Certifying higher-order polynomial interpretations

Cynthia Kop, Deivid Vale, and **Niels van der Weide**

This talk

Topic of this talk: our paper called

Certifying higher-order polynomial interpretations

What am I going to tell you?

- Motivation and context
- Brief overview of the results
- Some challenges in the formalization

First things first, do we know higher-order rewriting?

First things first, do we know higher-order rewriting?

If not, then here is the introduction.
Higher-order rewriting is about:

a style of simply-typed λ-calculae extended with a set of type-annotated symbols.

First things first, do we know higher-order rewriting?

If not, then here is the introduction.
Higher-order rewriting is about:

> a style of simply-typed λ-calculae extended with a set of type-annotated symbols.

So, we look at systems like this

$$\mathcal{R} := \begin{cases} \mathsf{map}\ F\ \mathsf{nil} \to \mathsf{nil} \\ \mathsf{map}\ F\ (x :: xs) \to (F\ x) :: \mathsf{map}\ F\ xs \end{cases}$$

But what do we want?

We want to **reason** about our systems.

- **Termination**: do our systems run forever?
- **Confluence**: do our systems give a unique outcome?
- **Complexity**: how fast do our systems run?

And what do we not want?

We **don't** want to reason about our systems **ourselves**.

- Systems can be quite large and have many rules
- Manual execution is tedious and error-prone

And what do we not want?

We **don't** want to reason about our systems **ourselves**.

- Systems can be quite large and have many rules
- Manual execution is tedious and error-prone

Enter the stage: **termination checkers**

AProVE, $\mathsf{T_TT_2}$, NaTT, SOL, Wanda, ...

However...

Developing termination checkers is difficult

- There could be **mistakes** in the proof of our theorems
- There could be **bugs** in the implementation

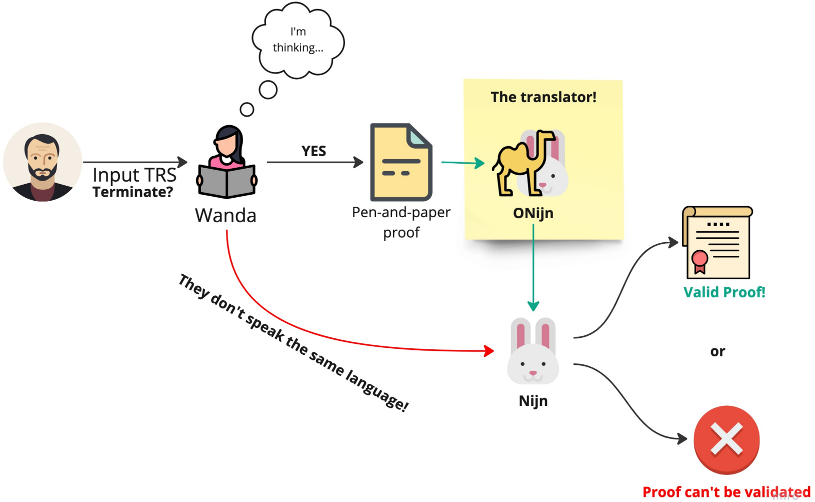
How can we provide guarantee that the output of our termination checkers is correct?

So... what did we do?

We introduce Nijn/ONijn. It includes:

- the **formalization** engine
  - a formalization in Coq of the theory of **higher-order** rewriting
  - a formalization of higher-order polynomial interpretation
- the **translation** engine
  - an OCaml program that turns the output of termination checkers (like Wanda) into a Coq script.
- If that Coq script type checks, then the output was correct.

Overview of our work



ONijn's Output

coq_certificates > Mixed_HO_10_map.v

```
Require Import Nijn.Nijn.
Open Scope poly_scope.

Inductive base_types :=
| Ca
| Clist.
Global Instance decEq_base_types : decEq base_types.
Proof.
decEq_finite.
Defined.

Definition a :=
Base Ca.
Definition list :=
Base Clist.
```

ONijn's Output

```
Inductive fun_symbols :=
| Tcons
| Tmap
| Tnil.
Global Instance decEq_fun_symbols : decEq fun_symbols.
Proof.
decEq_finite.
Defined.
```

ONijn's Output

```
Definition fn_arity fn_symbols :=
match fn_symbols with
  | Tcons  ⇒  a → list → list
  | Tmap  ⇒  list → (a → a) → list
  | Tnil ⇒ list
end.
Definition cons {C} : tm fn_arity C _ :=
BaseTm Tcons.
Definition map {C} : tm fn_arity C _ :=
BaseTm Tmap.
Definition nil {C} : tm fn_arity C _ :=
BaseTm Tnil.
```

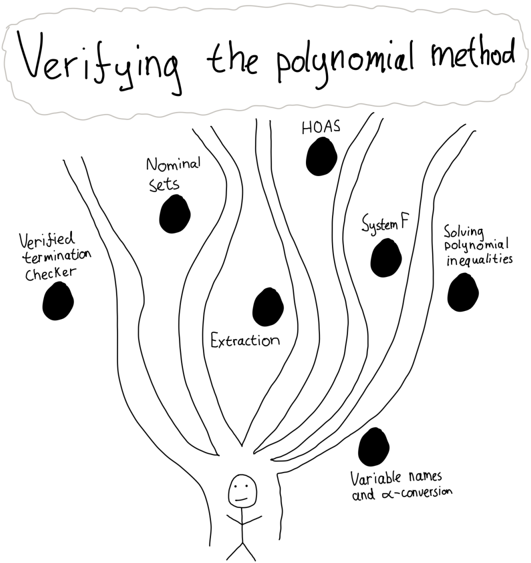
ONijn's Output

```
Program Definition rule_0 :=
    make_rewrite
    (_ ,, ·) _
    (map · nil · V 0)
    nil.
Program Definition rule_1 :=
    make_rewrite
    (_ ,, _ ,, _ ,, ·) _
    (map · (cons · V 0 · V 1) · V 2)
    (cons · ( V 2 · V 0) · (map · V 1 · V 2)).

Definition trs :=
  make_afs
    fn_arity
    (rule_0 :: rule_1 :: List.nil).

```

ONijn's Output

```

Definition map_fun_poly fn_symbols : poly · (arity trs fn_symbols) :=
match fn_symbols with
| Tcons ⇒
 λP
λP let y1 := P_var Vz in
(to_Poly (P_const 3 + P_const 2 * y1))
| Tmap ⇒
 λP let y0 := P_var (Vs Vz) in
λP let G1 := P_var Vz in
(to_Poly (P_const 3 * y0 + P_const 3 * y0 * (G1 ·P (y0))))
| Tnil ⇒
(to_Poly (P_const 3))
end.
Definition trs_isSN : isSN trs.
Proof.
solve_poly_SN map_fun_poly.
Qed.
```

Back to Formalization



So... what did we formalize?

More specifically, we formalized

- Higher-order rewriting systems
- Basic **constructive** theory of strong normalization
- The polynomial method

We also formalized **rule removal**, but that is not in the paper.

Main Challenges

To formalize the theory, we faced the following challenges:

- **Variables** (*names are handy, but difficult*)
- **Polynomials** (*actually, the precise definition of polynomials is quite interesting*)

Formalizing Variables: Challenges

Challenge: **variable names**

- Variables are identified up to α-**equivalence**
- **Variable capture** could occur
- We need to find fres variables in order to do **renaming**

Each of these aspects adds complication to the formalization, and that makes using variable names challenging.
So... what do we do?

Variable names

Variable names (according to N.G. de Bruijn)

My name is 1

De Bruijn Indices

Main ideas:

- Represent variables by **number**
- This number tells **to which** λ the variable refers to

De Bruijn Indices

Usually, we would write

$$\lambda x\, \lambda y\, \lambda f.\ f\ x\ (f\ y\ y)$$

De Bruijn Indices

Usually, we would write

$$\lambda x\, \lambda y\, \lambda f.\; f\; x\; (f\; y\; y)$$

N.G. de Bruijn would write

$$\lambda\, \lambda\, \lambda.\; 0\; 2\; (0\; 1\; 1)$$

De Bruijn Indices

We would also write

$$\lambda f.\ f\ (\lambda x.\ f\ x)$$

De Bruijn Indices

We would also write

$$\lambda f.\ f\ (\lambda x.\ f\ x)$$

whereas N.G. de Bruijn would write

$$\lambda\ .\ 0\ (\lambda\ .\ 1\ 0)$$

Advantages of De Bruijn Indices

- There's no variable capture
- There's no need to rename variable names
- α-equality coincides with syntactic equality

This **simplifies** implementing variables in a proof assistant.

Disadvantages of De Bruijn Indices

They are not readable.

Disadvantages of De Bruijn Indices

They are not readable. However,

- In our work, they are used in **internal representation**
- In the input, one still writes terms with variable names, and those are **converted** to the internal representation

So that's not my problem, but the proof assistant's problem

Polynomials

Fuhs and Kop define polynomials as follows

▶ **Definition 4.1** (Higher-Order Polynomial over $\mathbb{N}$). For a set $X = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ of variables, each equipped with a type, the set $Pol(X)$ of *higher-order polynomials* in X is given by the following clauses:

- if $n \in \mathbb{N}$, then $n \in Pol(X)$;
- if $p_1, p_2 \in Pol(X)$, then $p_1 + p_2 \in Pol(X)$ and $p_1 \cdot p_2 \in Pol(X)$;
- if $x_i : \tau_1 \Rightarrow \ldots \Rightarrow \tau_m \Rightarrow \iota \in X$ with $\iota \in \mathcal{B}$, and $p_1 \in Pol^{\tau_1}(X), \ldots, p_m \in Pol^{\tau_m}(X)$, then $x_i(p_1, \ldots, p_m) \in Pol(X)$;
  - here, $Pol^{\iota}(X) = Pol(X)$ for base types ι, and $Pol^{\sigma \Rightarrow \tau}(X)$ contains functions $\boldsymbol{\lambda} y.p \in \mathcal{WM}_\sigma$ with $p \in Pol^{\tau}(X \cup \{y\})$.

Polynomials

Fuhs and Kop define polynomials as follows

▶ **Definition 4.1** (Higher-Order Polynomial over $\mathbb{N}$). For a set $X = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ of variables, each equipped with a type, the set $Pol(X)$ of *higher-order polynomials* in X is given by the following clauses:

- if $n \in \mathbb{N}$, then $n \in Pol(X)$;
- if $p_1, p_2 \in Pol(X)$, then $p_1 + p_2 \in Pol(X)$ and $p_1 \cdot p_2 \in Pol(X)$;
- if $x_i : \tau_1 \Rightarrow \ldots \Rightarrow \tau_m \Rightarrow \iota \in X$ with $\iota \in \mathcal{B}$, and $p_1 \in Pol^{\tau_1}(X), \ldots, p_m \in Pol^{\tau_m}(X)$, then $x_i(p_1, \ldots, p_m) \in Pol(X)$;
  - here, $Pol^{\iota}(X) = Pol(X)$ for base types ι, and $Pol^{\sigma \Rightarrow \tau}(X)$ contains functions $\boldsymbol{\lambda} y.p \in \mathcal{WM}_{\sigma}$ with $p \in Pol^{\tau}(X \cup \{y\})$.

Here a lot is going on from a logical perspective.

What's going on???

- We define a set Pol^{τ} for every type τ
- We identify all Pol^{b} for base types b
- Note: we want to identify all Pol^{b}, because the base type doesn't matter

Coq finds the combination of these two steps complicated.

What did we tell Coq?

Our implementation in Coq:

```
Inductive base_poly {B : Type} : con B -> Type :=
| P_const : forall {C : con B}, nat -> base_poly C
| P_plus : forall {C : con B}, base_poly C -> base_poly C -> base_poly C
| P_mult : forall {C : con B}, base_poly C -> base_poly C -> base_poly C
| from_poly : forall {C : con B} {b : B},
                  poly C (Base b)
                  -> base_poly C
with poly {B : Type} : con B -> ty B -> Type :=
| P_base : forall {C : con B} {b : B}, base_poly C -> poly C (Base b)
| P_var : forall {C : con B} {A : ty B},
              var C A
              -> poly C A
| P_app : forall {C : con B} {A₁ A₂ : ty B},
              poly C (A₁ ⟶ A₂)
              -> poly C A₁
              -> poly C A₂
| P_lam : forall {C : con B} {A₁ A₂ : ty B},
              poly (A₁ ,, C) A₂
              -> poly C (A₁ ⟶ A₂).
```

Whaaat?

We discuss the implementation details in the paper.
Main idea:

- We define a type Pol^{τ} for every τ
- We define the type of base polynomials, and this type does **not** on the base type
- We can coerce between the two different types

But wait! There's more...

We also did:

- develop nice notations, so that the script is kinda readable
- formalize rule removal
- formalize nontermination
- automatically **solve polynomial inequalities**

All of this is not part of this talk.

Summary and Conclusion

- We made a formalization with the basic results of higher order rewriting.

Summary and Conclusion

- We made a formalization with the basic results of higher order rewriting.
- We also formalized the polynomial method.

Summary and Conclusion

- We made a formalization with the basic results of higher order rewriting.
- We also formalized the polynomial method.
- We made an OCaml program that turned the output of a termination checker into a Coq script.

Summary and Conclusion

- We made a formalization with the basic results of higher order rewriting.
- We also formalized the polynomial method.
- We made an OCaml program that turned the output of a termination checker into a Coq script.
- The certification method is effective: we could verify the output of Wanda on a set of 46 problems.