Towards an implementation in LambdaProlog of the two level Minimalist Foundation

Alberto Fiori

November 12, 2017

Contents

Introduction

In 2005 M.Maietti and G. Sambin [MS05] argued about the necessity of building a foundation for constructive mathematics to be taken as a common core among relevant existing foundations in axiomatic set theory, such as Aczel-Myhill's CZF and Mizar's TG set theory, or in category theory, such as the internal theory of a topos, or in type theory, such as Martin-Lofs type theory and Coquands Calculus of Inductive Constructions. Moreover, they asked the foundation to satisfy the proofs-as-programs paradigm, namely the existence of a realizability model where to extract programs from proofs. Finally, the authors wanted the theory to be appealing to standard mathematicians and therefore they wanted extensionality in the theory, e.g. to reason with quotient types and to avoid the intricacies of dependent types in intensional theories.

In the same paper they noticed that theories satisfying extensional properties, like extensionality of functions, can not satisfy the proofs-as-programs requirement. Therefore they concluded that in a proofs-as-programs theory one can only represent extensional concepts by modelling them via intensional ones in a suitable way, as already partially shown in some categorical contexts.

Finally, they ended up proposing a constructive foundation for mathematics equipped with two levels: an intensional level that acts as a programming language and is the actual proofs-as-programs theory; and an extensional level that acts as the set theory where to formalize mathematical proofs. Then, the constructivity of the whole foundation relies on the fact that the extensional level must be implemented over the intensional level, but not only this. Indeed, following Sambins forget-restore principle, they also required that extensional concepts must be abstractions of intensional ones as result of forgetting irrelevant computational information. Such information is then restored when extensional concepts are translated back at the intensional level.

In 2009 M. Maietti [Mai09] presented the syntax and judgements of the two levels, together with a proof that a suitable completion of the intentional level provides a model of the extensional one. The proof is constructive and based on a sequence of categorical constructions, the most important being the construction of a quotient model within the intensional level, where setoids are used to encode extensional equality in an intensional language, and a notion of canonical isomorphism between intensional dependent setoids.

In this work we will present an implementation of the following software components:

1. a type checker for the intensional level
2. a reformulation of the extensional level that allows to store sintactically proof objects that are later used to provide the information to be restored

when going from the extensional to the intensional level so to avoid general proof search during the interpretation

3. a type checker for the obtained extensional level

4. a translator from well-typed extensional terms and types to well-typed intensional terms and types

As a future extension we hope to implement also other software components:

1. a formal validation (in Abella) of our reformulation of the extensional level and of the correctness of the implementation

2. code extraction at the intensional level (in Haskell/Lisp)

3. a proof assistant at the extensional level (using the constraint programming functionality offered by ELPI)

Combining the translator with a proof extraction component it will be possible to extract programs from proofs written in the extensional level.

We have chosen λProlog [NM88](and in particular its recent implementation ELPI [Dun+15]) as the programming language to write the two type checkers and the interpreter. The benefits are that λProlog takes care of the intricacies of dealing with binders and alpha-conversion and moreover a λProlog implementation of a syntax-directed judgement is just made of simple clauses that are almost literal translations of the judgemental rules. This allows humans (and logicians in particular) to easily inspect the code to spot possible errors.

1 Higher Order Logic Programming

At the beginning of our work it was decided that the this project should be implemented using λProlog and ELPI, one of its implementations which allows for more flexibility (e.g. optional type declarations) more functionality (e.g. constraint programming, which will be useful in later phases of the project) and also improved performances compared to other major implementations of λProlog.

As a programming language λProlog is naturally a good fit for a binder-heavy project like our as it natively offer all the functionality required to handle α-equivalence, capture avoiding substitutions and higher-order unification; moreover the high level abstractions and backtracking allow for almost a 1-1 encoding of inference rules. As an example below we can see the comparison between a natural

deduction inference rule and a λProlog clause in the case of Π-Introduction. In natural deduction style we have the following inference rule

$$\frac{C(x)\ set\ [x \in B] \qquad B\ set}{\Pi_{x \in B} C(x)\ set}\ \Pi\text{-F}$$

while in λProlog we would write

```
ofType (setPi B C) KIND3 IE
    :- ofType B KIND1 IE
    ,  (pi x\ locDecl x B
        => ofType (C x) KIND2 IE)
    ,  pts_fun KIND1 KIND2 KIND3
    .
```

where `ofType` represent the *set* judgement, `locDecl` is used to locally introduce hypothesis in the context and `pts_fun` is used to decide whether the produced type is a set or a collection.

While the λProlog version is more verbose it should be noted that this one rule encode the Π-Formation rule for both the extensional and the intensional level and also for all combination of arguments kinds between sets, collections, propositions and small proposition.

1.1 Changes to the calculus

As pointed out in the introduction before we could begin the implementation we had to apply some changes to the calculi exposed in [Mai09].

1.1.1 Syntax Directed Rules

Between the rules of *mTT* and *emTT* there is a rule which states that equal type have the same elements, that is:

$$\frac{a \in A \qquad A = B}{a \in B}\ \text{conv}$$

This rule is problematic from an implementation point of view since it can be triggered at any moment and (at the extensional level) would start a proof search trying to prove $A = B$ for a unknown A. A simple solution is to preemptively compose this rule to all other rule that might need it; for example we can see that the elimination rule for dependents products changes from

$$\frac{f \in \Pi_{x \in B} C(x) \qquad t \in B}{\mathsf{apply}(f, t) \in C(t)}\ \Pi\text{-E}$$

to

$$\frac{f \in \Pi_{x \in B} C(x) \qquad t \in B' \qquad B = B'}{\mathsf{apply}(f, t) \in C(t)}\ \Pi\text{-E}$$

the main difference being that in the syntax directed version we check that the domain type of f and the inferred type of t are convertible instead of requiring syntactic equality like in the original rule. In this way the power of the calculus is surely not increased as the syntax directed rule is admissible in the calculus with the two previous rules, indeed

$$\frac{f \in \Pi_{x \in B} C(x) \qquad \dfrac{t \in B' \qquad B = B'}{t \in B}\ \text{conv}}{\mathsf{apply}(f, t) \in C(t)}\ \Pi\text{-E}$$

is a derivation of the same judgement from the same hypothesis.

For this change to be effective we also need to reformulate the conversion rules given by the calculi (e.g. $\mathsf{apply}(\lambda^B x.f, t) = f\{t/x\}$) in directed rules describing a reduction predicate $\rhd$ (e.g. $\mathsf{apply}(\lambda^B x.f, t) \rhd f\{t/x\}$) from which we will define equality as reducibility to a common term, which is equivalent to the original equality given that both calculi of the Minimalist Foundation have both the property of Church-Rosser confluence and strong normalization. This new relation will be implemented by the predicate `conv`.

The full modifications to the calculi in our implementation with respect to making the rule syntax directed are mostly about checking for convertibility/equality where mTT/emTT require syntactical equality. We plan to provide a full proof of the validity of this modification, and also of the followings, in Abella in the future.

1.1.2 Informative Proof-terms at the Extensional Level and Exstensional Equality

$$\frac{b \in B \qquad \mathsf{true} \in C(b) \qquad C(x)\ \mathit{prop}\ [x \in B]}{\mathsf{true} \in \exists_{x \in B} C(x)}\ \exists\text{-I} \qquad \frac{\mathsf{true} \in \mathsf{Eq}(A, a, b)}{a = b \in A}\ \text{Eq-E}$$

In the calculus emTT a token proof term true is used to represent all proofs of propositions and small propositions; this poses problems during the interpretation when we need to construct intensional proof terms given only the token true. Moreover the strong elimination rule for the propositional extensional equality turns every conversion in an undecidable problem of general proof search for $\mathsf{true} \in \mathsf{Eq}(A, a, b)$. To reasonably implement the extensional calculus and the interpretation we need a way to construct intensional proofs from extensional proof and a way to contain the undecidability of the strong elimination rule for the extensional propositional equality.

A first approach to the problem could have been to have both the extensional type checker and the interpretation work on full derivations; this is the method chosen by Maietti in [Mai09] and would solve the problems regarding the missing informations in the extensional calculus and language, but was discarded as a solution because from an implementation standpoint the code to encode a derivation would be almost a copy of the code needed to encode terms, yet with subtle, but non trivial, differences.

The chosen solution was to both add informative proof terms to the extensional level and weaken the elimination rule of the propositional extensional equality. The first part of the solution makes the extensional rule for proposition more similar to the intensional ones

$$\frac{b \in B \qquad p \in C(b) \qquad C(x)\ prop\ [x \in B]}{< b,_{\exists}\ p > \in \exists_{x \in B} C(x)}\ \exists\text{-I}$$

so that during the interpretation more informations are available.

The second part modifies the extensional propositional equality elimination rule to a weaker but equivalent form; the rule

$$\frac{\mathsf{true} \in \mathsf{Eq}(A, a, b)}{a = b \in A}\ \text{Eq-E}$$

becomes

$$\frac{}{a = b \in A\ [\mathsf{h} \in \mathsf{Eq}(A, a, b)]}\ \text{Eq-E}$$

This weaker form has nicer computational properties, the most important one being that it is always deterministic and terminating to check if it is possible to apply given a specific context while still allowing to locally definitions in a context via implication or dependents types.

A drawback of this solution is that it is complex to use even simple lemmas; to this problem we offer two solutions: to manually use the cut elimination of the calculus or to introduce a new *let in* construct

$$\mathsf{let}\ x\ :=\ t_1 \in T\ \mathsf{in}\ t_2$$

used to introduce locally typed definitions.

As of this writing this *let in* in not yet implemented as there many different ways the needed functionality can be obtained each with different compromises and it is not entirely clear which implementation method is the most adeguate to our project. In our simple cases it is acceptable to manually perform the cut elimination to locally include the desired lemma so we plan to implement a proper *let in* once there is a strong need for it.

1.1.3 ξ-Rule

Another difference between the calculus in Maietti's work and ours is that our model validates the ξ-rule at the intensional level, removing it would require a syntax directed rule for explicit substitution.

2 The Interpretation

One of the main result of [Mai09] is a categorical proof of the validity of an interpretation of an extensional type theory into an intensional one.

In particular first a quotient model Q(mTT) is built on the theory mTT, this model is given as a category where objects are pairs of $(A, =_A)$ (abbreviated to $A_=$ when $=_A$ it is clear from the context) where A is an intensional type called support and

$$x =_A y \; props \; [x \in A, \; y \in A]$$

is an equivalence relation on A; which means that there are terms rfl_A, symm_A and tra_A such that the following judgements hold in mTT

$$\begin{aligned} \mathsf{rfl}_A(x) &\in x =_A x \; [x \in A] \\ \mathsf{sym}_A(x, y, u) &\in y =_A x \; [x \in A, \; y \in A, \; u \in x =_A y] \\ \mathsf{tra}_A(x, y, z, u, v) &\in x =_A z \; [x \in A, \; y \in A, \; z \in A, \; u \in x =_A y, v \in y =_A z] \end{aligned}$$

In the following we will refers to the objects of Q(mTT) as setoids.

The morphisms of Q(mTT) from $(A, \; =_A)$ to $(B, \; =_B)$ are all the well-typed terms of mTT

$$f(x) \in B(x) \; [x \in A]$$

that preserve the setoid equality, that is there exist a proof-term

$$\mathsf{pr}_1(x, y, z) \in f(x) =_B f(y) \; [x \in A, \; y \in A, \; z \in x =_A y]$$

Equality between two morphisms f,g : $(A, =_A) \to (B, =_B)$ holds if they maps equal elements of $A_=$ to equal elements of $B_=$, that is if there is a proof-term

$$\mathsf{pr}_2(x) \in f(x) =_B g(x) \; [x \in A]$$

Starting from the category Q(mTT) we can define also *dependent setoids*, given a setoid $A_=$ we say that

$$B_=(x) \; [x \in A_=]$$

is a dependent setoid if its support is a dependent type $B(x) \; type \; [x \in As]$, if its relation is a dependent proposition

$$y =_{B(x)} y' \; [x \in A, \; y \in B(x), \; y' \in B(x)]$$

and if there is a substitution morphism

$$\sigma^{x_2}_{x_1}(d, y) \in B(x_2)\ [x_1 \in A,\ x_2 \in A,\ d \in x_1 =_A x_2,\ y \in B(x_1)]$$

that satisfies the properties given by definition 4.9 in [Mai09], that is it needs to preserve the equality of $B_=$, to be proof-irrelevant by not depending on the particular choice of the proof-term $d \in x_1 = x_2$ (so that we can write $\sigma^{x_2}_{x_1}(y)$ in place of $\sigma^{x_2}_{x_1}(d, y)$), there need to always exists the identity substitution [1] $\sigma^{x_1}_{x_1}(\mathsf{rfl}_{B(x_1)}(x_1), -)$ and lastly substitution morphisms need to be composable, that is we require

$$\sigma^{x_3}_{x_2}(\sigma^{x_2}_{x_1}(y)) =_{B(x_3)} \sigma^{x_3}_{x_1}(y)$$

in the appropriate context[2].

Over these substitution we can define *canonical isomorphism* between two type B_1 and B_2 as the only σ between them. In our code we have implemented these isomorphisms as with the predicate `tau B B' P`.

The reason these construction are needed is because the strong elimination rule of the extensional propositional equality can derive a judgemental equality from just an hypothetical assumption. For an example we can see that a judgement like $\mathsf{true} \in \mathsf{Eq}(C, c, d)\ [c \in C, d \in C]$ is a well formed but not derivable judgement in emTT, but if we add one more assumption to the context we obtain

$$\mathsf{true} \in \mathsf{Eq}(C + C, \mathsf{inl}(c), \mathsf{inl}(d))\ [c \in C, d \in C, h \in \mathsf{Eq}(C, c, d)]$$

which is derivable since we can use the rules $\mathsf{prop} - \mathsf{true}$ and E-Eq to derive the judgement $c e.g.\ d \in C$ from the given context.

This can be seen also in the fact that this judgement

$$\forall_{x \in \mathbb{1}} \forall_{f \in \Pi(\mathsf{Eq}(\mathbb{1}, x, \star), \mathbb{1})} \mathsf{Eq}(\mathbb{1}, \star, x) \Rightarrow \mathsf{Eq}(\mathbb{1}, \mathsf{apply}(f, \mathsf{eq}(\mathbb{1}, \star)), \mathsf{apply}(f, \mathsf{eq}(\mathbb{1}, \star)))\ \ props^3$$

[1] Which in general does not need to be the identity morphism, for example

$$\sigma^x_x(\mathsf{rfl}_A(x), w) \equiv \mathsf{elim}_+(w, (z).\mathsf{inl}(\sigma^x_x(\mathsf{rfl}_A(x)(x), z)), (z).\mathsf{inr}(\sigma^x_x(\mathsf{rfl}_A(x)(x), z)))$$

if $w \in B + C\ [x \in A]$

[2] It is worth noting that in the Minimalist Foundation the only type that directly takes terms as arguments are the propositional equalities (written Eq and Id in emTT and mTT respectively). For any $B(x)$ *type* $[x \in A]$ the only possible occurrences of x are as an argument of Eq (resp. Id if in mTT), together with the absence of strong elimination of terms towards types this means that conversion of types is always only reduced to conversion of terms and that types containing no propositions cannot be truly dependent. For example the canonical way to construct a type *list of type C of length n* is via a dependent sum $\Sigma(\mathsf{List}(C), (l).P(l, n))$ where $P(l, n)$ is the proposition *length of l equals n.*

is derivable in the empty context even if there is a mismatch between the types of f and $\mathsf{eq}(\mathbb{1}, \star)$. Here for example we would want to apply a canonical isomorphism to $[\![\mathsf{eq}(\mathbb{1}, \star)]\!]$[4] so that its type match with the domain of $[\![f]\!]$; after the correction we obtain $\mathsf{apply}(f, \tau^{[\![\mathsf{Eq}(\mathbb{1},x,\star)]\!]}_{[\![\mathsf{Eq}(\mathbb{1},\star,\star)]\!]}([\![\mathsf{eq}(\mathbb{1}, \star)]\!]))$ which is well-typed in mTT

From this introduction to the structure of the interpretation we can see that the method used to force the semantics of emTT on mTT (via Q(mTT)) is to require the creation of suitable proof terms thus restricting the available language to a subset that preserve the extensional semantic.

We can se another more explicit application of this method in the interpretation of dependent products, indeed in this case a function term of emTT is interpreted as a pair whose first projection is an intensional function and whose second projection is a proof that the first projection preserves the setoid equalities of Q(mTT).

From the point of view of the implementation this means that we have to produce various proof for each of the type constructor we wish to include in our implementation.

As of now we have implemented all that is needed to fully interpret the subset of emTT comprised of singletons, dependent products, implications, universal quantifications, propositional equality. We believe that this subset exhibit enough of the complexity of the interpretation and feel confident that the structure of our implementation will be able to cover also the needs of the remaining constructors.

3 The Code

All the code was written in λProlog (in particular it was written for the ELPI implementation [Dun+15] of λProlog) and make extensive use of many high level features that in other languages would not be available. Still our code does not use many of the more advanced features offered by ELPI like constraint programming which we plan to use in the next phase of the project when we will start to implement a kernel for the planned proof checker

In this section we will show some extract of the code and explain how we obtained the desired functionalities.

3.1 Project Structure and Extensibility

We have divided the code 4 categories `main`, `calc`, `test`, `debug`. The first, `main` holds type declaration for the other modules and define the general functionality of the theory shared by all type constructor of both levels. the second is a folder containing a `.elpi` file for each type constructor; in each of these files we implement

[4] In this work we use the double bracket notation for the interpretation instead of the λ^{I} used in [Mai09]

the typing and conversion rules for that particular type constructor. For example in `calc/setPi.elpi` we keep all definitions related to dependent products, like the implementation of Π-I or the interpretation of a function term.

```
of (lambda B F) (setPi B C) IE
    :- ofType B _ IE
    ,  (pi x\ locDecl x B => isa (F x) (C x) IE)
    .

interp (lambda B F) R
    :- spy(of (lambda B F) (setPi B C) ext)
    ,  spy(interp (setPi B C) (setSigma (setPi Bi Ci) H ))
    ,  macro_interp B ( x\_\_\xi\_\_\ interp (F x) (Fi xi))
    ,  setoid_eq B EquB
    ,  macro_interp B (x1\x2\h\x1i\x2i\hi\tau_proof_eq
        (F x1)
        (F x2)
        (C x2)
        (K_EQU x1i x2i hi))
    ,  R = pair (setPi Bi Ci) (H) (lambda Bi Fi)
        (forall_lam Bi x1\
         forall_lam Bi x2\
             forall_lam (EquB x1 x2) h\ K_EQU x1 x2 h)
    .
```

The last two, `test` and `debug`, hold utilities to quickly test the predicates from `main` and `calc` in addition to useful predicates needed to inspect the runtime behaviour of the type checking and of the interpretation.

3.2 Explanation of the Predicates

We can divide the predicate defined in our implementation in three groups: judgement predicates, conversion predicates and interpretation predicates.

3.2.1 `of`, `isa`, `ofType`, `isaType` and `locDecl`

In the first group are the predicate corresponding to the judgements $A\ type\ [\Gamma]$ or $a \in A\ [\Gamma]$. We use the `of Term Type Level` predicate to represent the judgement $Term \in Type\ [\Gamma]$ at the level indicated by the third argument, we also define another predicate `isa Term Type Level` to check if the inferred type of `Term` is convertible with `Type` at level `Level`; a major difference between these two predicate is that for `of` the first and last argument are inputs and the middle is an output,

while for `isa` all three arguments are inputs. Similarly `ofType Type Kind Level` and `isaType Type Kind Level` implement the corresponding functionality at the type level.

`locDecl` is used to form the context where `of` will look up the types of variables via the clause `of X Y _ :- locDecl X Y.`. We have no need to have a parameter to indicate the level for `locDecl` because extensional and intensional variables are disjoint and we can assume that we will never try to compute the extensional type of an intensional variable. Moreover as of now our implementation validates the ξ-rule at the intensional level so that also the conversion can be level agnostic.

3.2.2 `pts_*` Predicates

In the Minimalist Foundation in general the kind of a type is not unique but admit a most general unifier, this unifier will be the intended output of `ofType` and the logic to find it is implemented in the various PTS predicates. As an example a dependent product $\Pi(B, C)$ is never a proposition and is a set if both A *set* and $B(x)$ *set* $[x \in A]$ hold.

This can be expressed with the following code:

```
pts_leq A A.
pts_leq set col.
pts_leq props set.
pts_leq propc col.
pts_leq props propc.
pts_leq props col.

pts_fun A B set :- pts_leq A set, pts_leq B set, !.
pts_fun _ _ col.
```

Other variations allow to decide the most general kind of universal/existential quantifiers, connectives or lists.

3.2.3 `conv`, `dconv`, `hnf` and `hstep`

While conversion at the intensional level is almost straightforward at the extensional level we need to be more careful as a naive implementation would be extremely inefficient due to the rule Eq-E

$$\frac{\mathsf{true} \in \mathsf{Eq}(C, c, d)\ [c \in C, d \in C]}{c = d \in C}$$

The solution we have chose to implement consist of two step, first we weaken the Eq-E rule to perform a context lookup instead of trying to prove a judgement and then we divide the conversion in three different operations.

```
conv A A :- !.
conv A B :- locDecl _ (propEq _ A B).
conv A B :- (hnf A A'), (hnf B B'), (dconv A' B').
```

The predicate that will actually implement the conversion judgement will be `conv`, for this predicate both arguments will be inputs. When called between two terms the predicate will first check if the two terms are unifiable at the meta-level, if this fail then it will search for a context declaration that states the equality it wants; if this also fail then if will reduces both term to head normal form and apply a `dconv` step on the head normal forms. `dconv` role is to propagate the conversion to subterms; it assumes to receive two terms in head normal form (as it will only ever be called on output of `hnf`) and and will check for conversion on the corresponding subterms of its arguments.

```
dconv (setSigma B C) (setSigma B' C')
    :- (conv B B')
    ,  (pi x\ locDecl x B => conv (C x) (C' x))
    .
dconv (pair B C BB CC) (pair B' C' BB' CC')
    :- (conv B B')
    ,  (pi x\ locDecl x B => conv (C x) (C' x))
    ,  (conv BB BB)
    ,  (conv CC CC')
    .
dconv (elim_setSigma Pair M MM) (elim_setSigma Pair' M' MM')
    :- (conv Pair Pair')
    ,  (of Pair (setSigma B C))
    ,  (pi z\ locDecl z (setSigma B C) => conv (M z) (M' z))
    ,  (pi x\ pi y\  locDecl x B
        => locDecl y (C x)
        => conv (MM x y) (MM' x y))
    .
```

In general it will be needed to declare a `dconv` clause for each new non-constant constructor (constant constructors are all handled by a `dconv A A :- !.` clause) the main reason to use `dconv` is that is does not trigger the context lookup associated with the Eq-E rule.

3.2.4 `tau_*`, `setoid_*` and `interp`

The main entry point of the interpretation is the predicate `interpr`, for example the typical structure of our test is

1. Construct a suitable extensional type
2. Typecheck the type at the exstensional level
3. Interpret the type to the instensional level
4. Typecheck this last result.

During the interpretation many other auxiliary predicates are needed. Here we will give a description of the most important ones.

As explained above during the interpretation we need to correct the types of interpreted terms to compensate for the different rules at the two levels; in our code we have chosen to implement this functionality with the predicate `tau T1 T2 P`. `tau` takes in input 2 mutually convertible extensional types and return a λProlog function which convert elements of type `T1` to elements of type `T2`. It is important to note that `P` is a pure λ-function at the meta-level and thus in the conversion between the two types `T1` and `T2` we can not use backtracking or patter matching; thankfully a pure function is all we need because canonical isomorphism only apply η-expansions and do not actually inspect the term they are applied to, only its type and that is completely know to `tau`. As an example we show how this type correction would look like for a *disjoint sum* for a variable $w \in [\![B(x) + C(x)]\!]$ $[\![x \in A]\!]$ when transitioning to a type $[\![B(x') + C(x')]\!]$

$$\sigma_x^{x'}(w) \equiv \mathsf{Elim}_+(w, (y_1).\sigma_x^{x'}(y_1), (y_2).\sigma_x^{x'}(y_2))$$

.

We can see that we have applied structural recursion over the types even when computing a function over terms, this is helpful when treating types with more than one canonical constructor and when dealing with variables, but in general it is necessary since we cannot assume that all terms will have a canonical head normal form.

It is important to note that all predicates working at the interpretation level will always assume to handle well formed and well typed terms and types so that we do not need to fill our code with an enormous amount of runtime checks.

A related predicate is `tau_trasp`, which at the moment is conceptually the most complex predicate in the code and its aim is to implement the second and third property of required of substitution morphisms: preserving equalities of Q(mTT).

In formulas this means that the following hypothetical judgement should be derivable for some choice of d_2

$$d_2 \in \sigma^{x_2}_{x_1}(y) =_{B(x_2)} \sigma^{x_2}_{x_1}(y')\ [x_1 \in A,\ x_2 \in A,\ d_1 \in x_1 = x_2,\\ y \in B(x_1),\ y \in B(x_2), w \in y =_{B(x_1)} y'] \quad (3.1)$$

`tau_trasp` role is exactly to find a constructive and functional (after taking as inputs $B(x_1)$ and $B(x_2)$) transformation that produces such d_2 from y, y' and d_1.

This predicate becomes most useful when during the interpretation of dependent products, since in Q(mTT) a dependent function is represented as a pair containing a mTT function f and a proof that f preserve the extensionality of Q(mTT), similarly to how it is done in [Nec97].

Another problem is that in dealing with dependent products is that we need to automatically generate a proof that the interpretations of emTT function will preserve the equalities of Q(mTT). The generation of these proofs is handled by the predicate `tau_proof_eq`.

A family of related predicates is built around the setoids of Q(mTT), the principal one `setoid_eq` takes an extensional type in input and return an mTT proposition representing the setoid equality of the interpretation of the given type, three related predicates `setoid_refl`, `setoid_symm` and `setoid_trans` produce proofs that the proposition returned by `setoid_eq` is indeed an equivalence relation.

References

[Dun+15] Cvetan Dunchev et al.
"ELPI: Fast, Embeddable,\lambda Prolog Interpreter".
In: *Logic for Programming, Artificial Intelligence, and Reasoning*.
Springer. 2015,
Pp. 460–468.

[Mai09] Maria Emilia Maietti.
"A minimalist two-level foundation for constructive mathematics".
In: *Annals of Pure and Applied Logic* 160.3 (2009), pp. 319–354.

[MS05] Maria Emilia Maietti and Giovanni Sambin.
"Toward a minimalist foundation for constructive mathematics".
In: *From Sets and Types to Topology and Analysis: Practicable Foundations for Constructive Mathematics* 48 (2005), pp. 91–114.

[Nec97] George C Necula.
"Proof-carrying code".

In: *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*.
ACM. 1997,
Pp. 106–119.

[NM88] Gopalan Nadathur and Dale Miller.
"An overview of Lambda-PROLOG".
In: (1988).

A General Predicates

```
%%— dependents products: setPi
type setPi mttType →
     (mttTerm → mttType) →
     mttType.
type lambda mttType →
     (mttTerm → mttTerm) →
     mttTerm.
type app mttTerm →
     mttTerm →
     mttTerm.

%%— propositional conjunction: and
type and mttType →
     mttType →
     mttType.
type pair_and mttType →
     mttType →
     mttTerm →
     mttTerm →
     mttTerm.
type p1_and, p2_and mttTerm → mttTerm.


%%— dependet sums: setSigma

type setSigma mttType →
     (mttTerm → mttType) →
     mttType.
type pair mttType →
     (mttTerm → mttType) →
     mttTerm → mttTerm →
     mttTerm.
type elim_setSigma mttTerm →
     (mttTerm → mttType) →
     (mttTerm → mttTerm → mttTerm) →
     mttTerm.

%%— existential quantifier: exist
```

```
type exist mttType →
     (mttTerm → mttType) →
     mttType.
type pair_exist mttType →
     (mttTerm → mttType) →
     mttTerm →
     mttTerm →
     mttTerm.

type elim_exist mttTerm →
     mttType →
     (mttTerm → mttTerm → mttTerm) →
     mttTerm.

%%— universal quantifier: forall
type forall mttType →
     (mttTerm → mttType) →
     mttType.
type forall_lam mttType →
     (mttTerm → mttTerm) →
     mttTerm.
type forall_app mttTerm →
     mttTerm →
     mttTerm.

%%— intensional propositional equality: propId
type propId mttType →
     mttTerm →
     mttTerm →
     mttType.
type id mttType →
     mttTerm →
     mttTerm.
type elim_id mttTerm →
     (mttTerm → mttTerm → mttType) →
     (mttTerm → mttTerm) →
     mttTerm.

%%— implication: implies
type implies mttType →
     mttType →
```

```
    mttType.
type impl_lam mttType →
    (mttTerm → mttTerm) →
    mttTerm.
type impl_app mttTerm →
    mttTerm →
    mttTerm.

%%— local definitions: letIn
type letIn mttType
    → mttTerm
    → (mttTerm → mttTerm)
    → mttTerm
    .

%%— propositional disjunction: or
type or mttType → mttType → mttType.
type inl_or, inr_or mttType
    → mttType
    → mttTerm
    → mttTerm
    .
type elim_or mttType
    → mttTerm
    → (mttTerm → mttTerm)
    → (mttTerm → mttTerm)
    → mttTerm
    .

%%— disjoint sum: setSum
type setSum mttType → mttType → mttType.
type inl, inr mttType →
    mttType →
    mttTerm →
    mttTerm.
type elim_setSum (mttTerm → mttType) →
    mttTerm →
    (mttTerm → mttTerm) →
    (mttTerm → mttTerm) →
    mttTerm.
```

```
%%— singleton set/unit type: singleton
type  singleton mttType.
type  star mttTerm.
type  elim_singleton mttTerm →
    (mttTerm → mttType) →
    mttTerm → mttTerm.

/  UNITYPED COMPUTATIONAL PREDICATES  /
type hstep, dconv, hnf, conv, interp A → A → prop.

/ MTT PREDICATES /
kind mttTerm, mttType, mttKind, mttLevel type.
type ext, int mttLevel.
type col, set, propc, props mttKind.


type locDecl mttTerm → mttType → prop.
type locDeclType mttType → mttKind → prop.
type ofType mttType → mttKind → mttLevel → prop.
type of, isa mttTerm → mttType → mttLevel → prop.

type locDef mttTerm → mttType → mttTerm → prop.

type forall mttType →
    (mttTerm → mttType) →
    mttType.

hnf A B ⊢ hstep A C, !, hnf C B.
hnf A A.

conv A A ⊢ ! .
conv A B ⊢ (locDecl _ (propEq _ A B) ).
conv A B
    ⊢ spy(hnf A A')
    ,  spy( hnf B B')
    ,  spy( dconv A' B')
    .

dconv A A ⊢ !.

pts_leq A A.
```

```
pts_leq props set.
pts_leq props col.
pts_leq props propc.
pts_leq set col.
pts_leq propc col.


pts_prop props props props ⊢ !.
pts_prop _ _ propc.

pts_fun A B set
    ⊢ spy(pts_leq A set)
    , spy(pts_leq B set)
    , !
    .
pts_fun _ _ col.

pts_for A props props ⊢ pts_leq A set, !.
pts_for _ _ propc.

%ofType A KIND IE ⊢ locDeclType A KIND.

isaType Type Kind IE
    ⊢ spy(ofType Type Kind' IE)
    , spy(pts_leq  Kind' Kind)
    .

of (fixMe2 M T ) T int
    ⊢ !
    , print "|<| Found a FixMe!  |>|"
    , print M
    , term_to_string T S, print S
    .

isa (fixMe M) T int
    ⊢ !
    , print "|<| Found a FixMe!  |>|"
    , print M
    , term_to_string T S, print S
    .
```

```
isa Term TY IE
    ⊢ spy(of Term TY' IE)
    ,  spy(conv TY' TY)
    .


of X Y _ ⊢  locDecl X Y .


tau_proof_eq A A T H'
    ⊢ interp_isa A T Ai
    ,  setoid_refl T H
    ,  H' = H Ai
    .

tau_proof_eq A B T Hi
    ⊢ spy(locDecl H (propEq T' A B)), !
    ,  spy(interp_isa H (propEq T A B) Hi)
    .

tau_proof_eq A B T Hi
    ⊢ (locDecl H (propEq T' B A))
    ,  spy(interp_isa H (propEq T B A) Hi')
    ,  spy (setoid_symm T Q)
    ,  Hi = Q Hi'
    .



tau A A (x \ x) ⊢ !.

tau_trasp A A (x\y\h\ h) ⊢ !.


%interpret X:_ext T in un Xi di tipi Ti
interp_isa X T Xi
    ⊢ spy(of X T_inf ext)
    ,  spy(interp X Xi')
    ,  spy(tau T_inf T F)
    ,  spy(Xi = F Xi')
```

```
    .

locDecl (k_propId Te)
    (forall T t1\
        forall T t1'\
        implies (E t1 t1')
            (forall T t2\ forall T t2'\
                implies (E t2 t2')
                    (implies (E t1 t2)
                        (E t1' t2'))))
    ⊢ interp Te T
    ,  setoid_eq Te E
    .

setoid_symm T (x\ fixMe "prova di symmetria").

macro_tau B B' Q
    ⊢ spy(setoid_eq B EquB)
    ,  spy(interp B Bi)
    ,  spy(interp B' Bi')
    ,  spy(pi x1 \ pi x2 \ pi h\
         pi x1i\ pi x2i\ pi hi\
            locDecl x1 B
        ⇒ locDecl x2 B'
        ⇒ locDecl x1i Bi
        ⇒ locDecl x2i Bi'
        ⇒ interp x1 x1i
        ⇒ interp x2 x2i
        ⇒ (locDecl h (propEq B x1 x2))
        ⇒ (locDecl hi (EquB x1i x2i))
        ⇒ interp h hi
        ⇒ spy(Q x1 x2 h x1i x2i hi)
        )
    .
macro_interp B Q ⊢ macro_tau B B Q.
```

B Dependent Products

```
ofType (setPi B C) KIND3 IE
    ⊢ (ofType B KIND1 IE)
    ,  (pi x\ locDecl x B
        ⇒ (ofType (C x) KIND2 IE))
    ,  spy(pts_fun KIND1 KIND2 KIND3)
    .

of (lambda B F) (setPi B C) IE
    ⊢ spy (ofType B _ IE)
    ,  spy (pi x\ locDecl x B ⇒ isa (F x) (C x) IE)
    .

of (app Lam X) (CX) IE
    ⊢ spy(of Lam (setPi B C) IE)
    ,  spy(isa X B IE)
    ,  CX = C X
    .


hstep (app LAM Bb) (F Bb)
    ⊢ of LAM (setPi B C) IE
    ,  (ofType B _ IE)
    ,  (isa Bb B IE)
    ,  hnf LAM (lambda B' F)
    ,  conv B B'
    ,  (pi x\ locDecl x B ⇒ isa (F x) (C x) IE)
    ,  (pi x\ locDecl x B ⇒ ofType (C x) _ IE)
    .

dconv (setPi B C) (setPi B' C')
    ⊢ (conv B B')
    ,  (pi x\ locDecl x B ⇒ conv (C x) (C' x))
    .
dconv (app F X) (app F' X')
    ⊢ (conv F F')
    , (conv X X')
    .

dconv (lambda B F) (lambda B' F')
    ⊢ (conv B B')
```

```
    ,  pi x\  locDecl x B ⇒ (conv (F x) (F' x))
    .

interp (setPi B C) T
    ⊢ spy(interp B Bi)
    ,  spy(pi x\ pi xi\ locDecl x B
        ⇒ locDecl xi Bi
        ⇒ interp x xi
        ⇒ interp (C x) (Ci xi))
    ,  spy(setoid_eq B EquB)
    ,  spy(pi x\ pi xi\ locDecl x B
        ⇒ locDecl xi Bi
        ⇒ interp x xi
        ⇒ setoid_eq (C x) (EquC xi))
    ,  spy(pi x1 \ pi x2 \ pi h\
         pi x1i\ pi x2i\ pi hi\ locDecl x1 B
        ⇒ locDecl x2 B
        ⇒ locDecl x1i Bi
        ⇒ locDecl x2i Bi
        ⇒ interp x1 x1i
        ⇒ interp x2 x2i
        ⇒ (locDecl h (propEq B x1 x2))
        ⇒ (locDecl hi (EquB x1i x2i))
        ⇒ interp h hi
        ⇒ spy(tau (C x1) (C x2)
            (TauC x1i x2i hi)))
    ,  T = setSigma (setPi Bi Ci) f\
         (forall (Bi) x1\
         (forall Bi x2\
         (forall (EquB x1 x2) h\
         (EquC x2
             (TauC x1 x2 h (app f x1))
             (app f x2)))))
    .


interp (app F X) R
    ⊢ spy(of F (setPi B C) ext)
    ,  spy(interp_isa X B Xi)
    ,  spy(interp F Fi)
    ,  spy(of Fi T int)
```

```
    ,  spy(T = (setSigma PI _))
    ,  R = (app
        (elim_setSigma Fi (_\PI) (x\y\x) )
        Xi)
    .


interp (lambda B F) R
    ⊢ spy(of (lambda B F) (setPi B C) ext)
    ,  spy(interp (setPi B C)
        (setSigma (setPi Bi Ci) H ))
    ,  macro_interp B
        ( x\_\_\xi\_\_\ interp (F x) (Fi xi))
    ,  setoid_eq B EquB
    ,  macro_interp B
        (x1\x2\h\x1i\x2i\hi\
            tau_proof_eq (F x1) (F x2) (C x2)
                (K_EQU x1i x2i hi))
    ,  R = pair (setPi Bi Ci) (H) (lambda Bi Fi)
        (forall_lam Bi x1\
        forall_lam Bi x2\
        forall_lam (EquB x1 x2) h\
            K_EQU x1 x2 h)
    .


setoid_eq (setPi B C) P
    ⊢ spy(interp B Bi)
    ,  spy(pi x\ pi xi\ locDecl x B
        ⇒ locDecl xi Bi
        ⇒ interp x xi
        ⇒ interp (C x) (Ci xi))
    ,  spy(pi x\ pi xi\locDecl x B
        ⇒ locDecl xi Bi
        ⇒ interp x xi
        ⇒ (interp (C x) (Ci xi)
            , setoid_eq (C x) (EquC xi)))
    ,  P = (f\ g\
        forall Bi x\
        EquC x
        (app (elim_setSigma f
```

```
            (_\setPi Bi Ci) (x\y\x) ) x)
        (app (elim_setSigma g
            (_\setPi Bi Ci) (x\y\x) ) x))
    .

tau_proof_eq (app F X1) (app F X2) T H
    ⊢ of F (setPi B T') ext
    ,  spy(tau_proof_eq X1 X2 B G)
    ,  spy(interp F Fi)
    ,  spy(of Fi (setSigma TyF MorF) int)
    ,  PI1 = (c\ elim_setSigma c
        (_\ TyF) (x \ y\ x))
    ,  P2Fi = elim_setSigma Fi
        (c \ MorF (PI1 c))
        (x\ y\ y)
    ,  spy(interp_isa X1 B X1i)
    ,  spy(interp_isa X2 B X2i)
    ,  spy(tau
        (propEq (T' X2) (app F X1) (app F X2))
        (propEq (T) (app F X1) (app F X2))
        TAU)
    ,  H = TAU
        (forall_app
        (forall_app
        (forall_app P2Fi X1i) X2i) G)
    .

tau (setPi B C) (setPi B' C') P
    ⊢ spy(interp (setPi B C) (setSigma T1 T2))
    ,  spy(T1 = setPi Bi Ci)
    ,  spy(interp (setPi B' C') (setSigma T1' T2'))
    ,  spy(T1' = setPi Bi' Ci')
    ,  spy(setoid_eq B' EquB')
    ,  spy(macro_interp B
        (_\x2\_\_\x2i\_\
            setoid_eq (C x2) (EquC x2i)))
    ,  spy(tau B' B FB)
    ,  spy(macro_tau B B'
        (x\x'\_\xi\xi'\_\
            tau (C x) (C' x') (FC' xi xi')))
    ,  spy(macro_interp B (x1\x2\_\x1i\x2i\_\
```

```
        tau (C x1) (C x2) (FCC x1i x2i)))
    ,  spy(tau_trasp B' B KB)
    ,  spy(macro_tau B B' x\x'\_\xi\xi'\hi\
        tau_trasp (C x) (C' x') (KC' xi xi' hi))
    ,  P = (w\ elim_setSigma w
        (_\setSigma T1' T2') f\p\
        pair T1' T2'
        (lambda Bi' x\ FC' (FB x) x (app f (FB x)))
        (forall_lam Bi' y1'\
        forall_lam Bi' y2'\
        forall_lam (EquB' y1' y2') d'\
        KC' (FB y2')
            y2'
            d'
            (FCC (FB y1') (FB y2') (app f (FB y1')))
            (app f (FB y2'))
            (forall_app
                (forall_app
                (forall_app p (FB y1'))
                (FB y2'))
                (KB y1' y2' d'))))
    .

tau_trasp (setPi B C) (setPi B' C') P
    ⊢ spy(macro_tau B B' x\x'\_\xi\xi'\hi\
        tau_trasp (C x) (C' x') (KC' xi xi' hi))
    ,  spy(tau B' B FB)
    ,  P = f\g\d\ forall_lam B' y'\
            KC' (FB y')
                y'
                d
                (app f (FB y'))
                (app g (FB y'))
                (forall_app d (FB y'))

    .
```

C Propositional Equalities

Extensional

```
%% calc_Eq.elpi


type propEq mttType → mttTerm →
     mttTerm → mttType.
type eq    mttType →
     mttTerm → mttTerm.

pts_eq K props ⊢ pts_leq K set, !.
pts_eq _ propc.

ofType (propEq A AA1 AA2) KIND ext
     ⊢ ofType A KIND' ext
     ,  pts_eq KIND' KIND
     ,  isa AA1 A ext
     ,  isa AA2 A ext
     .


of (eq C Cc) (propEq C Cc Cc) ext
     ⊢  spy(of Cc C ext)
     .



%dstep A B ⊢ of _ ()

dconv (propEq A AA1 AA2)
       (propEq A' AA1' AA2')
     ⊢ spy(conv A A')
     ,  spy(conv AA1 AA1')
     ,  spy (conv AA2 AA2')
     .

dconv (eq A AA) (eq A' AA')
     ⊢ conv A A'
     ,  conv AA AA'
     .

interp (propEq A Aa1 Aa2) R
```

```
    ⊢ spy(setoid_eq A EquA)
    ,  spy(interp_isa Aa1 A Aa1')
    ,  spy(interp_isa Aa2 A Aa2')
    ,  spy(R = (EquA Aa1' Aa2'))
    .

interp (eq  A Aa)  T
    ⊢ spy(setoid_refl A ReflA)
    ,  spy(interp Aa Aa')
    ,  T = (ReflA Aa')
    .

setoid_refl (propEq _ _ _)
            (_\id singleton star).
setoid_eq (propEq A Aa1 Aa2)
          (_\ _\ (propId singleton star star)).

tau (propEq T_ T1 T2) (propEq T T1' T2') (F)
    ⊢ spy(tau_proof_eq T1 T1' T F1)
    ,  spy(tau_proof_eq T2 T2' T F2)
    ,  spy(interp_isa T1 T T1i)
    ,  spy(interp_isa T2 T T2i)
    ,  spy(interp_isa T1' T T1i')
    ,  spy(interp_isa T2' T T2i')
    ,  spy(interp T Ti)
    ,  F = x\ impl_app (
        impl_app (
        forall_app (
        forall_app (
        impl_app (
        forall_app (
        forall_app (k_propId T)
            T1i)
            T1i')
            F1)
            T2i)
            T2i')
            F2) x
    .
tau_trasp (propEq _ _ _ )
    (propEq _ _ _)
```

```
    (h\h'\k\ k).

tau_proof_eq  _ _ (propEq T A B)
    (id singleton star).
```

Intensional

```
ofType (propId A AA1 AA2) KIND IE
    ⊢ isa AA1 A int
    ,  isa AA2 A int
    ,  ofType A KIND1 int
    ,  (spy(pts_leq KIND1 set , KIND = props), !
       ;  KIND = propc)
    .

of (id A AA) (propId A AA AA) int
    ⊢ ofType A _ int
    ,  isa AA A int
    .

of (elim_id P C CC) (C AA1 AA2) int
    ⊢ (of P (propId A AA1 AA2) int)
    , (pi x\ pi y\ locDecl x A
        ⇒ locDecl y A
        ⇒ isaType (C x y) propc int)
    , (pi x\ locDecl x A
        ⇒ of (CC x) (C x x) int)
    .

hstep (elim_id (id A AA) C CC) (CC AA)
    ⊢ (isa AA A int)
    ,  (pi x\ pi y\ locDecl x A
        ⇒ locDecl y A
        ⇒ isaType (C x y) propc int)
    ,  (pi x\ locDecl x A
        ⇒ of (CC x) (C x x) int)
    .

dconv (id A AA) (id A' AA')
    ⊢ (conv A A')
```

```
      ,  (conv AA AA')
      .

dconv (propId A AA1 AA2) (propId A' AA1' AA2')
    ⊢ spy (conv A A')
      ,  spy (conv AA1 AA1')
      ,  spy (conv AA2 AA2')
      .
```