Some applications of monads in proof theory

Thomas Powell

Technische Universität Darmstadt

Oberseminar Mathematische Logik
Ludwig-Maximilians-Universität Munich

12 July 2017

What is this talk about?

I will present, very informally, two independent projects which both revolve around the notion of a **monad**. These are:

1. A denotational complexity semantics for higher type programs.

2. A variant of Gödel's functional interpretation with state.

I am a proof theorist, so **apologies in advance** to any

- category theorists

- functional programmers

in the audience.

What is a monad (in the context of functional programming)?

Imagine that X denotes a type in some lambda calculus or functional language. A monad (T, η, μ) comprises

- A map T, which to each X associates a new *monadic* type TX, together with a lifting of functions $f : X \to Y$ to functions $Tf : TX \to TY$ between monadic types.

TX should be viewed as a normal type X enriched with some additional structure which represents a procedural feature.

- A *unit* operation $\eta_X : X \to TX$, which maps any plain $x \in X$ to some 'neutral' $\eta_X(x) \in TX$.

- A *join* operation $\mu_X : TTX \to TX$ which combines, or 'flattens', two layers of monadic information.

Naturally there are a bunch of commuting diagrams, which I won't draw here. But a particularly important idea is that an object $a \in TX$ can be *bound* with a function $f : X \to TY$ to produce an object $b \in TY$. This is achieved via a map

$$\text{bind} : TX \to (X \to TY) \to TY$$

which can be defined as

$$TX \xrightarrow{Tf} TTY \xrightarrow{\mu_Y} TY$$

What this means is that we can carry out actions in sequence, feeding the monadic output of each action into the next (often this is referred to as a 'pipeline'). The composition

$$X \xrightarrow{f} Y \xrightarrow{g} Z \xrightarrow{h} \cdots$$

is enriched as

$$TX \xrightarrow{Tf} TTY \xrightarrow{\mu_Y} TY \xrightarrow{Tg} TTZ \xrightarrow{\mu_Z} TZ \xrightarrow{Th} \cdots$$

If this all looks a bit mysterious, examples are on the way...

I. A complexity monad

- $TX := \mathbb{N} \times X$, and for $f : X \to Y$ we define $Tf : \mathbb{N} \times X \to \mathbb{N} \times Y$ to be

$$(n, x) \mapsto (n, f(x)).$$

Objects $(n, x) \in TX$ represent a term $x \in X$ and a number n which denotes a 'complexity' of x.

- The unit $\eta_X : X \to \mathbb{N} \times X$ is given by

$$x \mapsto (0, x)$$

i.e. a neutral object has no complexity.

- The join operation $\mu_X : \mathbb{N} \times (\mathbb{N} \times X) \to \mathbb{N} \times X$ is given by

$$(m, (n, x)) \mapsto (m + n, x)$$

i.e. the combination of two complexities is their sum.

How do we bind an object $(n, x) \in \mathbb{N} \times X$ to a function $X \to \mathbb{N} \times Y$? The latter can be visualised as a pair of functions

$$x \mapsto (c(x), f(x))$$

i.e. that takes some object x and returns a value $f(x)$ together with a complexity $c(x)$ of $f(x)$.

We bind the pair (n, x) to $x \mapsto (c(x), f(x))$ as follows:

$$(n, x) \mapsto (n, (c(x), f(x))) \mapsto (n + c(x), f(x)).$$

Thus the binding operation allows us to also takes into account the complexity of x: We imagine that $f(x)$ is evaluated as follows:

1. evaluate x (complexity $= n$)

2. apply f to x and evaluate this (complexity $= c(x)$)

So the total complexity of evaluating $f(x)$ is $n + c(x)$.

II. The state monad

- $TX := S \to X \times S$, and for $f : X \to Y$ we define $Tf : (S \to X \times S) \to S \to Y \times S$ to be

$$a = (a_0, a_1) \mapsto (\rho \mapsto (f(a_0\rho), a_1\rho)).$$

Objects are functions, which take an initial state and return an output in X together with a final state.

- The unit $\eta_X : X \to S \to X \times S$ is given by

$$x \mapsto (\rho \mapsto (x, \rho)).$$

i.e. a neutral object leaves the state unchanged.

- The join operation $\mu_X : [S \to (S \to X \times S) \times S] \to S \to X \times S$ is given by

$$F \mapsto (\rho \mapsto (F_0\rho)(F_1\rho))$$

i.e. two state functions are combined by feeding one into the other.

How do we bind an object $a : S \to X \times S$ to a function $f : X \to S \to Y \times S$ to obtain an object of type $S \to Y \times S$?

The composition of Tf with the join operation μ_Y is given by

$$a \mapsto (\rho \mapsto (f(a_0\rho), a_1\rho)) \mapsto (\rho \mapsto f(a_0\rho)(a_1\rho)).$$

So the binding operation takes the output state of TX and gives it to $X \to TY$. We imagine the overall computation as follows:

1. take some initial state ρ and feed it into $a \in TX$ to obtain a value $a_0\rho \in X$ and a new state $\rho_1 := a_1\rho$.

2. apply f to $a_0\rho$ and feed in the intermediate state ρ_1 to obtain a final value $f(a_0\rho)(\rho_1)_0$ together with a final state $\rho_2 := f(a_0\rho)(\rho_1)_1$.

PART 1

A denotational complexity semantics for higher type programs

This work has been in the pipeline for some time, and grew out of the following simple question:

What is the complexity of a higher order functional?

Complexity is a vast subject, but surprisingly little of it concerns higher order objects. In fact, even when studying the complexity of a higher-order programming language, often the focus is only on type 1 programs and so the issue of higher-order complexity is circumvented (usually in a rather clever way).

But there are a number of very interesting semantics of higher-order programming languages which *do* provide a definition of complexity for all finite types. These are typically (implicitly or explicitly) based on the complexity monad we have just seen.

My interest is in generalising these semantics and setting them in a uniform framework.

For simplicity, imagine we are working with a simple call-by-value functional language. The main ideas could be adapted to other contexts.

Let $e : \texttt{nat}$ be some closed expression such that $e \to^* \underline{n}$.

Normally we interpret e as the natural number represented by the numeral $\underline{n}$:

$$[\![e]\!] = n.$$

But what if we also want information on the *cost* of evaluating e? Suppose that $e \to^k \underline{n}$.

Then we could interpret e as a pair, corresponding to a cost and a value i.e.

$$[e] = (k, n).$$

Now suppose that $t : \mathtt{nat} \to \mathtt{nat}$ is a closed expression and $t \to^* \lambda x.s(x)$.

Normally we interpret t as a function $f : [\![\mathtt{nat}]\!] \to [\![\mathtt{nat}]\!]$ such that if $e \to^* \underline{n}$ and $s(\underline{n}) \to^* \underline{m}$ then $f(n) = m$ i.e.

$$[\![te]\!] = [\![t]\!][\![e]\!] = f(n) = m.$$

But what about the complexity of t? Suppose that $t \to^l \lambda x.s(x)$. Then we could define $[t] = (l, f)$.

But we also want information about the complexity of s. Suppose that $s(\underline{n}) \to^{c(n)} \underline{m}$. Then we could define

$$[t] = (l, \underbrace{\lambda n.(1 + c(n), f(n))}_{\text{`size'}})$$

In particular, this definition is *compositional* i.e. we can compute $[te]$ from $[t]$ and $[e] = (k, n)$:

$$[te] = [t] \star [e] = [t] \star (k, n) = (k + l + 1 + c(n), f(n)) = (k + l + 1 + c(n), m).$$

What is the complexity of a higher-order functional? Let's work with a concrete example $\texttt{map} : (\texttt{nat} \to \texttt{nat}) \times \texttt{nat}^* \to \texttt{nat}^*$ defined by

$$\texttt{map}(h, []) \to [] \qquad \texttt{map}(h, x :: a) \to h(x) :: \texttt{map}(h, a)$$

The term $\texttt{map}$ is already in normal form, so $[\texttt{map}] = (0, \_)$. What is the 'size' of $\texttt{map}$?

Suppose $\texttt{map}$ takes as arguments a value $v : \texttt{nat} \to \texttt{nat}$ of size (c, f) and a list of numerals $[\underline{a}_1, \ldots, \underline{a}_j]$. Then

$$\texttt{map}(v, [\underline{a}_1, \ldots, \underline{a}_j]) \to^{1+j+\sum_{i \leq j} c(a_i)} [f(a_1), \ldots f(a_j)].$$

So we could define

$$[\texttt{map}] = (0, \lambda(c, f), \underline{a}.(1 + |\underline{a}| + \sum c(a_i), [f(a_1), \ldots, f(a_n)]))$$

and we would have $[\texttt{map}(t, e)] = [\texttt{map}] \star ([t], [e])$.

Underlying all of this is the notion of a *monadic translation*. Define $[-]$ on types as

$$[D] := C \times \underbrace{[\![D]\!]}_{s(D)}$$

$$[X \to Y] := C \times \underbrace{(s(X) \to [Y])}_{s(X \to Y)}$$

For all types we have $[X] = C \times s(X)$, the idea being that the C is some structure which contains intensional information about objects $t : X$, while $s(X)$ represents a 'size' or *potential* (at ground types the usual denotation).

- In a traditional denotational semantics, we would have (at base types):

$$\text{Whenever } e \to^* \underline{n} \text{ then } [\![e]\!] = n.$$

- Our denotational semantics aims to capture something more, for example:

$$\text{Whenever } e \to^k \underline{n} \text{ then } [e] = (k, n).$$

Example I. A strict semantics.

$C := \{\mathbf{1}, \bot\}$, and $[t]$ is given by

$$\begin{aligned}
[x]\,\rho &:= (\mathbf{1}, \rho(x)) \\
[\mathsf{0}]\,\rho &:= (\mathbf{1}, 0) \\
[\mathsf{s}]\,\rho &:= (\mathbf{1}, \lambda n.(\mathbf{1}, n+1)) \\
[\lambda x.t]\,\rho &:= (\mathbf{1}, \lambda a.\,[t]\,\rho^a_x) \\
[ts]\,\rho &:= (\mathsf{AND}([t]_0\,, [s]_0\,, ([t]_1\,[s]_1)_0), ([t]_1\,[s]_1)_1) \\
[fx]\,\rho &:= [r]\,\rho
\end{aligned}$$

for recursive functions $fx \to r$.

The intensional part captures termination: If $e \to^* n$ then $[e] = (\mathbf{1}, n)$ and vice versa.

Example IIa. An exact cost semantics.

$C := \mathbb{N}_\perp$, and $[t]$ is given by

$$\begin{aligned}
[x]\,\rho &:= (0, \rho(x)) \\
[\mathtt{0}]\,\rho &:= (0, 0) \\
[\mathtt{s}]\,\rho &:= (0, \lambda n.(0, n+1)) \\
[\lambda x.t]\,\rho &:= (0, \lambda a.\,[t]_+\,\rho^a_x) \\
[ts]\,\rho &:= ([t]_0 + [s]_0 + ([t]_1\,[s]_1)_0, ([t]_1\,[s]_1)_1) \\
[fx]\,\rho &:= [r]_+\,\rho
\end{aligned}$$

for recursive functions $fx \to r$.

The intensional part captures cost: If $e \to^k n$ then $[e] = (k, n)$ and vice versa.

Example IIb. A bounded cost semantics.

$C := \mathbb{N}_\perp$, and $[t]$ is given by

$$
\begin{aligned}
[x]\,\rho &:= (0, \rho(x)) \\
[\mathtt{0}]\,\rho &:= (0, 0) \\
[\mathtt{s}]\,\rho &:= (0, \lambda n.(0, n+1)) \\
[\lambda x.t]\,\rho &:= (0, \lambda a.\, [t]_+\, \rho^a_x) \\
[ts]\,\rho &:= ([t]_0 + [s]_0 + ([t]_1\, [s]_1)_0, ([t]_1\, [s]_1)_1) \\
[fx]\,\rho &:= \bigvee [r]_+\, \rho
\end{aligned}
$$

for recursive functions $fx \to r$.

The intensional part bounds the cost: If $e \to^k n$ then $[e] = (l, n)$ with $k \leq l$ and vice versa.

I pick these three because they have each been studied by different people and in different contexts.

Strict semantics: If $[e]_0 = \mathbf{1}$ then $[e]$ can be reduced to a normal form - adequacy results of this kind are proven by Berger (2005) and are used to establish strong normalisation of λ-calculi extended with bar recursion operators.

Exact costs: Denotational cost semantics first explored by Sands (1990) among others, generalised and lifted to a categorical setting by Van Stone (2003).

Bounded costs: A cost semantics which is sound w.r.t. a higher-type *bounding* relation $\sqsubseteq$ is studied for variants of system T by Danner et al. (2012 & 2015). Extended to call-by-name PCF by Kim (2016).

Problem. In general, soundness and particularly adequacy seem to be difficult to prove: The more complex the relationship between $t : X$ and the component $[t]_0 \in C$, the more intricate and messy the resulting induction tends to be.

Can we give a uniform framework and adequacy proof which captures a wide range of monadic translations, including those which bound the cost of programs?

Proofs of this kind typically have

- an important combinatorial part - does the translation work for the building blocks of our language?
- a quite technical but rather uniform domain-theoretic part verifying that it works for arbitrary terms.

Therefore it makes sense to *seperate* these parts if possible.

$$\text{Adequacy proof} = \underbrace{\text{Combinatorial part}}_{\text{easy to check}} + \underbrace{\text{Domain-theoretic part}}_{\text{uniform}}$$

Recall that

$$[D] := C \times \underbrace{[\![D]\!]}_{s(D)}$$

$$[X \to Y] := C \times \underbrace{(s(X) \to [Y])}_{s(X \to Y)}$$

Suppose that

- $I_X(e, c)$ is an arbitrary 'cost' relation between closed terms $e : X$ and total objects of $c \in C$ while
- $S_D(v, s)$ is a 'size' relation between values of type D and $s \in [\![D]\!]$ defined at all ground types.

Define the relation $P_X(e, \alpha)$ between closed terms $e : X$ and $\alpha \in [X]$ as follows:

$$P_D(e, \alpha) := \alpha_0 \neq \bot \Rightarrow \exists v(e \to^* v \wedge I_D(e, \alpha_0) \wedge S_D(v, \alpha_1))$$

$$P_{X \to Y}(e, \alpha) := \alpha_0 \neq \bot \Rightarrow \exists v \left(\left\{ \begin{array}{l} e \to^* v \wedge I_{X \to Y}(e, \alpha_0) \\ \wedge \underbrace{\forall w, \beta(S_X(w, \beta) \Rightarrow P_Y(vw, \alpha_1 \beta))}_{S_{X \to Y}(v, \alpha_1)} \end{array} \right. \right)$$

All previous translations are simple instances of this. In particular:

Strict semantics:

- $C = \{\mathbf{1}, \bot\}$
- $I_X(e, \mathbf{1})$ always true,
- $S_{\mathtt{nat}}(\underline{n}, m) := (n = m)$
- $P_X(e, \alpha) \Leftrightarrow (\alpha_0 = \mathbf{1} \Rightarrow \exists v(e \to^* v \wedge \alpha_1 \approx [\![v]\!]))$

where $\alpha_1 \approx [\![v]\!]$ can be read as α_1 is 'strictly denoted' by $[\![v]\!]$.

Bounded costs:

- $C = \mathbb{N}_\bot$
- $I_X(e, k) := \forall e'(e \to^i e' \to i \leq k)$
- $S_{\mathtt{nat}}(\underline{n}, m) := (n \leq m)$
- $P_X(e, \alpha) \Leftrightarrow (\alpha_0 \neq \bot \Rightarrow \exists v(e \to^k v \wedge k \leq \alpha_0 \wedge v \sqsubseteq \alpha_1))$

where $\sqsubseteq$ is a essentially the bounding relation of Danner et al. (2012 & 2015).

Aim. A general semantics of the form

$$
\begin{aligned}
[x]\,\rho &:= (c_x, \rho(x)) \\
[\mathtt{0}]\,\rho &:= (c_{\mathtt{0}}, 0) \\
[\mathtt{s}]\,\rho &:= (c_{\mathtt{s}}, \lambda n.(c'_{\mathtt{s}}, n+1)) \\
[\lambda x.t]\,\rho &:= (c_{\lambda x.t}, \lambda a.\Phi_t([t]\,\rho^a_x)) \\
[ts]\,\rho &:= (m([t]_0\,, [s]_0\,, ([t]_1\,[s]_1)_0), ([t]_1\,[s]_1)_1) \\
[fx]\,\rho &:= \Psi_f([r]\,\rho)
\end{aligned}
$$

for recursive functions $fx \to r$, where

- c_x, $c_{\mathtt{0}}$, $c_{\mathtt{s}}$ and $c_{\lambda x.t}$ are elements of a 'cost domain' C;
- $m : C \times C \times C \to C$ is a continuous function;
- Φ_t and Ψ_f are continuous functions $[X] \to [X]$, where $r, t : X$.

We want a set of conditions on these components in terms of I_X and $S_{\mathtt{nat}}$ such that:

Theorem. For all closed terms $e : X$ we have $P_X(e, [e])$.

The difficultly in proving a theorem of this kind for arbitrary terms lies in the fact that we allow arbitrary (potentially non-terminating) recursive functions. However, we can initially avoid this by looking at finitary systems with *bounded* recursion (via bounded fixpoints $\mathbf{fix}_n$ or stratified rewrite systems $f_n x \to r_{(n-1)}$).

Let $e_{(n)}$ denote e with all function symbols replaced by f_n.

Lemma (combinatorial part) For all closed terms $e_{(n)} : X$ we have $P_X(e_{(n)}, [e_{(n)}])$.

Proof. Induction on n and typing of e - it's here that we do the important work.

Lemma (domain-theoretic part) Suppose that $[e]_0 \neq \bot$. Then there is some n such that $[e_{(n)}]_0 = [e]_0$ and $[e_{(n)}]_1 \sqsubseteq [e]_1$.

Proof. Standard.

Theorem. For all closed terms $e : X$ we have $P_X(e, [e])$.

Some results (from the drawer)

- Extension of existing cost semantics. In particular we can generalise bounding relation of Danner et al. to a standard call-by-value higher order language with arbitrary recursion.
- Provide a uniform framework in which a variety of cost semantics can be understood.
- Enable one to obtain *new* monadic denotational semantics for which soundness and adequacy can be easily verified.

This is all work in progress! However the main goal for the future would be to utilise the translations to *analyse* programs. For example:

- Can we automatically solve the extracted recursive equations which e.g. characterise cost of a program?
- Can we give a set of conditions which guarantee that this cost functional can be defined in a weak system?

PART 2

A variant of Gödel's functional interpretation with state

This work is slightly more recent, and arises from the following simple question:

What is the computational meaning of a non-constructive proof?

Many answers to this question are centered around the notion of 'learning'.

- The epsilon calculus
- Coquand's semantics of evidence
- Avigad's update procedures
- Aschieri and Berardi's learning realizability.

I have always wanted to relate my favourite proof interpretation - Gödel's functional intepretation - to those above, and in particular highlight that underneath its syntax, the functional interpretation conceals an extremely elegant semantics based on learning.

My favourite simple example: The drinkers paradox

What is the computational meaning of the following classical statement?

$$\exists x \forall y (P(x) \to P(y))$$

Don't worry if you have never seen the functional interpretation - it's actually quite intuitive!

We first double negate and then use the functional interpretation to Skolemise:

$$\begin{aligned} &\neg\neg\exists x \forall y (P(x) \to P(y)) \\ &\rightsquigarrow \neg\exists f \forall x \neg (P(x) \to P(fx)) \\ &\rightsquigarrow \forall f \exists x (P(x) \to P(fx)). \end{aligned}$$

In other words, while we cannot compute an 'ideal' witness for $\exists x$ which is valid for all y, we can compute an 'approximate' witness for $\exists x$ which is valid relative to some function f.

Such an approximation can be computed by the following simple program:

$$Xf := \mathrm{case}(P(f0), 0, f0).$$

There are two possibilities:

1. $P(f0)$ is true, in which case

$$Xf \downarrow 0 \quad \text{and} \quad P(0) \to P(f0),$$

2. $P(f0)$ is false, in which case

$$Xf \downarrow f0 \quad \text{and} \quad P(f0) \to P(f(f0)).$$

However, our program is only executable if the predicate $P(x)$ is decidable. This is a well-known problem with the original variant of the functional interpretation, which has since been circumvented in various ways.

I propose a new solution.

While we're at it, I should list together some other issues with the original functional interpretation which have been highlighted in various places:

- As already mentioned, computationally neutral formulas need to be decidable - this is not the case for e.g. predicate logic, set theory, nonstandard analysis.
- Definition by case functionals are difficult to interpret categorically.
- Extracted programs can be highly inefficient, and end up checking the same cases repeatedly.
- The *meaning* of an extracted program typically lies in how it interacts with the 'mathematical environment' via case distinctions, but with the traditional presentation this is difficult to visualise.

The first two problems are traditionally solved by the Diller-Nahm interpretation, which collects a finite list of potential witnesses.

Alternative solution: Keep case distinctions, but in the form of interactions with a global state which represents the mathematical environment.

Back to our simple example... Suppose that we have a global state S which at any one time contains a finite list of atomic formulas, and that definition by case constants are replaced by a query to the global state, governed by some mapping $\sigma : \mathit{Environment} \to \{\mathbf{true}, \mathbf{false}\}$. Let

$$Xf[\pi] := \mathrm{query}_{\pi}(P(f0), 0, f0).$$

There are three possibilities.

1. $P(f0) \notin \mathrm{dom}(\pi)$ and the state accepts $P(f0)$ i.e. $\sigma(P(f0)) = \mathbf{true}$:

$$\langle \pi \mid Xf \rangle \downarrow \langle \pi :: P(f0) \mid 0 \rangle \quad \text{and} \quad \pi \wedge P(f0) \to (P(0) \to P(f0)).$$

2. $P(f0) \notin \mathrm{dom}(\pi)$ and the state rejects $P(f0)$ i.e. $\sigma(P(f0)) = \mathbf{false}$:

$$\langle \pi \mid Xf \rangle \downarrow \langle \pi :: \neg P(f0) \mid f0 \rangle \quad \text{and} \quad \pi \wedge \neg P(f0) \to (P(f0) \to P(ff0)).$$

3. $P(f0) \in \mathrm{dom}(\pi)$ and the state uses its existing knowledge:

$$\langle \pi \mid Xf \rangle \downarrow \langle \pi \mid x \rangle \quad \text{and} \quad \pi \to (P(x) \to P(fx))$$

where $x = 0$ or $f0$ depending on π.

A more general idea:

Traditionally, if

$$\mathcal{T} \vdash \forall x \exists y A(x, y)$$

then we can extract a function $f : X \to Y$ such that

$$\mathcal{T} \vdash \forall x A(x, f(x)).$$

We develop a new soundness proof so that instead we extract a program $f : X \to S \to Y \times S$ (i.e. $X \to TY$ for state monad T) satisfying

$$\mathcal{T} \vdash_{\sigma} \forall x, \pi(f(x)[\pi]_1 \to A(x, f(x)[\pi]_0)),$$

which is valid for *any* state function σ.

This is an extension of the usual functional interpretation:

- If quantifier-free formulas in $\mathcal{T}$ are decidable, then we can just define state queries to accept only true formulas i.e. $\sigma(A) := \chi_A$, and then

$$\mathcal{T} \vdash_\chi \forall x(f(x)[\,]_1 \to A(x, f(x)[\,]_0))$$

and since $f(x)[\,]_1$ is always true and so we regain the usual functional interpretation with realizer $\lambda x.f(x)[\,]_0 : X \to Y$.

- More generally, for any fixed theorem we can range over all possible states σ and come up with a finite sequence of possible realizers (i.e. a Herbrand disjunction). For example

$$\mathcal{T} \vdash_{\sigma_0} P(f0) \to (P(0) \to P(f0))$$
$$\mathcal{T} \vdash_{\sigma_1} \neg P(f0) \to (P(f0) \to P(f(f0)))$$

and therefore $(P(0) \to P(f0)) \vee (P(f0) \to P(f(f0)))$.

I claim that a functional interpretation with global state addresses all of the aforementioned problems:

- We do *not* require decidability of quantifier-free formulas: The state is simply given instructions, and collects a list of formulas which it supposes to be true. In case we do have decidability, we regain the usual functional interpretation, and if not, we can produce some variant of Herbrand's theorem.
- We isolate definition by case functionals as a side-effect, which can be modelled as a monad. This may lead to an elegant categorical semantics of the functional interpretation.
- Efficiency: A given formulas is only ever checked once. From this point onwards it is recorded in the state, which is referenced every time there is a query. Moreover, we can refine how the state works to improve efficiency e.g. by adding additional rules of inference.
- The learning semantics implicit in the functional interpretation is now presented completely transparently. When a realizer evaluates, it returns a value together with a final state which contains everything it has 'learned' about the mathematical environment during a computation.

Example A: Ramsey's theorem for pairs

Classical statement: For any colouring $c : \mathbb{N} \times \mathbb{N} \to \{0, 1\}$, there exists an infinite set $X \subseteq \mathbb{N}$ that is pairwise monochromatic.

Finitized statement: For any colouring $c : \mathbb{N} \times \mathbb{N} \to \{0, 1\}$ and functional $\varepsilon : \mathcal{P}(\mathbb{N}) \to \mathbb{B}$, there exists a finite approximation $X_\varepsilon \subseteq \mathbb{N}$ to a monochromatic set, which is valid up to the point $\varepsilon(X_\varepsilon)$.

From the classical proof of Ramsey's theorem, we would extract a program $X_\varepsilon : S \to \mathcal{P}(\mathbb{N}) \times S$, which from an initial empty state [] would result in a computation

$$\langle [\,] \mid \emptyset \rangle \to \langle \pi_1 \mid X_1 \rangle \to \ldots \to \langle \pi_n \mid X_n \rangle$$

where $\pi_n \to$ 'X_n a sufficiently good approximation'. Here,

- the X_i are finite subsets of $\mathbb{N}$;
- the π_i is the current state, which contains atomic formulas of the form $c(m, n) = b$.

Example B: Bolzano-Weierstrass theorem for rationals

Classical statement: Any sequence (x_i) of rationals in the $[0,1]$ contains a subsequence (x_{gi}) which converges to some real number $\alpha \in [0,1]$.

Finitized statement: Given some sequence (x_i) in $[0,1]$ and functional $\varepsilon : \mathbb{R} \times \mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ there exists some pair $\alpha_\varepsilon \in [0,1]$ and $g_\varepsilon : \mathbb{N} \to \mathbb{N}$ such that the finite subsequence $x_{g_\varepsilon 0}, \ldots, x_{g_\varepsilon N}$ satisfies $|x_{g_\varepsilon i} - \alpha_\varepsilon| < 2^i$ for all $i \leq N$, where $N := \varepsilon(\alpha_\varepsilon, g_\varepsilon)$.

From the classical proof of the theorem, we would extract a program $\alpha_\varepsilon, g_\varepsilon : S \to \mathbb{R} \times \mathbb{N}^{\mathbb{N}} \times S$, which from an initial empty state [] would result in a computation

$$\langle [\,] \mid 0, \mathrm{id} \rangle \to \langle \pi_1 \mid \alpha_1, g_1 \rangle \to \ldots \to \langle \pi_n \mid \alpha_n, g_n \rangle$$

where $\pi_n \to$ 'α_n, g_n are sufficiently good approximations'. Here,

- the α_i and g_i are (finitely defined) reals and functions respectively;
- the π_i is the current state, which contains atomic formulas of the form $x_n \in [q_1, q_2]$ for $q_1, q_2 \in \mathbb{Q}$.

I hope to put all of this into writing at some point... (famous last words)

Thank You!