# Internship report MPRI 2 Reverse engineering on arithmetic proofs

François Thiré



# Internship report – MPRI 2
# Reverse engineering on arithmetic proofs

François Thiré
supervised by Gilles Dowek (ENS Cachan - LSV) &
Stéphane Graham-Lengrand (Ecole polytechnique - LIX)

August 21, 2016

## The general context

Computer science and more specifically programming activities come with *bugs*. However, today, more and more programs are used in critical applications in transportation, energy and health. Proof checkers and automatic provers have been developed to prove that programs are correct. Nowadays, dozens of such tools exist. As programming languages, each tool has its own (dis)advantages. Since proofs are built using a logic, each proof checker is based upon a logic. But some logics may be more powerful than others. To overcome this, some proof checker – called *logical framework* – can embed several different logics. Logical frameworks have the advantage to separate the logic from the proof. This allows to identify which part of the logic is used in the proof and where.

But writing a proof that is correct in a proof checker is something tedious. There are proof assistants that help by using a tactic system for example. Otherwise some theorems might be proved automatically with automatic provers. But when one wants to prove a theorem with an automatic prover, this latter generally only gives an answer without any proof. It is therefore natural for such automatic provers to wonder if it is safe to trust an automatic prover? After all, it is only a program and programs have bugs...

## The research problem

DEDUKTI is a logical framework that implements the $\lambda\Pi-$modulo theory, an extension of the simply typed lambda calculus with dependent types and rewriting rules. It aims to be a back-end for other proof checkers by compiling proofs from these proof checkers to DEDUKTI. This may also increase re-usability of proofs between proof checkers. However if a logic is more powerful than an other, a theorem in the first logic may not be a theorem in the second.

During this internship, we consider arithmetic theorems since many proof checker are able to check arithmetic proofs. One problem that we study in this master thesis is to translate arithmetic proofs coming from a powerful proof checker,– in our case MATITA – to a less powerful proof checker – HOL –. This translation needs to modify the logic used in proofs and that is why DEDUKTI is handy here.

But a lot of arithmetic theorems are proved also by automatic provers. Indeed, today a lot of easy arithmetic theorems are proved by this kind of tool. But most of them do not give a proof if it claimes to prove a theorem. Since for these kind of tool, constructing a full proof may be tiresome, they prefer to give a *certificate*, a sketch of a proof. However, any automatic prover can implement its own certificate format. To answer this problem, Zakaria Chihani & Dale Miller proposed a certificate framework: Foundational Proof Certificate (FPC) [CMR13]. This framework aims to provide a certificate format shared by many automatic provers so that from the latter, a full proof might be reconstructed. However, for now, no certificate format is given for arithmetic proofs. A second problem addressed in this internship is to answer what kind of certificate is needed for *linear* arithmetic proofs (arithmetic without multiplication).

### My contribution

Translating proofs from MATITA to HOL requires to do several transformations on them. We have identified four *features* particular to MATITA that have to be removed so that proofs may be after translated into HOL. On these four features, two have been completely removed, the two others are left for future work. Some transformation on proofs have been made manually, but generally it can be automated. This led me to develop new tools like *universo* and *deduktipli* that transform proofs into others.

We also give a certificate format for *linear* arithmetic proofs and give an answer on how to reconstruct a proof from this certificate format by giving a full proof in sequent calculus.

### Arguments supporting its validity

On the translation of MATITA proofs to HOL, two features of MATITA have been fully removed inside the proofs. This increase our confidence on the possibility to remove the two others one. Moreover, tools that have been developed are quite general so that I think it should be possible to use it for other systems implemented in other proof checkers if they can be translated in DEDUKTI.

Besides, the new certificate format proposed in this report is very simple so that any automatic provers should be able to provide one, and also allows to rebuild a full proof from it.

### Summary and future work

Translating arithmetic proofs from MATITA to HOL is a process that combines manual transformations and automatic transformations. It would be interesting to know if the actual manual transformations may be automated. Besides, the tools that I have developed so far are for the moment experimental. They can be extended in different ways. Of course, MATITA is an example here and the generality of these methods remains to be investigated. It also remains to implement the new certificate format in CHECKERS, the tool that implement the FPC developed in the Parsifal team.

# 1 Introduction

Proving theorems in mathematics is generally a hard task. Moreover, with long proofs, the chance that the researcher has made an error is high. To overcome these problems, there are two kinds of tools:

- proof checkers and proof assistants[1] that allows the user to write a proof and the computer automatically checks if the proof is correct,
- automatic provers where the user just enter the theorem and waits for an answer. Generally either *yes*, *no* or *I don't know*.

A proof checker allows to **formally** check that a proof is correct. Formally, because proofs and theorems have to be written in a specific language so that an algorithm is able to check automatically the correction of a proof.

Nowadays, a lot of programming languages have been developed. The reason is not because one may be more powerful[2] than an other, but it is rather because that mechanisms of a language allow users to program more efficiently. The same phenomenon exists with proof checkers but contrary to programming languages, some theorems may be provable in one proof checker and not in another. This depends on the theory (and the logic) behind the proof checker. Indeed, there is no notion of *universal* logic. Therefore, theorems provable depends on the logic that is used. This leads to the development of *logical frameworks*, that are proof checkers that may embed different logics. Section 2 is devoted to the presentation of proof checkers used during this master thesis.

However, each proof checker comes with its own library (as for a programming language) and it is difficult to reuse a proof from one proof checker to an other. Yet, developing a proof might be something tedious that we do not want to do for each proof checker. Indeed, some proofs needed hundreds of man-month to be formally encoded into a proof checker like the *four color theorem* or the *Feit-Thompson theorem* [Gon07] [Gon13]. Therefore it would be nice – when possible – to be able to translate a proof from one proof checker to another.

When one proof checker is less powerful than an other like HOL with COQ, this has already be made in [KW10]. However, what about the opposite? Of course, for some proofs it is not possible, but for most of them it should be. In section 3, we focus on the translation of arithmetic proofs developed in a proof checker named MATITA to another one named HOL.

The use of DEDUKTI will be handy here for several reasons:

- it is a logical framework, so it separates the logic from the proofs,
- it is a very simple system,
- it has been created for this purpose.

One problem that arises with proof checkers is that some theorems might not be hard to prove but quite tiresome, especially for arithmetic proofs. Automatic provers are tools that allow to (dis)prove theorems automatically. Instead of building directly a proof, they implement algorithms and heuristics allowing to refute or not the initial statement. However, one problem with these tools is that most of them do not justify their answer by giving a proof. So how one can trust their result? Should they give a full proof? And in which formalism if that so? In general, an automatic prover will only supply a certificate in its own format. The use of a Foundational Proof Certificate (FPC) have been proposed in [CMR13]. This framework define certificates format so that it is easy for an automatic prover to elaborate one and so that it is also easy to reconstruct the proof from this certificate. In general, there is a trading to do: a simple proof implies a long certificate and vice versa.

[1] From now on, proof checkers will refer to as both proof assistants and proof checkers since the former generally implement its own proof checker

[2] in the sense that more functions are expressible

However for the moment, FPC support only first-order logic and some extensions have been proposed [CM16]. We explore in section 4 the problem of adding *linear* arithmetic in this framework.

# 2 Proof Checkers

This internship uses several formal systems implemented in different proof checkers. This section aims to provide a brief summary of the systems that will be used later in this report. Proof checkers detailed in this section use the proposition-as-type principle. So that a theorem corresponds to a type and its proof is a function that implements this type in the formal language used by the proof checker. Therefore, the correction of a proof in the formal system comes back to type check the proof. Most of proof checkers are buit around a kernel that implements only a type checker.

## 2.1 DEDUKTI

DEDUKTI is an implementation of the $\lambda\Pi$-modulo theory introduced by Cousineau & Dowek in [CD07].

$\lambda\Pi$-modulo theory is an enhanced version of the $\lambda\Pi$ calculus where rewriting rules are added.

### The $\lambda\Pi$ calculus

The $\lambda\Pi$ calculus is a $\lambda$ calculus with dependent types. A dependent type is a type that contains values. For example, the type of the function that maps a number $n$ to the type of vectors of size $n$ is expressed as:

$$\Pi\ (n : nat).\ (vector\ n)$$

The consequence is that since the type of vectors depends on a value –here a natural number–, then *vector* 1 and *vector* 2 are two different types. $\Pi$ is a binder (like $\lambda$) that allows to index family of types by abstracting over values or types.

The syntax of terms of the $\lambda\Pi$ calculus is given below:

$\langle sort \rangle ::= \texttt{Type} \mid \texttt{Kind}$

$\langle term \rangle ::= \langle sort \rangle \mid x \mid \langle term \rangle\ \langle term \rangle \mid \lambda x : \langle term \rangle.\ \langle term \rangle \mid \Pi x : \langle term \rangle.\ \langle term \rangle$

$\langle context \rangle ::= \emptyset \mid \langle context \rangle$ ',' $x : \langle term \rangle$

Notice that types and terms are mixed together, this is a side effect of dependent types since values may appear inside types.

The typing derivation rules are presented in Figure 1. With dependent types, a context might be ill-formed. Therefore, we need a judgment for well-formed contexts.

### Rewriting on terms

The denomination modulo in the $\lambda\Pi-$calculus modulo theory comes from that we add rewriting rules. So first, we need to extend contexts so that they include rewriting rules:

$\langle context \rangle ::= \ldots \mid \langle term \rangle \longrightarrow \langle term \rangle$

However, if one allows any rule, the convertibility test will be undecidable and therefore type checking will be also undecidable. Therefore, one needs to *type check* a rewriting rule before adding it to the system to maintain the decidability of the system. This is expressed with the following judgment:

$$\Gamma \vdash l \longrightarrow r$$

$$\frac{}{\emptyset \text{ well-formed}}\ (\text{Empty}) \qquad \frac{\Gamma \vdash A : s}{\Gamma, x : A \text{ well-formed}}\ (\text{Decl}) \qquad \frac{\Gamma \text{ well-formed}}{\Gamma \vdash \texttt{Type} : \texttt{Kind}}\ (\text{Type})$$

$$\frac{\Gamma \text{ well-formed} \qquad (\text{when } (x : A) \in \Gamma)}{\Gamma \vdash x : A}\ (\text{Var}) \qquad \frac{\Gamma \vdash A : \texttt{Type} \qquad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Pi(x : A).\ B : s}\ (\text{Prod})$$

$$\frac{\Gamma \vdash A : \texttt{Type} \qquad \Gamma, x : A \vdash B : s \qquad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda(x : A).\ t : \Pi(x : A).\ B}\ (\text{Prod}) \qquad \frac{\Gamma \vdash t : (\Pi(x : A).\ B) \qquad \Gamma \vdash t' : A}{\Gamma \vdash (t\ t') : \left\{{}^{t'}/_{x}\right\} B}\ (\text{App})$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A : s \qquad \Gamma \vdash B : s \qquad (\text{when } A \equiv_\beta B)}{\Gamma \vdash t : B}\ (\text{Conv})$$

Figure 1: $\lambda\Pi$-calculus

| Dedukti | $\lambda\Pi-$calculus modulo | Syntactic Construct |
|---|---|---|
| `x:A => b` | $\lambda x : A.b$ | Abstraction |
| `x : A -> b` | $\Pi x : A.b$ | Product |
| `A -> B` | $A \to B$ | Arrow Type |
| `def x : A.` | $x : A$ | Definable symbol declaration |
| `x : A.` | $x : A$ | Static symbol declaration |
| `[x_1,...,x_n] l -> r.` | $l \longrightarrow r$ | Rewrite Rule |

Figure 2: Dedukti syntax

But defining precisely this judgment is difficult. Details on how to infer such judgment is defined in [Sai15]. Especially, since higher order unification is undecidable, only Miller patterns [Mil91] are allowed on the left side of the rules (roughly such pattern is a symbol applied to some arguments).

So, we add a rule for deriving well-typed contexts:

$$\frac{\Gamma \text{ well-formed} \qquad \Gamma \vdash l \longrightarrow r}{\Gamma, l \longrightarrow \text{ well-formed}}\ (\text{Rule})$$

Finally, the convertibility test is now enhanced with rewriting rules so that $\equiv_{\beta\Gamma}$ refers to the congruence induced by $\beta-$reduction and by the rewriting rules in $\Gamma$:

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : s \qquad A \equiv_{\beta\Gamma} B}{\Gamma \vdash t : B}\ (\text{Conv})$$

Since we are going to write some DEDUKTI code, we give in Figure 2 a small description of the syntax used in DEDUKTI. Notice that DEDUKTI makes a difference between *definable* symbols and *static* symbols. The difference is that the head symbol of a rewrite rule have to be definable. This information is used by DEDUKTI to deduce that static symbol are injectives since they cannot appear at the head of a rewrite rule.

The following example shows how to write in DEDUKTI the function *plus*:

```
nat : Type.
0 : nat.
S : nat -> nat.

def plus : nat -> nat -> bat.
[y] plus 0 y --> y.
[x,y] plus (S x) y --> S (plus x y).
```

## 2.2 COQ/ MATITA

COQ and MATITA are two proofs checker based on the Calculus Of Inductive Constructions (CiC). CiC is composed of the Calculus Of Constructions (CoC) plus inductive types. CoC is no more than the $\lambda\Pi-$calculus with polymorphism and type constructors. That is, to the rules of figure 1, we add the following rules

$$\frac{\Gamma \vdash A : \texttt{Kind} \qquad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Pi(x : A).\ B : s}\ (\text{Prod})$$

These rules allow to add polymorphism and type constructors in the system. Polymorphism allows to express functions of type:

$$\forall \alpha, \alpha \to \alpha$$

where $\alpha$ is a sort, and so its type is `Kind` while type constructors allow to construct the type of lists parameterized by the type of its elements. Indeed, it is now possible to construct the type $\texttt{Type} \to \texttt{Type}$, and we can declare a variable *list* of type $\texttt{Type} \to \texttt{Type}$ and so build the types *list nat*, *list bool*, etc.

## 2.3 Inductive types

Inductive types are built with three kinds of rules:

- constructors
- a recursor rule
- elimination rules

One of the most famous inductive type is the natural numbers. Its constructors are:

$$\frac{}{0 : \mathbb{N}}\ (\text{Zero}) \qquad\qquad \frac{}{S : \mathbb{N} \to \mathbb{N}}\ (\text{Succ})$$

its recursor rule is:

$$\frac{\Gamma \vdash P : \mathbb{N} \to \texttt{Type} \qquad \Gamma \vdash a : P\ 0 \qquad \Gamma \vdash f : \Pi(x : \mathbb{N}).\ P\ x \to P\ (S\ x)}{\Gamma \vdash \mathbf{R}_{\mathbb{N}}\ P\ a\ f : \Pi(n : \mathbb{N}).\ P\ n)}\ (\mathbb{N}_{rec})$$

and its elimination rules[3] are:

$$\frac{(\mathbf{R}_{\mathbb{N}}\ P\ a\ f\ 0) : \mathbb{N}}{a : \mathbb{N}}\ (\mathbf{R}_0) \qquad\qquad \frac{(\mathbf{R}_{\mathbb{N}}\ P\ a\ f\ (S\ x)) : \mathbb{N}}{(f\ x\ (\mathbf{R}_{\mathbb{N}} P\ a\ f\ x)) : \mathbb{N}}\ (\mathbf{R}_S)$$

The recursor allows to write inductive definitions on type. For example, the function *plus* may be defined as

$$plus : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \equiv \lambda(x : \mathbb{N}).\ (\mathbf{R}_{\mathbb{N}}\ (\Pi(x : \mathbb{N}).\ \mathbb{N})\ x\ (\lambda x\ \lambda y\ (S\ y)))$$

In COQ and in MATITA, the recursor rule and the elimination rules are added automatically from the constructors.

```
type : Type.
o : type.
i : type.

arrow : type -> type -> type.
def eta : type -> Type.

[x,y] eta (arrow x y) --> eta x -> eta y.

impl : eta o -> eta o -> eta o.

forall : a:type -> ((eta a) -> (eta o)) -> eta o.

def eps : (eta o) -> Type.

[x,y] eps (impl x y) --> eps (x) -> (eps y).

[a, p] eps (forall a p) --> x:(eta a) -> eps (p x).
```

Figure 3: HOL in DEDUKTI

### 2.4 HOL

As it is suggested by its name, HOL is a logic that allows quantification at all orders. It has been implemented in several proof assistants like HOL LIGHT and HOL 4, to name a few. In HOL, terms have only two sorts that are the basic types $\iota$ and the logical propositions $o$. We give in Figure 3 an embeding of HOL in DEDUKTI.

We now give a small explanation of this embedding. The symbol `type` represents HOL types in DEDUKTI. An encoding of the identify function $\lambda x.x$ needs to be able to express arrow types in HOL. However, `arrow i i` is not a type of DEDUKTI, therefore we need a lift operator `eta` that transport types of HOL to types of DEDUKTI.

But the following term is ill-typed since `eta(i)` $\to$ `eta(i)` is different from `arrow i i`.

```
def id : eta(arrow i i) := (x:eta i) => x
```

This justify the first rewriting rule. Then to encode the proposition

$$\forall x, x \Rightarrow x$$

one needs to add the $\forall$ connective named `forall` and the $\Rightarrow$ connective named `impl`.

But one more time, checking the term

```
def id_prop : eta(forall o (x => impl x x)) := (x:eps prop) => x
```

implies to add the two other rewriting rules. From this simple and minimal logic, every other usual logical connectives may be defined.

### 2.5 CHECKERS

CHECKERS has been developed in the purpose to check proofs coming from automatic provers. CHECKERS is implemented in $\lambda-$prolog and is based on the FPC using a classical focused calculus [CMR13].

A focused proof system is a calculus that allows to reduce the search space while searching for a proof of a theorem. It was first invented for linear logic by Andreoli in [And92] but it has been extended to

[3] In practice, elimination rules are implemented as rewriting rules and not deductive rules

constructive logic and classical logic by Liang & Miller in [LM09]. A focused system alternates between *asynchronous* and *synchronous* phases.

Asynchronous phases are made of inference rules that are invertible. In sequent calculus, such rules are for example

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \ (\wedge \text{ left}) \qquad \frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \vee B, \Delta} \ (\vee \text{ right})$$

But if we take the right existential rule

$$\frac{\Gamma \vdash \left\{{}^{t}/_{x}\right\} A, \Delta}{\Gamma \vdash \exists x.A, \Delta} \ \exists \text{ Right}$$

it is not invertible since the goal may be true but not the premise. In particular, guessing the good $t$ may be difficult. Such information could be find inside a certificate for example.

Therefore, CHECKERS only apply invertible rules during the asynchronous phase while during the synchronous phase, CHECKERS may use (if needed) information from the certificate.

CHECKERS is built around a kernel that only implements a focusing framework named LKU with a slightly modification. To each rule, a predicate is added as premise:

- on invertible rules, this predicate is called a *clerk*,
- on non-invertible rules, this predicate is called an *expert*.

Experts predicate are the one that may extract information from the certificate while clerks only manage some bookkeeping. The LKU system may be found in [LM11].

Supporting a new certificate only asks to modify experts behavior to take into account this new format. If the expert is badly implemented, this will not produce an incorrect proof since experts are just helpers. So the correctness of the system is unchanged, it is proved once and for all.

## 3 From MATITA to HOL using DEDUKTI

MATITA is a proof checker based on the CiC while HOL is just an acronym for Higher-Order-Logic. But since Higher-Order-Logic is strictly embedded in the CiC, some proofs of MATITA can not be expressed in HOL. However, most of the proofs written in MATITA do not *need* all of the features of the CiC even if the proofs use these features. For example, we shall see that every terms in MATITA uses a feature called "universes". However, in HOL there is no universe and still it is possible to write arithmetic proofs. So it is reasonable to believe that the concept of universe is not necessary to construct arithmetic proofs.

Since it is possible to prove arithmetic theorems in HOL, there must be a way to express arithmetic proofs of MATITA in HOL. In order to do so, we use DEDUKTI.

DEDUKTI is an handy tool for this, since it allows us to separate the logic from the proofs themselves. Because we are going to modify the logic, then it will be easier to identify which parts of the logic are used and to locate where inside a proof term.

In [Ass15], *Ali Assaf* proves that CiC can be encoded in the $\lambda\Pi-$modulo theory adding rewrites rules to encode eliminator rules and recursive definitions. This encoding is implemented in KRAJONO that transforms proofs and theorems from MATITA to DEDUKTI. So most of the work of this internship consists in transforming these proofs from DEDUKTI to HOL. If `A.ma` is a MATITA proof file, then KRAJONO will produce a file `A.dk`. Since there is a separation between proofs and the logic, the file `A.dk` will depend on another file that is `cic.dk` which is a hand-written file that contains an encoding of the CiC in DEDUKTI( [Ass15] [CD07]). This encoding is provided in Appendix C.

A first goal is for every proof file that comes from MATITA, change its dependency so that the file no longer depends on `cic.dk` but on `hol.dk` that is an other hand-written file that contains an encoding of HOL in DEDUKTI(the samme as in Figure 3). A second goal will be then to transform these proofs from DEDUKTI to HOL.

In order to achieve the first goal, there is 4 features of MATITA that we need to remove/transform:

1. universes
2. propositional dependents products
3. inductive types and recursive functions (on any inductive type)
4. logical connectives

In this master internship we have focused on the first and the second objectives. We have started investigating the third and the fourth but they are left for future work.

## 3.1 The MATITA arithmetic library

Using KRAJONO, it is possible to translate the MATITA arithmetic library into DEDUKTI. This constitute about 30Mo of proofs spread in 30 files. Among these 30 files, we use 20 of them that contains classical theorems on arithmetic library. The figure 5 in Appendix B shows the dependency between these files.

Since some proofs might not be expressible in HOL or because it might be tedious to do so, we are going to cut some proofs (or even files among them). However, we set an intermediate goal that is to proof the Fermat's little theorem in HOL from the proof in MATITA.

## 3.2 Removing universes

The development of the MATITA arithmetic library expressed in DEDUKTI uses seven universes. However, in HOL, such universes do not exist. We explore in the following paragraphs how it is possible to suppress universes from MATITA proofs.

### Universes in Matita

Universes of MATITA in DEDUKTI are expressed this way

```
Sort : Type.

z: Nat.
s : Nat -> Nat.

prop : Sort.
type : Nat -> Sort.
```

A universe is a term of type `Sort` and it has the form `prop` or `type n` for every natural number `n`.

Seven universes are used in the MATITA arithmetic library. But we know that to express these proofs in HOL, only two are necessary : `prop` and `type 0`. Therefore, one need to plane DEDUKTI terms to have only these two universes.

Unfortunately, there is no easy way to do this. One may try to replace each universe `type (n+1)` by `type n` but eventually, there is going to have a type checking issue.

To solve this problem, I have built a tool named UNIVERSO. This tool takes a DEDUKTI file using the CiC logic and *tries* to minimize the number maximal of universes needed. This tool represents about 500 hundred lines of Ocaml and has been developed in less than a week. However, it makes the assumption that the maximal number of universes needed is two.

The tool procedes in four steps:

| convertibility issue | constraints generated |
|---|---|
| $?1 \stackrel{?}{=} ?2$ | $?1 = ?2$ |
| $?1 \stackrel{?}{=} \text{succ } ?2$ | $?2 < ?1$ |
| $?1 \stackrel{?}{=} \text{rule } ?2\ ?3$ | $?1 = ?3$ |

Table 1: constraints generated

1. First, replace each terms of type `sort` by a universe variable except for `prop`
2. Second, generate some constraints on these variables
3. Third, solve these constraints
4. Fourth, replace the universe variable by its solution

Since universes is a hierarchy indexed by natural number, the solver assign a natural number to each universe variable. And so, the constraints might use the natural order on natural numbers. Two types of constraints are used :

- $uv_1 = uv_2$
- $uv_1 < uv_2$

Typing terms is obviously needed to generate the constraint, that is why UNIVERSO uses DEDUKTI to generate the constraints.

We list here the other *constructors* of type `sort`.

```
def succ : Sort -> Sort.
[] succ prop --> type z.
[i : Nat] succ (type i) --> type (s i).

def rule : Sort -> Sort -> Sort.
[s1 : Sort] rule s1 prop --> prop.
[s2 : Sort] rule prop s2 --> s2.
[i : Nat, j : Nat] rule (type i) (type j) --> type (m i j).

def max : Sort -> Sort -> Sort.
[s1 : Sort] max s1 prop --> s1.
[s2 : Sort] max prop s2 --> s2.
[i : Nat, j : Nat] max (type i) (type j) --> type (m i j).
```

The first step replaces universes constructors by a universe variable.

After the first step, DEDUKTI terms are ill-typed since all information on universes is lost. But, we can use the convertibility test of DEDUKTI to recover these information.

The type-checker of DEDUKTI raises a typing error when it can not convert two terms. So UNIVERSO *bypass* the convertibility test by adding some cases.

For example, suppose that DEDUKTI tries to convert $uv_1$ with $uv_2$, then UNIVERSO adds a case returning *OK*, but adding the constraint $uv_1 = uv_2$. The other constraints are detailed in table 1. One may notice that no constraint are generated for the constructor `max`. This is because DEDUKTI can always use one rewriting rule on `max` since `prop` is not replaced by a universe variable.

One may notice that for the constructor `rule`, the constraint generated is false, but it is true if there are at most two universes. This is one reason that explains why UNIVERSO is not correct in general.

If at the end there exists $i$ and $j$ such that $uv_i = uv_j$ and $uv_i < uv_j$, then we get an inconsistency. Otherwise, there is a solution.

Since we are in the hypothesis that there is at most two universes, the constraint ?1 <?2 is interpreted by $S?1 = ?2$.

Despite of these approximation, UNIVERSO applied to the MATITA arithmetic library produces files that are well-typed, well almost!

First, I got an inconsistency. I had to change a lemma used in a proof. The reason is because the proof used an inductive principle of equality on the sort `type 0` but now, when minimizing universes, only `prop` was needed.

Second, the result files did not type check because only three times, there was an occurrence of a third universe. Since my generation of constraint was inconsistent with three universes, it is not surprising. However, it was sufficient to modify by hand these terms so that they well-type. The idea was to replace by hand some occurrence of `type 0` by `type 1` were it was needed. This occured when the constructor `rule` was used to type check the file.

On this three terms, two occurs inside a rewriting rule. And it was possible to suppress this third universe by modifying the rule:

If for example there were a symbol $A$ such that:

```
def A : Sort -> Type -> Type.
[x,y] A x y --> y.
```

and that this symbols occurs this way :

```
def B : Type -> Type.
B n --> A (type 1) n.
```

Then, it is possible to replace this last rule by

```
B n --> n.
```

For the third term, it was the proof of a theorem, and I decided to transform the theorem into an axiom for now. This theorem expresses a property on *big operators* like $\Sigma$ and $\Pi$ expressing that

$$\Sigma_{i\in[a;b]\wedge p(i)} f(i) = \Sigma^{b}_{a\leq i\wedge p(i)} f(i)$$

So finally, we get a library in which there is only two universes at most. For the moment, UNIVERSO is not completely automatic. For MATITA, it needed some help to replace one lemma, and at the end to replace some terms. This leads us to wonder if it is possible for such a tool to be fully automatic or it at some point, it will always need human intervention.

Moreover, UNIVERSO does not deal with bound variables, and therefore it remains variables of type `Sort`. Since HOL does not have a notion of universes, these variables have to disappear. One easy way to make these variables disappear is to duplicate terms that contains these variables. A case where the variable will be `Prop` and one case where the variable will be `Type` for each variable. By chance, there is at most one such variable in a term.

In order to do so, I developed an other tool: DEDUKTIPLI. For performances matter, DEDUKTI does not keep in memory an AST of every symbol definition. Therefore, it is not possible for the user to have a global view of the file that is parsed. So one goal of DEDUKTIPLI was to add an AST to DEDUKTI. Besides, *Raphaël Cauderlier* – an other member of Deducteam – and I needed to generate side effects while a DEDUKTIfile was parsed, we decided to add monads inside of DEDUKTIPLI, it allows us to add definitions and terms easily.

I used DEDUKTIPLI so that it can catch bound variables of type `Sort` in these terms and so that it may replace every occurrence of the old terms by the new one. Deciding which one should be use wasn't very difficult, since every universe is in canonical form (thanks to universo).

DEDUKTIPLI is a tool that is written in Ocaml (300 hundred lines of code) in top of DEDUKTI.

### 3.3 Dependent products

In the CoC[4], there are the four following rules[5]:

```
prodPP : (a : prop) -> (TermP a -> prop) -> prop
prodPT : (a : prop) -> (TermP a -> type) -> type
prodTP : (a : type) -> (TermT a -> prop) -> prop
prodTT : (a : type) -> (TermT a -> type) -> type
```

However, only three of them are allowed in HOL namely `prodPP`, `prodTP` and `prodTT`. HOL does not allow to create a type that depends on proofs.

But `prodPT` is used several time in the MATITA arithmetic library (only 25 but on critical definitions and theorems). By looking at terms, we noticed that in general, the final type was not depending on a proposition as in the following example:

```
a:Prop => prodPT a (a':Prop => nat)
```

where it safes to replace the term above by just the term

```
a:Prop => nat
```

This may be done also by using DEDUKTIPLI. Although, doing such modification in a theorem may have several repercussions. In particular, this will ask to modify every term that uses this theorem. And sometimes, just removing the argument might not work since it can be used somewhere else.

Doing such transformation in the MATITA arithmetic library implies the suppression of some theorems for the reason above. About 20 theorems have been removed but none of them was used to prove the Fermat little theorem.

### 3.4 Logical connectives of MATITA in HOL

Logical connectives in MATITA are inductive types. But translating inductive types in HOL is problematic. One way to get around this is to first translate logical connectives of MATITA to an encoding of constructive predicate logic in DEDUKTI. This encoding is detailed in Appendix D.

This is not completely easy, since the return type of the MATITA connectives is a dependent type. For example conjonction in MATITA is encoded in DEDUKTI as

```
def And : Prop -> Prop -> Prop
```

the only way to construct and `and` type is by using the constructor `conj`

```
def conj : A:Prop -> B:Prop -> A -> B -> And A B.
```

which takes two propositions $A$ and $B$ a proof of these propositions.

The recursor rule (called match) is

```
def match_And_prop :
  A:Prop -> B:Prop ->
  return_type:((And A B) -> Prop) ->
  case_conj:(a:A -> b:B -> (return_type (conj A B a b))) ->
  z:(And A B) -> (return_type z).
```

The computational rule associated to the recursor rule is :

```
[__1, __, case_conj, return_type, _B, _A]
  match_And_prop _A _B return_type case_conj (conj _A _B __ __1)
  -->
  case_conj __ __1.
```

[4] rules of CoC are in appendix Appendix C

[5] here we use the HOLnotations. In CoC, `prop` is `Type` and `type` is `Kind`

The problem is that when we translate the MATITA conjonction to the one of DEDUKTI, the computational rule becomes

```
[return , case , x , A , B] match_And_prop A B return case x -->
  case (fst A B x) (snd A B x).
```

and we loose the information that a type `and` can only be constructed by the constructor `conj` that makes this last rule ill-typed because the left member of the rule has not the same type as the right member of the rule. A solution is to change the type of `return_type` to `Prop`. This may be achieved the same way as we did for the product rule by using DEDUKTIPLI.

# 4 Reconstruction of proofs from certificates

When one asks an automatic prover to solve a problem in *linear* arithmetic (the Quantifier Free Linear Real Arithmetic (QF_LRA) logic), this latter generally reduces the problem to solving $n$ linear inequalities and using the simplex algorithm. The objectives of this section is to

1. to explain what kind of certificate is needed to reconstruct the proof the problem and,
2. to identify arithmetic lemmas needed to reconstruct the proof,
3. to give that proof in sequent calculus.

We choose the sequent calculus since the focused system in CHECKERS is based on sequent calculus.

We also choose QF_LRA logic because it allows to express arithmetic theorems but also as we shall see, the certificate format will be very simple. This follows on from the fact that proving any theorem in QF_LRA logic may be reduce to proving several contradictions with a context that contains only inequalities. Proving a contradiction from inequalities in the QF_LRA logic is equivalent to prove an inequality of the form $n < 0$ with $n$ positive or $n \leq 0$ with $n$ positive. We decided that this inequality is produced by computing a linear combination of these inequalities where the coefficients $n_i$ will be given by the certificate. This is the only information contained inside the certificate.

The full proof is organized as follow:

1. Transform the sequent

   $$\frac{}{\Gamma \vdash A}$$

   to the sequent

   $$\frac{}{\sigma_1, \ldots, \sigma_n \vdash \bot}$$

   where $\sigma_1, \ldots, \sigma_n$ are inequalities.

2. Use the certificate to compute the following sum for each sub-goal generated at the previous step:

   $$\sum c_i \times \sigma_i$$

3. Using arithmetic properties, computing the previous sum to get a contradiction.

This is summarized in figure 4 where $\Pi_i$ is the proof computed at step $i$.

In the next sections we are going to describe the terms used in the QF_LRA logic and also the proofs $\Pi_2$ and $\Pi_3$. The proof $\Pi_1$ is left for future work.In this internship, we focused on the case were inequalities are only strict inequalities. The case were there is both large and strict inequalities is left for future work. We also make the assumption that inequalities are in normal form that will be detailed in 4.1.

$$\dfrac{\dfrac{\dfrac{\dfrac{\sigma_{1_1},\dots,\sigma_{1_{n_1}},\sum n_i\times\sigma_i, n<0\vdash\bot}{\Pi_3}}{\sigma_{1_1},\dots,\sigma_{1_{n_1}},\sum n_i\times\sigma_i\vdash\bot}}{\Pi_2}}{\sigma_{1_1},\dots,\sigma_{1_{n_1}}\vdash\bot} \quad \dots \quad \dfrac{\vdots}{\sigma_{m_1},\dots,\sigma_{m_{n_m}}\vdash\bot}}{\dfrac{\Pi_1}{\Gamma\vdash A}}$$

Figure 4: sketch of the full proof

### 4.1 Statement of the problem

Before stating the formal problem, we need to introduce some definitions. Let us specify first the syntax of terms that is used.

The grammar of terms is composed of formulas $(\sigma_1,\sigma_2,\dots)$, of expressions $(e_1,e_2,\dots)$ and of variables $(x_1,x_2,\dots)$ following this specification:

- For each integer $c$ there is an expression in the grammar $\overline{c}$. If moreover, $c$ is non-negative, then we use the notation $\overline{n}$.
- If $e_1$ and $e_2$ are two expressions, then $e_1+e_2$ is also en expression
- If $c$ is an integer and $e$ is en expression, then $\overline{c}\times e$ is also an expression
- If $e$ is en expression then $e<0$ is a formula
- If $e_1$ and $e_2$ are two expressions, then $e_1=e_2$ is a formula

The following notation will be useful to represent the left part of inequalities.

**Notation 1.** *If $l$ and $l'$ are two natural numbers, $e$ and $e_i$ $(l\leq i<l')$ are expressions, then*

$$[e]\overset{l'}{\underset{i=l}{+}}e_i := \begin{cases} e & \text{if } l'<l \\ \left([e]\overset{l'-1}{\underset{i=l}{+}}e_i\right)+e_{l'+1} & \text{if } l'\geq l \end{cases}$$

This definition looks like the definition of the operator $\sum$ in which we specify an associativity order on the terms and the sum is parameterized by the first term.

We also introduce the following notations that overload the symbols $+$ and $\times$ on inequalities:

**Notation 2.** *If $n$ is an natural number and $\sigma$ is a formula of the form $e<0$, then*

$$\overline{n}\times\sigma := (\overline{n}\times e)<0$$

**Notation 3.** *If $\sigma_1$ and $\sigma_2$ are two inequalities of the form $e_1<0$ and $e_2<0$, then*

$$\sigma_1+\sigma_2 := e_1+e_2<0$$

Moreover, we introduce a $\underline{\times}$ operator in order to be able to make the difference when $e$ is an expression of the form $e_1+e_2$ between $\overline{c}\times(e_1+e_2)$ and $\overline{c}\times e_1+\overline{c}\times e_2$.

**Notation 4.** *If $c$ is an integer, $l$ and $l'$ are two natural numbers, and $\sigma$ is an inequality of the form*

$$[\overline{d}]\overset{l'}{\underset{i=l}{+}}c_i\times x_i<0$$

*then*

$$\overline{c}\underline{\times}\sigma := [\overline{c\times d}]\overset{l'}{\underset{i=l}{+}}(\overline{c\times c_i}\times x_i)<0$$

as for $\times$ we also define a $\underline{+}$ operator

**Notation 5.** *$\sigma_1$ and $\sigma_2$ are two inequalities of the form*

$$[\overline{d}] \underset{i=l}{\overset{l'}{+}} \overline{c_i} \times x_i < 0$$

*then*

$$\sigma_1 \underline{+} \sigma_2 := [\overline{d_1 + d_2}] \underset{i=l}{\overset{l'}{+}} (\overline{c_{i_1} + c_{i_2}}) \times x_i < 0$$

Finally we are going to use the following notation

**Notation 6.** *$\overline{n} \underline{\times} \sigma_{1,\ldots,i}$ denotes the inequality in normal form obtained by adding the inequalities $\overline{n_1} \underline{\times} \sigma_1, \ldots, \overline{n_i} \underline{\times} \sigma_i$*

For this proof, we set $m$ as the number of inequalities, and $k$ as the number of variables. In order to make the things easier, let us do the following assumptions:

- Each inequality $\sigma_i$ with $0 \leq i < m$ is given in a normal form

$$[d_i] \underset{j=1}{\overset{k}{+}} (c_j \times x_j) < 0$$

- The certificate computed by the SMT solver contains $n$ coefficients $n_0, \ldots, n_{m-1}$ (natural numbers) such that the computation of

$$\sum_{i=0}^{m-1} n_i \times \sigma_i$$

gives an inequality $m < 0$ with $m$ a natural number

### 4.2 Axiom schemas

Instead of using traditional axioms with a $\forall$ quantifier, we use axiom schemas using the *for all* of the meta-language. A first reason for that is that it makes proofs easier. A second reason is that each instance of these axioms might be proved using a more general axiom using the $\forall$ operator. These allow us to be more general in the theory that will be effectively used. The axiom schemas that we need are listed below: for every constant $c, c_1, c_2$, for every natural number $n$, for every expression $e, e_1, e_1', e_2, e_2'$ and for every varialbe $x$,

$$\overline{c_1} + \overline{c_2} = \overline{c_1 + c_2} \tag{$AS_1$}$$

$$e_2 = e_2' \Rightarrow (e_1 + e_2) = (e_1 + e_2') \tag{$AS_2$}$$

$$e_1 = e_1' \Rightarrow (e_1 + e_2) = (e_1' + e_2) \tag{$AS_3$}$$

$$e = e' \Rightarrow e < 0 \Rightarrow e' < 0 \tag{$AS_4$}$$

$$e < 0 \rightarrow \overline{n} \times e < 0 \tag{$AS_5$}$$

$$\overline{n} \times (e + \overline{c} \times x) = \overline{n} \times e + \overline{n \times c} \times x \tag{$AS_6$}$$

$$\overline{c} \times (e \times x) = (\overline{c} \times e) \times x \tag{$AS_7$}$$

$$e_1 < 0 \to e_2 < 0 \to (e_1 + e_2) < 0 \tag{$AS_8$}$$

$$e_1 + e_2 = e_2 + e_1 \tag{$AS_9$}$$

$$(e_1 + e_2) + e_3 = e_1 + (e_2 + e_3) \tag{$AS_{10}$}$$

$$e_1 + (e_2 + e_3) = (e_1 + e_2) + e_3 \tag{$AS_{11}$}$$

$$\overline{n} \times x + \overline{c} \times x = (\overline{n + c}) \times x \tag{$AS_{12}$}$$

$$e_1 + \overline{0} \times e_2 = e_1 \tag{$AS_{13}$}$$

$$e + \overline{0} = e \tag{$AS_{14}$}$$

$AS_1$ is needed to transform the term $2 + 2$ into 4. $AS_2$, $AS_3$ and $AS_4$ are instanciation of Leibniz equality. They will be used to rewrite sub-terms inside a term. All the other axioms are arithmetic properties that are needed during the proof.

Using the equality $t = u$ inside a large term $A$, needs as many congruence axioms as the depth of $t$. To apply an equality inside a term, we define a notion of *context proof* parameterized by a proof $\Pi$ defined by the following grammar schema:

$$C[\Pi] ::= [\Pi] \mid C[\Pi] + e \mid e + C[\Pi]$$

where $e$ is an expression. These new proofs are defined inductively as:

$[\Pi]$ is defined as the proof $\Pi$.

$C[\Pi] + e$ is defined as the following proof :

$$\frac{\Gamma \vdash AS_3 \qquad \dfrac{[\Pi']}{\Gamma \vdash e_1 = e_1'}}{\Gamma \vdash e_1 + e = e_1' + e}\, app_2$$

Where $\Pi'$ is the proof of $C[\Pi]$.

$e + C[\Pi]$ is defined similarly than $C[\Pi] + e$ using the $AS_2$ axiom.

Using this new definition, we overload the definition

$$[d] \overset{l'}{\underset{i=l}{+}} e_i$$

with

$$[\Pi] \overset{l'}{\underset{i=l}{+}} e_i$$

that is the proof defined inductively as

$$[\Pi] \overset{l'}{\underset{i=l}{+}} e_i := \begin{cases} [\Pi] & \text{if } l' < l \\ C[[\Pi] \overset{l'-1}{\underset{i=l}{+}} e_i] + e_{l'+1}) & \text{if } l' \geq l \end{cases}$$

It is generally more convenient to work with *modus ponens* when applying an axiom schema to an argument. Even if *modus ponens* is not a rule in sequent calculus, this rule may be derived in sequent calculus:

$$\dfrac{\dfrac{[\Pi_0]}{\Gamma \vdash A \Rightarrow B} \qquad \dfrac{\dfrac{[\Pi_1]}{\Gamma \vdash A} \quad \dfrac{}{B \vdash B}\,\text{axiom}}{\Gamma, A \Rightarrow B \vdash B}\,\text{left} \Rightarrow}{\Gamma \vdash B}\,\text{cut}$$

More generally, one can prove by induction that for all $n$, the $app_n$ rule is admissible:

$$\dfrac{\dfrac{[\Pi_0]}{\Gamma \vdash A_1 \Rightarrow \cdots \Rightarrow A_n \Rightarrow B} \quad \dfrac{[\Pi_1]}{\Gamma \vdash A_1} \quad \ldots \quad \dfrac{[\Pi_n]}{\Gamma \vdash A_n}}{\Gamma \vdash B}\, app_n$$

### 4.3 Proof

During the proofs, we are going to use the following shortcuts:

- $\Gamma$ denotes the list of inequalities $\sigma_1, \ldots, \sigma_m$
- $\Sigma$ denotes the list of axiom schemas defined previously
- $\Delta$ denotes the list of inequalities $n_1 \underline{\times} \sigma_1, \ldots, n_m \underline{\times} \sigma_m$
- $\Theta$ denotes the list of inequalities $\underline{\sigma}_1, \ldots, \underline{\sigma}_{1,\ldots,m}$

Moreover, if $L$ is a list, we note $L_i$ the list that contains the first $i$ elements of $L$. We recall that we want to prove the following sequent

$$\dfrac{}{\Sigma, \sigma_1, \ldots, \sigma_m \vdash \bot}$$

from the certificate $n_1, \ldots, n_m$.

#### First step

The goal of the first step is to prove the inequalities $n_i \underline{\times} \sigma_i$ for all $i$ such that $0 \leq i < k$ and adding them to the context with a *cut*. The proof of the first step is then

$$\dfrac{\dfrac{[\Pi_{n_1 \underline{\times} \sigma_1}]}{\Sigma, \Gamma \vdash n_1 \underline{\times} \sigma_1} \qquad \dfrac{\dfrac{[\Pi_{n_2 \underline{\times} \sigma_2}]}{\Sigma, \Gamma, n_1 \underline{\times} \sigma_1 \vdash n_2 \underline{\times} \sigma_2} \quad \dfrac{\dfrac{[\Pi_a]}{\Sigma, \Gamma, \Delta \vdash \overline{d} < 0}\,\text{cut}}{\vdots}\,\text{cut}}{\Sigma, \Gamma, n_1 \underline{\times} \sigma_1 \vdash \overline{d} < 0}\,\text{cut}}{\Sigma, \Gamma \vdash \overline{d} < 0}\,\text{cut}$$

where $\Pi_{n_i \underline{\times} \sigma_i}$ is a proof that distributes the constant $n_i$ to the inequality $\sigma_i$ that it detailed just below while $\Pi_a$ will be detailed after.

**A proof of $n_i \underline{\times} \sigma_i$:** suppose that $\sigma_i = e < 0$ with

$$e = [d_i] \underset{l=1}{\overset{k}{+}} \overline{c_l} \times x_l$$

Let $e_j$ denotes the term

$$[\overline{n_i} \times [d_i] \underset{l=1}{\overset{j}{+}} \overline{c_l} \times x_l] \underset{l=j+1}{\overset{k}{+}} \overline{n_i \times c_l} \times x_l$$

Notice that $e_0 < 0$ is $n \underline{\times} \sigma$ and $e_k < 0$ is $n \times \sigma$. Then, the proof $\Pi_{n \underline{\times} \sigma}$ is:

$$\dfrac{\dfrac{\dfrac{}{\Sigma, \Gamma \vdash AS_4}\,\text{axiom} \quad \dfrac{[\Pi_{e_{i+1}=e_i}]}{\Sigma, \Gamma \vdash e_1 = e_0} \quad \dfrac{\dfrac{\dfrac{}{\Sigma, \Gamma \vdash AS_5}\,\text{axiom} \quad \dfrac{}{\Sigma, \Gamma \vdash e_k < 0}\,\text{axiom}}{\Sigma, \Gamma, \vdash e_k < 0}\,app_1}{\vdots}}{\Sigma, \Gamma \vdash e_1 < 0}\,app_2}{\Sigma, \Gamma \vdash e_0 < 0}\,app_2$$

If one denote $\Pi_i$ as the following proof

$$\dfrac{}{\Sigma, \Gamma \vdash \overline{n} \times [d] \underset{l=0}{\overset{i+1}{+}} \overline{c_l} \times x_l = \overline{n} \times [d] \underset{l=0}{\overset{i}{+}} \overline{c_l} \times x_l + (\overline{n \times c_{i+1}}) \times x_{i+1})}\,\text{instance of } AS_6$$

Then $\Pi_{e_{i+1}=e_i}$ is defined as $[\Pi_i] \underset{l=i+1}{\overset{k}{+}} c_l \times x_l$. It remains to construct the proof $\Pi_a$.

### Second step

The purpose of the second step is to add two inequalities together and get a new inequality in normal form. Applying the axiom schema $AS_8$ is not sufficient since the inequality obtained is not in normal form. Therefore one needs to use associative and commutative properties of the addition to get an inequality in normal form. Here is the proof $\Pi_a$:

$$\dfrac{\dfrac{[\Pi_{\underline{\sigma}_{1,2}}]}{\Sigma, \Gamma, \Delta \vdash \underline{\sigma}_{1,2}} \quad \dfrac{\dfrac{[\Pi_b]}{\Sigma, \Gamma, \Delta, \Theta \vdash \overline{d} < 0}}{\vdots}}{\Sigma, \Gamma, \Delta \vdash \overline{d} < 0}\,cut}{\Sigma, \Gamma, \Delta \vdash \overline{d} < 0}\,cut$$

The proof $[\Pi_{\underline{\sigma}_{1,\ldots,m}}]$ relies on the following lemma:
For all expressions $e_1, e_2$, constants $c_1, c_2$ and variable $x$,

$$(e_1 + \overline{c_1} \times x) + (e_2 + \overline{c_2} \times x) = (e_1 + e_2) + \overline{c_1 + c_2} \times x \qquad (L_{15})$$

The proof of $L_{15}$ is given informally here but a formal proof may be easily reconstructed:

$$\begin{aligned}
(e_1 + \overline{c_1} \times x) + (e_2 + \overline{c_2} \times x) &= e_1 + (\overline{c_1} \times x + (e_2 + \overline{c_2} \times x)) && (\text{ by } AS_{10}) \\
&= e_1 + (\overline{c_1} \times x + (\overline{c_2} \times x + e_2)) && (\text{ by } AS_9) \\
&= e_1 + ((\overline{c_1} \times x + \overline{c_2} \times x) + e_2) && (\text{ by } AS_{11}) \\
&= e_1 + ((\overline{c_1 + c_2}) \times x + e_2) && (\text{ by } AS_{12}) \\
&= e_1 + (e_2 + (\overline{c_1 + c_2}) \times x) && (\text{ by } AS_9) \\
&= (e_1 + e_2) + (\overline{c_1 + c_2}) \times x && (\text{ by } AS_{10})
\end{aligned}$$

Now, suppose that $\sigma_{1,\ldots,i} = [d] \underset{l=1}{\overset{k}{+}} \overline{c_l} \times x_l < 0$ and that $\sigma_{i+1} = [d'] \underset{l=1}{\overset{k}{+}} \overline{c'_l} \times x_l < 0$. We define $\Pi_o$ as

$$\frac{}{[d]\underset{l=1}{\overset{o-1}{\pm}}\overline{c_l}\times x_l+\overline{c_o}\times x)+([d']\underset{l=1}{\overset{k}{\pm}}\overline{c'_l}\times x_l+\overline{c'_o}\times x)=([d]\underset{l=1}{\overset{o-1}{\pm}}\overline{c_l}\times x_l+[d']\underset{l=1}{\overset{k}{\pm}}\overline{c'_l}\times x_l)+\overline{c_o+c'_o}\times x_o}\ \text{instance of } L_{15}$$

Now we can give the proof $[\Pi_{\overline{n}\underline{\times}\sigma_{1,\ldots,i+1}}]$:

$$\dfrac{\Sigma,\Gamma,\Delta\vdash AS_4 \qquad [\Pi_o]\ \underset{l=k}{\overset{k}{+}}c_l\times x_l \qquad [\Pi_{\underline{\sigma}_{1,\ldots,i},\overline{n_{i+1}}\underline{\times}\sigma_{i+1}}]}{\dfrac{\Sigma,\Gamma,\Delta\vdash e_k<0}{\dfrac{}{\Sigma,\Gamma,\Delta\vdash\underline{\sigma}_{1,\ldots,i+1}}\ cut}\qquad \dfrac{\dfrac{}{\Sigma,\Gamma,\Delta,e_k\vdash\underline{\sigma}_{1,\ldots,i+1}}\ cut}{\Sigma,\Gamma,\Delta\vdash\underline{\sigma}_{1,\ldots,i+1}}\ cut}\qquad \dfrac{\dfrac{[AS_1]\ \underset{l=0}{\overset{k}{+}}c_l\times x_l}{\Sigma,\Gamma,\Delta,e_k,\ldots,e_2\vdash\underline{\sigma}_{1,\ldots,i+1}}\qquad \dfrac{}{\Sigma,\Gamma,\Delta,e_k,\ldots,e_1\vdash\underline{\sigma}_{1,\ldots,i+1}}\ axiom}{\vdots}\ cut$$

where $e_o$ denotes the expression

$$[[d]\underset{j'=1}{\overset{o-1}{\pm}}\overline{c_{j'}}\times x+[d']\underset{j'=1}{\overset{o-1}{\pm}}\overline{c'_{j'}}\times x]\underset{j=o}{\overset{k}{\pm}}\overline{c_j+c'_j}\times x$$

and $\Pi_{\sigma,\sigma'}$ denotes the proof

$$\frac{\Sigma,\Gamma,\Delta\vdash AS_8 \qquad \Sigma,\Gamma,\Delta\vdash\sigma \qquad \Sigma,\Gamma,\Delta\vdash\sigma'}{\Sigma,\Gamma,\Delta\vdash\sigma+\sigma'}\ app_2$$

**Third step**

At this point, the environment of the sequent contains the inequality $\underline{\sigma}_{1,\ldots,i+1}$ that should be

$$[\overline{d}]\underset{j=1}{\overset{k}{\pm}}\overline{0}\times x_j<0$$

otherwise the certificate is incorrect. Let $e_i$ denotes the expression

$$[\overline{d}]\underset{j=i}{\overset{k}{\pm}}\overline{0}\times x_j$$

Notice that $\underline{\sigma}_{1,\ldots,i+1}=e_1<0$. It remains to apply the $AS_{13}$ for each variable. Let $\Pi_i$ denotes the proof

$$\frac{}{\Sigma,\Gamma,\Delta\vdash\overline{d}+\overline{0}\times x_i=\overline{d}}\ \text{instance of } AS_{13}$$

and let $\Pi_{e_i}$ denotes the proof

$$[\Pi_i]\underset{j=i}{\overset{k}{\pm}}\overline{0}\times x_j$$

then the proof of the fourth step is

$$\dfrac{\Sigma,\Gamma,\Delta,\underline{\sigma}_{1,\ldots,i+1}\vdash AS_4 \qquad \dfrac{[\Pi_{e_k}]}{\Sigma,\Gamma,\Delta,\underline{\sigma}_{1,\ldots,i+1}\vdash e_k=\overline{d}} \qquad \dfrac{\dfrac{}{\Sigma,\Gamma,\Delta,\underline{\sigma}_{1,\ldots,i+1}\vdash e_1<0}\ \text{axiom}}{\vdots}}{\Sigma,\Gamma,\Delta,\underline{\sigma}_{1,\ldots,i+1}\vdash e_k<0}\ app_2}{\Sigma,\Gamma,\Delta,\underline{\sigma}_{1,\ldots,i+1}\vdash\overline{d}<0}$$

This conclude the proof of the sequent

$$\frac{}{\Sigma,\sigma_1,\ldots,\sigma_m\vdash\bot}$$

# 5 Conclusion

In this master thesis, we have explored two ways to answer to re-usability problems with proof checkers:

- Translate a proof from one proof checker into an other
- Reconstruct proofs from a certificate

On the DEDUKTI side, we have seen that this asks to develop some new tools. These tools allows us to remove some features like universes or depend products when it is possible. However, this process may discard some theorems. One main difficulty is to known what can be done automatically and what should be done manually. That is why experimental tools like Universo and Deduktipli may be enhance in several ways.

Future work is to finish the translation of MATITA proofs to HOL and also to improve the tools UNIVERSO and DEDUKTIPLI.

**UNIVERSO** only works with two universes. It should be interesting to extend it so that it works with any number of universes. This asks to add a $\vee$ and $\leq$ operators on constraints. To solve these constraints, the easiest way probably would be to use an SMT solver. However, the solution might not be easy to use to reconstruct terms. This would ask to add *lifting* in some places and for now it is not easy to see where these *lifting* would be necessary.

**DEDUKTIPLI** is an experimental tool. One regret with that tool is that it is mostly used to rewrite terms without using directly the rewriting system of DEDUKTI. It would be nice if it would be possible to use rewrite rules of DEDUKTI to rewrite terms with deduktipli.

On the CHECKERS side, we have given a proof in the sequent calculus that allows to reconstruct proofs coming from SMT solvers expressed in the QF_LRA logic. Future work is to implement this proof in CHECKERS. Even if for now the proof is quite complete, it would be nice to use the proof search part of CHECKERS (implemented by the focusing) to avoid all these complicated things with associative and commutative properties of the operator plus.

# Appendix A Bibliography

# References


[And92] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *J. Log. Comput.*, 2(3):297–347, 1992.

[Ass15] Ali Assaf. *A framework for defining computational higher-order logics. (Un cadre de définition de logiques calculatoires d'ordre supérieur)*. PhD thesis, École Polytechnique, Palaiseau, France, 2015.

[CD07] Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In Simona Ronchi Della Rocca, editor, *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings*, volume 4583 of *Lecture Notes in Computer Science*, pages 102–117. Springer, 2007.

[CM16] Zakaria Chihani and Dale Miller. Proof certificates for equality reasoning. *Electr. Notes Theor. Comput. Sci.*, 323:93–108, 2016.

[CMR13] Zakaria Chihani, Dale Miller, and Fabien Renaud. Foundational proof certificates in first-order logic. In Maria Paola Bonacina, editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, pages 162–177. Springer, 2013.

[Gon07] Georges Gonthier. The four colour theorem: Engineering of a formal proof. In Deepak Kapur, editor, *Computer Mathematics, 8th Asian Symposium, ASCM 2007, Singapore, December 15-17, 2007. Revised and Invited Papers*, volume 5081 of *Lecture Notes in Computer Science*, page 333. Springer, 2007.

[Gon13] Georges Gonthier. Engineering mathematics: the odd order theorem proof. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 1–2. ACM, 2013.

[KW10] Chantal Keller and Benjamin Werner. Importing HOL light into coq. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *Lecture Notes in Computer Science*, pages 307–322. Springer, 2010.

[LM09] Chuck Liang and Dale Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theor. Comput. Sci.*, 410(46):4747–4768, 2009.

[LM11] Chuck Liang and Dale Miller. A focused approach to combining logics. *Ann. Pure Appl. Logic*, 162(9):679–697, 2011.

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Log. Comput.*, 1(4):497–536, 1991.

[Sai15] Ronan Saillard. *Typechecking in the lambda-Pi-Calculus Modulo : Theory and Practice. (Vérification de typage pour le lambda-Pi-Calcul Modulo : théorie et pratique).* PhD thesis, Mines ParisTech, France, 2015.

# Appendix B MATITAarithmetic library

The following picture is a dependency graph between files of the MATITAarithmetic library. If $B$ depends on $A$ and if there exists a $C$ such that $C$ depends on $A$ and $B$ depends on $C$, then the dependency of $A$ to $B$ is not shown for sake of readability.

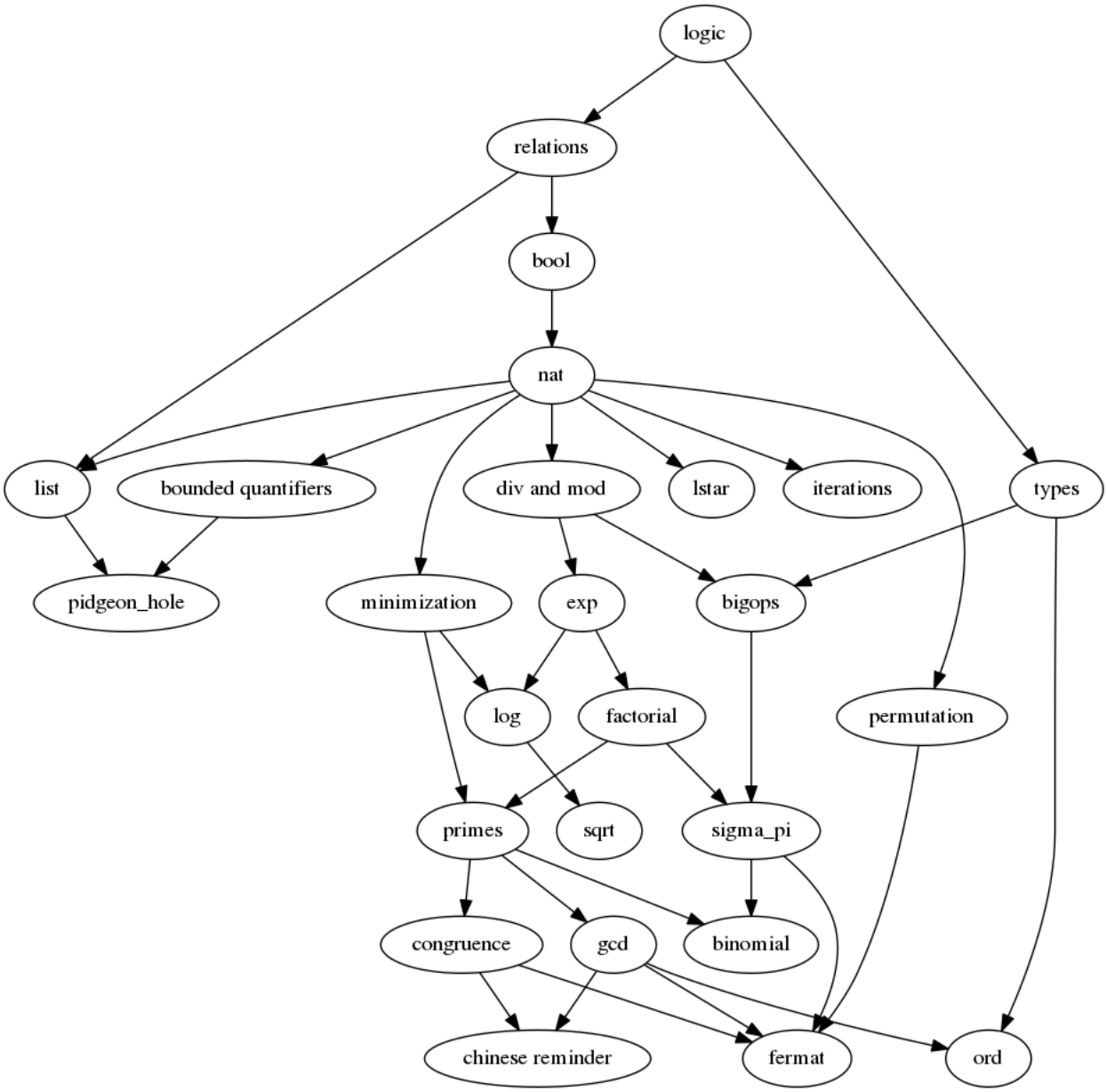


Figure 5: MATITA arithmetic library

# Appendix C CiC in DEDUKTI

```
Nat : Type.

z : Nat.
s : Nat -> Nat.

def m : Nat -> Nat -> Nat.
[i : Nat] m i z --> i.
[j : Nat] m z j --> j.
[i : Nat, j : Nat] m (s i) (s j) --> s (m i j).

(; Sorts ;)

Sort : Type.

prop : Sort.
type : Nat -> Sort.

(; Universe successors ;)
def succ : Sort -> Sort.
[] succ prop --> type z.
[i : Nat] succ (type i) --> type (s i).

(; Universe cumulativity ;)
def next : Sort -> Sort.
[] next prop --> type z.
[i : Nat] next (type i) --> type (s i).

(; Universe product rules ;)
def rule : Sort -> Sort -> Sort.
[s1 : Sort] rule s1 prop --> prop.
[s2 : Sort] rule prop s2 --> s2.
[i : Nat, j : Nat] rule (type i) (type j) --> type (m i j).

def max : Sort -> Sort -> Sort.
[s1 : Sort] max s1 prop --> s1.
[s2 : Sort] max prop s2 --> s2.
[i : Nat, j : Nat] max (type i) (type j) --> type (m i j).

(; Types and terms ;)

Univ : s : Sort -> Type.
def Term : s : Sort -> a : Univ s -> Type.

univ : s : Sort -> Univ (succ s).
def lift : s1 : Sort -> s2 : Sort -> a : Univ s1 -> Univ (max s1 s2).
def prod : s1 : Sort -> s2 : Sort -> a : Univ s1 ->
           b : (Term s1 a -> Univ s2) -> Univ (rule s1 s2).

[s : Sort] Term _ (univ s) --> Univ s.
[s1 : Sort, s2 : Sort, a : Univ s1] Term _ (lift s1 s2 a) --> Term s1 a.
[s1 : Sort, s2 : Sort, a : Univ s1, b : (Term s1 a -> Univ s2)]
  Term _ (prod s1 s2 a b) --> x : Term s1 a -> Term s2 (b x).
```

```
(; Canonicity rules ;)

[s : Sort] max s s --> s.
[s1 : Sort, s2 : Sort, s3 : Sort]
  max (max s1 s2) s3 --> max s1 (max s2 s3).
[s1 : Sort, s2 : Sort, s3 : Sort]
  rule (max s1 s3) s2 --> max (rule s1 s2) (rule s3 s2).
[s1 : Sort, s2 : Sort, s3 : Sort]
  rule s1 (max s2 s3) --> max (rule s1 s2) (rule s1 s3).

[s : Sort, a : Univ s] lift s s a --> a.
[s1 : Sort, s2 : Sort, s3 : Sort, a : Univ s1]
  lift _ s3 (lift s1 s2 a) -->
  lift s1 (max s2 s3) a.
[s1 : Sort, s2 : Sort, s3 : Sort, a : Univ s1, b : Term s1 a -> Univ s2]
  prod _ s2 (lift s1 s3 a) b -->
  lift (rule s1 s2) (rule s3 s2) (prod s1 s2 a b).
[s1 : Sort, s2 : Sort, s3 : Sort, a : Univ s1, b : Term s1 a -> Univ s2]
  prod s1 _ a (x => lift s2 s3 (b x)) -->
  lift (rule s1 s2) (rule s1 s3) (prod s1 s2 a (x => b x)).
```

# Appendix D Constructive Predicate logic in DEDUKTI

```
prop : Type.
type : Type.

eps : prop -> Type.
eta : type -> Type.

def True : prop
[] eps True --> z:prop -> (eps z) -> (eps z).

def False : prop.
[] eps False --> z:prop -> eps z.

def Imp : prop ->
          prop -> prop.
[x,y] eps (Imp x y) --> eps x -> eps y.

def Not : prop -> prop.
[x] eps (Not x) --> eps (Imp x False).

def And : prop ->
          prop -> prop
[x,y] eps (And x y) -->
                z:prop -> (eps x -> eps y -> eps z) -> eps z.

def Or : prop ->
         prop -> prop.
[x,y] hol.TermP (Or x y) -->
                z:prop -> (eps x -> eps z) -> (eps y -> eps z) -> eps z.

def Ex : A:type -> (eta A -> prop) -> prop.
[A, f] eps (Ex A f) --> z:prop -> (x:eta A -> eps (f x) -> eps z)
                              -> eps z.

def I : eps True := _ => z => z.

def pair : A:prop -> B:prop -> eps A -> eps B -> eps (And A B)
         :=
           A => B => a => b => z => f => f a b.

def fst : A:prop -> B:prop -> eps (And A B) -> eps A :=
          A => B => and => and A (a => b => a).

def snd : A:prop -> B:prop -> eps (And A B) -> eps B :=
          A => B => and => and B (a => b => b).

def injl : A:prop -> B:prop -> eps A -> eps (Or A B) :=
           A => B => a => z => fa => fb => fa a.

def injr : A:prop -> B:prop -> eps B -> eps (Or A B) :=
           A => B => b => z => fa => fb => fb b.

def split : A:prop -> B:prop -> C:prop ->
            eps (Or A B) -> (eps A -> eps C) -> (eps B -> eps C) -> eps C :=
```

```
            A => B => C => or => fa => fb => or C fa fb.

def ex_witness : A:type -> f:(eta A -> prop) -> eps (Ex A f) -> eta A.
[A, f, x, p] ex_witness A f (z => f => f x p) --> x.

def ex_proof : A:type -> f:(eta A -> prop) -> E:eps (Ex A f)
                -> eps (f (ex_witness A f E)).
[A, f, x, p] ex_proof A f (z => f => f x p) --> p.
```