Proof interpretations with imperative features

Thomas Powell

Technische Universität Darmstadt

Minisymposium on Applied Proof Theory and the Computational Content of Mathematics

Joint Congress of the ÖMV and DMG, Salzburg

14 September 2017

WARNING: The usual text to formula ratio is inverted in this talk!

1. Some general remarks

2. A functional interpretation with global state (A tiny step towards applications in computer science)

3. Some slightly bigger steps towards applications in computer science

1. Some general remarks

Proof interpretations allow us to extract programs from proofs:

$$\text{if } \mathcal{P} \text{ derives } \forall x \exists y A_0(x, y) \text{ then } \mathcal{P}' \text{ derives } \forall x A_0(x, f(x))$$

where f and $\mathcal{P}'$ are obtained formally from $\mathcal{P}$.

An old idea

Suppose that:

- $A_0(x, y)$ is the specification of a program i.e. how the input x should be related to the output y;
- $\mathcal{P}$ is a proof that a program satisfying the specification exists.

Then we can extract:

1. An actual program f satisfying the specification;
2. A verification proof $\mathcal{P}'$;
3. A bound on the complexity of f;

and so on...

So why aren't all programmers using proof interpretations?

"Is your list sorting program as good as one a human would write?"

"We use C++. What good to us is a program written in System T?"

"Your extracted program takes up ten pages of text. How does it even work?"

"Group X can already do this and have developed an extremely successful tool."

"Could your technique for synthesising programs be easily used by someone working at the Guardian newspaper?"

These are all valid points...

... which highlight the following problems:

Efficiency: It's easy to extract a brute force algorithm, but much more difficult to produce something intelligent, comparable to what a human would write.

Language: Formally extracted programs are typically presented in an abstract language like System T. Real programming languages tend to follow a completely different paradigm, with concepts such as global state, concurrency, and so on...

Scale: Formal verification is a huge business and lots of sophisticated tools have already been developed. On top of this, most big proof assistants (Coq, NUPRL, etc) can extract programs from proofs. A small community in proof theory, dedicated to a particular style of program extraction, cannot possibly compete with this directly.

Accessibility: Ultimately, methods for synthesising verified programs are only useful if they can be used by a non-specialist.

So what useful things can those of us who study traditional proof theoretic techniques do in this direction?

- We need to understand how proof interpretations work at a very low level. What makes e.g. a data sorting program 'good' or 'efficient' is the way in which it carries out *comparisons* between elements, and very informally:

  $$\text{underlying algorithm} \sim \text{structure of contractions in formal proof}$$

- We need to find narrow areas of application in which proof theoretic tools happen to be powerful e.g. where it is easy to prove the existence of a program but hard to construct one.
- We need to adapt traditional techniques to settings in which programmers work e.g. imperative paradigms.
- We need to actively look for interesting results in mathematics and logic which are prompted by these efforts, so that our research is interesting and worthwhile in any case.

Obviously **formalization** is also crucial, but I'm not an expert on this so I won't talk about it.

A functional interpretation with global state
(A tiny step towards applications in computer science)

Imagine that we take our favourite formulation of System T (primitive recursive functionals in all finite types) and extend it 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.

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

Suppose that $\mathcal{T}$ is some classical theory and $\mathcal{S}$ an appropriate extension of system T. The normal program extraction theorem for the functional interpretation states that whenever $\mathcal{T} \vdash \forall x \in X \exists y \in Y\, A_0(x, y)$ then

$$\mathcal{S} \vdash A_0(x, f(x))$$

for some extracted $f : X \to Y$.

In our new modified interpretation, the extracted program now has type $g : S \to X \to S \times Y$. It takes as input an initial state π, together with an argument x, and returns a pair $(g_0\pi x, g_1\pi x)$ i.e.

- a final state $g_0\pi x \in S$, which is an extension of π with information learned during the computation.
- a value $g_1\pi x \in Y$.

The key idea is that whenever our realizing term has to decide the truth of an atomic formula in our environment (e.g. when interpreting a contraction), we add this information to our state. The conclusion of the program extraction theorem is now modified to

$$\mathcal{S} \vdash \bigwedge_{P \in g_0\pi x} P \to A_0(x, g_1\pi x)$$

Example. $A :\equiv \exists x \forall y (P(y) \to P(x))$.

Elements of our environment are predicates of the form $P(t)$.

The negative + functional interpretation of A is given by
$\forall \phi \exists x (P(\phi x) \to P(x))$.

We can either define $g\pi\phi := (\pi :: P(\phi 0), \phi 0)$. Then

$$\bigwedge_{Q \in \pi} Q \wedge P(\phi 0) \to (P(\phi(\phi 0)) \to P(\phi 0))$$

Or we can define $g\pi\phi := (\pi :: \neg P(\phi 0), 0)$. Then

$$\bigwedge_{Q \in \pi} Q \wedge \neg P(\phi 0) \to (P(\phi 0) \to P(0))$$

Note: If $P(x)$ is decidable then we can use this to choose which realizer to use. But we do not require decidability - in this case we have two perfectly valid realizers, which combine to form a Herbrand disjunction.

A theoretical aside: Equipping the functional interpretation with a global state gives us a uniform way of extending the interpretation to theories without decidability of quantifier-free formulas.

1. If our state elements are decidable, then whenever we carry out a definition by cases, we simply add what we have learned to our current state. Then in particular setting our initial state to $\emptyset$, every element of $g_0\emptyset x$ will be true, and therefore

$$\mathcal{S} \vdash \bigwedge_{P \in g_0 \pi x} P \to A_0(x, g_1 \pi x) \Rightarrow \mathcal{S} \vdash \forall x A_0(x, g_1 \emptyset x).$$

2. If not, there are 2^n realizers $g^1, \ldots, g^{2^n}$ where n is the number of atomic predicates in our proof, and we have

$$\mathcal{S} \vdash A_0(x, g_1^1 \emptyset x) \vee \ldots \vee A_0(x, g_1^{2^n} \emptyset x).$$

However, such results are not our main purpose - this sort of thing has already been studied in (Gerhardy & Kohlenbach 2005), (Ferreira & Ferreira 2017).

Extracted programs return a realizer, together with an output state with records all 'interactions with the environment'.

- The state helps isolate the algorithm underlying the extracted program, which can be analysed. For a program on lists, the state would list the comparisons $s_i \leq s_j$ which took place. We might have $f_1 \pi s = g_1 \pi s$ but

$$|f_0 \pi s| < |g_0 \pi s|,$$

  and so f would be more efficient than g.
- The setting allows us to easily refine the functional interpretation. We can avoid repeated case distinctions by first checking whether or not P is in the domain of π (similar to Trifonov 2011). 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 acutally use.

There is potentially some interesting category theory going on.... We are essentially applying a *monadic transformation* to realizers:

$$(X \to Y) \mapsto (X \to TY)$$

where $TY := S \to S \times X$ is the state monad.

But now is not the time to go into this...

Some slightly bigger steps towards applications in computer science

Develop a richer imperative functional interpretation, which:

1. Is capable of extracting programs written in a fully imperative language (not just a functional language with state) i.e. which look like

$$
\begin{aligned}
&i, j := 1, 1 \\
&\texttt{while}(i < n) \\
&\quad j := i * j \\
&\quad i := i + 1
\end{aligned}
$$

2. Verifies extracted programs in some higher-order variant of Hoare logic. Program extraction theorem may look like: If $\mathcal{T} \vdash \forall x \exists y A_0(x, y)$ then we can extract a program p such that

$$HL \vdash \{\pi, [i := x]\}\ p\ \{\pi', [j := y], A_0(x, y)\}.$$

There is very little precedent here: Closest are e.g. (Poernomo, Crossley & Wirsing 2005) and (Berger, Seisenberer & Woods 2013).

Understand the role that ineffective principles play in 'packaging computational content'.

Fully constructive proofs produce programs *directly*. Typically the program looks very similar to the proof.

Ineffective proofs yield programs *indirectly*. Typically the program is much more complicated than the proof.

This is becuase classical logic encodes algorithmic ideas such as backtracking, which are (in the case of the functional interpretation) unpacked by adding and eliminating double negations. So a small ineffective step can contain a lot of computational steps.

We should use this to our advantage and *embrace ineffectiveness* as a compact language in which to write complex programs.

Thank you!