Gödel's functional interpretation and the extraction of imperative programs from proofs

Thomas Powell

Technische Universität Darmstadt

Humboldt Kolleg: Proof Theory as Mathesis Universalis
Villa Vigoni, Como

25 July 2017

"Proof Theory as Mathesis Universalis"

Proof theory is often unfairly perceived to be a discipline concerned only with metamathematics, occupying a world different from that of 'real' science.

However, proof theory is about more than just formalism - it's a powerful way of thinking which can lead to genuine insights and applications in ordinary mathematics and computer science.

For this reason, it is important that proof theory is flexible, and that formal systems and techniques evolve to reflect developments in science.

The historical context of this talk

1930s Gödel develops his functional interpretation to prove (relative) consistency of Peano arithmetic.

1960s Kreisel demonstrates that proof interpretations can be used to 'unwind' witnesses from proof of existential statements.

1990s- Kohlenbach develops Kreisel's idea into the *proof mining* program, leading to new quantitative results in approximation theory, functional analysis, ergodic theory...

2000s- Applications in computer science begin to emerge, including

- The complexity analysis of termination techniques (Buchholz, Steila et al., P.),
- The formal extraction of verified programs (Berger, Schwichtenberg, ...)

My aim is to briefly introduce a new variant of Gödel's functional interpretation, based on several concepts from the theory of programming languages, namely:

- a global state;
- imperative programs;
- monadic tranformations.

It is also inspired by research on the semantics of classical logic, including:

- Hilbert's epsilon calculus;
- Coquand's semantics of evidence;
- Aschieri's learning based realizability interpretations.

Two caveats

1. Rather than being an end in itself, it's just a first step towards more interesting applications (sketched in the last slides!)

2. None of this has been published!

Gödel's functional interpretation (a survey in seven slides)

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

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

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

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

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

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:

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

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

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

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:

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

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 is a systematic way of transforming ineffective statements into effective ones (a proof interpretation):

$$A \mapsto \forall z \exists x A^*(z, x)$$

such that

1. $A \Leftrightarrow \forall z \exists x A^*(z, x)$ over classical logic and quantifier-free choice, and

2. $A^*(z, x)$ is 'computationally neutral' (usually quantifier-free, but not always...).

In our example, $A :\equiv \exists x(D(x) \to \forall y D(y))$ and $A^*(f, x) :\equiv D(x) \to D(fx)$.

Gödel's functional interpretation is a systematic way of transforming ineffective statements into effective ones (a proof interpretation):

$$A \mapsto \forall z \exists x A^*(z, x)$$

such that

1. $A \Leftrightarrow \forall z \exists x A^*(z, x)$ over classical logic and quantifier-free choice, and

2. $A^*(z, x)$ is 'computationally neutral' (usually quantifier-free, but not always...).

In our example, $A :\equiv \exists x(D(x) \to \forall y D(y))$ and $A^*(f, x) :\equiv D(x) \to D(fx)$.

Theorem (Gödel 1958). If A is provable in Peano arithmetic, then from the proof of A we can extract a term t of system T such that $\forall z A^*(z, tz)$.

In our example, we want a term t satisfying

$$\forall f(D(tf) \to D(f(tf)))...$$

GOAL:

$$\forall f(D(tf) \to D(f(tf)))$$

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

GOAL:

$$\forall f(D(tf) \to D(f(tf)))$$

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

If $D(f0)$ is true:

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

$$\mathrm{and}(D(\underbrace{0}_{tf}) \to D(f(\underbrace{0}_{tf})) \checkmark$$

GOAL:

$$\forall f(D(tf) \to D(f(tf)))$$

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

If $D(f0)$ is true:

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

$$\mathrm{and}(D(\underbrace{0}_{tf}) \to D(f(\underbrace{0}_{tf})) \checkmark$$

If $D(f0)$ is false:

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

$$\mathrm{and}(D(\underbrace{f0}_{tf}) \to D(f(\underbrace{f0}_{tf})) \checkmark$$

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 practise, 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.

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 practise, 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.

One can think of case distinctions as being 'interactions with the mathematical environment'.

In our example, the mathematical environment consists of some predicate $D(x)$, and our program t comprises exactly one interaction with the environment, namely

$$\text{is } D(f0) \text{ true or false?}$$

A real 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.

Interpreted statement (roughly): 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 $X \subseteq \mathbb{N}$ to a monochromatic set, which is valid up to the point $\varepsilon(X)$.

A real 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.

Interpreted statement (roughly): 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 $X \subseteq \mathbb{N}$ to a monochromatic set, which is valid up to the point $\varepsilon(X)$.

From the classical proof of Ramsey's theorem, we would extract a program

$$t : (\mathcal{P}(\mathbb{N}) \to \mathbb{B}) \to \mathcal{P}(\mathbb{N})$$

which for any ε produces an approximation $t\varepsilon$ which works up to point $\varepsilon(t\varepsilon)$.

The program would typically involve a *vast* number of case distinctions of the form

$$\text{is } c(m, n) = b \text{ true or false?}$$

Two old problems with case definitions:

1. Programs extracted by the functional interpretation are only computable in settings where computationally neutral formulas are decidable. This is not the case for e.g.

- Predicate logic (Gerhardy/Kohlenbach 2005),
- Set theory (Burr 2000),
- Nonstandard analysis (van den Berg et al. 2012).

Two old problems with case definitions:

1. Programs extracted by the functional interpretation are only computable in settings where computationally neutral formulas are decidable. This is not the case for e.g.

- Predicate logic (Gerhardy/Kohlenbach 2005),
- Set theory (Burr 2000),
- Nonstandard analysis (van den Berg et al. 2012).

2. It is notoriously difficult to establish an elegant categorical semantics of the functional interpretation (de Paiva 1991), due to the fact that case distinctions are fundamentally asymmetric (this is an extreme oversimplification!).

Two old problems with case definitions:

1. Programs extracted by the functional interpretation are only computable in settings where computationally neutral formulas are decidable. This is not the case for e.g.

- Predicate logic (Gerhardy/Kohlenbach 2005),
- Set theory (Burr 2000),
- Nonstandard analysis (van den Berg et al. 2012).

2. It is notoriously difficult to establish an elegant categorical semantics of the functional interpretation (de Paiva 1991), due to the fact that case distinctions are fundamentally asymmetric (this is an extreme oversimplification!).

Both of these problems can be addressed by replacing the functional interpretation with the Diller-Nahm interpretation, which avoids interaction with the environment by collecting finite sequences of *potential* witnesses.

However, I am a big fan of Gödel's original interpretation, and propose an alternative solution:

Rather than avoiding case distinction, enrich the functional interpretation with ideas from imperative programming languages, allowing extracted programs to access a global state which is responsible for interaction with the environment.

However, I am a big fan of Gödel's original interpretation, and propose an alternative solution:

Rather than avoiding case distinction, enrich the functional interpretation with ideas from imperative programming languages, allowing extracted programs to access a global state which is responsible for interaction with the environment.

I claim that we can address both of the preceding problems, but have the following additional benefits:

- We equip the functional interpretation with a clean and elegant semantics in terms of learning;
- We improve the efficiency of extracted programs so that individual case distinctions are only ever checked once;
- We extract programs with procedural features, taking a step towards the formal extraction of verified imperative programs.

A functional interpretation with state

Traditionally, proof interpretations extract programs in some variant of Gödel's system T, which are typically conceived as purely functional programs.

In a purely functional language, the order in which we carry out computations has no effect on their output. For example, consider the program:

$$\mathsf{add}(x, 0) \to x \quad \mathsf{add}(x, sy) \to s(\mathsf{add}(x, y)).$$

Traditionally, proof interpretations extract programs in some variant of Gödel's system T, which are typically conceived as purely functional programs.

In a purely functional language, the order in which we carry out computations has no effect on their output. For example, consider the program:

$$\mathsf{add}(x, 0) \to x \quad \mathsf{add}(x, sy) \to s(\mathsf{add}(x, y)).$$

Let $\underline{n} := s^{(n)}(0)$. Then we would have

1. $\mathsf{QUERY} : \mathsf{add}(\underline{2}, \underline{5})$ $\quad$ $\mathsf{RETURN} : \underline{7}$
2. $\mathsf{QUERY} : \mathsf{add}(\underline{3}, \underline{1})$ $\quad$ $\mathsf{RETURN} : \underline{4}$

Traditionally, proof interpretations extract programs in some variant of Gödel's system T, which are typically conceived as purely functional programs.

In a purely functional language, the order in which we carry out computations has no effect on their output. For example, consider the program:

$$\mathsf{add}(x, 0) \to x \quad \mathsf{add}(x, sy) \to s(\mathsf{add}(x, y)).$$

Let $\underline{n} := s^{(n)}(0)$. Then we would have

1. $\mathsf{QUERY} : \mathsf{add}(\underline{2}, \underline{5})$ $\mathsf{RETURN} : \underline{7}$
2. $\mathsf{QUERY} : \mathsf{add}(\underline{3}, \underline{1})$ $\mathsf{RETURN} : \underline{4}$

Changing the order of computation makes no difference:

1. $\mathsf{QUERY} : \mathsf{add}(\underline{3}, \underline{1})$ $\mathsf{RETURN} : \underline{4}$
2. $\mathsf{QUERY} : \mathsf{add}(\underline{2}, \underline{5})$ $\mathsf{RETURN} : \underline{7}$

Suppose we enrich our language with a simple *global state* consisting of a single global variable i. At any moment in a computation, i is assigned some value $\underline{n}$, which we write as $[i := \underline{n}]$.

Now functions can both change and access the value stored in the global variable i. For example

$$\langle [i := \underline{n}] \mid \mathsf{add}'(x, 0) \rangle \to \langle [i := 0] \mid x \rangle$$
$$\langle [i := \underline{n}] \mid \mathsf{add}'(x, sy) \rangle \to \langle [i := \underline{n}] \mid s(\mathsf{add}'(x, y)) \rangle.$$

Suppose we enrich our language with a simple *global state* consisting of a single global variable i. At any moment in a computation, i is assigned some value $\underline{n}$, which we write as $[i := \underline{n}]$.

Now functions can both change and access the value stored in the global variable i. For example

$$\langle [i := \underline{n}] \mid \mathsf{add}'(x, 0) \rangle \to \langle [i := 0] \mid x \rangle$$
$$\langle [i := \underline{n}] \mid \mathsf{add}'(x, sy) \rangle \to \langle [i := \underline{n}] \mid s(\mathsf{add}'(x, y)) \rangle.$$

In our original pure language the function add carried out addition:

$$\mathsf{QUERY} : \mathsf{add}(\underline{a}, \underline{b}) \qquad \mathsf{RETURN} : \underline{a + b}$$

In our extended language, add' carries out the additional task of resetting the state to $[i := 0]$ i.e.

$$\mathsf{QUERY} : \langle [i := \underline{n}] \mid \mathsf{add}'(\underline{a}, \underline{b}) \rangle \qquad \mathsf{RETURN} : \langle [i := 0] \mid \underline{a + b} \rangle$$

We can also define functions which access the state:

$$\langle [i := \underline{n}] \mid \mathsf{addstate}(x) \rangle \to \langle [i := \underline{n}] \mid \mathsf{add}(x, \underline{n}) \rangle.$$

But the presence of a global state means that the order of computation matters!

We can also define functions which access the state:

$$\langle [i := \underline{n}] \mid \mathsf{addstate}(x) \rangle \to \langle [i := \underline{n}] \mid \mathsf{add}(x, \underline{n}) \rangle.$$

But the presence of a global state means that the order of computation matters! Suppose we begin by setting $[i := \underline{1}]$. Then we have

1. $\mathsf{QUERY} : \langle [i := 1] \mid \mathsf{addstate}(3) \rangle$ $\quad$ $\mathsf{RETURN} : \langle [i := 1] \mid \underline{4} \rangle$
2. $\mathsf{QUERY} : \langle [i := 1] \mid \mathsf{add}'(\underline{2}, \underline{5}) \rangle$ $\quad$ $\mathsf{RETURN} : \langle [i := 0] \mid \underline{7} \rangle$

We can also define functions which access the state:

$$\langle [i := \underline{n}] \mid \mathsf{addstate}(x) \rangle \to \langle [i := \underline{n}] \mid \mathsf{add}(x, \underline{n}) \rangle.$$

But the presence of a global state means that the order of computation matters! Suppose we begin by setting $[i := \underline{1}]$. Then we have

1. $\mathsf{QUERY} : \langle [i := 1] \mid \mathsf{addstate}(3) \rangle$ $\quad$ $\mathsf{RETURN} : \langle [i := 1] \mid \underline{4} \rangle$
2. $\mathsf{QUERY} : \langle [i := 1] \mid \mathsf{add}'(\underline{2}, \underline{5}) \rangle$ $\quad$ $\mathsf{RETURN} : \langle [i := 0] \mid \underline{7} \rangle$

but

1. $\mathsf{QUERY} : \langle [i := 1] \mid \mathsf{add}'(\underline{2}, \underline{5}) \rangle$ $\quad$ $\mathsf{RETURN} : \langle [i := 0] \mid \underline{7} \rangle$
2. $\mathsf{QUERY} : \langle [i := 0] \mid \mathsf{addstate}(3) \rangle$ $\quad$ $\mathsf{RETURN} : \langle [i := 0] \mid \underline{3} \rangle$

So what could all this have to do with the functional interpretation?

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, allow extracted programs to access and collect information in a global state, which is resposible for deciding whether a predicate should be 'accepted' or not via a function

$$\sigma : \mathcal{F} \to \{\mathsf{true}, \mathsf{false}\}$$

So what could all this have to do with the functional interpretation?

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, allow extracted programs to access and collect information in a global state, which is resposible for deciding whether a predicate should be 'accepted' or not via a function

$$\sigma : \mathcal{F} \to \{\mathsf{true}, \mathsf{false}\}$$

At a given point in time, our global state will be a finite list of 'accepted' formulas $\pi := [P_1, \dots, P_k]$ from the class $\mathcal{F}$. We can interact with the state via the function

$$\langle \pi \mid \mathsf{askstate}(P, s, t) \rangle \to \begin{cases} \langle \pi \mid s \rangle & \text{if } P \in \pi \\ \langle \pi \mid t \rangle & \text{if } \neg P \in \pi \\ \langle \pi :: P \mid s \rangle & \text{if } \sigma(P) = \mathsf{true} \\ \langle \pi :: \neg P \mid s \rangle & \text{if } \sigma(P) = \mathsf{false} \end{cases}$$

Back to our previous example:

$$\forall f(D(tf) \to D(f(tf)))$$

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

Back to our previous example:

$$\forall f(D(tf) \to D(f(tf)))$$

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

$\underline{D(f0) \in \pi}$ then $\langle \pi \mid tf \rangle \to \langle \pi \mid 0 \rangle$ and $\bigwedge \pi \to (D(0) \to D(f0))$

Back to our previous example:

$$\forall f(D(tf) \to D(f(tf)))$$

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

$\underline{D(f0) \in \pi}$ then $\langle \pi \mid tf \rangle \to \langle \pi \mid 0 \rangle$ and $\bigwedge \pi \to (D(0) \to D(f0))$

$\underline{\neg D(f0) \in \pi}$ then $\langle \pi \mid tf \rangle \to \langle \pi \mid f0 \rangle$ and $\bigwedge \pi \to (D(f0) \to D(f(f0)))$

Back to our previous example:

$$\forall f(D(tf) \to D(f(tf)))$$

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

$\underline{D(f0) \in \pi}$ then $\langle \pi \mid tf \rangle \to \langle \pi \mid 0 \rangle$ and $\bigwedge \pi \to (D(0) \to D(f0))$

$\underline{\neg D(f0) \in \pi}$ then $\langle \pi \mid tf \rangle \to \langle \pi \mid f0 \rangle$ and $\bigwedge \pi \to (D(f0) \to D(f(f0)))$

$\underline{\text{Otherwise, if } \sigma(D(f0)) = \mathsf{true}}$ then $\langle \pi \mid tf \rangle \to \langle \pi :: D(f0) \mid 0 \rangle$ and

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

Back to our previous example:

$$\forall f(D(tf) \to D(f(tf)))$$

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

$\underline{D(f0) \in \pi}$ then $\langle \pi \mid tf \rangle \to \langle \pi \mid 0 \rangle$ and $\bigwedge \pi \to (D(0) \to D(f0))$

$\underline{\neg D(f0) \in \pi}$ then $\langle \pi \mid tf \rangle \to \langle \pi \mid f0 \rangle$ and $\bigwedge \pi \to (D(f0) \to D(f(f0)))$

$\underline{\text{Otherwise, if } \sigma(D(f0)) = \mathsf{true}}$ then $\langle \pi \mid tf \rangle \to \langle \pi :: D(f0) \mid 0 \rangle$ and

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

$\underline{\sigma(D(f0)) = \mathsf{false}}$ then $\langle \pi \mid tf \rangle \to \langle \pi :: \neg D(f0) \mid f0 \rangle$ and

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

Recall that the functional interpretation is given by $A \mapsto \forall z \exists x A^*(z, x)$.

Traditional soundness

If $\mathcal{T} \vdash A$ then we can extract a pure term t such that for any z we have

$$A^*(z, tz).$$

Recall that the functional interpretation is given by $A \mapsto \forall z \exists x A^*(z, x)$.

Traditional soundness

If $\mathcal{T} \vdash A$ then we can extract a pure term t such that for any z we have

$$A^*(z, tz).$$

New soundness

If $\mathcal{T} \vdash A$ then we can extract a state sensitive term t such that for any z and input state π_I we have

$$\bigwedge \pi_O \to A^*(z, tz).$$

First advantage: We no longer require neutral formulas to be decidable, as the state function σ can be completely arbitrary!

Silly example

Let $R \in \mathcal{F}$ where R is the Riemann hypothesis. Then we can compute a realizer for

$$\exists b \in \{0,1\}(b = 0 \leftrightarrow R)$$

by setting $b := \mathsf{askstate}(R, 0, 1)$. Depending on the value of $\sigma(R)$ we end up with

$$\text{either} \quad R \to (0 = 0 \leftrightarrow R) \quad \text{or} \quad \neg R \to (1 = 0 \leftrightarrow R).$$

First advantage: We no longer require neutral formulas to be decidable, as the state function σ can be completely arbitrary!

Silly example

Let $R \in \mathcal{F}$ where R is the Riemann hypothesis. Then we can compute a realizer for

$$\exists b \in \{0,1\}(b = 0 \leftrightarrow R)$$

by setting $b := \mathsf{askstate}(R, 0, 1)$. Depending on the value of $\sigma(R)$ we end up with

$$\text{either} \quad R \to (0 = 0 \leftrightarrow R) \quad \text{or} \quad \neg R \to (1 = 0 \leftrightarrow R).$$

However, in the case the all formulas in $\mathcal{F}$ are decidable we can just set $\sigma(P) := \chi(P)$. Then for arbitrary z, setting $\pi_I := []$ we would have

$$\vdash \bigwedge \pi_0 \quad \text{and} \quad \vdash \bigwedge \pi_0 \to A^*(z, tz)$$

and therefore we reobtain $\vdash \forall z A^*(z, tz)$ i.e. the original functional interpretation.

On the other hand, we are able to compute a finite sequence of witnesses by restricting ourselves to a finite set $\mathcal{F}$ of state formulas, and letting σ range over all possible functions $\mathcal{F} \to \{\mathsf{true}, \mathsf{false}\}$: We then have

$$\bigwedge \pi_O^i \to A^*(z, tz)$$

for $i = 1, \ldots, n$ where $n = 2^{|\mathcal{F}|}$, and therefore

$$A^*(z, t_1 z) \vee \ldots \vee A^*(z, t_n z).$$

On the other hand, we are able to compute a finite sequence of witnesses by restricting ourselves to a finite set $\mathcal{F}$ of state formulas, and letting σ range over all possible functions $\mathcal{F} \to \{\mathsf{true}, \mathsf{false}\}$: We then have

$$\bigwedge \pi_O^i \to A^*(z, tz)$$

for $i = 1, \ldots, n$ where $n = 2^{|\mathcal{F}|}$, and therefore

$$A^*(z, t_1 z) \vee \ldots \vee A^*(z, t_n z).$$

In our running example, $\mathcal{F} = \{D(f0)\}$ and so we obtain

$$(D(t_1 f) \to D(f(t_1 f))) \vee (D(t_2 f) \to D(f(t_2 f)))$$

where $t_1 f = 0$ and $t_2 f = f0$.

A CATEGORICAL ASIDE. It is well known that programs which interact with a global state can be very elegantly captured using the concept of a *monad*.

The *state monad* T maps each type X in our programming language to the type

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

where S is our global state. I.e., instead of a pure term $t : X$ we have a function

$$\pi_I \mapsto (t, \pi_O).$$

The state monad is extremely convenient in allowing us to formalise everything above.

However, it may also allow for a more elegant categorical model of the functional interpretation, but this is just a conjecture!

Two further advantages:

1. In contrast to case-distinction, the global state *remembers* which formulas it has checked, and does not repeatedly query the same formula:

$$\text{We never have} \quad \langle \pi \mid t \rangle \to^* \langle \pi :: P \mid t' \rangle \to^* \langle \pi :: P :: P \mid t'' \rangle \to \ldots$$

because queries to the state always check whether or not we already have an answer. This could lead to more **efficient** extracted programs.

Two further advantages:

1. In contrast to case-distinction, the global state *remembers* which formulas it has checked, and does not repeatedly query the same formula:

$$\text{We never have} \quad \langle \pi \mid t \rangle \to^* \langle \pi :: P \mid t' \rangle \to^* \langle \pi :: P :: P \mid t'' \rangle \to \dots$$

because queries to the state always check whether or not we already have an answer. This could lead to more **efficient** extracted programs.

2. The global state makes **explicit** the elegant learning semantics which is **implicit** in the original functional interpretation. An extracted program now returns:

- A realizer (as before);
- A final state, containing everything the program has learned about the mathematical environment.

Our real example revisited: 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.

Interpreted statement (roughly): 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 $X \subseteq \mathbb{N}$ to a monochromatic set, which is valid up to the point $\varepsilon(X)$.

Our real example revisited: 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.

Interpreted statement (roughly): 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 $X \subseteq \mathbb{N}$ to a monochromatic set, which is valid up to the point $\varepsilon(X)$.

From the classical proof of Ramsey's theorem, we would extract a program

$$\pi_I, \varepsilon \mapsto \pi_O, t\varepsilon$$

which for any ε and initial state π_I produces an approximation $t\varepsilon$ which works up to point $\varepsilon(t\varepsilon)$, together with a final state π_O.

The final state would be a sequence of the form

$$\pi_O := [c(m_1, n_1) = b_1, \ldots, c(m_N, n_N) = b_N]$$

for some (probably very large) N, containing information we have learned about our colouring.

This is only a small idea!
But there are bigger goals for the future...

The notion of a state is central to imperative languages:

$$
\begin{aligned}
&i := 0 \\
&\texttt{while}(i < n) \\
&\quad \texttt{print}(i) \\
&\quad i := i + 1
\end{aligned}
$$

Can we use the functional interpretation to extract efficient, verified imperative programs?

The notion of a state is central to imperative languages:

$$
\begin{aligned}
&i := 0 \\
&\texttt{while}(i < n) \\
&\quad \texttt{print}(i) \\
&\quad i := i + 1
\end{aligned}
$$

Can we use the functional interpretation to extract efficient, verified imperative programs?

Simple example: $\mathcal{T} \vdash \forall n \exists p(p > n \wedge \text{prime}(p))$

Extract a verified program in some imperative language which takes as input a natural number n and returns a prime number p greater than n.

There is some precedent here e.g.

- *Extracting Imperative Programs from Proofs: In-place Quicksort.* Berger, Seisenberger, Woods, 2014.

Programs extracted using functional interpretations are usually verified to be correct in some variant of Heyting arithmetic.

But there are other logics designed to reason about programs with state e.g. *Hoare logic*.

Can we adapt the functional interpretation so that it not only returns imperative programs, but verifies them using Hoare logic?

A truly imperative functional interpretation

If $\mathcal{T} \vdash A$ then we can extract a program P such that for any z and input state π_I we have

$$\{\pi_I, [x := 0]\}\ P\ \{\pi_O, [x := \underline{n}], A^*(z, n)\}.$$

Could obtain new consistency proofs in terms of Hoare logic...

Thank you