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.ITP.2023.30

Related Version *Preprint version*: `https://arxiv.org/abs/2302.11892` [32]

Supplementary Material *Software (Coq Formalization)*: `github.com/nmvdw/Nijn` [31]
Software (Proof script generator): `github.com/deividrvale/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], $\mathsf{T_TT_2}$ [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

14th International Conference on Interactive Theorem Proving (ITP 2023).
Editors: Adam Naumowicz and René Thiemann; Article No. 30; pp. 30:1–30:20
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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 [9, 19, 3], recursive path orders [20, 5], rule removal, and multiple kinds of interpretations [8, 23, 34, 18]. 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.

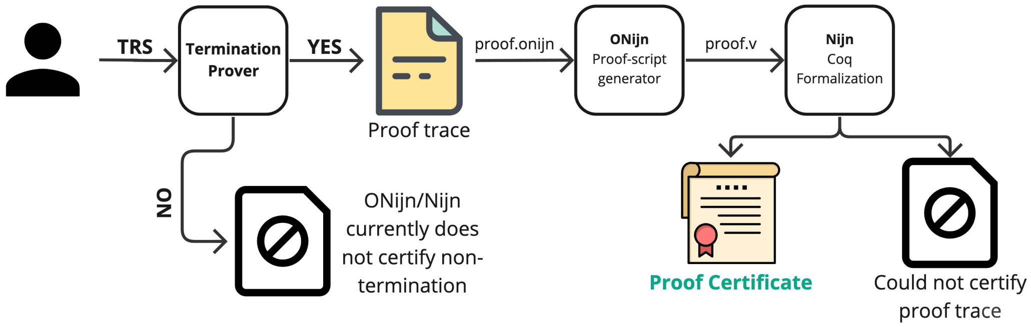
TRS
Termination Prover
YES
Proof trace
proof.onijn
ONijn Proof-script generator
proof.v
Nijn Coq Formalization
NO
ONijn/Nijn currently does not certify non-termination
Proof Certificate
Could not certify proof trace

Figure 1 Nijn/ONijn schematics.

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:

```
spec    proof comments
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 $(\!|A|\!)$ to $(\!|B|\!)$ where $(\!|A|\!)$, $(\!|B|\!)$ denote the interpretations of A, B respectively. The second step is to map inhabitants of a type A to elements of $(\!|A|\!)$, 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 $\ell \to r$. 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:

```
Inductive ty (B : Type) : Type :=
| Base : B → ty B
| Fun : ty B → ty B → 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, $[x_0 : A_0; \ldots ; x_n : A_n]$ is the context with variables x_0 of type A_0, …, x_n of type A_n. However, we use nameless variables in our development, so we do not keep track of their names. Consequently, a context is represented by a list of types. 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 2.2** (`con`). The type of **variable contexts** over a type `B` is defined as follows.

```
Inductive con (B : Type) : Type :=
| Empty : con B
| Extend : ty B → con B → 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

```
Inductive var {B : Type} : con B → ty B → Type :=
| Vz : forall {C : con B} {A : ty B}, var (A ,,  C) A
| Vs : forall {C : con B} {A1 A2 : ty B}, var C A2 → var (A1 ,, C) A2.
```

Let us consider an example of a context and some variables. Suppose that we have a base type denoted by `b`. Then we can form the context `Base b ,, Base b ⟶ Base b ,, Empty`. In this context, we have two variables. The first one, which is `Vz`, has type `Base b`, and the second variable, which is `Vs Vz`, has type `Base b ⟶ Base b`, The context that we discussed corresponds to $[x_0 : b; x_1 : b \longrightarrow b]$. The variable `Vz` represents x_0, while `Vs Vz` represents x_1.

In Definition 2.4 below we define the notion of *well-typed terms-in-context* which consists of those expressions such that there is a typing derivation. We use dependent types to ensure well-typedness of such expressions. The type of terms depends on a simple type `A : ty B` (which represents the object-level type of the expression) and context `C : con B` that carries the types of all free variables in the term. We also need to type function symbols. Hence, we require a type `F : Type` of function symbols and `ar : F → ty B`, which maps `f : F` to a simple type `ar f`.

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

```
Inductive tm {B : Type} {F : Type} (ar : F → ty B) (C : con B) : ty B → Type :=
| BaseTm : forall (f : F), tm ar C (ar f)
| TmVar : forall {A : ty B}, var C A → tm ar C A
| Lam : forall {A1 A2 : ty B}, tm ar (A1 ,, C) A2 → tm ar C (A1 ⟶ A2)
| App : forall {A1 A2 : ty B}, tm ar C (A1 ⟶ A2) → tm ar C A1 → tm ar C A2.
```

For every function `f : F` we have a term `BaseTm f` of type `ar f`. Every variable `v` gives rise to a term `TmVar v`. For λ-abstractions, given a term `s : tm ar (A1 ,, C) A2`, there is a term λ `s : tm ar C (A1 ⟶ A2)`, namely `Lam s`. The last constructor represents term application. If we have a term `s : tm ar C (A1 ⟶ A2)` and a term `t : tm ar C A1`, we get a term `s · t : tm ar C A2`, which is defined to be `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 α-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

▶ **Definition 2.5** (`rewriteRule`). The type of **rewrite rules** is defined as follows:

```
Record rewriteRule {B : Type} {F : Type} (ar : F → ty B) :=
 make_rewrite {
    vars_of : con B ;
    tar_of : ty B ;
    lhs_of : tm ar vars_of tar_of ;
    rhs_of : tm ar vars_of tar_of }.
```

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

▶ **Definition 2.6** (`afs`). The type of **algebraic functional systems** is defined as follows

```
Record afs (B : Type) (F : Type) :=
make_afs { arity : F → ty B ; list_of_rewriteRules : list (rewriteRule arity) }.
```

As usual, every AFS induces a rewrite relation on the set of terms, which we denote by ~>. The formal definition is found in `RewritingSystem.v`. The rewrite relation ~> 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 β-reduction, and we have a constructor for the rewrite rules of the AFS.

▶ **Example 2.7** (`map_afs`). Let us encode $\mathcal{R}_{\mathsf{map}}$ in Coq. It is composed of two rules: $\mathsf{map}\, F\, \mathsf{nil} \to \mathsf{nil}$ and $\mathsf{map}\, F\, (\mathsf{cons}\, x\, xs) \to \mathsf{cons}\, (F\, x)\, (\mathsf{map}\, F\, xs)$. We start with base types.

```
Inductive base_types := TBtype | TList.
Definition Btype : ty base_types := Base TBtype.
Definition List : ty base_types := Base TList.
```

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

```
Inductive fun_symbols := TNil | TCons | TMap.
```

The arity function `map_ar` maps each function symbol in `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} : tm map_ar 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 R 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 $s_0, s_1, \ldots$ such that $R(s_n, s_{n+1})$, 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 [24, 8]. 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.

```
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 $>, \geq$ such that $>$ is well-founded and $\geq$ 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 := {|
  carrier := unit ;
  gt _ _ := False ;
  ge _ _ := True |}.
```

```
Definition prod_CompatRel (X Y : CompatRel) :
CompatRel := {|
  carrier := X ∗ Y ;
  gt x y := fst x > fst y ∧ snd x > snd y ;
  ge x y := fst x >= fst y ∧ snd x >= snd y |}.
```

Note that unit_CompatRel is the compatible relation on the type with only one element, for which the ordering is trivial. In addition, prod_CompatRel is the compatible relation on the product, for which we compare elements coordinate-wise. We write X ∗ Y for prod_CompatRel X Y.

▶ **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.
```

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 variables are interpreted as weakly monotonic functions. 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 3.6** (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 g ∘wm f : X →wm Z to be their composition.
- 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, we get λwm f : X →wm (Y →wm Z). For every x : X and y : Y, we define λwm 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.

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 3.7** (sem_Var)**.** We interpret variables with the following function

```
Fixpoint sem_Var {C : con B} {A : ty B} (v : var C A) : sem_Con C →wm sem_Ty A
  := match v with
     | Vz ⇒ fst_wm
     | Vs v ⇒ sem_Var v ∘wm snd_wm
     end.
```

We need the following data in order to provide semantics to terms. An arity function ar : F → ty B, together with its interpretation semF : forall (f : F), sem_Ty (ar f), and an *application operator* given by

```
semApp : forall (A1 A2 : ty B), (sem_Ty A1 →wm sem_Ty A2) * sem_Ty A1 →wm 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 : sem_Ty A1 →wm sem_Ty A2 to x : 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 ∼> x', then we have semTm x > 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 semApp, which abstractly represents the interpretation of application. To deduce strong normalization in this setting, we add requirements about semApp in Section 3.2. As a result, in concrete instantiations of this method, we need to provide an actual definition for semApp. We see this in Section 4.2.

▶ **Definition 3.9** (sem_Tm)**.** Given a function semF : forall (f : F), sem_Ty (ar f), the semantics of terms is given by

```
Fixpoint sem_Tm {C : con B} {A : ty B} (t : tm ar C A) : sem_Con C →wm sem_Ty A :=
  match t with
  | BaseTm f  ⇒ const_wm (semF f)
  | TmVar v   ⇒ sem_Var v
  | λ f       ⇒ λwm (sem_Tm f)
  | f · t     ⇒ semApp _ _ ∘wm ⟨sem_Tm f , sem_Tm t ⟩
  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 semApp abstract. This choice is essential if we want to use the interpretation method for both *rule removal* and the *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: *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 `semB`;
- a function `semF : forall (f : F), sem_Ty (ar f)`;
- a function

```
semApp : forall (A1 A2 : ty B), (sem_Ty A1 →wm sem_Ty A2) * sem_Ty A1 →wm sem_Ty A2
```

such that the following axioms are satisfied

- each `semB b` is well-founded and inhabited;
- if `f > f'`, then `semApp _ _ (f , x) > semApp _ _ (f' , x)`;
- if `x > x'`, then `semApp _ _ (f , x) > semApp _ _ (f , x')`;
- we have `semApp _ _ (f , x) >= f x` for all `f` and `x`;
- for every rewrite rule `r`, substitution `s`, and element `x`, we have

```
semTm (lhs r [ s ]) x > semTm (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 β-reduction in our interpretations. Since rewrite sequences can contain both rewrite rules and β-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 β-reduction, which is a famous theorem proven by Tait [27]. The strong normalization of β-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 `base_poly` of **base polynomials** and `poly` of **higher-order polynomials** by mutual induction as follows:

```
Inductive base_poly {B : Type}
    : con 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} {1A 2A : ty B},
  poly C (1A ⟶2 A)→
   poly C 1A→
   poly C 2A
| P_lam : forall {C : con B} {1A 2A : ty B},
  poly (1A ,,  C) 2A → poly C (1A ⟶2 A).
```

We can make expressions of base polynomials using `P_const` (constants), `P_plus` (addition), and `P_mult` (multiplication). In addition, `from_poly` takes an inhabitant of `poly C (Base b)` and returns a base polynomial in context `C`. Using `P_base`, we can turn a base polynomial into a polynomial of any base type. The constructors, `P_var`, `P_app`, and `P_lam`, are remniscent of the simply typed lambda calculus. We get variables from `P_var`, λ-abstraction from `P_lam`, and application from `P_app`. Note that combining `from_poly` and `P_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 `poly` and to add a constructor

```
P_plus : forall {C : con B} (b1 b2 : B),
         poly C (Base b1) → poly C (Base b2) → poly C (Base b1)
```

However, we refrained from doing so: if we were to use `P_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 `base_poly`, which only depends on the variables being used. We can freely add and multiply inhabitants of `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 `base_poly` and `poly C (Base b)`, and that is what `P_base` and `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 $\mathsf{Pol}(X)$, which consists of polynomial expressions, and for every type A a set $\mathsf{Pol}^A(X)$. The set $\mathsf{Pol}^A(X)$ is defined by recursion: for base type, it is the set of polynomials over X and for function types $A_1 \longrightarrow A_2$, 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 `base_poly C` and `poly C A` correspond to $\mathsf{Pol}(X)$ and $\mathsf{Pol}^A(X)$ 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 $\mathsf{Pol}^{b_1}(X)$ and $\mathsf{Pol}^{b_2}(X)$ are equal. This is not the case in our definition: instead we use constructors `from_poly` and `P_base` to relate `base_poly C` and `poly C (Base b)`.

In the polynomial method, the interpretation key sends every base type to `nat_CompatRel`, and in what follows, we write ⟦ `C` ⟧`con` and ⟦ `A` ⟧`ty` for the interpretation of contexts and types respectively. Note that every polynomial `P : poly C A` gives rise to a weakly monotonic function `sem_poly P : ⟦ C ⟧con →wm ⟦ A ⟧ty` and that every base polynomial `P : base_poly C` gives rise to `sem_base_poly P : ⟦C ⟧con →wm nat_CompatRel`. These two functions are defined using mutual recursion and every constructor is interpreted in the expected way: `sem_poly`.

In order to actually use `base_poly C` and `poly C A`, we provide convenient notations for operations on polynomials. More concretely, we define notations $+$, $*$, and ·P 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 MathClasses [26]. For details, we refer the reader to the formalization.

▶ **Example 4.2** (`map_fun_poly`). We continue with Example 2.7 and provide a polynomial interpretation to the system `map_afs` as follows:

```
Definition map_fun_poly fn_symbols : poly •(arity trs fn_symbols) :=
   match fn_symbols with
   | Tnil ⇒ to_Poly (P_const 3)
   | 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)))
   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 $G_1 : \mathbb{N} \to_{\mathrm{wm}} \mathbb{N}$ to $3y_0 + 3y_0G_1(y_0)$.

4.2 Polynomial Interpretation

Using polynomials, we deduce strong normalization under certain circumstances using Theorem 3.11. Suppose that for all function symbols `f` we have a polynomial `J : poly • (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 `⟦ A ⟧ty` as follows

```
Fixpoint min_el_ty (A : ty B) : minimal_element ⟦A ⟧ty
   := match A with
      | Base _ ⇒ nat_minimal_element
      | A1 ⟶ A2 ⇒ min_el_fun_space (min_el_ty A2)
      end.
```

Here `nat_minimal_element` is defined to be 0, and `min_el_fun_space (min_el_ty A2)` is the constant function on `(min_el_ty A2)`.

In order to define the semantics of application, we need several operations involving `⟦ A ⟧ty`. First, we consider *lower value functions.*

▶ **Definition 4.4** (`lvf`). We define the **lower value function** as follows

```
Fixpoint lvf {A : ty B} : ⟦ A ⟧ty →wm nat_CompatRel :=
  match A with
    | Base _ ⇒ id_wm
    | A1 ⟶ A2 ⇒ lvf ∘wm apply_el_wm (min_el_ty A1)
  end.
```

Note that we construct 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 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 $A = A_1 \longrightarrow \ldots \longrightarrow A_n \longrightarrow B$. Then they define $\mathsf{lvf}_A(f) := f(\bot_{A_1}, \ldots, \bot_{A_n})$, where $\bot_A$ is the minimum element of the interpretation of A. Next, we define two addition operations on ⟦ A ⟧ty.

▶ **Definition 4.5** (plus_ty_nat)**.** Addition of natural numbers and elements on ⟦ A ⟧ty is defined as follows

```
Fixpoint plus_ty_nat {A : ty B} : ⟦A ⟧ty ∗ nat_CompatRel →wm ⟦A ⟧ty
  := match A with
     | Base _ ⇒ plus_wm
     | A1 ⟶ A2 ⇒
       let f := fst_wm ∘wm snd_wm in
       let x := fst_wm in
       let n := snd_wm ∘wm snd_wm in
       λwm (plus_ty_nat ∘wm ⟨f ·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 ⟶ 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}
  : ⟦ A1 ⟶ A2 ⟧ty ∗ ⟦ A1 ⟧ty →wm ⟦ A2 ⟧ty
  := let f := fst_wm in
     let x := snd_wm in
     plus_ty_nat ∘wm ⟨f ·wm x ,  lvf ∘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 $\mathcal{R}$ 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 ~> r *in $\mathcal{R}$ we have* semTm l x > semTm r x *for all* x*. Then $\mathcal{R}$ 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 $G_0 : \mathbb{N} \to_{\text{wm}} \mathbb{N}$ be weakly monotonic. For rule `map_nil`, we need to prove that for all G_0, the constraint $12 + G_0(0) + 9G_0(3) > 3$ holds. For the second rule, `map_cons`, the constraint is: $12 + 4y_0 + 12y_1 + G_0(0) + (3y_0 + 9y_1 + 9)G_0(3 + y_0 + 3y_1) > 3 + y_0 + 12y_1 + 3G_0(0) + G_0(y_0) + 9y_1 G_0(y_1)$, for all $y_0, y_1 \in \mathbb{N}$ and G_0.

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:

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

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

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

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 $\mathcal{R}_{\mathsf{map}}$ 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 $\mathcal{R}_{\mathsf{map}}$ 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 $\mathcal{R}_{\mathsf{map}}$ as an example. The proof trace for this system starts with `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.

```
YES
Signature: [
  cons : a -> list -> list ;
  map : list -> (a -> a) -> list ;
  nil : list
]
Rules: [
  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.

```
Interpretation: [
  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 $\mathcal{R}_{\mathsf{map}}$, 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:

```
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 `-o` option requires the file path to the resulting Coq script. The resulting Coq script can be verified by Nijn as follows:

```
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 `--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 map AFS.

```
./wanda.exe -d rem --formal Mixed_HO_10_map.afs
```

The setting `-d rem` sets Wanda to disable rule removal. The option `--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 $\mathcal{R}_{\mathsf{map}}$ 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 1 Experimental Results.

	Wanda			Nijn/ONijn		
Technique	# YES	Pct.	Avg. Time	# Certified	Perc.	Avg. Time
Poly, no rule removal	46	23%	0.07s	46	100%	4.06s

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

1 Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer, and Kathrin Stark. POPLMark reloaded: Mechanizing proofs by logical relations. *J. Funct. Program.*, 29:e19, 2019. doi:10.1017/S0956796819000170.

2 Ariane Alves Almeida and Mauricio Ayala-Rincón. Formalizing the dependency pair criterion for innermost termination. *Sci. Comput. Program.*, 195:102474, 2020. doi:10.1016/j.scico.2020.102474.

3 Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theor. Comput. Sci.*, 236(1-2):133–178, 2000. doi:10.1016/S0304-3975(99)00207-8.

4 Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. doi:10.1007/978-3-662-07964-5.

5 Frédéric Blanqui, Jean-Pierre Jouannaud, and Albert Rubio. The computability path ordering. *Log. Methods Comput. Sci.*, 11(4), 2015. doi:10.2168/LMCS-11(4:3)2015.

6 Frédéric Blanqui and Adam Koprowski. CoLoR: a Coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. *Math. Struct. Comput. Sci.*, 21(4):827–859, 2011. doi:10.1017/S0960129511000120.

7 Evelyne Contejean, Pierre Courtieu, Julien Forest, Olivier Pons, and Xavier Urbain. Automated certified proofs with cime3. In Manfred Schmidt-Schauß, editor, *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications, RTA 2011, May 30 - June 1, 2011, Novi Sad, Serbia*, volume 10 of *LIPIcs*, pages 21–30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2011. doi:10.4230/LIPIcs.RTA.2011.21.

8 Carsten Fuhs and Cynthia Kop. Polynomial Interpretations for Higher-Order Rewriting. In Ashish Tiwari, editor, *23rd International Conference on Rewriting Techniques and Applications (RTA'12) , RTA 2012, May 28 - June 2, 2012, Nagoya, Japan*, volume 15 of *LIPIcs*, pages 176–192. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012. doi:10.4230/LIPIcs.RTA.2012.176.

9 Carsten Fuhs and Cynthia Kop. A Static Higher-Order Dependency Pair Framework. In Luís Caires, editor, *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, volume 11423 of *Lecture Notes in Computer Science*, pages 752–782. Springer, 2019. doi:10.1007/978-3-030-17184-1_27.

10 Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. Analyzing program termination and complexity automatically with aprove. *J. Autom. Reason.*, 58(1):3–31, 2017. doi:10.1007/s10817-016-9388-y.

11 Jürgen Giesl, Albert Rubio, Christian Sternagel, Johannes Waldmann, and Akihisa Yamada. The termination and complexity competition. In Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 156–166, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-17502-3_10.

12 Raúl Gutiérrez and Salvador Lucas. mu-term: Verify termination properties automatically (system description). In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 436–447. Springer, 2020. doi:10.1007/978-3-030-51054-1_28.

13 Makoto Hamana. Theory and practice of second-order rewriting: Foundation, evolution, and SOL. In Keisuke Nakano and Konstantinos Sagonas, editors, *Functional and Logic Programming - 15th International Symposium, FLOPS 2020, Akita, Japan, September 14-16,*

2020, Proceedings, volume 12073 of *Lecture Notes in Computer Science*, pages 3–9. Springer, 2020. doi:10.1007/978-3-030-59025-3_1.

14 Martin Hofmann and Thomas Streicher. The groupoid model refutes uniqueness of identity proofs. In *Proceedings of the Ninth Annual Symposium on Logic in Computer Science (LICS '94), Paris, France, July 4-7, 1994*, pages 208–212. IEEE Computer Society, 1994. doi:10.1109/LICS.1994.316071.

15 Cynthia Kop. *Higher Order Termination: Automatable Techniques for Proving Termination of Higher-Order Term Rewriting Systems.* PhD thesis, Vrije Universiteit Amsterdam, 2012.

16 Cynthia Kop. WANDA - a higher order termination tool (system description). In Zena M. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference)*, volume 167 of *LIPIcs*, pages 36:1–36:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.FSCD.2020.36.

17 Cynthia Kop. Wanda's source code repository, 2023. URL: https://github.com/hezzel/wanda.

18 Cynthia Kop and Deivid Vale. Tuple Interpretations for Higher-Order Complexity. In Naoki Kobayashi, editor, *6th International Conference on Formal Structures for Computation and Deduction, FSCD 2021, July 17-24, 2021, Buenos Aires, Argentina (Virtual Conference)*, volume 195 of *LIPIcs*, pages 31:1–31:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.FSCD.2021.31.

19 Cynthia Kop and Femke van Raamsdonk. Dynamic Dependency Pairs for Algebraic Functional Systems. *Log. Methods Comput. Sci.*, 8(2), 2012. doi:10.2168/LMCS-8(2:10)2012.

20 Adam Koprowski. Coq formalization of the higher-order recursive path ordering. *Appl. Algebra Eng. Commun. Comput.*, 20(5-6):379–425, 2009. doi:10.1007/s00200-009-0105-5.

21 Martin Korp, Christian Sternagel, Harald Zankl, and Aart Middeldorp. Tyrolean Termination Tool 2. In Ralf Treinen, editor, *Rewriting Techniques and Applications, 20th International Conference, RTA 2009, Brasília, Brazil, June 29 - July 1, 2009, Proceedings*, volume 5595 of *Lecture Notes in Computer Science*, pages 295–304. Springer, 2009. doi:10.1007/978-3-642-02348-4_21.

22 Keiichirou Kusakari, Yasuo Isogai, Masahiko Sakai, and Frédéric Blanqui. Static dependency pair method based on strong computability for higher-order rewrite systems. *IEICE Trans. Inf. Syst.*, 92-D(10):2007–2015, 2009. doi:10.1587/transinf.E92.D.2007.

23 Friedrich Neurauter and Aart Middeldorp. Revisiting matrix interpretations for proving termination of term rewriting. In Manfred Schmidt-Schauß, editor, *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications, RTA 2011, May 30 - June 1, 2011, Novi Sad, Serbia*, volume 10 of *LIPIcs*, pages 251–266. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2011. doi:10.4230/LIPIcs.RTA.2011.251.

24 J.C. van de Pol. *Termination of Higher-order Rewrite Systems.* PhD thesis, University of Utrecht, 1996. URL: https://www.cs.au.dk/~jaco/papers/thesis.pdf.

25 José-Luis Ruiz-Reina, José-Antonio Alonso, María-José Hidalgo, and Francisco-Jesús Martín-Mateos. Formalizing Rewriting in the ACL2 Theorem Prover. In John A. Campbell and Eugenio Roanes-Lozano, editors, *Artificial Intelligence and Symbolic Computation, International Conference AISC 2000 Madrid, Spain, July 17-19, 2000, Revised Papers*, volume 1930 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2000. doi:10.1007/3-540-44990-6_7.

26 Bas Spitters and Eelis van der Weegen. Type classes for mathematics in type theory. *Math. Struct. Comput. Sci.*, 21(4):795–825, 2011. doi:10.1017/S0960129511000119.

27 William W. Tait. Intensional Interpretations of Functionals of Finite Type I. *J. Symb. Log.*, 32(2):198–212, 1967. doi:10.2307/2271658.

28 René Thiemann and Christian Sternagel. Certification of Termination Proofs Using CeTA. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich,*

Germany, August 17-20, 2009. Proceedings, volume 5674 of *Lecture Notes in Computer Science*, pages 452–468. Springer, 2009. doi:10.1007/978-3-642-03359-9_31.

29 Deivid Vale and Niels van der Weide. Onijn documentation, 2022. URL: https://deividrvale.github.io/nijn-coq-script-generation/onijn/index.html.

30 Deivid Vale and Niels van der Weide. deividrvale/nijn-coq-script-generation: First Release of public API, May 2023. doi:10.5281/zenodo.7915736.

31 Niels van der Weide and Deivid Vale. nmvdw/nijn: 1.0.0, May 2023. doi:10.5281/zenodo.7913023.

32 Niels van der Weide, Deivid Vale, and Cynthia Kop. Certifying higher-order polynomial interpretations. *CoRR*, abs/2302.11892, 2023. doi:10.48550/arXiv.2302.11892.

33 Johannes Waldmann. Matchbox: A tool for match-bounded string rewriting. In Vincent van Oostrom, editor, *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings*, volume 3091 of *Lecture Notes in Computer Science*, pages 85–94. Springer, 2004. doi:10.1007/978-3-540-25979-4_6.

34 Akihisa Yamada. Multi-dimensional interpretations for termination of term rewriting. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 273–290. Springer, 2021. doi:10.1007/978-3-030-79876-5_16.

35 Akihisa Yamada, Keiichirou Kusakari, and Toshiki Sakabe. Nagoya Termination Tool. In Gilles Dowek, editor, *Rewriting and Typed Lambda Calculi - Joint International Conference, RTA-TLCA 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8560 of *Lecture Notes in Computer Science*, pages 466–475. Springer, 2014. doi:10.1007/978-3-319-08918-8_32.