arXiv:2302.11892v2 [cs.LO] 5 Aug 2023

高阶多项式解释的认证

Niels van der Weide
I拉德堡德大学计算与信息科学研究所，荷兰奈梅亨
Deivid Vale
I
拉德堡德大学计算与信息科学研究所，荷兰奈梅亨
Cynthia Kop
计算与信息科学研究所，拉德堡德大学，奈梅亨，荷兰

摘要

高阶重写是一种用于编写高阶程序并研究其性质的框架。其中一种性质是终止性：对于所有输入，程序最终都会停止执行并产生输出。人们已经开发了若干工具，用于检查高阶重写系统是否具有终止性。然而，开发此类工具十分困难，也容易出错。本文提出了一种为高阶项重写系统的终止性证明提供认证的方法。我们形式化了一种用于证明终止性的特定方法，即多项式解释方法。此外，我们还给出一个程序，将包含终止性证明高级描述的证明轨迹处理为形式化的 Coq 证明脚本，以便由 Coq 进行检查。我们通过认证 Wanda（一种用于高阶重写的终止性分析工具）生成的高阶多项式解释证明，展示了该方法的实用性。

2012 ACM 分类体系 计算理论 → 逻辑与验证；计算理论 → 等式逻辑与重写

关键词和短语 高阶重写、Coq、终止性、形式化

相关版本 已发表的会议版本： https://drops.dagstuhl.de/opus/volltexte/2023/18405 [32]

补充材料 *Coq* 形式化（*GitHub* 仓库）： nmvdw/Nijn [31]证明脚本生成器（*GitHub* 仓库）： deividrvale/nijn-coq-script-generation [30]

资助 *Niels van der Weide*：本研究由荷兰科学研究组织（NWO）资助的 NWO 项目“等式的力量” OCENW.M20.380 提供支持。
Deivid Vale：作者获得了 NWO Top 项目“通过高阶重写实现的隐式复杂性”（NWO 612.001.803/7571）的支持。</output>
Cynthia Kop：作者获得了 NWO Top 项目“通过高阶重写实现的隐式复杂性”（NWO 612.001.803/7571）以及 NWO VIDI 项目“受约束的高阶重写与程序等价”（NWO VI.Vidi.193.075）的支持。</output>

致谢 作者感谢 Dan Frumin 在理解和使用 Ltac方面提供的帮助。

1 引言

自动证明终止性是项重写中的一个重要问题。为此，人们开发了许多工具，例如 AProVE [10], NaTT [35], MatchBox [33]、Mu-Term [12], SOL [13], TTT_2 [21] 和 Wanda [16], ，它们在每年举办的终止性竞赛 [11]中相互角逐。除了基础的（一阶）项重写外，这些工具还包括用于分析字符串重写、条件重写和高阶重写等问题的工具。

开发终止性工具是一项困难且容易出错的工作。一方面，实现的终止性技术可能存在错误。这在高阶项重写中尤为突出，因为部分应用、类型结构、β-归约，以及不同高阶重写形式体系之间往往无法完美迁移的技术，使得证明通常十分复杂。因此，

即使是已发表的高阶重写论文中也发现了错误，这并不令人意外。另一方面，工具开发者很容易无意中漏掉一项测试，未能检查应用特定终止性技术的条件是否满足；也可能在不同的高阶形式体系之间转换方法时出错。

更令人头疼的是，终止性证明通常复杂且具有很强的技术性，因此很难人工评估终止性证明器输出的正确性。许多基准测试包含数百条规则，而现代终止性工具还会采用数十年来发展出的各种证明方法。例如，单个终止性证明可能会结合使用依赖对 [3, 9, 19], 递归路径序 [5, 20],规则删除，以及多种类型的解释 [8, 18, 23, 34]。这使得查找程序中的错误变得十分困难。

因此，有必要对终止性证明器的输出进行形式化认证，最好能够自动完成。提供此类认证有两种常见的工程策略。第一种策略是将认证器构建为证明助手中的一个库，同时配备工具来读取证明器的输出并构造形式化证明，我们称之为证明脚本。随后由证明助手验证该证明脚本。这种系统设计的例子包括 Cochinelle/CiME3 [7]和 CoLoR/Rainbow [6]。第二种策略是在形式化中纳入经认证算法，用于检查证明器输出的正确性。这样便可以通过代码提取来提取整个认证器，并将其作为独立程序使用。因此，这种方法不需要由独立工具生成证明脚本，但形式化成本更高。IsaFoR/CeTA [28] 采用了这种方法。

然而，在高阶重写方面，可选方案十分有限。Cochinelle [7] 和 IsaFoR/CeTA [28] 都只考虑一阶重写。CoLoR/Rainbow [6]确实包含了对 HORPO [20]早期定义的形式化。由于我们这里使用的项形式体系与 [20], 所用的不同，因此我们的结果无法直接兼容。例如，参见 [2, 25] ，其中介绍了更多关于重写的形式化成果。

本文介绍了一种新的组合 Nijn/ONijn，用于认证高阶重写终止性证明。我们遵循前面提到的第一种系统设计：Nijn 是一个 Coq 库，提供对底层高阶重写理论的形式化；ONijn 是一个证明脚本生成器，它根据对终止性证明的最简描述（我们称之为证明轨迹）生成 Coq 证明脚本。随后，证明脚本利用 Nijn 中的结果，检查所跟踪证明的正确性。下图示意了使用 Nijn/ONijn 生成证明证书的基本步骤。

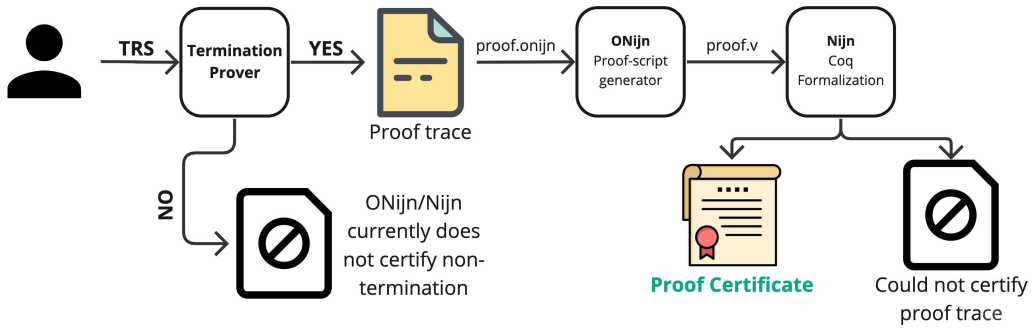

图 1 Nijn/ONijn 示意图

虽然 Nijn 由 Coq 检查，因此是我们工具中经过认证的核心部分，但用 OCaml（ONijn）实现的证明脚本生成目前尚未经过认证，必须予以信任。对于

这一原因，我们有意让 ONijn 尽可能简单，不由其执行任何检查或计算。委托给 ONijn 的唯一任务，是将终止性证明器给出的证明轨迹解析为 Coq 证明脚本。此外，在 Coq 中检查多项式终止性证明的正确性本质上是不完备的，因为这需要一种解决任意多项式上不等式的方法，而这在一般情况下是不可判定的。

贡献。 本文的主要贡献可概括如下：

- 我们对高阶代数函数系统进行了形式化（定义 2.6）；基于弱单调代数，对解释方法进行了形式化证明（定理 3.11）；对高阶多项式方法进行了形式化（定理 4.7）；开发了一种策略，可自动求解使用高阶多项式方法时产生的约束（第 4.3 节）；开发了一个 OCaml 程序，可将终止性证明器的输出转换为表示终止性证明的 Coq 脚本（第 5 节）。

技术概述。 本文围绕 Nijn 展开。Nijn 是一个用于形式化高阶重写的 Coq 库 [31]。该形式化基于内涵依赖类型论，并扩展了两个公理：函数外延性 和恒等证明唯一性 [14]。目前，该库中形式化的终止性判据是高阶多项式方法，该方法首次发表于 [8]。该工具 `coqwc` 统计的代码行数如下：

```
spec    proof comments
5497     1985      272 total
```

高阶解释方法大致如下。首先，以组合方式将类型解释为良序结构（定义 3.3）。例如，我们将基类型解释为自然数（采用通常的序）。然后，我们将函数类型 $A \Rightarrow B$ 解释为从 $(\!|A|\!)$ 到 $(\!|B|\!)$ 的弱单调函数集合，其中 $(\!|A|\!)$、$(\!|B|\!)$ 分别表示 A、B 的解释。第二步是将某一类型的项 A 映射到 $(\!|A|\!)$的元素，这由定义 3.9 表达。

这种解释称为扩展单调代数，仅在 [8], 中并不足以保证终止性。为保证终止性，我们将项应用（定义 4.6）和函数符号都解释为强单调泛函。此外，必须以一种使系统的规则严格定向的方式解释项，即对于所有规则 $\ell \to r$，都有 $[\![\ell]\!] > [\![r]\!]$。这意味着，每当一个项中触发重写时，该项的解释都会严格递减。因此，可以保证终止性。在这里，我们使用终止性模型（定义 3.10）来汇集这些必要条件。

在高阶情形下确立这一技术正确性的主要结果由定理 3.11 表述。对于熟悉一阶重写中解释方法的读者而言，定理 4.7 并不令人意外。它本质上是将 Manna–Ness 判据与高阶多项式相结合，再加上高阶情形所需的额外技术细节。

2 Coq 中高阶重写的基础

本节将介绍形式化代数函数系统 （AFS）所需的基本构造，包括类型、上下文、变量、项和重写规则。本节最后将阐述如何在 Coq 中以构造性的方式表达终止性。

2.1 项与重写规则

我们先来定义简单类型。

▶ **定义 2.1** (ty)。 **简单类型**在类型 B 上定义如下:

```
Inductive ty (B : Type) : Type :=
| Base : B → ty B
| Fun : ty B → ty B → ty B.
```

B 的元素称为**基类型**。每个元素 b: B 都会导出一个简单类型Base b ; 若 A1、 A2 是简单类型，那么 Fun A1 A2也同样是简单类型。我们用 A1 ⟶ A2 表示 Fun A1 A2。

为了给可能含有自由变量的项定类型，我们需要（变量）上下文。从概念上说，上下文是由变量及其相应类型构成的列表。例如，$[x_0 : A_0; \ldots; x_n : A_n]$ 是变量 x_0 的类型为 A_0、.....、 x_n 的类型为 A_n的上下文。不过，在我们的论述中使用无名变量，因此不记录变量名。因此，上下文用类型列表表示。于是，我们只考虑列表$[A_0, \ldots, A_n]$。然而，我们仍需要在项中引用自由变量。为此，我们用上下文中的索引位置来表示它们。例如，在上下文 $[A_0; \ldots; A_n]$ 中， $n+1$ 个位置索引为 0、1、.....、 n，我们将它们用作变量。

▶ **定义 2.2** (con)。 类型 B 上的**变量上下文**定义如下 .

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

我们用 • 表示 Empty , 用 A , C 表示 Extend A C。

▶ **定义 2.3** (var)。 我们将上下文 C 中类型为 A 的**变量**的类型 var C A 定义为

```
Inductive var {B : Type}: con B → ty B → Type : =
```

我们来看一个关于上下文和一些变量的例子。假设我们有一个记作 b 的基类型。于是可以构造上下文 Base b, Base b ⟶ Base b, Empty。在这个上下文中，有两个变量。第一个变量 Vz 的类型是 Base b，第二个变量 Vs Vz 的类型是 Base b ⟶ Base b。我们讨论的上下文对应于 $[x_0 : b; x_1 : b \longrightarrow b]$。变量 Vz 表示 x_0，而 Vs Vz 表示 x_1。

在下面的定义 2.4 中，我们定义上下文中的良类型项这一概念，它由那些存在类型推导的表达式组成。我们使用依赖类型来确保此类表达式的良类型性。项的类型依赖于一个简单类型 A : ty B（表示该表达式的对象层类型）和上下文 C : con B , 其中记录了项中所有自由变量的类型。我们还需要为函数符号指定类型。因此，我们要求有一个类型 F : Type 函数符号的类型，以及 ar : F → ty B，它将 f : F 映射到一个简单类型 ar f。

▶ **定义 2.4** (tm)。 我们将**良类型项**的类型定义如下

```
Inductive tm {B : Type} {F : Type} (ar : F → ty B) (C : con B) : ty B → Type :=
```

对于每个函数 `f : F`，我们都有一个类型为 `ar f` 的项 `BaseTm f`。每个变量 `v` 都会产生一个项 `TmVar v`。对于 λ-抽象，给定一个项 `s : tm ar(A1, C) A2`，就有一个项 `λ s : tm ar C (A1 ⟶ A2)`，即 `Lam s`。最后一个构造子表示项应用。如果我们有一个项 `s : tm ar C (A1 ⟶ A2)` 和一个项 `t : tm ar C A1`，就得到一个项`s · t : tm ar C A2`，其定义为 `App s t`。

虽然使用 de Bruijn 指数书写项可能更繁琐，但确实有若干优点。最重要的是，它消除了对 α-等价的需求，因此判断项是否相等只需进行简单的语法检查。

我们的重写规则概念与 [8] 中的表述略有不同。主要区别在于，我们既不对规则的左侧施加模式限制，也不要求右侧的自由变量必须出现在左侧。这一选择简化了形式化工作，因为定义具体的项重写系统时，无需检查这一特定条件。请注意，IsaFoR [28] 中也采用了相同的简化

▶ **定义 2.5** (`rewriteRule`)。 **重写规则** 的类型定义如下：

```
Record rewriteRule {B : Type} {F : Type} (ar :
```

上下文 `vars_of` 承载规则中使用的变量，而类型 `tar_of` 用于保证 `lhs_of` 和 `rhs_of` 是相同类型的项。

▶ **定义 2.6** (`afs`)。 **代数函数系统** 的类型定义如下

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

通常，每个 AFS 都会在项集合上诱导出一个重写关系，我们将其记作 `~>`。形式化定义见 `RewritingSystem.v`。重写关系 `~>` 定义为一步关系在传递性以及与项构造子相容性下的闭包。在 Coq 中，我们使用归纳类型来定义这一关系。每个重写步都由一个构造子表示。具体来说，我们有一个构造子用于重写应用的左侧和右侧，有一个构造子用于 β-归约，还有一个构造子用于表示 AFS 的重写规则。

▶ **例 2.7** (`map_afs`)。 我们在 Coq 中编码 $\mathcal{R}_{\text{map}}$。它由两条规则组成：map F空 →空 和 map F(cons $x\,xs$) →cons ($F\,x$) (map $F\,xs$)。我们从基类型开始。

```
Inductive base_types := TBtype | TList.
```

缩写 `Btype` 和 `List` 旨在让基类型的使用更加简便。该系统中有三个函数符号：

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

元数函数 `map_ar` 将 `fun_symbols` 中的每个函数符号映射到其类型。

```
Definition map_ar f :  ty base_types:
```

因此， TNil 是一个列表，给定 Btype 和 List的一个元素，函数符号 TCons 会给出一个 List。同样，我们引入一些缩写，以简化函数符号的使用。

```
Definition Nil {C}:  tm map_ar C _ := BaseTm TNil. Definition Cons
```

第一条规则 map F nil $\to$ nil 编码为以下 Coq 结构：

```
Program Definition map_nil : =
```

请注意，我们只定义了 make_rewrite前两个参数的模式，而上下文（_ ， •）中的类型以及规则的类型都未指定。只要提供长度正确的上下文模式，Coq 就能自动填补这些空缺。在这条重写规则中，只有一个自由变量。因此，变量 TmVar Vz 指的是上下文中唯一的变量。此外，我们使用迭代的 let-语句来模拟变量名。对于上下文中的每个位置，我们都在 Coq 中引入一个变量，并在规则的左侧和右侧使用它。这样能让规则更易于人类阅读。事实上，这条规则的左侧 map Fnil 在代码中表示为 Map · f · Nil 。 map 的第二条规则也按照相同思路编码。

```
Program Definition map_cons :=

  (_,  _ ,  _,  •) _ (
```

综上，我们得到一个 AFS，称之为 map_afs。

```
Definition map_afs := make_afs map_ar(map_nil :: map_cons :: nil)】【。
```

2.2 终止性

强正规化通常定义为不存在无限重写序列。在排中律成立的经典环境中，这样的定义已经足够。然而，我们处于构造性环境中，因此关注的是更强的定义。因此，我们需要一个以肯定形式表述的构造性谓词，它能推出不存在无限重写序列。以下定义体现了这一思想

▶ **定义 2.8** (WellfoundedRelation.v)。 关系 R 的**良基性谓词**定义如下

```
Inductive isWf {X :  Type} (R :  X → X → Type) (      =
```

如果对每个元素都成立良基性谓词，则一个关系是**良基的**。

```
Definition Wf {X :  Type} (R :  X → X → Type) :
```

请注意，这一定义此前已被多次考虑过，例如在 [4]以及 CoLoR [6]中。如果对于所有满足 `R x y` 的 `y`， `x` 都是良基的，则元素 `x` 是良基的。请注意，如果不存在满足 `R x y`的 `y`，那么 `x` 就是真空地良基的。从重写的角度来看，这一定义恰当地刻画了强正规化的概念。事实上，当且仅当对于每个满足 s 且重写到 s' 的 s'，项 s 都是强正规化的时，项 s 才是强正规化的。

良基性与无限重写序列的存在相矛盾，即使是在构造性语境中也是如此。因此，它确实给出了更强的条件。

▶**命题 2.9** （no_infinite_chain）。 **</style}如果 是良基的，那么不存在无限序列 、、……，使得对于所有 ，都有** （s_n、 s_{n+1}） *</style}*。

接下来，我们用良基谓词来定义强正规化。

▶ **定义 2.10** (is_SN)。 如果对于每个上下文 `C` 和每种类型 `A`，上下文 `C` 中类型为 `A` 的项之间的重写关系都是良基的，那么代数函数系统就是**强正规化的**。我们将其形式化如下：

```
Definition isSN {B F : Type} (X : afs B F) :  Prop :
```

3 高阶解释方法

本节中，我们将形式化用于代数函数系统的弱单调代数方法。我们将为上一节引入的句法构造提供类型论语义，并给出一个充分条件，使得这类语义可用于证明强正规化。

3.1 解释类型和项

在弱单调代数中，类型被解释为集合，并带有一个良基序和一个拟序 [8, 24]。因此，我们首先定义兼容关系。直观地说，这些关系构成了语义的定义域。

▶**定义 3.1** (CompatibleRelation.v)。**兼容关系**定义如下

```
Record CompatRel: = {carrier: > Type ;
```

我们分别用 `x > y` 和 `x >= y` 表示 `gt x y` 和 `ge x y`。

记录 `CompatRel` 包含表达 `>` 与 `>=` 之间相容性所需的数据。它需要满足的条件定义在下方的类型类 `isCompatRel` 中。

```
Class isCompatRel (X : CompatRel) :={gt_trans : forall {
```

注意， gt_trans 在 isCompatRel 中的性质可由 compat 和 ge_gt推出。带有通常序的自然数类型 nat是满足 isCompatRel的数据的第一个例子。我们将其记为 nat_CompatRel。这个类型类实质上刻画了 [18]中引入的扩展良基集的概念。**扩展良基集**是一个集合，配有相容的序 $>$， $\geq$ ，其中 $>$ 是良基的，且 $\geq$ 是拟序。这一相容性要求对应于类型类 isCompatRel中的公理 compat 。不过，由于我们在此定义中并不要求 $>$ 是良基的，因此我们改称其为兼容关系。更具体地说，如果 X 的类型为 CompatRel，且满足类型类 isCompatRel中的约束，那么 X 就是一个兼容关系。

为了对简单类型（定义 2.1）进行解释，我们先固定一个基类型 B : Type 以及一个解释 semB : B → CompatRel ，使得每个 semB b 都是一个兼容关系。每当 semB 满足这一性质时，我们就称其为 B的**解释键**。我们将箭头类型解释为函数型兼容关系，即其载体中的项都是函数的兼容关系。我们感兴趣的函数类是弱单调映射。

▶ **定义 3.2** (MonotonicMaps.v)。**弱单调映射**定义如下

```
Class weakMonotone {X Y : CompatRel} (f : X ⇝ Y) :
```

```
Record weakMonotoneMap (X Y: CompatRel): =
```

类 weakMonotone 说明了函数何时是弱单调的，而记录 weakMonotoneMap 的一个元素由一个函数及其弱单调性的证明组成。接着，我们定义 fun_CompatRel ，其类型为 CompatRel ，表示从 X 到 Y的**函数型兼容关系**。定义如下：

```
Definition fun_CompatRel (X Y : CompatRel) : CompatRel : =
```

下文中，我们用 X →wm Y 表示 fun_CompatRel X Y。类型的语义由解释键 semB参数化。其定义如下：

▶ **定义 3.3** (sem_Ty)。假设 A : ty B ，且 semB 是 B的解释键。那么

```
Fixpoint sem_Ty (A : ty B) : CompatRel :
```

我们还展示了如何解释上下文。为此，我们需要解释空上下文和上下文扩展。针对这两者，我们定义兼容关系的单位和积。

▶ **定义 3.4** (Examples.v)。 **单位** 和**积** 兼容关系：

```
Definition unit_CompatRel: D
```

注意， unit_CompatRel 是仅含一个元素的类型上的兼容关系，其序是平凡的。此外， prod_CompatRel 是积上的兼容关系，我们对元素进行逐坐标比较。我们将 X ∗ Y 记作 prod_CompatRel X Y。

▶ **定义 3.5** (sem_Con)。 上下文的解释如下

```
Fixpoint sem_Con(C : con B) : CompatRel :
```

接下来，我们为变量和项赋予语义。我们这里采用的方法与高阶重写中通常采用的方法略有不同。例如在 [8, 18, 24]，中，上下文信息被提升到元层次，而变量则通过赋值的概念来解释。相比之下，在我们的设定中，类型上下文处于语法层次，变量被解释为弱单调函数。因此，对于每个项t : tm C A，我们都赋予一个从 sem_Con C 到 sem_Ty A的映射。接下来，我们需要以下弱单调函数。

▶ **定义 3.6** (Examples.v)。 我们定义以下弱单调函数。

- 给定 y : Y，我们用 const_wm y: X →wm y 表示常值函数。给定 f : X →wm Y 和 g : Y →wm Z，我们将 g ∘wm f : X →wm Z 定义为它们的复合。第一投影为 fst_wm: X ∗ Y →wm X，它将有序对 (x , y) 映射为 x；第二投影为 snd_wm: X ∗ Y →wm Y，它将 (x , y) 映射为 y。给定 f : X →wm Y 和 g : X →wm Z，我们有函数 〈 f , g 〉 : X →wm (Y ∗ Z)。对于x : X，我们将 〈 f , g 〉 x 定义为 (f x , g x)。给定 f : Y ∗ X →wm Z，我们得到 λwm f : X →wm(Y →wm Z)。对于任意 x : X 和 y : Y，我们将 λwm f y x 定义为 f (y , x)。给定 f : X →wm(Y →wm Z) 和 x : X →wm Y，我们得到 f ·wm x : X →wm Z，它将每个 a : X映射为 f a (x a)。给定 x : X，我们有一个弱单调函数 apply_el_wm x : (X →wm Y) →wm Y , 它将 f : X →wm Y 映射为 f x。

回顾一下，变量由上下文中的位置表示，而上下文又被解释为弱单调积（定义 3.5）。这样，我们就可以将上下文中某个位置上的变量解释为该位置对应类型的解释。

▶ **定义 3.7** (sem_Var)。 我们使用以下函数解释变量

```
Fixpoint sem_Var {C : con B} A : ty B} (v : var C A) :
```

为了给项赋予语义，我们需要以下数据：一个元数函数ar : F → ty B，以及它的解释 semF : forall (f : F), sem_Ty (ar f)，还有一个应用算子，定义如下

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

用于解释项应用。

▶ **注 3.8.** 最初有一种看似合理但不正确的猜想：将 f : sem_Ty A1 →wm sem_Ty A2 应用于 x : sem_Ty A1解释为 f x。然而，这种解释有一个明显的缺点。归根结底，我们希望从解释中推出强正规化，而主要思路是：如果有一个重写x ∼> x'，那么就有 semTm x > semTm x'。如果把项的应用解释为函数的实际应用，就无法满足这一要求。确实，如果有x < x'，并不能保证也有 f x < f x'，因为 f 只是弱单调的。

处理这一问题有两种方法。一种是将函数类型解释为严格单调映射 [18]。采用这种方法时，对应用的这种解释是有效的。然而，这需要付出代价，因为 λ 抽象的解释会变得更加困难。

另一种方法（本文采用的方法）也见于 [8]。我们在解释方法中加入一个参数，即 semApp，用它抽象地表示应用的解释。为了在这种设定下推导出强正规化，我们在第 3.2 节中加入了关于semApp 的要求。因此，在具体实例化这种方法时，我们需要为 semApp 提供一个实际定义。我们将在第 4.2 节中看到这一点。

▶ **定义 3.9** (sem_Tm)。给定一个函数 semF : forall (f : F), sem_Ty (ar f)，项的语义n-由以下内容给出

```
Fixpoint sem_Tm {C : con B} {A : ty B} (t : tm ar C A) :
```

请注意，我们本可以选择一种固定的应用解释方式。我们在形式化中沿用 Fuhs 和 Kop [8]所采用的方法，并将 semApp 保持为抽象形式。如果要将解释方法同时用于规则删除和依赖对 方法，这一选择至关重要。更多详情请参见 [15, 第 4 章及 6] 。

3.2 AFS 的终止性模型

现在，我们已经为定义本节的核心概念做好了所有必要准备：终止性模型。从代数函数系统的一个终止模型中，可以得到

类型和项的一种解释。此外，每条重写规则在这种解释下都得到“满足”。

▶ **定义 3.10** (Interpretation)。 设 $\mathcal{R}$ 是一个基类型为 B 、函数符号为 F 的代数函数系统。**R** 的终止模型 由以下部分组成：

- 一个解释键 semB；一个函数 semF： forall
- (f : F), sem_Ty (ar f)；一个函数
-

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

满足以下公理

- 每个 semB b 都是良基且非空的；若 f > f’，则 semApp _ _ (f ,
- x) > semApp _ _ (f’ , x)；若 x > x’，则 semApp _ _ (f , x)
- > semApp _ _ (f , x’)；对于所有 f 和 x，有 semApp _ _ (f , x)
- >= f x ；对于每条重写规则 r 、替换 s 和元素 x，有
-

```
semTm (lhs r [ s ]) x > semTm (rhs r [ s ]) x.
```

虽然每条重写规则的左侧都大于右侧，但在我们的解释中，这一点对 β-归约并不成立。由于重写序列可能同时包含重写规则和 β-归约，因此无法保证这类序列严格递减。因此，仅凭这一点还不足以得出强正规化的结论，我们采用 Kop [15] 使用的方法。具体而言，Kop 使用 规则删除来证明：若 β-归约具有强正规化性质，则强正规化也随之成立；这是 Tait [27] 证明的一个著名定理。 β-归约的强正规化已被多次形式化，相关概述见 [1]。现在我们推导本节的主要定理。

▶**定理 3.11** (afs_is_SN_from_Interpretation)。 设 R 为一个代数函数系统。若我们有 X 的终止模型，则 X 具有强正规化性质。

4 高阶多项式方法

4.1 多项式

本节将第 3 节的内容具体化到一个实例，即 多项式方法 [8]。为此，我们定义高阶多项式的记号。

▶ **定义 4.1** (Polynomial.v)。 我们通过相互归纳定义 base_poly 的**基础多项式** 类型以及 poly 的**高阶多项式** 类型如下：

```
Inductive base_poly{B : Type}:                    with poly{B : Type} : con B → ty B → Type :=
```

我们可以使用 P_const （常量）、 P_plus （加法）和 P_mult （乘法）构造基多项式表达式。此外， from_poly 接受 poly C(Base b)中的一个元素，并返回上下文 C中的一个基多项式。使用 P_base，我们可以将基多项式转换为任意基类型的多项式。这些构造子 P_var、P_app 和 P_lam让人想起简单类型 λ 演算。我们从 P_var获得变量，从 P_lam获得 λ-抽象，并从 P_app获得应用。请注意，结合 from_poly 和 P_var，我们可以在基多项式中使用变量。

我们来谈谈所做的设计选择，以及这些选择如何影响多项式的定义。我们的要求之一是能够对不同基类型上的多项式进行加法和乘法。在实际例子中经常会用到这一点，例如例 4.2。函数符号的参数可能来自不同的基类型，我们希望在多项式表达式中同时使用这两种类型。

一种可能的做法是只使用类型 poly ，并添加一个构造子 r

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

然而，我们没有这样做：如果使用 P_const，那么若不明确说明基类型，elaborator 就无法确定实际类型。相反，我们使用了一种不依赖于实际基类型的基多项式类型。这正是 base_poly 的作用，它只取决于所使用的变量。我们可以自由地对 base_poly 的项进行加法和乘法；如果使用常数，则无需明确提及基类型。此外，我们还能在 base_poly 和 poly C（Base b） 之间转换，而 P_base 和 from_poly 使我们能够做到这一点。

请注意，我们对高阶多项式的定义与 Fuhs 和 Kop 给出的定义颇为相似 [8, 定义 4.1]。他们定义了一个集合 Pol(X)，其中包含多项式表达式；对于每个类型 A ，还定义了一个集合 Pol^A(X)。集合 Pol^A(X) 通过递归定义：对于基类型，它是 X 上的多项式集合；对于函数类型 $A_1 \longrightarrow A_2$，它由表达式 $\Lambda(y : A_1).P$ 构成，其中 P 是一个类型为 A_2 的多项式，并使用了一个额外变量 $y : A_1$。我们的 base_poly C 和 poly C A 分别对应 Pol(X) 和 Pol^A(X)。不过，两者之间存在一些差异。首先，Fuhs 和 Kop 要求变量必须完全应用，而我们允许变量部分应用。其次，Fuhs 和 Kop 对多项式的定义使得对于任意两个基类型 b_1， b_2 ，类型 Pol^{b_1}(X) 和 Pol^{b_2}(X) 相等。我们的定义则并非如此：我们改用构造子 from_poly 和 P_base 来关联 base_poly C 与 poly C(Base b)。

在多项式方法中，解释键将每个基类型映射到 nat_CompatRel；下文中，我们分别用 ⟦ C ⟧con 和 ⟦ A ⟧ty 表示上下文和类型的解释。注意，每个多项式 P : poly C A 都会导出一个弱单调函数 sem_poly P: ⟦ C ⟧con →wm ⟦ A ⟧ty ，而每个基多项式 P : base_poly C都会导出 sem_base_poly P : ⟦C ⟧con →wm nat_CompatRel。这两个函数通过互递归定义，并且每个构造子都按预期方式进行解释： sem_poly。

为了实际使用 base_poly C 和 poly C A，我们为多项式运算提供了便捷的记法。具体而言，我们定义记法 +、 ∗和 ·P ，分别表示加法、乘法和应用。由于我们需要能够对不同类型的多项式进行加法，因此这些运算必须重载。为此，我们同样使用 MathClasses [26] 中的类型类。详情请参阅形式化内容。

▶ **例 4.2** (map_fun_poly)。 我们继续讨论例 2.7，并为该系统 map_afs 提供如下多项式解释：

```
Definition map_fun_poly fn_symbols :  poly •(arity trs fn_symbols): =
```

```
match fn_symbols with| Tnil ⇒ to_Poly (P_const 3)
```

非正式地说， `nil` 的解释是常数 3。 `cons` 的解释是将 $y_1 : \mathbb{N}$ 映射到 $3 + 2y_1$ 的函数，而 `map` 被解释为将 $y_0 : \mathbb{N}$和 $G_1 : \mathbb{N} \to_{\mathrm{wm}} \mathbb{N}$ 映射到 $3y_0 + 3y_0 G_1(y_0)$ 的函数。

4.2 多项式解释

利用多项式，我们可以在某些条件下使用定理 3.11 推导出强正规化。假设对于所有函数符号 `f` ，都有一个多项式 `J` ： `poly • (arity X f)`，现在我们需要给应用提供解释。遵循 Fuhs 和 Kop [8],的方法，我们使用一种通用方法来解释应用。我们首先在每种类型的解释中构造一个极小元素。

▶ **定义 4.3** (`min_el_ty`)。 对于每个简单类型 `A` ，我们定义⟦ `A` ⟧`ty` 的一个极小元素，如下所示

```
Fixpoint min_el_ty(A :  ty B) :
```

此处 `nat_minimal_element` 定义为 0，而 `min_el_fun_space(min_el_ty A2)` 是`(min_el_ty A2)` 上的常值函数。

为了定义应用的语义，我们需要若干涉及⟦ `A` ⟧`ty`的操作。首先，我们考虑下值函数。

▶ **定义 4.4** (`lvf`)。我们定义**下值函数**如下

```
Fixpoint lvf {A :  ty B} :  ⟦ A ⟧ty →wm nat_CompatRel:
```

请注意，我们直接将 `lvf` 构造为弱单调函数。此外，我们重用了定义 3.6 中定义的组合子。因此，无需另外证明该函数是单调的。

在 Kop 和 Fuhs [8], 的表述中，这个定义的写法有所不同，但等价。他们没有递归地定义 lvf_A ，而是考察完全应用；在我们的设定中，这样做会更加复杂。更具体地说，由于我们使用的是简单类型，必须满足 $A = A_1 \longrightarrow \ldots \longrightarrow A_n \longrightarrow B$。接着，他们定义 $\mathrm{lvf}_A(f) := f(\bot_{A_1}, \ldots\ldots, \bot_{A_n})$，其中 $\bot_A$ 是 A的解释中的最小元素。接下来，我们在 ⟦ `A` ⟧`ty` 上定义两种加法运算。

▶ **定义 4.5** (`plus_ty_nat`)。 自然数和 ⟦ `A` ⟧`ty` 上元素的加法定义如下

```
Fixpoint plus_ty_nat {A : ty B} : ⟦ A ⟧ty * nat_CompatRel →wm ⟦A ⟧ty:
```

函数 plus_ty_nat 允许我们将任意自然数加到类型解释中的元素上。请注意，定义 4.5 中有两种情况。首先，类型 A可能是基类型。在这种情况下，我们将两个自然数相加，并使用通常的加法运算。在第二种情况下，我们处理的是函数类型 A1 ⟶ A2。所得函数使用与相关自然数进行逐点加法来定义。现在，一切就绪，可以定义应用的解释了。

▶ **定义 4.6** (p_app)。 应用被解释为以下函数

```
Definition p_app {A1 A2:   ty B}:
```

如果 A1 和 A2 都是基类型，那么 p_app(f , x) 会化简为 f x + x。注意， p_app满足定理 3.11 的要求。因此，我们得到如下结果。

▶**定理 4.7** (poly_Interpretation)。 设 R 为一个 *AFS*。假设对于每个函数符号 f ，都存在一个多项式 p_fun_sym f ，使得对于 R 中的所有重写规则 l ∼> r ，对所有 x，均有 semTm l x > semTm r x 。那么 R 有一个终止模型。

4.3 约束求解策略

请注意，为了使用定理 4.7 正式验证一个系统的终止性证明，我们需要提供一个多项式解释，并证明对于所有规则 $\ell \to r$， $[\![\ell]\!] > [\![r]\!]$ 都成立。这会在 Coq 上下文中引入必须解决的不等式证明目标。

▶ **例 4.8**。 我们来看一个具体例子。我们使用例 4.2 中给出的多项式来证明例 2.7 的强正规化。这个例子引入了两个不等式，每条规则对应一个。令 $G_0 : \mathbb{N} \to_{\mathrm{wm}} \mathbb{N}$ 是弱单调的。对于规则 map_nil，我们需要证明对于所有 G_0，约束 $12 + G_0(0) + 9G_0(3) > 3$ 成立。对于第二条规则 map_cons，约束为：对于所有 y_0， $y_1 \in \mathbb{N}$ 和 G_0，都有 $12 + 4y_0 + 12y_1 + G_0(0) + (3y_0 + 9y_1 + 9)G_0(3 + y_0 + 3y_1) > 3 + y_0 + 12y_1 + 3G_0(0) + G_0(y_0) + 9y_1G_0(y_1)$。

为这类不等式寻找见证元十分繁琐，因此我们希望将这一任务自动化。为此，我们开发了一种策略（solve_poly），能够自动求解由定理 4.7 导出的不等式。该策略本质上是试图模拟人们在纸笔证明中求解这些目标的方式，而 Wanda 也采用了相同的方法。

▶ **例 4.9.** 我们将说明如何求解由例 4.8 中提到的 map_cons 产生的约束。第一步是找出不等式两边的匹配项

并将它们相减。在本例中， $3 + y_0 + 12y_1 + G_0(0)$ 出现在两边，相减后得到以下约束：

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

第二步是结合针对高阶变量 G_0 的论证来利用其单调性。注意，0、 y_0 和 y_1 均小于或等于 $3 + y_0 + 3y_1$，因为它们都是自然数。由于 G_0 弱单调，我们得到

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

现在我们可以将原始约束化简为

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

由于 $3y_0 + 9y_1 + 9 \geq 9y_1 + 3$，我们有

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

这足以得出结论： `map_cons` 的约束均已满足。

该策略 `solve_poly` (`solve_poly`) 遵循上述步骤。请注意，我们使用策略 `nia`，这是 Coq 中的一种策略，可以求解非线性整数算术中的不等式和方程。具体而言， `solve_poly` 的工作方式如下：

- 首先，我们为每条重写规则生成一个目标，并展开假设，使上下文中的每个变量要么是自然数，要么是函数。对于每个 f类型为函数类型的变量，我们寻找有序对$(x,\ y)$，使得 $f(x)$ 出现在左侧，而 $f(y)$ 出现在右侧。我们尝试使用 `nia` 判断能否根据假设证明 $x < y$ 。如果可以，我们就将 $x < y$ 加入假设中；否则，继续尝试。然后使用 `nia`求解添加额外假设后的目标。

请注意， `solve_poly` 并不完备，因为 `nia` 是不完备的。因此，如果 Coq 接受了使用该策略的证明，那么该证明就是正确的。然而，如果证明未被接受，并不代表该证明一定是错误的。借助本节讨论的内容，我们可以写出例 4.2 中给出的多项式，而该策略能够验证强正规化。

5 生成证明脚本

本节讨论验证框架的实际应用。原则上，可以手动将重写系统编码为 Coq 文件，并使用我们提供的形式化来验证其终止性证明。不过，这样做十分繁琐。事实上，在例 2.7 中，我们使用了缩写，使$\mathcal{R}_{\mathsf{map}}$ 的形式化描述更易读。规则更多的重写系统很难手动编码。此外，要正式证明终止性，我们还需要对证明进行编码。我们在例 4.2 中做到了这一点。$\mathcal{R}_{\mathsf{map}}$ 及其终止性证明的完整形式化编码见文件 `Map.v`。

5.1 多项式解释的证明轨迹

手动编码的困难促使人们采用证明轨迹。证明轨迹是一种便于人类阅读的编码，其中包含一个项重写系统，以及重构

终止性证明所需的关键信息，并可将其重构为 Coq 脚本。让我们再次以$\mathcal{R}_{\text{map}}$ 为例。该系统的证明轨迹以 YES 开头，表示我们要为该系统提供终止性证明。接下来是一个列表，其中编码了该系统的签名和规则。

```
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)
]
```

请注意，规则中的自由变量无需声明，也无需提供其类型信息。Coq 可以自动推断这些信息。证明轨迹的最后一部分描述了签名中每个函数符号的解释。

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

利用上述证明轨迹中提供的信息，我们可以完整地重构对$\mathcal{R}_{\text{map}}$的终止性进行形式化证明，该证明使用了定理 4.7。证明轨迹的完整说明见 [29], ONijn 的应用程序接口。证明轨迹不是 Coq 文件，因此我们需要进一步将其编译为合适的 Coq 脚本。图 1 中的示意图展示了所需的步骤。我们使用 ONijn 将证明轨迹编译为 Coq 脚本。调用方式如下：

```
onijn path/to/proof/trace.onijn -o path/to/proof/script.v
```

此处，第一个参数是证明轨迹文件的路径，而 `-o` 选项要求提供生成的 Coq 脚本的文件路径。生成的 Coq 脚本可由 Nijn 按如下方式验证：

```
coqc path/to/proof/script.v
```

如何在本地安装 ONijn/Nijn 的说明可在 [29] 找到。

5.2 验证 Wanda 的多项式解释

值得注意的是，在我们的认证框架中，终止性证明器是抽象的。这意味着我们不受特定终止性工具的限制。因此，我们可以验证任何实现了此处所述解释方法并能够输出 ONijn 格式证明轨迹的终止性工具。

由于Wanda [16] 是一个实现了 [8],中所述解释方法的终止性工具，因此它是我们的首个验证候选对象。我们为Wanda 添加了运行时参数 `--formal`，使其能够输出ONijn 格式的证明轨迹。在 [16] 中可以找到如何调用Wanda的详细说明。例如，下面演示如何在Wanda 上运行映射AFS。

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

该设置 `-d rem` 会将 Wanda 设为禁用规则删除。该选项 `--formal` 会将 Wanda 设为仅使用多项式解释，并将证明输出为ONijn 证明轨迹。使用这些选项运行 Wanda，即可得到我们用于上述$\mathcal{R}_{\text{map}}$ 的证明轨迹。包含此参数的Wanda最新版本可在 [17]找到。

下表描述了我们使用上述设置验证 Wanda 输出结果的实验评估。基准测试集由 Wanda 输出的 46 个项重写系统组成YES ，验证过程中仅使用多项式解释，不进行规则删除。每个系统的认证时间限制为 60 秒。

实验在一台配备 M1 Pro 2021 处理器和 16GB 内存的机器上运行。认证期间，Nijn 的内存使用量介于 400MB 到 750MB 之间。我们在 https://github.com/deividrvale/nijn-coq-script-generation 提供了实验基准测试。

	Wanda			Nijn/ONijn		
技术	# 是	Pct.	平均时间	# 已认证	百分比	平均时间
多项式，不进行规则删除	46	23%	0.07秒	46	100%	4.06秒

表 1 实验结果

因此，我们可以仅使用多项式解释，为 Wanda 证明为强正规化的所有终止重写系统提供证明。 .

6 结论与未来工作

我们对高阶重写中的多项式方法进行了形式化。这不仅涵盖了代数函数系统等基本概念，还包括解释方法以及将该方法实例化为多项式。此外，我们还展示了如何根据终止性证明器的输出生成 Coq 脚本。这使我们能够验证其输出，并构造强正规化的形式证明。我们还将这些工具应用于一个具体实例，即检查 Wanda 的输出。

这项工作有许多可扩展之处。首先，可以对更多高阶重写技术进行形式化，例如元组解释 [18] 和依赖对 [19,22]。也可以将 HORPO 集成到我们的框架中 [20]。其次，在当前的形式化中，每次使用多项式方法时，应用的解释都是固定的。也可以为用户提供自行选择解释的选项。第三，目前我们的工作只集成了 Wanda。还可以进一步扩展，使其能够直接集成其他工具。

参考文献

1 Andreas Abel、Guillaume Allais、Aliya Hameer、Brigitte Pientka、Alberto Momigliano、Steven Schäfer 和 Kathrin Stark。POPLMark 重制版：通过逻辑关系机械化证明。《函数式编程杂志》，29:e19，2019。 doi:10.1017/S0956796819000170。
2 Ariane Alves Almeida 和 Mauricio Ayala-Rincón。形式化最内层终止性的依赖对判据。《科学计算程序》，195:102474，2020。 doi:10.1016/j.scico.2020.102474。
3 Thomas Arts 和 Jürgen Giesl。使用依赖对实现项重写的终止性。《理论计算机科学》，236(1-2):133–178，2000。 doi:10.1016/S0304-3975(99)00207-8。4 Yves Bertot 和 Pierre Castéran。交互式定理证明与程序开发——*Coq'Art*：归纳构造演算。《理论计算机科学文献》。EATCS 丛书。Springer，2004。 doi:10.1007/978-3-662-07964-5。

18 参考文献

5 Frédéric Blanqui、Jean-Pierre Jouannaud 和 Albert Rubio。可计算性路径序。《逻辑方法与计算机科学》，11(4)，2015。 doi:10.2168/LMCS-11(4:3)2015。6 Frédéric Blanqui 和 Adam Koprowski。CoLoR：关于良基重写关系及其在终止性证书自动验证中的应用的 Coq 库。《数学结构与计算机科学》，21(4):827–859，2011。 doi:10.1017/S0960129511000120。7 Evelyne Contejean、Pierre Courtieu、Julien Forest、Olivier Pons 和 Xavier Urbain。使用 cime3 的自动化认证证明。载于 Manfred Schmidt-Schauß（编），《第 22 届国际重写技术与应用会议论文集》，RTA 2011，2011 年 5 月 30 日至 6 月 1 日，诺维萨德，塞尔维亚，第 10 卷，LIPIcs，第 21–30 页。Schloss Dagstuhl - Leibniz-Zentrum für Informatik，2011。 doi:10.4230/LIPIcs.RTA.2011.21。8 Carsten Fuhs 和 Cynthia Kop。高阶重写的多项式解释。载于 Ashish Tiwari（编），《第 23 届国际重写技术与应用会议（RTA'12）》，RTA 2012，2012 年 5 月 28 日至 6 月 2 日，名古屋，日本，第 15 卷，LIPIcs，第 176–192 页。Schloss Dagstuhl - Leibniz-Zentrum für Informatik，2012。 doi:10.4230/LIPIcs.RTA.2012.176。9 Carsten Fuhs 和 Cynthia Kop。静态高阶依赖对框架。载于 Luís Caires（编），《编程语言与系统——第 28 届欧洲编程研讨会》，ESOP 2019，作为欧洲软件理论与实践联合会议的一部分举行，ETAPS 2019，布拉格，捷克共和国，2019 年 4 月 6–11 日，论文集，第 11423 卷，《计算机科学讲义》，第 752–782 页。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 和 René Thiemann。使用 aprove 自动分析程序终止性与复杂度。《自动推理杂志》，58(1):3–31，2017。 doi:10.1007/s10817-016-9388-y。11 Jürgen Giesl、Albert Rubio、Christian Sternagel、Johannes Waldmann 和 Akihisa Yamada。终止性与复杂度竞赛。载于 Dirk Beyer、Marieke Huisman、Fabrice Kordon 和 Bernhard Steffen（编），《系统构造与分析工具和算法》，第 156–166 页，Cham，2019。Springer International Publishing。 doi:10.1007/978-3-030-17502-3_10。12 Raúl Gutiérrez 和 Salvador Lucas。mu-term：自动验证终止性（系统描述）。载于 Nicolas Peltier 和 Viorica Sofronie-Stokkermans（编），《自动推理——第 10 届国际联合会议》，IJCAR 2020，巴黎，法国，2020 年 7 月 1–4 日，论文集，第二部分，第 12167 卷，《计算机科学讲义》，第 436–447 页。Springer，2020。 doi:10.1007/978-3-030-51054-1_28。13 Makoto Hamana。二阶重写的理论与实践：基础、演进与 SOL。载于 Keisuke Nakano 和 Konstantinos Sagonas（编），《函数式与逻辑编程——第 15 届国际研讨会》，FLOPS 2020，秋田，日本，2020 年 9 月 14–16 日，论文集，第 12073 卷，《计算机科学讲义》，第 3–9 页。Springer，2020。 doi:10.1007/978-3-030-59025-3_1。14 Martin Hofmann 和 Thomas Streicher。群胚模型否证恒等证明唯一性。载于《第九届计算机科学逻辑年会论文集》（LICS '94），巴黎，法国，1994 年 7 月 4–7 日，第 208–212 页。IEEE计算机学会，1994。 doi:10.1109/LICS.1994.316071。15 Cynthia Kop。高阶终止性：证明高阶项重写系统终止性的可自动化技术。博士论文，阿姆斯特丹自由大学，2012。

16 Cynthia Kop. WANDA——一种高阶终止工具（系统描述）。载于 Zena M. Ariola 编，计算与演绎形式结构第五届国际会议（FSCD 2020），2020年6月29日至7月6日，法国巴黎（线上会议），LIPIcs 第167卷，页码 36:1–36:19。Schloss Dagstuhl - Leibniz-Zentrum für Informatik，2020。doi:10.4230/LIPIcs.FSCD.2020.36。17 Cynthia Kop。Wanda 源代码仓库，2023。URL： https://github.com/hezzel/wanda。18 Cynthia Kop 和 Deivid Vale。高阶复杂性的元组解释。载于 Naoki Kobayashi 编，计算与演绎形式结构第六届国际会议（FSCD 2021），2021年7月17日至24日，阿根廷布宜诺斯艾利斯（线上会议），LIPIcs 第195卷，页码 31:1–31:22。Schloss Dagstuhl - Leibniz-Zentrum für Informatik，2021。 doi:10.4230/LIPIcs.FSCD.2021.31。19 Cynthia Kop 和 Femke van Raamsdonk。代数函数系统的动态依赖对。逻辑方法与计算机科学，8(2)，2012。 doi:10.2168/LMCS-8(2:10)2012。20 Adam Koprowski。高阶递归路径排序的 Coq 形式化。应用代数、工程与通信计算，20(5-6):379–425，2009。 doi:10.1007/s00200-009-0105-5。21 Martin Korp、Christian Sternagel、Harald Zankl 和 Aart Middeldorp。蒂罗尔终止性工具 2。载于 Ralf Treinen 编，《重写技术与应用》，第20届国际会议（RTA 2009），2009年6月29日至7月1日，巴西巴西利亚，会议论文集，《计算机科学讲义》第5595卷，页码 295–304。Springer，2009。doi:10.1007/978-3-642-02348-4_21。22 Keiichirou Kusakari、Yasuo Isogai、Masahiko Sakai 和 Frédéric Blanqui。基于高阶重写系统强可计算性的静态依赖对方法。IEICE 信息与系统汇刊，92-D(10):2007–2015，2009。 doi:10.1587/transinf.E92.D.2007。23 Friedrich Neurauter 和 Aart Middeldorp。重新审视用于证明项重写终止性的矩阵解释。载于 Manfred Schmidt-Schauß 编，重写技术与应用第22届国际会议论文集（RTA 2011），2011年5月30日至6月1日，塞尔维亚诺维萨德，LIPIcs 第10卷，页码 251–266。Schloss Dagstuhl - Leibniz-Zentrum für Informatik，2011。 doi:10.4230/LIPIcs.RTA.2011.251。24 J.C. van de Pol。高阶重写系统的终止性。博士论文，乌得勒支大学，1996。URL: https://www.cs.au.dk/~jaco/papers/thesis.pdf。25 José-Luis Ruiz-Reina、José-Antonio Alonso、María-José Hidalgo 和 Francisco-Jesús Martín-Mateos。在 ACL2 定理证明器中形式化重写。载于 John A. Campbell 和 Eugenio Roanes-Lozano 编，人工智能与符号计算，AISC 2000 国际会议，2000年7月17日至19日，西班牙马德里，修订论文，《计算机科学讲义》第1930卷，页码 92–106。Springer，2000。 doi:10.1007/3-540-44990-6_7。26 Bas Spitters 和 Eelis van der Weegen。类型论数学中的类型类。数学结构与计算机科学，21(4):795–825，2011。 doi:10.1017/S0960129511000119。27 William W. Tait。有限类型泛函的内涵解释 I。符号逻辑杂志，32(2):198–212，1967。 doi:10.2307/2271658。28 René Thiemann 和 Christian Sternagel。使用 CeTA 认证终止性证明。载于 Stefan Berghofer、Tobias Nipkow、Christian Urban 和 Makarius Wenzel 编，高阶逻辑定理证明，第22届国际会议（TPHOLs 2009），2009年8月17日至20日，德国慕尼黑，会议论文集，《计算机科学讲义》第5674卷，

,

20 参考文献

《计算机科学讲义》，页码 452–468。Springer，2009。 doi:10.1007/978-3-642-03359-9\31.29 Deivid Vale 和 Niels van der Weide。Onijn 文档，2022。URL： https://deividrvale.github.io/nijn-coq-script-generation/onijn/index.html。30 Deivid Vale 和 Niels van der Weide。deividrvale/nijn-coq-script-generation：公共 API 首次发布，2023年5月。 doi:10.5281/zenodo.7915736。31 Niels van der Weide 和 Deivid Vale。nmvdw/Nijn：1.0.0，2023年5月。 doi:10.5281/zenodo.7913023。32 Niels van der Weide、Deivid Vale 和 Cynthia Kop。高阶多项式解释的认证。载于 Adam Naumowicz 和 René Thiemann 编，国际交互式定理证明第14届会议（ITP 2023），莱布尼茨信息学国际会议论文集（LIPIcs）第268卷，页码 30:1–30:20，德国 Dagstuhl，2023。Schloss Dagstuhl – Leibniz-Zentrum für Informatik。URL： https://drops.dagstuhl.de/opus/volltexte/2023/18405, doi:10.4230/LIPIcs.ITP.2023.30。33 Johannes Waldmann。Matchbox：一种匹配有界字符串重写工具。载于 Vincent van Oostrom 编，重写技术与应用第15届国际会议（RTA 2004），2004年6月3日至5日，德国亚琛，会议论文集，《计算机科学讲义》第3091卷，页码 85–94。Springer，2004。doi:10.1007/978-3-540-25979-4_6。34 Akihisa Yamada。用于项重写终止性的多维解释。载于 André Platzer 和 Geoff Sutcliffe 编，自动推理——CADE 28——自动推理第28届国际会议，2021年7月12日至15日，线上会议，会议论文集，《计算机科学讲义》第12699卷，页码 273–290。Springer，2021。 doi:10.1007/978-3-030-79876-5_16。35 Akihisa Yamada、Keiichirou Kusakari 和 Toshiki Sakabe。名古屋终止性工具。载于 Gilles Dowek 编，重写与类型化 Lambda 演算联合国际会议（RTA-TLCA 2014），作为维也纳逻辑夏季会议（VSL 2014）的一部分举行，2014年7月14日至17日，奥地利维也纳，会议论文集，《计算机科学讲义》第8560卷，页码 466–475。Springer，2014。 doi:10.1007/978-3-319-08918-8_32。