Theories as inductive types

Colin Rothgang

Supervised by
Prof. Dr. Alan T. Huckleberry
Dr. Florian Rabe

A thesis submitted in partial fulfillment
of the requirements for the degree of
Bachelor of Science
(Mathematics)

at
Jacobs University Bremen

January 15, 2020

JACOBS
UNIVERSITY

Statutory Declaration

Family Name, Given/First Name	Rothgang, Colin
Matriculation number	30001034
What kind of thesis are you submitting: Bachelor-, Master- or PhD-Thesis	Bachelor Thesis

English: Declaration of Authorship

I hereby declare that the thesis submitted was created and written solely by myself without any external support. Any sources, direct or indirect, are marked as such. I am aware of the fact that the contents of the thesis in digital form may be revised with regard to usage of unauthorized aid as well as whether the whole or parts of it may be identified as plagiarism. I do agree my work to be entered into a database for it to be compared with existing sources, where it will remain in order to enable further comparisons with future theses. This does not grant any rights of reproduction and usage, however.

This document was neither presented to any other examination board nor has it been published.

German: Erklärung der Autorenschaft (Urheberschaft)

Ich erkläre hiermit, dass die vorliegende Arbeit ohne fremde Hilfe ausschließlich von mir erstellt und geschrieben worden ist. Jedwede verwendeten Quellen, direkter oder indirekter Art, sind als solche kenntlich gemacht worden. Mir ist die Tatsache bewusst, dass der Inhalt der Thesis in digitaler Form geprüft werden kann im Hinblick darauf, ob es sich ganz oder in Teilen um ein Plagiat handelt. Ich bin damit einverstanden, dass meine Arbeit in einer Datenbank eingegeben werden kann, um mit bereits bestehenden Quellen verglichen zu werden und dort auch verbleibt, um mit zukünftigen Arbeiten verglichen werden zu können. Dies berechtigt jedoch nicht zur Verwendung oder Vervielfältigung.

Diese Arbeit wurde noch keiner anderen Prüfungsbehörde vorgelegt noch wurde sie bisher veröffentlicht.

15.5.2019, Colin Rothgang
Date, Signature

Theories as inductive types

Colin Rothgang

January 15, 2020

Abstract

Inductive types are a prime example in which formal and informal mathematics differ. In informal mathematics induction (like other basic language features) is a pragmatic feature, i.e. while every inductive argument is rigorous, the induction principle itself is kept pragmatic. In formal mathematics the induction principle must be formally defined, creating a tension between informal and formal mathematics. This makes informal mathematics a lot more general than formal mathematics and begs the question how general (formal) inductive types can be.

In order to formalize induction in maximal generality, we consider an extremely general structure for which inductive arguments can be carried out: the set of expressions over a given formal systems (specified by a theory). We therefore consider a completely general theory and try to interpret it as an inductive type without restricting to a specific foundation. We would like to allow all theories, but have to restrict in practice because a formal induction principle is not known for all theories. We design a foundation-independent structural feature for inductive types, which elaborates theories as inductive types including the various induction principles. We also define a convenience feature for definitions of functions by induction over inductive types. We discuss further generalizations and limitations of these features and show they suffice to define all primitive recursive functions. Furthermore, we design a feature for the definition of proofs by induction. We implement these structural features as an extension to the foundation-independent formal system MMT and use them to define a number of examples of inductive types and inductively-defined functions. In particular, we define natural numbers and basic arithmetical functions, lists, the zip function over a lists of given length and free monoids.

Contents

1 Introduction and Related Work

Motivation and related work Two major schools of thought in philosophy of mathematics are Platonism and Formalism. Platonism asserts that mathematical entities exists in an abstract mathematical universe and mathematicians merely discover these entities, whereas Formalism asserts that mathematical entities exist only as formulae in formal languages. The Formalist approach, requires a formal definition of mathematical entities, rather than merely describing them by their properties. This makes it less suitable for mathematical research. For formalization of mathematics in computer-readable formats, however, the Formalist approach is required.

In order to minimize the difficulties expressing mathematics formally, using a strong foundation allowing easy construction of most common mathematical structures and definition patterns is imperative. An important example are inductive types, which are used as a basic language feature in various languages, including the languages used by the interactive-theorem provers Mizar [TB85], HOL Light [Har96] and Coq [Coq15].

At the same time, a main issue for formalized mathematics is the usage of different languages based on strongly different foundations. For example the languages used by Mizar, HOL Light and Coq are based on Tarski-Grothendieck set-theory, simple times and logic, and dependent-types using propositions-as-types. it has been proposed [KR16] to use a foundation independent language, in which other formal languages including their foundations can be expressed. This language can then serve as an intermediate language for translations. The foundation-independent language and system Mmt [RK13; Rab14] (meta-meta-language or meta-meta-tool) has been developed, to allow expressing and reasoning about other formal languages including their foundations.

Following the same idea, we design a language feature for inductive types without committing to a specific foundation. This feature interprets a theory as the type of all well-typed terms which can be constructed by the declarations of the theory, the *type* of the theory. Our investigation is based on the Mmt language and system, as Mmt provides universally applicable theories, without committing to a particular logic.

There are three ways to define a language feature like induction in a formal language:

1. Including it as a primitive feature in the language (as done in Coq)
2. Elaborating inductive definitions into a concrete definition in the language (as done in Isabelle/HOL)
3. Elaborating inductive definitions into an axiomatization of the defined type (as done in PVS)

In order to stay as foundation independent as possible, we envision to use the third method and elaborate inductive definitions into an axiomatization.

For such tasks *structural features* were proposed [Ian17]. The idea of structural features is to elaborate a theory using a (structural) language feature to an elaboration, a theory axiomatizing the theory without using the language feature. In practice this approach has barely been used in Mmt up to this point.

Contribution In this paper, we develop a structural feature for *inductive types* and to extend the implementation of Mmt to elaborate them using a minimal foundation.

The elaboration of inductive types is based on two different ideas, an abstract idea and a construction idea:

1. We can formalize a theory as its initial model (a model of a theory is intuitively a second theory over which all expression are defined and a morphism from the first to the second theory).
2. We can elaborate a theory as the type of well-typed terms over the declarations of the theory.

We implement this structural feature as a plugin to the MMT system, as well as a number of convenience features and describe how they can be used to define functions by induction. We show that our features allows the definition of all primitive recursive functions and discuss possible generalizations.

Overview Two concepts are fundamental for our investigation, MMT theories and structural features. They are introduced in sections 2. Section 3 describes the design of the structural feature in its simplest form and proves its soundness, further generalizations and limitations of the design of the feature are discussed in section 4. In section we describe 5 how functions can be defined by induction over inductive types and section 6 discusses advanced additional features, namely the definition of the type of a theory and proofs by induction over inductive types. Finally, the implementation of the feature(s) is outlined in section 7.

2 Preliminaries

2.1 Theories and theory morphisms in Mmt

One fundamental concept for MMT are declarations and theories:

Definition 2.1 (Declaration).
A (symbol) declaration in MMT is of the form

$$\begin{array}{lll} Dec & ::= & n : o[= o] & \text{symbol declaration} \\ o & ::= & n \mid o(o^*) \mid o(Dec^*, o) & \text{MMT expression} \end{array}$$

, where n is a name and the expressions o after the colon and the equality sign are the (optional) type and the (optional) definien. For our purpose, we assume every symbol declaration has a type, but not necessarily a definien. Besides symbol declarations we also consider derived declarations. Derived declarations are of the form:

$$Dec \quad ::= \quad n : \ n\ o^* = \{Dec^*\} \qquad \text{Derived declarations}$$

Definition 2.2 (Theories).
We will introduce a simplified (non-modular) definition of theories. The full version can be found in [RK13]. A *theory* $\mathcal{T}$ is of the form

$$\begin{array}{llll} TDec & ::= & T\ Dec^* = Dec^* & \text{atomic theory declaration} \\ T & ::= & t \mid t(o*) & \text{theories} \end{array}$$

, where $D_1, \ldots, D_q$ are the *theory-parameters* of t and Dec are declarations in MMT.

For each theory $\mathcal{T}$ let dom($\mathcal{T}$) denote the set of symbols declared in the theory $\mathcal{T}$ and Obj($\mathcal{T}$) the set of well-formed terms generated by the declarations in dom($\mathcal{T}$). Finally, we need to introduce the notion of theory morphisms of a theory.

Definition 2.3 (Theory morphism).
Given theories $\mathcal{A}$ and $\mathcal{B}$ a theory-morphism $\mathfrak{m}: \mathcal{A} \to \mathcal{B}$ is of the form

$MDec$	$::=$	$M : T \to T = \{Ass^*\}$	atomic morphism declarations
Ass	$::=$	$c := o$	assignment to symbol
M	$::=$	$\mathfrak{m} \mid \mathrm{id}\,T \mid M; M$	morphisms

such that for each symbol $c \in \mathrm{dom}(\mathcal{A})$ the morphism $\mathfrak{m}$ contains exactly one assignment $c := o$ with $o \in \mathrm{Obj}(\mathcal{B})$. Furthermore, a well-typed theory-morphism $\mathfrak{m}$ must *preserve judgement* i.e.

- $\mathfrak{m}$ preserves types: $c : tp \Rightarrow \mathfrak{m}(c) : \mathfrak{m}(tp)$ and
- $\mathfrak{m}$ preserves definitions: $c = def \Rightarrow \mathfrak{m}(c) = \mathfrak{m}(def)$.

A well-typed theory-morphism $\mathfrak{m}: \mathcal{A} \to \mathcal{B}$ induces a judgement preserving mapping $\mathrm{Obj}(\mathcal{A}) \to \mathrm{Obj}(\mathcal{B})$ i.e. $\vdash_{\mathcal{A}} o : o' \Rightarrow \vdash_{\mathcal{B}} \mathfrak{m}(o) : \mathfrak{m}(o')$ and $\vdash_{\mathcal{A}} o = o' \Rightarrow \vdash_{\mathcal{B}} \mathfrak{m}(o) = \mathfrak{m}(o')$ as shown in [Rab14]. In fact MMT theories form a category with respect to theory morphisms [MR19].

2.2 Structural features

Structural features can be used to extend the declaration-level syntax of a language without having to extend the foundation of the language. Instead structural features allow to elaborate (derived) declarations using the feature to declarations in the basic language.

Definition 2.4 (Derived declaration).
Derived declarations are declarations build using the third production in definition 2.1. In this paper we assume derived declarations D_d are of the form:

$$D_d \quad ::= \quad l : f\ F^* = S^*$$

Given a derived declaration $l : \mathfrak{f}\ F_1\ F_2\ \ldots\ F_n = S_1\ \ldots\ S_m$, we call l its *name*, f its *feature*, F_i its arguments and S_j its internal declarations. We can regard the internal declarations S_j to form a theory with theory parameters F_i, the *body* of the derived declaration. For the purpose of this paper, we will assume that derived declarations occur only inside theories.

Definition 2.5 (Outer Context of a derived declaration).
Given a derived declaration D_d contained in a theory $\mathcal{T}$, its *outer context* $\mathrm{Con}(D_d)$ is the union of the set of theory-parameters D_i of $\mathcal{T}$ with the set of arguments F_j of D_d.

We will use a slightly different definition for structural features than proposed in [Ian17], which is more practical for our purposes.

Definition 2.6 (Structural feature).
A structural feature is a triple $(\mathfrak{f}, \epsilon, \nu)$, where $\mathfrak{f}$ the the *name* of the feature, ϵ its *elaboration function* and ν its *validity predicate*. The validity predicate is a predicate on the set of derived declarations of the form $l : \ \mathfrak{f}\ F_1\ \ldots\ F_n = S_1 \ldots\ S_m$. A derived declaration satisfying the predicate is said to be *well-formed*. The elaboration function must map any well-formed derived declaration D_d of the form $l : \ \mathfrak{f}\ F_1\ \ldots\ F_n = S_1 \ldots\ S_m$ to a list of declarations called the *external declarations* of D_d. The set of external declarations of D_d is also called its *elaboration*, will be denoted by $\mathcal{E}$ and is often regarded as a theory.

Definition 2.7 (Soundness of a structural feature).
A structural feature $\mathfrak{F}$ is called sound if it elaborates each well-formed derived declaration of feature $\mathfrak{F}$ to a well-formed and well-typed elaboration.

In other words a structural feature is called sound if its elaboration preserves soundness.

3 Design of the structural feature for inductive types

In this section we will describe the design of a structural feature $\mathfrak{inductive} := (\mathfrak{inductive}, \epsilon_{\mathfrak{inductive}}, \nu_{\mathfrak{inductive}})$ for (mutually) inductive types and illustrate it on the example of natural numbers.

Firstly, we will discuss as a motivating example how to formalize natural numbers as an inductive type. Then, we will discuss which sorts of inductive declarations we want to support. Based on this, we will choose and describe the (minimal) foundation necessary to formalize inductive definitions. Finally, we will discuss how inductive definitions can be elaborated within this framework.

3.1 Motivating example

We will use natural numbers as a motivating example. Natural numbers $\mathbb{N}$ can be seen as the inductively-defined type n with two constructors $z : n$ and $s : n \rightarrow n$. In our language natural numbers can then be defined as:

Example (Natural numbers).

$$\begin{aligned}&\textit{inductive}\ \mathrm{nat}() \ =\\ &\quad n :\ \mathsf{type}\\ &\quad z :\ n\\ &\quad s :\ n\ \rightarrow\ n\end{aligned}$$

We want to elaborate this into a foundation-independent axiomatization of natural numbers. The idea is to elaborate this inductive type into the Peano axioms of natural numbers. In our system they can be formalized as:

$n : \mathit{type}$

$z : n$

$s : n \rightarrow n$

$\mathit{injective\_s}_0 : \{x_0 : \text{nat},\ x_1 : \text{nat}\}\ x_0 \dot{\neq} x_1 \ \rightarrow\ s\ x_0 \dot{\neq} s\ x_1$

$\mathit{no\_conf\_s\_z} : \{x_0 : \mathit{nat}\}\ z \dot{\neq} s\ x_0$

$\mathit{induct\_n} : \{n' : \mathsf{type},\ z' : n',\ s' : n' \ \rightarrow\ n'\}\ n \ \rightarrow\ n'$

$\mathit{induct\_z} : \{n' : \mathsf{type},\ z' : n',\ s' : n' \ \rightarrow\ n'\}\ z' \ \doteq\ \mathit{induct/n}\ n'\ z'\ s'\ \ z$

$\mathit{induct\_s} : \{n' : \mathsf{type},\ z' : n',\ s' : n' \ \rightarrow\ n',\ x_0 : n\}$

$\qquad s'\ \mathit{induct/n}\ n'\ z'\ s'\ \ x_0 \ \doteq\ \mathit{induct/n}\ n'\ z'\ s'\ \ (s\ x_0)$

Here the last three axioms $\mathit{induct\_*}$ formalize the principle of induction: Given a set n' and constructors z' and s' satisfying the same axioms, there is an isomorphism $(n, z, s) \ \rightarrow\ (n', z', s')$. Our declarations formalize the following equivalent universal property: Given (n', z', s') there exists a (unique) morphism $(n, z, s) \ \rightarrow\ (n', z', s')$.

In general, we will specify inductively-defined types of an inductive type as internal declarations as in above example. The elaboration will also follow the same basic idea.

3.2 Supported inductive types

We will now discuss which sorts of internal declarations we want to support with our feature. In order to decide on which declarations to support, we will look at existing implementations of inductive types in other formal systems like Isabelle/HOL, Coq or PVS and discuss which sorts of declarations they support.

All implementations allow for non-mutual induction with simple-typed first-order constructors (i.e. constructors only allow for 0-ary arguments). Most systems allow parametric constructors (whose type may dependent on a type-argument) and for mutually inductively-defined types. Additionally, some systems like Coq support certain dependently-typed constructors and provide limited support for constructors of higher-order (e.g. with higher-order arguments only in positive positions). We would like to allow for mutual induction and dependently-typed, polymorphicly-typed and shallow polymorphic (a limited form of dependent-typed declarations, as explained below) internal declarations. Additionally, we will see that our design will allow for certain outgoing declarations (term-level declarations which are not themselves constructors) and for constructors with a definien (their importance is discussed in section 5). An overview of different systems and the supported internal declarations is given in below table:

System	Isabelle/HOL	Coq	PVS	our feature
mutual induction	yes	yes	no	yes
parametric	type parameters	arbitrary	arbitrary	arbitrary
dependently typed constructors	no	yes	predicate subtypes	yes
higher order constructors	no	limited	no	limited
outgoing	no	no	no	yes
defined constructors	no	no	no	yes

3.3 Logical Framework with shallow-polymorphism (PLF)

Based on our choice of supported internal declarations, we decide to use the logical framework (LF) extended with shallow polymorphism (PLF) as our Foundation. PLF is strong enough to express the declarations we need, but still a rather minimal system, supporting the goal of using this feature to represent inductive types in formal systems using differing foundations.

LF is a logical framework based on the $\lambda - \pi$ calculus (the typed-lambda calculus with an dependently-typed product (quantifying over types (i.e. (typed variables) only)). Therefore, LF supports first order dependent-function types with the proposition-as-types principle to represent first-order minimal logic. This logic allow the logical connectives $\wedge, \vee, \Rightarrow, \neg$ and a primitive symbol $\bot$ defined in terms of $\Rightarrow, \neg$ and $\bot$. They correspond to $\rightarrow, \neg$ and $\bot$ in LF. There are three sorts of objects in LF: types, kinds (in particular the universal kind `type`) and expressions (build from the former two).

We will additionally allow for *shallow-polymorphism*, i.e. declarations may use Π types (corresponding to universal quantification) quantifying over kinded variables, however only in the beginning of their type. More formally, shallow-polymorphic types must be of the form $\{d_1 : D_1, d_2 : D_2, \ldots, d_r : D_r\}\ A_1 \rightarrow A_2 \rightarrow \ldots \rightarrow A_s \rightarrow A_{s+1}$. Here the $\{d_1 : D_1, d_2 : D_2, \ldots, d_r : D_r\}$ denotes the Pi-type with arguments d_i of type D_i.

We will also assume an typed equality and inequality symbol $\doteq$ and $\not\doteq$, which can be defined in this framework.

Given a declaration $decl :\ \{d_1 : D_1, d_2 : D_2, \ldots, d_r : D_r\}\ A_1 \rightarrow A_2 \rightarrow \ldots \rightarrow A_s \rightarrow A_{s+1}$ in PLF, we will call $tp := \{d_1 : D_1, d_2 : D_2, \ldots, d_r : D_r\}\ A_1 \rightarrow A_2 \rightarrow \ldots \rightarrow A_s \rightarrow A_{s+1}$ its type and A_{s+1} its *return type*. There are two different sorts of types of a declarations:

1. A declaration is called a *Type-Level* declaration iff its return type is a kind
2. A declaration is called a *Term-Level* declaration iff its return type is a type

Furthermore, a declaration of the form $decl : A_1 \to A_2 \to \ldots \to A_s \to A_{s+1}$ is called *simply-typed*. Non simply-typed declarations are called *dependently-typed*. The type of a simply-typed declaration is called a simple type, the type of a dependent-typed declaration is called a dependent type.

3.4 Main design idea

Since we want to support mutual inductive definitions as well as shallow-polymorphic types, it will be useful to specify the inductively-defined types as type definitions in the internal declarations. Afterwards, the constructors of each type are listed.

Let D_{ind} be a derived declaration $l :\ f\ E_1\ \ldots\ F_p = S_1, \ldots, S_m$ of the feature $\mathfrak{inductive}$ contained in a theory $\mathcal{T}$. Let $\mathcal{I}$ denote its body. For now, we will assume that D_{ind} takes no parameters, i.e. $p = 0$. We will discuss the case $p > 0$ in section 4.1. We want to describe how the internal declarations S_i are elaborated. In our example, the declared inductive type is nat, it takes no arguments E_i and the internal declarations are n, z and s.

We will differentiate between Type-Level and Term-Level internal declarations. We denote the set of Type-Level declarations of D_{ind} by $\mathrm{TPL}(D_{ind})$ and the set of Term-Level declarations of D_{ind} by $\mathrm{TML}(D_{ind})$. We define the types *inductively-defined* by D_{ind} as the set of return types of the Type-Levels in $\mathrm{TPL}(D_{ind})$ and denote them by $\mathrm{types}(D_{ind})$.

We distinguish two sorts of Term-Level internal declarations, *constructors* of Type-Levels $tpl \in \mathrm{TPL}(D_{ind})$ defined by D_{ind} and *outgoing Term-Level* declarations. Constructors for a Type-Level $tpl \in \mathrm{TPL}(D_{ind})$ are Term-Levels tml whose return type is a type defined by D_{ind}. Let $\mathrm{Constr}(tpl)$ denotes the set of all constructors of a Type-Level tpl. In case of natural numbers, the only Type-Level declaration is n, the Term-Level declarations are z and s, both are constructors of the Type-Level declaration n.

All other internal declarations (for instance untyped declarations or declarations of a higher-type universe than kinds), will not be accepted by $\mathfrak{inductive}$ (i.e. the derived declaration D_{ind} fails the validity predicate $\nu_{\mathfrak{inductive}}$). We will also reject internal declarations containing a constructor with an argument of higher-order depending on an inductively-defined type. This will be discussed in more detail in subsection 4.4.

We will regard each inductively-defined type $\mathrm{tp}(tpl)$ as the type of well-typed expressions over the constructors of tpl. In particular, for each term tm of type $\mathrm{tp}(tpl)$ there is a unique Term-Level tml and unique arguments $x_1, \ldots, x_s$ for tml s.t. $tml\ x_1\ \ldots\ x_s \doteq tm$.

The basic idea for elaborating D_{ind} is to use this property to define each of the the types defined by D_d by formally expressing this property.

Therefore, the elaboration of the internal declarations for $\mathfrak{inductive}$ should consist of:

1. All the Type-Level declarations and their constructors. For this purpose it suffices to copy all internal declarations into the elaboration.
2. Declarations stating that each term of an inductively-defined type $\mathrm{tp}(tpl)$ generated by the constructors $\mathrm{Constr}(tpl)$ is generated uniquely (*no-confusion declarations*)
3. Declarations stating that the inductively-defined types contain no more than the terms generated by the respective constructors (*no-junk declarations*)

For the no-confusion declarations, we will need to assume that the constructors are not dependently-typed, to produce all required declarations. Nevertheless, we will try to develop a design that works also for dependently-typed constructors as far as possible, to ease the development of an improved feature that can also work with dependently-typed constructors (as envisioned in section 8).

We will express these requirements as follows in the elaboration.

The copied internal declarations By copying the internal declarations into the elaboration we ensure the elaboration is a theory morphism.

No-confusion For each constructor we need declarations stating its injectivity. Additionally, for each constructor we need declarations stating that for each other constructor of the same return type their images are disjoint. For dependent-typed constructors these declaration are not well-typed (as the equality requires both sides to have the same type), so we are forced to assume the constructors are simple-typed (meaning the type has no named arguments used in the type of the constructor).

No-junk This will be the trickiest part. The idea is to use our abstract idea already mentioned in the section 1 to represent the no-junk requirements. We formalize it by stating that any model $\mathcal{E}'$ of the elaboration is (up to isomorphism) the initial model $\mathcal{I}_{\mathcal{I}}$ of $\mathcal{I}$, i.e. for any model $\mathcal{E}'$ of the internal declarations, there exists a (unique) morphism $\mathcal{I}_{\mathcal{I}} \to \mathcal{E}'$ and hence $\mathcal{E} \to \mathcal{E}'$. For each internal declaration d_{int} in $\mathcal{I}$ we will denote its image under this morphism by d'_{int}. We can represent this requirement by including one declaration for each internal declaration as follows: For each Type-Level tpl of type $\mathrm{tp}(tpl)$ we state the existence of a function $induct/tpl : tpl \to tpl'$ given the declarations $S'_i \in \mathcal{E}'$ for each S_i. Let $induct$ denote the resulting map mapping each inductively-defined type $\mathrm{tp}(tpl)$ to the corresponding type $\mathrm{tp}(tpl')$ in $\mathcal{E}'$ and each term tm of an inductively-defined-type $\mathrm{tp}(tpl)$ to the corresponding term $induct/tpl(tm)$ in $\mathcal{E}'$ and acting by identity on all other types and terms. Then, we can define the pushout $induct_*$ of $induct$ at the level of product types, mapping each component term separately via $induct$. Interpreting the argument list of an internal declaration as a n-tuple of the arguments, we can apply $induct_*$ to said argument list. For each constructor

$tml : argsTp \to \mathrm{tp}(tpl)$ of tpl we obtain the requirement for the map $induct_{tpl}$ to respects the type and (if existent) definien of tml as shown in the commutative diagram in figure 1 and state it in the the corresponding declaration $induct/tml$.

$$\begin{tikzcd}
\boxed{argsTp} \arrow[r, "tml"] \arrow[d, "induct_*"] & \boxed{\mathrm{tp}(tpl)} \arrow[d, "induct"] \\
\boxed{induct_*(argsTp)} \arrow[r, "tml'"] & \boxed{induct(\mathrm{tp}(tpl)) \overset{!}{=} \mathrm{tp}(tpl')}
\end{tikzcd}$$

Figure 1: The no-junk requirement for a constructor *tml* of tp(*tpl*).

For an outgoing Term-Level *tml* the requirement is very similar, except that the return type tp of tml is not an inductively-defined type and hence untouched by *induct*, so $tp = ind(tp)$. Thus, the diagram collapses collapses to the following diagram in figure 2:

$$\begin{tikzcd}
\boxed{argsTp} \arrow[d, "induct_*"] \arrow[dr, "tml"] & \\
\boxed{induct_*(argsTp)} \arrow[r, "tml'"] & \boxed{tp}
\end{tikzcd}$$

Figure 2: The no-junk requirement for an outgoing Term-Level *tml*.

To each declaration in the elaboration we additionally add all parameters in $\mathrm{Con}(D_{ind})$ (similar to the design of parametric theories).

3.5 Concrete construction

3.5.1 Formal definition of the $\mathfrak{inductive}$ feature

Definition 3.1 (The validity predicate $\nu_{\mathfrak{inductive}}$).
A derived declaration of the feature $\mathfrak{inductive}$ is accepted by the predicate ν_ind iff is it well-formed and –typed as a theory and all internal declarations are supported as described in section 3.2 (i.e. they have shallow-polymorphic types and no arguments of higher-order in an inductively-defined type in negative positions).

Definition 3.2 (The elaboration function $\epsilon_{\mathfrak{inductive}}$).
The elaboration function $\epsilon_{\mathfrak{inductive}}$ elaborates a well-formed derived declarations $\mathfrak{D}$ of feature $\mathfrak{inductive}$ into an elaboration consisting of exactly the following declarations:

1. A copy of each internal declarations

2. An injectivity declaration for each constructor and a no-confusion declaration for each pair of constructors of the same inductively-defined type as defined in section 3.5.2.
3. One *induct* declaration for each internal declaration as defined in section 3.5.2.

Definition 3.3 (The $\mathfrak{inductive}$ feature).
The $\mathfrak{inductive}$ feature is a triple $(\mathfrak{inductive}, \epsilon_{\mathfrak{inductive}}, \nu_{\mathfrak{inductive}})$, with $\nu_{\mathfrak{inductive}}$ and $\epsilon_{\mathfrak{inductive}}$ as defined in the above definitions 3.1 and 3.2.

We will now describe in detail how the external declarations are defined.

3.5.2 Construction of the external declarations

Firstly, we copy all internal declarations into the elaboration. In the example nat, we simply need to copy the internal declarations n, z and s into the elaboration.

$$n : \ type$$

$$z : \ n$$

$$s : \ n \ \to \ n$$

This obviously preserves soundness, since the internal declarations form a well-typed theory, otherwise the derived declaration is not well-formed.

Secondly, we build the no-confusion declarations: For each constructor $tml : \{d_1, d_2, \ldots, d_r\}\ A_1 \to A_2 \to \ldots \to A_s \to \mathrm{tp}(tpl)\ y_1\ \ldots\ y_t$ of tpl (for some (not necessary all) arguments y_i of tpl), we build an injectivity declaration $injective\_tml\_i$ for each argument $a_i : A_i$ of tml in which tml is not dependently-typed. This way we state that if any argument to tml is different so will be the result, which is clearly equivalent to the injectivity of tml. Wlog. we assume that those are exactly the named arguments D_i (otherwise we consider them as unnamed arguments), i.e. we build

$$\begin{aligned} injective\_tml\_i : \ & \{d_1, \ldots, d_r, a_1 : A_i, \ldots, a_s : A_s, a'_1 : A_1, \ldots a'_s : A_s\}\ (a_i \dot{\neq} a'_i) \\ & \to (tml\ d_1\ \ldots\ d_r\ a_1\ \ldots\ a_s \dot{\neq} tml\ d_1\ \ldots\ d_r\ a'_1\ \ldots,\ a_s) \end{aligned}$$

for each $1 \leq i \leq s$. In the example nat, we generate the injectivity axiom for s (z takes no arguments, so no injectivity declaration is required):

$$injective\_s : \ \{x_0 : \ \mathrm{nat},\ x_1 : \ \mathrm{nat}\}\ x_0 \dot{\neq} x_1 \ \to \ s\ x_0 \dot{\neq} s\ x_1$$

Since the types of the arguments of the constructors have the correct types by definition the application of tml to its arguments is well-typed. As the constructor tml is not dependently-typed the type of the constructor applied to the arguments $d_1, \ldots, d_r, a_1, \ldots, a_s$ and $d_1, \ldots, d_r, a'_1, \ldots, a'_s$ are also equal for both sides of the $\doteq$

symbol, so the injectivity declarations are well-typed. As they are clearly well-formed Mmt declarations it follows that these declarations preserve soundness.

Thirdly, assuming tml is non dependently-typed (i.e. $r = 0$) we also build the non-confusion declarations stating the image of tml is disjoint with the image of any other non-dependently-typed constructor of tpl. For every other such constructor $tml' : B_1 \to B_2 \to \ldots \to B_t \to tpl\ y_1\ \ldots\ y_t$ of $\mathrm{tp}(tpl)$ we build the declaration $no\_conf\_tml\_tml'$ stating that the images of tml and tml' are disjoint:

$$no\_conf\_tml\_tml' : \{a_1 : A_1, \ldots, a_s : A_s, b_1 : B_1, \ldots, b_t : B_t\}$$
$$tml\ a_1\ \ldots\ a_s \not\doteq tml'\ b_1\ \ldots\ b_t$$

In the example nat we generate the no-confusion axiom for z and s:

$$no\_conf\_z\_s : \{x_0 : nat\}\ z \not\doteq s\ x_0$$

Soundness of the no_conf declarations follows similarly to the soundness of the injectivity declarations: Well-formedness as Mmt declarations in obvious and the application of the constructors is well defined by the definition of their arguments. The well-typedness of the $\doteq$ i.e. checking that the terms on both sides of the $\doteq$ symbol have same type is by assumption, otherwise the declaration is not generated, since then the terms are definitely not equal. This can be checked, since the constructors are not dependently-typed.

Fourthly, we build the no-junk declarations: For each Type-Level $tpl : \{d_1, \ldots, d_r\}\ A_1 \to \ldots \to A_s \to \mathsf{type}$ with constructors $c_1, \ldots, c_m$, we build an induct declaration $induct/tpl$:

$$induct/tpl : \{c'_1, \ldots, c'_m, d_1, \ldots, d_r, a_1 : A_i, \ldots, a_s : A_s\}$$
$$\mathrm{tp}(tpl) \to \mathrm{tp}(tpl')$$

In case of nat, we generate the no-junk declaration for the Type-Level n:

$$induct/n : \{n' : \mathsf{type},\ z' : n',\ s' : n' \to n'\}\ n \to n'$$

Here soundness is obvious as the declaration's type is a well-formed function type.

For each constructor $tml : \{e_1, e_2, \ldots, e_r\}\ B_1 \to B_2 \to \ldots \to B_s \to \mathrm{tp}(tpl)$ $y_1\ \ldots\ y_t$ of tpl, we build the corresponding no-junk declaration $induct/tml$:

$$induct/tml : \{c'_1, \ldots, c'_m, e_1, \ldots, e_r, b_1 : B_i, \ldots, b_s : B_s\}$$
$$induct/tpl\ \mathfrak{M}\ (tml\ e_1\ \ldots\ e_r\ b_1\ \ldots\ b_s) \doteq$$

$$tml'\ induct(e_1)\ \dots\ induct(e_r)\ induct(b_1)\ \dots\ induct(b_s)$$

, where $\mathfrak{M}$ denotes the list of named arguments for $induct/tpl$ ($n'\ z'\ s'$ in our example nat). In the example nat, we generate no-junk declarations for the constructors z and s of n :

$$induct/z :\ \{n' :\ \mathsf{type},\ z' :\ n',\ s' :\ n'\ \rightarrow\ n'\}\ z'\ \doteq\ induct/n\ n'\ z'\ s'\ z$$
$$induct/s :\ \{n' :\ \mathsf{type},\ z' :\ n',\ s' :\ n'\ \rightarrow\ n',\ x_0 :\ n\}$$
$$s'\ induct/n\ n'\ z'\ s'\ x_0\ \doteq\ induct/n\ n'\ z'\ s'\ s\ x_0$$

Soundness follows from the definition of the arguments to the constructor making its application to the arguments well-typed (analogously for the primed constructor and arguments), the well-typedness of constructor arguments of an inductively-defined type is ensured by definition of the $induct$ function. Finally, the well-typedness of the $\doteq$ symbol follows by the definition of the $induct$ function.

Finally, for each outgoing Term-Level $tml : \{e_1, e_2, \dots, e_r\}\ B_1 \rightarrow B_2 \rightarrow \dots \rightarrow B_s \rightarrow tp$ for some $tp : \mathsf{type}$, we build the no-junk declaration $induct/tml$:

$$induct/tml :\ \{c'_1, \dots, c'_m, e_1, \dots, e_r, b_1 : B_i, \dots, b_s : B_s\}$$
$$(tml\ e_1\ \dots\ e_r\ b_1\ \dots\ b_s)\ \doteq$$
$$tml'\ induct(e_1)\ \dots\ induct(e_r)\ induct(b_1)\ \dots\ induct(b_s)$$

In the example nat, there are no outgoing Term-Levels, so no further declarations need to be generated. The well-typedness of the application of the outgoing Term-Level to its arguments follows by their definitions, the well-typedness of the primed constructor to its arguments follows by definition of the arguments and of the $\mathfrak{inductive}$ function.

By assumption tml is not dependently-typed, hence the types of the terms of both sides of the $\doteq$ symbol are the same and the $\doteq$ is well-typed. Thus, the no-junk declarations for outgoing Term-Levels are well-typed.

All in all, we elaborate the derived declaration nat:

$$inductive\ \mathrm{nat}()\ =$$
$$n :\ \mathsf{type}$$
$$z :\ n$$
$$s :\ n\ \rightarrow\ n$$

to the elaboration:

$$n :\ type$$

$z : n$

$s : n \to n$

$injective\_s_0 : \{x_0 : \text{nat},\ x_1 : \text{nat}\}\ x_0 \dot{\neq} x_1 \to s\ x_0 \dot{\neq} s\ x_1$

$no\_conf\_z\_s : \{x_0 : nat\}\ z \dot{\neq} s\ x_0$

$induct/n : \{n' : \mathsf{type},\ z' : n',\ s' : n' \to n'\}\ n \to n'$

$induct/z : \{n' : \mathsf{type},\ z' : n',\ s' : n' \to n'\}\ z' \doteq induct/n\ n'\ z'\ s'\ \ z$

$induct/s : \{n' : \mathsf{type},\ z' : n',\ s' : n' \to n',\ x_0 : n\}$
$\qquad s'\ induct/n\ n'\ z'\ s'\ \ x_0 \doteq induct/n\ n'\ z'\ s'\ \ (s\ x_0)$

3.6 Soundness

In this subsection, we will sketch the soundness-proof for the inductive feature.

Theorem 3.4. inductive *is sound.*

Sketch of proof of soundness of the inductive-feature. Here we need to check that the elaboration is well-typed and it it is satisfiable i.e. it has a model. For the second part we assume that all outgoing term-levels are compatible (i.e. consisting) with the no-junk and the no-confusion conditions and with each other. This assumption is necessary, as this question is in general undecidable. In this case, the elaboration will alway have a model (namely the initial model) as detailed above.

The well-typedness of the elaboration follows from the well-typedness of each external declaration, as shown in above section 3.5. □

4 Generalizations and limitations

4.1 Parametric inductive types

In this subsection, we will generalize the design of the feature inductive to also allow for *parametric inductive types.* We allow to two types of parameters of inductive types, for *fixed parameters* and for *variable parameters* (recursively uniform parameters and recursively non-uniform parameters in Coq).

Fixed parameters E_i are parameters from the outer context of the derived declaration (parameters to the derived declaration or a theory containing it). We add them as parameters to each external declaration i.e. each external declaration $d_{ext} : \{d_1 \ldots, d_s\}\ tp$ becomes $d_{ext} : \{E_1, \ldots, E_n, d_1 \ldots, d_s\}\ tp'$, where tp' is like tp with all references to another internal declaration replaced by that reference applied to all fixed parameters. In the no junk declarations, we need to supply these fixed parameters only once for the model and not for each declaration in the model. Consequently, when defining functions by induction fixed parameters must be specified only once.

Variable parameters are named parameters of individual internal declarations. They are treated like other arguments of internal declarations (shallow polymorphism ensures this will be well-typed). This is described in detail in section 3.5. In the no-junk declarations and inductive definitions they are must be specified like other arguments of internal declarations for each declaration in the model.

In particular, if all internal declarations share a variable parameter, this parameter can be specified differently for different constructors when defining functions by inductions. This allows to define a wider class of functions:

For example given the below inductive type:

$$\begin{aligned}
&\mathit{inductive}\ \mathit{finite\_sequence}() =\\
&\qquad \mathit{seq} :\ \{a :\ \mathsf{type},\ m :\ n\}\ \mathsf{type}\\
&\qquad \mathit{empty} : \{a :\ \mathsf{type}\}\ \mathit{seq}\ a\ z\\
&\qquad \mathit{add} :\ \{a :\ \mathsf{type},\ m :\ n\}\ \mathit{seq}\ a\ m\ \to\ a\ \to\ \mathit{seq}\ a\ (s\ m)
\end{aligned}$$

we define the following inductively-defined function (generating the sequence $0, 1, \dots, p$):

$$\begin{aligned}
&\mathit{inductive\_definition}\ \mathit{nat\_sequence}(p :\ n)\ : \mathit{finite\_sequence}() =\\
&\qquad \mathit{seq}\ =\ [a :\ \mathsf{type}]\ \mathit{seq}\ a\ p\\
&\qquad \mathit{empty}\ =\ [a :\ \mathsf{type}]\ \mathit{seq}\ (\mathit{leq}\ z)\ z\\
&\qquad \mathit{add} :\ [a :\ \mathsf{type},\ q :\ n,\ \mathit{fseq} :\ \mathit{seq}\ a\ p,\ x :\ a]\ \mathit{add}\ (\mathit{leq}\ (s\ (\mathit{min}\ a)))\\
&\qquad\qquad (\mathit{min}\ a)\ \mathit{fseq}\ (s\ (\mathit{min}\ a))\ .
\end{aligned}$$

Here $\mathit{geq} :\ n\ \to\ \mathsf{type}$ denotes the type of natural number less or equal to a given natural number and max the projection from $\mathit{leq}\ n$ down to the argument n. In this definition we use the information provided by the variable type parameter a in the definition. The definition therefore can't be implemented (this way) without using variable parameters.

4.2 Polymorphic types

By the choice of PLF as the foundation, internal declarations may use shallow-polymorphism. By design of the inductive feature, the no-junk declarations take a list of declarations potentially using shallow-polymorphism (exactly when the corresponding internal declarations use it) as arguments. This way, the no-junk declarations will have named arguments, which are themselves shallow polymorphic. Therefore, the type of these no-junk declarations will be a second-order polymorphic-type. As PLF only allows for first-order dependent types, these no-junk declarations are not well-typed in PLF.

This issue arises already in the example *list* given in appendix A.

This issue arises only for dependently-typed internal declarations and only for variable parameters, but as they are relatively common this is still a significant restriction. However, the elaboration must always be well-typed if the internal declarations are a well-typed theory.

To resolve this issue, a different foundation than PLF must be used for the elaboration: In particular, there is no problem if plain LF is used as Foundation or arbitrary dependent-types are allowed in the foundation (as in for instance Martin-Löf type theory). Alternatively, one could allow second-order dependent-types in the foundation, but require first-order dependent-typed internal declarations.

We leave it to the users of the feature to choose an appropriate foundation.

4.3 No-confusion axioms for dependently-typed constructors

As already hinted, in the section 3 the no-confusion axioms cannot be generated for dependently-typed constructors (at least not within LFP), so currently only axioms stating injectivity in arguments in which a constructor is not dependently typed are generated and no no-confusion axioms for dependent-typed constructors.

An example in which this issue occurs is the inductive type:

$$\begin{aligned}&Cardinality() =\\ &\qquad \text{Cardinality} : \mathsf{type}\\ &\qquad card : \{a : \text{ int}\}\ \text{Cardinality}\end{aligned}$$

, where $\text{int } m$ denotes the interval $((0, m))$ of natural numbers between 0 and m and int the type of all such intervals. Then, Cardinality is the type of equivalence classes of sets of same cardinality (there is only one each) and card the corresponding quotient map.

This is a well-known problem and in general undecidable. For this reason, only certain sorts of dependently-typed constructors can be supported by any feature for inductive definitions. This is explained in more detail in section 4.5.

4.4 Constructors with higher-order arguments

In this design of the structural feature inductive, we are elaborating the internal declarations to an elaboration, which axiomatizes the initial model of the internal declaration up to isomorphism (i.e. the initial model is the only model (up to isomorphism) of the elaboration). This construction doesn't work however, for constructors with arguments of higher-order in an inductively-defined type: The idea is that due to the *induct*$_*$ allowing to define inductive functions on the inductively-defined types, there are more function of inductively-defined types in the elaboration than in the internal declarations, since in the elaboration inductive functions over the inductively-defined types can be defined, but not in the internal declarations. Thus, there would be more terms of inductively-defined types in the elaboration than constructible by the constructors in $\mathcal{I}$ violating the construction idea of inductively-defined types being elaborated to the well-typed expressions over the constructors.

An example is the inductive type obtained by adding addition as additional constructor to the definition of natural numbers.

In fact this is a well-known problem and and leads to the restriction to strictly-positive constructors of inductive types for formal systems. In order to elaborate also constructors with high-order arguments of inductively-defined types, we will therefore need a different approach to design the elaboration function.

4.5 Dependently-typed induction

Inspecting our definition, we notice that the inductively-defined functions a user can define are always simply-typed in the inductive arguments. The most practically harmful incarnation of this limitation are inductive proofs. For instance we cannot define the proof that addition of natural numbers is commutative without referring to the previous case in the inductive step, which is not possible using the *induct* declarations and the $inductive\_definition$ feature outlined in section 5.1. When we inspect the requirements for claims to be proven using inductive proofs, we realize that they are exactly the strict logical relations on the internal declarations viewed as a theory.[RS13]

In order to resolve this issue we extended our elaboration with additional declarations allowing also the definitions of inductive proofs dependent on the inductive arguments. This is described in more detail in section 5.

This feature supports only very limited dependent-typed declarations, in particular shallow-polymorphism. The reason is that dependent-typed declarations cannot be supported in full generality, as it would lift equality judgements to the type-level and make the type-system undecidable. This problem is known as the interplay of dependent types and equality in higher-order type theory (HoTT). Nevertheless, certain classes of dependent-typed declarations can be allowed, like for instance shallow-polymorphic types. As these already cover the most important cases of dependent-typed constructors we decided to only support shallow-polymorphic typed constructors.

4.6 Arbitrary recursion

The aim of this feature is to support inductive types but not arbitrary recursion, in particular inductively-defined functions will always terminate. Therefore, inductive definitions are not Turing complete and thus strictly weaker than arbitrary recursion (for instance we cannot define the Ackermann function via induction). However, as will be seen later, the current feature already allows to conveniently specify primitive recursive definitions, which include most practically useful recursive definitions. This will be shown in subsection 5.2.

4.7 Quotients of inductive types

One interesting generalization of this feature is considering quotients of inductive types. This is a delicate special case of inductive types with extra conditions, since just adding the axiom that terms identified by the quotient map are equal as an outgoing constructor leads to a contradiction with the no-confusion declarations.

In order to support quotients of inductive types we therefore have to weaken the no-confusion declarations as required by the quotient map. For this reason, we currently don't support directly defining quotients of inductive types.

5 Inductive definitions

5.1 Inductively-defined functions

The idea of structural features has already been illustrated by the example of natural numbers. Once they are defined, functions for instance addition or multiplication of natural numbers can be defined by induction. For example we can define addition and multiplication of natural numbers as below.

$$plus: \; n \to n \to n$$

$$\quad = [m, n] \;\; ((induct/n \; (n \to n) \; ([x] \;\; x) \;\; [u] \;\; ([x] \;\; s \; (u \; x))) \;\; m) \, n$$

$$times: \; n \to n \to n$$

$$\quad = [m, n] \;\; ((induct/n \; (n \to n) \; ([x] \;\; z) \;\; [u] \;\; ([x] \;\; plus \; (u \; x) \; x)) \;\; m) \, n$$

In order to make definitions of inductive-types as convenient as possible, we implemented a utility feature for inductive definitions. Using this utility feature the definition of plus becomes:

$$inductive\_definition \; plus() \;:\; \mathrm{nat}() =$$

$$\quad n = (n \;\to\; n)$$

$$\quad z = ([x] \; x)$$

$$\quad s = ([u, x] \; s \; (u \; x))$$

Here nat() denotes the type of the derived declaration, it consists of the name of the corresponding inductively-defined type applied to concrete values for the arguments of the outer context of this inductively-defined type (they can however be references to new variables declared in this derived declaration). In particular, the body of the derived declaration should typecheck itself. This can be checked conveniently, when writing inductive definitions in an IDE.

Furthermore, we add some further declarations to the elaboration, allowing for limited pattern-matching over terms of an inductively defined type.[1] To conveniently allow for pattern matching we implemented another convenience feature for pattern matching. The syntax look like the syntax for inductive definitions, but there need not be a definien for each constructor case. Consequently, a pattern match will return an option type. We also implemented a convenience feature allowing to check, whether a term of an inductively-defined type was constructed by a given constructor of that type.

Examples for these features can be found in section A of the appendix.

5.2 Primitive recursion

In this subsection, we will briefly outline how arbitrary primitive recursive functions can be defined using the features outlined above.

For this we will use an additional structural feature **records** for record types (n-tuples with named fields) developed in parallel to **inductive**. The primitive recursive function are natural number valued function defined by the following 5 axioms:

1. The 0-ary constant function z is primitive recursive
2. The 1-ary sucessor function s in the definitin of natural numbers is primitive recursive

Both can be defined trivially using the **inductive** feature as a side-product of the inductive definition of natural numbers already discussed.

3. For every $n \geq 1$ and each $1 \leq i \leq n$, the n-ary projection function P_i^n, which returns its i-th argument, is primitive recursive.

This is possible, as the projection functions are defined by the **records** feature which can be used to define n-ary inductively-defined functions.

Additionally, we can define further primitive recursive functions using the next 2 axioms:

4. Given a k-ary primitive recursive function f, and k many m-ary primitive recursive functions $g_1, \ldots, g_k$, the composition of f with the g_i, i.e. the function $h(x_1, \ldots, x_m) = f(g_1(x_1, \ldots, x_m), \ldots, g_k(x_1, \ldots, x_m))$, is primitive recursive.

[1]In Coq and Agda, arbitrary recursion is allowed and soundness is ensured by a termination checker (so they are not complete either), if the termination checker is turned off, the languages become complete but unsound.

5. Given a k-ary primitive recursive function f, and a $(k+2)$-ary primitive recursive function g, the $(k+1)$-ary function h defined as follows is also primitive recursive:

$$h(z, x_1, \dots, x_k) = f(x_1, \dots, x_k)$$
$$h(s(y), x_1, \dots, x_k) = g(y, h(y, x_1, \dots, x_k), x_1, \dots, x_k).$$

Functions defined using 4. can be defined directly (using lambda functions). Functions using 5. can be defined using induction over natural numbers and direct definitions using lambda functions.

Therefore, the inductive feature allows to define all primitive recursive functions.

6 Advanced features

6.1 Theories as inductive types

The idea of this feature is to use the body of a theory as internal declarations. The elaboration will intuitively formalize the types of constructible terms over the theory and inductively-defined functions on them, i.e. all well-formed expressions over the declarations defined by the theory.

This is important, for the following reasons:

1. Because often inductive types are already formalized in some theory.
2. To allow for meta-reasoning: The reflection allows to push down from module level to object level. One example could be reasoning about logics, e.g. proving (or at least stating) completeness of FOL.

This feature becomes especially powerful once higher-order constructors are allowed, as meta-reasoning usually involves inductive definitions over higher-order constructor.

This feature and the structural feature records for record types (see section 5.2) are similar to the Mmt extension described in [MRK18] interpreting theories as record types. The main difference in our approach is that they realize the language extension as a foundational language feature (extending the foundation of the Mmt language), whereas we elaborate derived declarations into elaborations in the basic Mmt language which can be parsed without extending the foundation. On the other hand designing a feature as a primitive language feature allows to define it in higher generality.

It might therefore be interesting for future work to also design a feature for inductive types as a foundational language feature in Mmt.

6.2 Inductive Proofs

Once types and functions can be defined inductively, it it important to also be able to reason about these using inductive proofs. For this purpose we we generate additional external declarations similar (but slightly more powerful) to the no-confusion declarations, which can be used for inductive proofs.

To proof a statement by induction about one (or several mutually) inductively-defined types we need a proposition P_{tp} for each statement. Then, for each constructor $tml : \{e_1 : E_1,\ e_2 : E_2,\ \ldots,\ e_r : E_r\}\ B_1 \to B_2 \to \ldots \to B_s \to tp$ of a type tp, we will generate a declaration stating that given arguments $b_1 : B_1,\ \ldots,\ b_s : B_s$ and proofs of the corresponding propositions for the b_i of an inductively-defined type (denoted by $b_{i_1}, \ldots, b_{i_t}$), we obtain a proof for $P_{tp}\ (tml\ e_1\ \ldots\ e_r\ b_1\ \ldots\ b_s)$. We will therefore generate the following inductive proof declaration for tml:

$$ind\_proof/tml :\ \{e_1 : E_1,\ e_2 : E_2,\ \ldots,\ e_r : E_r,\ b_1 :\ B_1,\ \ldots,\ b_s : B_s$$
$$,\ p_{i_1} :\ P_{B_{i_1}},\ \ldots,\ p_{i_t} :\ P_{B_{i_t}}\}\ P_{tp}\ (tml\ e_1\ \ldots\ e_r\ b_1\ \ldots\ b_s)$$

Finally, we implemented a convenience feature for inductive proofs. One example of an inductive proof using this feature is the following:

$z\_plus\_n :\ \{m :\ n\} \vdash ((z + m)\ \doteq\ m)$

$z\_plus\_z : \vdash (z + z)\ \doteq\ z$

$z\_plus\_suc :\ m :\ n \vdash (z + m)\ \doteq\ m\ \to \vdash (z + (s\ m))\ \doteq\ (s\ m)$

$ind\_proof\ z\_neutral()\ :\ \mathrm{nat}()\ =$

$\quad n = [m] \vdash (nat/z + m)\ \doteq\ m$

$\quad z = z\_plus\_z$

$\quad s = z\_plus\_suc$

7 Implementation of the structural feature for inductive types

Due to the foundation independent nature of MMT, most of its functionality is implemented in so-called extensions of the MMT API implementing the basic system. In particular, there is an extension mmt-lf implementing LF and PLF and an interface for structural features in the MMT API. The implementation of the inductive feature uses this interface to extends mmt-lf with the additional feature inductive.

One interesting aspect of the implementation is the implementation of the $induct$ function heavily used in section 3 for the case of terms. Firstly, the implementation keeps track of the types of each term at every stage. The type tp of the argument tm to the $induct$ function is matched with the types $\mathrm{tp}(tpl)$ for each Type-Level tpl and it is checked for which arguments $d_1, \ldots, d_r, a_1, \ldots, a_s$ of tpl (if any) these types agree (in case of dependent-types). If they don't, we keep the term. Otherwise, we can construct the resulting term $induct(tm)$ as follows:

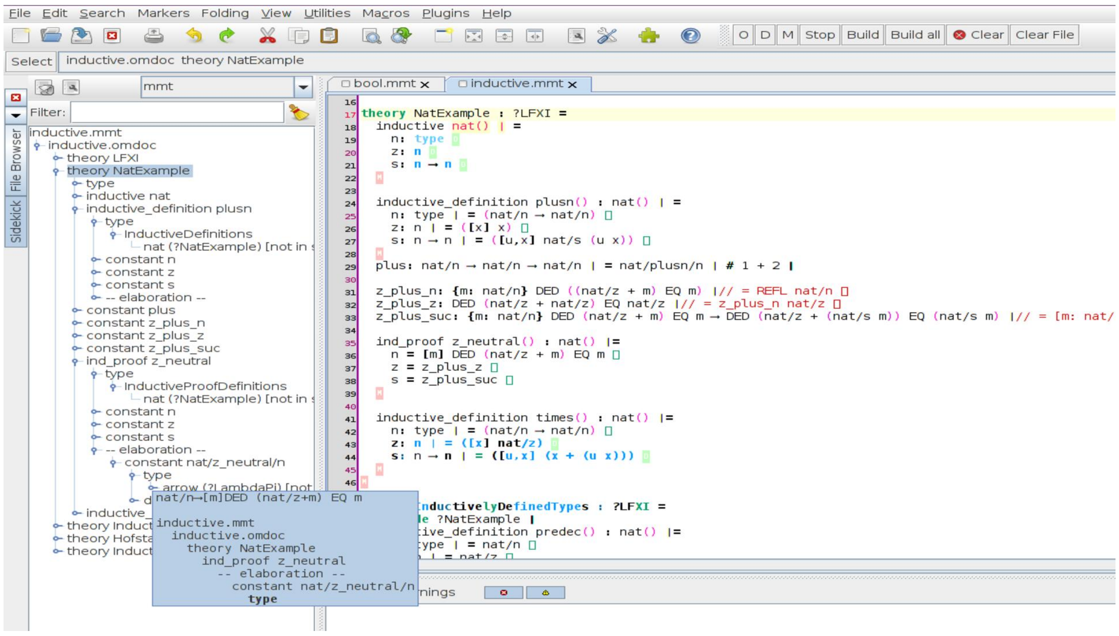
$$induct(tm) := induct/tpl\ E_1\ \ldots\ E_p\ c'_1\ \ldots\ c'_r\ d_1\ \ldots\ d_r\ a_1\ \ldots\ a_s\ tm\,.$$

Here, the E_i and c'_i are passed as arguments to the corresponding $induct/tml$ function and the d_i and a_i are inferred by matching tp against $\mathrm{tp}(tpl)$. Since induct acts by identity on the results of outgoing Term-Levels, the implementation of the no-junk declarations for Term-Levels doesn't need to distinguish between constructors and outgoing Term-Levels. The elaboration for inductive proofs are generated similarly.

One interesting property of the implementation is its laziness, meaning that the elaboration is only evaluated (computed), once it is actually used. This is useful primarily for efficiency reasons.

The structural feature 𝔦𝔫𝔡𝔲𝔠𝔱𝔦𝔳𝔢 as described in this paper is already fully implemented, in fact the elaboration for free monoids given in the appendix is the output of this structural feature. The source code for the structural feature can be found in this folder. This feature is included in the MMT system since release 13.

See figure 7 for an example of how the outlined features are used in practice. Further examples can be found in appendix A.


File Edit Search Markers Folding View Utilities Macros Plugins Help
O D M Stop Build Build all Clear Clear File
Select
inductive.omdoc theory NatExample
mmt
Filter:
Sidekick File Browser
bool.mmt
inductive.mmt

```
inductive.mmt
  inductive.omdoc
    theory LFXI
    theory NatExample
      type
      inductive nat
      inductive_definition plusn
        type
          InductiveDefinitions
            nat (?NatExample) [not in
        constant n
        constant z
        constant s
        -- elaboration --
      constant plus
      constant z_plus_n
      constant z_plus_z
      constant z_plus_suc
      ind_proof z_neutral
        type
          InductiveProofDefinitions
            nat (?NatExample) [not in
        constant n
        constant z
        constant s
        -- elaboration --
          constant nat/z_neutral/n
            type
              arrow (?LambdaPi) [not
            d
      inductive_
    theory Induct
    theory Hofsta
    theory Induct
```

```
theory NatExample : ?LFXI =
  inductive nat() ❘ =
    n: type ❙
    z: n ❙
    s: n → n ❙
  ❚

  inductive_definition plusn() : nat() ❘ =
    n: type ❘ = (nat/n → nat/n) ❙
    z: n ❘ = ([x] x) ❙
    s: n → n ❘ = ([u,x] nat/s (u x)) ❙
  ❚
  plus: nat/n → nat/n → nat/n ❘ = nat/plusn/n ❘ # 1 + 2 ❙

  z_plus_n: {m: nat/n} DED ((nat/z + m) EQ m) ❘// = REFL nat/n ❙
  z_plus_z: DED (nat/z + nat/z) EQ nat/z ❘// = z_plus_n nat/z ❙
  z_plus_suc: {m: nat/n} DED (nat/z + m) EQ m → DED (nat/z + (nat/s m)) EQ (nat/s m) ❘// = [m: nat/

  ind_proof z_neutral() : nat() ❘=
    n = [m] DED (nat/z + m) EQ m ❙
    z = z_plus_z ❙
    s = z_plus_suc ❙
  ❚

  inductive_definition times() : nat() ❘=
    n: type ❘ = (nat/n → nat/n) ❙
    z: n ❘ = ([x] nat/z) ❙
    s: n → n ❘ = ([u,x] (x + (u x))) ❙
  ❚
❚

nductivelyDefinedTypes : ?LFXI =
le ?NatExample ❙
tive_definition predec() : nat() ❘=
type ❘ = nat/n ❙
```

```
nat/n→[m]DED (nat/z+m) EQ m

inductive.mmt
  inductive.omdoc
    theory NatExample
      ind_proof z_neutral
        -- elaboration --
          constant nat/z_neutral/n
            type
```

Figure 3: Screenshot of an MMT IDE with an example inductive type, inductively-defined function and proof by induction.

The feature(s) including its implementation is also documented here.

8 Conclusion and outlook

In this paper, we identified inductive definitions as a common basic feature of formal languages for representation of formal mathematics and presented a method to formalize inductive definitions in a formal system assuming only a minimal foundation using structural features in MMT. We implemented this structural feature as an extension of MMT and discussed how inductively-defined functions can be declared for inductive types using this feature. Our design allows for mutual induction, shallow polymorphic types, parametric polymorphism in constructors and as far as possible even constructors with higher-order arguments. Additionally, the design allows for outgoing Term-Levels which allow to conveniently define basic utility functions or additionally required properties (e.g. further axioms in case of algebraic structures) on an inductive type along with the definition of the inductive type itself, which no other system allows up to this point (see table 3.2). An interesting generalization of this is allowing the definition of quotients of inductive types from outgoing Term-Levels stating the relations to quotient out by. Since it also makes the elaboration significantly more complex, we leave this for future work.

Having an structural feature for simple typed inductive types with constructors of only first-order arguments (of inductively-defined types in negative positions), the next step is to look at inductive types with constructors of higher-order arguments. Another further step is trying to give definiens for the external declarations. The main idea to realize these two goals is to elaborate inductive types to the types of well-typed formal expressions over the internal declarations.

This could be realized using so-called quotations, interpreting function types of inductively-defined types as function types defined in the theory of internal declarations.

This is a new approach for implementing inductive types and promises to avoid a limitation currently present in every other formal system.

References

[BC04] Y. Bertot and P. Castéran. *Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[BW99] Stefan Berghofer and Markus Wenzel. "Inductive Datatypes in HOL — Lessons Learned in Formal-Logic Engineering". In: *Theorem Proving in Higher Order Logics*. Ed. by Yves Bertot et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 19–36. ISBN: 978-3-540-48256-7.

[Coq15] Coq Development Team. *The Coq Proof Assistant: Reference Manual*. Tech. rep. INRIA, 2015.

[Cro+95] Judy Crow et al. "A Tutorial Introduction to PVS". In: (1995). URL: `http://www.csl.sri.com/papers/wift-tutorial/`.

[Har96] J. Harrison. “HOL Light: A Tutorial Introduction”. In: *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*. Springer, 1996, pp. 265–269.

[Ian17] Mihnea Iancu. “Towards Flexiformal Mathematics”. PhD thesis. Bremen, Germany: Jacobs University, 2017.

[Joh37] Ingebrigt Johansson. “Der Minimalkalkül, ein reduzierter intuitionistischer Formalismus”. de. In: *Compositio Mathematica* 4 (1937), pp. 119–136. URL: `http://www.numdam.org/item/CM_1937__4__119_0`.

[KR16] Michael Kohlhase and Florian Rabe. “QED Reloaded: Towards a Pluralistic Formal Library of Mathematical Knowledge”. In: *Journal of Formalized Reasoning* 9.1 (2016), pp. 201–234. URL: `http://jfr.unibo.it/article/download/4570/5733`.

[MR19] D. Müller and F. Rabe. “Structuring Theories with Implicit Morphisms”. In: *Recent Trends in Algebraic Development Techniques*. Ed. by J. Fiadeiro and I. Tutu. to appear. Springer, 2019.

[MRK18] D. Müller, F. Rabe, and M. Kohlhase. “Theories as Types”. In: *Automated Reasoning*. Ed. by D. Galmiche, S. Schulz, and R. Sebastiani. Springer, 2018, pp. 575–590.

[NPW02] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, 2002.

[Owr+99] Sam Owre et al. “PVS Language Reference”. In: (Oct. 1999).

[Rab14] Florian Rabe. “How to Identify, Translate, and Combine Logics?” In: *Journal of Logic and Computation* (2014). DOI: `10.1093/logcom/exu079`.

[RK13] Florian Rabe and Michael Kohlhase. “A Scalable Module System”. In: *Information & Computation* 0.230 (2013), pp. 1–54. URL: `http://kwarc.info/frabe/Research/mmt.pdf`.

[RS13] Florian Rabe and Kristina Sojakova. “Logical Relations for a Logical Framework”. In: *ACM Transactions on Computational Logic* (2013). URL: `http://kwarc.info/frabe/Research/RS_logrels_12.pdf`.

[TB85] A. Trybulec and H. Blair. “Computer Assisted Reasoning with MIZAR”. In: *Proceedings of the 9th International Joint Conference on Artificial Intelligence*. Ed. by A. Joshi. Morgan Kaufmann, 1985, pp. 26–28.

Appendix

A Examples of basic inductive types and their elaborations

The following tests (and a few more) are all implemented in MMT and can be found at https://gl.mathhub.info/MMT/examples/blob/master/source/inductive.mmt. All examples typecheck correctly (the elaborations themselves are not typechecked however).

Further definitions for natural numbers

Based on the inductive definition of natural numbers, we can define the following functions by induction:

$inductive\_definition\ predec()\ :\ nat()\ =$

$\quad n:\ \mathsf{type}\ =\ n$

$\quad z:\ n\ =\ z$

$\quad s:\ n\ \to\ n\ =\ ([u]\ u)$

$inductive\_definition\ monus()\ :\ nat()\ =$

$\quad n:\ \mathsf{type}\ =\ (n\ \to\ n)$

$\quad z:\ n\ =\ ([x]\ x)$

$\quad s:\ n\ \to\ n\ =\ ([u,x]\ predec/n\ (ux))$

Inductively defined lists

$inductive\ list(a:\ \mathsf{type})\ =$

$\quad list:\ \{a:\ \mathsf{type}\}\ \mathsf{type}$

$\quad nil:\ \{a:\ \mathsf{type}\}$

$\quad cons: \{a:\ \mathsf{type}\}\ a \to \mathsf{list}\ a \to \mathsf{list}\ a$

is elaborated to:

$list:\ \{a:\ \mathsf{type}\}\ \mathsf{type}$

$nil:\ \{a:\ \mathsf{type}\}\ \mathsf{list}\ a$

$cons:\ \{a:\ \mathsf{type}\}\ a \to \mathsf{list}\ a \to \mathsf{list}\ a$

$injective\_cons\_1:\ \{a:\ \mathsf{type},\ x_{00}: a\ ,\ x_1:\ \mathsf{list}\ a,\ x_{01}:\ a\}$

$$\text{DED } x_{00} \dot{\neq} x_{01} \;\to\; cons\ a\ x_{00}\ x_1 \dot{\neq} s\ a\ x_{01}\ x_1$$

$$injective\_cons\_2 : \{a : \mathsf{type},\ x_{00} : a\ ,\ x_1 : \text{ list } a,\ x_{11} : \text{ list } a\}$$

$$\text{DED } x_{00} \dot{\neq} x_{01} \;\to\; cons\ a\ x_0\ x_{10} \dot{\neq} s\ a\ x_0\ x_{11}$$

$$no\_conf\_cons : \{a : \mathsf{type},\ x_{00} : a,\ x_{10} : \text{ list } a,\ x_{01} : a,\ x_{11} : \text{ list } a\}$$

$$\text{DED } cons\ a\ x_{00}\ x_{10} \dot{\neq} cons\ a\ x_{01}\ x_{11}$$

$$induct/list : \{a : \mathsf{type},\ l' : \mathsf{type} \to \mathsf{type},\ nil' : \{a' : \mathsf{type}\} \text{ list } a',$$

$$cons' : \{a' : \mathsf{type}\}\ a' \to \text{list } a' \to \text{list } a'\}\ List\ a \;\to\; \text{list } a'$$

$$induct/nil : \{a : \mathsf{type},\ l' : \mathsf{type} \to \mathsf{type},\ nil' : \{a : \mathsf{type}\} \text{ list } a,$$

$$\text{DED } cons' : \{a : \mathsf{type}\}\ a \to \text{list } a \to \text{list } a,\ x : a\}$$

$$nil'\ a\ (induct/list\ a\ l'\ nil'\ cons'\ x)' \;\doteq\; induct/list\ a\ l'\ nil'\ cons' nil\ a$$

$$induct/cons : \{a : \mathsf{type},\ l' : \mathsf{type} \to \mathsf{type},\ nil' : \{a : \mathsf{type}\} \text{ list } a,$$

$$cons' : \{a : \mathsf{type}\}\ a \to \text{list } a \to \text{list } a,\ x_0 : a,\ x_1 : \text{ list } a\}$$

$$\text{DED } cons'\ x_0\ (induct/list\ a\ l'\ nil'\ cons'\ a\ x_1)'$$

$$\doteq\; induct/list\ a\ l'\ nil'\ cons'\ (cons\ a\ x_0\ x_1)$$

Then, we can use induction to define the functions:

$$single : \{a : \mathsf{type}\}\ a \to list\ a \;=\; [a : \mathsf{type},\ x : a]\ cons\ x\ nil$$

$$reverse : \{a : \mathsf{type}\}\ list\ a \to list\ a$$

$$= [a,\ l]\ \ induct/list\ list\ a\ ([tp]\ \ nil)$$

$$[tp,\ x,\ xs]\ \ concat\ tp\ (reverse\ xs)\ (single\ x)$$

$$length : \{a : \mathsf{type}\}\ list\ a \to nat$$

$$= [a,\ l]\ \ induct/list\ list\ a\ ([tp]\ \ z)$$

$$[tp,\ x,\ xs]\ \ s\ (length\ xs)$$

Further basic examples

Similarly, we can define the inductively-defined types:

$$inductive\ vector(a : \mathsf{type}) =$$

$$vec : \{m : n\}\ \mathsf{type}$$

$$empty : vec\ z$$

$$add : \{m : n\}\ vec\ m \;\to\; a \;\to\; vec\ (s\ m)$$

$\mathit{inductive}\ \mathit{list}2() =$

$\quad \mathit{list} : \ \mathsf{type} \ \rightarrow \ \mathsf{type}$

$\quad \mathit{nil} : \ \{a : \ \mathsf{type}\} \ \mathit{list}\ a$

$\quad \mathit{cons} : \ \{a : \ \mathsf{type}\} \ a \ \rightarrow \ \mathit{list}\ a \ \rightarrow \ \mathit{list}\ a$

$\quad \mathit{length}2 : \ \{a : \mathsf{type}\} \ \mathit{list}\ a \ \rightarrow \ n$

$\quad \mathit{length2\_nil} : \ \{a : \ \mathsf{type}\} \ \mathrm{DED} \ (\mathit{length}2a(\mathit{nil}\ a) \ \doteq \ z)$

$\quad \mathit{length2\_cons} : \ \{a : \mathsf{type}, \ \mathit{hd} : \ a, \ \mathit{tl} : \ \mathit{list}\ a\}$

$\qquad\qquad\qquad \mathrm{DED} \ \mathit{length}2\ a\ (\mathit{cons}\ a\ \mathit{hd}\ \mathit{tl}) \ \doteq \ s\ (\mathit{length}2\ a\ \mathit{tl})$

$\mathit{inductive}\ \mathit{list}3() =$

$\quad \mathit{list} : \ \mathsf{type} \ \rightarrow \ \mathsf{type}$

$\quad \mathit{nil} : \ \{a : \ \mathsf{type}\} \ \mathit{list}\ a$

$\quad \mathit{cons} : \ \{a : \ \mathsf{type}\} \ a \ \rightarrow \ \mathit{list}\ a \ \rightarrow \ \mathit{list}\ a$

$\quad \mathit{head} : \ \{a : \ \mathsf{type}, \ l : \mathit{list}\ a\} \ \mathrm{DED} \ (l \not\doteq \mathit{nil}\ a) \ \rightarrow \ a$

$\quad \mathit{head\_cons} : \ \{a : \ \mathsf{type}, \ \mathit{hd} : \ a, \ \mathit{tl} : \ \mathit{list}\ a,$

$\qquad\qquad\qquad P : \ \mathrm{DED} \ ((\mathit{cons}\ a\ \mathit{hd}\ \mathit{tl}) \not\doteq \mathit{nil}\ a)\}$

$\qquad\qquad\qquad \mathrm{DED} \ (\mathit{head}\ a\ (\mathit{cons}\ a\ \mathit{hd}\ \mathit{tl})\ P) \ \doteq \ \mathit{hd}$

The first one now allows us to define the inductively defined function *zip* assuming that product types are defined:

$\mathit{product\_type} : \ \mathsf{type} \ \rightarrow \ \mathsf{type} \ \rightarrow \ \mathsf{type} \ \# \ 1 * 2 \ = \ [a, \ b] \ \mathit{pair/type}\ a\ b$

$\mathit{record\_term}\ \mathit{pairs}(A : \ \mathsf{type}, \ B : \ \mathsf{type}, \ x : \ A, \ y : \ B) \ : \ \mathit{pair}(A, \ B) \ =$

$\quad a : \ A \ = \ x$

$\quad b : \ B \ = \ y$

$\mathit{pair\_term} : \ \{a : \ \mathsf{type}, \ b : \ \mathsf{type}\} \ a \ \rightarrow \ b \ \rightarrow \ a * b \ = \ [a, \ b, \ x, \ y] \ \mathit{pairs/make}\ a\ b\ x\ y$

$\mathit{inductive\_definition}\ \mathit{zip}(a : \ \mathsf{type}, \ b : \ \mathsf{type}) \ : \ \mathit{vector}(a, \ b) \ =$

$\quad \mathit{vec} : \ \{m : \ n\} \ \mathsf{type} \ = \ [m] \ \mathit{vec}\ b\ m \ \rightarrow \ \mathit{vec}\ (a * b)\ m$

$\quad \mathit{empty} : \ \mathit{vec}\ \mathit{nat}/z \ = \ [l] \ \mathit{empty}\ (a * b)$

$$add : \; \{m : \; n\} \; vec \; m \; \rightarrow \; a \; \rightarrow \; vec \; (s \; m) \; =$$

$$[m, \; xs, \; x] \; induct/vec \; b \; ([t, \; p] \; vec \; (a * t) \; p) \; ([t] \; empty \; (a * t)) \; ([t, \; p, \; ys, \; y]$$

$$add \; (a * t) \; p \; ys \; (pair\_term \; a \; t \; x \; y)) \; (s \; m)$$

Free Monoid
Finally, a somewhat more interesting example is:

$$inductive \; FM(a : \; \mathsf{type}) =$$

$$fm : \; \mathsf{type}$$

$$gen : \; a \; \rightarrow \; fm$$

$$unit : \; fm$$

$$comp : \; fm \; \rightarrow \; fm \; \rightarrow \; fm$$

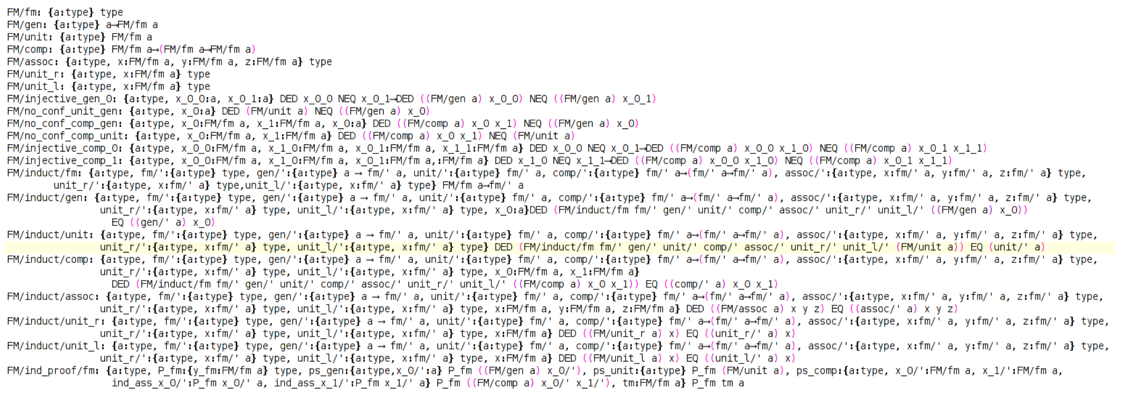
$$assoc : \; \{x : \; fm, \; y : \; fm, \; z : \; fm\}$$

$$\text{DED} \; comp \; (comp \; x \; y) \; z \; \doteq \; comp \; x \; (comp \; y \; z)$$

$$unit\_r : \; \{x : \; fm\} \; \text{DED} \; comp \; x \; unit \; \doteq \; x$$

$$unit\_l : \; \{x : \; fm\} \; \text{DED} \; comp \; unit \; x \; \doteq \; x$$

its elaboration is given in figure 4.



```
FM/fm: {a:type} type
FM/gen: {a:type} a→FM/fm a
FM/unit: {a:type} FM/fm a
FM/comp: {a:type} FM/fm a→(FM/fm a→FM/fm a)
FM/assoc: {a:type, x:FM/fm a, y:FM/fm a, z:FM/fm a} type
FM/unit_r: {a:type, x:FM/fm a} type
FM/unit_l: {a:type, x:FM/fm a} type
FM/injective_gen_0: {a:type, x_0_0:a, x_0_1:a} DED x_0_0 NEQ x_0_1→DED ((FM/gen a) x_0_0) NEQ ((FM/gen a) x_0_1)
FM/no_conf_unit_gen: {a:type, x_0:a} DED (FM/unit a) NEQ ((FM/gen a) x_0)
FM/no_conf_comp_gen: {a:type, x_0:FM/fm a, x_1:FM/fm a, x_0:a} DED ((FM/comp a) x_0 x_1) NEQ ((FM/gen a) x_0)
FM/no_conf_comp_unit: {a:type, x_0:FM/fm a, x_1:FM/fm a} DED ((FM/comp a) x_0 x_1) NEQ (FM/unit a)
FM/injective_comp_0: {a:type, x_0_0:FM/fm a, x_1_0:FM/fm a, x_0_1:FM/fm a, x_1_1:FM/fm a} DED x_0_0 NEQ x_0_1→DED ((FM/comp a) x_0_0 x_1_0) NEQ ((FM/comp a) x_0_1 x_1_1)
FM/injective_comp_1: {a:type, x_0_0:FM/fm a, x_1_0:FM/fm a, x_0_1:FM/fm a,:FM/fm a} DED x_1_0 NEQ x_1_1→DED ((FM/comp a) x_0_0 x_1_0) NEQ ((FM/comp a) x_0_1 x_1_1)
FM/induct/fm: {a:type, fm/':{a:type} type, gen/':{a:type} a → fm/' a, unit/':{a:type} fm/' a, comp/':{a:type} fm/' a→(fm/' a→fm/' a), assoc/':{a:type, x:fm/' a, y:fm/' a, z:fm/' a} type,
        unit_r/':{a:type, x:fm/' a} type,unit_l/':{a:type, x:fm/' a} type} FM/fm a→fm/' a
FM/induct/gen: {a:type, fm/':{a:type} type, gen/':{a:type} a → fm/' a, unit/':{a:type} fm/' a, comp/':{a:type} fm/' a→(fm/' a→fm/' a), assoc/':{a:type, x:fm/' a, y:fm/' a, z:fm/' a} type,
                unit_r/':{a:type, x:fm/' a} type, unit_l/':{a:type, x:fm/' a} type, x_0:a}DED (FM/induct/fm fm/' gen/' unit/' comp/' assoc/' unit_r/' unit_l/' ((FM/gen a) x_0))
                  EQ ((gen/' a) x_0)
FM/induct/unit: {a:type, fm/':{a:type} type, gen/':{a:type} a → fm/' a, unit/':{a:type} fm/' a, comp/':{a:type} fm/' a→(fm/' a→fm/' a), assoc/':{a:type, x:fm/' a, y:fm/' a, z:fm/' a} type,
                unit_r/':{a:type, x:fm/' a} type, unit_l/':{a:type, x:fm/' a} type} DED (FM/induct/fm fm/' gen/' unit/' comp/' assoc/' unit_r/' unit_l/' (FM/unit a)) EQ (unit/' a)
FM/induct/comp: {a:type, fm/':{a:type} type, gen/':{a:type} a → fm/' a, unit/':{a:type} fm/' a, comp/':{a:type} fm/' a→(fm/' a→fm/' a), assoc/':{a:type, x:fm/' a, y:fm/' a, z:fm/' a} type,
                unit_r/':{a:type, x:fm/' a} type, unit_l/':{a:type, x:fm/' a} type, x_0:FM/fm a, x_1:FM/fm a}
                  DED (FM/induct/fm fm/' gen/' unit/' comp/' assoc/' unit_r/' unit_l/' ((FM/comp a) x_0 x_1)) EQ ((comp/' a) x_0 x_1)
FM/induct/assoc: {a:type, fm/':{a:type} type, gen/':{a:type} a → fm/' a, unit/':{a:type} fm/' a, comp/':{a:type} fm/' a→(fm/' a→fm/' a), assoc/':{a:type, x:fm/' a, y:fm/' a, z:fm/' a} type,
                unit_r/':{a:type, x:fm/' a} type, unit_l/':{a:type, x:fm/' a} type, x:FM/fm a, y:FM/fm a, z:FM/fm a} DED ((FM/assoc a) x y z) EQ ((assoc/' a) x y z)
FM/induct/unit_r: {a:type, fm/':{a:type} type, gen/':{a:type} a → fm/' a, unit/':{a:type} fm/' a, comp/':{a:type} fm/' a→(fm/' a→fm/' a), assoc/':{a:type, x:fm/' a, y:fm/' a, z:fm/' a} type,
                unit_r/':{a:type, x:fm/' a} type, unit_l/':{a:type, x:fm/' a} type, x:FM/fm a} DED ((FM/unit_r a) x) EQ ((unit_r/' a) x)
FM/induct/unit_l: {a:type, fm/':{a:type} type, gen/':{a:type} a → fm/' a, unit/':{a:type} fm/' a, comp/':{a:type} fm/' a→(fm/' a→fm/' a), assoc/':{a:type, x:fm/' a, y:fm/' a, z:fm/' a} type,
                unit_r/':{a:type, x:fm/' a} type, unit_l/':{a:type, x:fm/' a} type, x:FM/fm a} DED ((FM/unit_l a) x) EQ ((unit_l/' a) x)
FM/ind_proof/fm: {a:type, P_fm:{y_fm:FM/fm a} type, ps_gen:{a:type,x_0/':a} P_fm ((FM/gen a) x_0/'), ps_unit:{a:type} P_fm (FM/unit a), ps_comp:{a:type, x_0/':FM/fm a, x_1/':FM/fm a,
                  ind_ass_x_0/':P_fm x_0/' a, ind_ass_x_1/':P_fm x_1/' a} P_fm ((FM/comp a) x_0/' x_1/'), tm:FM/fm a} P_fm tm a
```

Figure 4:   The elaboration of the inductively defined free monoid.

Further examples (both directly using the external declarations and using the utility features) can be found in this repository.