Computational interpretations of classical reasoning: From the epsilon calculus to stateful programs

Thomas Powell

Abstract

The problem of giving a computational meaning to classical reasoning lies at the heart of logic. This article surveys three famous solutions to this problem - the epsilon calculus, modified realizability and the Dialectica interpretation - and re-examines them from a modern perspective, with a particular emphasis on connections with algorithms and programming.

1 Introduction

This paper grew out of two talks I gave on the subject of proof interpretations: The first at a conference on Hilbert's epsilon calculus at the University of Montpellier in 2015, and the second at the Humboldt Kolleg workshop on the mathesis universalis held in Como in 2017. These events were notable in that they brought together researchers from mathematics, computer science and philosophy, and as a consequence, speakers were faced with the challenge of making their ideas appealing and comprehensible to each of these groups.

The latter workshop was particularly fascinating in its focus on the *mathesis universalis*: The search for a universal mode of thought, or language, capable of capturing and connecting ideas from all of the sciences. It struck me then that part of my research has been driven, in a certain sense, by the desire to uncover universal characteristics behind the central objects of my own field of study - proof interpretations - and in particular to conceive of a language in which the algorithmic ideas which underlie them can be elegantly expressed. My initial motivation when writing this article was to simply collect some of my own thoughts in this direction and put them down on paper.

In doing so, however, I realised that an article of this kind could also serve another purpose: To give an accessible and high level overview of a number of proof interpretations which play a central role in the history of logic but are often poorly understood outside of specialist areas of application. Proof interpretations are, by their very nature, highly syntactic objects, and gaining an understanding of how they actually work when applied to a concrete proof tends to require a certain amount of hands-on experience. But perhaps a simple

and informal case study tackled in parallel by several proof interpretations could form an accessible and insightful introduction to these techniques which would complement the many excellent textbooks on the subject?

I settled on an article which combines both of these goals. In the first main section I give an overview of three very well-known computational interpretations of classical logic: Hilbert's epsilon calculus, Kreisel's modified realizability together with the A translation, and finally Gödel's Dialectica interpretation. My tactic is to suppress many of the complicated definitions and to focus instead on a very simple example - the drinkers paradox - and sketch how each interpretation would deal with it in turn. In doing so I am able to highlight some of the key features which characterise each of the interpretations, but are often invisible until one has acquired a working knowledge of how they are applied in practise.

In the second part, I turn my attention to the relationship between the interpretations and core algorithmic ideas which underlie them. I begin with a general discussion on proof interpretations and the roles they play in modern proof theory, then focus on two ideas which feature in current research on program extraction: learning procedures and stateful programs. Both of these have been used by myself and others to characterise algorithms connected to classical reasoning, and together offer an illustration of how traditional proof theoretic techniques can be reinterpreted in a modern setting. Here I focus primarily on the Dialectica interpretation, and continue to use the drinkers paradox as a running example, in order to facilitate a direct comparison with the earlier section.

Prerequisites

I have sought to capture the spirit of the *mathesis universalis* in another way, by making this article comprehensible for as general an audience as possible. To this end, I assume little more than an acquaintance with first-order logic and formal reasoning, though I expect a passing familiarity with the typed lambda calculus will make what follows a lot easier to read. Just in case, I mention a few crucial things here. The *finite types* are defined inductively by the following clauses:

- $\mathbb{N}$ is a type;
- if ρ and τ are types, $\rho \to \tau$ is the type of functions from ρ to τ.

Often is it convenient to talk about product types $\rho \times \tau$ in addition. Functions which take functions as an argument are called *functionals*.

System T is the well-known calculus of primitive recursive functionals in all finite types, whose terms consist of variables $x^\rho, y^\rho, z^\rho, \ldots$ for each type, symbols for zero $0 : \mathbb{N}$ and successor $s : \mathbb{N} \to \mathbb{N}$, allow the construction of new terms via lambda abstraction $\lambda x.t$ and application $t(s)$, and finally, contain recursors R_ρ of each type, which allow the definition of primitive recursive functionals. Further details on System T can be found in e.g. [7].

A note on terminology

Proof interpretations are often inconsistently named and confused with one another. In this paper, I will use the term *functional interpretation* to denote any proof interpretation which maps proofs to functionals of higher-type: Thus both realizability and the Dialectica interpretation are functional interpretations. The latter is also commonly referred to as Gödel's functional interpretation, and consequently often as just 'the functional interpretation'. I am as guilty as anyone for propagating such confusion elsewhere, but here, since I discuss a number of proof interpretations, I will rigidly stick to the name *Dialectica*.

2 The drinkers paradox

The running example throughout this paper will be a simple theorem of classical predicate logic, commonly known as the *drinkers paradox*. This nickname apparently originates with Smullyan [49], and refers to the following popular formulation of the theorem in natural language:

> In any pub, there is someone that, if they are drinking, then everyone in the pub is drinking.

Of course, in pure logical terms, the drinkers paradox is nothing more than the simple first-order formula

$$\exists n(P(n) \to \forall m P(m))$$

where here, since we will primarily work in the setting of classical arithmetic, P is assumed to be some decidable predicate over the natural numbers $\mathbb{N}$. In fact, to make things slightly simpler for the functional interpretations (and more interesting for the epsilon calculus) we will study the following prenexation of the drinkers paradox:

$$\mathrm{DP} \quad : \quad \exists n \forall m(P(n) \to P(m)).$$

which will henceforth be labelled DP.

The drinkers paradox is appealing for the proof theorist because it has a one-line proof in classical logic which doesn't give us any way to actually compute the 'canonical drinker' n. It goes as follows: Either everyone in the pub is drinking, in which case we can set $n := 0$, or there is at least one person m who is not drinking, in which case we set $n := m$.

As such, any computational interpretation of classical logic has to have some way of dealing with the drinkers paradox, and as we will see, despite its apparent simplicity, DP illuminates several key features of the interpretations, which is precisely why it makes such a convenient working example.

Before moving on, a rather compact, semi-formal derivation of DP in a Hilbert-style calculus is provided in Figure 1 - semi-formal because several of our inferences conflate a number of steps. Though we will refrain from carrying out any formal manipulations on proof trees, this will be used later to provide

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\neg P(k) \to P(k) \to P(m)}{\neg P(k) \to \forall m(P(k) \to P(m))}\,\forall\mathrm{r}}{\neg P(k) \to \exists n \forall m(P(n) \to P(m))}\,\exists\mathrm{ax}}{\exists k \neg P(k) \to \exists n \forall m(P(n) \to P(m))}\,\exists\mathrm{r} \qquad \dfrac{\dfrac{\dfrac{P(m) \to P(0) \to P(m)}{\forall k P(k) \to P(0) \to P(m)}\,\forall\mathrm{ax}}{\forall k P(k) \to \forall m(P(0) \to P(m))}\,\forall\mathrm{r}}{\forall k P(k) \to \exists n \forall m(P(n) \to P(m))}\,\exists\mathrm{ax}}{\exists k \neg P(k) \lor \forall k P(k) \to \exists n \forall m(P(n) \to P(m)) \lor \exists n \forall m(P(n) \to P(m))}\,(*)}{\exists n \forall m(P(n) \to P(m)) \lor \exists n \forall m(P(n) \to P(m))}\,\mathrm{LEM}}{\exists n \forall m(P(n) \to P(m))}\,\mathrm{ctr}
$$

Figure 1: An informal Hilbert-style derivation of DP

some insight into how the different techniques act on the proof in order to extract witnesses. Here $\forall$ax and $\forall$r refer to instances of the $\forall$-axiom and rule respectively (and similarly for the existential quantifier), while LEM denotes an instance of the law of excluded-middle and ctr an instance of contraction. The inference $(*)$ combines several steps which will typically not be relevant from a computational point of view.

3 Three famous interpretations

In the first main part of this article, I give what is intended to be an accessible outline of three of the most famous computational interpretations of classical logic: Hilbert's epsilon calculus, Kreisel's modified realizability and Gödel's Dialectica interpretation. The reader who wants to see a full exposition (and indeed full definitions) of these interpretations is encouraged to consult one of the many references I will provide on the way. My aim here is *not* to give a comprehensive or detailed introduction, but to offer a case study which I hope will not only provide some insight into how the interpretations work in practise, but will hint at the deep connections between each of them.

It is important to point out that by focusing on just three interpretations, I inevitably exclude several significant approaches to program extraction, notably those originating from the French tradition such as Krivine's classical realizability [29]. This a reflection of the fact that I am somewhat familiar with the former, and much less so with the latter. Moreover, each of the interpretations I mention have a certain historical and philosophical significance, and feature in research outside of mathematical logic, which I feel also justifies my choice.

A note on precision

Proof interpretations are formal maps on derivations, and as such, act in a very precise way on our proof of the drinkers paradox. Here, I give somewhat *imprecise* versions of the formal extraction procedure, because I want to avoid superfluous bureaucratic details as much as possible. My emphasis here is on the salient features of the interpretations. A good example of this attitude

is my cavalier approach to negative translations, which technically speaking have to be carefully specified and are, for example, sensitive to whether they are embeddings into intuitionistic or minimal logic. For the sake of clarity I skirt such issues here and only mention them in passing. Naturally, the reader curious about these subtleties will find a full explanation in the standard texts, and detailed references will be given here.

3.1 The epsilon calculus

Hilbert's epsilon calculus, developed in a series of lectures in the early 1920s, constitutes one of the first attempts at grounding classical first-order theories in a computational setting. Much like the Dialectica interpretation, it was first presented as a technique for proving the consistency of arithmetic, in which it comes hand-in-hand with the corresponding substitution method. For a more detailed introduction to the epsilon calculus itself, together with an extensive list of references, the reader is directed to [8].

While the epsilon calculus remains a topic of interest in mathematical logic, philosophy and linguistics, concrete applications in mathematics or computer science remain limited in comparison to the interpretations that follow - a phenomenon which we discuss in Section 4. Nevertheless, of the three interpretations we present in this section, the epsilon calculus is perhaps unrivalled in the simplicity and elegance of its main idea: To first bring predicate logic down to the propositional level by replacing all quantifier instances by a piece of syntax called an *epsilon term*, and to then systematically eliminate these terms via a backtracking algorithm, thus leaving us in a finitary system in which existential statements are assigned explicit realizers.

Let's look at this idea more closely. Very informally, the epsilon calculus assigns to each predicate $A(x)$ an epsilon term $\epsilon x A(x)$, whose intended interpretation is a choice function which selects a witness for $\exists x A(x)$ whenever the latter is true. The idea is then to *replace* the quantified formula $\exists x A(x)$ with $A(\epsilon x A(x))$, since under this interpretation these two formulas would be equivalent. The universal quantifier is dealt with similarly: bearing in mind that $\forall x A(x) \leftrightarrow \neg\exists x \neg A(x)$ over classical logic, we interpret $\forall x A(x)$ as $A(\epsilon x \neg A(x))$, since $\neg A(\epsilon x \neg A(x))$ holds only if $\neg\forall x A(x)$.

We now want to transform proofs in predicate logic to proofs in the epsilon calculus. So what happens if we replace all instances of quantifiers in a proof with the corresponding epsilon term? It turns out that the quantifier rules are trivially eliminated with epsilon terms in place of quantifiers: If

$$\dfrac{\begin{array}{c}\vdots \\ A(x) \to B\end{array}}{\exists x A(x) \to B}$$

occurs in our proof then we simply replace the free variable x with $\epsilon x A(x)$ and we get a derivation

$$\vdots$$
$$A(\epsilon x A(x)) \to B.$$

On the other hand, if we have an instance of the quantifier *axiom*

$$A(t) \to \exists x A(x)$$

we now need to introduce a new axiom which governs the corresponding epsilon term, namely:

$$(*) \quad A(t) \to A(\epsilon x A(x)).$$

Axioms of the form (*) are called *critical axioms*. In simple terms, the epsilon calculus is the quantifier-free calculus we obtain by removing quantifiers and their axioms and rules, and adding epsilon terms together with the critical axioms. Note that for first-order theories like arithmetic things are already a bit more complicated, but for now we have enough structure to give an epsilon-style proof of the drinkers paradox!

Referring to our derivation in Figure 1, let us use the following abbreviations:

$$\begin{aligned}\epsilon_k &:= \epsilon k \neg P(k) \\ \epsilon_m[n] &:= \epsilon m \neg (P(n) \to P(m)) \\ \epsilon_n &:= \epsilon n (P(n) \to P(\epsilon_m[n]))\end{aligned}$$

whose meaning in the epsilon calculus is indicated below:

$$\begin{aligned}P(\epsilon_k) &\leftrightarrow \exists k \neg P(k) \\ (P(n) \to P(\epsilon_m[n])) &\leftrightarrow \forall m (P(n) \to P(m)) \\ P(\epsilon_n) \to P(\epsilon_m[\epsilon_n]) &\leftrightarrow \exists n \forall m (P(n) \to P(m))\end{aligned}$$

Note that the final formula is the drinkers paradox itself.

Figure 2 shows the translation of our first-order derivation of DP into one in the epsilon calculus. The translation is quite straightforward, and uses nothing more than the principles sketched above: For each quantifier rule we simply replace the variable in question by the relevant epsilon term (and the rule vanishes), whereas each quantifier axiom is interpreted by the corresponding critical axiom, which is made explicit in Figure 2. The proof is shorter due to the elimination of the quantifier rules, and the instance of LEM now applies to the decidable predicate $P(\epsilon_k)$, and so is now computationally benign.

So far so good: We have transformed a proof in classical predicate logic to a quantifier-free version in which the main logical content is now encoded in the critical axioms. Can we get rid of these in some way in order to give the proof a computational interpretation?

This is the role played by epsilon substitution, which forms the core of the epsilon calculus technique. In short: Any first-order proof can only use finitely many instances of the critical axioms. Therefore, if for all relevant formulas $A(x)$ which feature in the proof, we can find a *concrete approximation* for the epsilon term $\epsilon x A(x)$, which rather than satisfying *all* critical axioms $A(t) \to A(\epsilon x A(x))$,

$$
\dfrac{\dfrac{\dfrac{\dfrac{\neg P(\epsilon_k) \to P(\epsilon_k) \to P(\epsilon_m[\epsilon_k])}{\neg P(\epsilon_k) \to (P(\epsilon_n) \to P(\epsilon_m[\epsilon_n]))}\,\text{C2} \qquad \dfrac{\dfrac{P(\epsilon_k) \to P(0) \to P(\epsilon_k)}{P(\epsilon_k) \to (P(0) \to P(\epsilon_m[0]))}\,\text{C1}}{P(\epsilon_k) \to (P(\epsilon_n) \to P(\epsilon_m[\epsilon_n]))}\,\text{C3}}{\neg P(\epsilon_k) \vee P(\epsilon_k) \to (P(\epsilon_n) \to P(\epsilon_m[\epsilon_n])) \vee (P(\epsilon_n) \to P(\epsilon_m[\epsilon_n]))}}{(P(\epsilon_n) \to P(\epsilon_m[\epsilon_n])) \vee (P(\epsilon_n) \to P(\epsilon_m[\epsilon_n]))}\,\text{LEM}}{P(\epsilon_n) \to P(\epsilon_m[\epsilon_n])}\,\text{ctr}
$$

Critical axioms:

C1 $P(\epsilon_k) \to P(\epsilon_m[0])$

C2 $(P(\epsilon_k) \to P(\epsilon_m[\epsilon_k])) \to (P(\epsilon_n) \to P(\epsilon_m[\epsilon_n]))$

C3 $(P(0) \to P(\epsilon_m[0])) \to (P(\epsilon_n) \to P(\epsilon_m[\epsilon_n]))$

Figure 2: An informal epsilon-style derivation of DP

satisfy just those which are used in the proof, then we have a way of finding explicit witnesses for existential statements. The epsilon substitution method is an algorithm for eliminating critical formulas in this way.

It goes as follows: We first assign all relevant epsilon terms the canonical value 0. We then examine our finite list of critical formulas until we find one which fails. This would mean that there is some term formula A and term t such that $A(t)$ is true but $A(\epsilon x A(x))$ false under our current assignment. But we can now use the this information to repair our epsilon term, by setting $\epsilon x A(x) := t$, which from now on serves as a witness for $\exists x A(x)$. The process is then repeated, until all critical formulas have been repaired.

This may sound quite straightforward, but epsilon substitution is in fact a complicated business! In the case of arithmetic, several erroneous algorithms were initially proposed, before a correct technique involving transfinite induction up to ε_0 was finally given by Ackermann in [2] - a modern treatment of which is provided by Moser [33]. The difficulty with the substitution method is primarily due to the presence of *nested* epsilon terms, whereby trying to fix one critical axiom can suddenly invalidate several others which were previously considered fixed, leading to a subtle backtracking procedure. Further details of the general algorithm are far beyond the scope of this paper, but we *are* able to provide a simple version for the case of the drinkers paradox!

First we should remark that in our proof, only two of our epsilon terms, ϵ_k and ϵ_n, represent existential (as opposed to universal) quantifiers, so we only attempt to find approximations for these (we explain this in more detail below). Let us first try $\epsilon_k = \epsilon_n = 0$. Then C2 and C3 are trivially satisfied, but not necessarily C1. In this case there are two possibilities: Either we get lucky and $P(0) \to P(\epsilon_m[0])$ holds, in which case we're done, or our guess fails because $P(0) \wedge \neg P(\epsilon_m[0])$. We now learn from our failure and repair the broken epsilon terms, setting $\epsilon_k := \epsilon_m[0]$. Now C1 holds and C3 remains the same as before,

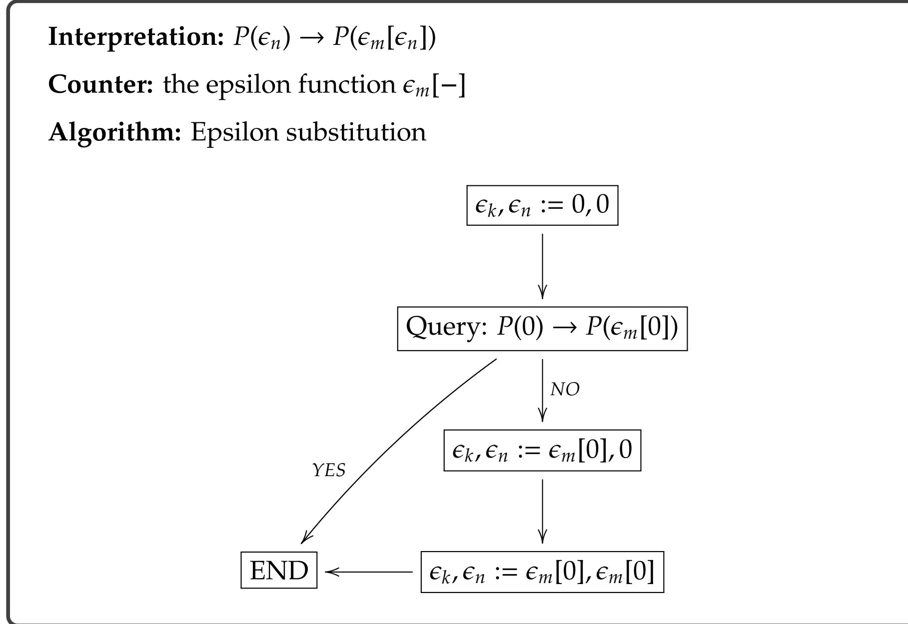

Figure 3: Interpretation of DP via epsilon calculus

but under our new assignment C2 becomes

$$(P(\epsilon_m[0]) \to P(\epsilon_m[\epsilon_m[0]])) \to (P(0) \to P(\epsilon_m[0]))$$

which from our assumption $P(0) \wedge \neg P(\epsilon_m[0])$ is now false. So we repair this in turn and set $\epsilon_n := \epsilon_m[0]$. A quick run through of each critical axiom under the assignment $\epsilon_k = \epsilon_n := \epsilon_m[0]$ reveals that we're done.

The substitution method generalises this strategy of guessing, learning from the failure of our guesses, and repairing. As such, despite its complexity, it offers a beautifully clear computational semantics for classical logic: The building of approximations to non-computational objects via a game of trial and error. This has inspired a number of more modern approaches to program extraction, which we will mention later.

As we already pointed out, there are no critical formulas for $\epsilon_m[n]$. Rather, this plays the role of a function variable, and our witness for ϵ_n is constructed relative to this variable. Without wanting to go into more detail here, $\epsilon_m[n]$ is a placeholder which represents how the drinkers paradox might be used as a lemma in the proof of another theorem. As such, we do not have an ideal witness for $\exists n$ in the drinkers paradox, but rather an approximate witness relative to $\epsilon_m[n]$. We will see this phenomenon repeat itself throughout this paper, where in each setting there will be a specific structure which plays the role of a 'counter argument', which represents the universal quantifier and against which our approximate witness is computed.

We summarise our epsilon-style interpretation of DP in Figure 3, giving a flowchart representation of the corresponding substitution algorithm.

3.2 Modified realizability

Before we move on to the world of realizability, let us consider the following elegant interpretation of DP as a game between two players, who are traditionally named ∃loise and ∀belard. ∃loise's goal is to establish the truth of DP by choosing a witness for $\exists n$, whereas her opponent ∀belard is tasked with disproving DP by producing a counter witness for $\forall m$. Fortunately for ∃loise, the rules of the game dictate that she can backtrack and change her mind!

So how might the game go? ∃loise begins by picking an arbitrary witness for $\exists n$, let's say $n := 0$, thereby claiming that $\forall m(P(0) \to P(m))$ is true. ∀belard responds by playing some m with the aim of showing that ∃loise's guess was wrong. There are now two possibilities: Either $P(0) \to P(m)$ is true and ∃loise was right all along, in which case she wins, or ∀abelard's challenge was successful and $P(0) \wedge \neg P(m)$ holds. But now ∃loise responds by simply changing her mind and playing $n := m$. Now any further play m' from ∀belard is destined to fail since $P(m) \to P(m')$ will always be true!

What we have done is describe a winning strategy for ∃loise in a quantifier-game corresponding to DP. That there is a correspondence between classical validity and winning strategies in this sense is well-known and widely researched, dating back to e.g. Novikoff in [35], and today game semantics is an important topic in logic. Here it will form a useful sub-theme and will help us characterise the behaviour of certain functionals which arise from classical proofs.

Modified realizability is one of several forms of realizability which arose as a concrete implementation of the Brouwer-Heyting-Kolmgorov (BHK) interpretation of intuitionistic logic. Introduced by Kreisel in [28], modified realizability works in a *typed* setting, and allows us to transform proofs in intuitionistic arithmetic to terms in the typed lambda calculus.

As a clean and elegant formulation of the BHK interpretation in the typed setting, variants of modified realizability have proven extremely popular techniques for extracting programs from proofs. In particular, refinements of the interpretation form the theoretical basis for the Minlog system [1], a proof assistant which automates program extraction and is primarily motivated by the synthesis of verified programs (see e.g. [11, 12] for examples of this). For a comprehensive account of the interpretation itself, the reader is directed to e.g. [23, Chapter 5] or [48, Chapter 7].

As with the epsilon calculus, modified realizability consists of two components: An interpretation and a soundness proof. The interpretation maps each formula A in Heyting arithmetic to a new formula $x \text{ mr } A$, where x is a potentially empty tuple of variables whose length and type depends on the structure of A. The interpretation is quite simple, so we state it in Figure 4.

The soundness proof for modified realizability states that whenever A is provable in Heyting arithmetic, then there is some term t of System T (the

$$
\begin{aligned}
x \text{ mr } A &:\equiv A \text{ for prime formulas } A\\
x, y \text{ mr } A \wedge B &:\equiv x \text{ mr } A \wedge y \text{ mr } B\\
b, x, y, \text{ mr } A \vee B &:\equiv (b = 0 \to x \text{ mr } A) \wedge (b \neq 0 \to y \text{ mr } B)\\
f \text{ mr } A \to B &:\equiv \forall x (x \text{ mr } A \to fx \text{ mr } B)\\
z, y \text{ mr } \exists x A(x) &\equiv y \text{ mr } A(z)\\
f \text{ mr } \forall x A(x) &\equiv \forall x (fx \text{ mr } A(x))
\end{aligned}
$$

Figure 4: Modified realizability

typed lambda calculus of primitive recursive functionals in all finite types) satisfying t mr A, which can moreover be formally extracted from the proof:

$$\text{HA} \vdash A \quad \text{implies} \quad \text{System T} \vdash t \text{ mr } A.$$

So far, what we have described applies only to intuitionistic theories, so we need an additional step to extend the interpretation to classical logic. This turns out to be rather subtle, and usually relies on a combination of the Gödel-Gentzen negative translation together with some variant of the so-called Dragalin/Friedman/Leivant trick, also known as the A-translation. A detailed discussion of this technique is beyond the scope of this paper, but we briefly present a variant due to [13] and describe how it acts on DP.

First the negative translation. Negative translations are well-known methods for embedding classical logic in intuitionistic logic. There are several variants of the translation, each of which involves strategically inserting double negations in certain places in a logical formula in such a way that if A is provable in classical logic, then A^N is provable intuitionistically. In particular, we would have

$$\text{PA} \vdash A \quad \text{implies} \quad \text{HA} \vdash A^N.$$

Negative translations are widely used and and the relationship between the many varieties is well understood (see [18] for a detailed study). We deliberately avoid going into more detail and specifying a translation to use here. We simply state without justification that a suitable negative translation of the drinkers paradox is the following:

$$\text{DP}^N :\equiv \neg\neg\exists n \forall m (P(n) \to \neg\neg P(m))$$

which is provable in intuitionistic (and in fact minimal) logic. Now to the A-translation. The reason we need an intermediate step in addition to the negative translation is that the negative translation alone results in a formula with no computational meaning at all: Since $\perp$ is a prime formula, we have

$$f \text{ mr } \neg A \equiv \forall x (x \text{ mr } A \to \perp)$$

i.e. f is just an empty tuple. In order to make realizability 'sensitive' to the negative translation, we treat $\perp$ as a new predicate, which in particular has a

special realizability interpretation

$$x \text{ mr } \bot$$

where x has some predetermined type τ. With this adjustment we would have

$$f \text{ mr } \neg A \equiv \forall x(x \text{ mr } A \to fx \text{ mr } \bot)$$

where fx has the non-empty type τ. Now that realizability interacts with negated formulas, we are ready to give the drinkers paradox a computational interpretation. The first step is to examine the negated principle DP^N, and carefully apply the clauses of Figure 4 together with the special interpretation of $\bot$.

We now do this in detail. We first assume that the decidable predicate $P(n)$ can be coded as a prime formula $t(n) = 0$ and thus has an empty realizer. So starting with the inner formula it is not to difficult to work out that

$$f \text{ mr } (P(n) \to \neg\neg P(m)) \equiv P(n) \to \forall a((P(m) \to a \text{ mr } \bot) \to fa \text{ mr } \bot)$$

where $f : \tau \to \tau$. The next step is to treat the quantifiers, and we end up with

$$\begin{aligned} & e, g \text{ mr } \exists n \forall m(P(n) \to \neg\neg P(m)) \\ & \equiv \forall m(gm \text{ mr } (P(e) \to \neg\neg P(m))) \\ & \equiv \underbrace{\forall m(P(e) \to \forall a((P(m) \to a \text{ mr } \bot) \to gma \text{ mr } \bot))}_{Q(e,g)} \end{aligned}$$

where for convenience we label this formula $Q(e, g)$ as indicated. Note that types of these realizers are given by $e : \mathbb{N}$ and $g : \mathbb{N} \to \tau \to \tau$. Finally, interpreting the whole formula yields

$$\begin{aligned} & \Phi \text{ mr } \neg\neg\exists n \forall m(P(n) \to \neg\neg P(m)) \\ & \equiv \forall p(p \text{ mr } \neg\exists n \forall m(P(n) \to \neg\neg P(m)) \to \Phi p \text{ mr } \bot) \qquad (1) \\ & \equiv \forall p(\forall e, g(Q(e, g) \to peg \text{ mr } \bot) \to \Phi p \text{ mr } \bot) \end{aligned}$$

where now $p : \sigma$ and $\Phi : \sigma \to \tau$ for $\sigma := \mathbb{N} \to (\mathbb{N} \to \tau \to \tau) \to \tau$.

Just as we did not give a formal description of the epsilon substitution method, we will not present in detail how a concrete functional Φ satisfying the above is rigorously extracted from the proof of the negative translation of DP. Rather, we simply present a term which does the trick and carefully explain why. Let's define

$$\Phi p := p0h_p \quad \text{where} \quad h_p := \lambda m^{\mathbb{N}}, a^{\tau} \left(a \text{ if } P(0) \to P(m) \text{ else } pm(\lambda n^{\mathbb{N}}, b^{\tau}.b)\right).$$

We need to prove that Φ satisfies (1) above. In order to do this, we have to simultaneously unwind (1) together with the definition of Φ. Our goal is to show that $\Phi p \text{ mr } \bot$ whenever p satisfies the premise of (1). So let's assume the latter, i.e.

$$\forall e, g(Q(e, g) \to peg \text{ mr } \bot) \qquad (2)$$

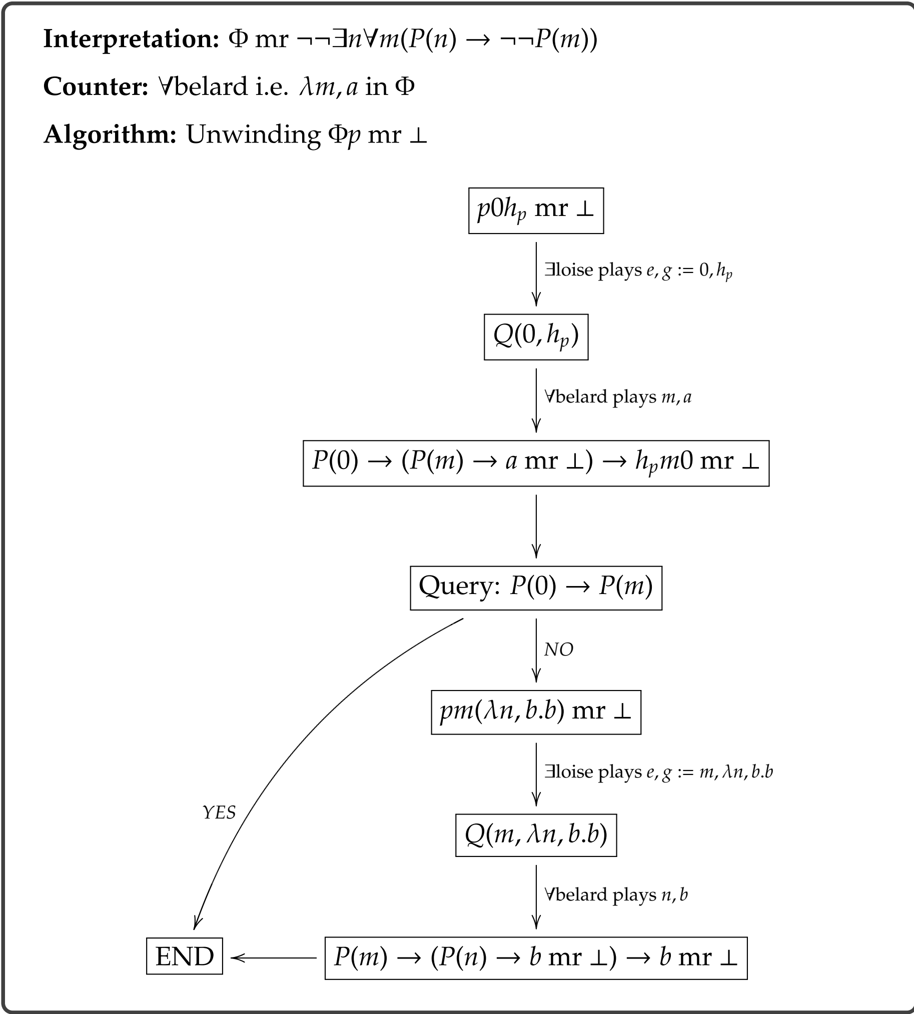

Figure 5: Modified realizability interpretation of DP

and first instantiate $e, g := 0, h_p$, which gives us

$$Q(0, h_p) \to \underbrace{p0h_p}_{=\Phi p} \text{ mr } \bot$$

It is now enough to show that $Q(0, h_p)$ holds, since then we have Φp mr $\bot$ as indicated. Referring back to the definition of $Q(e, g)$ above, this in turn boils down to showing that, for any m, a:

$$P(0) \to (P(m) \to a \text{ mr } \bot) \to h_p ma \text{ mr } \bot. \tag{3}$$

There are now two cases. In the first, $P(0) \to P(m)$ holds, which means that $h_p ma = a$ and so (3) becomes

$$P(0) \to (P(m) \to a \text{ mr } \bot) \to a \text{ mr } \bot.$$

This is now true, since either $P(0)$ is false and it follows trivially or $P(m)$ holds, and so a mr $\bot$ follows from the assumption $P(m) \to a$ mr $\bot$.

In the second case we have $P(0) \wedge \neg P(m)$, which means that $h_p ma = pm(\lambda n, b.n)$ and (3) becomes

$$P(0) \to (P(m) \to a \text{ mr } \bot) \to pm(\lambda n, b.b) \text{ mr } \bot,$$

which appears less simple to validate. But the trick is to now go back to our main assumption (2), this time instantiating $e, g := m, \lambda n, b.n$. We are now done so long as $Q(m, \lambda n, b.b)$ holds, which written out properly is

$$P(m) \to (P(n) \to b \text{ mr } \bot) \to b \text{ mr } \bot.$$

But this now trivially true for any n, b since by assumption we have $\neg P(m)$! Therefore climbing back up: We have established $pm(\lambda n, b.b)$ mr $\bot$ and hence $h_p ma$ mr $\bot$, which in turn proves $Q(0, h_p)$ and thus $p0h_p$ mr $\bot$.

At first glance, the verification of our realizer looks somewhat convoluted, jumping back and forth between our various hypothesis until one by one they have been discharged. However, things become much clearer if we visualise the above argument in terms of the game sketched at the beginning of the section.

Let's look at it again, this time giving each step a game-theoretic reading. In order to verify Φp mr $\bot$ we set $e, g := 0, h_p$ in (2), which can be seen as a first attempt by ∃loise to prove DP. In this context, ∀belard's job is to disprove $Q(0, h_p)$, which he does by choosing some m, a and hoping that (3) fails. Either $P(0) \to P(m)$ is true, in which case (3) holds and ∃loise wins, or ∀belard chose a good counterexample and $P(0) \wedge \neg P(m)$. ∃loise now appeals to the premise of (2) a second time, setting $e, g := m, \lambda n, b.n$, and wins unless $Q(m, \lambda n, b.b)$ can be shown false. But under our assumption $\neg P(m)$ this is impossible, and so ∃loise has a winning strategy.

What is interesting here is how the winning strategy is encoded by the components of our term Φ: ∃loise's moves are arguments for the term p, whereas ∀belard's move is represented by the internal function abstraction $\lambda m, a$.

The reader interested in exploring the the connection between realizability, negative translations and games could take [16] as a starting point. A fascinating illustration of this phenomenon in the case of the axiom of choice is provided by [9], which also inspired my own work in [44].

We finish off this section, as before, with a diagrammatic representation of the algorithm which underlies this realizability interpretation of DP. Note that where for the epsilon calculus the epsilon function $\epsilon_m[-]$ played the role of the 'counter', here the same thing is represented by the player ∀belard, or more precisely, the function abstraction within our term Φ.

3.3 Dialectica interpretation

We now come to our final computational interpretation of classical logic: Gödel's functional, or Dialectica interpretation. The name 'Dialectica' refers to the journal in which the interpretation was first published in 1958 [20], though the interpretation had already been conceived the 1930s and presented in lectures from the early 1940s onwards. An English translation of the original article can be found in Volume II of the *Collected Works* [21], which is preceded by an illuminating introduction by Troelstra.

The original purpose of the Dialectica interpretation was to produce a relative consistency proof for Peano arithmetic. The interpretation maps the first-order theory of arithmetic to the primitive recursive functionals in finite types, thereby showing that the consistency of the former follows from that of the latter. The interpretation was soon extended to full classical analysis by Spector, in another groundbreaking article [50], which is also a fascinating read due to the extensive footnotes from Kreisel, who put together and completed the paper following Spector's early death in 1961.

Much like the epsilon calculus, then, the Dialectica interpretation has its origins in Hilbert's program and the problem of consistency, though in contrast it has achieved great success as a tool in modern applied proof theory, being central to the proof mining program initiated by Kreisel in the 1960s and brought to maturity by Kohlenbach from the 1990s.

For a comprehensive introduction to the interpretation the reader is directed to the Avigad and Feferman's *Handbook* chapter [7] or the textbook of Kohlenbach [23], particularly Chapters 8-10. The latter is also the standard reference for *applications* of the Dialectica interpretation in mathematics (and applied proof theory in general).

The basic set up of the interpretation bears a close resemblance to modified realizability, although there are crucial differences between the two. The Dialectica assigns to each formula A of Heyting arithmetic a new formula of the form $\exists x \forall y |A|^x_y$, where $|A|^x_y$ is quantifier-free and x and y are tuples of variables whose length and type depend on the structure of A. The inner part of the interpretation is defined by induction on formulas, which we give in Figure 6 below.

When one first sees the definition of the Dialectica interpretation, it is immediate that it departs from the usual BHK style on which realizability is based,

$$
\begin{aligned}
|A|^x_y &:\equiv A \text{ for prime formulas } A \\
|A \wedge B|^{x,u}_{y,v} &:\equiv |A|^x_y \wedge |B|^u_v \\
|A \vee B|^{b^{\mathbb{N}},x,u}_{y,v} &:\equiv (b = 0 \to |A|^x_y) \wedge (b \neq 0 \to |B|^u_v) \\
|A \to B|^{f,g}_{x,v} &:\equiv |A|^x_{gxv} \to |B|^{fx}_v \\
|\exists z A(z)|^{z,x}_y &:\equiv |A(z)|^x_y \\
|\forall z A(z)|^f_{z,y} &:\equiv |A(z)|^{fz}_y
\end{aligned}
$$

Figure 6: The Dialectica interpretation

and the interpretation of implication in particular seems rather ad-hoc until one understands where it comes from! The point here is that Dialectica interpretation acts as a kind of Skomelisation, pulling all the quantifiers to the front of the formula but preserving logical validity, so that

$$A \leftrightarrow \exists x \forall y |A|^x_y$$

provably in the usual higher-type formulation of classical logic plus a quantifier-free form of choice.

For most of the logical connectives, there is one obvious way of defining the interpretation. However, in the case of implication we have several ways of pulling the quantifiers out to the front, which make use of classical logic to a greater or lesser degree. Gödel chose the option which Skolemises implication in the 'least non-constructive way', using only Markov's principle and a weak independence of premise. A detailed explanation of all this is given in [7, Section 2.3] and [23, pp. 127–129]. I highlight it here because the treatment of implication can often seem mystifying until one sees that it is defined the way it is for a reason!

As with modified realizability, the Dialectica interpretation comes equipped with a soundness proof: If A is provable in Heyting arithmetic then there is some term t of System T satisfying $\forall y |A|^t_y$, and moreover, this term can be extracted from the proof of A:

$$\text{HA} \vdash A \quad \text{implies} \quad \text{System T} \vdash \forall y |A|^t_y.$$

We are now in a similar situation to the last section: We have a computational interpretation of intuitionistic arithmetic, which we now need to extend in some way to deal with classical logic. It turns out that the same trick works: We just combine the Dialectica interpretation with the negative translation. But crucially, for the Dialectica we do not need an intermediate step corresponding to the A-translation, since negated formulas are already considered computational thanks to the interpretation of implication: We have

$$\neg A \text{ is interpreted as } \exists g \forall x |A \to \bot|^g_x \equiv \exists g \forall x (|A|^x_{gx} \to \bot) \tag{4}$$

On top of this, the Dialectica allows us to simplify our negative translated formulas considerably. If A is a prime formula then $\neg\neg A$ is interpreted as itself, and in the case of arithmetic $\neg\neg A$ is intuitionistically equivalent to A. Therefore when interpreting a complicated formula we can continually remove inner double negations which apply only to quantifier-free inner parts, which saves us a lot of effort!

The reader may wonder why we didn't go ahead and do this in the case of modified realizability. This is another consequence of not needing to treat $\bot$ as a special symbol: The variant of the A translation we gave uses implicitly that DP^N is provable in *minimal* logic, in which case we can replace $\bot$ with anything and the proof still goes through. However, the removal of double negations before quantifier-free formulas requires ex-falso quodlibet and hence the simplified form of DP^N we will consider below cannot be dealt with by the A translation. For more on this rather subtle point see [23, Chapter 14].

So let's now focus on how the Dialectica interpretation deals with DP, or more specifically, its negative translation, we which already gave in the previous section as

$$\neg\neg\exists n\forall m(P(n) \to \neg\neg P(m)).$$

As discussed above, the first remarkable difference with realizability is that the inner double negation essentially vanishes. Since $P(n) \to \neg\neg P(m)$ is quantifier-free, it can be encoded as a prime formula and we have

$$(P(n) \to \neg\neg P(m)) \text{ is interpreted as } (P(n) \to \neg\neg P(m)) \leftrightarrow (P(n) \to P(m))$$

since in Heyting arithmetic $\neg\neg P(m) \leftrightarrow P(m)$. Therefore, looking at how the Dialectica interprets quantifiers and removing our inner double negation in this way, we have that

$$\exists n\forall m(P(n) \to \neg\neg P(m)) \text{ is interpreted as } \exists n\forall m(P(n) \to P(m))$$

Therefore in order to deal with DP^N, we need to interpret a formula of the form $\neg\neg B$ where $B :\equiv \exists n\forall m(P(n) \to P(m))$ is an $\exists\forall$ formula. It's here that the interpretation of implication comes into play. To interpret the outer negations, let's first look at $\neg B$. Following (4), this is interpreted as

$$\exists g\forall n\neg(P(n) \to P(gn))$$

and therefore negating a second time and removing the inner double negations, the interpretation of $\neg\neg B$, and hence the Dialectica interpretation of the drinkers paradox, is given by

$$\exists\Phi\forall g(P(\Phi g) \to P(g(\Phi g))). \tag{5}$$

Our challenge is to find a functional Φ which satisfies (5). It turns out that such a functional is a lot easier to write down than that for modified realizability, and so this time we want to hint - very informally nevertheless - at how such a term can be extracted from the semi-formal proof given in Figure 1, as it highlights some important features of the *soundness* of the Dialectica interpretation.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\neg P(k) \to P(k) \to P(m)}{\forall m(\neg P(k) \to P(k) \to P(m))}\ \forall\mathrm{r}}{\forall g(\neg P(k) \to P(k) \to P(gk))}\ \exists\mathrm{ax}}{\forall k, g(\neg P(k) \to P(k) \to P(gk))}\ \exists\mathrm{r} \qquad \dfrac{\dfrac{\dfrac{P(m) \to P(0) \to P(m)}{P(m) \to P(0) \to P(m)}\ \forall\mathrm{ax}}{\forall m(P(m) \to P(0) \to P(m))}\ \forall\mathrm{r}}{\forall g(P(g0) \to P(0) \to P(g0))}\ \exists\mathrm{ax}}{\dfrac{\dfrac{\forall g(\neg P(g0) \vee P(g0) \to (P(g0) \to P(g(g0))) \vee (P(0) \to P(g0)))}{\forall g((P(g0) \to P(g(g0))) \vee (P(0) \to P(g0)))}\ \mathrm{LEM}}{\forall g(P(\Phi g) \to P(g(\Phi g)))}\ \mathrm{ctr}}
$$

Figure 7: An informal extraction of Φ

The main idea is to extract our functional Φ recursively over the structure of the proof, a process which could be roughly described as follows: There are two main branches of the proof, which are then combined using excluded-middle to yield two potential witnesses, which are reduced to an exact witness via contraction, which computationally speaking induces a case distinction.

Let's first focus on the left-hand branch, which establishes

$$\exists k \neg P(k) \to \exists n \forall m(P(n) \to P(m)).$$

The functional interpretation of the negative translation of the above is equivalent to

$$\exists \Psi_1 \forall k, g(\neg P(k) \to P(\Psi_1 kg) \to P(g(\Psi_1 kg)))$$

which is very easily satisfied by $\Psi_1 kg := k$ i.e.

$$\forall k, g(\neg P(k) \to P(k) \to P(gk))$$

Now we turn to the right-hand branch, which establishes

$$\forall k P(k) \to \exists n \forall m(P(n) \to P(m))$$

This is interpreted as

$$\exists \Psi_2, \Psi_3 \forall g(P(\Psi_3 g) \to P(\Psi_2 g) \to P(g(\Psi_2 g)))$$

which is also easily satisfied by $\Psi_2 g = 0$ and $\Psi_3 g = g0$ i.e.

$$\forall g(P(g0) \to P(0) \to P(g0))$$

We now mimic the combining of the two branches by setting $k := g0$ (note that this would formally correspond to several steps). We obtain

$$\forall g(\neg P(g0) \vee P(g0) \to (P(g0) \to P(g(g0))) \vee (P(0) \to P(g0)))$$

and so eliminating the now quantifier-free instance of LEM we end up with

$$\forall g((P(g0) \to P(g(g0))) \vee (P(0) \to P(g0))).$$

The final instance of contraction is dealt with by carrying out a definition-by-cases: Setting

$$\Phi g := 0 \text{ if } P(0) \to P(g0) \text{ else } g0$$

we have

$$\forall g(P(\Phi g) \to P(g(\Phi g)))$$

and so we're done. Of course, with a little thought we could have come up with such a program without going through the formal extraction. However, we want to illustrate how the construction of realizers mimics the formal proof, which we sketch via our proof tree in Figure 7. Note that, technically speaking, the application of the negative translation to the proof in Figure 1 would result in a new, bigger proof tree which establishes DP^N, and over which our realizer would be extracted. However, for readability we conflate this heavily in Figure 7, since it's only the main structure we want to highlight.

Figure 8 gives our usual summary of the algorithm underlying the interpretation. In this case it is quite succinct: Our functional Φ is nothing more than a straightforward case distinction and our 'counter' is just a simple function g. In this way, our interpretation is neither embellished by a nested function calls as in modified realizability, nor with an explicit backtracking algorithm as for the epsilon calculus.

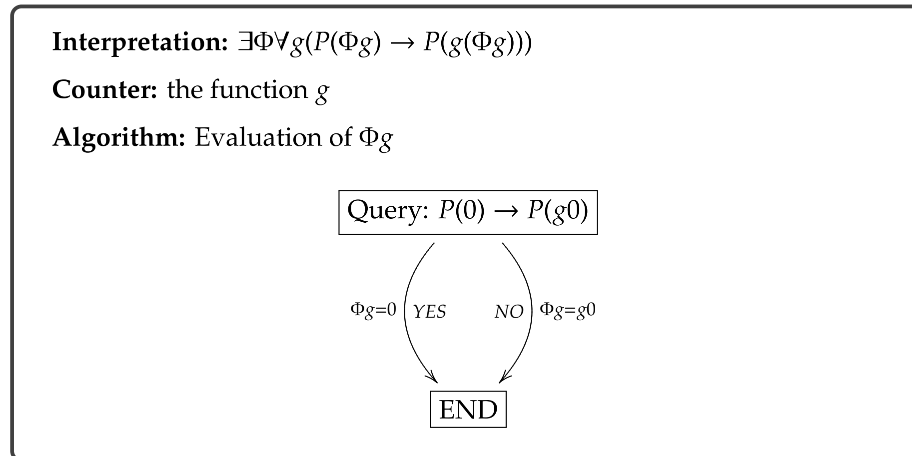

Figure 8: Dialectica interpretation of DP

3.4 Final remarks

It will hopefully not have escaped the reader that in the simple case of the drinkers paradox, all three interpretations arrive at essentially the same basic algorithm:

- Try a default value $n := 0$;

- Query the truth value of $P(0) \to P(m)$ where m is some counter value obtained from the environment;
- In the case of failure, update $n := m$.

This useful feature of the drinkers paradox allows us to focus on the very different ways in which each interpretation describes the underlying algorithm. Notably, the interaction with a counter value is present in each of the above examples, but encoded using a variety of structures, from a simple function in the case of the Dialectica to an internal abstraction in the case of modified realizability. At this point, it is perhaps worth pausing a moment to discuss these differences and the way in which they have affected how each interpretation has been used since Hilbert's program.

4 Interlude: A general perspective on proof interpretations

It may seem remarkable that as we slowly approach a century since the emergence of the epsilon calculus, a comprehensive comparison and assessment of the well-established computational interpretations of classical logic is still lacking - though a number of studies in this direction (such as [51]) have been produced.

Comparing proof interpretations is difficult. For a start, they are acutely sensitive to the way in which they are applied: An unnecessary double negation that would be routinely removed by a human applying a proof interpretation by hand could result in a considerable blow-up in complexity for a program formally extracted by a machine. Thus the Dialectica interpretation as a tool in proof mining is different to the Dialectica interpretation as an extraction mechanism in a proof assistant. Closely related to this is the fact that proof interpretations are in a constant state of flux, undergoing adaptations and refinements in quite specific directions as they become established in contrasting areas of application. So when comparing, for instance, the Dialectica to modified realizability, we have to quite carefully specify which of the many variants we have in mind.

Another problem lies in deciding the *nature* of the comparison we wish to make. For example, it is clear that in terms of their definitions, modified realizability and Dialectica belong very much to the same family of interpretations (an impression which is made precise in [36]), whereas the epsilon calculus is different species altogether. Nevertheless, in terms of the programs they produce for DP, it can be argued that the Dialectica has more in common with the epsilon calculus than it does with realizability. Without wanting to read too much into our simple case study, the point is simply that there is more to a proof interpretation than meets the eye, and a similarity in the definitional structure of two interpretations is not necessarily reflected when one examines the programs they produce.

As such, it is difficult to make sweeping claims about the relationship between proof interpretations without a certain amount of qualification. Never-

theless, it is undoubtedly the case that over the years, each of the three interpretations of classical logic mentioned above has developed a reputation, closely tied to the domains in which they primarily feature, and moreover posses certain characteristics which make them appealing for certain applications, and less appealing for others.

Perhaps the most striking feature of Hilbert's epsilon calculus is that it contains, implicitly, an elegant computational explanation of classical reasoning. Classical logic allows us to construct 'ideal objects' which are represented by epsilon terms, and the substitution methods shows us how to build approximations to these objects via a form of trial and error. In this way, the epsilon calculus can be seen as a direct precursor to much later work such as Coquand's game semantics [16], Avigad's update procedures [6], the Aschieri/Berardi learning-based realizability [5] and much more besides, which are all concerned in some way with a computational semantics of classical arithmetic via backtracking, or learning, a topic which we explore in more detail later.

All of this aside, the epsilon calculus plays active role in structural proof theory, where it has deep links with cut elimination [30], and the variants of the substitution method are still studied, particularly with regards to complexity issues [34]. In addition, the syntactic representation of quantifiers offered by the epsilon terms has been utilised by researchers working with proof assistants such as Isabelle and HOL [19]. Epsilon terms also feature in philosophy and linguistics, and the reader interested in this is encouraged to consult the many references provided in [8].

However, when it comes to concrete applications for the purpose of program extraction in mathematics or computer science, the epsilon calculus has been far superseded by the functional interpretations, which dominate this area for a number of reasons.

One property which the functional interpretations share is that the classical interpretations both factor through a natural intuitionistic counterpart. While the epsilon calculus applies directly to classical logic, the functional interpretations deal with this separately via the negative translation. For many applications of program extraction, particularly those which lean towards formal verification, the underlying proofs are indeed purely intuitionistic, in which case it makes far more sense to work with a direct implementation of the BHK interpretation such as modified realizability.

Another important feature of the functional interpretations is that the programs they extract are expressed as simple and clean lambda terms rather than an intricate transfinite recursion, as in the substitution method (though transfinite recursion is of course present in System T, it is conveniently encoded using the higher type recursors). In particular, lambda terms can be viewed in a very direct sense as functional programs, and are easily translated into real functional languages like Haskell or ML. Therefore when it comes to the formal extraction of functional programs from intuitionistic proofs, variants of modified realizability are an obvious choice, and their success in this area is proven by the wide range of applications, ranging from real number computation [12] to the synthesis of SAT solvers [11] (see the Minlog page at [1] for

further examples).

For applications of proof interpretations in mathematics, in the sense of the *proof mining* program [23], it is the Dialectica interpretation that has the starring role. Since applications of this kind typically deal with classical logic, they would in principle would also be suited to the epsilon calculus, and indeed early examples of proof mining made use of epsilon substitution [27]. However, the fact that the Dialectica interpretation extracts functional programs is also crucial for modern proof mining, due to the direct use of *mathematical properties* of those functionals.

In proof mining, the Dialectica interpretation is typically used in its monotone variant, which technically speaking is a combination of the usual Dialectica interpretation with a bounding relation on functionals known as majorizability. It is precisely this combination of proof interpretation and majorizability which enables the extraction of highly uniform, low complexity bounds from proofs which use what at first glance appear to be computationally intractable analytical principles such as Heine-Borel compactness. For a much more detailed discussion on the role the Dialectica interpretation plays in proof mining, see [24]. The main point to take away here is that the 'functional' part of functional interpretations is a crucial part of their success!

In the inevitably simplistic picture I have drawn above, I have deliberately contrasted the algorithmic elegance of the epsilon calculus with the applicability of the functional interpretations. A natural question is whether we can combine these two features in some way. This forms the main narrative of the remainder of the paper.

As we have seen, the functional interpretations deal with classical logic somewhat indirectly via negative translations. While in the case of realizability this can often be given a nice presentation in terms of games, in the case of the Dialectica interpretation in particular, the algorithm which is contained in the normalization of the term extracted from a negated proof is often very difficult to understand, and this is particularly the case once one moves away from simple examples like the drinkers paradox and analyzes 'real' theorems from mathematics, where complex and abstract forms of recursion start to play a role.

Of the three interpretations, it is the Dialectica which has featured most prominently in my own research. I was drawn to it by its importance to the proof mining program, and the rich variety of mathematical theorems which have been studied using this interpretation, to which I also made a few small contributions.

In my own case studies [37, 38, 39, 41, 42] which focused on relatively strong theorems from mathematical analysis and well quasi-order theory, I was surprised by how the operational behaviour of the extracted terms were initially quite difficult to understand, but after carefully analysing them it became clear that they implemented rather clever backtracking algorithms. This led me to start thinking about the relationship between the Dialectica and the epsilon calculus, or more specifically, to ask whether terms extracted by the Dialectica

in certain settings could be characterised as an epsilon-style procedure.

Although nowadays the epsilon calculus has become something of a specialised topic, in the past it was very much in the minds of those who pioneered applied proof theory. Early case studies by Kreisel were based on the substitution method, which in particular was used to prove soundness of the no-counterexample interpretation [26, 27], a close relative of the Dialectica interpretation (but see [22]).

Particularly fascinating for me is the possibility that, while working on his groundbreaking extension of the Dialectica to full classical analysis, Spector was already thinking of a new way to deal with the axiom of choice using an epsilon-style algorithm. In his first footnote to [50], Kreisel writes of the draft of the paper he received from Spector before the latter's death:

> *This last half page states that the proof of the Gödel translation of axiom F would use a generalization of Hilbert's substitution method as illustrated in the special case of* §12.1. *However Spector's notes do not contain any details, so that it is not quite clear how to reconstruct the proof he had in mind.*

Here axiom F is essentially the negative translation of the axiom of countable choice, which Spector had already interpreted using his schema of bar recursion in all finite types. While we do not know precisely what Spector intended, the idea of trying to replace the complicated forms of recursion which underlie the Dialectica with something more transparent is of great relevance now that proof interpretations are primarily used for practical program extraction as opposed to consistency proofs, and I talk about some more recent research in this direction in Section 5.1.

The search for a clear algorithmic representation of terms extracted by the Dialectica led me to think, more generally, of how one could utilise concepts from the theory of programming languages to help capture what is going on underneath the syntax of the Dialectica. To this end I worked on incorporating the notion of a global state into the interpretation via the state monad, and I discuss stateful programs in a much broader context in Section 5.2.

5 Epsilon style algorithms and the Dialectica interpretation

In Section 3 I presented three old and well established computational interpretations of classical logic, by sketching how they act in the simple case of the drinkers paradox. My stress here was on the historical context in which they arose, and on highlighting key features of the interpretations which help explain the roles they play in modern proof theory.

My goal in this section is to focus more closely on the connection between them, and more specifically, to demonstrate how the core idea underlying the epsilon calculus can help us better understand the Dialectica interpretation. As such, I consider two concepts in which epsilon substitution style algorithms can be elegantly phrased, but which are both much more general: Learning algorithms and stateful programs.

I have used each of these to study the Dialectica interpretation, in [45] and [46] respectively, and these studies will form the main narrative of this section. Nevertheless, learning and programming with state feature much more broadly in modern approaches to program extraction, and I was certainly not the first to utilise them in this way. Therefore my priority here is to explain both in suitably general terms that their application in a broader context can be appreciated.

5.1 Learning algorithms

As we have seen, the notion of learning as a computational semantics of classical logic is already explicitly present in the epsilon calculus, and can actually be seen in each of the interpretations of the drinkers paradox in Section 3. In fact, over the years this idea has continually resurfaced in a variety of different settings, some of which have already been mentioned (and a detailed survey of which would be an extensive work in its own right).

In this section, I will introduce a specific form of learning phrased in the language of all finite types, which I found useful in clarifying certain aspects of the Dialectica interpretation in [45], characterising extracted programs as epsilon-style procedures. I will briefly sketch how this relates to the drinkers paradox, but here our running example is far less illuminating, and so I will go on to present a Skolem form DP^ω of the drinkers paradox, which is solved using a simple kind of learning procedure that appears frequently in the literature.

Computational interpretations of classical logic via learning are particularly meaningful when one focuses on $\forall\exists\forall$-formulas. By a simple adaptation of our argument in Section 3.3 we see that the combination of the negative translation and Dialectica interpretation carries out the following transformation on such formulas:

$$\forall a \exists x \forall y Q(a,x,y) \mapsto \exists \Phi \forall a, g Q(a, \Phi a g, g(\Phi a g))$$

which happens to also coincide with Kreisel's no-counterexample interpretation for formulas of this type. Here, we can visualise $\Phi a g$ as constructing an approximation to the non-constructive object x in the following sense: Rather than demanding some x such that $Q(a,x,y)$ is valid for all y, we compute for any given g an x such that $Q(a,x,gx)$ holds.

Such reformulations of $\forall\exists\forall$-theorems are widely studied in proof mining: In the case of convergence they go under the name of 'metastability'. Here x is a number, and one is typically concerned with producing a computable bound for the approximation of x i.e. $\forall a, g \exists x \leq \Phi a g Q(a,x,gx)$ (see [25] for an interesting discussion of this and several related concepts). However, x could be a more complicated object, such as a maximal ideal or a choice sequence, and in this case approximations to x are often built using a complex forms of higher-order recursion whose normalization is subtle and whose operational meaning can be difficult to intuit, but which can be cleanly and elegantly presented as learning procedures.

So what is a learning procedure? In my own version of this well-known idea [45], I start with the idea of a learning *algorithm*. A learning algorithm is

assigned a type ρ, τ, and, roughly speaking, seeks to build approximations to an object of type ρ using building blocks of type τ. Formally, it is given by a tuple $\mathcal{L} = (Q, \xi, \oplus)$ where the components are to be understood as follows:

- Q is a predicate on ρ which tells us if a suitably 'good' approximation has been found;
- $\xi : \rho \to \tau$ is a function which takes a current approximation x and returns a new building block $\xi(x)$;
- $\oplus : \rho \times \tau \to \rho$ combines x with a building block y to form a new approximation $x \oplus y$.

Given any initial object x_0, a learning algorithm triggers a learning *procedure* $x_0, x_1, x_2, \ldots$, which is defined recursively by

$$x_{i+1} = \begin{cases} x_i & \text{if } Q(x_i) \\ x_i \oplus \xi(x_i) & \text{otherwise} \end{cases}$$

The basic idea is very simple. At each stage in the procedure we check whether or not x_i is a good approximation. If it is, we leave it unchanged, if not, we assume that by its failure we have learned a new piece of information $\xi(x_i)$, which we can use to update the approximation $x_{i+1} = x_i \oplus \xi(x_i)$. We are typically interested in learning algorithms whose learning procedures terminate at some point, by which we mean there is some k such that $Q(x_k)$ holds. In this case we say that x_k is the *limit* of the procedure and write $\lim \mathcal{L}[x] := x_k$.

The Dialectica interpretation of the drinkers paradox can be solved by an extremely simple learning algorithm of length at most two: Given our counter function g we define $\mathcal{L}$ by

$$Q(x) = P(x) \to P(gx) \qquad \xi = g \qquad x \oplus y = y$$

and it is easy to see that $\Psi g := \lim \mathcal{L}[0]$ works. Of course this is essentially the same as the program

$$\Phi g = 0 \text{ if } P(0) \to P(g0) \text{ else } g0,$$

but here the learning that we see in the epsilon calculus is made explicit. For the sake of completeness, we give the summary of the algorithm in Figure 9.

Unsurprisingly, learning algorithms in [45] were not introduced to help better understand the drinkers paradox, but to give an algorithmic description of how complicated objects such as choice sequences are built, which is more or less the role they play elsewhere in the literature. The main part of my own paper is in fact the study of learning algorithms on infinite sequences, which are rather subtle and involve non-trivial termination arguments.

To give a the reader a flavour of a meaningful learning procedure, let's take instead of a single predicate P a sequence (P_n) of decidable predicates, and consider the following sequential form of the drinkers paradox:

$$\mathrm{DP}^{\omega} \quad : \quad \exists f^{\mathbb{N} \to \mathbb{N}} \forall n, m (P_n(fn) \to P_n(m))$$

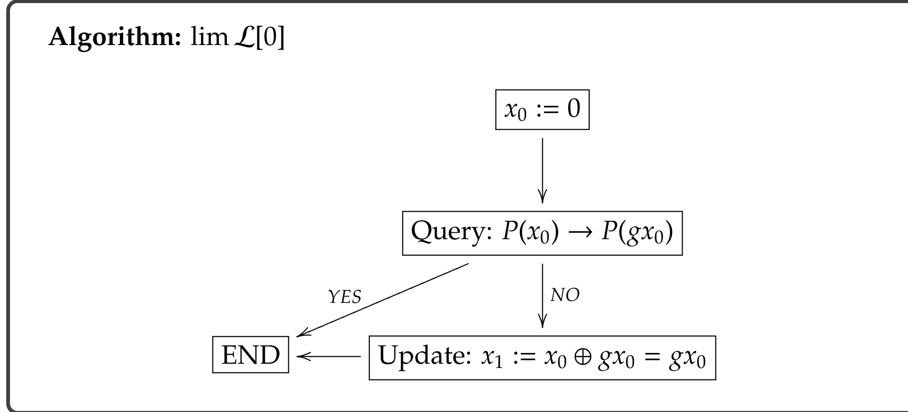

Figure 9: Interpretation of DP via learning procedure

which is provable using classical logic in conjunction with the axiom of countable choice. The Dialectica interpretation of the negative translation of this (before bringing the $\exists f$ out to the front) is given by

$$\forall \omega, \phi \exists f (P_{\omega f}(f(\omega f)) \to P_{\omega f}(\phi f))$$

where ω, ϕ are functionals of type $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$. In other words, we need to build an approximation to the Skolem function f which works, not for all n and y, but just for ωf and ϕf. We can construct such an f in ω, ϕ by taking the limit of the learning algorithm $\mathcal{L}[\lambda i.0]$ for $\mathcal{L}$ of type $\mathbb{N} \to \mathbb{N}, \mathbb{N} \times \mathbb{N}$ given by

- $Q(f) := P_{\omega f}(f(\omega f)) \to P_{\omega f}(\phi f)$
- $\xi(f) := (\omega f, \phi f)$
- $f \oplus (n, x) := f[n \mapsto x]$

where $f[n \mapsto x]$ is the function f updated with the value x at argument n. It's immediately clear that once the underlying learning procedure terminates with some f satisfying $Q(f)$ then we're done, as this this is just the definition of a realizer for the functional interpretation of DP^{ω}. What is less clear is what the procedure does and why it terminates!

The learning procedure generates a sequence of functions

$$f_0, f_1, f_2, \ldots$$

where $f_{i+1} = f_i[n_i \mapsto x_i]$ (we write $(n_i, x_i) := (\omega f_i, \phi f_i)$ for simplicity). Whenever our current attempt f_i fails i.e. $\neg Q(f_i)$ holds, we know in particular that $\neg P_{n_i}(x_i)$, and so we have learned something about our predicates P_i. Updating $f_{i+1} = f[n_i \mapsto x_i]$ means that f_{i+1} is now a valid choice sequence at point n_i, since $P_{n_i}(x_i) \to P_{n_i}(y)$ for all y. Looking at the procedure as a whole, it's clear that

f_{i+1} is in fact a valid choice sequence for all of $n_0, \ldots, n_i$, and so our learning procedure can be viewed as building an increasingly accurate approximation to the ideal choice function f.

Now we need to impose a condition on ω and ϕ, namely that they are *continuous*, which means that they only look at a finite part of their input. This is not an unreasonable assumption, since all computable functionals in particular are continuous, but it nevertheless means that our learning procedure cannot be defined as a term of System T, which is a good indication of the fact that we are going beyond the usual soundness theorem for the Dialectica interpretation, since DP^ω is not provable in Peano arithmetic.

For each n, the sequence $f_0(n), f_1(n), f_2(n), \ldots$ changes at most once, and thus the functions $f_0, f_1, f_2, \ldots$ tend to some limit. By continuity of ω, this means that $\omega f_0, \omega f_1, \omega f_2, \ldots$ eventually becomes constant, and so at some point j we must have $n_{j+1} = \omega f_{j+1} \in \{n_0, \ldots, n_j\}$ and so in particular $\neg P_{n_{j+1}}(f_{j+1}(n_{j+1}))$, which implies $Q(f_{j+1})$, and hence termination of the procedure. A diagram of the learning procedure as a whole is given in Figure 10.

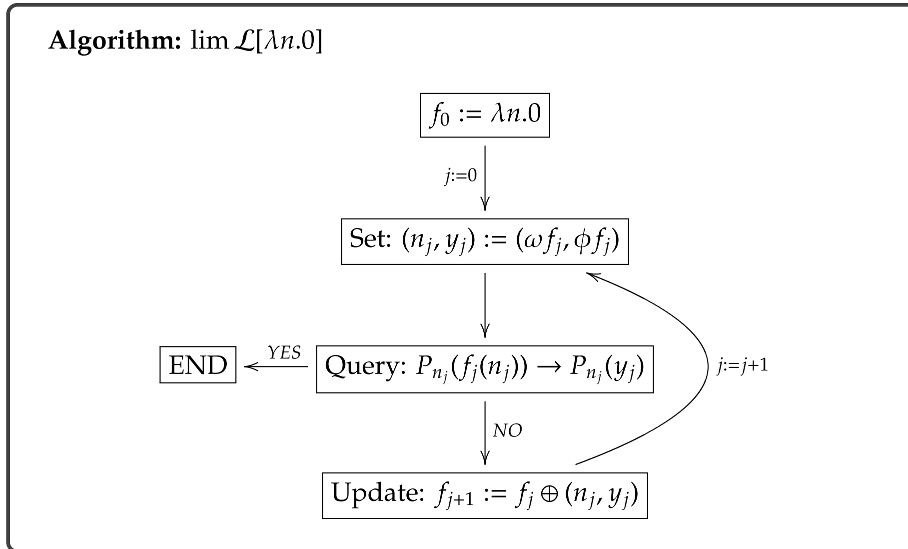

Figure 10: Interpretation of DP^ω via learning procedure

The reader will hopefully have noticed that our learning procedure is nothing more than a simple version of epsilon substitution, formulated in the higher-order language of the Dialectica interpretation and used to produce a realizer for the Dialectica interpretation of a choice principle! We imagine the function f in DP^ω as finding some element such that $\neg P_i(m)$ holds, if it exists, and as such identify fn with $\epsilon m \neg P_n(m)$. Our learning procedure starts by setting $fn = 0$ for all n, and then, via a process of trial and error, repeatedly corrects the function until a sufficiently good approximation has been found. The current critical

formula is given by $Q(f)$.

The idea of solving DP^ω in this way can already be found in §12.1 of Spector [50], and it is precisely this that he claims to have generalised to solve the functional interpretation of full countable choice. While his proof was never recovered, learning procedures have since emerged in several places, and many works on this topic can be characterised as the study of more complicated versions of this idea.

My own work in this direction has been primarily concerned with the Dialectica interpretation itself and the development of Spector's idea. In [45] I study a more general class of learning procedures, and show in particular that Spector's bar recursion can be seen as a kind of 'forgetful' learning procedure. I extend this idea in [47], exploiting the connection established in [43] between bar recursion and Berger's open recursion [10], and devise a learning procedure which solves the Dialectica interpretation of a simple form of Zorn's lemma.

However, the relationship between classical logic and learning has also been developed by other authors in different contexts. To give a few examples: In [9], a variation of the algorithm above is captured in a realizability setting via a form of recursion in higher-types now known as the BBC functional, which has since been studied in greater depth in [10]. In [6], learning procedures are called *update procedures*, and are used to prove the 1-consistency of arithmetic.

Finally, and perhaps most importantly, in [5] and later works by these authors (see in particular [3, 4]), a special form of realizability is developed which is based entirely on the notion of learning. Here, learning is captured through a 'state of knowledge', which contains a finitary approximation to a Skolem function. Realizers for existential statements are evaluated relative to this state, and they either succeed, or discover an error in the current state, in which case the state is updated and the process repeats. In this, more than any of the other works mentioned here, learning is characterised as a stateful program, which brings me to the final part of the paper.

5.2 Stateful programs

Learning procedures are illuminating precisely because they capture the computational content of classical reasoning on an algorithmic level, through the *evolution of a state*. The idea of a computation as an action on a state is the foundation of the imperative programming paradigm, on which many major programming languages - including C++, Java and Python - are partially based. This begs a much broader question: Can we utilise ideas from the theory of imperative languages to more clearly and elegantly express the computational meaning of classical reasoning?

This point of view has been explored primarily in the French style of program extraction via e.g. Krivine realizability, in which there is a greater tendency to capture the computational content of classical logic via control operators such as call/cc, rather than through logical techniques like negative translations (see [31] for an interesting discussion of this).

However, my emphasis here is somewhat different: Namely on adapting

traditional proof interpretations like the Dialectica so that the operational behaviour of extracted programs can be better understood. The benefits for this are great: the Dialectica has undoubtedly the richest catalogue of applications of all proof interpretations, and so a characterisation of extracted terms as stateful programs could lead to fascinating connections between imperative languages and non-constructive principles from everyday mathematics.

This final section discusses ideas in this direction. My main goal will be to indicate how stateful programs can be simulated in a functional setting, and describe how they can be used to enrich the Dialectica interpretation and give it a more imperative flavour. Though this takes as its inspiration both the epsilon substitution and the learning procedures just presented, work of this kind is more general and focused on breaking down proof interpretations on a much lower level. It is also far less developed, with only a few recent works which view the traditional interpretations from this perspective, and as such this section can be seen as an extended conclusion which looks ahead to future research.

I should start by clarifying the difference between imperative and functional programs. Simply put, imperative programs are built using commands which act on some underlying state. A while loop can be seen as a simple program written in an imperative style. Consider the following formulation of the factorial function:

```
i := 1
j := 1
while i < n do
    i := i + 1
    j := i · j
print j
```

The loop is preceded by some variable assignments which determine an initial state, each iteration of the loop modifies the state until it terminates, and finally the output of the computation is read off from the state.

Another program which fits the imperative paradigm is of course the epsilon substitution method, where one imagines epsilon terms as global variables: Our initial state assigns to each epsilon term the value 0, and each stage of the method updates the state by correcting an epsilon term until a suitable approximation is found. The closely related learning procedures of the last section can be seen in a similar way.

Programs extracted using the functional interpretations are, however, primarily expressed in some abstract functional language. In functional languages, the emphasis is on *what* a program should do, rather than *how* is does it. Programs are written as simple recursive equations and the concept of global state is not naturally present. In the functional style, the factorial function would be specified as just

$$f(0) = 1 \qquad f(n+1) = (n+1) \cdot f(n)$$

The use of simple higher-order functional languages like System T is a central feature of the functional interpretations, and as highlighted in Section 4, a key component of their success. Nevertheless, there *is* a natural way of incorporating imperative features into functional calcui: The use a *monad*, a structure familiar to any functional programmer. A full technical definition of a monad and it's role in programming is far beyond the scope of this paper (see e.g. [32, 52]), but we will give quick sketch of the *state monad*, which is of relevance in this section.

Suppose we are working in a simple functional calculus like System T and we want to capture some overriding global state which keeps track of certain aspects of the computations. To this end, we introduce to our calculus a new type S of states, together with an mapping T on types defined by

$$T\rho := S \to \rho \times S.$$

This is the state monad. Monads come equipped with two operators: A *unit* of type $\rho \to T\rho$ and a *bind* of type $T\rho \to (\rho \to T\tau) \to T\tau$. For the state monad these are given by the maps

$$\begin{aligned} \mathrm{unit}(x) &:= \lambda s.(x, s) \\ \mathrm{bind}(a)(b) &:= \lambda s.bxs' \quad \text{where } (x, s') = as \end{aligned}$$

In our enriched language, base types ρ are interpreted as monadic types $T\rho = S \to \rho \times S$, while function types $\rho \to \tau$ are interpreted as objects of type $\rho \to T\tau = \rho \to S \to \tau \times S$. The unit map specifies a neutral translation from terms of plain type to the corresponding monadic type: For the state monad, neutrality means that the state remains unchanged. The bind map allows us to apply monadic functions to monadic arguments, as we will see below.

In the pure functional world, a term of ground type ρ is just a program which evaluates to some value of that type. Under the state monad, it becomes a term which takes an initial state s_1 and returns a pair (u, s_2) consisting of a value of type ρ and a final state s_2. Similarly, in the pure word, a term of type $\rho \to \tau$ is a function which takes an input x and returns an output y. Under the state monad, it becomes a function which takes an input x together with an initial state s_1 and returns an output-final state pair (y, s_2).

To emphasise this point, consider our two versions of the factorial function. The purely functional one has type $\mathbb{N} \to \mathbb{N}$: it takes an input a number n and evaluates to $n!$. The imperative version, however, acts on an underlying state: It takes as input the number n together with the initial state $(i, j) := (1, 1)$ and returns the output $n!$ together with the final state $(i, j) := (n, n!)$. In this sense it has type $\mathbb{N} \to S \to \mathbb{N} \times S$.

The bind operator is responsible for function application on the monadic level: It takes a monadic argument $a : T\rho$ together with a monadic function $b : \rho \to T\tau$ and returns a monadic output $\mathrm{bind}(a)(b) : T\tau$. This output mimics the following procedure: We take our initial state s_1 and plug it into our argument a to obtain a pair (u, s_2) of type $\rho \times S$. We treat s_2 as an intermediate state in the

computation, and plug both it and the value u into the monadic function b to obtain a pair (v, s_3) of type $\tau \times S$.

The purpose of all this is to try to convey to the reader how stateful functions like our while loop can be simulated in a functional environment via the state monad. We can, more generally, translate the finite types as a whole to a corresponding hierarchy of monadic types, and using the unit and bind operations define a translation on pure terms of System T to monadic terms in a variety of ways, depending on what kind of computation we are trying to simulate and what kind of information we aim to capture in our state. Again, this is presented in detail in [32, 52].

So how can all this be applied to the Dialectica interpretation? Suppose we have a proof of a simple $\forall\exists$ statement $\forall x \exists y Q(x, y)$: The Dialectica interpretation would normally extract a pure program

$$f : \rho \to \tau$$

which takes an input x of type ρ and returns an output y of type τ satisfying $Q(x, y)$. However, using the state monad together with a suitable translation on terms, we could instead set up the interpretation so that it extracts a state sensitive program

$$b : \rho \to S \to \tau \times S$$

which takes an input x and initial state s_1, and returns an output pair $bxs = (y, s_2)$, where y satisfies $Q(x, y)$ and the final state s_2 tells us something about what our function *did* during the computation, the idea being to make explicit in some sense the underlying substitution, or learning algorithm.

In a recent paper [46], I explore some of the things which a global state can do in this context, focusing in particular on the program's 'interaction with the mathematical environment'. In other words, when performing a computation on some ambient mathematical structure, be it a predicate, a bounded monotone sequence, a colouring of a graph etc., the state monad is used to track the way in which the program queries this environment as it attempts to build an approximation to some object related to that structure e.g. an approximate drinker, an interval of metastability, an approximate monochromatic set etc. In this way, the state monad allows us to smoothly capture the epsilon style procedures which *implicitly* underlie the normalization of extracted functional terms.

Let's try to illustrate this with a very simple example using, for the final time, the drinkers paradox. Note that the $\forall\exists$ formula we have in mind here is the (partial) Dialectica interpretation of DP^N, namely $\forall g \exists x (P(x) \to P(gx))$. Suppose that our mathematical environment consists of a single predicate P, and that our states are defined to be finite piece of information about this predicate i.e. for $s \in S$ we have

$$s = [Q_0(n_0), \ldots, Q_{k-1}(n_{k-1})]$$

for where Q_i is either P or $\neg P$. Our realizer could then take a state s as an input, and whenever it tests P on a given value it will add this information

to the state. For the usual drinkers paradox, we would get the following variation of the program given in Section 5, which now has the monadic type $(\mathbb{N} \to \mathbb{N}) \to S \to \mathbb{N} \times S$:

$$\Phi gs := \begin{cases} \langle 0, s :: \neg P(0) \rangle & \text{if } \neg P(0) \\ \langle 0, s :: P(0) :: P(g0) \rangle & \text{if } P(g0) \\ \langle g0, s :: P(0) :: \neg P(g0) \rangle & \text{otherwise} \end{cases}$$

This breaks our usual case distinction into parts, starting by testing the truth value of $P(0)$. If this is false, we know that $P(0) \to P(g0)$ is true and thus 0 is realizer for DP, and we add what we have learned about P to the state. On the other hand, if $P(0)$ is true, the truth value of $P(0) \to P(g0)$ is still undetermined, so we must now test $P(g0)$, leading to two possible outcomes. Unlike the pure program, our stateful program returns a *reason* as to why the witness it has returned works, giving us the information about the mathematical environment which has determined its choice. Alternatively put, our program returns a record of the underlying substitution procedure carried out by the term. As usual we present this as an algorithm in Figure 11.

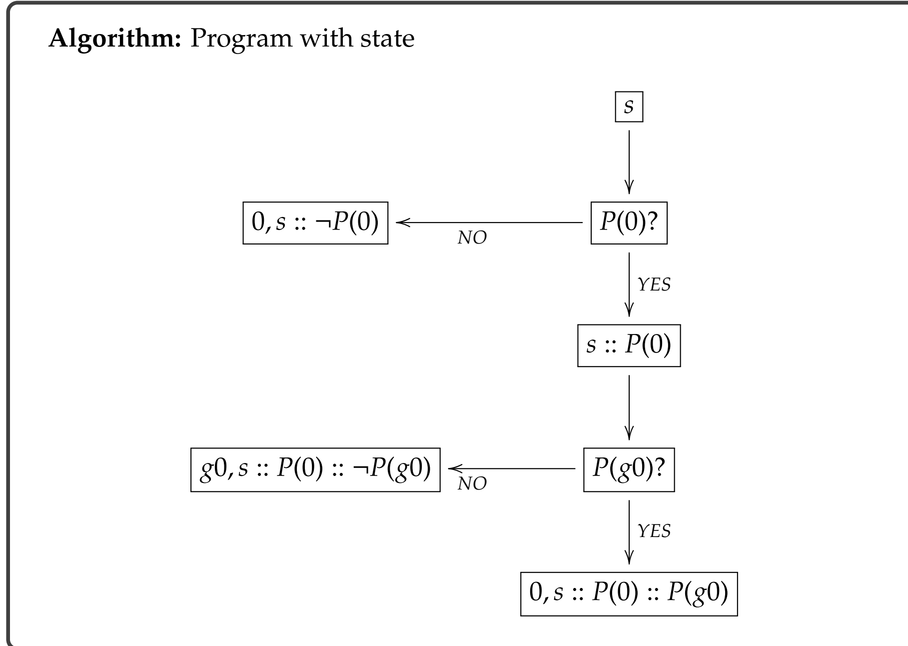

Figure 11: Interpretation of DP with state

We emphasise that this is just a simple illustration of how the state could be used to enhance the programs we extract - to just record the interactions which have taken place with the mathematical environment. In particular, the realizer

makes no use of what is currently written in the state. One could change this so that it first looks to see whether a truth value of e.g. $P(0)$ is already present in s, and only then proceed to test $P(0)$. In cases where the cost of evaluating P is high, this would improve the efficiency of the extracted program, allowing us to store previously computed values so that they can be accessed at a later stage in the computation. More sophisticated variants are possible: Where P is not decidable, we can simply make an arbitrary choice for its truth value and make the validity of our output witness dependent on the truth of the state. In [46] a very general framework for working with state is presented via an abstract soundness theorem, and a range of applications are discussed, ranging from Herbrand's theorem to program synthesis.

I am certainly not alone in turning towards concepts such as monads to try to better explain what is going on under the syntax of traditional interpretations, although to date there is comparatively little research in this direction. Similar work, also pertaining to the Dialectica interpretation, has been carried out by Pedrot in [40]. Berger et al. have used the state monad to automatically extract list sorting programs using modified realizability [14], while the state monad is utilised by Birolo in [15] to give a more general version of Aschieri-Berardi learning realizability. Interestingly, monads also feature prominently in the construction of the Dialectica categories [17] - abstract presentations of the interpretation which formed early models of linear logic - though there they play a somewhat different role.

6 Conclusion

I began this paper with an introduction to the epsilon calculus, a proof theoretic technique almost a century old and originating in the foundational crisis of mathematics. I concluded in the present day by sketching how stateful programs are being used to capture algorithmic aspects of proof interpretations and characterise the operational behaviour of extracted programs.

Yet the ideas which underpin the latter are in some way already present in the substitution method, illustrating how certain universal characteristics seem to underlie computational interpretations of classical logic, manifesting themselves in different ways in different settings.

In my short presentations of the three traditional proof interpretations, I hope to have hinted at the deep connections that they share, which are not necessarily apparent in their formal definitions. Some of these connections have been made more explicit in modern research, in particular the utilisation of epsilon substitution-like procedures in the setting of functional interpretations, which formed the main topic of the second part of this paper.

As proof theory moves away from foundational concerns and towards applications in mathematics and computer science, is has become increasingly relevant to utilise new languages and techniques to modernise traditional proof theoretic methods. It will be fascinating to see how the proof interpretations which formed the subject of this article evolve in the future.

Acknowledgements. I am grateful to the anonymous referee, together with Ulrik Buchholtz, Felix Canavoi and Sam Sanders, whose corrections and suggestions improved this paper considerably.

References

[1] http://www.mathematik.uni-muenchen.de/~logik/minlog/. Official homepage of MINLOG, as of January 2018.

[2] W. Ackermann. Zur widerspruchsfreiheit der zahlentheorie. *Mathematische Annalen*, 117:162–194, 1940.

[3] F. Aschieri. *Learning, Realizability and Games in Classical Arithmetic*. PhD thesis, Universita degli Studi di Torino and Queen Mary, University of London, 2011.

[4] F. Aschieri. Transfinite update procedures for predicative systems of analysis. In *Computer Science Logic (CSL'11)*, volume 12 of *Leibniz International Proceedings in Informatics*, pages 1–15, 2011.

[5] F. Aschieri and S. Berardi. Interactive learning-based realizability for Heyting arithmetic with EM1. *Logical Methods in Computer Science*, 6(3), 2010.

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

[7] J. Avigad and S. Feferman. Gödel's functional ("Dialectica") interpretation. In S. R. Buss, editor, *Handbook of Proof Theory*, volume 137 of *Studies in Logic and the Foundations of Mathematics*, pages 337–405. North Holland, Amsterdam, 1998.

[8] J. Avigad and R. Zach. The epsilon calculus. In *The Stanford Encyclopedia of Philosophy*. revised edition nov 2013 edition, 2013. available online at https://plato.stanford.edu/entries/epsilon-calculus/.

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

[10] U. Berger. A computational interpretation of open induction. In *Proceedings of LICS 2004*, pages 326–334. IEEE Computer Society, 2004.

[11] U. Berger, A. Lawrence, F. Forsberg, and M. Seisenberger. Extracting verified decision procedures: DPLL and resolution. *Logical Methods in Computer Science*, 11(1:6):1–18, 2015.

[12] U. Berger, K. Miyamoto, and H. Schwichtenberg. Logic for Gray-code computation. In D. Probst and P. Schuster, editors, *Concepts of Proof in Mathematics, Philosophy, and Computer Science*. De Gruyter, 2016.

[13] U. Berger and H. Schwichtenberg. Program extraction from classical proofs. In D. Leivant, editor, *Logic and Computational Complexity workshop (LCC'94)*, volume 960 of *Lecture Notes in Computer Science*, pages 77–97. 1995.

[14] U. Berger, M. Seisenberger, and G. Woods. Extracting imperative programs from proofs: In-place quicksort. In *Proceedings of TYPES 2013*, volume 26 of *LIPIcs*, pages 84–106, 2014.

[15] G. Birolo. *Interative Realizability, Monads and Witness Extraction*. PhD thesis, Universitá degli Studi di Torino, 2012.

[16] T. Coquand. A semantics of evidence for classical arithmetic. *Journal of Symbolic Logic*, 60:325–337, 1995.

[17] V. de Paiva. *The Dialectica Categories*. PhD thesis, University of Cambridge, 1991. Published as Technical Report 213, Computer Laboratory, University of Cambridge.

[18] G. Ferreira and P. Oliva. On the relation between various negative translations. In *Logic, Construction, Computation*, volume 3 of *Ontos-Verlag Mathematical Logic*, pages 227–258. 2012.

[19] Martin Giese and Wolfgang Ahrendt. Hilbert's epsilon-terms in automated theorem proving. In *TABLEAUX*, volume 1617 of *LNCS*, pages 171–185. Springer, 1999.

[20] K. Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. *dialectica*, 12:280–287, 1958.

[21] K. Gödel. *Collected Works*, volume II. Oxford University Press, New York, 1990.

[22] U. Kohlenbach. On the no-counterexample interpretation. *Journal of Symbolic Logic*, 64:1491–1511, 1999.

[23] U. Kohlenbach. *Applied Proof Theory: Proof Interpretations and their Use in Mathematics*. Monographs in Mathematics. Springer, 2008.

[24] U. Kohlenbach. Kreisel's 'shift of emphasis' and contemporary proof mining. Chapter for forthcoming book 'Intuitionism, Computation, and Proof: Selected themes from the research of G. Kreisel', 2018.

[25] U. Kohlenbach and P. Safarik. Fluctuations, effective learnability and metastability in analysis. *Annals of Pure and Applied Logic*, 165:266–304, 2014.

[26] G. Kreisel. On the interpretation of non-finitist proofs, Part I. *Journal of Symbolic Logic*, 16:241–267, 1951.

[27] G. Kreisel. On the interpretation of non-finitist proofs, Part II: Interpretation of number theory. *Journal of Symbolic Logic*, 17:43–58, 1952.

[28] G. Kreisel. Interpretation of analysis by means of functionals of finite type. In A. Heyting, editor, *Constructivity in Mathematics*, pages 101–128. North-Holland, Amsterdam, 1959.

[29] J.-L. Krivine. Realizability in classical logic. In *Interactive Models of Computation and Program Behaviour*, volume 27 of *Panoramas et Synthèses*, pages 197–229. Société Mathématique de France, 2009.

[30] G. Mints. Cut elimination for a simple formulation of the epsilon calculus. *Annals of Pure and Applied Logic*, 152(1–3):148–169, 2008.

[31] A. Miquel. Existential witness extraction in classical realizability and via a negative translation. *Logical Methods in Computer Science*, 7(2:2):1–47, 2011.

[32] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[33] G. Moser. Ackermann's substitution method (remixed). *Annals of Pure and Applied Logic*, 142(1–3):1–18, 2006.

[34] G. Moser and R. Zach. The epsilon calculus and Herbrand complexity. *Studia Logica*, 82(1):133–155, 2006.

[35] P.S. Novikoff. On the consistency of certain logical calculi. *Matematiceskij Sbornik*, 12(54):230–260, 1943.

[36] P. Oliva. Unifying functional interpretations. *Notre Dame Journal of Formal Logic*, 47(2):263–290, 2006.

[37] P. Oliva and T. Powell. A constructive interpretation of Ramsey's theorem via the product of selection functions. *Mathematical Structures in Computer Science*, 25(8):1755–1778, 2015.

[38] P. Oliva and T. Powell. A game-theoretic computational interpretation of proofs in classical analysis. In *Gentzen's Centenary: The Quest for Consistency*, pages 501–532. Springer, 2015.

[39] P. Oliva and T. Powell. Spector bar recursion over finite partial functions. *Annals of Pure and Applied Logic*, 168(5):887–921, 2017.

[40] P-M. Pedrot. *A Materialist Dialectica*. PhD thesis, Université Paris Diderot, 2015.

[41] T. Powell. Applying Gödel's Dialectica interpretation to obtain a constructive proof of Higman's lemma. In *Proceedings of Classical Logic and Computation '12*, volume 97 of *EPTCS*, pages 49–62, 2012.

[42] T. Powell. *On Bar Recursive Interpretations of Analysis*. PhD thesis, Queen Mary University of London, 2013.

[43] T. Powell. The equivalence of bar recursion and open recursion. *Annals of Pure and Applied Logic*, 165(11):1727–1754, 2014.

[44] T. Powell. Parametrised bar recursion: A unifying framework for realizability interpretations of classical dependent choice. *Journal of Logic and Computation*, 2015. (advance access).

[45] T. Powell. Gödel's functional interpretation and the concept of learning. In *Proceedings of Logic in Computer Science (LICS 2016)*, pages 136–145. ACM, 2016.

[46] T. Powell. A functional interpretation with state. In *Proceedings of Logic in Computer Science (LICS 2018)*. ACM, 2018.

[47] T. Powell. Well quasi-orders and the functional interpretation. To appear in: Schuster, P., Seisenberger, M. and Weiermann, A. eds., *Well Quasi-Orders in Computation, Logic, Language and Reasoning*, Trends in Logic, Springer., 2018.

[48] H. Schwichtenberg and S. Wainer. *Proofs and Computations*. Perspectives in Logic. Cambridge University Press, 2011.

[49] R. Smullyan. *What is the Name of this Book? The Riddle of Dracula and Other Logical Puzzles*. Prentice Hall, 1978.

[50] C. Spector. Provably recursive functionals of analysis: a consistency proof of analysis by an extension of principles in current intuitionistic mathematics. In F. D. E. Dekker, editor, *Recursive Function Theory: Proc. Symposia in Pure Mathematics*, volume 5, pages 1–27. American Mathematical Society, Providence, Rhode Island, 1962.

[51] T. Trifonov. *Analysis of Methods for Extraction of Programs from Non-Constructive Proofs*. PhD thesis, Ludwig-Maximilians-Universität Munich, 2011.

[52] P. Wadler. Monads for functional programming. In *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, 1995.