Math. Struct. in Comp. Science (2017), *vol.* 27, *pp.* 993–1031. © Cambridge University Press 2015
doi:10.1017/S0960129515000468 First published online 29 October 2015

Constructive forcing, CPS translations and witness extraction in Interactive realizability[†]

FEDERICO ASCHIERI

Institute of Discrete Mathematics and Geometry, Vienna University of Technology
Wiedner Haupstrasse 8-10 1040 Vienna, Austria
Email: federico.aschieri@tuwien.ac.at

Received 30 October 2014; revised 18 August 2015

In Interactive realizability for second-order Heyting Arithmetic with EM_1 and SK_1 (the excluded middle and Skolem axioms restricted to Σ^0_1-formulas), realizers are written in a classical version of Girard's System F. Since the usual reducibility semantics does not apply to such a system, we introduce a constructive forcing/reducibility semantics: though realizers are not computable functionals in the sense of Girard, they can be forced to be computable. We apply this semantics to show how to extract witnesses for realizable Π^0_2-formulas. In particular, a constructive and efficient method is introduced. It is based on a new '(state-extending-continuation)-passing-style translation' whose properties are described with the constructive forcing/reducibility semantics.

1. Introduction

The main goal of this work is to provide an efficient solution to the *witness extraction* problem in the theory of Interactive realizability, in particular for the systems developed in Aschieri (2013) and Aschieri and Berardi (2011). The techniques that we shall introduce to achieve our aim, however, are part of a much *larger program* (Miquel 2011), which is to understand the computational content of the forcing method in proof theory (Avigad 2004). We shall define a new notion of *constructive forcing*, as opposed to the notion of *classical forcing*, which was introduced in Cohen (1967) to prove the relative consistency of the negation of the Axiom of Choice and the Continuum Hypothesis, and since then largely employed as a powerful general tool in proof theory. Constructive forcing has a direct computational content which classical forcing lacks, while perfectly capable of replacing altogether classical forcing in the applications to classical logic we have in mind. Associated to constructive forcing there is a *new continuation-passing-style* (CPS) *translation* of the lambda terms used in Interactive realizability. The relationship between constructive forcing and our CPS translation is similar to the one between negative translations and ordinary CPS translations (Griffin 1990; Murthy 1990). In our case from a typing derivation of a term t, by the translation we get a term 'forcing t to be

[†] This work was supported by the LABEX MILYON (ANR-10-LABX-0070) of Université de Lyon, within the program 'Investissements d'Avenir' (ANR-11-IDEX-0007) operated by the French National Research Agency (ANR).

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

computable,' while in the latter case one obtains a term realizing the negative translation of the starting type.

1.1. *Classical forcing in proof theory*

Classical forcing was first presented by Cohen as a syntactical translation, mapping any formula of set theory A into the formula 'A is forced.' Only afterwards it was presented as a model construction technique, used to build objects not already present in some starting model of set theory. In the world of proof theory, of course, only syntax exists, and one in general is not even able to talk about models, especially when interpreting classical theories in constructive ones. Forcing has thus been established in proof theory as a syntactical translation, following the original approach of Cohen. In order to interpret a theory T_1 into a theory T_2, one must show that if T_1 proves A then T_2 proves 'A is forced.' It is possible to obtain a conservation result whenever, in T_2, A is equivalent to 'A is forced' (Avigad 2004).

A fascinating use of forcing consists in approximating infinite non-computational objects by means of finite ones. A simple but powerful example can be found in Avigad (2003), which is one of the works which inspired us the most in the effort to obtain the results of this paper. Let us consider classical Peano Arithmetic PA with Skolem function symbols $\Phi_0, \Phi_1, \ldots, \Phi_n, \ldots$ and Skolem axiom scheme SK:

$$\forall x \forall y.\ \psi(x, y) \to \psi(x, \Phi_n(x)),$$

which is an axiom of countable choice over individuals. If we denote the formula 'A is forced' with $\Vdash A$, then Avigad showed that

$$\mathsf{PA} + \mathsf{SK} \vdash A \implies \mathsf{PA} \vdash \Vdash A$$

and that for all arithmetical formulas (those not containing the Skolem function symbol Φ)

$$\mathsf{HA} \vdash A \leftrightarrow \Vdash A.$$

Thus, $\mathsf{PA} + \mathsf{SK}$ is conservative over PA for arithmetical formulas. The fact that this result has also an easy model theoretic proof should not lead to the conclusion that it is trivial. The semantical proof is non-constructive and if one wants to eliminate Skolem axioms from proofs, it is useless. Moreover, for a long time only very complicated effective translations had been known, while Avigad's is simple – but deep.

Finally, one can obtain a constructive proof for any $\forall\exists$-formula just by translating PA into HA by Gödel's double negation translation and Friedman's translation (see e.g. Kohlenbach (2008)). If we denote with $(_)^N$ this translation, we have

$$\mathsf{PA} + \mathsf{SK} \vdash A \implies \mathsf{HA} \vdash (\Vdash A)^N.$$

In the case A is a $\forall\exists$-formula, then $(\Vdash A)^N$ is equivalent to A, closing the circle. If one wants also a program out of the final constructive proof, it is sufficient to apply modified realizability (Kreisel 1959).

Summing up: one has to apply to the original proof in $\mathsf{PA}+\mathsf{SK}$ *four different translations* to extract some constructive content! Needless to say, it is very difficult to understand

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

the final program, and almost impossible to optimize it by hand. The trouble is, the forcing translation eliminates choice functions by approximating them, and the double negation translation eliminate the excluded middle by again approximating it in an implicit way. The resulting proof and program are thus *approximations* of *approximations*. In classical forcing, one has the impression of understanding how infinite objects are approximated, but the second approximation given by the negative translation essentially destroys the simplicity of the first one. For example, in forcing proofs, conditions and information always increase, while in the resulting programs, approximations not only lose their nature and are no more simple to describe, they also go through a process of construction and destruction. Technically, *monotonicity* is lost. These considerations still hold if we replace negative translation and intuitionistic realizability by Krivine's classical realizability (Krivine 2011; Miquel 2011). We can say without hesitation that final responsible for this situation is classical forcing: the results about it can only be proved *classically*.

1.2. *Witness extraction in Interactive realizability*

Classical forcing would yield a solution also to the witness extraction problem in Interactive realizability. That is, given a realizer $t \Vdash \forall x^{\mathtt{N}} \exists y^{\mathtt{N}} P x y$, with P atomic predicate, one wants to extract a non-trivial program taking as input any number n and yielding as output a number m such that $P n m$ holds. t, here, is a term belonging to a classical version of Girard's System F – which is essentially the original system augmented with a function symbol $\Phi : \mathtt{N} \to \mathtt{N} \to \mathtt{N}$, representing the Skolem functions $\Phi_0, \Phi_1, \ldots, \Phi_n, \ldots$ used in the logical language. One would like to use Girard's reducibility in order to reason about terms, and show among other things that the terms of type $\mathtt{N}$ denote natural numbers. Using SK, one can do that without problems. Then, it is possible to use the classical forcing translation and obtain a proof that every term is 'forced to be computable;' afterwards, one applies to that proof a negative translation and intuitionistic realizability. The resulting program would be an 'evaluator' of the original term t, capable of extracting a witness out of it. As clear from the previous discussion, this evaluator would work – but in a perfectly obscure and fairly unoptimized way.

1.3. *Constructive forcing*

What we need is a notion of *constructive forcing*. This notion should explain what is forcing in the real world, what is really an approximation, how far one can *actually* approximate infinite objects. None of that is explained by nested translations, so a new approach is really needed. The problem with classical forcing is that it is monotone: intuitively, if a goal of the construction is obtained in some state, one wants the goal to remain achieved without further work in all the future states. In other words, one has the aim of defining a winning construction strategy, that works *independently* of the 'context'. Technically, monotonicity in classical forcing is obtained by *unrestricted quantification over all futures states*, over all stronger conditions. This is desperately too much from the constructive point of view: usually, one is only able to formulate construction strategies

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

which are winning *relative* to the context. If we want to interpret forcing constructively, we must *restrict quantification* to the future states that *really matter* in computer science: the states in the *future of computation*.

Our new constructive forcing framework is a significant refinement and extension of the approach in Aschieri (2011c) and its correctness proof is completely intuitionistic. It is composed of two ingredients:

— First, we define a constructive forcing/computability semantics, explaining that although an interactive realizer is not computable, it can be *constructively forced to behave* like a computable functional. We call our notion of forcing *constructive forcing* as opposed to *classical forcing*. The difference between the two versions of forcing is that while classical forcing is a relation between *states* and *formulas*, constructive forcing is the intuitionistic realizability interpretation of a relation between *states* and *proofs* (actually, proof terms thanks to Curry–Howard); moreover, while in classical forcing quantification over future conditions is unrestricted, in constructive forcing the future that can be considered is only the future given by a *continuation of the computation*. The result is that the second has an intuitionistic meta-theory while that of the first is classical. This means that constructive forcing has direct computational content, but it is still capable of interpreting classical theories.

— Secondly, we define a new CPS translation, which, in particular, manipulates state-extending continuations. When applied to a realizer t, it produces a program that *forces* the computability of t. Moreover, that program is able to backtrack efficiently at the right points of computations and does not forget precious information when backtracking (a common defect among computational interpretations of classical logic, such as Berardi et al. (1998) and Krivine (2009)).

These new techniques have plenty of potential applications: from the epsilon substitution method (Mints et al. 1996) to update procedures (Avigad 2002; Aschieri 2011b), and basically everywhere forcing is or can be used in proof theory. Of course, there is still a long way to formulate a meaningful constructive forcing for set theory, where one may want to cover axioms such as the well ordering of real numbers or ultrafilter existence (Krivine 2011). With this work, we aim to describe how in our opinion the beginning of the path should look like.

1.4. *Plan of the paper*

In Section 2, we recall the term calculus in which interactive realizers are written, namely an extension of Girard's System F plus a constant symbol Φ for a Skolem function.
In Section 3, we recall the witness extraction problem in Interactive realizability.
In Section 4, we present the notion of constructive forcing.
In Section 5, we present a new CPS translation that manipulates states and continuations over them. We prove that every term of our calculus can be constructively forced to be computable by its CPS translation.
In Section 6, we briefly compare our approach with others.

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

2. The term calculus $\mathcal{F}_{\text{Class}}$

In this section, we introduce the typed lambda calculi that are used to define Interactive realizability in Aschieri (2013): system $\mathcal{F}$ and $\mathcal{F}_{\text{Class}}$. $\mathcal{F}$ is a completely standard extension of Girard's system F (see Girard (1989)) with some syntactic sugar: numerals, primitive recursion at all types, finite partial functions over $\mathbb{N}$ and simple primitive recursive operations over them. Equivalently, $\mathcal{F}$ may be seen as an extension of Gödel's system T (as one may find it in the exposition of Girard (1989)) with polymorphism. $\mathcal{F}_{\text{Class}}$ is obtained from $\mathcal{F}$ by adding on top of it a function symbol $\Phi : \mathtt{N} \to (\mathtt{N} \to \mathtt{N})$ intended to represent Skolem functions $\Phi_0, \Phi_1, \ldots, \Phi_n, \ldots$. Actually, as we shall see, it is of the same Turing degree of an oracle for the Halting problem. The symbol is totally inert from the computational point of view and so terms will always be computed with respect to some approximation of Φ.

With respect to Aschieri (2013), we use a Curry-style type system instead of a Church-style one and we leave outside product types and booleans, which can be easily coded in $\mathcal{F}$. While everything would have worked just fine even using the original system, the present approach spares us the painful Church notations needed to keep track of types within terms.

2.1. *Updates*

In order to define $\mathcal{F}$, we have first to define the concept of update, which is nothing but a finite partial function over $\mathbb{N}$. We use the appellative 'update,' because interactive realizers of atomic formulas return finite partial functions as new pieces of information that they have learned about Φ; updates represent new associations input–output that are intended to correct (and in this sense, *update*) wrong oracle values used in computations.

Definition 1 (Updates and consistent union).

1. An update set U, shortly an *update*, is a finite set of triples of natural numbers representing a finite partial function from $\mathbb{N}^2$ to $\mathbb{N}$.
2. Two triples (a, n, m) and (a', n', m') of numbers are *consistent* if $a = a'$ and $n = n'$ implies $m = m'$.
3. Two updates U_1, U_2 are consistent if $U_1 \cup U_2$ is an update.
4. $\mathbb{U}$ is the set of all updates.
5. The *consistent union* $U_1 \mathcal{U} U_2$ of $U_1, U_2 \in \mathbb{U}$ is $U_1 \cup U_2$ minus all triples of U_2 which are inconsistent with some triple of U_1.

We think of a triple (a, n, m) contained in an update as the code of a possible witness m for a Σ^0_1-formula $\exists y^{\mathtt{N}}.\mathsf{P}_a(n, y)$ (see Definition 2 below) . The fact that every update is a partial *function* allows in each update at most one witness for each formula $\exists y^{\mathtt{N}}.\mathsf{P}_a(n, y)$.

2.2. *The system* $\mathcal{F}$

System $\mathcal{F}$ is formally described in Figure 1. We shall write $t : T$, whenever the empty context types t with type T, that is, $\vdash t : T$. A numeral is a term of the form $\mathsf{S}(\mathsf{S}(\ldots 0))$.

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

Types

$$A, B ::= X \mid \mathtt{N} \mid \mathtt{U} \mid A \to B \mid \forall X A$$

Constants

$$c ::= \mathsf{R} \mid 0 \mid \mathsf{S} \mid \mathsf{mkupd} \mid \mathsf{is} \mid \mathsf{get} \mid \Cup \mid \overline{U} \text{ (for every } U \in \mathbb{U})$$

Terms

$$t, u ::= c \mid x \mid tu \mid \lambda x\, u$$

Typing Contexts

$$\Gamma = x_1 : A_1, \ldots, x_n : A_n \qquad x_i \neq x_j, \text{ for } i \neq j$$

Typing Rules for Variables and Constants

$$\Gamma, x : A \vdash x : A$$
$$\Gamma \vdash 0 : \mathtt{N}$$
$$\Gamma \vdash \mathsf{S} : \mathtt{N} \to \mathtt{N}$$
$$\Gamma \vdash \overline{U} : \mathtt{U} \text{ (for every } U \in \mathbb{U})$$
$$\Gamma \vdash \Cup : \mathtt{U} \to \mathtt{U} \to \mathtt{U}$$
$$\Gamma \vdash \mathsf{is} : \mathtt{U} \to \mathtt{N} \to \mathtt{N} \to \mathtt{Bool}$$
$$\Gamma \vdash \mathsf{get} : \mathtt{U} \to \mathtt{N} \to \mathtt{N} \to \mathtt{N}$$
$$\Gamma \vdash \mathsf{mkupd} : \mathtt{N} \to \mathtt{N} \to \mathtt{N} \to \mathtt{U}$$
$$\Gamma \vdash \mathsf{R} : \forall X.\ X \to (\mathtt{N} \to (X \to X)) \to \mathtt{N} \to X$$

Typing Rules for Composed Terms

$$\frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash tu : B} \qquad \frac{\Gamma, x : A \vdash u : B}{\Gamma \vdash \lambda x\, u : A \to B}$$

$$\frac{\Gamma \vdash u : \forall X A}{\Gamma \vdash u : A[B/X]} \qquad \frac{\Gamma \vdash u : A}{\Gamma \vdash u : \forall X A} \text{ with } X \text{ non free in the types of } \Gamma$$

Reduction Rules The usual reduction rule for lambda calculus $(\lambda x\, u)t \mapsto u[t/x]$ (see (Sorensen and Urzyczyn 2006)) plus the rules for recursion,

$$\mathsf{R}\, uv0 \mapsto u \qquad \mathsf{R}\, uv\mathsf{S}(t) \mapsto vt(\mathsf{R}\, uvt)$$

plus the following ones, assuming a, n, m be numerals:

$$\mathsf{is}\,\overline{U}\, a\, n \mapsto \begin{cases} \mathtt{True} & \text{if for some numeral } m,\ (a,n,m) \in U \\ \mathtt{False} & \text{otherwise} \end{cases}$$

$$\mathsf{get}\,\overline{U}\, a\, n \mapsto \begin{cases} m & \text{if for some numeral } m,\ (a,n,m) \in U \\ 0 & \text{otherwise} \end{cases}$$

$$\overline{U}_1 \Cup \overline{U}_2 \mapsto \overline{U_1 \mathcal{U} U_2}$$

$$\mathsf{mkupd}\, a\, n\, m \mapsto \overline{U}, \text{with } U = \{(a,n,m)\}$$

Fig. 1. System $\mathcal{F}$.

For every update $U \in \mathbb{U}$, there is in $\mathcal{F}$ a constant $\overline{U} : \mathtt{U}$, where $\mathtt{U}$ is a new base type representing $\mathbb{U}$. We write $\emptyset$ for $\overline{\emptyset}$. The usual $\mathtt{Bool}$ is treated as a defined typed (see Definition 3). In $\mathcal{F}$, there are four operations involving updates (see Figure 1):

1. The first operation is denoted by the constant $\mathsf{is} : \mathtt{U} \to \mathtt{N}^2 \to \mathtt{Bool}$. is takes as arguments an update constant $\overline{U}$ and two numerals a, n; it returns $\mathtt{True}$ if $(a,n,m) \in U$ for some

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

$m \in \mathtt{N}$ (that is, if the pair (a,n) is in the domain of the partial map U); it returns `False` otherwise.

2. The second operation is denoted by the constant $\mathsf{get} : \mathtt{U} \to \mathtt{N}^2 \to \mathtt{N}$. get takes as arguments an update constant $\overline{U}$ and two numerals a,n; it returns m if $(a,n,m) \in U$ for some $m \in \mathtt{N}$ (that is, if (a,n) belongs to the domain of the partial function U); it returns 0 otherwise.
3. The third operation is denoted by the constant $\mathsf{mkupd} : \mathtt{N}^3 \to \mathtt{U}$. mkupd takes as arguments three numerals a,n,m and transforms them into (the constant coding in $\mathcal{T}$) the update $\{(a,n,m)\}$.
4. The forth operation is denoted by the constant $\mathbb{U} : \mathtt{U}^2 \to \mathtt{U}$. $\mathbb{U}$ takes as arguments two update constants and returns the update constant denoting their consistent union.

We observe that the constants $\overline{U}, \mathsf{is}, \mathsf{get}, \mathsf{mkupd}$ and the type `U` are just syntactic sugar and, for computational purposes, they might be avoided by coding finite partial functions into natural numbers. However, it is essential to have the type `U` primitively encoded, because the definition of the forcing relation absolutely needs to distinguish numerals from update constants. For example, the identification of `N` and `U` made in Aschieri (2011c) leads to correct, but less efficient programs. Indeed, from the logical point of view, the type `U` corresponds to atomic predicates, while `N` to semantical objects, which are very different entities. Now, since the type `U` is required, at the end the constants $\overline{U}, \mathsf{is}, \mathsf{get}, \mathsf{mkupd}$ are needed as well.

Some terms of Gödel's system T will be used to represent predicates.

Definition 2 (Predicates of system T).

1. A binary *predicate* of T is any closed normal term $P : \mathtt{N}^2 \to \mathtt{Bool}$ of Gödel's system T.
2. We assume $\mathsf{P}_0, \mathsf{P}_1, \mathsf{P}_2, \ldots$ is an arbitrary enumeration of all binary predicates of T.

In the following, we shall need a good notation for typing derivations in $\mathcal{F}$. Had we used a Church-style system, this notation would have been provided by the terms themselves. But Curry-typed terms lose information about the structure of their own typing derivation. Indeed, the notation we are going to use for a derivation of a type judgment $\Gamma \vdash t : T$ is nothing but a Church-typed term corresponding to that derivation. So we define the concept of typing derivation of a typed judgment $\Gamma \vdash t : T$ by induction as follows:

1. c is a derivation of $\Gamma \vdash c : T$, for every constant c of type T.
2. x^A is a derivation of $\Gamma, x : A \vdash x : A$.
3. If $\mathbf{t}$ is a derivation of $\Gamma \vdash t : A \to B$ and $\mathbf{u}$ is a derivation of $\Gamma \vdash u$, then $\mathbf{tu}$ is a derivation of $\Gamma \vdash tu : B$.
4. If $\mathbf{u}$ is a derivation of $\Gamma, x : A \vdash u : B$, then $\lambda x^A \mathbf{u}$ is a derivation of $\Gamma \vdash \lambda x\, u : A \to B$.
5. If $\mathbf{u}$ is a derivation of $\Gamma \vdash u : A$, with the type variable X not occurring free in Γ, then $\Lambda X \mathbf{u}$ is a derivation of $\Gamma \vdash u : \forall XA$.
6. If $\mathbf{u}$ is a derivation of $\Gamma \vdash u : \forall XA$, then $\mathbf{u}B$ is a derivation of $\Gamma \vdash u : A[B/X]$.

As usual when working with polymorphic lambda calculus (Girard 1989), one can define product types $A \times B$. Then, for every pair of terms t_1 and t_2, there is a term $\langle t_1, t_2 \rangle$ such that if $\Gamma \vdash t_1 : A$ and $\Gamma \vdash t_1 : A$, then $\Gamma \vdash \langle t_1, t_2 \rangle : A \times B$. Moreover, for every term u such that $\Gamma \vdash u : A \times B$, one can define projections $\pi_0 u$ and $\pi_1 u$ such that $\Gamma \vdash \pi_0 u : A$ and

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

$\Gamma \vdash \pi_1 u : B$. Then, by using the equation of $\mathcal{F}$, one has $\pi_0\langle t_1, t_2\rangle = t_1$ and $\pi_1\langle t_1, t_2\rangle = t_2$. One can also define the type $\mathtt{Bool}$ and terms $\mathtt{True} : \mathtt{Bool}$ and $\mathtt{False} : \mathtt{Bool}$. Moreover, for all terms t, u_1, u_2 such that $\Gamma \vdash t : \mathtt{Bool}$, $\Gamma \vdash u_1 : T$ and $\Gamma \vdash u_2 : T$, one can define a term $\mathsf{if}\ t\ \mathsf{then}\ u_1\ \mathsf{else}\ u_2$ such that $\Gamma \vdash \mathsf{if}\ t\ \mathsf{then}\ u_1\ \mathsf{else}\ u_2 : T$. Then, by using the equation of $\mathcal{F}$, one has $\mathsf{if}\ \mathtt{True}\ \mathsf{then}\ u_1\ \mathsf{else}\ u_2 = u_1$ and $\mathsf{if}\ \mathtt{False}\ \mathsf{then}\ u_1\ \mathsf{else}\ u_2 = u_2$.

Definition 3 (Product types, boolean type).

1. For all types A, B of system $\mathcal{F}$, we define a type
$$A \times B := \forall X.(A \to B \to X) \to X,$$
where X is a variable not occurring free in A, B.

2. For every pair of terms t_1 and t_2 of $\mathcal{F}$, we define a term
$$\langle t_1, t_2\rangle := \lambda x.\, x\, t_1 t_2.$$

3. For every term u of system $\mathcal{F}$, we define two terms
$$\pi_0 u := u(\lambda x \lambda y.\, x) \qquad \pi_1 u := u(\lambda x \lambda y.\, y).$$

4. We define
$$\mathtt{Bool} := \forall X.\, X \to X \to X$$
$$\mathtt{True} := \lambda x \lambda y.\, x : \mathtt{Bool} \qquad \mathtt{False} := \lambda x \lambda y.\, y$$
$$\mathsf{if}\ t\ \mathsf{then}\ u_1\ \mathsf{else}\ u_2 := t u_1 u_2.$$

Notation. For notational convenience and to define in a more readable way terms of type $A \times B \to C$, given any term u we define
$$\lambda\langle x_0, x_1\rangle\, u := \lambda x\, u[\pi_0 x/x_0\ \pi_1 x/x_1],$$
where x is a fresh variable not appearing in u. We observe that for any terms t_0, t_1
$$(\lambda\langle x_0, x_1\rangle\, u)\langle t_0, t_1\rangle = u[t_0/x_0\ t_1/x_1]$$
and that if $\Gamma, x_0 : A, x_1 : B \vdash u : C$, then $\Gamma \vdash \lambda\langle x_0, x_1\rangle\, u : A \times B \to C$.

System $\mathcal{F}$ is obtained from system T adding polymorphism and new operations on atomic types. Of course, as proved in Aschieri (2013) as straightforward application of Girard's reducibility (Girard 1989), it enjoys strong normalization and has the uniqueness-of-normal-form property.

Theorem 1. $\mathcal{F}$ enjoys strong normalization and weak-Church-Rosser (uniqueness of normal forms) for all closed terms of atomic types.

The following normal form theorem also holds.

Lemma 2 (Normal form property for $\mathcal{F}$). Assume A is an atomic type. Then, any closed normal term $t \in \mathcal{F}$ of type A is either a numeral $n : \mathtt{N}$ or an update constant $\overline{U} : \mathtt{U}$.

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

From now onwards, for every pair of terms t_1, t_2 of system $\mathcal{F}$, we shall write $t_1 = t_2$ iff they are the same term modulo the equality rules corresponding to the reduction rules of system $\mathcal{F}$ (equivalently, if they have the same normal form).

2.3. *The system $\mathcal{F}_{Class}$*

We now define a classical extension of $\mathcal{F}$, that we call $\mathcal{F}_{\text{Class}}$, with a constant symbol $\Phi : \mathtt{N}^2 \to \mathtt{N}$ denoting a non-computable map of the same Turing degree of an oracle for the Halting problem. The terms of $\mathcal{F}_{\text{Class}}$ are used to represent non-computable realizers.

Definition 4 (Systems $\mathcal{F}_{\text{Class}}$ and $\mathcal{T}_{\text{Class}}$). Define $\mathcal{F}_{\text{Class}} = \mathcal{F} + \Phi$ and $\mathcal{T}_{\text{Class}} = \mathsf{T} + \Phi$, where $\Phi : \mathtt{N} \to \mathtt{N} \to \mathtt{N}$ is a new constant symbol and T is Gödel's system.

For every numeral a, Φa – which we shall denote with Φ_a – is intended to represent a *Skolem function* for the formula $\exists y^{\mathtt{N}}\, \mathsf{P}_a xy$, taking as argument a number x and returning some y such that $\mathsf{P}_a xy$ if any exists, and an arbitrary value otherwise. There is no set of computable reduction rules for the constant Φ, and therefore no set of computable reduction rules for $\mathcal{F}_{\text{Class}}$.

Each (in general, non-computable) term $t \in \mathcal{F}_{\text{Class}}$ is associated to a set $\{t[s] \mid s \in \mathcal{F}, s : \mathtt{N}^2 \to \mathtt{N}\} \subseteq \mathcal{F}$ of computable terms we call its 'approximations,' one for each closed term $s : \mathtt{N}^2 \to \mathtt{N}$ of $\mathcal{F}$, which is thought as a computable approximation of the oracle Φ.

Definition 5 (Approximation at state s). We define as follows:

1. A *state* is a closed term of type $\mathtt{N} \to \mathtt{N} \to \mathtt{N}$ of $\mathcal{F}$. We define $\mathtt{S} := \mathtt{N} \to \mathtt{N} \to \mathtt{N}$.
2. Assume $t \in \mathcal{F}_{\text{Class}}$ and s is a state. The 'approximation of t at state s' is the term $t[s]$ of $\mathcal{F}$ obtained from t by replacing each constant Φ with s.

One interprets any $t[s] \in \mathcal{F}$ as a learning process evaluated with respect to the information taken from an approximation s of Φ. Here, we consider an approximation of Φ to be an arbitrary term $s : \mathtt{N}^2 \to \mathtt{N}$; s may be correctly in agreement with Φ on some arguments, but wrong on other ones. Consequently, we are going to consider the set of (a, n) such that $\mathsf{P}_a n s_a(n) = \mathtt{True}$ as the real 'domain' of s (again, with $s_a(n)$ we shall sometimes denote san). We are also going to define a term $\oplus$, which takes as argument a term $f : \mathtt{N}^2 \to \mathtt{N}$ and an update $\overline{U}$, and changes the values of f according to U. This is one of the fundamental operations of our computational model: realizers compute updates to correct wrong values of oracle approximations with new good values that they have previously learned and stored in the updates.

Definition 6 (Domain, updates of functions).

1. If s is a state, we denote with $\mathsf{dom}(s)$ the set of pairs of numerals (a, n) such that $\mathsf{P}_a n s_a(n) = \mathtt{True}$. If U is an update, we denote with $\mathsf{dom}(U)$ the set of pairs of numerals (a, n) such that $(a, n, m) \in U$ and $\mathsf{P}_a nm = \mathtt{True}$; we say that U is *sound* if for every $(a, n, m) \in U$, we have $\mathsf{P}_a nm = \mathtt{True}$.
2. If s is a state and $\overline{U}$ is an update constant, we define

$$\overline{U} \preceq s \iff \mathsf{dom}(\overline{U}) \subseteq \mathsf{dom}(s).$$

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

3. We define a term $\oplus : \mathtt{S} \to \mathtt{U} \to \mathtt{S}$ as follows:

$$\oplus := \lambda s \lambda u \lambda x \lambda y \ \mathsf{if}\ (\mathsf{is}\, u\, x\, y)\ \mathsf{then}\ (\mathsf{get}\, u\, x\, y)\ \mathsf{else}\ sxy.$$

With an abuse of notation, we will write $t_1 \oplus t_2$ in place of $\oplus t_1 t_2$.

If s, r are states, we formalize what it means that r is at least as good an approximation as s by defining

$$s \leqslant r \iff \forall a, n.\ s_a(n) \neq r_a(n) \implies (a,n) \notin \mathsf{dom}(s) \wedge (a,n) \in \mathsf{dom}(r).$$

Intuitively, if $s \leqslant r$, then r can be obtained by changing some of the values of s that make s a wrong approximation of Φ.

Another possibility that might have seemed reasonable is to define r to be as good as s whenever for all a, n, it holds that $(a,n) \in \mathsf{dom}(s)$ implies $(a,n) \in \mathsf{dom}(r)$. However, such a definition is not compatible with the results we shall prove in this paper: Proposition 5 would just turn to be false, for $c = \Phi_i$, with no hope of fixing it.

3. Witness extraction with Interactive realizability

In this section, we recall the witness extraction problem for Π^0_2-formulas in Interactive realizability for $\mathsf{HAS} + \mathsf{EM}_1 + \mathsf{SK}_1$ (Aschieri 2013): this is the motivation for all of this work.

Given a term $t : \mathtt{N} \to (\mathtt{N} \times \mathtt{U})$ of $\mathcal{F}_{\text{Class}}$ realizing $\forall x^{\mathtt{N}} \exists y^{\mathtt{N}} P x y$, where P is an atomic recursive predicate, one is asked to extract from t a non-trivial program taking as input a numeral n and yielding as output a witness for the formula $\exists y^{\mathtt{N}} P n y$ (that is, a numeral m such that $P n m = \mathtt{True}$). In the case of Interactive realizability, the problem of computing that witness can be reduced to finding a 'zero' for a suitable term u of type $\mathtt{U}$, that is a state $s : \mathtt{S}$ such that $u[s] = \varnothing$. Indeed, given any numeral n and state s, the very definition of Interactive realizability requires the following implications to hold:

$$t \ \Vdash\vdash \ \forall x^{\mathtt{N}} \exists y^{\mathtt{N}} P x y,$$

$$\Longrightarrow$$

$$t \ \Vdash\vdash_s \ \forall x^{\mathtt{N}} \exists y^{\mathtt{N}} P x y,$$

$$\Longrightarrow$$

$$tn \ \Vdash\vdash_s \ \exists y^{\mathtt{N}} P n y,$$

$$\Longrightarrow$$

$$\pi_0(tn)[s] = m \wedge \pi_1(tn) \ \Vdash\vdash_s \ P n m,$$

$$\Longrightarrow$$

$$(1)\ \pi_1(tn)[s] = \varnothing \implies P n m = \mathtt{True},$$

$$(2)\ \pi_1(tn)[s] = \overline{U} \text{ implies that } U \text{ is sound and } \mathsf{dom}(U) \cap \mathsf{dom}(s) = \varnothing.$$

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

Therefore, if s is a zero of $\pi_1(tn)$, then $\pi_0(tn)$ is equal in the state s to some witness m of the formula $\exists y^{\mathbb{N}} P n y$. Intuitively, a zero for $\pi_1(tn)$ represents a sufficient amount of information to compute the required witness. A state which is not a zero, instead, represents an incorrect approximation of Φ and the update generated by $\pi_1(tn)$ represents a correction which has to be made. We remark that if $\pi_1(tn)[s] = \overline{U}$ and $\mathsf{dom}(U) \subseteq \mathsf{dom}(s)$, then $\overline{U} = \emptyset$. Such an s is called a prefixed point and it is therefore sufficient for witness extraction. In general:

Definition 7 (Prefixed points). Given a term $u : \mathtt{U}$ and a state s, we write $u[s] \preceq s$ if and only if $u[s] = \overline{U}$, for some update U, and $\mathsf{dom}(U) \subseteq \mathsf{dom}(s)$. If $u[s] \preceq s$, we call s a *prefixed point* of u.

In the rest of the paper, we will show how to compute efficiently a prefixed point for *any* closed term $v : \mathtt{U}$ of system $\mathcal{F}_{\text{Class}}$, no matter u is a realizer or not. We describe two methods for accomplishing this task; we conjecture they implement the very same idea of learning process, but they are radically different from the efficiency point of view. The *Iterative Method* is very simple, easy to understand and provides an algorithm that can be directly written in pure lambda calculus; but the proof of correctness uses the Axiom of Choice. The *State-Extending-Continuation-Passing-Style* (SECPS) *Method* is more sophisticated, dramatically more efficient and provides an algorithm that can be represented in System $\mathcal{F}$; the correctness proof is purely intuitionistic. But the first method is useful to understand the second, which is *nothing but an optimization* of the first.

3.1. *The Iterative Method*

If $t \Vdash \forall x^{\mathbb{N}} \exists y^{\mathbb{N}} P x y$, n is a numeral and $u := \pi_1(tn)$, the interpretation of $u : \mathtt{U}$ is that of a *state-extending operator*: given a state s, since $u[s]$ represents new information improving the approximation s of the oracle Φ, it is natural to associate to u the state-extending operator $\lambda s.\, s \oplus u[s]$. The idea of the Iterative Method (Aschieri 2013) is to start from an arbitrary state and apply this state-extending operator until a zero of u is reached. Such series of state extensions represents a terminating learning process.

Theorem 3 (Zero theorem). Let P be an atomic predicate of Gödel's T and suppose $t \Vdash \forall x^{\mathbb{N}} \exists y^{\mathbb{N}} P x y$. Let n be any numeral, define $u := \pi_1(tn)$ and let s be any state. Define, by induction on n, a sequence $\{s_n\}_{n \in \mathbb{N}}$ of states as follows:

$$s_0 := s,$$

$$s_{n+1} := s_n \oplus u[s_n] \overset{\text{def } 6}{=} \lambda x \lambda y \ \mathsf{if}\ (\mathsf{is}\, u[s_n]\, x\, y)\ \mathsf{then}\ (\mathsf{get}\, u[s_n]\, x\, y)\ \mathsf{else}\ s_n\, x\, y).$$

Then, there exists a number k such that $u[s_k] = \emptyset$.

Any state in the sequence $\{s_n\}_{n \in \mathbb{N}}$ defined in the statement of the Zero Theorem 3, can be produced by iterating a sufficient number of times the function $\lambda s.\, s \oplus u[s]$. Therefore, the eventual zero of u – which is going to be encountered during the generation of the sequence – can be computed directly by means of any fixed point combinator of pure lambda calculus. It is thus very easy to implement the Iterative Method in a purely

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

functional language – even in a classical Curry–Howard setting. No control operators, thus, let alone CPS translations.

Why then did the need for control operators and continuations ever arise in computational interpretations of classical logic? (Griffin 1990; Parigot 1992; Krivine 2009). The deep reason is that classical proofs are always interpreted as programs learning by trials and errors. The implementation of such learning processes usually needs *backtracking*: the ability to jump back when a mistake has been recognized and restart the computation at the point the mistake was first made. Since control operators are natural tools for implementing backtracking, it was inevitable that they entered the scene.

In our case, backtracking is implemented automatically in the iteration of the state-extending operator $\lambda s.\, s \oplus u[s]$ of Theorem 3. More precisely, let us consider again the chain $\{s_n\}_{n\in\mathbb{N}}$ defined in the statement of Theorem 3. When u is evaluated in s_n, and the result is different from $\varnothing$, then $s_{n+1} = s_n \oplus u[s_n] \neq s_n$. In particular, for some pair of numerals (i, j), $s_{n+1}(i, j) \neq s_n(i, j)$. The normalization of $u[s_{n+1}]$ is perfectly equal to the normalization of $u[s_n]$ up to the first point in which $s_{n+1}(i, j)$ is computed: this is the *backtracking point*. Instead of putting control operators in $u[s_n]$ to save all possible backtracking points and to jump directly to the right one (which is discovered by reducing $u[s_n]$ to an update), the Iterative Method recomputes $u[s_{n+1}]$ from scratch. In this way, the backtracking point is trivially rencountered, but with a great waste of resources.

For example, suppose that $s_n := \lambda x\lambda y\, y : \mathtt{S}$ is the identity state. Let us consider the following pseudo-program:

$$u := \mathsf{if}\ \Phi_0 1 + \Phi_0 2 + \Phi_0 3 + \Phi_0 4 = 10\ \mathsf{then}\ \{(0,4,1)\}\ \mathsf{else}\ \varnothing.$$

Then

$$u[s_n] = (\mathsf{if}\ 1 + 2 + 3 + 4 = 10\ \mathsf{then}\ \{(0,4,1)\}\ \mathsf{else}\ \varnothing) = \{(0,4,1)\}.$$

Now, how to evaluate u in the state $s_{n+1} := s_n \oplus \{(0,4,1)\}$? Since

$$u[s_{n+1}] = (\mathsf{if}\ \Phi_0 1[s_{n+1}] + \Phi_0 2[s_{n+1}] + \Phi_0 3[s_{n+1}] + \Phi_0 4[s_{n+1}] = 10\ \mathsf{then}\ \{(0,4,1)\}\ \mathsf{else}\ \varnothing),$$

it is inefficient to evaluate the whole term from scratch, because the only value of Φ which has been updated is $\Phi_0 4$: the terms $\Phi_0 1, \Phi_0 2, \Phi_0 3$ are all given by s_{n+1} the same value they had in s_n, that is, $1, 2, 3$ respectively. Rather, one should restart the evaluation from the backtracking point, which is an already partially evaluated term of the form

$$u[s_{n+1}] = \mathsf{if}\ 6 + \Phi_0 4[s_{n+1}] = 10\ \mathsf{then}\ \{(0,4,1)\}\ \mathsf{else}\ \varnothing.$$

That would spare us the evaluation and the addition of the first three terms of the sum, without exploiting the fact that this computation has already been done in the normalization of $u[s_n]$. While in this toy example we had a small sum, terms of system $\mathcal{F}_{\text{Class}}$ can take a very long time to be evaluated and avoiding to perform multiple times the same computation is mandatory.

In general, what we would like to do is to 'suspend' the normal flow of the normalization whenever evaluating some term $\Phi_n(m)$ in some state. First, we evaluate $\Phi_n(m)$ in s, but then we should look ahead in the computation and see whether an update (n, m, l) is ever produced: if it is the case, we assign immediately to $\Phi_n(m)$ the new value l, without

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

computing what was before. The aim of the next two sections is to define a CPS translation of terms $\mathcal{F}_{\text{Class}}$ into terms of $\mathcal{F}$, that allows to implement these ideas. Another possibility would have been to define an *abstract machine*, in analogy to what have been done in Krivine (2009), which would evaluate in a proper way the terms of $\mathcal{F}_{\text{Class}}$. We shall not follow this line, since we aim at the stronger result: being able to represent the programs extracted from classical proofs by purely functional terms of System $\mathcal{F}$.

4. Constructive forcing

The SECPS method defines a translation $[\![_]\!]$ of the terms of System $\mathcal{F}_{\text{Class}}$ into $\mathcal{F}$. $[\![t]\!]$ will be a program that manipulates continuations on states, and uses them to implement backtracking. There are two ways of interpreting $[\![t]\!]$:

1. syntactically: it is an efficient evaluator of the expression $t : \mathtt{U}$, a refined way to compute its intended denotation, that is, its value in a prefixed point of t itself. Using a computer science language, $[\![t]\!]$ *is not significantly different from* t as algorithm: it is just a *compiled* version of t that can be executed more efficiently.
2. semantically: it is the denotation of t in a non-standard model of $\mathcal{F}_{\text{Class}}$, in which terms of type $\mathtt{N}$ are interpreted as non-standard natural numbers (see also Aschieri (2011c)).

It is important to stress that while $[\![t]\!]$ is considerably more complicated than the original t, we should not worry: it is not difficult to understand its final behaviour, because there is *nothing* new to understand. Like in real world, we use a high-level language – System $\mathcal{F}_{\text{Class}}$ – for writing programs, and use a low-level 'machine' language (System $\mathcal{F}$) for executing them. The translation adds nothing, it is automatic and at the end one can ignore its existence: it is there just for efficiency.

Moreover, as opposed to the standard CPS-translations, ours has an associated *constructive forcing* semantics, which allows us to describe and prove exactly what the translation is supposed to do – an elementary requirement which is often not satisfied in the world of syntactical translations! All the tools that we are going to introduce will have the aim of defining a model of computation for a certain non-standard arithmetic, which amounts to finding non-standard counterparts of standard operations. Namely, the main question will be the following: how to define recursion over non-standard integers?

We should look to the evaluation of a term $t : \mathtt{U}$ of $\mathcal{F}_{\text{Class}}$ as depending on a state, starting from an arbitrary one. The state is to be interpreted as *global*, as not static, but evolving during evaluation and floating around the term: as in ordinary programming languages with variable assignments stored by environments. In standard programming languages, one does not care if the state changes during computations, and in particular if a variable is evaluated differently at two different times: one *wants* that. On the contrary, here we must be very careful. The state is supposed to represent a *single* Skolem function Φ, so there is an important

— *consistency condition*: different occurrences of a term $\Phi_n(m)$ inside a term must be evaluated the same, even if the global state changes.

That being fixed as a firm principle, the idea of the SECPS method, as we said, is to interpret the oracle Φ_a as a kind of control operator. During computations, whenever the

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

value of $\Phi_a(n)$ is asked, one uses the approximation $s_a(n)$ given by the current state s. But if the final value of a computation is an update U, it might contain a triple (a,n,m): the value of s at point (a,n) was incorrect and must be corrected with the value m. The idea is that the program interpreting $\Phi_a(n)$ should look at the future of the computation: if any state $s' > s$ is ever encountered such that s' incorporates some correction (a,n,m) made to s, then it returns m, otherwise it returns $s_a(n)$. The future of the computation, as usual, is given by a state-extending continuation.

Definition 8 (State-extending continuations). A closed term $k : \mathtt{S} \to \mathtt{S}$ of $\mathcal{F}$ is said to be a *state-extending continuation* if for all states s, $s \leqslant k(s)$. We denote with $\mathbb{K}$ the set of all state-extending continuations.

There is, however, a complication. The idea explained above works with terms of the form $\Phi_a(n)$, where n is a numeral, but not with terms of the form $\Phi_a(t)$, when $t : \mathtt{N}$ is a term of $\mathcal{F}_{\text{Class}}$. Assume that the current state is s. The current continuation applied to s tells us how s, after having evaluated $\Phi_a(t)$, evolves to a final state s' in the future of the computation. The consistency condition requires the evaluation of $\Phi_a(t)$ to give the same result in all the states encountered towards the final state s'. A problem thus arises when computing $t[s] = n$. To make sure that t is not going to change value, one has to evaluate once again t in the state s'. If $t[s'] = n$ and $s_a(n) = s'_a(n) = m$, the program interpreting $\Phi_a(t)$ should return m. But it may happen that $t[s'] = n' \neq n$. In this case, the program interpreting $\Phi_a(t)$ must look again in the future $s'' = k(s')$ evaluate $t[s''] = n''$ and so on... How do we know, *constructively*, that one cannot end up stuck in an infinite loop? The key idea is that, starting from any state, one can *force* t to have a value which is going to be stable in all the future states encountered in the continuation of the computation. To express this precisely, we need the notion of *modulus of forcing*.

4.1. *Moduli of forcing and constructive forcing at type* N

Classically, a term $t : \mathtt{N}$ of System $\mathcal{F}_{\text{Class}}$ denotes a natural number, as soon as Skolem function is chosen for interpreting Φ. But from the constructive point of view, t is not a natural number, for we cannot compute its value. Worse of all, t may have infinite values, one for each state! More precisely, in the Interactive realizability framework, there is no way whatsoever even in the classical sense to associate to t a definite natural number as a value. Since we consider as states all possible computable approximations of all possible Skolem functions that may interpret Φ, different states may approximate different functions. Constructively, however, we can *force* t *to behave like a single natural number* for 'the rest of computation.' In other words, there is an extension s' of the current state s such that t will have the same values in all the states of the computation produced after s'. In particular, a state s *forces* t *to belong to* $\mathbb{N}$ if for all $k \in \mathbb{K}$ there exists a state $s' \geqslant s$ such that t is equal to the same numeral when evaluated in any state r such that $s' \leqslant r \leqslant k(s')$. Intuitively, a modulus of forcing represents the computational content of that notion of forcing. The concept of modulus of forcing is an adaptation of a notion of constructive convergence due to S. Berardi, which has been extensively motivated and exploited in Aschieri (2011c) in order to define a translation of System $\mathcal{T}_{\text{Class}}$ into Gödel's

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

System $\mathcal{T}$. It can be seen as a no-counterexample interpretation of a classical notion of forcing, as we shall show in Section 6.3.

Definition 9 (Modulus of forcing and constructive forcing at type $\mathtt{N}$). Let $t : \mathtt{N}$ be a closed term of $\mathcal{F}_{\text{Class}}$. For all states s, r, we define

$$t \downarrow [s,r] \stackrel{\text{def}}{\equiv} \forall q^{\mathtt{S}}.\ s \leqslant q \leqslant r \to t[q] = t[s].$$

A closed term $\mathcal{M} : (\mathtt{S} \to \mathtt{S}) \to (\mathtt{S} \to \mathtt{S})$ of $\mathcal{F}$ is a *modulus of forcing* for t if:

1. $\forall k \in \mathbb{K}.\ \mathcal{M}_k \in \mathbb{K}$,
2. $\forall k \in \mathbb{K}.\ \forall s^{\mathtt{S}}.\, t \downarrow [\mathcal{M}_k(s), k(\mathcal{M}_k(s))]$.

Whenever $\mathcal{M}$ is a modulus of forcing for t, we write $\mathcal{M} \Vdash t \in \mathtt{N}$ and we say that $\mathcal{M}$ *forces* t *to belong to* $\mathtt{N}$.

Similarly to Aschieri (2011c), we are going to show that given two terms t_1 and t_2, if each one of them has a modulus of forcing, then there is a modulus of forcing that works simultaneously for both of them. In particular, we can define a binary operation $\sqcup$ between moduli of forcing such that, for every pair of moduli $\mathcal{M}, \mathcal{N}$, $\mathcal{M} \sqcup \mathcal{N}$ is 'more general' than both $\mathcal{M}$ and $\mathcal{N}$. Here, for every $\mathcal{M}_1, \mathcal{M}_2$, we call $\mathcal{M}_2$ more general than $\mathcal{M}_1$, if for every term t, if $\mathcal{M}_1$ is a modulus of forcing for t then also $\mathcal{M}_2$ is a modulus of forcing for t. We this terminology, we may see $\mathcal{M} \sqcup \mathcal{N}$ as an upper bound of the set $\{\mathcal{M}, \mathcal{N}\}$, with respect to the partial order induced by the relation 'to be more general than.'

We now want to illustrate how to arrive at the definition of $\mathcal{M} \sqcup \mathcal{N}$, where $\mathcal{M} \Vdash t_1 \in \mathtt{N}$ and $\mathcal{N} \Vdash t_2 \in \mathtt{N}$. To achieve the goal, we must reason in continuation-style. First, for any terms $u, v : \mathtt{S} \to \mathtt{S}$, define as usual their composition $u \circ v := \lambda s.\, u(vs)$. We observe that $\mathbb{K}$ is closed by composition: $u, v \in \mathbb{K} \implies u \circ v \in \mathbb{K}$. Now fix any $k \in \mathbb{K}$ and state s. The global task to be performed is to extend s to a state s', apply the continuation k to s' and verify $t_1, t_2 \downarrow [s', k(s')]$. Let us think sequentially: suppose we have already extended s to make t_1 constant in the desired interval. What are we left to do? Well, we have to extend again the state in order to the same thing for t_2: this is done by the term/continuation $\mathcal{N}_k : \mathtt{S} \to \mathtt{S}$; afterwards, we have to apply k; so, we are left with the continuation $k \circ \mathcal{N}_k$. So the first extension of the state s needed to make constant t_1 is given by $\mathcal{M}_{k \circ \mathcal{N}_k}$, because t_1 has to be stable during all of the state extensions needed to make stable t_2 and then made by k; then one applies $\mathcal{N}_k$ to the result, which makes t_2 constant. Therefore:

Proposition 1 (Joint forcing). Suppose $\mathcal{M} \Vdash t_1 \in \mathtt{N}$ and $\mathcal{N} \Vdash t_2 \in \mathtt{N}$. Let $k : \mathtt{S} \to \mathtt{S}$ and $s : \mathtt{S}$ be variables whose free occurrences are to be typed with the shown types. Define

$$\mathcal{M} \sqcup \mathcal{N} := \lambda k\, \lambda s.\, \mathcal{N}_k(\mathcal{M}_{k \circ \mathcal{N}_k}(s)).$$

Then, $\mathcal{M} \sqcup \mathcal{N} \Vdash t_1 \in \mathtt{N}$ and $\mathcal{M} \sqcup \mathcal{N} \Vdash t_2 \in \mathtt{N}$.

Proof. Set

$$\mathcal{L} := \mathcal{M} \sqcup \mathcal{N}.$$

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

We have

$$k : \mathtt{S} \to \mathtt{S}, s : \mathtt{S} \vdash \mathcal{N}_k(\mathcal{M}_{k\circ\mathcal{N}_k}(s)) : \mathtt{S} \to \mathtt{S},$$

so $\mathcal{L} : (\mathtt{S} \to \mathtt{S}) \to (\mathtt{S} \to \mathtt{S})$, as required. First, we check property 1 of Definition 9. For all $k \in \mathbb{K}$, we have $\mathcal{N}_k \in \mathbb{K}$ by Definition 9 point 1 and so $k \circ \mathcal{N}_k \in \mathbb{K}$; since also $\mathcal{M}$ has property 1 of Definition 9, we have $\mathcal{M}_{k\circ\mathcal{N}_k} \in \mathbb{K}$. Thus, for all $k \in \mathbb{K}$ and state s

$$\mathcal{L}_k(s) = \mathcal{N}_k(\mathcal{M}_{k\circ\mathcal{N}_k}(s)) \geqslant s.$$

Therefore, for all $k \in \mathbb{K}$, $\mathcal{L}_k \in \mathbb{K}$ and we are done.

Secondly, we check property 2 of Definition 9. Fix a continuation $k \in \mathbb{K}$ and a state s. Since $\mathcal{M} \Vdash t_1 \in \mathtt{N}$, we have that

$$t_1 \downarrow [\mathcal{M}_{k\circ\mathcal{N}_k}(s), k \circ \mathcal{N}_k(\mathcal{M}_{k\circ\mathcal{N}_k}(s))]. \tag{1}$$

Moreover, since $\mathcal{N} \Vdash t_2 \in \mathtt{N}$, we have

$$t_2 \downarrow [\mathcal{N}_k(\mathcal{M}_{k\circ\mathcal{N}_k}(s)), k(\mathcal{N}_k(\mathcal{M}_{h\circ\mathcal{N}_k}(s)))]. \tag{2}$$

But the starting point of the interval in (2) is greater or equal to the starting point of the interval in (1), for $\mathcal{N}_k \in \mathbb{K}$, while their ending points are equal. Hence also

$$t_1 \downarrow [\mathcal{N}_k(\mathcal{M}_{k\circ\mathcal{N}_k}(s)), k(\mathcal{N}_k(\mathcal{M}_{k\circ\mathcal{N}_k}(s)))]$$

and hence both t_1 and t_2 are constant in the interval $[\mathcal{L}_k(s), k(\mathcal{L}_k(s))]$ by definition of $\mathcal{L}$. □

Interestingly, the construction of $\mathcal{M} \sqcup \mathcal{N}$ can be reinterpreted as yet another incarnation of the product of selection functions(Escardo and Oliva 2009).†

The notion of modulus of forcing, similarly to what happens in Aschieri (2011c), gives rise to an interesting non-standard model of Gödel's System $\mathcal{T}$. In this model, integers are pairs $(\mathcal{M}, f)$, where f is a function from states to numbers and $\mathcal{M}$ is a modulus of forcing for that function. On one hand, functions are elected to interpret numbers because, *constructively*, terms $t : \mathtt{N}$ of $\mathcal{F}_{\text{Class}}$ should not be regarded as single numbers, but as *collections* of numbers, one for each state. On the other hand, functions are paired with moduli of forcing, which help them to assume the specific form of a *single* natural number, whenever needed in the computation. It is thus reasonable to associate to every term of $\mathcal{F}_{\text{Class}}$ the non-standard number it represents.

† We wish to thank one reviewer for this observation. Although the reinterpretation is slightly involved and plays no role in the following, it is worth sketching it. First of all, we observe that $\mathcal{M}, \mathcal{N}$ can be recoded as selection functions $\epsilon_1, \epsilon_2 : (X \to R) \to X$. It is enough to choose $X := \mathtt{S}$ and $R := \mathtt{S} \times (\mathtt{S} \times \mathtt{S})$, with the aim of viewing an element h of type $X \to R$ as coding the first two inputs of the moduli of forcing $\mathcal{M}, \mathcal{N}$: $(\lambda s\ \pi_1\pi_1(hs))$ represents a continuation of type $\mathtt{S} \to \mathtt{S}$, $\pi_0 h(\lambda x\ 0)$ represents the input state of $\mathcal{M}$ and $\pi_0\pi_1(h(\lambda x\ 0))$ the input state of $\mathcal{N}$. Then, we can then define

$$\epsilon_1 := \lambda h\ \mathcal{M}(\lambda s\ \pi_1\pi_1(hs))\pi_0(h(\lambda x\ 0)),$$

$$\epsilon_2 := \lambda h\ \mathcal{N}(\lambda s\ \pi_1\pi_1(hs))\pi_0\pi_1(h(\lambda x\ 0)),$$

Using the product $\epsilon_1 \otimes \epsilon_2 : ((X \times X) \times R) \to (X \times X)$, one can see that $\mathcal{M} \sqcup \mathcal{N}$ is equivalent to:

$$\lambda k\ \lambda s\ \pi_1((\epsilon_1 \otimes \epsilon_2)(\lambda r\ (s, (\pi_0 r, k\pi_1 r)))).$$

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

Definition 10 (Non-standard natural numbers). We define

$$^*\mathtt{N} := \{\langle \mathcal{L}, t\rangle \mid \mathcal{L} = \langle \mathcal{M}, g\rangle, g : \mathtt{S} \to \mathtt{N} \in \mathcal{F}, \forall s^{\mathtt{S}}\, g(s) = t[s] \text{ and } \mathcal{M} \Vdash t \in \mathtt{N}\}$$

(all terms in the definition are supposed to be closed).

The notion of modulus of forcing, if extended to terms of type U, is sufficient to write down in $\mathcal{F}$ a program that implements the Iterative Method for finding zeros. Indeed, that is the approach followed in Aschieri (2011c). But in order to perform better, we need the additional notion of *modulus of prefixed point*.

4.2. *Moduli of prefixed point and constructive forcing at type* U

The concept modulus of prefixed point is a concrete example of a notion of 'atomic interactive realizability' introduced in Berardi and de' Liguoro (2008). Interestingly, as well as the notion of modulus of forcing alone is not sufficient to write an efficient optimization of the Iterative Method, the notion of modulus of prefixed point is not enough either. But when combined together, the two notions are able to produce the desired result.

Intuitively, a modulus of prefixed point represents the computational content of the following constructive notion of forcing for terms t of type U: a state s *forces* t to belong to U if for all $k \in \mathbb{K}$ there exists $s' \geqslant s$ such that $k(s')$ is a prefixed point of t. In other words, there is an extension s' of s such that the final state of the computation that starts from s' will be a prefixed point of t. The idea is that the interpretation of the type U is the set of all terms of type U having 'stable' prefixed points reachable from any state.

Definition 11 (Modulus of prefixed point and constructive forcing at type U**).** A term $\mathcal{M} : (\mathtt{S} \to \mathtt{S}) \to (\mathtt{S} \to \mathtt{S})$ of $\mathcal{F}$ is a *modulus of prefixed point* for t if:

1. $\forall k \in \mathbb{K}.\ \mathcal{M}_k \in \mathbb{K}$,
2. $\forall k \in \mathbb{K}\ \ \forall s^{\mathtt{S}}.\ s' = k(\mathcal{M}_k(s)) \implies t[s'] \preceq s'$.

Whenever $\mathcal{M}$ is a modulus of prefixed point for t, we write $\mathcal{M} \Vdash t \in \mathtt{U}$ and we say that $\mathcal{M}$ *forces* t *to belong to* U.

The same construction of Proposition 1 allows to build a single modulus of prefixed point for two distinct terms, provided a modulus is given for each of the two terms.

Proposition 2 (Joint forcing at type U**).** Suppose $\mathcal{M} \Vdash t_1 \in \mathtt{U}$ and $\mathcal{N} \Vdash t_2 \in \mathtt{U}$. Then, $\mathcal{M} \sqcup \mathcal{N} \Vdash t_1 \in \mathtt{U}$ and $\mathcal{M} \sqcup \mathcal{N} \Vdash t_2 \in \mathtt{U}$.

Proof. By definition,

$$\mathcal{M} \sqcup \mathcal{N} := \lambda k\, \lambda s.\, \mathcal{N}_k(\mathcal{M}_{k \circ \mathcal{N}_k}(s)).$$

Let $k \in \mathbb{K}$ and s be a state. We must show that

$$s' = k((\mathcal{M} \sqcup \mathcal{N})ks) \implies t_1[s'] \preceq s' \wedge t_2[s'] \preceq s'.$$

By definition of s' and $\sqcup$, we have that

$$s' = k(\mathcal{N}_k(\mathcal{M}_{k \circ \mathcal{N}_k}(s))).$$

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

By definition of $\mathcal{M} \Vdash t_1 \in \mathtt{U}$, we get $t_1[s'] \preceq s'$. By definition of $\mathcal{N} \Vdash t_2 \in \mathtt{U}$, we get $t_2[s'] \preceq s'$. □

It is natural to associate to every term of type $\mathtt{U}$ all its possible moduli of prefixed point.

Definition 12 (Non-standard updates). We define

$$^*\mathtt{U} := \{\langle \mathcal{M}, t\rangle \mid \mathcal{M} \Vdash t \in \mathtt{U}\}$$

(all terms in the definition are supposed to be closed).

4.3. *Constructive forcing at all types*

Now that we have a notion of forcing at base types, we lift it to all types. Our goal is to define a forcing relation $\mathcal{M} \Vdash t \downarrow T$, whose intuitive meaning should be: $\mathcal{M}$ *forces the term* $t \in \mathcal{F}_{\text{Class}}$ *to behave like a computable functional of higher-type* T. At the same time, one can read the forcing relation model-theoretically: $\mathcal{M}$ is the *denotation of* t *in a non-standard model*. Since we are in an impredicative setting, we first have to determine some general conditions that our notion of forcing must satisfy.

As usual, we define a notion of candidate, which in our case is a set of pair of terms closed by the intensional equality of $\mathcal{F}$ and equipped with a (partial) monad operation $\mathcal{N}$ (in the sense of Moggi (1991)). A candidate C of type $U \times V$ represents a 'possible' forcing relation. Consequently, the first element of each pair in C is a term of type U which is to be seen as a possible modulus of forcing, while the second element is a term of type V which should be forced by the modulus to be computable at type V. Intuitively, the crucial property that we have to impose is the following: if t of $\mathcal{F}_{\text{Class}}$ is forced to belong to $\mathtt{N}$, and for every numeral m, $u(m)t_1 \dots t_n$ belongs to some pair of C, then also $u(t)t_1 \dots t_n$ belongs to some pair of C. Since we have to be effective, we ask the monad operation $\mathcal{N}$ to be a realizer of that property: given (i) a term $\mathcal{L}$ such that for every numeral m, $\langle \mathcal{L}m, u(m)t_1 \dots t_n\rangle \in C$; (ii) a representation of t as a non-standard number $(\mathcal{M}, g)$; then $(\mathcal{N}\mathcal{M}g\mathcal{L}, u(t)t_1 \dots t_n)$ must be in C.

The candidate requirement is a sort of 'induction' principle: if one can force something for all natural number parameters, then one can force it for all non-standard natural numbers. Such a property will be essential when interpreting recursion over non-standard numbers.

Definition 13 (Forcing candidates). Let C be a set of closed $\mathcal{F}_{\text{Class}}$-terms of closed type $U \times V$, closed under the intensional equality of $\mathcal{F}$ (i.e. if $t \in C$ and $t = t'$, then $t' \in C$). Let

$$\mathcal{N} : \mathsf{Cand}_U := ((\mathtt{S} \to \mathtt{S}) \to (\mathtt{S} \to \mathtt{S})) \to (\mathtt{S} \to \mathtt{N}) \to (\mathtt{N} \to U) \to U$$

be a closed term of system $\mathcal{F}$. We define the relation $\mathcal{N} \Vdash C \in \mathsf{Cand}_{U\times V}$ – representing the notion '$\mathcal{N}$ realizes that C is a *forcing candidate* of type $U \times V$" – to hold if for all terms $\mathcal{M}, t, g, \mathcal{L}$, the following three conditions:

1. $\mathcal{M} \Vdash t \in \mathtt{N}$;
2. for all states s, $g(s) = t[s]$;

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

3. $\mathcal{L} : \mathtt{N} \to U$ is a term of $\mathcal{F}$ such that for every numeral m, $\langle \mathcal{L}m, u(m)t_1 \dots t_n\rangle \in C$ together imply that
4. $\langle \mathcal{N}\mathcal{M}g\mathcal{L}, u(t)t_1 \dots t_n\rangle \in C$,

(where $t_1 \dots t_n$ is any sequence of terms). If $\mathcal{N} \Vdash C \in \mathsf{Cand}_{U\times V}$, then C is said to be a *forcing candidate* (of type $U \times V$). Whenever the type of the terms in C is known or not relevant, we shall just write $\mathcal{N} \Vdash C \in \mathsf{Cand}$.

For convenience, we now introduce to the language a type constant symbol for every forcing candidate. For each type T in this extended language, we define two types of $\mathcal{F}$: a type $|T|$, which represents the type in $\mathcal{F}$ of a term of type T in the extended language, and a type $[\![T]\!]$, which represents the interpretation of T in the forcing model.

Definition 14 (Extended types, interpretation of types).

1. For every forcing candidate C of type $U \times V$, we introduce a type constant $\mathcal{C}$ to the language of types of system $\mathcal{F}_{\text{Class}}$; we define $\dot{\mathcal{C}} = C$ and $|\mathcal{C}| := V$. An *extended type* is defined by induction as either a type constant $\mathcal{C}$, or a type variable, or an expression $U \to V$ or $\forall X\, U$, with U and V extended types and X a type variable. We define $|\forall X\, U| := \forall X\, |U|$, $|U \to V| := |U| \to |V|$, $|\mathtt{N}| = \mathtt{N}$, $|\mathtt{U}| = \mathtt{U}$, $|X| := X$, for X variable.
2. For every extended type T, we define a type $[\![T]\!]$ by induction on T as follows.
   a. $T = X$, with X atomic. Then,
   $$[\![X]\!] := X.$$
   b. $T = \mathtt{N}$. Then,
   $$[\![\mathtt{N}]\!] := ((\mathtt{S} \to \mathtt{S}) \to (\mathtt{S} \to \mathtt{S})) \times (\mathtt{S} \to \mathtt{N}).$$
   c. $T = \mathtt{U}$. Then,
   $$[\![\mathtt{U}]\!] := (\mathtt{S} \to \mathtt{S}) \to (\mathtt{S} \to \mathtt{S}).$$
   d. $T = \mathcal{C}$, where $\dot{\mathcal{C}}$ is a forcing candidate of type $U \times V$. Then,
   $$[\![\mathcal{C}]\!] := U.$$
   e. $T = A \to B$. Then,
   $$[\![A \to B]\!] := [\![A]\!] \to [\![B]\!].$$
   f. $T = \forall X\, A$. Then,
   $$[\![\forall X A]\!] := \forall X.\, \mathsf{Cand}_X \to [\![B]\!].$$

We are now ready to define our forcing relation. The notion can be seen as a constructive extension of the standard Tait–Girard computability/reducibility predicate, defined in order to deal with non-computable functionals. It also appears as a kind of constructively refined version of Goodman's forcing (Goodman 1978) (see Section 6 for more about this point).

Definition 15 (Constructive forcing at higher types). Let T be a closed extended type, and $\mathcal{M} : [\![T]\!]$ and $t : |T|$ be closed terms respectively of $\mathcal{F}$ and $\mathcal{F}_{\text{Class}}$. We define the relation

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

$\mathcal{M} \Vdash t \downarrow T$ – representing the notion '$\mathcal{M}$ *forces* t *to be a computable functional of type* T' – by induction on the type T as follows:

1. $T = \mathtt{N}$. Then, $\mathcal{M} \Vdash t \downarrow \mathtt{N} \iff \langle \mathcal{M}, t\rangle \in {}^*\mathtt{N}$. That is,
$$\mathcal{M} \Vdash t \downarrow \mathtt{N} \iff \mathcal{M} = \langle \mathcal{L}, g\rangle, \forall s^{\mathtt{S}} g(s) = t[s] \text{ and } \mathcal{L} \Vdash t \in \mathtt{N}.$$
2. $T = \mathtt{U}$. Then, $\mathcal{M} \Vdash t \downarrow \mathtt{U} \iff \langle \mathcal{M}, t\rangle \in {}^*\mathtt{U}$. That is,
$$\mathcal{M} \Vdash t \downarrow \mathtt{U} \iff \mathcal{M} \Vdash t \in \mathtt{U}\,.$$
3. $T = \mathcal{C}$. Then,
$$\mathcal{M} \Vdash t \downarrow \mathcal{C} \iff \langle \mathcal{M}, t\rangle \in \dot{\mathcal{C}}.$$
4. $T = A \to B$. Then,
$$\mathcal{M} \Vdash t \downarrow A \to B \iff (\forall \mathcal{N}, u.\quad \mathcal{N} \Vdash u \downarrow A \implies \mathcal{M}\mathcal{N} \Vdash tu \downarrow B).$$
5. $T = \forall X\, A$. Then,
$$\mathcal{M} \Vdash t \downarrow \forall X A \iff (\forall \mathcal{N}.\quad \mathcal{N} \Vdash \dot{\mathcal{C}} \in \mathsf{Cand} \implies \mathcal{M}\mathcal{N} \Vdash t \downarrow A[\mathcal{C}/X]).$$

4.4. *Non-Standard updates are forcing candidates*

We are now going to prove that ${}^*\mathtt{U}$ (see Definition 12) is a forcing candidate. This amounts to finding a modulus of prefixed point for any term $u(t) : \mathtt{U}$, with $t : \mathtt{N}$, out of $\mathcal{L}$ and $(\mathcal{M}, g)$, assuming that for every numeral n, $\mathcal{L}_n$ is a modulus of prefixed point for $u(n)$, $\mathcal{M}$ is a modulus of forcing for t and for all states s, $g(s) = t[s]$. We take as input a $k \in \mathbb{K}$ and a state s; we have to compute a state $s' \geqslant s$ such that $k(s')$ is a prefixed point for $u(t)$.

The first – but wrong – solution that comes to mind is the following. Consider any $r \geqslant s$ and first compute $t[r] = n$; then use $\mathcal{L}_n k$ to find an $r' \geqslant r$ such that $k(r')$ is a prefixed point for $u(n)$. But is $k(r')$ a prefixed point for $u(t)$ too? Yes, *provided* $t[k(r')] = n$. So we have to exclude the possibility that $t[r] \neq t[k(r')]$. Now, $k(r')$ is produced from r by the immediate continuation of the computation k' – the one transforming r in to r' – followed by the global one k. Therefore, we have to determine exactly what it is k'. Then we can force the function g representing t to be constant for the rest of computation, which is $k \circ k'$.

But we have just determined that the continuation k' represents the sequence of operations: evaluate g in a state r (which is $t[r]$) obtaining n, apply $\mathcal{L}$ to n and k and use the result to extend the state r to some r', such that $k(r')$ is a prefixed point of $u(n)$. This continuation k', composed with k is the correct global one. Then $k \circ k'$ is given to $\mathcal{M}$ which will force g to be constant in such continuation. Finally, g is transformed into a natural number in the state computed by $\mathcal{M}_{k\circ k'}$ and given to $\mathcal{L}$.

We now define an operation of 'application' $\nabla_{{}^*\mathtt{U}}$, that applies the term $\mathcal{L} : \mathtt{N} \to [\![\mathtt{U}]\!]$ to the non-standard natural number $(\mathcal{M}, g)$, as we have described.

Proposition 3 (${}^*\mathtt{U}$ is a forcing candidate). Let $\mathcal{L} : \mathtt{N} \to (\mathtt{S} \to \mathtt{S}) \to (\mathtt{S} \to \mathtt{S})$, $\mathcal{M} : (\mathtt{S} \to \mathtt{S}) \to (\mathtt{S} \to \mathtt{S})$ and $g : \mathtt{S} \to \mathtt{N}$, $k : \mathtt{S} \to \mathtt{S}$, $s : \mathtt{S}$ be variables whose free occurrences are to be

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

interpreted with the shown types. Define

$$k' := \lambda s.\,(\mathcal{L}_{g(s)})ks,$$
$$k'' := \mathcal{M}_{k\circ k'}$$

and

$$\mathcal{L}\nabla_{\mathtt{U}}(\mathcal{M}, g) := \lambda k\, k' \circ k''.$$

Then,

$$\nabla^{\mathtt{U}} := \lambda\mathcal{M}\,\lambda g\,\lambda\mathcal{L}.\ \mathcal{L}\nabla_{\mathtt{U}}(\mathcal{M}, g) \Vdash {}^{*}\mathtt{U} \in \mathsf{Cand}.$$

Proof. The fact that ${}^{*}\mathtt{U}$ is closed under intensional equality is straightforward. Now, under the assumptions over the types of variables, k' and k'' can be given types $\mathtt{S} \to \mathtt{S}$; thus, $\mathcal{L}\nabla_{\mathtt{U}}(\mathcal{M}, g) : (\mathtt{S} \to \mathtt{S}) \to (\mathtt{S} \to \mathtt{S})$ has the correct type. According to Definition 13, suppose that:

1. $\mathcal{M} \Vdash t \in \mathtt{N}$;
2. for all states s, $g(s) = t[s]$;
3. $\mathcal{L} : \mathtt{N} \to (\mathtt{S} \to \mathtt{S}) \to (\mathtt{S} \to \mathtt{S})$ is a term of System $\mathcal{F}$ such that for every numeral m, $\langle \mathcal{L}m, u(m)t_1 \dots t_n\rangle \in {}^{*}\mathtt{U}$.

We have to prove that

$$\langle \mathcal{L}\nabla_{\mathtt{U}}(\mathcal{M}, g), u(t)t_1 \dots t_n\rangle \in {}^{*}\mathtt{U}$$

which by Definition 12 is to say that $\mathcal{L}\nabla_{\mathtt{U}}(\mathcal{M}, g)$ is a modulus of prefixed point for $u(t)t_1 \dots t_n$. So, let $h \in \mathbb{K}$ and r be a state. Then,

$$((\mathcal{L}\nabla_{\mathtt{U}}(\mathcal{M}, g))h = h' \circ h''$$

with

$$h' := \lambda s.\,(\mathcal{L}_{g(s)})hs,$$
$$h'' := \mathcal{M}_{h\circ h'}.$$

The fact that $h' \circ h'' \in \mathbb{K}$ follows from hypotheses on $\mathcal{M}$ and $\mathcal{L}$ and definition of h', h''. We are thus left to show that

$$r' = h(h' \circ h''(r)) \implies u(t)t_1 \dots t_n[r'] \preceq r'.$$

Since $h'' = \mathcal{M}_{h\circ h'}$ and $\mathcal{M} \Vdash t \in \mathtt{N}$, we obtain that

$$t \downarrow [h''(r), h \circ h'(h''(r))].$$

Let $m = g(h''(r))$. Since $h' = \lambda s.\,(\mathcal{L}_{g(s)})hs$, we have

$$h'(h''(r)) = (\mathcal{L}_m h)(h''(r)).$$

By hypothesis (3) on $\mathcal{L}$, $\mathcal{L}_m$ is a modulus of prefixed point for $u(m)t_1 \dots t_n$. Therefore, by definition of r'

$$u(m)t_1 \dots t_n[r'] \preceq r'.$$

By $h, h' \in \mathbb{K}$, it holds that $h''(r) \leqslant h'(h''(r)) \leqslant h(h'(h''(r))) = r'$. Therefore,

$$t[r'] = t[h''(r)] = g(h''(r)) = m.$$

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

It follows that

$$u(t)t_1 \dots t_n[r'] = u(m)t_1 \dots t_n[r'] \preceq r'$$

which is the thesis. □

4.5. *Non-standard natural numbers are forcing candidates*

We now prove that ${}^*\mathtt{N}$ (see Definition 10) is a forcing candidate. This amounts to computing the denotation of a term $u(t) : \mathtt{N}$ as a non-standard natural number, provided we have the denotation $(\mathcal{M}, g)$ of t and we can compute for every numeral n the denotation of $u(n)$ with a term $\mathcal{L}n$. The modulus of forcing for $u(t)$ is *exactly* the construction used in proving that ${}^*\mathtt{U}$ is a forcing candidate. The denotation of $u(t)$ as a function from states to numbers is just the map taking a state s, computing $t[s] = g(s) = n$, then the denotation of $u(n)$, which is $\pi_1\mathcal{L}n$, and applying it to s.

Given the slight difference between the definition of forcing at type $\mathtt{U}$ and type $\mathtt{N}$, we have to adapt the preceding proof.

Proposition 4 (Non-standard natural numbers are forcing candidates). Let $\mathcal{L} : \mathtt{N} \to ((\mathtt{S} \to \mathtt{S}) \to (\mathtt{S} \to \mathtt{S}) \times \mathtt{S} \to \mathtt{N})$, $\mathcal{M} : (\mathtt{S} \to \mathtt{S}) \to (\mathtt{S} \to \mathtt{S})$ and $g : \mathtt{S} \to \mathtt{N}$, $k : \mathtt{S} \to \mathtt{S}$, $s : \mathtt{S}$ be variables whose free occurrences are to be interpreted with the shown types. Define

$$k' := \lambda s.\, (\pi_0\mathcal{L}_{g(s)})ks,$$
$$k'' := \mathcal{M}_{k \circ k'},$$

and

$$\mathcal{L}\nabla_{{}^*\mathtt{N}}(\mathcal{M}, g) := \langle \lambda k\, k' \circ k'', \lambda s.\, (\pi_1\mathcal{L}_{g(s)})s \rangle.$$

Then,

$$\nabla^{\mathtt{N}} := \lambda\mathcal{M}\, \lambda g\, \lambda\mathcal{L}.\ \mathcal{L}\nabla_{\mathtt{N}}(\mathcal{M}, g) \Vdash {}^*\mathtt{N} \in \mathsf{Cand}.$$

Proof. The fact that ${}^*\mathtt{N}$ is closed under intensional equality is straightforward. Now, according to Definition 13, suppose that:

1. $\mathcal{M} \Vdash t \in \mathtt{N}$;
2. for all states s, $g(s) = t[s]$;
3. $\mathcal{L} : \mathtt{N} \to ((\mathtt{S} \to \mathtt{S}) \to (\mathtt{S} \to \mathtt{S}) \times \mathtt{S} \to \mathtt{N})$ is a term of System $\mathcal{F}$ such that for every numeral m, $\langle \mathcal{L}m, u(m)t_1 \dots t_n \rangle \in {}^*\mathtt{N}$.

We have to prove that

$$\mathcal{L}\nabla_{\mathtt{N}}(\mathcal{M}, g) = \langle \lambda k\, k' \circ k'', \lambda s.\, (\pi_1\mathcal{L}_{g(s)})s \rangle \in {}^*\mathtt{N}$$

and therefore by Definition 10 that: (i) $\lambda k\, k' \circ k''$ is a modulus of forcing for $u(t)t_1 \dots t_n$; (ii) that when evaluated in any state s, the former is equal to $(\pi_1\mathcal{L}_{g(s)})s$. (ii) is immediate: since $g(s) = t[s]$, and by hypothesis (3) for every numeral m, $(\pi_1\mathcal{L}_m)s = u(m)t_1 \dots t_n[s]$, we get

$$(\pi_1\mathcal{L}_{g(s)})s = u(g(s))t_1 \dots t_n[s] = u(t)t_1 \dots t_n[s].$$

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

We now prove (i). Let $h \in \mathbb{K}$ and r be any state. Then,

$$(\lambda k\, k' \circ k'')h = h' \circ h''$$

with

$$h' := \lambda s.\, (\pi_0 \mathcal{L}_{g(s)})hs,$$
$$h'' := \mathcal{M}_{h \circ h'}.$$

The fact that $h' \circ h'' \in \mathbb{K}$ follows from hypotheses on $\mathcal{M}$ and $\mathcal{L}$ and by inspection of the definition of h', h''. It remains to show that

$$u(t)t_1 \dots t_n \downarrow [h'(h''(r)), h(h'(h''(r)))].$$

Since $h'' = \mathcal{M}_{h \circ h'}$ and $\mathcal{M} \Vdash t \in \mathtt{N}$, we obtain that

$$t \downarrow [h''(r), h \circ h'(h''(r))].$$

Let $m = g(h''(r))$. Since $h' = \lambda s.\, (\pi_0 \mathcal{L}_{g(s)})hs$, we have

$$h'(h''(r)) = (\pi_0 \mathcal{L}_m)h(h''(r)).$$

Since $\pi_0 \mathcal{L}_m \Vdash u(m)t_1 \dots t_n \in \mathtt{N}$, we get that

$$u(m)t_1 \dots t_n \downarrow [h'(h''(r)), h(h'(h''(r)))].$$

By $h, h' \in \mathbb{K}$, it holds that $h''(r) \leqslant h'(h''(r)) \leqslant h(h'(h''(r)))$. Therefore, for every r' in the interval $[h'(h''(r)), h \circ h'(h''(r))]$

$$t[r'] = t[h''(r)] = g(h''(r)) = m.$$

It follows that

$$u(t)t_1 \dots t_n \downarrow [h'(h''(r)), h \circ h'(h''(r))]$$

which is the thesis. □

4.6. *The interpretations of types are forcing candidates*

We are now going to prove the fundamental Lemma saying that for each type T the forcing relation $\mathcal{N} \Vdash t \downarrow T$ is indeed a forcing candidate. Before, we need some notation.

Notation. In the following, we shall assume that the type variables of $\mathcal{F}_{\text{Class}}$ are $X_0, X_1, \dots, X_n \dots$ (but when the index is not important, we shall denote them with generical metavariables $X, Y, \dots$). To each type variable X_i we associate a unique term variable x_i.

We start by defining the operation ∇_T of 'application' of a term $\mathcal{L} : \mathsf{Nat} \to [\![T]\!]$ to a non-standard number $(\mathcal{M}, g)$. This is needed to show that the forcing relation is a forcing candidate. For that aim, one needs to find a $\mathcal{N} \Vdash u(t) : T$, provided one has $\langle \mathcal{M}, g \rangle \Vdash t \downarrow \mathtt{N}$ and for every numeral n, $\mathcal{L}n \Vdash u(n) \downarrow T$. Model-theoretically, one has to compute a denotation in $[\![T]\!]$ of a term $u(t) : T$, provided one is able to compute the denotation $(\mathcal{M}, g)$ of t and for every numeral n a denotation $\mathcal{L}n$ of $u(n)$. What we need is just a lifting of the definition $\mathcal{L} \nabla_T (\mathcal{M}, g)$ in Propositions 3, 4.

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

Definition 16 (Collection of moduli turned into a single modulus). Let T be a type without candidate constants. Let $\mathcal{L} : \mathtt{N} \to [\![T]\!]$, $\mathcal{M} : (\mathtt{S} \to \mathtt{S}) \to (\mathtt{S} \to \mathtt{S})$ and $g : \mathtt{S} \to \mathtt{N}$ be term variables whose occurrences are to be interpreted with the shown types. We define by induction on T a term $\mathcal{L}\nabla_T(\mathcal{M}, g)$ having type $[\![T]\!]$ in the context containing $x_i : \mathsf{Cand}_{X_i}$, for every free variable X_i of T.

1. $T = \mathtt{N}$. Then,
$$\mathcal{L}\nabla_T(\mathcal{M}, g) := \nabla^{\mathtt{N}} \mathcal{M} g \mathcal{L}$$
(see Proposition 4 for $\nabla^{\mathtt{N}}$).
2. $T = \mathtt{U}$. Then,
$$\mathcal{L}\nabla_T(\mathcal{M}, g) := \nabla^{\mathtt{U}} \mathcal{M} g \mathcal{L}$$
(see Proposition 3 for $\nabla^{\mathtt{U}}$).
3. T is a variable X_i. Then,
$$\mathcal{L}\nabla_T(\mathcal{M}, g) := x_i \mathcal{M} g \mathcal{L}.$$
4. $T = A \to B$. Then (interpreting $\mathcal{N} : [\![A]\!]$),
$$\mathcal{L}\nabla_T(\mathcal{M}, g) := \lambda \mathcal{N}.\, (\lambda m\, \mathcal{L}_m \mathcal{N}) \nabla_B(\mathcal{M}, g).$$
5. $T = \forall X_i\, A$. Then,
$$\mathcal{L}\nabla_T(\mathcal{M}, g) := \lambda x_i.\, (\lambda m\, \mathcal{L}_m x_i) \nabla_A(\mathcal{M}, g).$$

We define
$$\nabla^T := \lambda \mathcal{M}\, \lambda g\, \lambda \mathcal{L}.\, \mathcal{L}\nabla_T(\mathcal{M}, g).$$

Remark 1. $\mathcal{L}\nabla_T(\mathcal{M}, g)$ reduces to a term of the simple shape:
$$\lambda z_1 \dots \lambda z_n.\, v \mathcal{M} g (\lambda m^{\mathtt{N}} \mathcal{L}_m z_1 \dots z_n),$$
where v is either a variable x_i or $\nabla^{\mathtt{N}}$ or $\nabla^{\mathtt{U}}$. Thus, for any variable y, $\mathcal{L}\nabla_T(\mathcal{M}, g)[t/y]$ is equal to $\mathcal{L}[t/y]\nabla_T(\mathcal{M}, g)$, when $\mathcal{M}$ and g are closed.

We now prove that the operation ∇^T realizes that the interpretation of types is a forcing candidate. In particular, first we prove that the ∇_T is the operation needed for witnessing the closure property of a forcing candidate and thus the forcing relation is a forcing candidate; second, that in forcing every type T can be replaced by any candidate constant interpreting the forcing relation at T, which represents the usual comprehension lemma.

Lemma 4 (The interpretations of types are forcing candidates). Assume T is an extended type. The following hold:

1. Suppose that for $i = 1, \dots, n$, $\mathcal{N}_i \Vdash \dot{\mathcal{C}}_i \in \mathsf{Cand}$ and that $\overline{T} = T[\mathcal{C}_1/X_1 \dots \mathcal{C}_n/X_n]$, where T is a type without candidate constants. Suppose that for every numeral m
$$\mathcal{L}_m \Vdash u(m) t_1 \dots t_k \downarrow \overline{T}$$
and that $\mathcal{M} \Vdash t \in \mathtt{N}$ and $g = \lambda s.\, t[s]$. Then,
$$\mathcal{L}\nabla_T(\mathcal{M}, g)\, [\mathcal{N}_1/x_1 \dots \mathcal{N}_n/x_n] \Vdash u(t) t_1 \dots t_k \downarrow \overline{T}.$$

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

Therefore,

$$\nabla^T[\mathcal{N}_1/x_1 \dots \mathcal{N}_n/x_n] \Vdash \{\langle \mathcal{N}, t\rangle \mid \mathcal{N} \Vdash t \downarrow \overline{T}\} \in \mathsf{Cand}.$$

2. Suppose

$$\dot{\mathcal{C}} := \{\langle \mathcal{N}, t\rangle \mid \mathcal{N} \Vdash t \downarrow B\}.$$

Then,

$$\mathcal{N} \Vdash t \downarrow A[\mathcal{C}/X] \iff \mathcal{N} \Vdash t \downarrow A[B/X].$$

Proof. First of all, we define the following abbreviation:

$$[\vec{\mathcal{N}}/\vec{x}] := [\mathcal{N}_1/x_1 \dots \mathcal{N}_n/x_n].$$

1. The last line of the assertion is a simple corollary of the first part, which we prove by induction on T.
   a. $T = \mathtt{N}$ or $T = \mathtt{U}$. The thesis follows by Propositions 3 and 4.
   b. $T = X_i$. Then,

$$\begin{aligned}\mathcal{L}\nabla(\mathcal{M}, g)[\vec{\mathcal{N}}/\vec{x}] &= x_i \mathcal{M} g \mathcal{L}[\vec{\mathcal{N}}/\vec{x}] \\ &= \mathcal{N}_i \mathcal{M} g \mathcal{L}.\end{aligned}$$

   For every numeral m, we have by hypothesis

$$\mathcal{L}_m \Vdash u(m)t_1 \dots t_k \downarrow \mathcal{C}_i$$

   and thus by Definition 15

$$\langle \mathcal{L}m, u(m)t_1 \dots t_k\rangle \in \dot{\mathcal{C}}_i.$$

   Since $\mathcal{N}_i \Vdash \dot{\mathcal{C}}_i \in \mathsf{Cand}$, we obtain by Definition 13 that

$$\langle \mathcal{N}_i \mathcal{M} g \mathcal{L}, u(t)t_1 \dots t_k\rangle \in \dot{\mathcal{C}}_i$$

   and therefore by Definition 15

$$\mathcal{N}_i \mathcal{M} g \mathcal{L} \Vdash u(t)t_1 \dots t_k \downarrow \mathcal{C}_i$$

   which is the thesis.

   c. $T = A \to B$. Suppose $\mathcal{I} \Vdash t_{k+1} \downarrow A$. We must show that

$$\mathcal{L}\nabla_T(\mathcal{M}, g)[\vec{\mathcal{N}}/\vec{x}]\,\mathcal{I} \Vdash u(t)t_1 \dots t_k t_{k+1} \downarrow B.$$

   We have

$$\begin{aligned}&\mathcal{L}\nabla_T(\mathcal{M}, g)[\vec{\mathcal{N}}/\vec{x}]\,\mathcal{I} \\ &= (\lambda\mathcal{H}.\,(\lambda m\,\mathcal{L}_m\mathcal{H})\nabla_B(\mathcal{M}, g))\,\mathcal{I}\,[\vec{\mathcal{N}}/\vec{x}] \\ &= (\lambda m\,\mathcal{L}_m\mathcal{I})\nabla_B(\mathcal{M}, g)[\vec{\mathcal{N}}/\vec{x}].\end{aligned}$$

   By hypothesis, for every numeral m, $\mathcal{L}_m \Vdash u(m)t_1 \dots t_k \downarrow A \to B$. Therefore, for every numeral m

$$\mathcal{L}_m\mathcal{I} \Vdash u(m)t_1 \dots t_k t_{k+1} \downarrow B.$$

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

By induction hypothesis

$$(\lambda m\, \mathcal{L}_m \mathcal{I}) \nabla_B(\mathcal{M}, g)[\vec{\mathcal{N}}/\vec{x}] \Vdash u(t) t_1 \dots t_k \downarrow B$$

which is the thesis.

d. $T = \forall X_{n+1} A$. Suppose that $\mathcal{N}_{n+1} \Vdash \dot{\mathcal{C}}_{n+1} \in \mathsf{Cand}$. We must show that

$$\mathcal{L} \nabla_T(\mathcal{M}, g)[\vec{\mathcal{N}}/\vec{x}]\, \mathcal{N}_{n+1} \Vdash u(t) t_1 \dots t_k \downarrow A[\mathcal{C}_{n+1}/X_{n+1}].$$

We have

$$\begin{aligned} &\mathcal{L} \nabla_T(\mathcal{M}, g)[\vec{\mathcal{N}}/\vec{x}]\, \mathcal{N}_{n+1} \\ &= \mathcal{L} \nabla_T(\mathcal{M}, g)\, \mathcal{N}_{n+1}\, [\vec{\mathcal{N}}/\vec{x}] \\ &= (\lambda x_{n+1}.\, (\lambda m\, \mathcal{L}_m x_{n+1}) \nabla_A(\mathcal{M}, g))\, \mathcal{N}_{n+1}\, [\vec{\mathcal{N}}/\vec{x}] \\ &= (\lambda m\, \mathcal{L}_m \mathcal{N}_{n+1}) \nabla_A(\mathcal{M}, g)[\vec{\mathcal{N}}/\vec{x}]. \end{aligned}$$

By hypothesis on $\mathcal{L}$, for every numeral m,

$$\mathcal{L}_m \mathcal{N}_{n+1} \Vdash u(m) t_1 \dots t_k \downarrow A[\mathcal{C}_{n+1}/X_{n+1}].$$

Therefore, by induction hypothesis

$$(\lambda m\, \mathcal{L}_m \mathcal{N}_{n+1}) \nabla_A(\mathcal{M}, g)[\vec{\mathcal{N}}/\vec{x}] \Vdash u(t) t_1 \dots t_k \downarrow A[\mathcal{C}_{n+1}/X_{n+1}]$$

which is the thesis.

2. By induction on A.

a. A is a type variable. If $A \neq X$, the thesis is obvious. Suppose $A = X$. Then $A[\mathcal{C}/X] = \mathcal{C}$ and $A[B/X] = B$. Therefore,

$$\mathcal{N} \Vdash t \downarrow A[\mathcal{C}/X] \iff \mathcal{N} \Vdash t \downarrow \mathcal{C} \iff \langle \mathcal{N}, t \rangle \in \dot{\mathcal{C}} \iff \mathcal{N} \Vdash t \downarrow B = A[B/X].$$

b. The other cases are straightforward. □

5. The state-extending-continuation-passing-style method

5.1. *Forcing of constants*

From now on, we devote ourselves to the definition of our SECPS translation $[\![_]\!]$ of terms of System $\mathcal{F}_{\text{Class}}$ into System $\mathcal{F}$. In order to avoid repetition of trivial details, the translation is defined for *proof-like terms* of $\mathcal{F}_{\text{Class}}$: a term t is said to be proof-like if: (i) every occurrence in t of the constants Φ or mkupd is of the form Φ_i or mkupd_i, where i is some numeral; (ii) neither is nor get nor any update constant different from $\varnothing$ occurs in t. Indeed, that is the syntactic form of every interactive realizer extracted from some proof in $\mathsf{HAS} + \mathsf{EM}_1 + \mathsf{SK}_1$. Non proof-like terms, like for example $t\Phi$ or $\{(0, 1, 2)\}$, never appear in actual decorations of any axiom or inference rule (Aschieri 2013).

For each proof-like constant $c : T$ of $\mathcal{F}$, we now define a term $[\![c]\!]$, which is intended to satisfy the relation $[\![c]\!] \Vdash c \downarrow T$. $[\![c]\!]$ can be seen as the non-standard version of

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

the operation denoted by c. The definitions we are going to give perfectly illustrate the remarkable power of the crucial Lemma 4. When one is faced with a computational problem, the use of the lemma allows to automatically transform any solution over the domain of numerals – which is usually a lot easier to construct – into a general solution over the domain of non-standard numbers (i.e. terms of $\mathcal{F}_{\text{Class}}$). More precisely:

— $[\![\Phi_i]\!]$, represents the computational content of EM_1 and SK_1. In order to define it, one first constructs a term $\mathcal{L}$ that given a numeral n, a continuation $k \in \mathbb{K}$ and a current state s, has the task of choosing a state where to approximate $\Phi_i(n)$. $\mathcal{L}$ does that by looking at the future of the computation $s' = k(s)$; if s' disagrees with s on argument (i, n), $\mathcal{L}$ returns the state s' and Φ_i is approximated with $(s')_i(n)$; otherwise $\mathcal{L}$ returns the current state s, and Φ_i is approximated with $s_i(n)$. But we observe that the argument taken by Φ_i is in general a term t of $\mathcal{F}_{\text{Class}}$. An automatic use of Lemma 4 allows to translate the term $\mathcal{L}$ into one that forces $\Phi_i(t) \downarrow \mathtt{N}$. In other words, $[\![\Phi_i]\!]$ first of all forces t to be constant for the rest of the computation and then translates it into a numeral; then it applies the previous considerations.

— $[\![\mathsf{mkupd}_i]\!]$ represents the learning content of EM_1 and SK_1 relative to the predicate P_i. Given a continuation $k \in \mathbb{K}$, a current state s, and two terms $t_1, t_2 : \mathtt{N}$, it has the task of computing a state $s' \geqslant s$ such that

$$\mathsf{mkupd}_i t_1 t_2\, [k(s')] \preceq k(s').$$

Again, when t_1, t_2 are numerals n_1, n_2, the task is easy: it suffices to take $s' = s \oplus \{(i, n_1, n_2)\}$ (remember Definition 6 of $\oplus$); indeed,

$$\mathsf{mkupd}_i n_1 n_2\, [k(s')] = \{(i, n_1, n_2)\} \preceq s \oplus \{(i, n_1, n_2)\} = s' \leqslant k(s').$$

Now we are done, because by two applications of Lemma 4, if we are able to force uniformly $\mathsf{mkupd}_i n_1 n_2 \downarrow \mathtt{U}$, then we are able to force uniformly $\mathsf{mkupd}_i n_1 t_2 \downarrow \mathtt{U}$ and then $\mathsf{mkupd}_i t_1 t_2 \downarrow \mathtt{U}$.

— $[\![\mathbb{U}]\!]$ takes two moduli of prefixed points $\mathcal{M}$ and $\mathcal{N}$ respectively for $t_1 : \mathtt{U}$ and $t_2 : \mathtt{U}$, and must transform them into a single modulus of prefixed point for $t_1 \mathbb{U} t_2$. This recalls Proposition 2 and in fact the solution is the same construction $\mathcal{M} \sqcup \mathcal{N}$.

— $[\![\mathsf{R}]\!]$ is to represent recursion over non-standard natural numbers. The problem is that one does not know how to iterate some functional a 'number' of times given by a term $t : \mathtt{N}$ of System $\mathcal{F}_{\text{Class}}$. But because of Lemma 4, one does not have to worry about that. $[\![\mathsf{R}_A]\!]$ forces t to behave like a fixed number n for the rest of the computation and iterates the functional n times. In more detail, one constructs a term $\mathcal{N}$ which – given a $\mathcal{I} \Vdash u \downarrow \mathcal{C}$, a $\mathcal{L} \Vdash v \downarrow [\![\mathtt{N}]\!] \to \mathcal{C} \to \mathcal{C}$ and a numeral n – just iterates $\lambda n\, \mathcal{L} n^*$, starting from $\mathcal{I}$, n times, obtaining a term forcing $\mathsf{R}uvn \downarrow \mathcal{C}$. Then, given a term $t : \mathtt{N}$ and provided a $\mathcal{H} \Vdash \mathcal{C} \in \mathsf{Cand}$, one builds a term forcing $\mathsf{R}uvt \downarrow \mathcal{C}$ by just applying $\mathcal{H}$ to the non-standard denotation $(\mathcal{M}, g)$ of t and $\mathcal{N}$. This transforms the solution $\mathcal{N}$ over numerals to the more general solution over terms of $\mathcal{F}_{\text{Class}}$.

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

Definition 17 (Translation of constants). We define for every proof-like constant $c : T$ of $\mathcal{F}_{\text{Class}}$ a closed term $[\![c]\!] : [\![T]\!]$, accordingly to the form of c. All the free occurrences of the variables k and s below are to be typed with $k : \mathtt{S} \to \mathtt{S}, s : \mathtt{S}$.

1. $c = 0 : \mathtt{N}$. Let $u : \mathtt{N}$ be any term of System $\mathcal{F}$. We define
$$u^* := \langle \lambda k \lambda s\, s, \lambda s u \rangle.$$
Then,
$$[\![0]\!] := 0^*.$$
2. $c = \varnothing : \mathtt{U}$. Then,
$$[\![\varnothing]\!] := \lambda k \lambda s.\, s.$$
3. $c = \mathsf{S} : \mathtt{N} \to \mathtt{N}$. Then,
$$[\![\mathsf{S}]\!] := \lambda \langle \mathcal{M}, g \rangle\, \langle \mathcal{M}, \lambda s.\, \mathsf{S}(g(s)) \rangle.$$
4. $c = \Phi_i : \mathtt{N} \to \mathtt{N}$. Let
$$\mathcal{L} := \lambda n \langle \lambda k \lambda s \,\mathsf{if}\, s_i(n) = (ks)_i(n) \,\mathsf{then}\, s \,\mathsf{else}\, ks, \lambda s\, s_i(n) \rangle.$$
Then,
$$[\![\Phi]\!] := \lambda \langle \mathcal{M}, g \rangle\, \mathcal{L} \nabla_{\mathtt{N}} (\mathcal{M}, g).$$
5. $c = \mathbb{U} : \mathtt{U} \to \mathtt{U} \to \mathtt{U}$.
$$[\![\mathbb{U}]\!] := \lambda \mathcal{M} \lambda \mathcal{N}.\, \mathcal{M} \sqcup \mathcal{N}.$$
6. $c = \mathsf{mkupd}_i : \mathtt{N} \to \mathtt{N} \to \mathtt{U}$.
$$\mathcal{L} := \lambda n \lambda m \lambda k \lambda s.\, s \oplus \mathsf{mkupd}_i\, n\, m.$$
Then,
$$[\![\mathsf{mkupd}_i]\!] := \lambda \langle \mathcal{M}_1, g_1 \rangle \lambda \langle \mathcal{M}_2, g_2 \rangle\, (\lambda n.\, \mathcal{L} n \nabla_{\mathtt{N}} (\mathcal{M}_2, g_2)) \nabla_{\mathtt{N}} (\mathcal{M}_1, g_1).$$
7. $c = \mathsf{R} : \forall A.\, A \to (\mathtt{N} \to A \to A) \to \mathtt{N} \to A$. Assuming to use the typing context $\mathcal{H} : \mathsf{Cand}_A$, $\mathcal{L} : [\![\mathtt{N}]\!] \to [\![A]\!] \to [\![A]\!]$, $\mathcal{I} : [\![A]\!]$, $\mathcal{M} : (\mathtt{S} \to \mathtt{S}) \to (\mathtt{S} \to \mathtt{S})$, $g : \mathtt{S} \to \mathtt{N}$, define
$$\mathcal{N} := \lambda n\ \mathsf{R} \mathcal{I} (\lambda n\, \mathcal{L} n^*) n.$$
Then,
$$[\![\mathsf{R}]\!] := \lambda \mathcal{H} \lambda \mathcal{I}\, \lambda \mathcal{L}\, \lambda \langle \mathcal{M}, g \rangle.\, \mathcal{H} \mathcal{M} g \mathcal{N}.$$

We now prove that for any constant $c : T$, $[\![c]\!]$ forces c to be a computable functional of type T.

Proposition 5 (Forcing of constants). For every proof-like constant $c : T$
$$[\![c]\!] \Vdash c \downarrow T.$$

Proof. We proceed by cases, accordingly to the form of c.

1. $c : \mathtt{N}$, with $c = 0$. By Definition 17,
$$[\![0]\!] = \langle \lambda k \lambda s\, s, \lambda s\, 0 \rangle.$$

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

We have therefore to prove that $\lambda k \lambda s\, s$ is a modulus of forcing for 0, which is trivially true, since $0[s] = 0$ for every state s and that $(\lambda s\, 0)s = 0 = 0[s]$, which is again trivial. We conclude $[\![0]\!] \Vdash 0 \downarrow \mathtt{N}$.

2. $c = \varnothing$. By Definition 17 of $[\![\varnothing]\!]$,

$$[\![\varnothing]\!] := \lambda k \lambda s.\, s.$$

Let $k \in \mathbb{K}$ and s be a state. Let

$$s' = k([\![\varnothing]\!]ks) = k(s).$$

According to Definition 15, we have to prove that

$$[\![\varnothing]\!] \Vdash \varnothing \downarrow \mathtt{U}$$

and thus that $[\![\varnothing]\!]k \in \mathbb{K}$, which is trivially true, and that

$$\varnothing = \varnothing[s'] \preceq s'$$

which is again trivially true, since $\mathsf{dom}(\varnothing) = \varnothing \subseteq \mathsf{dom}(s)$.

3. $c = \mathsf{S} : \mathtt{N} \to \mathtt{N}$. By Definition 17,

$$[\![\mathsf{S}]\!] = \lambda \langle \mathcal{M}, g \rangle \langle \mathcal{M}, \lambda s.\, \mathsf{S}(g(s)) \rangle.$$

Suppose $\langle \mathcal{M}, g \rangle \Vdash t \downarrow \mathtt{N}$. Then, $\mathcal{M}$ is a modulus of forcing for t and for all states s, $t[s] = g(s)$. Obviously, $\mathcal{M}$ is also a modulus of forcing for $\mathsf{S}(t)$. Moreover, for all states s, $\mathsf{S}(t)[s] = \mathsf{S}(g(s))$. Hence,

$$[\![\mathsf{S}(t)]\!] \langle \mathcal{M}, g \rangle \Vdash \mathsf{S}(t) \downarrow \mathtt{N}$$

which is the thesis.

4. $c = \Phi_i$. Let $t : \mathtt{N}$ and suppose $\langle \mathcal{M}, g \rangle \Vdash t \downarrow \mathtt{N}$. By Definition 15, we have to prove that

$$[\![\Phi_i]\!] \langle \mathcal{M}, g \rangle \Vdash \Phi_i t \downarrow \mathtt{N}.$$

By Definition 17 of $[\![\Phi_i]\!]$,

$$[\![\Phi_i]\!] \langle \mathcal{M}, g \rangle = \mathcal{L} \nabla_{\mathtt{N}} (\mathcal{M}, g)$$

with

$$\mathcal{L} := \lambda n\, \langle \lambda k \lambda s\, \mathsf{if}\ s_i(n) = (ks)_i(n)\ \mathsf{then}\ s\ \mathsf{else}\ ks, \lambda s\, s_i(n) \rangle.$$

Since $\langle \mathcal{M}, g \rangle \Vdash t \downarrow \mathtt{N}$, if we prove that for every numeral n, $\mathcal{L}_n \Vdash \Phi_i n \downarrow \mathtt{N}$, we obtain by Lemma 4 (1.) that $\mathcal{L} \nabla_{\mathtt{N}} (\mathcal{M}, g) \Vdash \Phi_i t \downarrow \mathtt{N}$ and we are done. First, we have to check that for all states s

$$\pi_1 \mathcal{L}_n(s) = s_i(n) = \Phi_i(n)[s]$$

which is true. It remains us to show that, given any numeral n,

$$\pi_0 \mathcal{L}_n = \lambda k \lambda s\, \mathsf{if}\ s_i(n) = (ks)_i(n)\ \mathsf{then}\ s\ \mathsf{else}\ ks$$

is a modulus of forcing for $\Phi_i(n)$. We have to prove that given any $k \in \mathbb{K}$ and state s,

$$\Phi_i(n) \downarrow [(\pi_0 \mathcal{L}_n)_k(s), k((\pi_0 \mathcal{L}_n)_k(s))].$$

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

We have two possibilities:

i. $s_i(n) = (ks)_i(n)$. Since $s \leqslant k(s)$, we have either $(i,n) \in \mathsf{dom}(s)$ and so

$$\forall q^{\mathsf{S}}.\ s \leqslant q \leqslant k(s) \implies q_i(n) = s_i(n)$$

or $(i,n) \notin \mathsf{dom}(s)$ and hence $(i,n) \notin \mathsf{dom}(k(s))$ and therefore

$$\forall q^{\mathsf{S}}.\ s \leqslant q \leqslant k(s) \implies q_i(n) = (ks)_i(n).$$

We conclude

$$\begin{aligned}\Phi_i(n) \downarrow & [s, k(s)] \\ & = [(\pi_0 \mathcal{L}_n)_k(s), k((\pi_0(\mathcal{L}_n)_k(s))]\end{aligned}$$

by definition of $\mathcal{L}$.

ii. $s_i(n) \neq (ks)_i(n)$. Since $s \leqslant k(s)$, we have $(i,n) \in \mathsf{dom}(k(s))$. Since $k(s) \leqslant k(ks)$, we have $(ks)_i(n) = (k(ks))_i(n)$ and with the same reasoning as above we obtain

$$\begin{aligned}\Phi_i(n) \downarrow & [k(s), k(k(s))] \\ & = [(\pi_0 \mathcal{L}_n)_k(s), k((\pi_0 \mathcal{L}_n)_k(s))].\end{aligned}$$

5. $c = \mathbb{U} : \mathtt{U} \to \mathtt{U} \to \mathtt{U}$. By Definition 17,

$$[\![\mathbb{U}]\!] = \lambda \mathcal{M} \lambda \mathcal{N}.\, \mathcal{M} \sqcup \mathcal{N}_{\swarrow}$$

Suppose $\mathcal{M} \Vdash t_1 \downarrow \mathtt{U}, \mathcal{N} \Vdash t_2 \downarrow \mathtt{U}$. By Proposition 2, $\mathcal{M} \sqcup \mathcal{N} \Vdash t_1 \downarrow \mathtt{U}$ and $\mathcal{M} \sqcup \mathcal{N} \Vdash t_2 \downarrow \mathtt{U}$. Let $k \in \mathbb{K}$ and s be a state. We must show that

$$s' = k([\![\mathbb{U}]\!] \mathcal{M} \mathcal{N} k s) \implies \mathbb{U} t_1 t_2 [s'] \preceq s'.$$

Since $s' = k(\mathcal{M} \sqcup \mathcal{N} k s)$, we get $t_1[s'] \preceq s'$ and $t_2[s'] \preceq s'$. Since, the update $\mathbb{U} t_1 t_2 [s']$ is contained in the union of $t_1[s']$ and $t_2[s']$, we have that $\mathbb{U} t_1 t_2 [s'] \preceq s'$, which is the thesis.

6. $c = \mathsf{mkupd}_i : \mathtt{N} \to \mathtt{N} \to \mathtt{U}$. Let

$$\mathcal{L} := \lambda n \lambda m \lambda k \lambda s.\, s \oplus \mathsf{mkupd}_i\, n\, m.$$

By Definition 17,

$$[\![\mathsf{mkupd}_i]\!] := \lambda \langle \mathcal{M}_1, g_1 \rangle \lambda \langle \mathcal{M}_2, g_2 \rangle\, (\lambda n.\, \mathcal{L} n \nabla_{\mathtt{N}}(\mathcal{M}_2, g_2)) \nabla_{\mathtt{N}}(\mathcal{M}_1, g_1).$$

Suppose $\langle \mathcal{M}_1, g_1 \rangle \Vdash t_1 \downarrow \mathtt{N}$ and $\langle \mathcal{M}_2, g_2 \rangle \Vdash t_2 \downarrow \mathtt{N}$. Let n_1 and n_2 be two numerals. We want first to show that

$$\mathcal{L}\, n_1 n_2 = \lambda k \lambda s.\, s \oplus \mathsf{mkupd}_i\, n_1\, n_2 \Vdash \mathsf{mkupd}_i\, n_1\, n_2 \downarrow \mathtt{U}.$$

So, let $k \in \mathbb{K}$ and s be a state. We must show that

$$s' = \mathcal{L}\, n_1 n_2\, k\, s \implies \mathsf{mkupd}_i n_1 n_2\, [k(s')] \preceq k(s').$$

Indeed,

$$\mathsf{mkupd}_i n_1 n_2\, [k(s')] = \{(i, n_1, n_2)\} \preceq s \oplus \{(i, n_1, n_2)\} = \mathcal{L}\, n_1 n_2\, k\, s = s' \leqslant k(s').$$

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

Now, by Lemma 4, for every numeral n

$$\mathcal{L}n\nabla_{\mathtt{N}}(\mathcal{M}_2, g_2) \Vdash \mathsf{mkupd}_i\, n\, t_2 \downarrow \mathtt{U}.$$

Again by Lemma 4,

$$(\lambda n.\, \mathcal{L}n\nabla_{\mathtt{N}}(\mathcal{M}_2, g_2))\nabla_{\mathtt{N}}(\mathcal{M}_1, g_1) \Vdash \mathsf{mkupd}_i\, t_1 t_2 \downarrow \mathtt{U}$$

which is the thesis.

7. $c = \mathsf{R} : \forall A.\, A \to (\mathtt{N} \to A \to A) \to \mathtt{N} \to A$. Let

$$\mathcal{N} := \lambda n\ \mathsf{R}\,\mathcal{I}(\lambda n\, \mathcal{L}n^*)n.$$

By Definition 17,

$$[\![\mathsf{R}]\!] = \lambda\mathcal{H}\lambda\mathcal{I}\,\lambda\mathcal{L}\,\lambda\langle\mathcal{M}, g\rangle.\, \mathcal{H}\mathcal{M}g\mathcal{N}\swarrow$$

Suppose $\mathcal{H} \Vdash \mathcal{C} \in \mathsf{Cand}$, $\mathcal{I} \Vdash u \downarrow \mathcal{C}$, $\mathcal{L} \Vdash v \downarrow \mathtt{N} \to \mathcal{C} \to \mathcal{C}$ and $\langle\mathcal{M}, g\rangle \Vdash t \downarrow \mathtt{N}$. We have to prove that

$$[\![\mathsf{R}]\!]\,\mathcal{H}\mathcal{I}\mathcal{L}\langle\mathcal{M}, g\rangle = \mathcal{H}\mathcal{M}g\mathcal{N} \Vdash \mathsf{R}uvt \downarrow \mathcal{C}.$$

If we show that for all numerals n, $\mathcal{N}_n \Vdash \mathsf{R}uvn \downarrow \mathcal{C}$, then for all numerals n, $\langle\mathcal{N}_n, \mathsf{R}uvn\rangle \in \mathcal{C}$ and by Definition 13 of $\mathcal{H} \Vdash \mathcal{C} \in \mathsf{Cand}$ we obtain that

$$\langle\mathcal{H}\mathcal{M}g\mathcal{N}, \mathsf{R}uvt\rangle \in \dot{\mathcal{C}}$$

which is the thesis. We prove that by induction on n.
If $n = 0$, then

$$\mathcal{N}_0 = \mathsf{R}\,\mathcal{I}(\lambda n\, \mathcal{L}n^*)0 = \mathcal{I} \Vdash u = \mathsf{R}uv0 \downarrow \mathcal{C}\swarrow$$

If $n = \mathsf{S}(m)$, then

$$\begin{aligned}\mathcal{N}_{\mathsf{S}(m)} &= \mathsf{R}\,\mathcal{I}(\lambda n\, \mathcal{L}n^*)\mathsf{S}(m)\\ &= (\lambda n\, \mathcal{L}n^*)m(\mathsf{R}\mathcal{I}(\lambda n\, \mathcal{L}n^*)m)\\ &= \mathcal{L}m^*(\mathsf{R}\,\mathcal{I}(\lambda n\, \mathcal{L}n^*)m)\\ &= \mathcal{L}m^*\mathcal{N}_m.\end{aligned}$$

By induction hypothesis, $\mathcal{N}_m \Vdash \mathsf{R}uvm \downarrow \mathcal{C}$. Moreover, $m^* \Vdash m \downarrow \mathtt{N}$ and by hypothesis $\mathcal{L} \Vdash v \downarrow \mathtt{N} \to \mathcal{C} \to \mathcal{C}$. Hence,

$$\mathcal{L}m^*\mathcal{N}_m \Vdash vm(\mathsf{R}uvm) = \mathsf{R}uv\mathsf{S}(m) \downarrow \mathcal{C}$$

which is the thesis. □

5.2. *The adequacy theorem*

We are finally ready to define the translation of every term of $\mathcal{F}_{\text{Class}}$ into a term of $\mathcal{F}$. The translation is defined by induction on typing derivations, and thus can be seen as a transformation of Church-style terms of $\mathcal{F}_{\text{Class}}$ into Curry-typed terms of $\mathcal{F}$ that

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

mirrors the structure of the original term. That contrasts the usual 'forgetful' translation of Church-style terms into Curry-style terms that just erases all types and second-order abstractions ΛX_i (see e.g. Sorensen and Urzyczyn (2006), 11.4). Overall, our transformation leaves *unchanged* the structure of terms, and it is essentially a transformation of constants. Types occurring in terms are transformed in realizers that the types are candidates.

Definition 18 (Forcing for terms of $\mathcal{F}_{\text{Class}}$). Fix a derivation $\mathbf{d}$ of $\Gamma \vdash v : T$ in system $\mathcal{F}_{\text{Class}}$, with v proof-like, and assume $\Gamma = z_1 : A_1, \ldots, z_n : A_n$. We define by induction on the derivation $\mathbf{d}$ a term $[\![\mathbf{d}]\!]$ such that $[\![\Gamma]\!] \vdash [\![\mathbf{d}]\!] : [\![T]\!]$, where $[\![\Gamma]\!] = z_1 : [\![A_1]\!], \ldots, z_n : [\![A_n]\!]$:

1. $\mathbf{d} = c$, with c constant. We define $[\![\mathbf{d}]\!] = [\![c]\!]$ as in Definition 17.
2. $\mathbf{d} = z^{A_1}$, z variable. Then,
$$[\![z^{A_1}]\!] := z.$$
3. $\mathbf{d} = \mathbf{ut}$. Then,
$$[\![\mathbf{ut}]\!] := [\![\mathbf{u}]\!][\![\mathbf{t}]\!].$$
4. $\mathbf{d} = \lambda z^A \mathbf{u}$. Then,
$$[\![\lambda z^A \, \mathbf{u}]\!] := \lambda z \, [\![\mathbf{u}]\!].$$
5. $\mathbf{d} = \mathbf{u}V$, with V type. Then,
$$[\![\mathbf{u}V]\!] := [\![\mathbf{u}]\!] \nabla^V.$$
6. $\mathbf{d} = \Lambda X_i \, \mathbf{u}$. Then,
$$[\![\Lambda X_i \, \mathbf{u}]\!] := \lambda x_i \, [\![\mathbf{v}]\!].$$

We are now able to prove our main theorem. For every closed term u of $\mathcal{F}_{\text{Class}}$, $[\![u]\!]$ forces u to be a computable functional.

Theorem 5 (Adequacy theorem). Suppose $\mathbf{w}$ is a derivation of
$$\Gamma = z_1 : A_1, \ldots, z_n : A_n \vdash w : A$$
in $\mathcal{F}_{\text{Class}}$, with w proof-like, and $X_1, \ldots, X_m$ contain all the free type variables of $A_1, \ldots, A_n, A$. Suppose also
$$\mathcal{H}_1 \Vdash \dot{\mathcal{C}}_1 \in \mathsf{Cand}, \ldots, \mathcal{H}_m \Vdash \dot{\mathcal{C}}_m \in \mathsf{Cand}$$
and
$$\mathcal{M}_1 \Vdash t_1 \downarrow A_1[\mathcal{C}_1/X_1 \ldots \mathcal{C}_m/X_m], \ldots, \mathcal{M}_n \Vdash t_n \downarrow A_n[\mathcal{C}_1/X_1 \ldots \mathcal{C}_m/X_m].$$
Then,
$$[\![\mathbf{w}]\!][\mathcal{H}_1/x_1 \ldots \mathcal{H}_m/x_m][\mathcal{M}_1/z_1 \ldots \mathcal{M}_n/z_n]$$
$$\Vdash$$
$$w[t_1/z_1 \ldots t_n/z_n] \downarrow A[\mathcal{C}_1/X_1 \ldots \mathcal{C}_m/X_m].$$

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

Proof.
For any type B, set $\overline{B} := B[\mathcal{C}_1/X_1 \dots \mathcal{C}_m/X_m]$, For any term v, we set

$$\widetilde{v} := v[t_1/z_1 \cdots t_n/z_n]$$

and

$$\widehat{v} := v[\mathcal{H}_1/x_1 \dots \mathcal{H}_m/x_m][\mathcal{M}_1/z_1 \dots \mathcal{M}_n/z_n].$$

With that notation, we have to prove that $\widehat{[\![\mathbf{w}]\!]} \Vdash \widetilde{w} \downarrow \overline{A}$. The proof is by induction on the derivation $\mathbf{w}$ and proceeds by cases, accordingly to the last rule in the derivation.

1. $w = c$, $\mathbf{w} = c$, with c constant. Since $[\![c]\!]$ is closed and c does not have free variables, by Proposition 5
$$\widehat{[\![c]\!]} = [\![c]\!] \Vdash c = \widetilde{c} \downarrow \overline{A}$$
which is the thesis.
2. $w = z_i$, $\mathbf{w} = z_i^{A_i}$ for some $1 \leqslant i \leqslant n$. Then,
$$\widehat{[\![z_i^{A_i}]\!]} = z_i[\mathcal{M}_1/z_1 \dots \mathcal{M}_n/z_n] = \mathcal{M}_i \Vdash t_i = z_i[t_1/z_1 \cdots t_n/z_n] = \widetilde{w} \downarrow \overline{A}$$
which is the thesis.
3. $w = ut$, $\mathbf{w} = \mathbf{ut}$ with $\mathbf{u}$ derivation of $\Gamma \vdash u : B \to A$ and $\mathbf{t}$ derivation of $\Gamma \vdash t : B$. By induction hypothesis, $\widehat{[\![\mathbf{u}]\!]} \Vdash \widetilde{u} \downarrow \overline{B} \to \overline{A}$ and $\widehat{[\![\mathbf{t}]\!]} \Vdash \widetilde{t} \downarrow \overline{B}$. So,
$$\widehat{[\![\mathbf{ut}]\!]} = \widehat{[\![\mathbf{u}]\!]}\ \widehat{[\![\mathbf{t}]\!]} \Vdash \widetilde{u}\widetilde{t} = \widetilde{w} \downarrow \overline{A}$$
which is the thesis.
4. $w = \lambda z u$, $\mathbf{w} = \lambda z^B \mathbf{u}$ with $\mathbf{u}$ derivation of $\Gamma, z : B \vdash u : C$, $A = B \to C$. Suppose $\mathcal{M} \Vdash t \downarrow \overline{B}$. We have to prove that $\widehat{[\![\lambda z^B \mathbf{u}]\!]}\mathcal{M} \Vdash \widetilde{w}t \downarrow \overline{C}$. By induction hypothesis,
$$\widehat{[\![\lambda z^B \mathbf{u}]\!]}\mathcal{M} = (\lambda z\ \widehat{[\![\mathbf{u}]\!]})\mathcal{M} = \widehat{[\![\mathbf{u}]\!]}[\mathcal{M}/z] \Vdash \widetilde{u}[t/z] = \widetilde{w}t \downarrow \overline{A}$$
which is the thesis.
5. $\mathbf{w} = \mathbf{v}V$, where $\mathbf{v}$ is a derivation of $\Gamma \vdash w : \forall X B$ and $A = B[V/X]$. Assume
$$\dot{\mathcal{C}} := \{\langle \mathcal{N}, t\rangle \mid \mathcal{N} \Vdash t \downarrow \overline{V}\}.$$
By Lemma 4, point 1, and by the fact that only the variables $x_1, \dots, x_m$ can occur free in ∇^V, we obtain
$$\widehat{(\nabla^V)} = \nabla^V[\mathcal{H}_1/x_1 \dots \mathcal{H}_m/x_m] \Vdash \mathcal{C} \in \mathsf{Cand}.$$
By induction hypothesis, $\widehat{[\![\mathbf{v}]\!]} \Vdash \widetilde{w} \downarrow \forall X \overline{B}$. Therefore,
$$\widehat{[\![\mathbf{v}V]\!]} = \widehat{[\![\mathbf{v}]\!]}\ \widehat{(\nabla^V)} \Vdash \widetilde{w} \downarrow \overline{B}[\mathcal{C}/X].$$
By Lemma 4, point 2, we obtain
$$\widehat{[\![\mathbf{v}V]\!]} \Vdash \widetilde{w} \downarrow \overline{B}[\overline{V}/X] = \overline{A}$$
which is the thesis.

6. $\mathbf{w} = \Lambda X_i \mathbf{v}$, where $\mathbf{v}$ is a derivation of $\Gamma \vdash w : B$, $A = \forall X_i B$, $i > n$, and X_i does not occur free in Γ. Since $\widehat{[\![\mathbf{w}]\!]} = \lambda x_i \widehat{[\![\mathbf{v}]\!]}$, we have to assume that $\mathcal{H} \Vdash \mathcal{C} \in \mathsf{Cand}$, and then

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

show

$$(\lambda x_i \widehat{[\![\mathbf{v}]\!]})\mathcal{H} = \widehat{[\![\mathbf{v}]\!]}[\mathcal{H}/x_i] \Vdash \widetilde{w} \downarrow \overline{B}[\mathcal{C}/X_i].$$

But this follows by the induction hypothesis. □

As corollary of Theorem 5, we obtain the main result: every closed term of $\mathcal{F}_{\text{Class}}$ can be forced to be computable at any closed type.

Theorem 6. Suppose $t : A$ is any closed term of $\mathcal{F}_{\text{Class}}$ of closed type and let $\mathbf{t}$ be any of its typing derivations. Then,

$$[\![\mathbf{t}]\!] \Vdash t \downarrow A.$$

5.3. *Witness extraction with the state-extending-continuation-passing-style translation method*

At the end, we are able to show that our translation can be used for program extraction with Interactive realizability. Given any term $v : \mathtt{N} \to \mathtt{U}$ of $\mathcal{F}_{\text{Class}}$, we can easily obtain a prefixed point of vx, for any numeral x, by taking its typing derivation $\mathbf{v}$, translating it into $[\![\mathbf{v}]\!]$, and giving it as argument x^*, obtaining a modulus of prefixed point for vn. It then suffices to give it the identity continuation $\lambda s.\, s$. This construction is applied to the term $\lambda n\, \pi_1(tn) : \mathtt{N} \to \mathtt{U}$, where t is a realizer of $\forall x^{\mathtt{N}} \exists y^{\mathtt{N}} P x y$, with P atomic.

Theorem 7 (Program extraction via SECPS-translation). Let t be a proof-like term of $\mathcal{F}_{\text{Class}}$ and suppose that $t \Vdash\!\Vdash \forall x^{\mathtt{N}} \exists y^{\mathtt{N}} P x y$, with P atomic predicate of Gödel's T. Let $r : \mathtt{S}$ be any state, $n : \mathtt{N}$, $x : \mathtt{N}$ and $\mathbf{v}$ a derivation of the term $\lambda n\, \pi_1(tn) : \mathtt{N} \to \mathtt{U}$. Define

$$s := [\![\mathbf{v}]\!] x^* (\lambda z\, z) r$$

and a term $\mathsf{wit} : \mathtt{N} \to \mathtt{N}$ of $\mathcal{F}$ as follows:

$$\mathsf{wit} := \lambda x.\, \pi_0(tx)[s].$$

Then, for all numerals m, $Pm(\mathsf{wit}\, m) = \mathtt{True}$.

Proof. Let m be any numeral and $s' := s[m/x]$. Since $tm \Vdash\!\Vdash_{s'} \exists y^{\mathtt{N}} P m y$ and $\mathsf{wit}\, m = \pi_0(tm)[s']$, we have that

$$\pi_1(tm) \Vdash\!\Vdash_{s'} P m(\mathsf{wit}\, m)$$

which by definition of realizability in particular means that

$$\pi_1(tm)[s'] = \emptyset \implies P m(\mathsf{wit}\, m) = \mathtt{True}.$$

Thus, in order to obtain the thesis, it is enough to show that $\pi_1(tm)[s'] = \emptyset$. By the Adequacy Theorem 5,

$$[\![\mathbf{v}]\!] \Vdash \lambda n\, \pi_1(tn) \downarrow \mathtt{N} \to \mathtt{U}.$$

Since $m^* \Vdash m \downarrow \mathtt{N}$, we obtain that

$$[\![\mathbf{v}]\!] m^* \Vdash \pi_1(tm) \downarrow \mathtt{U}.$$

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

Therefore, by Definition 15, $\pi_1(tm)[s'] \preceq s'$ which means that

$$\mathsf{dom}(\pi_1(tm)[s']) \subseteq \mathsf{dom}(s').$$

Since $\pi_1(tm) \Vdash_{s'} Pm(\mathsf{wit}\, m)$, we also have that $\mathsf{dom}(\pi_1(tm)[s']) \cap \mathsf{dom}(s') = \varnothing$. Therefore, $\pi_1(tm)[s'] = \varnothing$. □

6. Conclusions and related works

We conclude the paper with some remarks.

6.1. *Interactive realizability and krivine classical realizability*

We have considered the witness extraction problem for the Interactive realizability interpretation of second-order Arithmetic $\mathsf{HAS} + \mathsf{EM}_1 + \mathsf{SK}_1$. Then, we have presented two program-extraction techniques. The second one is a new CPS translation, it has a nice model theoretic meaning and it appears as a first step towards a more efficient computational interpretation of classical logic. Indeed, the defect of the programs extracted from classical proofs is that they waste too many resources: when they backtrack, they tend to 'forget' precious information.

For example, let us examine Krivine classical realizability (Krivine 2009) interpretation of excluded middle with cc. Consider a computation of the form

$$\mathsf{cc}\, t \star \pi \succ t \star \mathsf{k}_\pi.\pi \succ \ldots \succ \mathsf{k}_\pi \star u.\rho \succ u \star \pi.$$

The penultimate instruction executed in this computation is an operation of backtracking: the process takes the term u and put it in the context π of the first instruction. Thus, everything between the first and last instruction of the process is discarded, except for u. But it is unreasonable that everything must be erased; it might happen, for example, that between the first and last instruction some witness for some instance of the excluded middle EM_1 is learned.

Krivine's realizability interpretation of choice axioms, such as countable choice and SK_1, suffers other inefficiency problems which, on the contrary, the interpretation of the excluded middle does not. Indeed, the choice functions are *defined* in the realizability model through the minimum principle. This approach leads to a waste of resources: a choice function is allowed to return any witness, while imposing some arbitrary criterion over the witnesses that must be returned may force a realizer to change witness even if its current one is correct.

Realizability based on states seems an answer to this kind of efficiency issues. In classical logic, for writing down more efficient programs, it seems necessary to describe exactly: (a) what the programs learn; (b) how the knowledge of programs varies during the execution; (c) how to prevent unnecessary knowledge losses and revisions. The programs obtained by our SECPS-translation not only are able to backtrack as programs with cc can do, but also keep all the useful information coming from a failure. In fact, when they backtrack (i.e. when the program $[\![\Phi_i]\!]$ is executed), they restart the computation into a state coming from the execution of a state-extending continuation and thus possibly much bigger than

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

the current one. That is, even in a dead branch of the execution, something useful might be learned and has to be kept. Moreover, no arbitrary definition of the choice function that interprets SK_1 is given and realizers are free to keep any witness they find during computations.

6.2. *Interactive realizability and bar recursive interpretations of analysis*

As another example of inefficiency problems in classical logic, let us consider Spector's computational interpretation of Analysis by means of bar recursion (Spector 1962). Albeit it provides a very interesting computational reduction of the countable choice axiom to recursion over well-founded trees, Spector's bar recursion is dramatically inefficient. Choice functions are approximated not just at the points that are asked in some particular computation; instead, one builds finite initial segments of the shape $f(0), f(1), f(2), \ldots, f(n)$, regardless the fact that some of these values are not at all asked.

A nice solution to this problem has been provided in Berardi et al. (1998). The idea is to approximate choice functions only with respect to values that are asked during the computation. For this, they have introduced a much more efficient bar recursion, that indeed may be called: *demand-driven* bar recursion. It satisfies this axiom (Berger 2005):

$$(\mathsf{DDBR})Y\,GHs = Y\,(\lambda n^{\mathtt{N}}\mathsf{if}\ n \in \mathsf{domain}(s)\ \mathsf{then}\ s(n)\ \mathsf{else}\ Gn(\lambda z^{A}(\mathsf{DDBR})Y\,GHs \cup (n,z)))$$

where s is a finite partial function $\mathtt{N} \to A$, with $n \in \mathsf{domain}(s)$ and $s \cup (n,z)$ having the expected meaning, and $Y : (\mathtt{N} \to A) \to \mathtt{N}$, $G : \mathtt{N} \to (A \to \mathtt{N}) \to A$.

Unfortunately, with each recursive call of DDBR, the functional Y is recomputed from scratch even if the partial function s changes only in one point (it becomes $s \cup (n,z)$). This inefficiency is similar to the one of the Iterative Method that we have pointed out in Section 3.1. Since the SECPS-translation was devised precisely to solve this issue, it provides a more efficient computational interpretation than demand-driven bar recursion (of course in our limited setting).

6.3. *Comparison with Goodman's forcing*

Also Goodman (1978) gave a forcing definition of what it means that a functional of finite type depending on some non-recursive function is 'computable'. The worlds considered by Goodman are partial functions order by inclusion and for him a term t of type $\mathtt{N}$ is 'forced to be computable of type $\mathtt{N}$ by a partial function p' if t is defined in p, that is, if p contains all the values needed for reducing t to a natural number. Instead, we use as states total functions; but we can easily rephrase Goodman's concept of being defined in our framework.

Since Goodman's definition refers to finite types, we concentrate on terms of $\mathcal{T}_{\text{Class}}$. We say that a term $t \in \mathcal{T}_{\text{Class}}$ is *defined in a state* $s : \mathtt{S}$, if for all $s' \geqslant s$, $t \downarrow [s,s']$. Intuitively, if t is defined in s, then s contains all the information necessary to compute t, since increasing the information of s yields the same value. Then, Goodman's forcing relation $s\,\mathbf{F}\,t \in A$,

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

where t is a term of $\mathcal{T}_{\text{Class}}$ of type A and s is a state, can be defined as:

$$s \mathbf{F}\ t \in \mathtt{N} \iff t \text{ is defined in } s,$$

$$s \mathbf{F}\ t \in A \to B \iff (\forall s' \geqslant s.\, s' \mathbf{F}\ u \in A \implies \exists s'' \geqslant s'.\, s'' \mathbf{F}\ tu \in B).$$

Then, one defines

$$\mathbf{F}\ t \in A \iff \forall s \exists s' \geqslant s.\, s' \mathbf{F}\ t \in A.$$

Unfortunately, from the constructive point of view, already the notion $s \mathbf{F} t \in \mathtt{N}$ is unsustainably strong. Indeed, to prove $\mathbf{F}\ t \in \mathtt{N}$ one needs a classical metatheory!

However, if one takes the Kreisel no-counterexample interpretation (Kreisel 1959) of $\mathbf{F}\ t \in \mathtt{N}$, one can define

$$s \Vdash t \in \mathtt{N} \iff \forall k \in \mathbb{K} \exists s' \geqslant s.\, t \downarrow [s, k(s)]$$

and $\Vdash t \in \mathtt{N}$ as $\forall s.\, s \Vdash t \in \mathtt{N}$. Then, applying intuitionistic realizability, one obtains our Definition 9 of forcing $\mathcal{M} \Vdash t \in \mathtt{N}$. To put it in another way, one replaces the unrestricted quantification over future worlds, typical of forcing semantics, with a restricted one: the future that can be considered is only the future of the computation.

6.4. *Connection with Escardo's forcing*

The ultimate reason why many results of this paper hold is continuity: realizers only need a finite amount of information about Skolem functions to be effective and thus the idea of approximating them works well. Escardo (2013) defines a new notion of effectful forcing to prove the continuity of Gödel's System T terms of type $(\mathtt{N} \to \mathtt{N}) \to \mathtt{N}$. It might be worth investigating whether there is a connection between our constructive forcing and Escardo's.

6.5. *Interactive realizability, Avigad's forcing and update procedures*

While at the time of Aschieri and Berardi (2010) it was not completely clear how Interactive realizability relates to forcing, it is now evident that the theory of Interactive realizability, as developed in this paper, is a constructivization of Avigad's forcing. In fact, we have proved conservativity of $\mathsf{HAS} + \mathsf{EM}_1 + \mathsf{SK}_1$ over HAS for Π^0_2-formulas.

Another contribution of this paper is related to Avigad's update procedures. The concept of update procedure (Avigad 2002; Aschieri 2011b) provides an axiomatization of the computational content of the epsilon substitution method. However, up to now update procedures have never been directly extracted from impredicative proofs. Here, we have showed how to do that, since realizers of Σ^0_1-formulas *are* update procedures.

References

Aschieri, F. (2011a). *Learning, Realizability and Games in Classical Arithmetic*, Ph.D. thesis. http://arxiv.org/abs/1012.4992

Aschieri, F. (2011b). Transfinite update procedures for predicative systems of analysis. In *Proceedings of Computer Science Logic, Leibniz International Proceedings in Informatics* **12** 20–34.

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

Aschieri, F. (2011c). A constructive analysis of learning in Peano arithmetic. *Annals of Pure and Applied Logic* **162** (11) 1448–1470.

Aschieri, F. (2013). Interactive realizability for second-order arithmetic with EM1 and SK1. *Mathematical Structures in Computer Science* **24** (6).

Aschieri, F. and Berardi, S. (2010). Interactive learning-based realizability for heyting arithmetic with EM_1. *Logical Methods in Computer Science* **6** (3).

Aschieri, F. and Berardi, S. (2011). A new use of Friedman's translation: Interactive realizability. In: *Logic, Construction, Computation, Ontos-Verlag Mathematical Logic*.

Avigad, J. (2002). Update procedures and 1-consistency of arithmetic. *Mathematical Logic Quarterly* **48** (1) 3–13.

Avigad, J. (2003). Eliminating definitions and Skolem functions in first-order logic. *ACM Transactions on Computational Logic* **4** (3) 402–415.

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

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.

Berardi, S. and de' Liguoro, U. (2008). A calculus of realizers for EM1 arithmetic. *Computer Science Logic, Lecture Notes in Computer Science* **5213** 215–229.

Berger, U. (2005). Strong normalization for applied Lambda calculi. *Logical Methods in Computer Science*.

Cohen, P. (1967). *Set Theory and the Continuum Hypothesis*, Dover publications.

Escardo, M. (2013). Continuity of Gödel's system T functionals via effectful forcing. *Mathematical Foundation of Programming Semantics, Electronic Notes in Theoretical Computer Science* **298** 119–141.

Escardo, M. and Oliva, P. (2010). Selection functions, bar recursion and backward induction. *Mathematical Structures in Computer Science* **20** (2) 127–168.

Friedman, H. (1978). Classically and intuitionistically provable recursive functions. *Lecture Notes in Mathematics* **669** 21–27.

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

Goodman, N.D. (1978). Relativized realizability in intuitionistic arithmetic of all finite types. *Journal of Symbolic Logic* **43** (1) 23–44.

Griffin, T. (1990). A formulae-as-type notion of control. In: *Proc. of POPL* 47–58.

Kohlenbach, U. (2008). *Applied Proof Theory*, Springer-Verlag, Berlin, Heidelberg.

Kreisel, G. (1951). On the interpretation of non-finitist proofs, part I. *Journal of Symbolic Logic* **16** (4) 241–267.

Kreisel, G. (1959). Interpretation of analysis by means of constructive functionals of finite types. In: Heyting, A. (ed.) *Constructivity in Mathematics*, North-Holland, Amsterdam, 101–128.

Krivine, J.-L. (2009). Realizability in classical logic. In *Interactive Models of Computation and Program Behaviour*. Panoramas et synthses, Socit Mathmatique de France **27** 197–229.

Krivine, J.-L. (2011). Realizability algebras: a program to well-order R. *Logical Methods in Computer Science* **7** (3).

Mints, G., Tupailo, S. and Bucholz, W. (1996). Epsilon substitution method for elementary analysis. *Archive for Mathematical Logic* **35** 103–130.

Miquel, A. (2011). Forcing as a program transformation. *Logic in Computer Science* 197–206.

Moggi, E. (1991). Notions of computations and Monads. *Journal of Logic and Computation* **93** (1) 55–92.

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press

Murthy, C. (1990). *Extracting Constructive Content from Mathematical Proofs*, Ph.D. thesis, Cornell University.

Oliva, P. and Streicher, T. (2008). On Krivine realizability interpretation of second-order classical arithmetic. *Fundamenta Informaticae*.

Parigot, M. (1992). Lambda-my-calculus: An algorithmic interpretation of classical natural deduction. In LPAR 1992: 190–201.

Sorensen, M.H. and Urzyczyn, P. (2006). Lectures on the Curry-Howard isomorphism. *Studies in Logic and the Foundations of Mathematics*, **149**, Elsevier.

Spector, C. (1962). Provably recursive functionals of analysis: A consistency proof of analysis by an extension of principles in current intuitionistic mathematics. In: Dekker (ed.) *Recursive Function Theory: Proceedings of Symposia in Pure Mathematics*, **5**. AMS, Providence.

https://doi.org/10.1017/S0960129515000468 Published online by Cambridge University Press