Certifying Higher-Order Polynomial Interpretations

Niels van der Weide
Institute for Computing and Information Sciences, Radboud University, Nijmegen, The Netherlands

Deivid Vale
Institute for Computing and Information Sciences, Radboud University, Nijmegen, The Netherlands

Cynthia Kop
Institute for Computing and Information Sciences, Radboud University, Nijmegen, The Netherlands

Abstract

Higher-order rewriting is a framework in which one can write higher-order programs and study their properties. One such property is termination: the situation that for all inputs, the program eventually halts its execution and produces an output. Several tools have been developed to check whether higher-order rewriting systems are terminating. However, developing such tools is difficult and can be error-prone. In this paper, we present a way of certifying termination proofs of higher-order term rewriting systems. We formalize a specific method that is used to prove termination, namely the polynomial interpretation method. In addition, we give a program that processes proof traces containing a high-level description of a termination proof into a formal Coq proof script that can be checked by Coq. We demonstrate the usability of this approach by certifying higher-order polynomial interpretation proofs produced by Wanda, a termination analysis tool for higher-order rewriting.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Equational logic and rewriting

Keywords and phrases higher-order rewriting, Coq, termination, formalization

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23


Supplementary Material Coq Formalization (GitHub repository): nmvdw/Nijn [31]
Proof script generator (GitHub repository): deividvale/nijn-coq-script-generation [30]

Funding Niels van der Weide: This research was supported by the NWO project “The Power of Equality” OCENW.M20.380, which is financed by the Dutch Research Council (NWO).
Deivid Vale: Author supported by NWO Top project “Implicit Complexity through Higher-Order Rewriting”, NWO 612.001.803/7571.
Cynthia Kop: Author supported by NWO Top project “Implicit Complexity through Higher-Order Rewriting”, NWO 612.001.803/7571 and the NWO VIDI project “Constrained Higher-Order Rewriting and Program Equivalence”, NWO VI.Vidi.193.075.

Acknowledgements The authors thank Dan Frumin for his help with understanding and using Ltac.

1 Introduction

Automatically proving termination is an important problem in term rewriting, and numerous tools have been developed for this purpose, such as AProVE [10], NaTT [35], MatchBox [33], Mu-Term [12], SOL [13], Tff2 [21] and Wanda [16], which compete against each other in an annual termination competition [11]. Aside from basic (first-order) term rewriting, this includes tools analyzing for instance string, conditional, and higher-order rewriting.

Developing termination tools is a difficult and error-prone endeavor. On the one hand, the termination techniques that are implemented may contain errors. This is particularly relevant in higher-order term rewriting, where the proofs are often very intricate due to partial application, type structure, beta-reduction, and techniques often not transferring
perfectly between different formalisms of higher-order rewriting. Hence, it should come as no surprise that errors have been found even in published papers on higher-order rewriting. On the other hand, it is very easy for a tool developer to accidentally omit a test whether some conditions to apply specific termination techniques are satisfied, or to incorrectly translate a method between higher-order formalisms.

To exacerbate this issue, termination proofs are usually complex and technical in nature, which makes it hard to assess the correctness of a prover’s output by hand. Not only do many benchmarks contain hundreds of rules, modern termination tools make use of various proof methods that have been developed for decades. Indeed, a single termination proof might, for instance, make use of a combination of dependency pairs [3, 9, 19], recursive path orders [5, 20], rule removal, and multiple kinds of interpretations [8, 18, 23, 34]. This makes bugs very difficult to find.

Hence, there is a need to formally certify the output of termination provers, ideally automatically. There are two common engineering strategies to provide such certification. In the first, one builds the certifier as a library in a proof assistant along with tools to read the prover’s output and construct a formal proof, which we call proof script. The proof script is then verified by a proof assistant. Examples of this system design are the combinations Cochinelle/CiME3 [7] and CoLoR/Rainbow [6]. In the second, the formalization includes certified algorithms for checking the correctness of the prover’s output. This allows for the whole certifier to be extracted, using code extraction, and be used as a standalone program. Hence, the generation of proof scripts by a standalone tool is not needed in this approach, but it comes with a higher formalization cost. IsaFoR/CeTA [28] utilizes this approach.

When it comes to higher-order rewriting, however, the options are limited. Both Cochinelle [7] and IsaFoR/CeTA [28] only consider first-order rewriting. CoLoR/Rainbow [6] does include a formalization of an early definition of HORPO [20]. Since here we use a different term formalism compared to that of [20], our results are not directly compatible. See for instance [2, 25] for more formalization results in rewriting.

In this paper, we introduce a new combination Nijn/ONijn for the certification of higher-order rewriting termination proofs. We follow the first aforementioned system design: Nijn is a Coq library providing a formalization of the underlying higher-order rewriting theory and ONijn is a proof script generator that given a minimal description of a termination proof (which we call proof trace), outputs a Coq proof script. The proof script then utilizes results from Nijn for checking the correctness of the traced proof. The schematic below depicts the basic steps to produce proof certificates using Nijn/ONijn.

![Figure 1 Nijn/ONijn schematics](image)

While Nijn is the certified core part of our tool since it is checked by Coq, the proof script
generation implemented in OCaml (ONijn) is not currently certified and must be trusted. For this reason, we deliberately keep ONijn as simple as possible and no checking or computation is done by it. The only task delegated to ONijn is that of parsing the proof trace given by the termination prover to a Coq proof script. Additionally, checking the correctness of polynomial termination proofs in Coq is an inherently incomplete task, since it would require a method to solve inequalities over arbitrary polynomials, which is undecidable in general.

Contributions The main contribution of this paper can be summarized as follows:
- we provide a formalization of higher-order algebraic functional systems (Definition 2.6);
- a formal proof of the interpretation method using weakly monotonic algebras (Theorem 3.11);
- a formalization of the higher-order polynomial method (Theorem 4.7);
- a tactic that automatically solves the constraints that arise when using the higher-order polynomial method (Section 4.3);
- an OCaml program that transforms the output of a termination prover into a Coq script that represents the termination proof (Section 5).

Technical Overview. This paper orbits Nijn, a Coq library formalizing higher-order rewriting [31]. The formalization is based on intensional dependent type theory extended with two axioms: function extensionality and uniqueness of identity proofs [14]. Currently, the termination criterion formalized in the library is the higher-order polynomial method, introduced in [8]. The tool coqwc counts the following amount of lines of code:

```
5497 1985 272 total
```

The higher-order interpretation method roughly works as follows. First, types are interpreted as well-ordered structures (Definition 3.3), compositionally. For instance, we interpret base types as natural numbers (with the usual ordering). Then we interpret a functional type \( A \Rightarrow B \) as the set of weakly monotonic functions from \( \langle A \rangle \) to \( \langle B \rangle \) where \( \langle A \rangle \), \( \langle B \rangle \) denote the interpretations of \( A, B \) respectively. The second step is to map inhabitants of a type \( A \) to elements of \( \langle A \rangle \), which is expressed here by Definition 3.9.

This interpretation, called extended monotonic algebras in [8], alone does not suffice for termination. To guarantee termination, we interpret both term application (Definition 4.6) and function symbols as strongly monotonic functionals. In addition, terms must be interpreted in such a way that the rules of the system are strictly oriented, i.e., \( [\ell] > [r] \), for all rules \( \ell \rightarrow r \). This means that whenever a rewriting is fired in a term, the interpretation of that term strictly decreases. As such, termination is guaranteed. Here we use termination models (Definition 3.10) to collect these necessary conditions.

The main result establishing the correctness of this technique in the higher-order case is expressed by Theorem 3.11. To the reader familiar with the interpretation method in first-order rewriting, Theorem 4.7 would be no surprise. It is essentially the combination of the Manna–Ness criterion with higher-order polynomials and the additional technicalities that are needed for the higher-order case.

### 2 The Basics of Higher-Order Rewriting in Coq

In this section, we introduce the basic constructs needed to formalize algebraic functional systems (AFSs) like types, contexts, variables, terms, and rewriting rules. We end the section with an exposition on how to express termination constructively in Coq.
2.1 Terms and Rewrite Rules

Let us start by defining simple types.

Definition 2.1 (ty). Simple types over a type B are defined as follows:

\[
\text{Inductive } \text{ty} (B : \text{Type}) : \text{Type} := \\
\mid \text{Base} : B \rightarrow \text{ty} B \\
\mid \text{Fun} : \text{ty} B \rightarrow \text{ty} B \rightarrow \text{ty} B.
\]

Elements of B are called base types. Every inhabitant b : B gives rise to a simple type Base b and if A1, A2 are simple types then so is Fun A1 A2. We write A1 → A2 for Fun A1 A2.

We need (variable) contexts in order to type terms that may contain free variables. Conceptually, a context is a list of variables with their respective types. For instance, we have two variables. The first one, which is \(x\), represents the object-level type of the expression, and context requires a type of all free variables in the term. We also need to type function symbols. Hence, we use nameless variables in our development, so we do not keep track of their names.

Definition 2.2 (con). The type of variable contexts over a type B is defined as follows.

\[
\text{Inductive } \text{con} (B : \text{Type}) : \text{Type} := \\
\mid \text{Empty} : \text{con} B \\
\mid \text{Extend} : \text{ty} B \rightarrow \text{con} B \rightarrow \text{con} B.
\]

We write • for Empty and A , C for Extend A C.

Definition 2.3 (var). We define the type var C A of variables of type A in context C as

\[
\text{Inductive } \text{var} \{B : \text{Type}\} : \text{con} B \rightarrow \text{ty} B \rightarrow \text{Type} := \\
\mid \text{Vz} : \forall \{C : \text{con} B\} \{A : \text{ty} B\}, \text{var} (A , C) A \\
\mid \text{Vs} : \forall \{C : \text{con} B\} \{A1 A2 : \text{ty} B\}, \text{var} C A2 \rightarrow \text{var} (A1 , C) A2.
\]

Let us consider an example of a context and some variables. Suppose that we have a context with variables \(x_0, x_1, \ldots, x_n\). Then we only consider the list \([A_0, \ldots, A_n]\). However, we still need to refer to the free variables in terms. In order to do so, we represent them through indexing positions in the context. For instance, in the context 

\[
[A_0; \ldots; A_n]
\]

we have \(n + 1\) position indexes 0, 1, \ldots, \(n\), which we use as variables.

Definition 2.4 (tm). We define the type of well-typed terms as follows.

\[
\text{Inductive } \text{tm} \{B : \text{Type}\} \{F : \text{Type}\} \{\text{ar} : F \rightarrow \text{ty} B\} \{C : \text{con} B\} : \text{ty} B \rightarrow \text{Type} := \\
\mid \text{BaseVar} : \forall \{A : \text{ty} B\}, \text{var} C A \rightarrow \text{tm} ar C A \\
\mid \text{Lam} : \forall \{A1 A2 : \text{ty} B\}, \text{tm} ar (A1 , C) A2 \rightarrow \text{tm} ar C (A1 \rightarrow A2) \\
\mid \text{App} : \forall \{A1 A2 : \text{ty} B\}, \text{tm} ar C (A1 \rightarrow A2) \rightarrow \text{tm} ar C A1 \rightarrow \text{tm} ar C A2.
\]
For every function \( f : F \) we have a term \( \text{BaseTm} f \) of type \( \text{ar} f \). Every variable \( v \) gives rise to a term \( \text{TmVar} v \). For \( \lambda \)-abstractions, given a term \( s : \text{tm ar} (A1 \rightarrow C) A2 \), there is a term \( \lambda s : \text{tm ar} (A1 \rightarrow C) A2 \), namely \( \text{Lam} s \). The last constructor represents term application. If we have a term \( s : \text{tm ar} (A1 \rightarrow C) \) and a term \( t : \text{tm ar} C A1 \), we get a term \( \text{App} s t \).

While it may be more cumbersome to write down terms using de Bruijn indices, it does have several advantages. Most importantly, it eliminates the need for \( \alpha \)-equivalence, so that determining equality between terms is reduced to a simple syntactic check.

Our notion of rewriting rules deviates slightly from the presentation in [8]. Mainly, we do not impose the pattern restriction on the left-hand side of rules nor that free variables on the right-hand side occur on the left-hand side. This choice simplifies the formalization effort because when defining a concrete TRS, one does not need to check this particular condition. Note that in IsaFoR [28] the same simplification is used.

\[\text{Definition 2.5 (rewriteRule).} \] The type of rewrite rules is defined as follows:

\[
\text{Record rewriteRule \{ B : Type \} \{ F : Type \} (ar : F \rightarrow \text{ty} B) :=}
\begin{align*}
\text{make_rewrite} \\
\{ \\
\text{vars_of} : \text{con} B \\
\text{tar_of} : \text{ty} B \\
\text{lhs_of} : \text{tm ar} \text{vars_of} \text{tar_of} \\
\text{rhs_of} : \text{tm ar} \text{vars_of} \text{tar_of} \}.
\end{align*}
\]

The context \text{vars_of} carries the variables used in the rule, and the type \text{tar_of} is used to guarantee that both the \text{lhs_of} and \text{rhs_of} are terms of the same type.

\[\text{Definition 2.6 (afs).} \] The type of algebraic functional systems is defined as follows

\[
\text{Record afs \{ B : Type \} \{ F : Type \} :=}
\begin{align*}
\text{make_afs} \{ \\
\text{arity} : F \rightarrow \text{ty} B \\
\text{list_of_rewriteRules} : \text{list} (\text{rewriteRule} \text{arity}) \}.
\end{align*}
\]

As usual, every AFS induces a rewrite relation on the set of terms, which we denote by \( \sim \). The formal definition is found in RewritingSystem.v. The rewrite relation \( \sim \) is defined to be the closure of the one-step relation under transitivity and compatibility with the term constructors. In Coq, we use an inductive type to define this relation. Each rewrite step is represented by a constructor. More specifically, we have a constructor for rewriting the left-hand and the right-hand side of an application, we have a constructor for \( \beta \)-reduction, and we have a constructor for the rewrite rules of the AFS.

\[\text{Example 2.7 (map.afs).} \] Let us encode \( R_{\text{map}} \) in Coq. It is composed of two rules: \( \text{map} F \text{nil} \rightarrow \text{nil} \) and \( \text{map} F (\text{cons} x \text{xs}) \rightarrow \text{cons} (F x) (\text{map} F \text{xs}) \). We start with base types.

\[
\text{Inductive base_types := Ttype | TList.} \\
\text{Definition Btype : ty base_types := Base Ttype.} \\
\text{Definition List : ty base_types := Base TList.}
\]

The abbreviations \text{Btype} and \text{List} is to smoothen the usage of the base types. There are three function symbols in this system:

\[
\text{Inductive fun_symbols := TNil | TCons | TMap.}
\]

The arity function \text{map_ar} maps each function symbol in \text{fun_symbols} to its type.
Definition map_ar f : ty base_types := match f with | TNil  ⇒ List | TCons ⇒ Btype → List → List | TMap ⇒ (Btype → Btype) → List → List end.

So, TNil is a list and given an inhabitant of Btype and List, the function symbol TCons gives a List. Again we introduce some abbreviations to simplify the usage of the function symbols.

Definition Nil {C} := BaseTm TNil.
Definition Cons {C} x xs : tm map_ar C _ := BaseTm TCons · x · xs.
Definition Map {C} f xs : tm map_ar C _ := BaseTm TMap · f · xs.

The first rule, map F nil → nil, is encoded as the following Coq construct:

Program Definition map_nil :=
  make_rewrite
  (_ , , •)
  (let f := TmVar Vz in Map · f · Nil)
  Nil.

Notice that we only defined the pattern of the first two arguments of make_rewrite, leaving the types in the context (_ , , •) and the type of the rule unspecified. Coq can fill in these holes automatically, as long as we provide a context pattern of the correct length. In this particular rewrite rule, there is only one free variable. As such, the variable TmVar Vz refers to the only variable in the context. In addition, we use iterated let-statements to imitate variable names. For every position in the context, we introduce a variable in Coq, which we use in the left- and right-hand sides of the rule. This makes the rules more human-readable.

Indeed, the lhs map F nil of this rule is represented as Map · f · Nil in code. The second rule for map is encoded following the same ideas.

Program Definition map_cons :=
  make_rewrite
  (_ , , _ , , _ , , •)
  (let f := TmVar Vz in let x := TmVar (Vs Vz) in let xs := TmVar (Vs (Vs Vz)) in Map · f · (Cons · x · xs))
  (let f := TmVar Vz in let x := TmVar (Vs Vz) in let xs := TmVar (Vs (Vs Vz)) in Cons · (f · x) · (Map · f · xs)).

Putting this all together, we obtain an AFS, which we call map_afs.

Definition map_afs := make_afs map_ar (map_nil :: map_cons :: nil).

2.2 Termination

Strong normalization is usually defined as the absence of infinite rewrite sequences. Such a definition is sufficient in a classical setting where the law of excluded middle holds. However, we work in a constructive setting, and thus we are interested in a stronger definition. Therefore, we need a constructive predicate, formulated positively, which implies there are no infinite rewrite sequences. This idea is captured by the following definition

Definition 2.8 (WellfoundedRelation.v). The well-foundedness predicate for a relation is defined as follows

薏 Definition 2.8 (WellfoundedRelation.v). The well-foundedness predicate for a relation is defined as follows
Inductive isWf { X : Type } ( R : X → X → Type ) ( x : X ) : Prop :=
| acc : ( forall ( y : X ), R x y → isWf R y ) → isWf R x.

A relation is well-founded if the well-foundedness predicate holds for every element.

Definition Wf { X : Type } ( R : X → X → Type ) :=
forall ( x : X ), isWf R x.

Note that this definition has been considered numerous times before, for example in [4] and in CoLoR [6]. An element x is well-founded if all y such that R x y are well-founded. Note that if there is no y such that R x y, then x is vacuously well-founded. From the rewriting perspective, this definition properly captures the notion of strong normalization. Indeed, a term s is strongly normalizing iff every s’ such that s rewrites to s’ is strongly normalizing.

Well-foundedness contradicts the existence of infinite rewrite sequences, even in a constructive setting. As such, it indeed gives a stronger condition.

 Proposition 2.9 (no_infinite_chain). If R is well-founded, then there is no infinite sequence s₀, s₁, ... such that R(sₙ, sₙ₊₁), for all n.

Next, we define strong normalization using well-founded predicates.

 Definition 2.10 (is SN). An algebraic functional system is strongly normalizing if for every context C and every type A the rewrite relation for terms of type A in context C is well-founded. We formalize that as follows:

Definition isSN { B F : Type } { X : afs B F } : Prop :=
forall ( C : con B ) ( A : ty B ), Wf ( fun ( t₁ t₂ : tm X C A ) ⇒ t₁ ~> t₂ ).

3 Higher-Order Interpretation Method

In this section, we formalize the method of weakly monotonic algebras for algebraic functional systems. We proceed by providing type-theoretic semantics for the syntactic constructions introduced in the last section and a sufficient condition for which such semantics can be used to establish strong normalization.

3.1 Interpreting types and terms

In weakly monotonic algebras, types are interpreted as sets along with a well-founded ordering and a quasi-ordering [8, 24]. For that reason, we start by defining compatible relations. Intuitively, these are the domain for our semantics.

 Definition 3.1 (CompatibleRelation.v). Compatible relations are defined as follows

Record CompatRel := {
  carrier :> Type ;
  gt : carrier → carrier → Prop ;
  ge : carrier → carrier → Prop }. 

We write x > y and x >= y for gt x y and ge x y respectively.

The record CompatRel consists of the data needed to express compatibility between > and >=. The conditions it needs to satisfy, are in the type class isCompatRel, defined below.
Certifying Higher-Order Polynomial Interpretations

Class `isCompatRel (X : CompatRel) := {
  gt_trans : forall (x y z : X), x > y → y > z → x > z ;
  ge_trans : forall (x y z : X), x >= y → y >= z → x >= z ;
  ge_refl : forall (x : X), x >= x ;
  compat : forall (x y : X), x > y → x >= y ;
  ge_gt : forall (x y z : X), x >= y → y > z → x > z ;
  gt_ge : forall (x y z : X), x > y → y >= z → x > z ;
}.

Note that the field `gt_trans` in `isCompatRel` follows from `compat` and `ge_gt`. The type `nat` of natural numbers with the usual orders is a first example of data that satisfies `isCompatRel`. We denote this one by `nat_CompatRel`. This type class essentially models the notion of extended well-founded set introduced in [18]. An extended well-founded set is a set together with compatible orders `>`, `≥` such that `>` is well-founded and `≥` is a quasi-ordering. This compatibility requirement corresponds to the axiom `compat` in the type class `isCompatRel`. However, since we do not require `>` to be well-founded in this definition, we instead call it a compatible relation. More specifically, `X` is a compatible relation if it is of type `CompatRel` and satisfies the constraints in the type class `isCompatRel`.

In order to interpret simple types (Definition 2.1), we start by fixing a type `B : Type` of base types and an interpretation `semB : B → CompatRel` such that each `semB b` is a compatible relation. Whenever `semB` satisfies such property we call it an interpretation key for `B`. We interpret arrow types as functional compatible relations, i.e., compatible relations such that the inhabitants of their carrier are functional. The class of functionals we are interested in is that of weakly-monotone maps.

▶ Definition 3.2 (MonotonicMaps.v). Weakly monotone maps are defined as follows

Class `weakMonotone {X Y : CompatRel} (f : X → Y) :=
  map_ge : forall (x y : X), x >= y → f x >= f y .

Record `weakMonotoneMap (X Y : CompatRel) :=
  make_monotone {
    fun_carrier :> X → Y ;
    is_weak_monotone : weakMonotone fun_carrier }.

The class `weakMonotone` says when a function is weakly monotonic, and an inhabitant of the record `weakMonotoneMap` consists of a function together with proof of its weak monotonicity. Then we define `fun_CompatRel` which is of type `CompatRel` and represents the functional compatible relations from `X` to `Y`. It is defined as follows:

Definition `fun_CompatRel (X Y : CompatRel) : CompatRel :={
  carrier := weakMonotoneMap X Y ;
  gt f g := forall (x : X), f x > g x ;
  ge f g := forall (x : X), f x >= g x ;
}.

In what follows, we write `X →_wm Y` for `fun_CompatRel X Y`. The semantics for a type is parametrized by an interpretation key `semB`. It is defined as follows:

▶ Definition 3.3 (sem_Ty). Assume `A : ty B` and `semB` is an interpretation key for `B`. Then

Fixpoint `sem_Ty (A : ty B) : CompatRel :=
  match A with
    | Base b ⇒ semB b
    | A1 → A2 ⇒ `sem_Ty A1 →_wm `sem_Ty A2
  end.
We also show how to interpret contexts, and to do so, we need to interpret the empty context and context extension. For those, we define the unit and product of compatible relations.

Definition 3.4 (Examples.v). The unit and product compatible relations:

```
Definition unit_CompatRel : CompatRel :=
| | ⇒ unit_CompatRel
| A , C ⇒ sem_Ty A ∗ sem_Con C
end.
```

Next, we give semantics to variables and terms. The approach we use here is slightly different from what is usually done in higher-order rewriting. In [8, 18, 24], for instance, context information is lifted to the meta-level and variables are interpreted using the notion of valuations. In contrast, in our setting, the typing context lives at the syntactic level and context extension. For those, we define the unit and product of compatible relations. Consequently, to every term `t : tm C A`, we assign a map from `sem_Con C` to `sem_Ty A`. In the remainder, we need the following weakly monotonic functions.

Definition 3.5 (sem_Con). Contexts are interpreted as follows

```
Fixpoint sem_Con (C : con B) : CompatRel :=
  match C with
  | • ⇒ unit_CompatRel
  | A , C ⇒ sem_Ty A ∗ sem_Con C
end.
```

Recall that variables are represented by positions in a context which in turn is interpreted as a weakly monotonic product (Definition 3.5). This allows us to interpret the variable at a position in a context as the corresponding interpretation of the type in that position.

Definition 3.6 (Examples.v). We define the following weakly monotonic functions.

- Given `y : Y`, we write `const_wm y : X → wm Y` for the constant function.
- Given `f : X → wm Y` and `g : Y → wm Z`, we define the composition `g ∘ f : X → wm Z`.
- We have the first projection `fst_wm : X ∗ Y → wm X`, which sends a pair `(x, y)` to `x`, and the second projection `snd_wm : X ∗ Y → wm Y`, which sends `(x, y)` to `y`.
- Given `f : X → wm Y` and `g : X → wm Z`, we have a function `(f, g) : X → wm (Y ∗ Z)`. For `x : X`, we define `(f, g) x` to be `(f x, g x)`.
- Given `f : Y → X → wm Z` and `g : X → wm Z`, we get the function `λ f : X → wm (Y → wm Z)`. For every `x : X` and `y : Y`, we define `λ f y x` to be `f (y, x)`.
- Given `f : X → wm (Y → wm Z)` and `x : X → wm Y`, we obtain `f ∘ wm x : X → wm Z`, which sends every `a : X` to `f a (x a)`.
- Given `x : X`, we have a weakly monotonic function `apply_el_wm x : (X → wm Y) → wm Y` which sends `f : X → wm Y` to `f x`.

We also show how to interpret contexts, and to do so, we need to interpret the empty context and context extension. For those, we define the unit and product of compatible relations.

Definition 3.7 (sem_Var). We interpret variables with the following function.
We need the following data in order to provide semantics to terms. An arity function
\( \text{ar} : F \to \text{ty} B \), together with its interpretation \( \text{semF} : \forall (f : F), \text{sem_Ty} (\text{ar} f) \), and an
application operator given by
\[
\text{semApp} : \forall (A1 A2 : \text{ty} B), (\text{sem_Ty} A1 \to \text{wm} \text{sem_Ty} A2) \ast \text{sem_Ty} A1 \to \text{wm} \text{sem_Ty} A2
\]
to interpret term application.

▶ Remark 3.8. A first, but incorrect, guess to interpret application would have been by
interpreting the application of \( f : \text{sem_Ty} A1 \to \text{wm} \text{sem_Ty} A2 \) to \( x : \text{sem_Ty} A1 \) by \( f x \). However,
there is a significant disadvantage of this interpretation. Ultimately, we want to deduce
strong normalization from the interpretation, and the main idea is that if we have a rewrite
\( x \leadsto x' \), then we have \( \text{semTm} x > \text{semTm} x' \). This requirement would not be satisfied if we
interpret application of our terms as actual applications as functions. Indeed, if we have
\( x < x' \), then one is not guaranteed that we also have \( f x < f x' \), because \( f \) is only weakly
monotone.

There are two ways to deal with this. One way is by interpreting function types as strictly
monotonic maps [18]. In this approach, this interpretation of application is valid. However,
it comes at a price, because the interpretation of lambda abstraction becomes more difficult.

Another approach, which we use here, is also used in [8]. We add a parameter to our
interpretation method, namely \( \text{semApp} \), which abstractly represents the interpretation of
application. To deduce strong normalization in this setting, we add requirements about
\( \text{semApp} \) in Section 3.2. As a result, in concrete instantiations of this method, we need to
provide an actual definition for \( \text{semApp} \). We see this in Section 4.2.

▶ Definition 3.9 (\( \text{sem_Tm} \)). Given a function \( \text{semF} : \forall (f : F), \text{sem_Ty} (\text{ar} f) \), the
semantics of terms is given by
\[
\text{Fixpoint} \quad \text{sem_Tm} \{ C : \text{con} B \} \{ A : \text{ty} B \} \{ t : \text{tm} \text{ar} C A \} : \text{sem_Con C} \to \text{wm} \text{sem_Ty} A :=
\begin{align*}
\text{match} \ t \ \text{with} \\
| \text{BaseTm} f & \Rightarrow \text{const_wm} (\text{semF} f) \\
| \text{TmVar} v & \Rightarrow \text{sem_Var} v \\
| \lambda f & \Rightarrow \lambda \text{wm} (\text{sem_Tm} f) \\
| f \cdot t & \Rightarrow \text{semApp} _ _ \circ \text{wm} (\text{sem_Tm} f, \text{sem_Tm} t)
\end{align*}
\end{array}
\]
end.

Notice that we could have chosen a fixed way of interpreting application. We follow the
same approach used by Fuhs and Kop [8] in our formalization and leave \( \text{semApp} \) abstract.
This choice is essential if we want to use the interpretation method for both \textit{rule removal}
and the \textit{dependency pair} approach. See [15, Chapters 4 and 6] for more detail.

3.2 Termination Models for AFSs

Now we have set up everything that is necessary to define the main notion of this section:
\textit{termination models}. From a termination model of an algebraic functional system, one obtains
an interpretation of the types and terms. In addition, every rewrite rule is ‘satisfied’ in this interpretation.

**Definition 3.10 (Interpretation).** Let $\mathcal{R}$ be an algebraic functional system with base type $B$ and function symbols $F$. A **termination model** of $\mathcal{R}$ consists of

- an interpretation key $\text{sem}_B$;
- a function $\text{sem}_F : \text{forall} \ (f : F), \ \text{sem}_T (\text{ar} f)$;
- a function

$$\text{sem}_\text{App} : \text{forall} \ (A1 \ A2 : \text{ty} B), \ (\text{sem}_\text{Tm} A1 \rightarrow \text{wm} \ \text{sem}_\text{Tm} A2) \ast \ \text{sem}_\text{Tm} A1 \rightarrow \text{wm} \ \text{sem}_\text{Tm} A2$$

such that the following axioms are satisfied

- each $\text{sem}_B b$ is well-founded and inhabited;
- if $f > f'$, then $\text{sem}_\text{App} (f, x) > \text{sem}_\text{App} (f', x)$;
- if $x > x'$, then $\text{sem}_\text{App} (f, x) > \text{sem}_\text{App} (f, x')$;
- we have $\text{sem}_\text{App} (f, x) = \text{sem}_\text{Tm} f x$ for all $f$ and $x$;
- for every rewrite rule $r$, substitution $s$, and element $x$, we have

$$\text{sem}_\text{Tm} (\text{lhs} r [s]) x > \text{sem}_\text{Tm} (\text{rhs} r [s]) x$$

Whereas the left-hand side of every rewrite rule is greater than its right-hand side, this does not hold for $\beta$-reduction in our interpretations. Since rewrite sequences can contain both rewrite rules and $\beta$-reduction, such sequences are not guaranteed to strictly decrease. As such, we need more to actually conclude strong normalization, and we follow the method used by Kop [15]. More specifically, Kop uses rule removal to show that strong normalization follows from the strong normalization of $\beta$-reduction, which is a famous theorem proven by Tait [27]. The strong normalization of $\beta$-reduction has been formalized numerous times and an overview can be found in [1]. Now we deduce the main theorem of this section.

**Theorem 3.11 (afs_is_SN_from_Interpretation).** Let $\mathcal{R}$ be an algebraic functional system. If we have a termination model of $X$, then $X$ is strongly normalizing.

### 4 The Higher-Order Polynomial Method

#### 4.1 Polynomials

In this section, we instantiate the material of Section 3 to a concrete instance, namely the *polynomial method* [8]. For that reason, we define the notation of *higher-order polynomial*.

**Definition 4.1 (Polynomial.v).** We define the type $\text{base}_\text{poly}$ of *base polynomials* and $\text{poly}$ of *higher-order polynomials* by mutual induction as follows:

```plaintext
Inductive base_poly {B : Type} : B → Type :=
| P_const : forall {C : con B},
  nat → base_poly C
| P_plus : forall {C : con B},
  (P1 P2 : base_poly C) → base_poly C
| P_mult : forall {C : con B},
  (P1 P2 : 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} {A : ty B},
  poly C (A → A) →
  poly C (A → C → A)
| P_lam : forall {C : con B} {A : ty B},
  poly (A → C → A) → poly (A _, C → A → C (A → A)).
```

CVIT 2016
We can make expressions of base polynomials using \(P_{\text{const}}\) (constants), \(P_{\text{plus}}\) (addition), and \(P_{\text{mult}}\) (multiplication). In addition, \(\text{from\_poly}\) takes an inhabitant of \(\text{poly}\ C\ (\text{Base}\ b)\) and returns a base polynomial in context \(C\). Using \(P_{\text{base}}\), we can turn a base polynomial into a polynomial of any base type. The constructors, \(P_{\text{var}}, P_{\text{app}},\) and \(P_{\text{lam}}\), are reminiscent of the simply typed lambda calculus. We get variables from \(P_{\text{var}},\) \(\lambda\)-abstraction from \(P_{\text{lam}},\) and application from \(P_{\text{app}}\). Note that combining \(\text{from\_poly}\) and \(P_{\text{var}}\), we can use variables in base polynomials.

Let us make some remarks about the design choices we made and how they affected the definition of polynomials. One of our requirements is that we are able to add and multiply polynomials on different base types. This is frequently used in actual examples, such as Example 4.2. Function symbols might use arguments from different base types, and we would like to use both of them in polynomial expressions.

One possibility would have been to only work with the type \(\text{poly}\) and to add a constructor \(P_{\text{plus}}:\ \forall\{C:\ \text{con}\ B\}\ ((b_1 : B),\ \text{poly}\ C\ (\text{Base}\ b_1) \rightarrow \text{poly}\ C\ (\text{Base}\ b_2)) \rightarrow \text{poly}\ C\ (\text{Base}\ b_1)\) However, we refrained from doing so: if we were to use \(P_{\text{const}}\), then the elaborator would be unable to determine the actual type if we do not tell the base type explicitly. Instead, we used a type of base polynomials that does not depend on the actual base type. This is the role of \(\text{base\_poly}\), which only depends on the variables being used. We can freely add and multiply inhabitants of \(\text{base\_poly}\), and if we were to use a constant, then we do not explicitly need to mention the base type. In addition, we are able to transfer between \(\text{base\_poly}\) and \(\text{poly}\ C\ (\text{Base}\ b)\), and that is what \(P_{\text{base}}\) and \(\text{from\_poly}\) enable us to do.

Note that our definition of higher-order polynomials is rather similar to the one given by Fuhs and Kop [8, Definition 4.1]. They define a set \(\text{Pol}(X)\), which consists of polynomial expressions, and for every type \(A\) a set \(\text{Pol}^A(X)\). The set \(\text{Pol}^A(X)\) is defined by recursion: for base type, it is the set of polynomials over \(X\) and for function types \(A_1 \rightarrow A_2\), it consists of expressions \(\Lambda (y : A_1).P\) where \(P\) is a polynomial of type \(A_2\) using an extra variable \(y : A_1\). Our \(\text{base\_poly}\ C\) and \(\text{poly}\ C\ A\) correspond to \(\text{Pol}(X)\) and \(\text{Pol}^A(X)\) respectively. However, there are some differences. First of all, Fuhs and Kop require variables to be fully applied, whereas we permit partially applied variables. Secondly, Fuhs and Kop define polynomials in such a way that for every two base types \(b_1, b_2\) the types \(\text{Pol}^{b_1}(X)\) and \(\text{Pol}^{b_2}(X)\) are equal. This is not the case in our definition: instead we use constructors \(\text{from\_poly}\) and \(P_{\text{base}}\) to relate \(\text{base\_poly}\ C\) and \(\text{poly}\ C\ (\text{Base}\ b)\).

In the polynomial method, the interpretation key sends every base type to \(\text{nat\_CompatRel}\), and in what follows, we write \([\ C\ ]\text{con}\) and \([\ A\ ]\text{ty}\) for the interpretation of contexts and types respectively. Note that every polynomial \(P : \text{poly}\ C\ A\) gives rise to a weakly monotonic function \(\text{sem\_poly}\ P : C\ [\ C\ ]\text{con} \rightarrow \text{wm}\ [\ A\ ]\text{ty}\) and that every base polynomial \(P : \text{base\_poly}\ C\) gives rise to \(\text{sem\_base\_poly}\ P : [C\ ]\text{con} \rightarrow \text{wm}\ \text{nat\_CompatRel}\). These two functions are defined using mutual recursion and every constructor is interpreted in the expected way: \(\text{sem\_poly}\).

In order to actually use \(\text{base\_poly}\ C\) and \(\text{poly}\ C\ A\), we provide convenient notations for operations on polynomials. More concretely, we define notations \(+,\ ·,\ \ast\) that represent addition, multiplication, and application respectively. These operations must be overloaded since we need to be able to add polynomials of different types. To do so, we similarly use type classes to \text{MathClasses}\ [26]. For details, we refer the reader to the formalization.

\textbf{Example 4.2} (\text{map\_fun\_poly}). We continue with Example 2.7 and provide a polynomial interpretation to the system \text{map\_afs} as follows:

\textbf{Definition} \text{map\_fun\_poly}\ \text{fn\_symbols} : \text{poly}\ • (\text{arity}\ \text{trs}\ \text{fn\_symbols}) :=
match fn_symbols with
| Tnill ⇒ to_Poly (P_const 3)
| Tcons ⇒ \P \let y1 := P_var Vz in
to_Poly (P_const 3 + P_const 2 + y1)
| Tmap ⇒ \P \let y0 := P_var (Vs Vz) in \P \let G1 := P_var Vz in
to_Poly (P_const 3 * y0 + P_const 3 * y0 * (G1 \P (y0)))
end.

Informally, the interpretation of nil is the constant 3. The interpretation of cons is the
function that sends \( y_1 : \mathbb{N} \) to \( 3 + 2y_1 \), and map is interpreted as the function that sends \( y_0 : \mathbb{N} \) and \( G_1 : \mathbb{N} \rightarrow \text{wm} \mathbb{N} \) to \( 3y_0 + 3y_0 G_1(y_0) \).

4.2 Polynomial Interpretation

Using polynomials, we deduce strong normalization under certain circumstances using Theo-
rem 3.11. Suppose that for all function symbols \( f \) we have a polynomial \( J : \text{poly} \bullet (\text{arity X f}) \),
and now we need to provide the interpretation for application. Following Fuhs and Kop [8],
we use a general method to interpret application. We start by constructing a minimal element
in the interpretation of every type.

► Definition 4.3 (min_el_ty). For every simple type \( A \) we define a minimal element of
\( \llbracket A \rrbracket_{\text{ty}} \) as follows

\[
\text{Fixpoint } \text{min_el_ty} \{ A : \text{ty} B \} : \text{minimal_element} \llbracket A \rrbracket_{\text{ty}} := \text{match } A \text{ with}
\]
\[| \text{Base } _\_ \Rightarrow \text{nat_minimal_element} \]
\[| A_1 \rightarrow A_2 \Rightarrow \text{min_el_fun_space} (\text{min_el_ty} A_2) \]
end.

Here \text{nat_minimal_element} is defined to be 0, and \text{min_el_fun_space} (\text{min_el_ty} A_2) is the
constant function on (\text{min_el_ty} A_2).

In order to define the semantics of application, we need several operations involving
\( \llbracket A \rrbracket_{\text{ty}} \). First, we consider lower value functions.

► Definition 4.4 (lvf). We define the lower value function as follows

\[
\text{Fixpoint } \text{lvf} \{ A : \text{ty} B \} : \llbracket A \rrbracket_{\text{ty}} \rightarrow \text{wm} \text{natCompatRel} := \text{match } A \text{ with}
\]
\[| \text{Base } _\_ \Rightarrow \text{id_wm} \]
\[| A_1 \rightarrow A_2 \Rightarrow \text{lvf} \circ \text{wm apply_el_wm} (\text{min_el_ty} A_1) \]
end.

Note that we construct \text{lvf} directly as a weakly monotonic function. In addition, we reuse
the combinators defined in Definition 3.6. As such, we do not need to prove separately that
this function is monotonic.

In Kop and Fuhs [8], this definition is written down in a different, but equivalent, way.
Instead of defining \text{lvf}_A recursively, they look at full applications, which would be more
complicated in our setting. More specifically, since we are working with simple types, we must
have that \( A = A_1 \rightarrow \ldots \rightarrow A_n \rightarrow B \). Then they define \( \text{lvf}_A(f) := f(\bot_{A_1}, \ldots, \bot_{A_n}) \),
where \( \bot_A \) is the minimum element of the interpretation of \( A \). Next, we define two addition
operations on \( \llbracket A \rrbracket_{\text{ty}} \).

► Definition 4.5 (plus_ty_nat). Addition of natural numbers and elements on \( \llbracket A \rrbracket_{\text{ty}} \) is defined as follows

\[
\text{match fn_symbols with}
\]
\[| Tnill ⇒ to_Poly (P_const 3)
| Tcons ⇒ \lambda \P \let y1 := P_var Vz in
to_Poly (P_const 3 + P_const 2 + y1)
| Tmap ⇒ \lambda \P \let y0 := P_var (Vs Vz) in \lambda \P \let G1 := P_var Vz in
to_Poly (P_const 3 * y0 + P_const 3 * y0 * (G1 \P (y0)))
end.

CVIT 2016
Fixpoint plus_ty_nat {A : ty B} : [A \ty * nat_CompatRel \to \wm [A \ty
:= match A with
  | Base _ \to plus_wm
  | A1 \to A2 \to
    let f := fst_wm \circ \wm snd_wm in
    let x := fst_wm in
    let n := snd_wm \circ \wm snd_wm in
    \wm (plus_ty_nat \circ \wm (f \cdot \wm x, n))
end.

The function plus_ty_nat allows us to add arbitrary natural numbers to elements of the
interpretation of types. Note that there are two cases in Definition 4.5. First of all, the type
A could be a base type. In that case, we are adding two natural numbers, and we use the usual
addition operation. In the second case, we are working with a functional type A1 \to A2.
The resulting function is defined using pointwise addition with the relevant natural number.
Now we have everything in place to define the interpretation of application.

Definition 4.6 (p_app). Application is interpreted as the following function

Definition p_app {A1 A2 : ty B}
:= \let f := fst_wm \in
\let x := snd_wm \in
plus_ty_nat \circ \wm (f \cdot \wm x, lvf \circ \wm x).

If both A1 and A2 are base types, then p_app (f, x) reduces to f x + x. Note that p_app
satisfies the requirements from Theorem 3.11. Hence, we obtain the following.

Theorem 4.7 (poly_Interpretation). Let R be an AFS. Suppose that for every function
symbol f we have a polynomial p_fun_sym f such that for all rewrite rules l \to r in R we
have semTm l x > semTm r x for all x. Then R has a termination model.

4.3 Constraint Solving Tactic

Notice that in order to formally verify a proof of termination of a system using Theorem 4.7,
we need to provide a polynomial interpretation and show that [\ell] > [r] holds for all rules
\ell \to r. This will introduce inequality proof goals into the Coq context that must be solved.

Example 4.8. Let us consider a concrete example. We use the polynomials given in
Example 4.2 to show strong normalization of Example 2.7. This example introduces two
inequalities, one for each rule. Let G0 : \N \to \wm \N be weakly monotonic. For rule map_nil,
we need to prove that for all G0, the constraint 12 + G0(0) + 9G0(3) > 3 holds. For the second
rule, map_cons, the constraint is: 12 + 4y0 + 12y1 + G0(0) + (3y0 + 9y1 + 9)G0(3 + y0 + 3y1) >
3 + y0 + 12y1 + 3G0(0) + G0(y0) + 9y1G0(3y1), for all y0, y1 \in \N and G0.

Finding witnesses for such inequalities is tedious, and we would like to automate this
task. For that reason, we developed a tactic (solve_poly) that automatically solves the
inequalities coming from Theorem 4.7. Essentially, this tactic tries to mimic how one would
solve those goals in a pen-and-paper proof, and the same method is used by Wanda.

Example 4.9. We show how to solve the constraint arising from map_cons mentioned
in Example 4.8. The first step is to find matching terms on both sides of the inequality
and subtract them. In our example, $3 + y_0 + 12y_1 + G_0(0)$ occurs on both sides, and after subtraction, we obtain the following constraint:

$$9 + 3y_0 + 9y_1 + (3y_0 + 9y_1 + 9)G_0(3 + y_0 + 3y_1) > 2G_0(0) + G_0(y_0) + 9y_1G_0(y_1).$$

The second step is combining the arguments for the higher-order variable $G_0$ use its monotonicity. Note that each of $0$, $y_0$, and $y_1$ is lesser than or equal to $3 + y_0 + 3y_1$, because they are natural numbers. Since $G_0$ is weakly monotonic, we get

$$2G_0(0) + G_0(y_0) + 9y_1G_0(y_1) \leq (9y_1 + 3)G_0(3 + y_0 + 3y_1).$$

Now we can simplify our original constraint to

$$9 + 3y_0 + 9y_1 + (3y_0 + 9y_1 + 9)G_0(3 + y_0 + 3y_1) > (9y_1 + 3)G_0(3 + y_0 + 3y_1).$$

Since $3y_0 + 9y_1 + 9 \geq 9y_1 + 3$, we have

$$(3y_0 + 9y_1 + 9)G_0(3 + y_0 + 3y_1) \geq (9y_1 + 3)G_0(3 + y_0 + 3y_1).$$

This is sufficient to conclude that the constraints for map_cons are satisfied.

The tactic solve_poly (solve_poly) follows the steps described above. Note that we use the tactic nia, which is a tactic in Coq that can solve inequalities and equations in nonlinear integer arithmetic. More specifically, solve_poly works as follows:

- First, we generate a goal for every rewrite rule, and we destruct the assumptions so that each variable in the context is either a natural number or a function.
- For every variable $f$ that has a function type, we look for pair $(x, y)$ such that $f(x)$ on the left-hand side and $f(y)$ occurs on the right-hand side. We try using nia whether we can prove $x < y$ from our assumptions. If so, we add $x < y$ to the assumptions, and otherwise, we continue.
- The resulting goals with the extra assumptions are solved using nia.

Note that solve_poly is not complete, because nia is incomplete. As such, if a proof using this tactic is accepted by Coq, then that proof is correct. However, if the proof is not accepted, then it does not have to be the case that the proof is false. With the material discussed in this section, we can write down the polynomials given in Example 4.2, and the tactic is able to verify strong normalization.

### 5 Generating Proof Scripts

In this section, we discuss the practical aspects of our verification framework. In principle one can manually encode rewrite systems as Coq files and use the formalization we provide to verify their own termination proofs. However, this is cumbersome to do. Indeed, in Example 2.7 we used abbreviations to make the formal description of $\mathcal{R}_{\text{map}}$ more readable. A rewrite system with many more rules would be difficult to encode manually. Additionally, to formally establish termination we also need to encode proofs. We did this in Example 4.2. The full formal encoding of $\mathcal{R}_{\text{map}}$ and its termination proof is found in the file Map.v.

#### 5.1 Proof traces for polynomial interpretation

This difficulty of manual encoding motivates the usage of proof traces. A proof trace is a human-friendly encoding of a TRS and the essential information needed to reconstruct the
termination proof as a Coq script. Let us again consider $R_{\text{map}}$ as an example. The proof trace for this system starts with \texttt{YES} to signal that we have a termination proof for it. Then we have a list encoding the signature and the rules of the system.

\texttt{YES}
Signature: [
  \texttt{cons : a -> list -> list ;
  map : list -> (a -> a) -> list ;
  nil : list}
]
Rules: [
  \texttt{map nil F => nil ;
  map (cons X Y) G => cons (G X) (map Y G)}
]

Notice that the free variables in the rules do not need to be declared nor their typing information provided. Coq can infer this information automatically. The last section of the proof trace describes the interpretation of each function symbol in the signature.

\texttt{Interpretation: [}
  \texttt{J(cons) = Lam[y0;y1].3 + 2*y1;
  J(map) = Lam[y0;G1].3*y0 + 3*y0 * G1(y0);
  J(nil) = 3}
]\]

We can fully reconstruct a formal proof of termination for $R_{\text{map}}$, which uses Theorem 4.7, with the information provided in the proof trace above. The full description of proof traces can be found in [29], the API for ONijn. Proof traces are not Coq files. So we need to further compile them into a proper Coq script. The schematics in Figure 1 describe the steps necessary for it. We use ONijn to compile proof traces to Coq script. It is invoked as follows:

\texttt{onijn path/to/proof/trace.onijn -o path/to/proof/script.v}

Here, the first argument is the file path to a proof trace file and the \texttt{-o} option requires the file path to the resulting Coq script. The resulting Coq script can be verified by Nijn as follows:

\texttt{coqc path/to/proof/script.v}

Instructions on how to locally install ONijn/Nijn can be found at [29].

5.2 Verifying Wanda’s Polynomial Interpretations

It is worth noticing that the termination prover is abstract in our certification framework. This means that we are not bound to a specific termination tool. So we can verify any termination tool that implements the interpretation method described here and can output proof traces in ONijn format.

Since Wanda [16] is a termination tool that implements the interpretation method in [8], it is our first candidate for verification. We added to Wanda the runtime argument \texttt{--formal} so it can output proof traces in ONijn format. In [16] one can find details on how to invoke Wanda. For instance, we illustrate below how to run Wanda on the \texttt{map} AFS.

\texttt{./wanda.exe -d rem --formal Mixed_HO_10_map.afs}
The setting \texttt{-d rem} sets Wanda to disable rule removal. The option \texttt{--formal} sets Wanda to only use polynomial interpretations and output proofs to ONijn proof traces. Running Wanda with these options gives us the proof trace we used for $R_{map}$ above. The latest version of Wanda, which includes this parameter, is found at [17].

The table below describes our experimental evaluation on verifying Wanda’s output with the settings above. The benchmark set consists of those 46 TRSs that Wanda outputs YES while using only polynomial interpretations and no rule removal. The time limit for certification of each system is set to 60 seconds.

The experiment was run in a machine with M1 Pro 2021 processor with 16GB of RAM. Memory usage of Nijn during certification ranges from 400MB to 750MB. We provide the experimental benchmarks at https://github.com/deividrvale/nijn-coq-script-generation.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Wanda</th>
<th>Nijn/ONijn</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>46</td>
<td>23%</td>
</tr>
</tbody>
</table>

Table 1 Experimental Results

Hence, we can certify all TRSs proven SN by Wanda using only polynomial interpretations.

6 Conclusions and Future Work

We presented a formalization of the polynomial method in higher-order rewriting. This not only included the basic notions, such as algebraic functional systems, but also the interpretation method and the instantiation of this method to polynomials. In addition, we showed how to generate Coq scripts from the output of termination provers. This allowed us to certify their output and construct a formal proof of strong normalization. We also applied our tools to a concrete instance, namely to check the output of Wanda.

There are numerous ways to extend this work. First, one could formalize more techniques from higher-order rewriting, such as tuple interpretations [18] and dependency pairs [19, 22]. One could also integrate HORPO into our framework [20]. Second, in the current formalization, the interpretation of application is fixed for every instance of the polynomial method. One could also provide the user with the option to select their own interpretation. Third, currently, only Wanda is integrated with our work. This could be extended so that there is direct integration for other tools as well.

References


23:20 REFERENCES


