Nijn/ONijn: A New Certification Engine for Higher-Order Termination[*]

Cynthia Kop, Deivid Vale, and Niels van der Weide

Institute for Computing and Information Sciences
Radboud University, Nijmegen, The Netherlands
{c.kop,deividvale,nweide}@cs.ru.nl

1 Introduction

In this short paper, we limn a new combination Nijn/ONijn for the certification of higher-order rewriting termination proofs. A complete version of this work has been accepted for publication at ITP 2023 [8]. We follow the following system design in Nijn/ONijn: Nijn [7] is the certifier, a Coq library providing a formalization of the underlying higher-order rewriting theory and ONijn [6] is a proof script generator, an application that when given a minimal description of a termination proof, i.e., *proof trace*, outputs a Coq *proof script*. The proof script is a fully formal description of the syntax signature used by the TRS and the specification of each rule in the system together with the formal steps needed to express its termination. The proof script then utilizes results from Nijn for checking the correctness of the traced proof. Examples of this system design are the combinations Cochinelle/CiME3 [2] and CoLoR/Rainbow [1].

The schematic below depicts the basic steps for producing proof certificates using Nijn/ONijn.

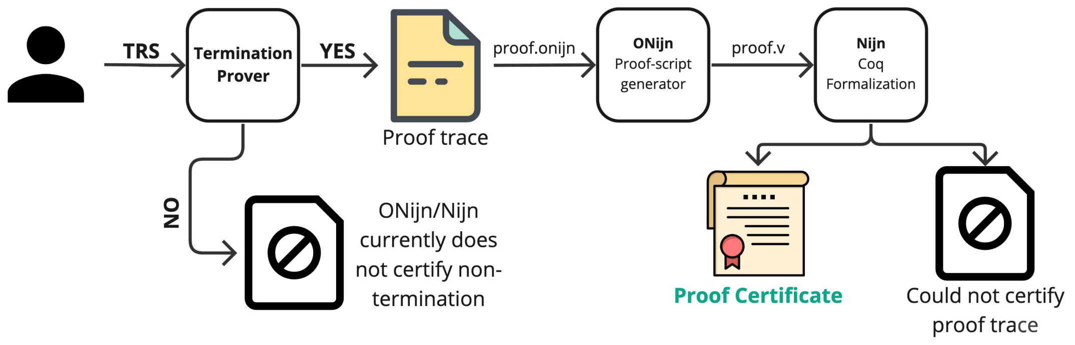
TRS
Termination Prover
YES
Proof trace
proof.onijn
ONijn Proof-script generator
proof.v
Nijn Coq Formalization
NO
ONijn/Nijn currently does not certify non-termination
Proof Certificate
Could not certify proof trace

Figure 1: Nijn/ONijn schematics

A *termination prover* in this schematic is an abstract entity responsible for producing proof traces. It can be either a human, proving termination manually, or a termination tool like Wanda [4], which uses programmed techniques and automated reasoning tools such as SAT/SMT solvers. Whenever a prover outputs a proof trace, we can use ONijn to process it into a formal proof script in Coq. At this moment, we have formalized the polynomial interpretation method.

[*]This work is supported by the following NWO projects: "Implicit Complexity through Higher-Order Rewriting", NWO 612.001.803/7571; NWO VIDI project "Constrained Higher-Order Rewriting and Program Equivalence", NWO VI.Vidi.193.075; and "The Power of Equality" NWO OCENW.M20.380.

Notice that producing the certificates for only this proof method is an inherently incomplete task, since it would require a method to solve inequalities over arbitrary polynomials, which is undecidable in general.

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 (small) as possible. The main task delegated to ONijn is that of parsing the proof trace given by the termination prover to a Coq proof script and perform sanitazation on the prover's input, so that syntax errors are avoided in the proof script. This approach does not pose significant drawbacks in our experience.

2 Encoding TRSs in Nijn

Let us encode $\mathcal{R}_{\mathsf{map}}$ in Coq using Nijn. This will be useful to demonstrate our choices in the formalization and show how to express rewriting systems directly in Coq. The file containing the full enconding can be found at `Map.v`. A simple example of a higher-order system is that of $\mathcal{R}_{\mathsf{map}}$. It represents the higher-order function that applies a function to each element of a list. Recall that $\mathcal{R}_{\mathsf{map}}$ is composed of two rules: $\mathsf{map}\, F\, \mathsf{nil} \to \mathsf{nil}$ and $\mathsf{map}\, F\, (\mathsf{cons}\, x\, xs) \to \mathsf{cons}\, (F\, x)\, (\mathsf{map}\, F\, xs)$. These rules are under a typing context where $F : \mathsf{nat} \Rightarrow \mathsf{nat}, x : \mathsf{nat}$, and $xs : \mathsf{list}$. We start by encoding 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, $\mathsf{map}\, F\, \mathsf{nil} \to \mathsf{nil}$, 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)).
```

3 Practical Aspects of Nijn/ONijn Certification

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 so. Indeed, in the last section 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. The full formal encoding of $\mathcal{R}_{\mathsf{map}}$ and its termination proof is found in the file `Map.v`.

3.1 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 the theory formalized in Nijn, with the information provided in the proof trace above. The full description of proof traces can be found in [6], 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 fig. 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 [6].

3.2 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 [4] is a termination tool that implements the interpretation method in [3], 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 [4] 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 [5].

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.

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

Table 1: Experimental Results

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

4 Conclusion and Future Plans

In this formalization effort, we were successful in certifying higher-order polynomial interpretations. This line of work is far from finished, however. The initial setup of Nijn/ONijn presented here bootstraps the foundation of a full-fledged certification engine for more complex higher-order termination proof techniques. For instance, incorporating the so-called higher-order dependency pair framework is our next immediate future work plan. This will allow us to significantly improve the number of systems we can certify.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

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

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

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

[8] Niels van der Weide, Deivid Vale, and Cynthia Kop. Certifying higher-order polynomial interpretations. In *Proc. ITP 2023 (to appear)*, 2023. URL: https://doi.org/10.48550/arXiv.2302.11892.