen that f gets applied to, say, $\star$. To remedy this issue, we use the function $\texttt{force}(f) = \lambda x.\texttt{let } x := x + 0 \texttt{ in } f(x)$. 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. 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 $F(\texttt{force}(f)) \mapsto^* i$. Therefore, our use of `force` ensures that f's arguments are integers.

Step 3. Let $\texttt{bound}(t, b) = \texttt{let } x := t \texttt{ in if } |x| < b \texttt{ then } x \texttt{ else } \bot$. 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 metatheory by computing the largest number occurring in the computation C_1, i.e., if $t_0 = F(\texttt{force}(f)) \mapsto t_1 \mapsto \cdots \mapsto i = t_n$, 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 metatheoretical 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 now instantiate our conclusion using b. It remains to prove that $\boldsymbol{\Pi} g : \mathbb{Z}^{\mathbb{Z}}.f =_{\mathbb{Z}^{\mathbb{Z}_b}} g \to F(f) =_{\mathbb{Z}} F(g)$, where $\mathbb{Z}_b$ is the type of integers strictly less than b. 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., $F(f) =_{\mathbb{Z}} F(g)$.

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

4. Strong continuity principle

We now prove $\mathtt{SCP}_{\downarrow}$ (Kleene and Vesley 1965, pp. 69–73) (Section 5 shows that $\mathtt{SCP}_{\downarrow}$ and $\mathtt{WCP}_{\downarrow}$ are equivalent). We need to come up with a Nuprl term of type $\boldsymbol{\Pi} n{:}\mathbb{N}.\mathcal{B}_n \to \mathbb{N}_{\mathtt{U}}$ that checks whether we have reached the modulus of continuity of a function. For that, we now use named exceptions as a probing mechanism to compute the modulus of continuity of a function. Instead of $\mathtt{SCP}_{\downarrow}$, we prove the following equivalent but slightly simpler statement (Kleene and Vesley 1965, pp. 71–72) (where the T in SCPT is for Test—see below):

$$\begin{aligned}\mathtt{SCPF}(F) \;=\; &\downarrow\, \boldsymbol{\Sigma} M{:}(\boldsymbol{\Pi} n{:}\mathbb{N}.\mathcal{B}_n \to \mathbb{N}_?).\\ &\quad \boldsymbol{\Pi} f{:}\mathcal{B}.\, \downarrow \boldsymbol{\Sigma} n{:}\mathbb{N}.M\ n\ f =_{\mathbb{N}} F(f)\\ &\qquad \wedge \boldsymbol{\Pi} n{:}\mathbb{N}.\mathtt{isint}(M\ n\ f) \to M\ n\ f =_{\mathbb{N}} F(f)\\ \mathtt{SCPT} \;=\; &\boldsymbol{\Pi} F{:}\mathcal{B} \to \mathbb{N}.\ \mathtt{SCPF}(F).\end{aligned}$$

Using our Coq formalization and making use of computations on terms that are only possible in the metatheory, 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 $\mathtt{SCP}_{\downarrow}$ are equivalent. We prove SCPT rather than $\mathtt{SCP}_{\downarrow}$ mainly because its realizer is simpler. Intuitively, the M part of SCPT's realizer is a simple test function of the form (we use here some ML-like pseudo-code and Section 4.4 presents the corresponding Nuprl expression):

```
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
```

while the one for $\mathtt{SCP}_{\downarrow}$ is a recursive search function of the form (we use here some ML-like pseudo-code and Section 4.9 presents the corresponding Nuprl expression):

```
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
```

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).

4.1. *Extension of Nuprl's computation system*

4.1.1. *Syntax.* We extend Nuprl with *names* [or unguessable atoms (Bickford 2008)], *named exceptions*, *exception handlers* and a *fresh* operator as follows:

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

$$
\begin{array}{lll}
v & ::= \cdots \mid \boldsymbol{a} & \text{(name value)} \\
ty & ::= \ldots & \\
 & \mid \mathtt{Name} & \text{(name type)} \\
 & \mid \mathtt{Exc}(t_1, t_2) & \text{(exception type)} \\
e & ::= \mathtt{exc}(t_1, t_2) & \text{(exception)} \\
t & ::= \ldots & \\
 & \mid e & \text{(exception)} \\
 & \mid \nu x.\boxed{t} & \text{(fresh)} \\
 & \mid \mathtt{try}_n\ \boxed{t}\ \mathtt{with}\ x.c & \text{(try/catch).}
\end{array}
$$

`Name` is a type of names and $\boldsymbol{a}$ stands for a name (names are constants). Names were introduced in Nuprl under the name of 'unguessable atoms' to reason about logical foundations for security (Bickford 2008). To account for names, Allen generalized his PER semantics (Allen 1987b) to a so-called *supervaluation* semantics that quantifies over all possible implementations of the `Name` type (Allen 2006). Names come with two metatheoretical 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 (Pitts 2013, Section 9.6; Pitts and Stark 1993) or Odersky's $\lambda\nu$-calculus (Odersky 1994; Pitts 2013, Section 9.4), we add two corresponding operators to Nuprl. Because we already have a test for equality operator for integers, we simply modify the operational semantics of ($\mathtt{if}\ t_1{=}t_2\ \mathtt{then}\ t_3\ \mathtt{else}\ t_4$) as follows:

$$
\begin{array}{lll}
\mathtt{if}\ \boldsymbol{a}{=}\boldsymbol{a}\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2 & \mapsto & t_1 \\
\mathtt{if}\ \boldsymbol{a}{=}\boldsymbol{b}\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2 & \mapsto & t_2, \text{ if } \boldsymbol{a} \neq \boldsymbol{b}.
\end{array}
$$

Our exceptions and handlers are similar to Lebresne's (Lebresne 2009). 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 $\mathtt{Exc}(t_1, t_2)$ 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 $\mathtt{exc}(\boldsymbol{a}, 0)$ and $\mathtt{exc}(\boldsymbol{b}, 0)$ have type $\mathtt{Exc}(\mathtt{Name}, \mathbb{Z})$ (among others); and $\mathtt{exc}(1, 0)$ has type $\mathtt{Exc}(\mathbb{Z}, \mathbb{Z})$. We also add exception handlers of the form $\mathtt{try}_n\ t\ \mathtt{with}\ x.c$, 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:

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

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 $\star$.

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

4.1.2. *Related exception-based systems* 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' (Gordon *et al.* 1979, p. 11). However, exceptions are often not accounted for in types.

The type $T ?_n$ is similar to Lebresne's type $A \uplus \{\epsilon\}$, where A is a type and ϵ is an exception (Lebresne 2008, 2009). Lebresne's Fx system (Lebresne 2009) provides type constructors to express two different levels of *corruption*. In addition to the ones mentioned above, 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, $\texttt{inl}(\texttt{exc}_a)$ could be of type $(\mathbb{N}+\texttt{Unit})^{\{a\}}$. Sacchini (2014) shows that corruption in the presence of dependent types has interesting consequences. We leave adding corruption types to Nuprl for future work. Lebresne (2009) 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 (2005) introduced EX_2 as an extension of Krivine's FA_2 system (Krivine 1993) 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.

Exceptions bear some resemblance with control operators such as call/cc [see for example Thielecke (2000, 2001) for a comparisons between exceptions and continuations, that show among other things that there are contextual equivalences that hold in the presence of exceptions but not in the presence of continuations, and vice versa]. Griffin (1990) and Murthy (1991) extended the formula-as-type/proof-as-program correspondence to classical systems by relating classical proofs to typed programs that make use of control operators such as the $\mathcal{C}$ operator (Felleisen *et al.* 1986), which is similar to call/cc. In these systems, control operators are used to provide algorithmic content for the classical double-negation elimination rule. Other prominent examples of such systems are Parigot's $\lambda\mu$-calculus (Parigot 1992) and de Groote's $\lambda^{\rightarrow}_{\mathrm{exn}}$-calculus (Groote 1995), which also uses control operators (ML-like exceptions in the case of $\lambda^{\rightarrow}_{\mathrm{exn}}$) to give algorithmic content to classical proofs. Nakano (1994) studied a CTT with catch and throw operations, such that one can extract programs that make use of these operations. Kameyama designed a simple calculus for dynamic catch/throw exception mechanism in a type-theoretic setting (Kameyama 2001). As opposed to Nakano's static exception mechanism, in his system the catch corresponding to a throw is determined dynamically. Function types are annotated by sets of types to keep track of the types of possible raised exceptions as in Lebresne (2009). Kameyama and Yonezawa (2008) studied Felleisen's dynamic control operators control and prompt (Felleisen 1988). They introduced a typed calculus for these operators and showed that these dynamic typed control operators can simulate the typed static ones [shift and reset (Danvy and Filinski 1990)], while the converse direction is not possible.

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

4.1.3. *Operational semantics of* $\boldsymbol{\nu}$. Let us now present the operational semantics of fresh and exception handlers. 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 (Odersky 1994) approach where $\boldsymbol{\nu}x.x$ is stuck and from Pitts' approach (Pitts 2010) 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 $x \neq y$),

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

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. With such a reduction rule, we would have $\boldsymbol{\nu}x.\texttt{inl}(x) \mapsto \texttt{inl}(\boldsymbol{a})$ and $(\texttt{let } y := \boldsymbol{a} \texttt{ in } x) \sim x$, whereas $\boldsymbol{\nu}x.\texttt{inl}(\texttt{let } y := \boldsymbol{a} \texttt{ in } x) \not\mapsto^* \texttt{inl}(\boldsymbol{a})$.

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 (Losch and Pitts 2011; Odersky 1994; Pitts 2013) [as opposed to using, e.g., stateful dynamic allocation (Losch and Pitts 2011) or the notion of *prevalues* (Jeffrey and Rathke 1999), 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 $x \neq y$. We cannot simply reduce $\boldsymbol{\nu}x.\langle 1, x\rangle$ to $\langle 1, x\rangle$ because x would become free. 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.

4.1.4. *Operational semantics of* `try`. Handlers of the form $(\texttt{try}_n\ e\ \texttt{with}\ x.c)$ catch exceptions of the form $\texttt{exc}(n, d)$, i.e., names have to match. For example,

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

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

When its principal argument is non-canonical or a variable, a handler computes exactly like the other non-canonical operators (except $\boldsymbol{\nu}$), i.e., $(\mathtt{try}_n\ e_1\ \mathtt{with}\ x.c) \mapsto (\mathtt{try}_n\ e_2\ \mathtt{with}\ x.c)$ if $e_1 \mapsto e_2$. 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:

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

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 computes to 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 $\mathtt{exc}_{\boldsymbol{a}}$, then $\mathtt{try}_\bot\ \mathtt{exc}_{\boldsymbol{a}}$ 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 as follows:

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

4.1.5. *Regarding the operational Semantics of* $\boldsymbol{\nu}$. In Section 4.1.3, we saw that we cannot reduce $\boldsymbol{\nu}x.\langle 1, x\rangle$ to $\langle 1, x\rangle$ because x would become free. Could we replace x with some default value such as, say, $\star$ or $\bot$? The problem with that is that any of these choices would break the properties of Howe's computational equivalence (Howe 1989). Say we pick $\star$ to replace v. Then, we can prove

$$\begin{array}{l}\mathtt{if}\ \boldsymbol{a}{=}\boldsymbol{b}\ \mathtt{then}\ \mathtt{inl}(0)\ \mathtt{else}\ \mathtt{inl}(1)\\ \preccurlyeq \mathtt{inl}(\mathtt{if}\ \boldsymbol{a}{=}\boldsymbol{b}\ \mathtt{then}\ 0\ \mathtt{else}\ 1),\end{array}$$

where $\boldsymbol{a}$ and $\boldsymbol{b}$ are two different names. Therefore, we expect to be able to prove using congruence that

$$\begin{array}{l}\boldsymbol{\nu}x.(\mathtt{if}\ x{=}\boldsymbol{b}\ \mathtt{then}\ \mathtt{inl}(0)\ \mathtt{else}\ \mathtt{inl}(1))\\ \preccurlyeq \boldsymbol{\nu}x.(\mathtt{inl}(\mathtt{if}\ x{=}\boldsymbol{b}\ \mathtt{then}\ 0\ \mathtt{else}\ 1)).\end{array}$$

However, because

$$\begin{array}{c}\boldsymbol{\nu}x.(\mathtt{if}\ x{=}\boldsymbol{b}\ \mathtt{then}\ \mathtt{inl}(0)\ \mathtt{else}\ \mathtt{inl}(1)) \mapsto^* \mathtt{inl}(0)\\ \boldsymbol{\nu}x.(\mathtt{inl}(\mathtt{if}\ x{=}\boldsymbol{b}\ \mathtt{then}\ 0\ \mathtt{else}\ 1)) \mapsto^* \mathtt{inl}(\mathtt{if}\ \star{=}\boldsymbol{b}\ \mathtt{then}\ 0\ \mathtt{else}\ 1)\\ \mathtt{inl}(0) \not\preccurlyeq \mathtt{inl}(\mathtt{if}\ \star{=}\boldsymbol{b}\ \mathtt{then}\ 0\ \mathtt{else}\ 1),\end{array}$$

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

then we would also get that

$$\boldsymbol{\nu} x.(\texttt{if } x{=}\boldsymbol{b} \texttt{ then inl}(0) \texttt{ else inl}(1))$$
$$\not\preccurlyeq \boldsymbol{\nu} x.(\texttt{inl}(\texttt{if } x{=}\boldsymbol{b} \texttt{ then } 0 \texttt{ else } 1)).$$

Similar examples can be constructed for different default values. For example, for $\bot$, we can use $\texttt{let } z := x \texttt{ in inl}(0)$ and $\texttt{inl}(\texttt{let } z := x \texttt{ in } 0)$.

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 (Howe 1989). Unfortunately, this is not easy to prove directly. Howe's 'trick' was to define another inductive relation $\preccurlyeq^*$, which is a congruence and contains $\preccurlyeq$ by definition.

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 Howe (1989) 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. An operator θ is extensional if for all $k \in \mathbb{N}$, and closed v (a value), $\overline{b}$ and $\overline{b'}$, if $\theta(\overline{b}) \mapsto^{k+1} v$ and $\overline{b} \preccurlyeq^* \overline{b'}$ then $v \preccurlyeq^* \theta(\overline{b'})$. We also get to use the following induction hypothesis IH: if for all closed t_1, t_2, t_3, if $t_1 \mapsto^k t_2$ and $t_1 \preccurlyeq^* t_3$, then $t_2 \preccurlyeq^* t_3$.

We extended all these definitions to deal with the fact that canonical forms can either be values or exception, and then, as mentioned below, using our new definition of $\preccurlyeq^*$ we were able to prove that $\boldsymbol{\nu}$ is extensional.

First, we modified Howe's definition of $\preccurlyeq^*$ to account for the fact that the binders of $\boldsymbol{\nu}$ expressions are only meant to be names: as shown in Section 4.1.3, when computing an expression of the form $\boldsymbol{\nu} x.t$, we only ever replace x by a name. Before we introduced $\boldsymbol{\nu}$, any bound variable of the computation system could potentially be substituted by any term. For example, the bound variable of a λ-abstraction can be substituted by any term when β-reducing an expression, i.e., $(\lambda x.t)\ u$ reduces to $t[x\backslash u]$ regardless of what u is (of what type it can have). One solution to distinguish between terms that stand for names and terms that do not is it to use types. Next paragraph explains why it was useful to be able to make this distinction. Intuitively, to prove that $\boldsymbol{\nu} x_1.t_1 \preccurlyeq^* \boldsymbol{\nu} x_2.t_2$, it should be enough to prove that $t_1[x_1\backslash \boldsymbol{a}] \preccurlyeq^* t_2[x_2\backslash \boldsymbol{a}]$ for some fresh enough $\boldsymbol{a}$. However, the former definition of $\preccurlyeq^*$ would require one to prove that $t_1[x_1\backslash u] \preccurlyeq^* t_2[x_2\backslash u]$ for any closed term u. Therefore, rather than turning Nuprl into a typed language, we added 'simple' type information to the definition of $\preccurlyeq^*$. Let `BindersTypes` be a function that, for a given operator, returns the types of the bound variables of its bound terms. The type of a bound variable can either be `NAME` or `ANY`. Let $\texttt{BindersTypes}(\boldsymbol{\nu}) = [[\texttt{NAME}]]$ because $\boldsymbol{\nu}$ has one subterm (the outer brackets) that has one bound variable (the inner brackets) where the variable stands for a name. The type of all the other bound variables is `ANY`. For example, $\texttt{BindersTypes}(\lambda) = [[\texttt{ANY}]]$ because a λ-abstraction has one subterm which has

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

one bound variable; given the spread operator (to destruct pairs), `BindersTypes` returns [[], [`ANY`, `ANY`]] because the first subterm of a term of the form (`let` $x, y = t_1$ `in` t_2) has no bound variables, while the second one has two; and given the decide operator (to destruct injections), `BindersTypes` returns [[], [`ANY`], [`ANY`]] because the first subterm of a term of the form (`case` t_1 `of inl`$(x) \Rightarrow t_2$ | `inr`$(y) \Rightarrow t_3$) has no bound variables, while the other two have one bound variable each. When extending the definition of $\preccurlyeq^*$ from terms to bound terms, we use `BindersTypes` to restrict what terms can be substituted for free variables. More precisely, with the new definition of $\preccurlyeq^*$, if t is a term of the form $\tau(b_1, \ldots, b_n)$, then $t \preccurlyeq^* u$ is true if there exists $b'_1, \ldots, b'_n$ such that $(b_1, \ldots, b_n) \preccurlyeq^* (b'_1, \ldots, b'_n)$ and $\tau(b'_1, \ldots, b'_n) \preccurlyeq u$, where $b_i \preccurlyeq^* b'_i$ is now defined as follows: b_i is a bound term of the form $l_1.t_1$, b'_i is a bound term of the form $l_2.t_2$, and $t_1[l_1 \backslash l] \preccurlyeq^* t_2[l_2 \backslash l]$, where l is now a list of terms such that the n^{th} element of the list is either (1) a variable as before if the nth element of the ith element of `BindersTypes`(τ) is `ANY` or (2) a name which is fresh w.r.t. both t and u otherwise (also, all the variables and names in l have to be distinct to each other). This modification of $\preccurlyeq^*$'s definition was inspired by Gordon's (Gordon 1995) and Jeffrey and Rathke's (Jeffrey and Rathke 1999) adaptations of Howe's method to typed λ-calculi. As mentioned above, 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.

Let us motivate the changes we made to $\preccurlyeq^*$ by sketching the proof that $\boldsymbol{\nu}$ is extensional: we have to prove that if $\boldsymbol{\nu}x.t \mapsto^{k+1} u$ (where u is canonical, i.e., either a value or an exception), and $x.t \preccurlyeq^* x'.t'$, then $u \preccurlyeq^* \boldsymbol{\nu}x'.t'$ (we also get to use IH mentioned above). Because $\boldsymbol{\nu}x.t$ reduces to a canonical form, this means that $t[x \backslash \boldsymbol{a}]$ reduces to either a value or an exception, for some 'fresh enough' $\boldsymbol{a}$. If it reduces to a value z, then $u = (\Downarrow_x (z[\boldsymbol{a} \backslash x]))$. Using IH, we get that $z \preccurlyeq^* t'[x' \backslash \boldsymbol{a}]$. Using Howe's Lemma 2, we get that $t'[x' \backslash \boldsymbol{a}]$ computes to a value w with same operator as z. We get that $z \preccurlyeq^* w$, where z is α-equal to some $z'[x \backslash \boldsymbol{a}]$ where $\boldsymbol{a}$ does not occur in z' and w is α-equal to some $w'[x \backslash \boldsymbol{a}]$ where $\boldsymbol{a}$ does not occur in w'. Therefore, $z'[x \backslash \boldsymbol{a}] \preccurlyeq^* w'[x \backslash \boldsymbol{a}]$. It then remains to prove that $\Downarrow_x z' \preccurlyeq^* \Downarrow_x w'$. For example, if $z' = \langle t_1, t_2 \rangle$ and $w' = \langle u_1, u_2 \rangle$, we have to prove that $\langle \boldsymbol{\nu}x.t_1, \boldsymbol{\nu}x.t_2 \rangle \preccurlyeq^* \langle \boldsymbol{\nu}x.u_1, \boldsymbol{\nu}x.u_2 \rangle$, which means that we have to prove that $\boldsymbol{\nu}x.t_1 \preccurlyeq^* \boldsymbol{\nu}x.u_1$ just knowing that $t_1[x \backslash \boldsymbol{a}] \preccurlyeq^* u_1[x \backslash \boldsymbol{a}]$. However, with Howe's definition of $\preccurlyeq^*$, we would have had to prove that $t_1[x \backslash r] \preccurlyeq^* u_1[x \backslash r]$ for all closed term r.

4.3. *Consistency in the presence of* $\boldsymbol{\nu}$

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 below (see Sections 2.4 and 2.5 for details regarding the validity of Nuprl's inference rules).

First, note that because exceptions are canonical forms as mentioned in Section 4.1.1 above, if $a \mapsto^*$ `exc`(t_1, t_2), then $a \preccurlyeq b$ if there exists u_1 and u_2 such that $b \mapsto^*$ `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

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

(such as **ifint** or **ifaxiom**) for exceptions that would check whether a term computes to an exception because we cannot catch an exception without knowing 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 below. 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.

Before adding exceptions to Nuprl's computation system, the following [convergence] rule was key to reason about the approximation types of the form $t_1 \preceq t_2$ (where $\mathbb{P}$ is $\mathbb{U}_i$ for some level i):

$$
\begin{array}{l}
H \vdash t_1 \preceq t_2 \\
\quad \text{BY [convergence]} \\
\quad H, y : \mathtt{halts}(t_1) \vdash t_1 \preceq t_2 \\
\quad H \vdash \mathtt{halts}(t_1) \in \mathbb{P}.
\end{array}
$$

Given our new definition of $\preccurlyeq$, this rule is not valid anymore. Now, in addition to proving that $t_1 \preceq t_2$ is true assuming that t_1 computes to a value, we also have to prove that $t_1 \preceq t_2$ is true assuming that t_1 computes to an exception. Therefore, the [convergence] rule can now be stated as follows:

$$
\begin{array}{l}
H \vdash t_1 \preceq t_2 \\
\quad \text{BY [convergence]} \\
\quad H, y : \mathtt{halts}(t_1) \vdash t_1 \preceq t_2 \\
\quad H, y : \mathtt{isexc}(t_1) \vdash t_1 \preceq t_2 \\
\quad H \vdash \mathtt{halts}(t_1) \in \mathbb{P} \\
\quad H \vdash \mathtt{isexc}(t_1) \in \mathbb{P}.
\end{array}
$$

Let us now discuss some implications of [convergence] having changed. Using this rule, we used to be able to prove that for all $a, b \in \mathtt{Top}$, $a + b \simeq b + a$. This is not true anymore because, e.g., if a raises an exception and b diverges then $a + b$ raises the exception, while $b + a$ diverges. Also if a raises an exception e_1 and b raises an exception e_2, then $a + b$ raises e_1, while $b + a$ raises e_2. However, we can still prove that for all $a \in \mathbb{Z}$ and $b \in \mathtt{Top}$, $a + b \simeq b + a$, and for all $a \in \mathtt{Top}$ and $b \in \mathbb{Z}$, $a + b \simeq b + a$.

Let us sketch these proofs. For now, let a and b be unconstrained, i.e., let a and b be in Top (as mentioned in Section 2.5, it is enough to assume that a and b are in Base). To prove $a + b \simeq b + a$, we have to prove $a + b \preceq b + a$ and $b + a \preceq a + b$. Our new [convergence] rule says that to prove $a + b \preceq b + a$, we can assume that $a + b$ either has a value or raises an exception. If $a + b$ has a value, we conclude as before, i.e., we use one of our rules that says that if $a + b$ has a value then a and b are integers. Now let us assume that $a + b$ raises

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

an exception e. Given our computation system, we can derive that either (1) a raises e, or (2) a computes to an integer and b raises e. We now need a rule to that effect:

$$
\begin{array}{l}
H \vdash \downarrow C \\
\quad \text{BY } \texttt{[AddExceptionCases]} \\
\quad H \vdash \texttt{isexc}(a+b) \\
\quad H, x : \texttt{isexc}(a) \vdash C \ \lfloor \textbf{ext } u_1 \rfloor \\
\quad H, x : a \in \mathbb{Z}, y : \texttt{isexc}(b) \vdash C \ \lfloor \textbf{ext } u_2 \rfloor \\
\quad H \vdash a \in \texttt{Base} \\
\quad H \vdash b \in \texttt{Base}.
\end{array}
$$

Note that this rule does not say exactly what we wrote above. Using this rule, if $\texttt{isexc}(a+b)$ we only get to assume that $\texttt{isexc}(a)$ in our second subgoal, and not that $a+b$ and a compute to the same exception. This can be recovered using the following rule:

$$
\begin{array}{l}
H \vdash C \ \lfloor \textbf{ext } c \rfloor \\
\quad \text{BY } \texttt{[ExceptionBisimulation]} \\
\quad H \vdash \texttt{exc}(n,d) \preceq t \\
\quad H, [u : \texttt{Base}], [v : \texttt{Base}], [x : t \simeq \texttt{exc}(u,v)] \vdash C \ \lfloor \textbf{ext } c \rfloor \\
\quad H, u : \texttt{Base}, v : \texttt{Base} \vdash t \in \texttt{Base}.
\end{array}
$$

Using this rule, we can derive that if $\texttt{isexc}(a)$ then there exists some u and v such that $a \simeq \texttt{exc}(u,v)$. We are trying to prove that $a+b \preceq b+a$. Therefore, it is enough to prove $\texttt{exc}(u,v)+b \preceq b+\texttt{exc}(u,v)$. By computation, we get that it is enough to prove $\texttt{exc}(u,v) \preceq b+\texttt{exc}(u,v)$. Now if b was to diverge, or raise an exception e different from (not computationally equivalent to) $\texttt{exc}(u,v)$, or compute to a value that is not an integer then $b+\texttt{exc}(u,v)$ would diverge, raise e, and get stuck, respectively. In all these three cases, we would not be able to prove $\texttt{exc}(u,v) \preceq b+\texttt{exc}(u,v)$. However, we can prove this result assuming that $b \in \mathbb{Z}$. Similarly, if $a \in \mathbb{Z}$ and $\texttt{isexc}(b)$ (third subgoal of `[AddExceptionCases]`) then we can conclude that $a+b \preceq b+a$.

Let us now discuss two peculiarities of `[AddExceptionCases]`, namely (1) we have to prove that a and b are in Base, and (2) our conclusion C is squashed. Regarding (1), $a \in \texttt{Base}$ and $b \in \texttt{Base}$ are used to prove that the hypotheses introduced in `[AddExceptionCases]`'s second and third subgoals are well-formed propositions over the hypotheses H. Regarding (2), because we have squashed our conclusion C, it means that the extract of the sequent is $\star$. If the conclusion was not squashed, what could be the extract? We prove this rule in Nuprl's metatheory by analyzing the expression $\texttt{isexc}(a+b)$. As mentioned above, we can derive that $a+b$ raises an exception e, and therefore that either a raises e, or a computes to an integer and b raises e. In the first case, we use `[AddExceptionCases]`'s second subgoal to prove that our conclusion is true, and in the second case we use `[AddExceptionCases]`'s third subgoal to prove that our conclusion is true. Therefore, if we could provide an extract for our sequent, it would be a term of the form if a is an exception then e_1 else e_2. However, to do that we would have to be able to catch all exceptions, which our computation system does not allow.

We have similar 'exception cases' rules for the other non-canonical operators of our computation system.

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

As for [AddExceptionCases], to use [ExceptionBisimulation] we have to prove that $t \in \mathtt{Base}$, which is enough to prove that the hypothesis $t \simeq \mathtt{exc}(u,v)$ is a well-formed proposition. In [ExceptionBisimulation]'s second subgoal, the hypothesis u, and v are hidden to ensure that u and v do not occur in the extract c. If they were not hidden, and because u and v do not occur in our [ExceptionBisimulation]'s main subgoal $H \vdash C\ \lfloor \mathrm{ext}\ c \rfloor$, instead of c, we would have to come up with a term that given t, computes t to an exception $\mathtt{exc}(a,b)$ and extract the name a and the data b from that exception, and finally replace u by a and v by b in c. However, to do that we would need to be able to take apart any exception, which would mean that we would be able to catch any exception, which is not allowed by our computation system.

Here are a few additional rules that we have proved regarding exceptions:

$$\begin{array}{l} H \vdash C \\ \quad \text{BY } \texttt{[ExceptionConverges]} \\ \quad H \vdash \mathtt{exc}(n,d) \preceq \bot \end{array}$$

$$\begin{array}{l} H \vdash C \\ \quad \text{BY } \texttt{[ExceptionNotValue]} \\ \quad H \vdash \star \preceq \mathtt{exc}(n,d) \end{array}$$

$$\begin{array}{l} H \vdash a + \mathtt{exc}(n,d) \simeq \mathtt{exc}(n,d) \\ \quad \text{BY } \texttt{[AddException]} \\ \quad H \vdash a \in \mathbb{Z}. \end{array}$$

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:

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

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

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

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 SCPT. This is why **bound** starts off by checking whether its argument x is an integer less than 0. If a computation that uses **bound** converges and along the way applies f to some term k, we will be guaranteed that k is a natural number.

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

4.5. *Well-typedness*

To prove SCPT, we first prove that $\mathtt{M}(F)$ has type $\boldsymbol{\Pi} n{:}\mathbb{N}.\mathcal{B}_n \to \mathbb{N}_?$. As mentioned above, this term does not respect computation, i.e., it is not extensional over $F{\in}\mathcal{B} \to \mathbb{N}$, i.e., for two equal functions F and G in $\mathcal{B} \to \mathbb{N}$, $\mathtt{M}(F)$ is not necessarily equal to $\mathtt{M}(G)$ in $\mathbb{N}_?$. 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 $f{\equiv}g{\in}\mathcal{B}_n$, 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 $f{\equiv}g{\in}\mathcal{B}_k$ and let us prove $\mathtt{test}(F,k,f){\equiv}\mathtt{test}(F,k,g){\in}\mathbb{N}_?$. Unfolding test's definition, we have to prove

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

As we show below in Section 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

$$F(\mathtt{bound}(k,f,\boldsymbol{a})){\equiv}F(\mathtt{bound}(k,g,\boldsymbol{a})){\in}\mathbb{N}_{?\boldsymbol{a}}. \tag{3}$$

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

$$\mathtt{bound}(k,f,\boldsymbol{a}){\equiv}\mathtt{bound}(k,g,\boldsymbol{a}){\in}(\mathbb{N}_{?\boldsymbol{a}})^{\mathbb{N}}. \tag{4}$$

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

$$\mathtt{Force}(F){\in}(\mathbb{N}_{?\boldsymbol{a}})^{\mathbb{N}} \to \mathbb{N}_{?\boldsymbol{a}}, \tag{5}$$

where $\mathtt{Force}(F)$ is defined as $\lambda f.F(\mathtt{force}(f))$. Is Force necessary? Cannot we simply prove $F{\in}(\mathbb{N}_{?\boldsymbol{a}})^{\mathbb{N}} \to \mathbb{N}_{?\boldsymbol{a}}$? In other words, can we find a F in $\mathcal{B} \to \mathbb{N}$, 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

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

(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 Equation (5) 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 Equation (5), it is trivial to deduce that the equality (3) is true using equality (4).

Let us now prove Equation (5). We have to prove that for all F in $\mathcal{B} \to \mathbb{N}$, and f and g such that $f{\equiv}g{\in}(\mathbb{N}_{?\boldsymbol{a}})^{\mathbb{N}}$,

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

Let $f_0 = \mathtt{force}(f)$ and $g_0 = \mathtt{force}(g)$. Note that $f_0{\equiv}g_0{\in}(\mathbb{N}_{?\boldsymbol{a}})^{\mathbb{N}}$. We have to prove $F(f_0){\equiv}F(g_0){\in}\mathbb{N}_{?\boldsymbol{a}}$. First, we define a function force0 so that the function f' defined as $\lambda x.\mathtt{force0}(x,f)$ computes as f_0, except that on natural numbers, when f_0 returns $\mathtt{exc}_{\boldsymbol{a}}$, f'

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

returns 0 (this 0 could be any natural number):

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

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 $\texttt{exc}_{\boldsymbol{a}}$. 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 $C[f_0] \mapsto^* \texttt{exc}_{\boldsymbol{a}}$ and $C[g_0] \mapsto^* \texttt{exc}_{\boldsymbol{a}}$. This gives us that $F(f_0){\equiv}F(g_o){\in}\mathbb{N}_{?\boldsymbol{a}}$

4.6. *Interlude: Reasoning about* $\boldsymbol{\nu}$

In Section 4.1.3, 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. (See lemma `cequivc_fresh_subst2` in https://github.com/vrahli/NuprlInCoq/blob/master/continuity/stronger_continuity_defs0.v.)

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:* $\downarrow$*-existence of a modulus of continuity*

The first property we prove about the function `M` defined above in Section 4.4 is that for all f in $\mathcal{B}$, $\downarrow\boldsymbol{\Sigma}n{:}\mathbb{N}.\texttt{M}(F)\ n\ f =_{\mathbb{N}} F(f)$. This condition says that for all f there exists a n such that `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 $F(f){\equiv}F(\texttt{force}(f)){\in}\mathbb{N}$. Hence, there exists a natural number k such that $F(\texttt{force}(f)) \mapsto^* k$. As in the proof of $\texttt{WCP}_{\downarrow}$, we first compute the maximum of all the numbers occurring in that computation, and we instantiate our conclusion with b a number that is strictly greater than this maximum. We now have to prove $\texttt{M}(F)\ b\ f{\equiv}F(f){\in}\mathbb{N}$, or equivalently $\texttt{test}(F, b, f){\equiv}F(\texttt{force}(f)){\in}\mathbb{N}$. Unfolding `test`'s definition, we have to prove

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

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

As in Section 4.5, because we are trying to prove that this ν is in $\mathbb{N}$ and because $\mathbb{N}$ is flat, it is enough to prove for some name a such that $a\#F$ and $a\#f$:

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

Again, let us use the same simulation technique as before to prove that in any context C with no occurrence of a, if $C[\mathtt{force}(f)] \mapsto^* k$ then $C[\mathtt{bound}(b,f,a)] \mapsto^* k$. 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 $C[\mathtt{bound}(b,f,a)] \mapsto^* C'[\mathtt{bound}(b,f,a)]$.

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

4.8. *2nd condition: Uniformity of the testing function*

The second property we prove about the function $\mathtt{M}$ defined above in Section 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 $\mathtt{try}_a\ F(\mathtt{bound}(n,f,a)) \sim k$, for some a such that $a\#F$ and $a\#f$. By typing, we get that $F(f)$ computes to a natural number k'. Because $\mathtt{try}_a\ F(\mathtt{bound}(n,f,a))$ computes to a canonical form (the natural number k), we deduce that $F(\mathtt{bound}(n,f,a))$ also computes to a canonical form. This canonical form is either (1) an exception or (2) a value.

1. If $F(\mathtt{bound}(n,f,a))$ computes to an exception, then we get a contradiction: either the term computes to $\mathtt{exc}_a$ and then we obtain that $\mathtt{try}_a\ F(\mathtt{bound}(n,f,a)) \mapsto^* \star$ and $\star \neq k$; or it computes to an exception e with some other name than a and then $\mathtt{try}_a\ F(\mathtt{bound}(n,f,a)) \mapsto^* e$ and $e \neq k$. In both cases, we get a contradiction.
2. We now assume that $F(\mathtt{bound}(n,f,a))$ 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 Section 4.7. We prove that in any context C with no occurrence of a, if $C[\mathtt{force}(f)]$ computes to the natural number k', then $C[\mathtt{bound}(n,f,a)]$ computes to either k' or $\mathtt{exc}_a$. 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 $C[\mathtt{bound}(b,f,a)] \mapsto^* C'[\mathtt{bound}(b,f,a)]$ or $C[\mathtt{bound}(b,f,a)] \mapsto^* \mathtt{exc}_a$. We get that either (1) $F(\mathtt{bound}(b,f,a)) \mapsto^* k'$ and therefore $k = k'$ because our computation system is deterministic; or (2) $F(\mathtt{bound}(b,f,a)) \mapsto^* \mathtt{exc}_a$ and we get a contradiction because $k \neq \mathtt{exc}_a$.

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

4.9. *Nuprl's strong continuity inference rule*

Using the fact that SCPT (defined at the beginning of Section 4) is true in Nuprl's metatheory, 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 SCPT in Nuprl, where the first (outer) existential quantifier is $\downarrow$-squashed and the second (inner) one is not squashed. This lemma can be accessed by clicking the following hyperlink: strong-continuity2-no-inner-squash. Alternatively, the reader can search for the lemma with that name available here: http://www.nuprl.org/LibrarySnapshots/Published/Version2/Standard/continuity/index.html, and similarly for the other Nuprl lemmas mentioned below and highlighted in green. 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 coloured 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

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

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

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

Finally, we trivially derive a version of SCP where, as mentioned in the introduction, the first (outer) $\underline{\Sigma}$ is $\downarrow\Sigma$, and the second (inner) one is Σ, as proved by the Nuprl lemma strong-continuity2-no-inner-squash-unique. Therefore, because these versions of SCP are equivalent to $\mathtt{SCP}_{\downarrow}$, we also refer to them as $\mathtt{SCP}_{\downarrow}$.

5. Relations between WCP and SCP

As mentioned in Section 1, Bridges and Richman (1987, p. 119) state that SCP is equivalent to 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 $\downarrow$ [see also Escardo and Xu (2015)]. Therefore, for that equivalence to be true, we probably need the $\downarrow$-truncated version of $\mathrm{AC}_{1,0}$, which is true in Nuprl as discussed below in Section 5.3. Therefore, we only need to prove that $\mathtt{SCP}_{\downarrow}$ and $\mathtt{WCP}_{\downarrow}$ are equivalent. We prove below that $\mathtt{WCP}_{\downarrow}$ is a trivial consequence of $\mathtt{SCP}_{\downarrow}$, and that $\mathtt{SCP}_{\downarrow}$ is a consequence of $\mathtt{WCP}_{\downarrow}$ using the same 'trick' as the one used by Bridges and Richman to prove that UCP follows from

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

the Fan Theorem and WCP (Bridges and Richman 1987, p. 113) (see Section 6.1 below on uniform continuity).

5.1. *SCP$_{\downarrow}$ implies WCP$_{\downarrow}$*

Let us sketch the proof that $\mathtt{SCP}_{\downarrow} \to \mathtt{WCP}_{\downarrow}$. Let $F \in \mathcal{B} \to \mathbb{N}$ and $f \in \mathcal{B}$. We prove the formula $C = \downarrow \mathbf{\Sigma} n{:}\mathbb{N}.\ \mathbf{\Pi} g{:}\mathcal{B}.f =_{\mathcal{B}_n} g \to F(f) =_{\mathbb{N}} F(g)$. Because our conclusion C is squashed, we can unsquash $\mathtt{SCP}_{\downarrow}$'s outer $\mathbf{\Sigma}$, and we get a $M \in \mathbf{\Pi} n{:}\mathbb{N}.\mathcal{B}_n \to \mathbb{N}_{\mathtt{U}}$ and a function

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

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 $B \in \mathbf{\Pi} m{:}\mathbb{N}.\mathtt{isl}(M\ m\ f) \to m =_{\mathbb{N}} n$.

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 $f =_{\mathcal{B}_n} g$. 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 $B' \in \mathbf{\Pi} m{:}\mathbb{N}.\mathtt{isl}(M\ m\ g) \to m =_{\mathbb{N}} n'$.

Because $f =_{\mathcal{B}_n} g$, 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. *WCP$_{\downarrow}$ implies SCP$_{\downarrow}$*

This section proves that the following 'skolemized' version of $\mathtt{WCP}_{\downarrow}$ implies $\mathtt{SCP}_{\downarrow}$ (see Nuprl lemma `weak-continuity-implies-strong1`):

$$\mathtt{skWCP} = \mathbf{\Pi} F{:}\mathcal{B} \to \mathbb{N}. \\ \downarrow \mathbf{\Sigma} M{:}\mathcal{B} \to \mathbb{N}. \\ \mathbf{\Pi} f, g{:}\mathcal{B}.(f =_{\mathbb{N}_{M(f)} \to \mathbb{N}} g) \to (F(f) =_{\mathbb{N}} F(g)).$$

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

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

$$\downarrow \mathbf{\Sigma} M{:}\mathbf{\Pi} n{:}\mathbb{N}.\mathcal{B}_n \to \mathbb{N}_{\mathtt{U}}. \\ \mathbf{\Pi} f{:}\mathcal{B}. \quad \mathbf{\Sigma} n{:}\mathbb{N}.M\ n\ f =_{\mathbb{N}_{\mathtt{U}}} \mathtt{inl}(F(f)) \\ \wedge\ \mathbf{\Pi} n{:}\mathbb{N}.\mathtt{isl}(M\ m\ f) \to M\ n\ f =_{\mathbb{N}_{\mathtt{U}}} \mathtt{inl}(F(f)).$$

Instantiating skWCP with F, we get (because our conclusion C is $\downarrow$-truncated, we can unsquash our hypothesis)

— a $M \in \mathcal{B} \to \mathbb{N}$ (F's modulus of continuity)

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

— and a $G \in \mathbf{\Pi} f,g{:}\mathcal{B}.(f =_{\mathbb{N}_{M(f)}\to\mathbb{N}} g) \to (F(f) =_{\mathbb{N}} F(g))$.

Bridges and Richman's trick is to also use the modulus of continuity of M. Therefore, instantiating `skWCP` with M, we get

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

Let us now unsquash our conclusion C and instantiate it with

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

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 Berger and Oliva (2006) and Escardo and Oliva (2015)]. If $a \in \mathcal{B}_n$ and $k \in \mathbb{N}$, then $a_{n,k} \in \mathcal{B}$. We now have to prove that assuming that $f \in \mathcal{B}$, then

$$\mathbf{\Sigma} n{:}\mathbb{N}.B\ n\ f =_{\mathbb{N}_\cup} \texttt{inl}(F(f)), \tag{6}$$

$$\mathbf{\Pi} n{:}\mathbb{N}.\texttt{isl}(B\ n\ f) \to (B\ n\ f =_{\mathbb{N}_\cup} \texttt{inl}(F(f))). \tag{7}$$

Equality (7) follows from G. We now prove Equation (6) by instantiating it with $m = \texttt{max}(M(f), X(f))$. We have to prove $B\ m\ f =_{\mathbb{N}_\cup} \texttt{inl}(F(f))$, i.e.,

$$\texttt{if } M(a_{m,0}) \leqslant m, \texttt{ then inl}(F(a_{m,0})) \texttt{ else inr}(\star) =_{\mathbb{N}_\cup} \texttt{inl}(F(f)). \tag{8}$$

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 (8), it remains to prove that its conditional is true, i.e., $M(a_{m,0}) \leqslant m = \texttt{max}(M(f), X(f))$. 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 we get to assume $M(f) =_{\mathbb{N}} M(a_{m,0})$, which gives us that $M(a_{m,0}) \leqslant m$.

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 (product type) and existential (sum type) quantifiers. We can prove that these rules are true in our predicative Coq model (Anand and Rahli 2014a,b) 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 to prove some of these rules, namely FunctionalChoice_on as defined in `http://coq.inria.fr/cocorico/CoqAndAxioms`.

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 $\downarrow\mathbf{\Sigma}$ or $\downarrow\mathbf{\Sigma}$. In that case, it is not obvious anymore which instances of AC are consistent with or provable in Nuprl.

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

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 $\mathrm{AC}_{n,m}$, where $n, m \in \{0, 1\}$ (Troelstra and VanDalen 1988, p. 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 Rahli and Bickford (2015), we proved the $\downarrow$-squashed version of $\mathrm{AC}_{0,0}$, where $\underline{\Sigma}$ is $\downarrow\Sigma$, once again conducting the proof first in the metatheory, and then reflecting the metatheoretical 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. Our proof is similar to the one presented in Berardi *et al.* (1998), where the authors added infinite sequences to their term syntax (denoted $\lambda\!\!\lambda x.M_x$, where M_1, M_2, ..., is an infinite sequence of terms) in order to prove that some bar recursion operator denoted Φ realizes the negative translation of the axiom of choice. In Rahli and Bickford (2015), we added all Coq functions from numbers to numbers to Nuprl's term syntax in order to prove the $\downarrow$-squashed version of $\mathrm{AC}_{0,0}$. We also proved directly in Nuprl the $\downarrow$-squashed versions of AC, where $\underline{\Sigma}$ is $\downarrow\Sigma$, namely $\mathrm{AC}_{0,\mathrm{X}}$ (see Nuprl lemma axiom-choice-0X-quot) and $\mathrm{AC}_{1,\mathrm{X}}$ (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, which says

$$\Pi A{:}\mathtt{Type}.\downarrow \mathtt{canonicalizable}(A) \to \Pi B{:}\mathtt{Type}.\mathtt{AC}(A, B), \tag{9}$$

where

$$\mathtt{canonicalizable}(A) = (\Sigma f{:}\mathtt{Base}^A.\Pi x{:}A.f(x) =_A x)$$

and

$$\begin{aligned}\mathtt{AC}(A, B) = \ & \Pi P{:}A \to B \to \mathbb{P}. \\ & \Pi a{:}A.\downarrow \Sigma b{:}B.\ P\ a\ b \\ & \Rightarrow\ \downarrow \Sigma f{:}B^A.\ \Pi a{:}A.P\ a\ f(a).\end{aligned}$$

We get $\mathrm{AC}_{0,\mathrm{X}}$ and $\mathrm{AC}_{1,\mathrm{X}}$, from the fact that both $\mathbb{N}$ and $\mathcal{B}$ are canonicalizable. To prove that $\mathbb{N}$ is canonicalizable, we instantiate $\mathtt{canonicalizable}(\mathbb{N})$ with $\lambda x.x$, which has the right type, i.e., $\mathtt{Base}^{\mathbb{N}}$ because $\mathbb{N}$ is a subtype of `Base`. To prove that $\mathcal{B}$ is canonicalizable, we instantiate $\mathtt{canonicalizable}(\mathcal{B})$ with $(\lambda f.\lambda x.\mathtt{if}\ 0 \leqslant x\ \mathtt{then}\ f(x)\ \mathtt{else}\ \bot)$, which has the right type, i.e., $\mathtt{Base}^{\mathcal{B}}$ because (with some difficulty) we can prove that if $f \in \mathcal{B}$ then $(\lambda x.\mathtt{if}\ 0 \leqslant x\ \mathtt{then}\ f(x)\ \mathtt{else}\ \bot)$ is in `Base`: see Nuprl lemma canonicalizable-baire-direct.

To prove Equation (9), we first show

$$\Pi A{:}\mathtt{Type}.\mathtt{choice}(A) \to \Pi B{:}\mathtt{Type}.\mathtt{AC}(A, B), \tag{10}$$

where

$$\mathtt{choice}(A) = \Pi P{:}A \to \mathbb{P}.(\Pi a{:}A.\downarrow P(a)) \iff (\downarrow \Pi a{:}A.P(a)),$$

and then we show that if a type A is canonicalizable, then it has a choice principle, i.e., $\mathtt{choice}(A)$:

$$\Pi A{:}\mathtt{Type}.\downarrow \mathtt{canonicalizable}(A)\ \Rightarrow\ \mathtt{choice}(A). \tag{11}$$

This is enough to prove Equation (9), but we now have to prove Equations (10) and (11). First, we prove the axiom of choice (10) as follows: Using $\mathtt{choice}(A)$, we rewrite our

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

hypothesis $\mathbf{\Pi}a{:}A.\downarrow\mathbf{\Sigma}b{:}B.\ P\ a\ b$, into the formula $\downarrow\mathbf{\Pi}a{:}A.\mathbf{\Sigma}b{:}B.P\ a\ b$, which we can now unsquash because our conclusion is squashed. Finally, we use the non-squashed axiom of choice on our hypothesis $\mathbf{\Pi}a{:}A.\mathbf{\Sigma}b{:}B.P\ a\ b$ to get $\mathbf{\Sigma}f{:}B^A.\mathbf{\Pi}a{:}A.P\ a\ f(a)$, and unsquash our conclusion, which is now the same as our hypothesis.

Let us now focus on Equation (11). To prove it, we first prove the following implication:

$$\mathbf{\Pi}A{:}\mathtt{Type}.A \sqsubseteq \mathtt{Base} \ \Rightarrow\ \mathtt{choice}(A), \tag{12}$$

where Nuprl's subtype relation $A \sqsubseteq B$ is defined as $\lambda x.x \in A \to B$. Given Equation (12), it is easy to derive Equation (11). Note that $A \sqsubseteq \mathtt{Base}$ implies $\downarrow\mathtt{canonicalizable}(A)$, but the other direction is not true. For example, as mentioned above, $\mathcal{B}$ is canonicalizable but it is not a subtype of Base because $\lambda x.x$ and $\lambda x.x + 0$ are equal in $\mathcal{B}$ but are not computationally equal, i.e., they are not equal in Base. Therefore, Equation (12) follows from Equation (11), but the other direction, which we need, is easy but slightly more complicated. To prove Equation (11) from Equation (12), we instantiate Equation (12) with $T \cap \mathtt{Base}$ and the rest follows easily.

Let us sketch the proof of Equation (12) here. Let us assume $A \sqsubseteq \mathtt{Base}$ and $\mathbf{\Pi}a{:}A.\downarrow P(a)$ for some type A and some predicate P on A, and let us prove $\downarrow\mathbf{\Pi}a{:}A.P(a)$ (the other direction is trivial). We assume that F inhabits our hypothesis $\mathbf{\Pi}a{:}A.\downarrow P(a)$, and we prove that the same F inhabits our conclusion $\downarrow\mathbf{\Pi}a{:}A.P(a)$. The proof now gets a bit technical because it relies heavily on the semantics of Nuprl's sequents and on the Base type. We now use our pointwise functionality rule which, in the case of equalities, says

$$\begin{array}{l} H, x : A, J \vdash t_1 = t_2 \in T \\ \quad \text{BY } \texttt{[functionality]} \\ \quad H, x : A, J \vdash A \in \mathbb{U}_i \\ \quad H, x : \mathtt{Base}, y : \mathtt{Base}, z : x = y \in A, J \vdash t_1 = t_2[x\backslash y] \in T. \end{array}$$

Using this rule, we now get to assume $F \in \mathtt{Base}$, $F' \in \mathtt{Base}$ and $F = F' \in \mathbf{\Pi}a{:}A.\downarrow P(a)$, and we have to prove $F = F' \in \downarrow\mathbf{\Pi}a{:}A.P(a)$. To prove this, it is enough to prove that $F \in \mathbf{\Pi}a{:}A.P(a)$ and $F' \in \mathbf{\Pi}a{:}A.P(a)$. Without loss of generality, we only prove $F \in \mathbf{\Pi}a{:}A.P(a)$ here. Using function extensionality, it is enough to prove that given $a \in A$, $F(a) \in P(a)$. It turns out that for all type T, $\downarrow T \cap \mathtt{Base}$ and $T \cap \mathtt{Base}$ are extensionally equal types, i.e., they have the same PERs, where $A \cap B$ is the binary intersection of A and B, defined as $\cap b{:}\mathbb{B}.\mathtt{if}\ b\ \mathtt{then}\ A\ \mathtt{else}\ B$, where $\mathbb{B}$ is the Boolean type. This extensional equality is only true because we intersect the types with Base, and this is why we used [functionality] above to get $F \in \mathtt{Base}$. Because $a \in A$ and $A \sqsubseteq \mathtt{Base}$, and because $F \in \mathtt{Base}$, then we get that $F(a) \in \mathtt{Base}$. Therefore, to prove that $F(a) \in P(a)$, and using the fact that $P(a) \cap \mathtt{Base}$ and $\downarrow P(a) \cap \mathtt{Base}$ are extensionally equal types, it is enough to prove $F(a) \in \downarrow P(a)$, which we get from our hypothesis $F = F' \in \mathbf{\Pi}a{:}A.\downarrow P(a)$. This concludes the proof of Equation (12) and therefore also of Equation (9).

6. Applications

Using $\mathtt{SCP}_{\downarrow}$, we proved in Nuprl a $\downarrow$-truncated version of the uniform continuity principle UCP (see Section 1) from the fan theorem, and then a fully unsquashed version of this

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

principle using Escardó and Xu's method (Escardo and Xu 2015) (see Section 6.1). We write $\mathtt{UCP}_{\downarrow}$ for the version of UCP, where $\underline{\Sigma}$ is $\downarrow\Sigma$. Using $\mathtt{UCP}_{\downarrow}$, we then proved Brouwer's theorem, which says that real functions defined on the unit interval are uniformly continuous (see Section 6.2).

We have recently proved the validity of a non-computational version of Bar Induction w.r.t. Nuprl's semantics using our Coq model (Rahli *et al.* 2017). From this version, we derived in Nuprl a version of BID, which has computation content. Following Kleene, given that $\mathtt{SCP}_{\downarrow}$ is true we also derived BIM (Kleene and Vesley 1965, p. 78) [see also Dummett's Theorem 3.8 (Dummett 1977, p. 64)]: see Nuprl lemma monotone-bar-induction1.

As discussed in Section 6.3, we could already prove the negation of the law of excluded middle without using continuity, and instead using Nuprl's 'computation types.' We can now also derive the negation of the law of excluded middle using continuity.

As mentioned in Section 1, Brouwer's continuity principle is traditionally stated using relations instead of functions as in WCP and SCP defined above. We show in Section 6.4 that we can derive a more traditional version of Brouwer's continuity principle stated using relations instead of functions, from SCP and a $\downarrow$-squashed version of $\mathrm{AC}_{1,0}$, which is true in Nuprl as mentioned above in Section 5.3.

We have also recently used exceptions to implement the constructive content of the completeness result of intuitionistic first-order logic proved in Constable and Bickford (2014). 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. *$\mathtt{UCP}_{\downarrow}$ follows from FT and $\mathtt{SCP}_{\downarrow}$.* Using BID, we proved that the Fan Theorem (FT) is true: see Nuprl lemma fan_theorem. We then derived $\mathtt{UCP}_{\downarrow}$ from FT and $\mathtt{SCP}_{\downarrow}$: see Nuprl lemma strong-continuity2-implies-uniform-continuity. Let us sketch that proof here. The version of FT that we have proved in Nuprl is

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

where $\mathtt{Dec}(T)$ is defined as $T+\neg T$. The fan theorem says that if X is a decidable bar, then it is uniform. We also use the following corollary of $\mathtt{SCP}_{\downarrow}$ for functions on the Cantor space instead of the Baire space:

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

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

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

$$\downarrow \mathbf{\Sigma} n{:}\mathbb{N}.\ \mathbf{\Pi} f, g{:}\mathcal{C}.f =_{\mathcal{C}_n} g \to F(f) =_{\mathbb{N}} F(g). \tag{13}$$

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

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

We now instantiate FT using $X = \lambda n.\lambda f.\mathtt{isl}(M\ n\ f)$. We now have to prove FT's first hypothesis (1) $\mathbf{\Pi} f{:}\mathcal{C}.\downarrow\mathbf{\Sigma} n{:}\mathbb{N}.\mathtt{isl}(M\ n\ f)$, which follows from G; as well as its second hypothesis (2) $\mathbf{\Pi} n{:}\mathbb{N}.\mathbf{\Pi} f{:}\mathcal{C}_n.\mathtt{Dec}(\mathtt{isl}(M\ n\ f))$, which is trivial; and we get a $k \in \mathbb{N}$ and $A \in \mathbf{\Pi} f{:}\mathcal{C}.\mathbf{\Sigma} n{:}\mathbb{N}_k.\mathtt{isl}(M\ n\ f)$. We unsquash and instantiate our conclusion (13) using k. We have to prove that $F(f) =_{\mathbb{N}} F(g)$ assuming that $f =_{\mathcal{C}_k} g$ for f and g in $\mathcal{C}$. From G, we get

— $\pi_1(G(f)) \in \mathbf{\Pi} n{:}\mathbb{N}.\mathtt{isl}(M\ n\ f) \to M\ n\ f =_{\mathbb{N}_{\mathtt{U}}} \mathtt{inl}(F(f))$
— $\pi_1(G(g)) \in \mathbf{\Pi} n{:}\mathbb{N}.\mathtt{isl}(M\ n\ g) \to M\ n\ g =_{\mathbb{N}_{\mathtt{U}}} \mathtt{inl}(F(g))$,

and from A (applying A to f), we get a $i \in \mathbb{N}_k$ and that $\mathtt{isl}(M\ i\ f)$. Therefore, because $f =_{\mathcal{C}_k} g$, we get $M\ i\ f =_{\mathbb{N}_{\mathtt{U}}} M\ i\ g$ and $\mathtt{isl}(M\ i\ g)$, which means that [from $\pi_1(G(f))$ and $\pi_1(G(g))$]

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

Therefore, $\mathtt{inl}(F(f)) =_{\mathbb{N}_{\mathtt{U}}} \mathtt{inl}(F(g))$, and we conclude that $F(f) =_{\mathbb{N}} F(g)$.

6.1.2. *$\texttt{UCP}_{\downarrow}$ follows from FT and skWCP.* Following Bridges and Richman's proof of their Theorem 3.2 (Bridges and Richman 1987, p. 113), which says that weak continuity and the fan theorem imply uniform continuity, we have proved the Nuprl lemma fan+weak-continuity-implies-uniform-continuity, which says that FT and skWCP imply $\texttt{UCP}_{\downarrow}$ (as mentioned in Section 5, skWCP and $\texttt{WCP}_{\downarrow}$ are equivalent). Their proof is slightly more involved than the one presented above in Section 6.1.1 and uses the 'trick' mentioned in Section 5.2 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 $\texttt{UCP}_{\downarrow}$.* Finally, we can get rid of the truncation operator $\downarrow$ in $\texttt{UCP}_{\downarrow}$ following exactly Escardó and Xu's proof (Escardo and Xu 2015, Section 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 (The Univalent Foundations Program 2013), we can untruncate it. Escardó and Xu's proof goes in three main steps: our Nuprl lemma uniform-continuity-pi-dec corresponds to their Lemma 4; our Nuprl lemma prop-truncation-implies corresponds to

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

their Lemma 5; and finally our 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 $\alpha \in \mathbb{R}$ is a regular sequence of integers. This means that $\alpha \in \mathbb{N}^+ \to \mathbb{Z}$ and $\boldsymbol{\Pi} n,m{:}\mathbb{N}^+.|n * \alpha(m) - m * \alpha(n)| \leqslant 2(n+m)$. This differs from, but is equivalent to, Bishop's definition of real numbers as regular sequences of rationals (Bishop and Bridges 1985). Two regular sequences α and β represent the same real number if $\boldsymbol{\Pi} n{:}\mathbb{N}^+.|\alpha(n) - \beta(n)| \leqslant 4$, 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 $\boldsymbol{\Pi} n{:}\mathbb{N}^+.\alpha(n) \leqslant \beta(n) + 4$, then $\alpha \leqslant \beta$.

The closed interval $[\alpha,\beta]$ is the set type $\{x : \mathbb{R} \mid \alpha \leqslant x \leqslant \beta\}$. 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

$$\mathrm{FUN}(f,\alpha,\beta) = \boldsymbol{\Pi} x,y{:}[\alpha,\beta].x =_r y \;\Rightarrow\; f(x) =_r f(y).$$

A stronger condition – called strong extensionality in Coq's CoRN library (Krebbers and Spitters 2011) – is

$$\mathrm{SFUN}(f,\alpha,\beta) = \boldsymbol{\Pi} x,y{:}[\alpha,\beta].f(x) \# f(y) \;\Rightarrow\; x \# y.$$

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

$$\begin{array}{l}\mathrm{CONT}(f,\alpha,\beta)\\ =\ \boldsymbol{\Pi}\epsilon > 0.\\ \quad\ \boldsymbol{\Sigma}\delta > 0.\\ \quad\quad \boldsymbol{\Pi} x,y{:}[\alpha,\beta].|x-y| \leqslant \delta \to |f(x) - f(y)| \leqslant \epsilon.\end{array}$$

(The Nuprl lemmas mentioned below are available at the following address: `http://www.nuprl.org/LibrarySnapshots/Published/Version2/Mathematics/reals/index.html`.) Using the fact from Section 6.1 that functionals of type $\mathcal{C} \to \mathbb{Z}$ are uniformly continuous, we proved in Nuprl that for intervals $[\alpha,\beta]$, we have both (see Nuprl lemmas real-fun-iff-continuous and real-sfun-iff-continuous):

$$\begin{array}{l}\mathrm{CONT}(f,\alpha,\beta) \Leftrightarrow \mathrm{FUN}(f,\alpha,\beta)\\ \mathrm{CONT}(f,\alpha,\beta) \Leftrightarrow \mathrm{SFUN}(f,\alpha,\beta).\end{array}$$

Thus, for Bishop's definition of real function, it is correct to say that all functions on closed intervals $[\alpha,\beta]$ are uniformly continuous (Brouwer's theorem).

6.3. *Status of the law of excluded middle*

↓-Squashed LEM. As explained in Anand and Rahli (2014a), following Crary (1998) as well as Kopylov and Nogin (2001), we have proved (Anand and Rahli 2014b, Section 5.2) that the following weak version of the law of excluded middle (LEM) is consistent with Nuprl:

$$\boldsymbol{\Pi} P{:}\mathbb{U}_i.\downarrow(P + \neg P).$$

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

Because the computational content of the disjoint union is erased (using the squashing operator $\downarrow$), one cannot use this to construct a 'magical' decider for all propositions.

Anand's proof of ¬LEM. We can also prove directly in Nuprl the negation of LEM:

$$\neg\Pi P{:}\mathbb{U}_i.P+\neg P.$$

The simplest proof we know of is due to Abhishek Anand. Let us repeat it here. First, let us prove that we cannot decide the halting problem:

$$\neg\Pi t{:}\texttt{Base}.\texttt{halts}(t)+\neg\texttt{halts}(t). \tag{14}$$

From Equation (14), it follows trivially that the non-squashed LEM is false. Let us prove Equation (14). Assume $\Pi t{:}\texttt{Base}.\texttt{halts}(t)+\neg\texttt{halts}(t)$. Let us call that function *magic*. Using congruence, and because $\bot \preccurlyeq \star$, we can prove

$$\begin{array}{l}(\lambda x.\texttt{if}\ (\mathit{magic}\ x)\ \texttt{then}\ \bot\ \texttt{else}\ \star)\ \bot\\ \leq(\lambda x.\texttt{if}\ (\mathit{magic}\ x)\ \texttt{then}\ \bot\ \texttt{else}\ \star)\ \star.\end{array}$$

Therefore, by computation, we get

$$\texttt{if}\ (\mathit{magic}\ \bot),\ \texttt{then}\ \bot\ \texttt{else}\ \star \leq \texttt{if}\ (\mathit{magic}\ \star)\ \texttt{then}\ \bot\ \texttt{else}\ \star. \tag{15}$$

The term $(\mathit{magic}\ \bot)$ has type $\texttt{halts}(\bot)+\neg\texttt{halts}(\bot)$. Because $\bot$ diverges, we get that $(\mathit{magic}\ \bot)$ is a right injection. Similarly, the term $(\mathit{magic}\ \star)$ has type $\texttt{halts}(\star)+\neg\texttt{halts}(\star)$. Because $\star$ converges, we get that $(\mathit{magic}\ \star)$ is a left injection. Therefore, from the approximation (15), we get $\star \leq \bot$, which we can prove to be false. This concludes our proof of Equation (14).

$\downarrow$-Squashed LEM. Similarly, we can prove the negation of the $\downarrow$-squashed LEM:

$$\neg\Pi t{:}\texttt{Base}.\downarrow(\texttt{halts}(t)+\neg\texttt{halts}(t)).$$

¬LEM from continuity. Using the weak continuity principle, following, e.g., Veldman's proof (Veldman 2001), we can prove that we cannot decide whether a sequence of natural number is equal to $\lambda x.0$ or not:

$$\neg\Pi s{:}\mathcal{B}.(s =_{\mathcal{B}} \lambda x.0)+\neg(s =_{\mathcal{B}} \lambda x.0). \tag{16}$$

Again, from Equation (16), it then follows trivially that the non-squashed LEM is false. Let us repeat the proof of Equation (16) here. Assume

$$\Pi s{:}\mathcal{B}.(s =_{\mathcal{B}} \lambda x.0)+\neg(s =_{\mathcal{B}} \lambda x.0).$$

Let us call that function *magic*. We instantiate $\texttt{WCP}_{\downarrow}$ using the following function F in $\mathcal{B} \to \mathbb{N}$: $\lambda s.\texttt{if}\ (\mathit{magic}\ s)\ \texttt{then}\ 0\ \texttt{else}\ 1$ and the following function f in $\mathcal{B}$: $\lambda x.0$. We get $\downarrow\Sigma n{:}\mathbb{N}.\ \Pi g{:}\mathcal{B}.f =_{\mathcal{B}_n} g \to F(f) =_{\mathbb{N}} F(g)$. Because we are now proving False, we can unsquash this hypothesis and we get a n, which is the modulus of continuity of F at f, as well as a function of type $\Pi g{:}\mathcal{B}.f =_{\mathcal{B}_n} g \to F(f) =_{\mathbb{N}} F(g)$. Let us now instantiate this formula using the following function g of type $\mathcal{B}$: $\lambda x.\texttt{if}\ x{<}n\ \texttt{then}\ 0\ \texttt{else}\ 1$. Because f and g agree upto n, we get $F(f) =_{\mathbb{N}} F(g)$, i.e.

$$\begin{array}{l}\texttt{if}\ (\mathit{magic}\ f)\ \texttt{then}\ 0\ \texttt{else}\ 1\\ =_{\mathbb{N}}\ \texttt{if}\ (\mathit{magic}\ g)\ \texttt{then}\ 0\ \texttt{else}\ 1.\end{array} \tag{17}$$

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

Because $f =_{\mathcal{B}} \lambda x.0$, we get that $(magic\ f)$ is a left injection, and because $g =_{\mathcal{B}} \lambda x.0$ is not true (starting from n the two sequences differ), we get that $(magic\ g)$ is a right injection. Therefore, from equality (17), we obtain $0 =_{\mathbb{N}} 1$, which is false and this concludes the proof of Equation (16).

6.4. *Relational continuity principle*

Brouwer's continuity principle is often stated using a relational form such as in Kleene and Vesley (1965), Dummett (1977) and Troelstra and VanDalen (1988). In that form, one does not assume the existence of a function of type $\mathcal{B} \to \mathbb{N}$ but one assumes that there exists a predicate P of type $\mathcal{B} \to \mathbb{N} \to \mathbb{P}$ such that for all f in $\mathcal{B}$, there exists a n in $\mathbb{N}$ such that $P\ f\ n$. If the existential is interpreted as being $\mathbf{\Sigma}$, then we can trivially obtain a function F of type $\mathcal{B} \to \mathbb{N}$ such that for all f in $\mathcal{B}$, $P\ f\ (F\ f)$. If the existential is interpreted as being $\downarrow\mathbf{\Sigma}$, then using the $\downarrow$-squashed version of $\mathrm{AC}_{1,0}$, which is true in Nuprl as explained in Section 5.3, we can deduce $\downarrow\mathbf{\Sigma}F{:}\mathcal{B} \to \mathbb{N}.\ \mathbf{\Pi}f{:}\mathcal{B}.P\ f\ (F\ f)$ from $\mathbf{\Pi}f{:}\mathcal{B}.\downarrow\mathbf{\Sigma}n{:}\mathbb{N}.\ P\ f\ n$. Therefore, because the $\mathbf{\Sigma}$ in the conclusion of $\mathtt{SCP}_{\downarrow}$ is $\downarrow$-squashed, we can deduce the following variant of $\mathtt{SCP}_{\downarrow}$:

$$
\begin{array}{l}
\mathbf{\Pi}P{:}\mathcal{B} \to \mathbb{N} \to \mathbb{P}.\\
\quad (\mathbf{\Pi}f{:}\mathcal{B}.\downarrow\mathbf{\Sigma}n{:}\mathbb{N}.\ P\ f\ n)\\
\quad \Rightarrow\ \ \downarrow\mathbf{\Sigma}M{:}(\mathbf{\Pi}n{:}\mathbb{N}.\mathcal{B}_n \to \mathbb{N}_{\mathtt{U}}).\\
\qquad\quad \mathbf{\Pi}f{:}\mathcal{B}.\\
\qquad\qquad \mathbf{\Sigma}n,k{:}\mathbb{N}.\\
\qquad\qquad\qquad\quad M\ n\ f =_{\mathbb{N}_{\mathtt{U}}} \mathtt{inl}(k)\\
\qquad\qquad\qquad \wedge\ P\ f\ k\\
\qquad\qquad\qquad \wedge\ \mathbf{\Pi}m{:}\mathbb{N}.\mathtt{isl}(M\ m\ f) \to m =_{\mathbb{N}} n.
\end{array}
$$

6.5. *Bounded continuity principle*

Let us now present another variant of Brouwer's continuity principle that is derivable from the one presented above in Section 6.4:

$$
\begin{array}{l}
\mathbf{\Pi}P{:}(\mathcal{B} \to \mathbb{N} \to \mathbb{P}).\\
\quad (\mathbf{\Pi}f{:}\mathcal{B}.\downarrow\mathbf{\Sigma}n{:}\mathbb{N}.\ P\ f\ n)\\
\quad \Rightarrow\ \ \downarrow\mathbf{\Sigma}M{:}(\mathbf{\Pi}n{:}\mathbb{N}.\mathcal{B}_n \to (\mathbb{N}_n{+}\mathtt{Unit})).\\
\qquad\quad \mathbf{\Pi}f{:}\mathcal{B}.\\
\qquad\qquad \mathbf{\Sigma}n{:}\mathbb{N}.\\
\qquad\qquad\quad \mathbf{\Sigma}k{:}\mathbb{N}_n.\\
\qquad\qquad\qquad\quad P\ f\ k\\
\qquad\qquad\qquad \wedge\ M\ n\ f =_{\mathbb{N}_n+\mathtt{Unit}} \mathtt{inl}(k)\\
\qquad\qquad\qquad \wedge\ \mathbf{\Pi}m{:}\mathbb{N}.\mathtt{isl}(M\ m\ f) \to m =_{\mathbb{N}} n.
\end{array}
$$

This version differs from the one presented in Section 6.4 as follows: here M is of type $(\mathbf{\Pi}n{:}\mathbb{N}.\mathcal{B}_n \to (\mathbb{N}_n{+}\mathtt{Unit}))$ instead of $(\mathbf{\Pi}n{:}\mathbb{N}.\mathcal{B}_n \to (\mathbb{N}{+}\mathtt{Unit}))$ in Section 6.4, i.e., we are guaranteed that the modulus of continuity n of P at f that M returns will be larger than the value k such that $(P\ f\ k)$ is true (or taking P as a function as in Section 6.4, that

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

$P\ f < n$) – this is especially useful to define bar recursion operators as done in Rahli *et al.* (2016).

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 (Gabbay and Pitts 1999). This work then led to the design of the so-called Nominal Logic (Pitts 2001), 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 (Bojanczyk *et al.* 2012; Cheney 2009, 2012; Cheney and Urban 2004, 2008; Crole and Nebel 2013; Fairweather *et al.* 2015; Pitts 2010; Pitts and Gabbay 2000; Pitts *et al.* 2015; Pottier 2007; Pouillard and Pottier 2010; Schopp and Stark 2004; Shinwell 2006; Shinwell *et al.* 2003; Westbrook *et al.* 2009) (to cite only a few). We now describe some of these systems.

FreshML. For example, FreshML (Pitts and Gabbay 2000; Shinwell *et al.* 2003) 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 higher-order abstract syntax in order to deal with names and binders. These ideas were then also ported to OCaml (Shinwell 2006). Following this line of work, Pure FreshML (Pottier 2007) is a pure (in the sense that name generation is not an observable side effect) version of FreshML (Shinwell *et al.* 2003) that ensures fresh atoms do not escape their scopes.

Nominal type theories. Schopp and Stark (Schopp 2006; Schopp and Stark 2004) 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 either be modelled as 'fresh functions' or as pairs, which are members of 'non-standard fresh' $\boldsymbol{\Pi}^*$ and $\boldsymbol{\Sigma}^*$ 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 $\mathcal{N}$ Gabbay and Pitts (1999). In our paper, we only focus on computational aspects of freshness.

Cheney designed simple nominal type theory (SNTT) (Cheney 2009), 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

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

designed with decidable typechecking in mind. Cheney extended SNTT to a dependent type theory, called $\lambda^{\Pi\mathcal{N}}$ (Cheney 2012), with dependent products (Π) and dependent name-abstraction types ($\mathcal{N}$). 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) (Westbrook 2008; Westbrook *et al.* 2009) can also be seen as an extension of SNTT with inductive constructions, or similarly as an extension of Calculus of Inductive Constructions (Coquand and Paulin 1988) with nominal features such as name abstraction and concretion operators, and pattern matching operators for names and name abstractions.

Pitts' Nominal System T (Pitts 2010) extends System T with nominal features such as a fresh operator ν à la Odersky and a name-swapping operator. As opposed to the systems mentioned above, this system has ordinary non-bunched contexts. FreshMLTT (Pitts *et al.* 2015) is a dependent type theory that has name-abstraction and name-concretion operators, as well as a name-swapping operator and a fresh operator ν 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 metatheory (Section 3), and (2) named exceptions to exhibit it in the theory (Section 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 Section 4.8, we used the fact that the exception $\texttt{exc}_{\boldsymbol{a}}$ cannot be caught by F if $\boldsymbol{a}\#F$. This would no 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 $\texttt{exc}_{\boldsymbol{a}}$, 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 (2012). This is left for future work. Also, it is not clear whether we can add non-deterministic computations without breaking our continuity proof, because we have used the fact that Nuprl's computation system is currently deterministic, for example in Section 4.8.

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 Section 4. Investigating these rules is left for future work.

Acknowledgements

We would like to thank our colleagues Abhishek Anand, Robert L. Constable, Rich Eaton, Evan Moran and Ross Tate for their helpful criticism. This work was partially supported by the SnT and the National Research Fund Luxembourg (FNR), through PEARL grant FNR/P14/8149128.

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

References

Allen, S. (2006). An abstract semantics for atoms in Nuprl, Technical report TR2006-2032, Cornell University.

Allen, S.F. (1987a). A non-type-theoretic definition of Martin-Löf's types. In: *Proceedings of Second IEEE Symposium on Logic in Computer Science* 215–221.

Allen, S.F. (1987b). *A Non-Type-Theoretic Semantics for Type-Theoretic Language*, PhD thesis, Cornell University.

Allen, S.F., Bickford, M., Constable, R.L., Eaton, R., Kreitz, C., Lorigo, L. and Moran, E. (2006). Innovations in computational type theory using Nuprl. *Journal of Applied Logic* **4** (4) 428–469, http://www.nuprl.org/.

Anand, A., Bickford, M., Constable, R.L. and Rahli, V. (2014). *A Type Theory with Partial Equivalence Relations as Types*, Presented at TYPES 2014.

Anand, A. and Rahli, V. (2014a). *Towards a Formally Verified Proof Assistant*, ITP 2014, Klein, G. and Gamboa, R. (eds.) Lecture Notes in Computer Science, vol. 8558, Springer, 27–44.

Anand, A. and Rahli, V. (2014b). Towards a formally verified proof assistant. Technical report, Cornell University, http://www.nuprl.org/html/Nuprl2Coq/.

Avigad, J. (2004). Forcing in proof theory. *Bulletin of Symbolic Logic* **10** (3) 305–333.

Bauer, A. and Pretnar, M. (2015). Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming* **84** (1) 108–123.

Beeson, M.J. (1985). *Foundations of Constructive Mathematics*, Springer.

Berardi, S., Bezem, M. and Coquand, T. (1998). On the computational content of the axiom of choice. *Journal of Symbolic Logic* **63** (2) 600–622.

Berger, U. and Oliva, P. (2006). Modified bar recursion. *Mathematical Structures in Computer Science* **16** (2) 163–183.

Bertot, Y. and Casteran, P. (2004). *Interactive Theorem Proving and Program Development*, Springer-Verlag, http://www.labri.fr/perso/casteran/CoqArt.

Bickford, M. (2008). Unguessable atoms: A logical foundation for security. In: *Proceedings of the 2nd International Conference on Verified Software: Theories, Tools, Experiments*, Lecture Notes in Computer Science, vol. 5295, Springer, 30–53.

Bishop, E. and Bridges, D. (1985). *Constructive Analysis*, Springer.

Bojanczyk, M., Braud, L., Klin, B. and Lasota, (S. 2012). Towards nominal computation. In: *POPL'12*, ACM, 401–412.

Bove, A., Dybjer, P. and Norell, U. (2009). A brief overview of Agda - a functional language with dependent types. In: *TPHOLs 2009*, Lecture Notes in Computer Science, vol. 5674, Springer, pp. 73–78, http://wiki.portal.chalmers.se/agda/pmwiki.php.

Brady, E. (2011). Idris —: systems programming meets full dependent types. In: *PLPV 2011*, ACM, 43–54.

Bridges, D. and Richman, F. (1987). *Varieties of Constructive Mathematics*, London Mathematical Society Lecture Notes Series, Cambridge University Press.

Cheney, J. (2009). A simple nominal type theory. *Electronic Notes in Theoretical Computer Science* **228** 37–52.

Cheney, J. (2012). A dependent nominal type theory. *Logical Methods in Computer Science* **8** (1) 1–29.

Cheney, J. and Urban, C. (2004). Alpha-prolog: A logic programming language with names, binding and a-equivalence. In: *ICLP 2004*, Lecture Notes in Computer Science, vol. 3132, Springer, 269–283.

Cheney, J. and Urban, C. (2008). Nominal logic programming. *ACM Transactions on Programming Languages and Systems* **30** (5) Article No. 26. doi:10.1145/1387673.1387675.

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

Cohen, P.J. (1963). The independence of the continuum hypothesis. *The National Academy of Sciences of the United States of America* **50** (6) 1143–1148.

Cohen, P.J. (1964). The independence of the continuum hypothesis ii. *The National Academy of Sciences of the United States of America* **51** (1) 105–110.

Constable, R.L. (1983). Constructive mathematics as a programming logic I: Some principles of theory. *Fundamentals of Computation Theory*, Lecture Notes in Computer Science,, vol. 158, Springer, 64–77.

Constable, R.L., Allen, S.F., Bromley, H.M., Cleaveland, W.R., Cremer, J.F., Harper, R.W., Howe, D.J., Knoblock, T.B., Mendler, N.P., Panangaden, P., Sasaki, J.T. and Smith, S.F. (1986). *Implementing Mathematics with the Nuprl Proof Development System*, Prentice-Hall, Inc.

Constable, R. and Bickford, M. (2014). Intuitionistic completeness of first-order logic. *Annals of Pure and Applied Logic* **165** (1) 164–198.

Constable, R.L. and Hickey, J. (2000). Nuprl's class theory and its applications. Foundations of Secure Computation, NATO ASI Series, Series F: Computer & System Sciences, IOS Press, 91–116.

Constable, R.L. and Smith, S.F. (1993). Computational foundations of basic recursive function theory. *Theoretical Computer Science* **121** (1&2) 89–112.

Coquand, T. and Jaber, G. (2010). A note on forcing and type theory. *Fundamenta Informaticae* **100** (1–4) 43–52.

Coquand, T. and Jaber, G. (2012). A computational interpretation of forcing in type theory. In: *Epistemology versus Ontology*, Logic, Epistemology, and the Unity of Science, Springer, 203–213.

Coquand, T. and Paulin, C. (1988). Inductively defined types. In: *COLOG-88, International Conference on Computer Logic*, Lecture Notes in Computer Science, vol. 417, Springer, 50–66.

Crary, K. (1998). *Type-Theoretic Methodology for Practical Programming Languages*, PhD thesis, Cornell University.

Crole, R.L. and Nebel, F. (2013). Nominal lambda calculus: An internal language for fm-cartesian closed categories. *Electronic Notes in Theoretical Computer Science* **298** 93–117.

Danvy, O. and Filinski, A. (1990). Abstracting control. In: *LISP and Functional Programming*, 151–160.

David, R. and Mounier, G. (2005). An intuitionistic λ-calculus with exceptions. *Journal of Functional Programming* **15** (1) 33–52.

de Groote, P. (1995). A simple calculus of exception handling. In: *TLCA '95*, Lecture Notes in Computer Science, vol. 902, Springer, 201–215.

Dummett, M.A.E. (1977). *Elements of Intuitionism*, 2nd ed., Clarendon Press.

Escardó, M. and Oliva, P. (2015). Bar recursion and products of selection functions. *Journal of Symbolic Logic* **80** (1) 1–28.

Escardó, M.H. (2007). Infinite sets that admit fast exhaustive search. In: *Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)* 443–452.

Escardó, M.H. (2008). Exhaustible sets in higher-type computation. *Logical Methods in Computer Science* **4** (3) 1–37.

Escardó, M.H. (2013). Continuity of Gödel's system T definable functionals via effectful forcing. *Electronic Notes in Theoretical Computer Science* **298** 119–141.

Escardó, M.H. and Xu, C. (2015). The inconsistency of a Brouwerian continuity principle with the Curry-Howard interpretation. In: *TLCA 2015*, LIPIcs, vol. 38, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik., 153–164.

Fairweather, E., Fernández, M., Szasz, N. and Tasistro, A. (2015). Dependent types for nominal terms with atom substitutions. In: *TLCA 2015,* LIPIcs, vol. 38, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik., 180–195.

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

Felleisen, M. (1988). The theory and practice of first-class prompts. In: Ferrante, J. and Mager, P. (eds.) *POPL 1988*, ACM Press, 180–190.

Felleisen, M., Friedman, D.P., Kohlbecker, E.E. and Duba, B.F. (1986). Reasoning with continuations. In: *LICS '86*, IEEE Computer Society, 131–141.

Gabbay, M. and Pitts, A.M. (1999). A new approach to abstract syntax involving binders. In: *Proceedings of the 14th Symposium on Logic in Computer Science (LICS 1999)* 214–224.

Gielen, W., de Swart, H.C.M. and Veldman, W. (1981). The continuum hypothesis in intuitionism. *Journal of Symbolic Logic* **46** (1) 121–136.

Girard, J.-Y., Taylor, P. and Lafont, Y. (1989). *Proofs and Types*, Cambridge University Press.

Gordon, A.D. (1995). Bisimilarity as a theory of functional programming. *Electronic Notes in Theoretical Computer Science* **1** 232–252.

Gordon, M.J.C., Milner, R. and Wadsworth, C.P. (1979). *Edinburgh LCF: A Mechanised Logic of Computation*, Lecture Notes in Computer Science, vol. 78, Springer-Verlag.

Griffin, T. (1990). A formulae-as-types notion of control. In: Allen, F.E. (ed.) *POPL 1990*, ACM Press, 47–58.

Hickey, J.J. (2001). *The MetaPRL Logical Programming Environment*, PhD thesis, Cornell University.

Hofmann, M. (1995). *Extensional Concepts in Intensional Type Theory*, PhD thesis, University of Edinburgh.

Howard, W.A. and Kreisel, G. (1966). Transfinite induction and bar induction of types zero and one, and the role of continuity in intuitionistic analysis. *Journal of Symbolic Logic* **31** (3) 325–358.

Howe, D.J. (1989). Equality in lazy computation systems. In: *LICS 1989*, IEEE Computer Society, 198–203.

Howe, D.J. (1996). Semantic foundations for embedding HOL in Nuprl. In: Wirsing, M. and Nivat, M. (eds.)*Algebraic Methodology and Software Technology*, Lecture Notes in Computer Science, vol. 1101, Springer-Verlag, Berlin, 85–101.

Jeffrey, A. and Rathke, J. (1999). Towards a theory of bisimulation for local names. In: *Proceedings of the 14th Symposium on Logic in Computer Science (LICS 1999)* 56–66.

Kameyama, Y. (2001). Dynamic control operators in type theory. In: *APLAS'01*, 1–11.

Kameyama, Y. and Yonezawa, T. (2008). Typed dynamic control operators for delimited continuations. In: Garrigue, J. and Hermenegildo, M.V. (eds.) FLOPS 2008, Lecture Notes in Computer Science, vol. 4989, Springer, 239–254.

Kleene, S.C. and Vesley, R.E. (1965). *The Foundations of Intuitionistic Mathematics, Especially in Relation to Recursive Functions*, North-Holland Publishing Company.

Kopylov, A. (2004). *Type Theoretical Foundations for Data Structures, Classes, and Objects*, PhD thesis, Cornell University.

Kopylov, A. and Nogin, A. (2001). Markov's principle for propositional type theory. In: *CSL 2001*, Lecture Notes in Computer Science, vol. 2142, Springer, 570–584.

Krebbers, R. and Spitters, B. (2011). Type classes for efficient exact real arithmetic in Coq. *Logical Methods in Computer Science* **9** (1) 1–27.

Kreisel, (1962). On weak completeness of intuitionistic predicate logic. *The Journal of Symbolic Logic* **27** (2) 139–158.

Krivine, J.-L. (1993). *Lambda-Calculus, Types and Models*, Ellis Horwood Series in Computers and their Applications, Masson.

Lebresne, S. (2008). A system F with call-by-name exceptions. *ICALP 2008*, Lecture Notes in Computer Science, vol. 5126, Springer, 323–335.

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

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

Longley, J. (1999). When is a functional program not a functional program?. In: *ICFP'99*, ACM, 1–7.

Lösch, S. and Pitts, A.M. (2011). Relating two semantics of locally scoped names. In: *CSL 2011*, LIPIcs, vol. 12, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 396–411.

Martin-Löf, P. (1984). Intuitionistic type theory. In: *Studies in Proof Theory*, Lecture Notes, no. 1, Bibliopolis, Napoli.

Miquel, A. (2000). A model for impredicative type systems, universes, intersection types and subtyping. In: *LICS 2000*, IEEE Computer Society, 18–29.

Miquel, A. (2001a). The implicit calculus of constructions. In: *TLCA*, 344–359.

Miquel, A. (2001b). *Le Calcul des Constructions Implicites: Syntaxe et Sémantique*, PhD thesis, Université Paris 7.

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

Murthy, C.R. (1991). An evaluation semantics for classical proofs. In: *LICS '91*, IEEE Computer Society, 96–107.

Nakano, H. (1994). A constructive logic behind the catch and throw mechanism. *Annals of Pure and Applied Logic* **69** (2–3) 269–301.

Nogin, A. (2002). *Theory and Implementings of an Efficient Tactic-Based Logical Framework*, PhD thesis, Cornell University.

Nogin, A. and Kopylov, A. (2006). Formalizing type operations using the 'image' type constructor. *Electronic Notes in Theoretical Computer Science* **165** 121–132.

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

Odersky, M. (1994). A functional theory of local names. In: *POPL'94*, ACM Press, 48–59.

Parigot, M. (1992). $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. In: Voronkov, A. (ed.) *LPAR'92*, Lecture Notes in Computer Science, vol. 624, Springer, 190–201.

Pitts, A.M. (2001). Nominal logic: A first order theory of names and binding. In: *TACS 2001*, Lecture Notes in Computer Science, vol. 2215, Springer, 219–242.

Pitts, A.M. (2010). Nominal system T. In: *POPL'10*, ACM, 159–170.

Pitts, A.M. (2013). *Nominal Sets: Names and Symmetry in Computer Science*, Cambridge University Press.

Pitts, A.M. and Gabbay, M. (2000). A metalanguage for programming with bound names modulo renaming. In: *MPC 2000*, Lecture Notes in Computer Science, vol. 1837, Springer, 230–255.

Pitts, A.M., Matthiesen, J. and Derikx, J. (2015). A dependent type theory with abstractable names. Mackie, I. and Ayala-Rincon, M. (eds.) *Proceedings of the LSFA 2014 Workshop*, Electronic Notes in Theoretical Computer Science, vol. 312, Elsevier, 19–50.

Pitts, A.M. and Stark, I.D.B. (1993). Observable properties of higher order functions that dynamically create local names, or what's new?. In: *MFCS'93*, Lecture Notes in Computer Science, vol. 711, Springer, 122–141.

Pottier, F. (2007). Static name control for FreshML. In: *Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)* 356–365.

Pouillard, P. and Pottier, F. (2010). A fresh look at programming with names and binders. In: *ICFP 2010*, ACM, 217–228.

Rahli, V. and Bickford, M. (2015). Coq as a metatheory for Nuprl with bar induction. Presented at CCC 2015. Available at `http://www.nuprl.org/html/Nuprl2Coq/barind.pdf`.

Rahli, V. and Bickford, M. (2016). A nominal exploration of intuitionism. In: Avigad, J. and Chlipala, A. (eds.) *CPP 2016*, ACM, 130–141.

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

Rahli, V., Bickford, M. and Anand, A. (2013). Formal program optimization in Nuprl using computational equivalence and partial types. In: *ITP'13*, Lecture Notes in Computer Science, vol. 7998, Springer, 261–278.

Rahli, V., Bickford, M. and Constable, R.L. (2016). A Story of Bar Induction in Nuprl. Extended version available at http://www.nuprl.org/html/Nuprl2Coq/bar-induction-long.pdf.

Rahli, V., Bickford, M. and Constable, R.L. (2017). Bar induction: The good, the bad, and the ugly. In: *Abstract book of LICS 2017: 32nd Annual ACM/IEEE Symposium on Logic in Computer Science*. Open Access.(http://hdl.handle.net/10993/31268)

Rathjen, M. (2005). Constructive set theory and Brouwerian principles. *Journal of UCS* **11** (12) 2008–2033.

Rathjen, M. (2006). A note on bar induction in constructive set theory. *Mathematical Logic Quarterly* **52** (3) 253–258.

Sacchini, J.L. (2014). *Exceptions in Dependent Type Theory*, Presented at TYPES'14. Available at http://www.pps.univ-paris-diderot.fr/types2014/abstract-18.pdf.

Schöpp, U. (2006). *Names and Binding in Type Theory*, PhD thesis, University of Edinburgh.

Schöpp, U. and Stark, I. (2004). A dependent type theory with names and binding. In: *CSL 2004*, Lecture Notes in Computer Science, vol. 3210, Springer, 235–249.

Shinwell, M.R. (2006). Fresh O'Caml: Nominal abstract syntax for the masses. *Electronic Notes in Theoretical Computer Science* **148** (2) 53–77.

Shinwell, M.R., Pitts, A.M. and Gabbay, M.J. (2003). FreshML: Programming with binders made simple. *SIGPLAN Notices* **38** (9) 263–274.

Smith, S.F. (1989). *Partial Objects in Type Theory*, PhD thesis, Cornell University.

The Univalent Foundations Program (2013). *Homotopy Type Theory: Univalent Foundations of Mathematics*. Available at http://homotopytypetheory.org/book, Institute for Advanced Study.

Thielecke, H. (2000). On exceptions versus continuations in the presence of state. In: Smolka, G. (ed.) *ESOP 2000*, Lecture Notes in Computer Science, vol. 1782, Springer, 397–411.

Thielecke, H. (2001). Contrasting Exceptions and Continuations. Available at http://www.cs.bu.edu/fac/kfoury/CVS-Working-Files/CS520-Fall06/Handouts/contrasting-exceptions-and-continuations.pdf.

TLCA 2015 (2015). LIPIcs, vol. 38, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

Troelstra, A.S. (1973). *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, Springer.

Troelstra, A.S. (1975). Non-extensional equality. *Fundamenta Mathematicae* **82** (4) 307–322.

Troelstra, A.S. (1977a). Aspects of constructive mathematics. In: *Handbook of Mathematical Logic*, North-Holland Publishing Company, 973–1052.

Troelstra, A.S. (1977b). A note on non-extensional operations in connection with continuity and recursiveness. *Indagationes Mathematicae* **39** (5) 455–462.

Troelstra, A.S. and van Dalen, D. (1988). *Constructivism in Mathematics an Introduction*, Studies in Logic and the Foundations of Mathematics, vol. 121, Elsevier.

van Atten, M. and van Dalen, D. (2002). Arguments for the continuity principle. *Bulletin of Symbolic Logic* **8** (3) 329–347.

Veldman, W. (2001). Understanding and using Brouwer's continuity principle. In: *Reuniting the Antipodes — Constructive and Nonstandard Views of the Continuum*, Synthese Library, vol. 306, Springer Netherlands, 285–302 (English).

Veldman, W. (2006). Brouwer's real thesis on bars. *Philosophia Scientiæ* **CS6**, 21–42.

Veldman, W. (2014). Brouwer's fan theorem as an axiom and as a contrast to Kleene's alternative. *Archive for Mathematical Logic* **53** (5–6) 621–693.

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press

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

Westbrook, E.M. (2008). *Higher-order Encodings with Constructors*, PhD thesis, Washington University.

Westbrook, E.M., Stump, A. and Austin, E. (2009). The calculus of nominal inductive constructions: An intensional approach to encoding name-bindings. In: *LFMTP '09*, ACM, 74–83.

Xu, C. (2015). *A Continuous Computational Interpretation of Type Theories*, PhD thesis, University of Birmingham.

Xu, C. and Hötzel Escardó, M. (2013). A constructive model of uniform continuity. In: *TLCA 2013*, Lecture Notes in Computer Science, vol. 7941, Springer, 236–249.

https://doi.org/10.1017/S0960129517000172 Published online by Cambridge University Press