Functional interpretations with imperative features

Thomas Powell

Technische Universität Darmstadt

Workshop on Mathematical Logic: Proof Theory, Constructive Mathematics

Mathematisches Forschungsinstitut Oberwolfach

6 November 2017

Proof interpretations allow us to give a computational interpretation to mathematical statements.

$$\text{If } \mathcal{T} \vdash A \text{ then } \mathcal{S} \vdash \forall x A_I(x, tx), \text{ where:}$$

- $A_I(x, y)$ is computationally neutral;
- $A \leftrightarrow \forall x \exists y A_I(x, y)$;
- t is a term of $\mathcal{S}$ formally extracted from the proof of A.

The world of proof interpretations includes:

- Foundational problems: $\mathrm{Con}(\mathcal{S}) \Rightarrow \mathrm{Con}(\mathcal{T})$.
- Proof mining: New quantitative results in numerical analysis, ergodic theory, convex optimization...
- Semantics: New categorical models of linear logic, semantics of classical logic.
- Formal program extraction: Implementation of proof interpretations in Minlog, Agda...

Can we use proof interpretations to synthesise verified programs?

Applications of this kind have been less successful, and come with a set of problems which are less prominent in other areas:

- **Algorithmic structure** In many cases we still don't really understand how programs extracted from proofs 'work' i.e. what their *algorithmic* content is.

- **Language** Extracted programs are typically expressed as terms in some abstract lambda calculus, which is quite different from the style in which 'normal' programs are written.

- **Usefulness** Why would a programmer use the functional interpretation to extract a list reversal program? Proof interpretations are typically powerful when applied to complex proofs whose computational meaning is not obvious...

A more concrete illustration of the problem...

Theorem (Higman): For any well quasi-order $(X, \leq)$, the set of words $(X^*, \leq_*)$ under the embeddability relation is a well quasi-order.

Proof: Infinite Ramsey's theorem + Least element principle + dependent choice / Zorn's lemma (in the form of open induction).

(P. 2012): The construction of a functional Φ in System T + bar recursion such that for any sequence of words u,

$$\Phi_0 u < \Phi_1 u \wedge u_{\Phi_0 u} \leq_* u_{\Phi_1 u}.$$

I still don't really understand how this term works...

One of my current interests is to understand Gödel's functional interpretation of strong classical theories, using concepts from imperative programming such as

- global state;
- monadic transformations;
- control flow statements;
- Hoare logic.

Why?

1. Applications of proof theory in computer science should make use of programming paradigms which are used in practice.

2. The above concepts provide us with a natural means of understanding the functional interpretation of non-trivial classical principles.

A outline of the talk:

First, I will sketch a very small idea:

A functional interpretation with global state.

Possibly this already exists (at least implicitly) somewhere else...

Second, I want to present some more difficult open questions.

Warning: None of this is published!

What is the computational meaning of the so-called Drinkers paradox?

$$\exists x(P(x) \to \forall y P(y)) \qquad P \text{ quantifier-free}$$

There is no *effective* way to find a witness for x, as its existence depends on the law of excluded-middle.

However, over classical logic and quantifier free choice we have the following series of equivalences:

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

Ineffective statement: *There exists some ideal drinker x such that if x drinks, then all people y drink.*

Effective reformulation: *For any function f there exists an approximate drinker x such that if x drinks, then person fx drinks.*

Gödel's functional interpretation (of classical logic) is a systematic way of doing this:

$$A \mapsto \forall z \exists x A_I(z, x)$$

In our example,

- $A :\equiv \exists x(P(x) \to \forall y P(y))$
- $A_I(f, x) :\equiv P(x) \to P(fx)$.

Theorem (Gödel 1958):

$$\text{If } \mathsf{PA} \vdash A \text{ then System T} \vdash \forall z A_I(z, tz)$$

where t is a term of System T formally extracted from the proof of A.

In our example, we want a term t satisfying

- $A_I(f, tf) \equiv P(tf) \to P(f(tf))$ for all f.

GOAL:

$$P(tf) \to P(f(tf))$$

$$\text{Define} \quad tf := \mathsf{case}(P(f0), 0, f0).$$

If $P(f0)$ is true:

$$tf = \mathsf{case}(P(f0), 0, f0) \to 0$$

$$\text{and}(P(\underbrace{0}_{tf}) \to P(f(\underbrace{0}_{tf})) \checkmark$$

If $P(f0)$ is false:

$$tf = \mathsf{case}(P(f0), 0, f0) \to f0$$

$$\text{and}(P(\underbrace{f0}_{tf}) \to P(f(\underbrace{f0}_{tf})) \checkmark$$

Case distinctions are a fundamental feature of programs extracted using the functional interpretation. Formally, they are required to interpret *contraction*

$$\frac{A \wedge A \to B}{A \to B}$$

Therefore in practice, if A is a mathematical theorem, then a program t satisfying $\forall z A^*(z, tz)$ will a complex term with numerous case distinctions, representing each instance of contraction in the formal proof.

Why do we care about this?

One can think of case distinctions as being

'interactions with the mathematical environment'

which give us a high-level description of how our extracted program behaves.

Suppose that we extend our lambda calculus with a new type S, which represents a *global state*.

For our purposes, this global state contains knowledge about our 'environment', and is something that *we* define. For example:

Mathematical Context	Elements of state
A finite list s	$s_i \leq s_j$
A sequence $(x_n)_{n\in\mathbb{N}}$ of rationals in $[0,1]$	$x_n \in [p,q]$
A first order logic	$P(x_1,\ldots,x_n)$

Objects $\pi \in S$ are finite list of elements or their negations e.g.

$$\pi = [(a_1 \leq a_2), \neg(a_3 \leq a_5), (a_1 \leq a_4)]$$

Previously we had: $\mathsf{case}(P, s, t) \to s$ if P true $\quad$ $\mathsf{case}(P, s, t) \to t$ if P false

IDEA: Rather than use case distinctions, extracted programs access and collect information via the global state.

At a given point in time, our global state will be a finite list of 'accepted' elements $\pi := [P_1, \ldots, P_k]$. We can interact with the state via the function

$$[\pi \mid \mathsf{askstate}(P, s, t)] \to \begin{cases} [\pi \mid s] & \text{if } P \in \pi \\ [\pi \mid t] & \text{if } \neg P \in \pi \\ [\pi :: P \mid s] \text{ or } [\pi :: \neg P \mid t] & \text{otherwise} \end{cases}$$

where we leave open for now how the 'or' is interpreted. For example:

- If elements of the state are decidable, we could simulate case distinction by choosing whichever of P or $\neg P$ is true.
- We could introduce a 'state function' σ : State elements $\to \{0, 1\}$.
- We could choose at random.

Back to our running example:

$$P(tf) \to P(f(tf))$$

Define $\quad tf := \mathsf{askstate}(P(f0), 0, f0).$

$\underline{P(f0) \in \pi}$ then $[\pi \mid tf] \to [\pi \mid 0]$ and $\bigwedge \pi \to (P(0) \to P(f0))$

$\underline{\neg P(f0) \in \pi}$ then $[\pi \mid tf] \to [\pi \mid f0]$ and $\bigwedge \pi \to (P(f0) \to P(f(f0)))$

$\underline{\text{Otherwise}}$, either $[\pi \mid tf] \to [\pi :: P(f0) \mid 0]$ and

$$\bigwedge \pi \wedge P(f0) \to (P(0) \to P(f0))$$

or alternatively $[\pi \mid tf] \to [\pi :: \neg P(f0) \mid f0]$ and

$$\bigwedge \pi \wedge \neg P(f0) \to (P(f0) \to P(f(f0)))$$

So how would this generalise?

Traditional soundness

If $\mathcal{T} \vdash A$ then we can extract a term $t : X \to Y$ such that

$$\mathcal{S} \vdash \forall x A_I(x, tx).$$

New soundness (much simplified...)

If $\mathcal{T} \vdash A$ then we can extract a state sensitive term $t : X \to S \to S \times Y$ such that

$$\mathcal{S} \vdash \forall x, \pi \left(\bigwedge t_0 x \pi \to A_I(x, t_1 x \pi) \right).$$

where

- π is the input state;
- $t_0 x \pi$ is the output state which contains additional information;
- $\lambda x.t_1 x \pi$ is our state sensitive realizing function.

Example: 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 π_0 would result in a computation

$$[\pi_0 \mid \emptyset] \to [\pi_1 \mid X_1] \to \ldots \to [\pi_n \mid X_n]$$

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 $c(m, n) = b$ (or their negations).

This is actually a generalisation of the usual functional interpretation: If all state elements are decidable, we just add true formulas to our state. Then setting $\pi = []$ we have

$$\mathcal{S} \vdash \forall x \bigwedge t_0 x[] \quad \text{and} \quad \mathcal{S} \vdash \forall x \left(\bigwedge t_0 x[] \to A_I(x, t_1 x[]) \right)$$

and so $\mathcal{S} \vdash \forall x A_I(x, t_1 x[])$.

On the other hand, for a fixed proof we are able to compute a finite sequence of witnesses by restricting ourselves to a finite set E of relevant state elements and considering $2^{|E|}$ state functions $E \to \{0, 1\}$, from which we obtain (assuming $\mathcal{S}$ admits some excluded-middle):

$$\mathcal{S} \vdash \forall x \bigvee \bigwedge t_0^i x[] \quad \text{and} \quad \mathcal{S} \vdash \forall x \left(\bigwedge t_0^i []x \to A_I(x, t_1^i []x) \right)$$

for $i = 1, \ldots, 2^{|E|}$, and therefore $\mathcal{S} \vdash \forall x \exists y \in [t_1^1 x[], \ldots, t_1^{2^{|E|}} x[]] A_I(x, y)$.

In our running example, $E = \{P(f0)\}$, $t_1^1 f[] = 0$ and $t_1^2 f[] = f0$ and in particular

$$\exists x \in [0, f0](P(x) \to P(f(x))).$$

But unifying functional interpretations is not the main goal here...

- The state helps isolate the algorithm underlying the extracted program, which can then be analysed. For a program on lists, we could choose the state so that it lists the comparisons $s_i \leq s_j$ which took place. We might have $f_1 s\pi = g_1 s\pi$ but
$$|f_0 s\pi| < |g_0 s\pi|,$$
and so f would be more efficient than g.
- The setting allows us to refine the functional interpretation. We can avoid repeated case distinctions by first checking whether or not P is in the domain of π. We can also impose logical relations on the state e.g.
$$\text{if } (s_i \leq s_j), (s_j \leq s_k) \in \pi \text{ then infer } (s_i \leq s_k)$$
- By writing extracted programs in a calculus with a state, we move a step closer to producing programs in the kind of language that programmers actually use.

Further thoughts and open questions...

Everything earlier was greatly simplified - in particular reasoning about a state at higher types is a bit more complicated...

One well-known way of formalising interactions with a state is via the state monad:

$$X \mapsto TX := S \to S \times X.$$

For Π_2-formulas $A :\equiv \forall x \exists y A_0(x, y)$

- We would normally extract a term $t : X \to Y$ satisfying $A_0(x, tx)$.
- We now extract a term $t : X \to TY$ satisfying

$$\underbrace{\forall \pi \left(\bigwedge t_0 x \pi \to A_0(x, t_1 x \pi) \right)}_{A_T(x,tx)}$$

More generally, we are applying to a monadic translation to types, given by

$$[D] := D \text{ for base types, and } [X \to Y] := [X] \to T[Y]$$

and this results in a corresponding monadic functional interpretation

$$A \mapsto A_T(\bar{x}, t\bar{x})$$

where now $t : [X] \to T[Y]$, and

$A_T(\bar{x}^{[X]}, \bar{y}^{[Y]})$ represents $A_I(x^X, y^Y)$ plus an additional monadic constraint

QUESTION: Is there a clean, abstract presentation of the above?

QUESTION: Are there other, concrete instances of T which are of interest to us (complexity, perhaps)?

QUESTION: How does this functional interpretation with state compare to

- 1. Other abstract treatments of functional interpretations (both syntactic and semantic);
- 2. Other 'learning based' computational interpretations e.g. interactive realizability of Aschieri-Berardi, epsilon calculus (!)

QUESTION: Can we extend this simple interpretation with state to a fully fledged imperative functional interpretation? Is this a more natural language for expressing the computational content of classical principles?

$$\text{Logic} + \text{Principles} \vdash A \Rightarrow \text{Functional calculus} \vdash A_I(x, tx)$$

where t is naturally expressed as some functional program.

$$\begin{aligned} \text{Logic} &\sim \lambda\text{-calculus} \\ \text{Induction} &\sim \text{Primitive recursion} \\ \text{Choice} &\sim \text{Bar recursion} \end{aligned}$$

Over classical logic, one can look at this slightly differently as

$$\begin{aligned} \text{Logic} &\sim \text{Basic operations} \\ \text{Least element principles} &\sim \text{Wellfounded loops} \\ \text{Zorn's lemma} &\sim \text{Self-referential loops} \end{aligned}$$

This is made more precise in (P. 2016) and (P. 2017).

QUESTION: Is something like the following possible?

$$\mathcal{T} \vdash A \Rightarrow \text{Hoare logic} \vdash \{P, i := x\}\ C\ \{Q, i := x, j := y, A_I(i, j)\}$$

- We would need a suitable higher-order variant of Hoare logic, which would include special while-rules for dealing with non-terminating loops which replace bar recursion.
- Programs would be written in an imperative style, and would return not just a realizer, but a final state which records interesting information about how the program was evaluated.
- Does not need to *replace* the usual interpretation - could combine it to extract programs in a hybrid language with both functional and imperative features.

To summarise:

QUESTION: How do extracted programs behave? What algorithm does my extracted program implement?

This is important for two reasons in particular:

1. When I want to synthesise a simple program (e.g. sorting a list), and want to formalise a claim that it carries out an efficient algorithm.

2. When I extract a complicated program I want to understand what it does!

Thank you!