Adequate encodings of logical systems in UTT

Nikos Mylonakis
LSI Department
Universitat Politècnica de Catalunya (UPC)

Abstract. In this paper, we present an existing and formalized type theory (UTT (Luo, 1994) (Goguen, 1994)) as a logical framework. We compare the resulting framework with *LF* (Harper et al., 1993) and give the representation of two significant type systems in the framework: the typed lambda calculus which is closely related to higher-order logic and a linear type system which is not possible to encode in *LF*..

1. Introduction

Type theories were initially used as a logical language for the foundations of mathematics. Since they also include a computational language (in particular a functional language), most of them have also been used as a framework for program development. Some type theories have also been used as logical frameworks like for example the LF type theory (Harper et al., 1993). Other formalisms which have been used as logical frameworks are Isabelle (Paulson, 1994) or rewriting logic (Martí-Oliet et al., 1993)

LF can be seen as a pure type system, that is a three-level typed lambda calculus (level of elements, types and kinds) with dependent Π-types. LF has been used to make adequate encoding of different logics. The encoding is based on the idea of judgements as types, where judgements are seen as families of types of their proofs.

UTT (Luo, 1994)(Goguen, 1994) (Uniform Theory of dependent types) is a type theory which adds to the Extended Calculus of Constructions (ECC (Luo, 1994)) the possibility to define inductive types. The whole type theory is encoded in the Martin-Löf Logical Framework (Nordström et al., 1990). A more refined view of UTT can differentiate two different universes:

- A universe of types in which different types coexist:Σ-types (a dependent type of tuples) Π-types (a dependent type of functions) and inductive types.
- A universe of propositions in which a higher order intuitionistic logic is defined. We refer to this universe using the constant **Prop**.

Mixing inductive types and the universe of propositions one can define inductive relations. In this type theory, inductive relations can be

seen as functional types which given some arguments of the appropriate type, return the proposition one has to prove to guarantee that the tuple formed by the given arguments belongs to the relation. We will use inductive types and inductive relations for the encoding of syntax and the encoding of the proof systems of our logical systems.

In this paper, first we present a novel encoding of the typed lambda calculus in UTT together with an explicit proof of the correctness of the representation which is closely related to the encoding of higher-order logic tackling the problem of an explicitly defined higher-order substitution operation. After that, the encoding of a fragment of a linear type theory which is not possible to encode in LF is presented. These encodings are adequate, in the sense that there exists a bijection between the closed derivations of a concrete judgement of the proof systems and the inhabitants of the application of the judgement to the inductive relation which encodes the proof systems.

It is not the aim of the paper to compare the different existing logical frameworks with ours but to solve the technical problems to give adequate encodings of logical systems in UTT and to show that the increase of expressivity of UTT solves some important limitations of the closest logical framework to the one that we propose which is LF. On the other hand, this increase of expressiveness does not seem to affect significatively to the efficiency of the proof assistance of this type theory. See (Pollack, 1995) for a generic solution to proof assistance of these type theories. Therefore, we encourage the development of future generations of open proof checkers for type theories with dependent and inductive types to be used as logical frameworks. Note that the techniques presented in this paper have been applied for the development of generic proof support for algebraic specification languages ((Hennicker et al, 1997), (Hennicker, 1997)) in (Mylonakis, 2000). See also (LEGO, 1998) for many other applications of UTT using the current proof checker of the type theory.

In the following, we present some of the advantages of UTT together with the new principle of encoding with respect to LF:

- In UTT it is possible to formalize metatheory of the encoded logic using a higher order logic with inductive principles associated to inductive types and inductive relations whereas LF has not this feature.

- Since in UTT the encodings of variables and contexts of sequents of logics are also encoded as inductive types (as a parameter of the inductive relation which encodes the logic), the properties of the type theory like for example weakening are not inherited by the encoded logic. These properties have to be proven for the concrete

object logic using for example the induction principles associated to the inductive relation which encodes the object logic. Thus, for example the consequence relation of the object logic does not have to be intuitionistic as in LF. See for example (Gardner, 1992) for a formal explanation.

– Finally, we believe that our encodings are more readable and easy to use in practice than the ones in LF since we do not use higher-order abstract syntax and our substitution operation does not depend on the implementation of $\beta-$reduction of the proof checker of LF which can eventually perform renamings of variables to avoid name clashes. In our approach, the encoding of syntax is more similar to the informal usual notation and the names of variables are preserved under substitution and from the encoded formulae we can always recover the original names of variables. This might not be very relevant for the encoding of first order logic, but we think that this is really important for the encoding of higher-order calculi including modularity or concurrency.

2. Logical Systems

The proof systems which are encoded in LF are usually formulated as natural deduction systems. See (Gardner, 1992) for a formal description of these systems. Basically, these systems are defined by a finite set of natural deduction rules. These kind of rules are defined by a set of n premises, a conclusion, and side conditions are allowed. *Premises* and *conclusions* are defined by sequents with schematic variables and therefore a rule denotes in general a set of (n+1)-tuples of sequents. An *instance* of a rule is a (n+1)-tuple of sequents of this set. In general, sequents are defined with judgements and for example the only judgement which is used to define first-order logic is $\phi \ \ true$ which means that the formula ϕ is derivable.

The sequent to define natural deduction systems in (Gardner, 1992) is $\Gamma \Rightarrow_X J$ where Γ is a set of judgements (normally referred as environment), J is a judgement and X a finite set of variables.

We will always work with logical systems defined by a finite set of rules with premises, conclusion and side conditions as in (Gardner, 1992) but we will not work with just the type of sequents presented in (Gardner, 1992): within a logical system, different kind of sequents can be defined.

In the following, we give some extra definitions of logical systems which are needed for the presentation of the encoding in UTT and its

proof of adequacy. The main definitions are given for sequents of the form $\Gamma \Rightarrow_X J$ but they are easily generalisable for any kind of sequents.

Definition 2.1 *The sequent $\Gamma \Rightarrow_X J$ is closed iff J and all the judgements in Γ are closed under X.*

Remark: *In the following, for any sequent S of a logical system Π including a set of free variables, we will assume predefined the property of closedness in the obvious equivalent way.*

Definition 2.2 *A rule is closed if the sequents of the premises and the sequent of its conclusion are closed.*

Definition 2.3 *The set of derivations of a sequent $\Gamma \Rightarrow_X J$ in the logical system Π is denoted by $\Delta_\Pi(\Gamma \Rightarrow_X J)$ and recursively defined as follows:*

- *if $J \in \Gamma$ then $\Gamma \Rightarrow_X J \in \Delta_\Pi(\Gamma \Rightarrow_X J)$.*
- *if $r \in \Pi$, $(\Gamma \Rightarrow_X J, \Gamma_1 \Rightarrow_{X_1} J_1, \ldots, \Gamma_n \Rightarrow_{X_n} J_n)$ is an instance of the rule r, $\delta_1 \in \Delta_\Pi(\Gamma_1 \Rightarrow_{X_1} J_1)$, ... and $\delta_n \in \Delta_\Pi(\Gamma_n \Rightarrow_{X_n} J_n)$, then $r(\Gamma \Rightarrow_X J, [\delta_1, \ldots, \delta_n]) \in \Delta_\Pi(\Gamma \Rightarrow_X J)$.*

Remark: *In the following, we will denote by $\Delta_\Pi(\mathcal{S})$ the set of derivations of the sequent $\mathcal{S}$ in the logical system Π and we will denote just by Δ_Π the whole set of derivations of the logical system Π.*

Definition 2.4 *A derivation of a sequent is closed iff the sequent is closed and its subderivations are closed, where the subderivations are the derivations of the instances of the first-rule premises of the derivation.*

3. Notation

Inductive types and inductive relations are defined in UTT by a set of constructors. Additionally, we will always assume predefined the induction principle and the primitive recursive operator associated to the inductive definition to define primitive recursive functions on that type. For example, for the inductive type $List : Type_0 \rightarrow Type_0$, which is defined by the following set of constructors

$$nil : \Pi A : Type_0 . List\ A$$
$$cons : \Pi A : Type_0 . A \ \rightarrow \ (List\ A) \ \rightarrow \ (List\ A)$$

the induction principle $Ind(List\ A)$ for any type $A : Type_0$ which we will use to reason about propositions of type $(List\ A) \to Prop$ is the following:

$$\Pi P : (List\ A)\ \to\ Prop.(P\ (nil\ A))\ \supset \\ (\forall a : A.\forall l : List\ A.(P\ l) \supset (P\ (cons\ A\ a\ l)))\ \supset\ (\forall l : List\ A.P\ l)$$

and the primitive recursion principle $Primrec\ (List\ A)$ with arity

$$Primrec\ (List\ A)\ : T \to\ (A \to (List\ A) \to T \to T)\ \to\ (List\ A) \to\ T$$

for any type $T : Type_0$ has the following computational rules:

$$Primrec\ (List\ A)\ bcl\ gcl\ (nil\ A)\ \ \to\ \ bcl$$
$$Primrec\ (List\ A)\ bcl\ gcl\ (cons\ A\ a\ l)\ \ \to$$
$$(gcl\ A\ a\ l\ (Primrec\ (List\ A)\ bcl\ gcl\ l))$$

For example the function reverse of a list with type $reverse : \Pi T : Type_0.(List\ T)\ \to\ (List\ T)$ is defined as follows:

$$\begin{aligned} reverse\ T\ l\ =\ & Primrec\ (List\ T)\ revbc\ revgc\ l \\ & where \\ & revbc\ =\ nil\ T \\ & revgc\ a\ l\ l'\ =\ add\_last\ a\ l' \end{aligned}$$

4. Adequate encoding of the typed lambda calculus

In this subsection we are going to present the adequate encoding of the typed lambda calculus and its substitution operation. One of the original formulation of the typed lambda calculus has the following three rules:

$$\frac{}{X \blacktriangleright x : \tau}\ x : \tau\ \in X \qquad (Ass)$$

$$\frac{X \cup \{x : \tau\} \blacktriangleright e : \tau'}{X \blacktriangleright \lambda x : \tau.e : \tau \to \tau'} \qquad (ABS)$$

$$\frac{X \blacktriangleright e : \tau \to \tau' \quad X \blacktriangleright e' : \tau}{X \blacktriangleright e\ e' : \tau'} \qquad (APPL)$$

where the possible types $(Type_{TLC}(B))$ are generated by a set of base types B and the constructor $\tau \to \tau'$ where $\tau, \tau' \in Type_{TLC}(B)$ and the set of preterms (variables, lambda abstraction and application) are denoted by $Term_{TLC}(B)$

An alternative presentation of the typed lambda calculus is to split the set of free variables in two: the initial set of free variables of the derivation and the set of bound variables of a variables which become free in the derivation process. We will denote this new set of free variables as a pair of the form (X, X') where the first is the initial set of free variables and the second the set of bound variables which have become free, and if the second component is empty we will normally denote the set $(X, [])$ just by X.

This split will be used to determine the difference between the last DeBruijn index assigned to the bound variables in the scope of every ocurrence of a variable in a higher-order term and the last index assigned in the original set of free variables. This index (which will be referred as bound level and it is an information which every variable in a higher-order term has) is necessary to update the indexes of the variables of the higher-order term which replaces a variable in the substitution operation.

Thus, the new formulation of the alternative definition of the typed lambda calculus has the following four rules:

$$\frac{}{(X, X') \blacktriangleright x : \tau} \; x : \tau \notin X', x : \tau \in X \qquad (Ass1)$$

$$\frac{}{(X, X') \blacktriangleright x : \tau} \; x : \tau \in X' \qquad (Ass2)$$

$$\frac{(X, X' \cup \{x : \tau\}) \blacktriangleright e : \tau'}{(X, X') \blacktriangleright \lambda x : \tau . e : \tau \to \tau'} \qquad (ABS)$$

$$\frac{(X, X') \blacktriangleright e : \tau \to \tau' \quad (X, X') \blacktriangleright e' : \tau}{(X, X') \blacktriangleright e \; e' : \tau'} \qquad (APPL)$$

And the substitution operation $\_\{\_/\_\} : Term_{TLC} \to Term_{TLC} \to X \to Term_{TLC}$ is inductively defined as follows:

$$\begin{array}{lcll} y \; \{t \; / \; x\} & = & t & , if x = y \\ & = & y & , otherwise \\ \lambda x : \tau . e \; \{e' \; / \; y\} & = & \lambda x' : \tau . ((e\{x'/x\}) \; \{e' \; / \; y\}) & , if x \neq y \\ & = & \lambda x : \tau . e & , if x = y \end{array}$$

$$\begin{array}{l} where \\ x \notin FV(e') \Rightarrow x' = x \wedge \\ x \in FV(e') \Rightarrow x' \notin FV(e') \wedge x' \notin FV(e) \wedge x' \notin BV(e), \end{array}$$

$$e \; e' \; \{t \; / \; x\} = e \; \{t \; / \; x\} \; (e' \; \{t \; / \; x\})$$

where $FV(e)$ denotes the set of free variables of e and $BV(e)$ denotes the set of bound variables of e in the usual way.

As we have mentioned in previous sections, for the encoding of this type system, the encoding of variables is not trivial and requires additionally to the variable name and its type, two variable indexes: one to denote the DeBruijn index and the other to denote the bound level of the variable. Variable names are defined as non-empty strings of characters and since we can assume that the infinite set of variables is countable, variable indexes are trivially defined as inductive types. Both indexes are assigned during the encoding of terms. The DeBruijn index for bounded variables start from the greatest index assigned to the free variables to avoid name clashes.

The DeBruijn index of the bound variables of the term which replaces a variable in the substitution operation must be updated. This update uses the bound level of the variable to be replaced. Additionally, the bound level of all the variables of the term which replaces a variable must also be updated using the bound level of the variable which is replaced. Note that we do not lose readability in this process because we always preserve the original names of the variables.

4.1. Encoding of variables

First we define the encoding of variables symbols, variable names as non-empty strings of variable names, and variable indexes isomorphic to the naturals numbers.

Definition 4.1 *The type Var_symbol is inductively defined by the following set of constructors:*

$$\begin{array}{l} a, \ldots, z : Var\_symbol \\ A, \ldots, Z : Var\_symbol \\ \_, ', \$: Var\_symbol \end{array}$$

Definition 4.2 *For any type $T : Type_0$, the inductive type $Nelist\, T$ is defined by the following constructors:*

$$\begin{array}{l} first\_Nel \ : T \ \rightarrow \ Nelist\, T \\ cons\_Nel \ : \ T \ \rightarrow \ (Nelist\, T) \ \rightarrow \ (Nelist\, T) \end{array}$$

Definition 4.3 *The type Var_name is defined as follows:*

$$Var\_name \ = \ Ne\_list\, Var\_symbol$$

Definition 4.4 *The type Var_index is inductively defined by the following set of constructors:*

$$first\_Vi : Var\_index$$
$$next\_Vi : Var\_index \rightarrow Var\_index$$

We assume predefined the following functions on variable indexes::

- $Eqbool\_Vi : Var\_index \rightarrow Var\_index \rightarrow Bool$ which is the boolean equality on variable indexes.
- $Ltbool\_Vi : Var\_index \rightarrow Var\_index \rightarrow Bool$ which is the function lower than on variable indexes.
- $add\_Vi : Var\_index \rightarrow Var\_index \rightarrow Var\_index$ which adds two variable indexes like they were naturals.
- $decr\_Vi : Var\_index \rightarrow Var\_index \rightarrow Var\_index$ which decrements a variable index like it was a natural.
- $substract\_Vi : Var\_index \rightarrow Var\_index \rightarrow Var\_index$ which substracts two variable indexes like they were naturals.

Next, we define the higher-order types of variables and higher-order variables.

Definition 4.5 *The inductive types $Holtype$ for a given set of base types B is defined by the following set of constructors:*

$$\{\, b\_Holt : Holtype \mid b \in B \,\} \cup$$
$$\{\, func\_Holt : Holtype \rightarrow Holtype \rightarrow Holtype \}$$

We assume predefined the equality function $Eqbool\_Hty : Holtype \rightarrow Holtype \rightarrow Bool$

Definition 4.6 *The type $Holvar$ is defined as:*

$$Holvar = pair\ Var\_name\ Holtype$$

Higher-order variables with indexes are defined as higher-order variables with two indexes: the first is the deBruijn index and the second is the bound level of the variable which is the number of bound variables which has the scope of an ocurrence of a variable in a term.

Definition 4.7 *The type $Holinvar$ is defined as:*

$$Holinvar = pair\ Holvar\ (pair\ Var\_index\ Var\_index)$$

We assume predefined the following function on $Holvar$ and $Holinvar$:

- $Eqbool\_Hvar : Holvar \to Holvar \to Bool$ which is the boolean equality function on higher-order variables.
- $Eqbool\_Hivar : Holinvar \to Holinvar \to Bool$ which is true if the two higher-order variables and their deBruijn indexes (not the bound level) are equal.
- $getindex\_Hiv : Holinvar \to Var\_index$ which given a higher-order variable with indexes returns the DeBruijn index.
- $getblevel\_Hiv : Holinvar \to Var\_index$ which given a higher-order variable with indexes returns the bound level.
- $assindex\_Hiv : Holinvar \to Var\_index \to Holinvar$ which given a higher-order variable with indexes and a variable index, assigns the variable index as deBruijn index to the variable.
- $assblevel\_Iv : Holinvar \to Var\_index \to Holinvar$ which given a higher-order variable with indexes and a variable index, assigns the variable index as bound level to the variable.
- $addindex\_Hiv : Holinvar \to Var\_index \to Holinvar$ which given a higher-order variable with indexes and a variable index, adds the variable index with the deBruijn index of the variable
- $addblevel\_Hiv : Holinvar \to Var\_index \to Holinvar$ which given a higher-order variable with indexes and a variable index, adds the variable index with the bound level of the variable.

4.2. Encoding of variable sets

Variable sets are defined as pairs of two pairs of a variable index and list of higher-order variables with indexes. The first pair denotes the set of free variables together with the last deBruijn index assign to the set of free variables and the second pair denotes the set of bound variables together with the last deBruijn index assign to bound variables. The deBruijn indexes of bound variables are always assigned after the deBruijn indexes of free variables.

Definition 4.8 *The type $Holvar\_set$ is defined as:*

$$Holvar\_set = pair\ (pair\ Var\_index\ (List\ Holinvar))\ (pair\ Var\_index\ (List\ Holinvar))$$

We assume predefined the following functions on $Holvar\_set$:

- $empty\_Hvst : Holvar\_set$ which returns the empty variable set
- $addfvar\_Hvst : Holvar \to Holvar\_set \to Holvar\_set$ which given a higher-order variable and a variable set, adds a free higher-order variable with indexes to the variable set. The bound level of the variable is always the first variable index.
- $addbvar\_Hvst : Holvar \to Holvar\_set \to Holvar\_set$ which given a higher-order variable and a variable set, adds a bound higher-order variable with indexes to the variable set. The bound level of the variable is always the first variable index.
- $getblevel\_Hvst : Holvar\_set \to Var\_index$ which given a variable set, returns the difference between the second variable index (the one of the bound variables) and the first variable index (the one of the free variables).
- $getvar\_Hvst : Holvar \to Holvar\_set \to Holinvar$ which given a higher-order variable hv and a variable set hvs, returns the higher-order variable with variable indexes with the greatest de-Bruijn index in the variable set and with bound level the index $getblevel\_Hvst\ hvs$.

Additionally, we define different inductive relations on $Holvar\_set$. First, and inductive relation which checks that a higher-order variable is in a list of higher-order variables with indexes

Definition 4.9 *The inductive relation*

$$Is\_in\_Hivl : \Pi v : Holvar.\Pi vs : List\ Holinvar.Prop$$

is defined by the following set of constructors:

$base\_Inhivl : \Pi hv : Holvar.\Pi hiv : Holinvar.\Pi hivl : List\ Holinvar.$
$\quad \Pi eqpr : (Eqbool\_Hvar\ hv\ (fst\ hiv)) =_{bool} true.$
$\quad Is\_in\_Hivl\ hv\ (cons\ hiv\ hivl)$

$genc\_Inhivl : \Pi hv : Holvar.\Pi hiv : Holinvar.\Pi hivl : list\ Holinvar.$
$\quad \Pi pr : Is\_in\_Hivl\ hv\ hivl.$
$\quad\quad Is\_in\_Hivl\ hv\ (cons\ Holinvar\ hiv\ hivl)$

Second, and inductive relation which checks that a higher-order variable is not in a list of higher-order variables with indexes

Definition 4.10 *The inductive relation*

$$Notisin\_Hivl : \Pi v : Holvar.\Pi vs : List\ Holinvar.Prop$$

is defined by the following set of constructors:

$$base\_Ninhivl : \Pi hv : Holvar.Notisin\_Hivl\ hv\ (nil\ Holinvar)$$

$$\begin{array}{l} genc\_Ninhivl : \Pi hv : Holvar.\Pi hiv : Holinvar.\Pi hivl : list\ Holinvar. \\ \quad \Pi eqpr : (Eqbool\_Hvar\ hv\ (fst\ hiv)) =_{bool} false. \\ \quad \Pi pr : Notisin\_Hivl\ hv\ hivl. \\ \quad\quad Notisin\_Hivl\ hv\ (cons\ Holinvar\ hiv\ hivl) \end{array}$$

After that, an inductive relation which checks that a higher-order variable is in the list of bound variables of a variable set and next an inductive relation which checks that a higher-order variable is not in the list of bound variables of a variable set.

Definition 4.11 *The inductive relation*

$$Isin\_boundv\_Hvs : \Pi hv : Holvar.\Pi vs : Holvar\_set.Prop$$

is defined by the following set of constructors:

$$\begin{array}{l} ctr\_Inbhvs : \Pi hv : Holvar.\Pi hvs : Holvar\_set. \\ \quad \Pi isinpr : Is\_in\_hivl\ hv\ (snd\ (snd\ hvs)).Isin\_boundv\_Hvs\ hv\ hvs \end{array}$$

Definition 4.12 *The inductive relation*

$$Notisin\_boundv\_Hvs : \Pi hv : Holvar.\Pi vs : Holvar\_set.Prop$$

is defined by the following set of constructors:

$$\begin{array}{l} ctr\_Ninbhvs : \Pi hv : Holvar.\Pi hvs : Holvar\_set. \\ \quad \Pi isinpr : Notisin\_hivl\ hv\ (snd\ (snd\ hvs)).Notisin\_boundv\_Hvs\ hv\ hvs \end{array}$$

Finally, an inductive relation which checks that a higher-order variable is in the list of free variables of a variable set.

Definition 4.13 *The inductive relation*

$$Isin\_freev\_Hvs : \Pi hv : Holvar.\Pi vs : Holvar\_set.Prop$$

is defined by the following set of constructors:

$$\begin{array}{l} ctr\_Inbhvs : \Pi hv : Holvar.\Pi hvs : Holvar\_set. \\ \quad \Pi isinpr : Is\_in\_hivl\ hv\ (snd\ (fst\ hvs)).Isin\_freev\_Hvs\ hv\ hvs \end{array}$$

4.3. Encoding of typed lambda terms and the substitution operation

In this subsection, we present the encoding of higher-order lambda terms and the substitution operation.

Definition 4.14 *The inductive type $Holterm$ is defined by the following set of constructors:*

$$holvar\_Htrm : Holinvar \rightarrow Holterm$$
$$abstr\_Htrm : Holinvar \rightarrow Holterm \rightarrow Holterm$$
$$appl\_Htrm : Holterm \rightarrow Holterm \rightarrow Holterm$$

In the following, we present the substitution operation on higher-order terms which given a variable index, a higher-order term ht, a higher-order term ht' and a free higher-order variable with indexes hiv, returns the higher-order term which is obtained by replacing all the appearences of the variable hiv in ht by ht'. Once a higher-order term is replaced by a variable, the variable indexes of the bound variables of the higher-order term must be updated and the bound level of every variable of the higher-order term must also be updated. The first parameter of the substitution operation (the first variable index which is not assigned to the set of free variables of ht and ht') is used to determine whether a variable is free or bound.

Definition 4.15 *The function*

$$subst\_Htrm : Var\_index \rightarrow Holterm \rightarrow Holterm \rightarrow Holinvar \rightarrow Holterm$$

are defined as follows:

$subst\_Htrm\ vi\ htrm\ htrm'\ hiv\ =$
  $Prim\_rec\ Holterm\ (holvarc\ vi\ htrm'\ hiv)\ (abstrc\ htrm'\ hiv)$
    $(applc\ htrm'\ hiv)\ htrm$
  $where$
  $holvarc\ vi\ htrm'\ hiv\ hiv'\ =$
    $Primrec\ Bool\ (update\_index\_Htrm\ vi\ (getblevel\_Hiv\ hiv')\ htrm')$
      $(holvar\_Htrm\ hiv')\ (Eqbool\_Hivar\ hiv\ hiv')$
  $abstrc\ htrm'\ hiv\ hiv'\ htrm\ htrmf\ =\ (abstr\_Htrm\ hiv'\ htrmf)$
  $applc\ htrm'\ hiv\ htrm\ htrm''\ htrmf\ htrmf''\ =$
    $appl\_Htrm\ htrmf\ htrmf''$

Definition 4.16 *The function $update\_index\_Htrm : Var\_index \rightarrow Var\_index \rightarrow Holterm \rightarrow Holterm$ is defined as follows:*

$update\_index\_Htrm\ vi\ bl\ htrm\ =\ Primrec\ Holterm$
  $(holvarc\ vi\ bl)\ (abstrc\ bl)\ applc\ htrm$
  $where$

$holvarc\ vi\ bl\ hiv\ =\ Primrec\ bool\ (addblevel\_Hiv\ bl\ hiv)$
$\quad (addblevel\_Hiv\ bl\ (addindex\_Hiv\ bl\ hiv))$
$\quad\quad Ltbool\_Vi\ (getindex\_Hiv\ hiv)\ vi)$
$abstrc\ bl\ hiv\ ht\ htf\ =$
$\quad abstr\_Htrm\ (addblevel\_Hivl\ bl\ (addindex\_Hivl\ bl\ hiv))\ htf$
$applc\ ht\ ht'\ htf\ htf'\ =\ appl\_Htrm\ htf\ htf'$

4.4. Encoding of the type system

The encoding of the type system of the typed lamda calculus is with an inductive relation with the same number of constructors as rules of the new definition of the type system:

Definition 4.17 *The inductive relation*

$$Wfhterm : Holvar\_set \to Holterm \to Holtype \to Prop$$

is defined by the following set of constructors:

$\{ass1\_tr : \Pi vs : Holvar\_set.\Pi hv : Holvar.$
$\quad \Pi pr : Notisin\_boundv\_Hvs\ hv\ vs.\Pi prin : Isin\_freev\_Hvs\ hv\ vs.$
$\quad Wfhterm\ \ vs\ \ (holvar\_Htrm\ \ (getvar\_Hvst\ hv\ vs))\ \ (snd\ hv)\} \cup$

$\{ass2\_tr : \Pi vs : Holvar\_set.\Pi hv : Holvar.\Pi pr : Isin\_boundv\_Hvs\ hv\ vs.$
$\quad Wfhterm\ \ vs\ \ (holvar\_Htrm\ (getvar\_Hvst\ hv\ vs))\ \ (snd\ hv)\} \cup$

$\{abs\_tr : \Pi vs : Holvar\_set.\Pi hv : Holvar.\Pi ht : Holterm.\Pi hty : Holtype.$
$\quad \Pi wft : Wfhterm\ (addbvar\_Hvst\ hv\ vs)\ ht\ hty.$
$\quad\quad Wfhterm\ vs\ (abstr\_Htrm\ (getvar\_Hvst\ hv$
$\quad\quad\quad (addbvar\_Hvst\ hv\ vs))\ ht)\ (func\_Holt\ (snd\ hv)\ hty),$

$appl\_tr : \Pi vs : Holvar\_set.\Pi ht, ht' : Holterm.\Pi hty, hty' : Holtype$
$\quad \Pi prt : Wfhterm\ vs\ ht\ (func\_Holt\ hty\ hty').$
$\quad \Pi prt' : Wfhterm\ vs\ ht'\ hty.$
$\quad\quad Wfhterm\ vs\ (appl\_Ht\ ht\ ht')\ hty'\}$

One can easily define encoding and decoding functions of types, variable names, list of variables, variable sets and higher-order terms with the following arities:

$$
\begin{array}{l}
\epsilon_{\tau} : Types(B) \rightarrow Holtype \\
\epsilon_{\tau}^{-1} : Holtype \rightarrow Types(B) \\
\epsilon_{vn} : X \rightarrow Var\_name \\
\epsilon_{vn}^{-1} : Var\_name \rightarrow X \\
\epsilon_{hvl} : [(X, Types(B))] \rightarrow (List\ Holvar) \\
\epsilon_{hvl}^{-1} : (List\ Holvar) \rightarrow [(X, Types(B))] \\
\epsilon_{vs} : ([X], [X]) \rightarrow (Holvar\_set) \\
\epsilon_{vs}^{-1} : (Holvar\_set) \rightarrow ([X], [X]) \\
\epsilon_{ht} : Holvar\_set \rightarrow Term(X) \rightarrow Holterm \\
\epsilon_{ht}^{-1} : Holvar\_set \rightarrow Holterm \rightarrow Term(X)
\end{array}
$$

and the encoding of derivations of typed lambda terms is inductively defined as follows:

$$
\begin{array}{l}
\epsilon_{td}\ Ass1((X, X') \blacktriangleright x : \tau) = \\
\quad ass\_tr(\epsilon_{vs}\ (X, X'))\ encx \\
\qquad (ctr\_Ninbhvs\ encx\ (\epsilon_{vs}\ (X, X')) \\
\qquad\quad (genc\_Ninhivl\ encx\ (getvar\_Hvst\ enchv'_n \\
\qquad\qquad (\epsilon_{vs}\ [hv'_1, \ldots, hv'_n])) \\
\qquad\quad (snd\ (snd\ (\epsilon_{vs}\ [hv'_1, \ldots, hv'_{n-1}]))) \\
\qquad\quad \lambda P : bool \rightarrow Prop.\lambda pr : P\ false.pr \\
\qquad\quad (\ldots(genc\_Ninhivl\ encx\ (getvar\_Hvst\ \ enchv'_n\ \epsilon_{vs}\ [\,]) \\
\qquad\quad (snd\ (snd\ (\epsilon_{vs}\ [\,]))) \\
\qquad\quad (base\_Ninhivl\ encx)) \ldots))) \\
\qquad (ctr\_Infhvs\ encx\ (\epsilon_{vs}\ (X, X')) \\
\qquad\quad (genc\_Inhivl\ encx\ (getvar\_Hvst\ enchv_n \\
\qquad\qquad (\epsilon_{vs}\ [hv_1, \ldots, hv_{i-1}, (x, \tau), hv_i, \ldots, hv_n])) \\
\qquad\quad (snd\ (fst\ (\epsilon_{vs}\ [hv_1, \ldots, hv_{i-1}, (x, \tau), hv_i, \ldots, hv_{n-1}]))) \\
\qquad\quad (\ldots(genc\_Inhivl\ encx\ (getvar\_Hvst\ enchv_i \\
\qquad\qquad (\epsilon_{vs}\ [hv_1, \ldots, hv_{i-1}, (x, \tau), hv_i])) \\
\qquad\quad (snd\ (fst\ (\epsilon_{vs}\ [hv_1, \ldots, hv_{i-1}, (x, \tau)])))) \\
\qquad\quad (base\_Hivs\ encx\ (getvar\_Hvst\ encx \\
\qquad\qquad (\epsilon_{vs}\ [hv_1, \ldots, hv_{i-1}, (x, \tau)])) \\
\qquad\quad (snd\ (fst\ (\epsilon_{vs}\ [hv_1, \ldots, hv_{i-1}, (x, \tau)])))) \\
\qquad\quad (\lambda P : bool \rightarrow Prop.\lambda pr : P\ true.pr))) \ldots)))
\end{array}
$$

$$
\begin{array}{l}
where\ X = [hv_1, \ldots, hv_{i-1}, (x,\tau), hv_i, \ldots, hv_n] \\
\quad encx\ =\ mkpair\ Holvar\ (\epsilon_{vn}\ x)\ (\epsilon_\tau\ \tau) \\
\quad enchv_n\ =\ mkpair\ Holvar\ (fst\ hv_n)\ (snd\ hv_n) \\
\quad enchv_i\ =\ mkpair\ Holvar\ (fst\ hv_i)\ (snd\ hv_i) \\
\quad X' = [hv'_1, \ldots, hv'_n] \\
\quad enchv'_n\ =\ mkpair\ Holvar\ (fst\ hv'_n)\ (snd\ hv'_n) \\
\quad \vdots \\
\quad enchv'_1\ =\ mkpair\ Holvar\ (fst\ hv'_1)\ (snd\ hv'_1)
\end{array}
$$

$$
\begin{array}{l}
\epsilon_{td}\ Ass2((X,X') \blacktriangleright x:\tau) = \\
\quad ass2\_tr(\epsilon_{vs}\ (X,X'))\ encx \\
\qquad (ctr\_Inbhvs\ encx\ (\epsilon_{vs}\ (X,X')) \\
\qquad\quad (genc\_Inhivl\ encx\ (getvar\_Hvst\ enchv'_n \\
\qquad\qquad (\epsilon_{vs}\ [hv'_1, \ldots, hv'_{i-1}, (x,\tau), hv'_i, \ldots, hv'_n])) \\
\qquad\quad (snd\ (snd\ (\epsilon_{vs}\ [hv'_1, \ldots, hv'_{i-1}, (x,\tau), hv'_i, \ldots, hv'_{n-1}]))) \\
\qquad\quad (\ldots (genc\_Inhivl\ encx\ (getvar\_Hvst\ enchv'_i \\
\qquad\qquad (\epsilon_{vs}\ [hv'_1, \ldots, hv'_{i-1}, (x,\tau), hv'_i])) \\
\qquad\quad (snd\ (snd\ (\epsilon_{vs}\ [hv'_1, \ldots, hv'_{i-1}, (x,\tau)])))) \\
\qquad\quad (base\_Hivs\ encx\ (getvar\_Hvst\ encx \\
\qquad\qquad (\epsilon_{vs}\ [hv'_1, \ldots, hv'_{i-1}, (x,\tau)])) \\
\qquad\quad (snd\ (snd\ (\epsilon_{vs}\ [hv'_1, \ldots, hv'_{i-1}, (x,\tau)])))) \\
\qquad\quad (\lambda P : bool \to Prop.\lambda pr : P\ true.pr))) \ldots))) \\
where\ X' = [hv'_1, \ldots, hv'_{i-1}, (x,\tau), hv'_i, \ldots, hv'_n] \\
\quad encx\ =\ mkpair\ Holvar\ (\epsilon_{vn}\ x)\ (\epsilon_\tau\ \tau) \\
\quad enchv'_n\ =\ mkpair\ Holvar\ (fst\ hv'_n)\ (snd\ hv'_n) \\
\quad enchv'_i\ =\ mkpair\ Holvar\ (fst\ hv'_i)\ (snd\ hv'_i)
\end{array}
$$

$$
\begin{array}{l}
\epsilon_{td}\ Abs((X,X') \blacktriangleright \lambda x:\tau.e:\tau \to \tau', [\delta]) = \\
\quad abs\_tr\ (\epsilon_{vs}\ (X,X'))\ (\epsilon_{vn}\ x, \epsilon_\tau\ \tau) \\
\qquad (\epsilon_{ht}\ (addbvar\_Hvst\ (\epsilon_{vn}\ x, \epsilon_\tau\ \tau)(\epsilon_{vs}\ (X,X')))\ e) \\
\qquad\quad (\epsilon_{td}\ \delta) \\
\qquad where \\
\qquad \delta \in \Delta_{\Pi_{TLC}}((X,X') \cup \{x:\tau\} \blacktriangleright e)
\end{array}
$$

$$
\begin{array}{l}
\epsilon_{td}\ Appl((X,X') \blacktriangleright e\ e' : \tau', [\delta, \delta']) = \\
\quad appl\_tr\ (\epsilon_{vs}\ (X,X'))\ (\epsilon_{ht}\ (\epsilon_{vs}\ (X,X'))\ e) \\
\qquad (\epsilon_{ht}\ (\epsilon_{vs}\ (X,X'))\ e')\ (\epsilon_\tau\ \tau)\ (\epsilon_\tau\ \tau') \\
\qquad (\epsilon_{td}\ \delta)\ (\epsilon_{td}\delta') \\
\qquad where \\
\qquad\ \delta' \in \Delta_{\Pi_{TLC}}((X,X') \blacktriangleright e:\tau), \\
\qquad \delta \in \Delta_{\Pi_{TLC}}((X,X') \blacktriangleright e:\tau \to \tau')
\end{array}
$$

4.5. Adequacy of the representation

Finally, we present the adequacy of the representation with the following theorem and its proof.

Theorem 4.18 *There exists a bijection between the closed derivations of a judgement* $((X, []) \blacktriangleright \phi : \tau)$ *and the normal forms of the proofs of the proposition*

$$Wfhterm\ (\epsilon_{vs}\ (X, []))\ (\epsilon_{ht}\ (\epsilon_{vs}\ X)\ \phi)\ (\epsilon_{\tau}\ \tau)$$

Proof:
This proof is not difficult because we have an exact correspondence between rules of the proof system and constructors of the inductive relation which encodes the proof system. First, we can easily prove that ϵ_{td} is injective and total To prove the bijection we define a decoding function with type

$$\begin{aligned}\epsilon_{td}^{-1} : &Wfhterm\ (\epsilon_{vs}\ (X, []))\ (\epsilon_{ht}\ (\epsilon_{vs}\ (X, []))\ e)\ (\epsilon_{\tau}\ \tau) \rightarrow \\ &\Delta_{\Pi_{TLC}}((X, [])\ \blacktriangleright\ e : \tau)\end{aligned}$$

inductively defined as follows:

$$\begin{aligned}&\epsilon_{td}^{-1}\ (ass1\_tr\ vs\ hv\ pr\ prin)\ = \\ &\quad ASS1((\epsilon_{vs}^{-1}\ vs) \blacktriangleright (\epsilon_{vn}^{-1}\ (fst\ hv) : (\epsilon_{\tau}^{-1}\ (snd\ hv)))\end{aligned}$$

$$\begin{aligned}&\epsilon_{td}^{-1}\ (ass2\_tr\ vs\ hv\ pr)\ = \\ &\quad ASS2((\epsilon_{vs}^{-1}\ vs) \blacktriangleright (\epsilon_{vn}^{-1}\ (fst\ hv) : (\epsilon_{\tau}^{-1}\ (snd\ hv)))\end{aligned}$$

$$\begin{aligned}&\epsilon_{td}^{-1}\ (abs\_tr\ vs\ hv\ ht\ hty\ dpr) = \\ &\quad \lambda ABS((\epsilon_{vs}^{-1}\ vs) \blacktriangleright \lambda(\epsilon_{vn}^{-1}\ (fst\ hv) : \epsilon_{\tau}^{-1}\ (snd\ hv)). \\ &\qquad (\epsilon_{ht}^{-1}\ (addbvar\_Hvst\ hv\ vs)\ ht) : \\ &\qquad ((snd\ hv) \rightarrow hty), [(\epsilon_{td}^{-1}(dpr))])\end{aligned}$$

$$\begin{aligned}&\epsilon_{td}^{-1}(appl\_tr\ vs\ ht\ ht'\ hty\ hty'\ wftpr\ wftpr') = \\ &\quad APPL\ (\epsilon_{vs}^{-1}\ vs)\ \blacktriangleright\ (\epsilon_{ht}^{-1}\ vs\ ht)\ \ (\epsilon_{ht}^{-1}\ vs\ ht') : hty', \\ &\qquad [(\epsilon_{td}^{-1}(wftpr)), (\epsilon_{td}^{-1}\ wftpr')]\end{aligned}$$

This decoding function is also injective and total and it holds by an easy induction that for all closed derivations $deriv \in \Delta_{\Pi_{TLC}}$ $\quad \epsilon_{td}^{-1}\ (\epsilon_{td}\ deriv) = deriv$ which is necessary to guarantee the bijection.

5. Encoding of a fragment of a linear type theory

In this section we give an adequate encoding of the functional fragment of SLR, a lambda calculus with modal and linear function spaces designed by (Hoffman, 1999). The main differences with respect to typed-lambda calculus is that contexts contains variables with aspects where an aspect is a pair containing the information whether the variable is linear or nonlinear and whether the variable is modal or nonmodal. As we mentioned in the introduction, it is possible to represent this type theory in our framework since our principle of encoding is not the same as in LF and we are able to represent and manipulate non-standard contexts like a linear one. Another difference with respect to lambda calculus is that there exists different functional spaces like for example a (linear, nonmodal) functional space and a (nonlinear, nonmodal) functional space. The formal semantics can be found in (Hoffman, 1999) and we do not detail it here because it is not necessary for our purposes. An interesting application of this type theory is to develop functional programs with polynomial time complexity.

Finally, we do not split contexts in contexts with free and bound variables because we do not represent the substitution operation.

The fragment of SLR which we are going to encode adequately in UTT is formally defined by the following definitions:

Definition 5.1 *An aspect is a pair* (l, m) *where* $l \in \{linear, nonlinear\}$ *and* $m \in \{nonmodal, modal\}$. *The aspects are ordered componentwise by* $nonlinear <: linear$ *and* $modal <: nonmodal$.

Definition 5.2 *The type expressions which we will consider are the following:*

$$
\begin{array}{lll}
\mathcal{T}_{SLR} ::= & N & \textit{natural numbers} \\
 & L(\mathcal{T}_{SLR}) & \textit{lists over } \mathcal{T}_{SLR} \\
 & T(\mathcal{T}_{SLR}) & \textit{binary trees labelled over } \mathcal{T}_{SLR} \\
 & \mathcal{T}_{SLR} \stackrel{a}{\rightarrow} \mathcal{T}_{SLR} & \textit{function space of aspect } a.
\end{array}
$$

$\mathcal{T}_{SLR} \stackrel{a}{\rightarrow} \mathcal{T}_{SLR}$ *is the generic notation used to define the type system but normally the different function spaces are denoted in this way:*

$\mathcal{T}_{SLR} \stackrel{a}{\rightarrow} \mathcal{T}_{SLR}$ *is* $\mathcal{T}_{SLR} \multimap \mathcal{T}_{SLR}$ *when* $a = \{linear, nonmodal\}$
$\mathcal{T}_{SLR} \stackrel{a}{\rightarrow} \mathcal{T}_{SLR}$ *is* $\mathcal{T}_{SLR} \rightarrow \mathcal{T}_{SLR}$ *when* $a = \{nonlinear, nonmodal\}$
$\mathcal{T}_{SLR} \stackrel{a}{\rightarrow} \mathcal{T}_{SLR}$ *is* $\Box\mathcal{T}_{SLR} \rightarrow \mathcal{T}_{SLR}$ *when* $a = \{nonlinear, modal\}$

Definition 5.3 *The expressions which we will consider are the following:*

$$\begin{array}{rcll} \Lambda_{SLR} & ::= & x & (variable) \\ & & \Lambda_{SLR}\,\Lambda_{SLR} & (application) \\ & & \lambda x : \mathcal{T}_{SLR}.\Lambda_{SLR} & (abstraction) \end{array}$$

Definition 5.4 *A context is a partial function from term variables to pairs of aspects and types typically written as a list of bindings of the form* $x \overset{a}{:} A$.

For any context Γ, $Dom(\Gamma)$ *denotes the set of variables bound in* Γ. *If* $x \overset{a}{:} A \in \Gamma$ *then* $\Gamma\ (x)$ *denotes* A *and* $\Gamma\ ((x))$ *denotes* a *and* Γ, Δ *denotes the union of the contexts* Γ *and* Δ *if* $Dom(\Gamma)$ *and* $Dom(\Delta)$ *are disjoint.*

The following judgements are used to define the type system:

- Γ *nonlinear* which means that all its bindings are of nonlinear aspect.
- $Disjoint\ \Gamma\ \Delta$ which means that the sets $Dom(\Gamma)$ and $Dom(\Delta)$ are disjoint.
- $\Gamma \vdash e : A$ which means that the expression e has type A in the context Γ.
- $\Gamma <: a$ which means that for all bindings $x \overset{a'}{:} A$ in Γ, $a' <: a$.

Definition 5.5 *The judgement* $\Gamma <: a$ *for any context* Γ *and any aspect* a *is inductively defined by the following rules:*

$$\frac{}{< > <: a} \quad (bc <:)$$

$$\frac{\Gamma <: a \qquad a' <: a}{\Gamma, \{x \overset{a'}{:} A\} <: a}\ x \notin Dom(\Gamma) \quad (gc <:)$$

Definition 5.6 *The judgement* $Disjoint\ \Gamma\ \Delta$ *for any context* Γ, Δ *is inductively defined by the following rules:*

$$\frac{}{Disjoint\ \ < >\ \ \Delta} \quad (bcdisj)$$

$$\frac{Disjoint\ \Gamma\ \Delta}{Disjoint\ \Gamma, \{x \overset{a}{:} A\}\ \Delta}\ x \notin Dom(\Delta) \wedge x \notin Dom(\Gamma) \quad (gcdisj)$$

Definition 5.7 *The judgement* $\Gamma \;\; nonlinear$ *for any context* Γ *is inductively defined by the following rules:*

$$\frac{}{<> \;\; nonlinear} \qquad (bcnl)$$

$$\frac{\Gamma \;\; nonlinear}{\Gamma, \{x \overset{a}{:} A\} \;\; nonlinear} \; x \notin Dom(\Gamma) \wedge fst(a) = nonlinear \qquad (gcnl)$$

Definition 5.8 *The functional fragment of the type system SLR is inductively defined by the following rules:*

$$\frac{\Gamma, \{x \overset{a}{:} A\} \vdash e : B}{\Gamma \vdash \lambda x : A.e : A \overset{a}{\to} B} \qquad (Tarri)$$

$$\frac{\begin{array}{c} Disjoint\, \Gamma\, \Delta_1 \quad Disjoint\, \Gamma\, \Delta_2 \quad Disjoint\, \Delta_1\, \Delta_2 \\ \Gamma, \Delta_1 \vdash e_1 : A \overset{a}{\to} B \quad \Gamma, \Delta_2 \vdash e_2 : B \quad \Gamma \; nonlinear \quad \Gamma, \Delta_2 <: a \end{array}}{\Gamma, \Delta_1, \Delta_2 \vdash e_1\, e_2 : B} \qquad (Tarre)$$

And, now we proceed with the encoding of the type theory in a similar way as the encoding of typed lambda calculus.

5.1. Encoding of variables, contexts and terms

First, we represent aspects and their relation operation.

Definition 5.9 *The inductive type* $Linearity$ *is defined by the following constructors:*

$$linear : Linearity$$
$$nonlinear : Linearity$$

Definition 5.10 *The inductive relation* $<: \_Lin : Linearity \to Linearity \to Prop$ *is inductively defined by the following constructor:*

$$nll : <: \_Lin\; nonlinear\; linear$$

Definition 5.11 *The inductive type* $Modality$ *is defined by the following constructors:*

$$modal : Modality$$
$$nonmodal : Modality$$

Definition 5.12 *The inductive relation* $<: \_Mod : Modality \to Modality \to Prop$ *is inductively defined by the following constructor:*

$$mnm : <: \_Mod\; modal\; nonmodal$$

Definition 5.13 *The type $SLRaspect$ is defined as $Pair\ Linearity\ Modality$.*

Definition 5.14 *The inductive relation $<: \_Asp : SLRaspect \rightarrow SLRaspect \rightarrow Prop$ is defined by the following set of constructors:*

$$refl : \Pi l : Linearity.\Pi m : Modality.\ <: \_Asp\ (mkpair\ SLRaspect\ l\ m)\ (mkpair\ SLRaspect\ l\ m)$$

$$compw :: \Pi l, l' : Linearity.\Pi m, m' : Modality.\ \Pi linr :<: \_lin\ l\ l'.\Pi modr :<: \_mod\ m\ m'.\ <: \_Asp\ (mkpair\ SLRaspect\ l\ m)\ (mkpair\ SLRaspect\ l'\ m')$$

We define an additional inductive relation on aspects to check whether an aspect is nonlinear.

Definition 5.15 *The inductive relation $Nonlin\_Asp : SLRaspect\ \rightarrow Prop$ is defined by the following constructor:*

$$nonlc\_Nlm : \Pi mod : Modality.Nonlin\_Asp\ (mkpair\ SLRaspect\ nonlinear\ mod)$$

Next, we define the types of the type theory.

Definition 5.16 *The inductive type $SLRtype$ is defined by the following set of constructors:*

$$\begin{aligned} &nat\ :\ SLRtype\\ &list\ :\ SLRtype\ \rightarrow\ SLRtype\\ &tree\ :\ SLRtype\ \rightarrow\ SLRtype\\ &lmfunc\ :\ SLRtype\ \rightarrow\ SLRaspect\ \rightarrow\ SLRtype \end{aligned}$$

And next, we define variables together with an operation to get the aspect of the variable, variables with indexes and contexts.

Definition 5.17 *The type $SLRvar$ is defined as*
$Pair\ (Pair\ Varname\ SLRtype)\ SLRaspect$

Definition 5.18 *The function $getaspect\_SLRv : SLRvar\ \rightarrow\ SLRaspect$ is defined as follows:*

$$getaspect\_SLRv\ svar\ =\ (snd\ svar)$$

Definition 5.19 *The type $SLRivar$ is defined as $Pair\ SLRvar\ Varindex$.*

Definition 5.20 *The type $SLRcontext$ is defined as*
$Pair\ Varindex\ (List\ SLRivar)$

We assume predefined the following functions and inductive relations of $SLRvar$,$SLRivar$ and $SLRcontext$ which are defined in a very similar way as the equivalent operations in the typed lambda calculus:

$Eqbool\_SLRv : SLRvar \to SLRvar \to Bool$
$Eqbool\_SLRiv : SLRivar \to SLRivar \to Bool$
$empty\_SLRctxt : SLRcontext$
$addvar\_SLRctxt : SLRvar \to SLRcontext \to SLRcontext$
$getvar\_SLRctxt : Varname \to SLRcontext \to SLRivar$
$concat\_SLRctxt : SLRcontext \to SLRcontext \to SLRcontext$
$Is\_in\_SLRctxt : Varname \to SLRcontext \to Prop$
$Not\_is\_in\_SLRctxt : Varname \to SLRcontext \to Prop$

We have additionally the following inductive relations on contexts which are needed to represent the different judgements on contexts (Γ $<:$ a,$Disjoint\,\Gamma\,\Delta$ and Γ $nonlinear$) used in the definition of this type theory.

Definition 5.21 *The inductive relation* $<:\_Ctxt : SLRcontext \to SLRaspect \to Prop$ *is defined by the following constructors:*

$bc <:: \Pi a : SLRaspect.\ <:\_Ctxt\ empty\_SLRctxt\ a$
$gc <:: \Pi slrc : SLRcontext.\Pi slrv : SLRvar.\Pi a : SLRaspect.$
$\Pi apr :<: \_Asp\ (getaspect\_SLRv\ slrv)\ a.\Pi slrcpr :<: \_Ctxt\ slrc\ a.$
$\Pi isinpr :\ Not\_is\_in\_SLRctxt\ (fst\ (fst\ slrv))\ (snd\ slrc).$
$<:\_Ctxt\ (addvar\_SLRctxt\ slrv\ slrc)\ a$

Definition 5.22 *The inductive relation* $Nonlinear\_Ctxt : SLRcontext \to Prop$ *is defined by the following constructors:*

$bcnl : Nonlinear\_Ctxt\ empty\_SLRctxt$
$gcnl : \Pi slrc : SLRcontext.\Pi slrv : SLRvar.\Pi nlpr : Nonlinear\_Ctxt\ slrc.$
$\Pi ninpr :\ Not\_is\_in\_SLRctxt\ (fst\ (fst\ slrv))\ (snd\ slrc).$
$\Pi nlapr : Nonlin\_Asp\ (getaspect\_SLRv\ slrv).$
$Nonlinear\_Ctxt\ (addvar\_SLRctxt\ slrv\ slrc)$

Definition 5.23 *The inductive relation* $Disjoint\_Ctxt : SLRcontext \to SLRcontext \to Prop$ *is defined by the following constructors:*

$bcdisj : \Pi slrc : SLRcontext.Disjoint\_Ctxt\ empty\_SLRctxt\ slrc$
$gcdisj : \Pi slrc, slrc' : SLRcontext.\Pi slrv : SLRvar.$
$\Pi ninpr :\ Not\_is\_in\_SLRctxt\ (fst\ (fst\ slrv))\ slrc.$
$\Pi ninpr :\ Not\_is\_in\_SLRctxt\ (fst\ (fst\ slrv))\ slrc'.$
$\Pi disjpr : Disjoint\_Ctxt\ slrc\ slrc'.$
$Disjoint\_Ctxt\ (addvar\_SLRctxt\ slrv\ slrc)\ slrc'$

Finally, we have the encoding of terms.

Definition 5.24 *The inductive type $SLRterm$ is defined by the following constructors:*

$$
\begin{array}{l}
var\_SLRt : SLRivar \rightarrow SLRterm \\
appl\_SLRt : SLRterm \rightarrow SLRterm \rightarrow SLRterm \\
abs\_SLRt : SLRivar \rightarrow SLRterm \rightarrow SLRterm
\end{array}
$$

5.2. Encoding of the type theory

We will also assume predefined the following encoding and decoding functions as in the typed lambda calculus:

$$
\begin{array}{l}
\epsilon_a : Aspect \rightarrow SLRaspect \\
\epsilon_a^{-1} : SLRaspect \rightarrow Aspect \\
\epsilon_{\tau SLR} : \mathcal{T}_{SLR} \rightarrow SLRtype \\
\epsilon_{\tau SLR}^{-1} : SLRtype \rightarrow \mathcal{T}_{SLR} \\
\epsilon_{ctxtSLR} : [Binding] \rightarrow SLRcontext \\
\epsilon_{ctxtSLR}^{-1} : SLRcontext \rightarrow [Binding] \\
\epsilon_{tSLR} : SLRcontext \rightarrow \Lambda_{SLR} \rightarrow SLRterm \\
\epsilon_{tSLR}^{-1} : SLRcontext \rightarrow SLRterm \rightarrow \Lambda_{SLR} \\
\epsilon_{Djctxt} : (Disjoint\ \Delta_1\ \Delta_2) \rightarrow \\
\quad (Disjoint\_Ctxt\ (\epsilon_{ctxtSLR}\ \Delta_1)\ (\epsilon_{ctxtSLR}\ \Delta_2)) \\
\quad \text{for any } \Delta_1, \Delta_2 \in [Binding] \\
\epsilon_{Djctxt}^{-1} : (Disjoint\_Ctxt\ sclr\ sclr') \rightarrow \\
\quad (Disjoint\ (\epsilon_{ctxtSLR}^{-1}\ sclr)\ (\epsilon_{ctxtSLR}^{-1}\ sclr'))
\end{array}
$$

for any $sclr, sclr' : SLRcontext$

$\epsilon_{nlctxt} : \Delta_{\Pi_{nlctxt}}(Nonlinear\ \Gamma) \rightarrow (Nonlinear\_Ctxt\ (\epsilon_{ctxtSLR}\ \Gamma))$
for any $\Gamma \in [Binding]$

$\epsilon_{nlctxt}^{-1} : (Nonlinear\_Ctxt\ slrc) \rightarrow \Delta_{\Pi_{nlctxt}}(Nonlinear\ (\epsilon_{nlctxt}^{-1}\ slrc))$
for any $slrc : SLRcontext$

$\epsilon_{<:ctxt} : \Delta_{\Pi_{<:ctxt}}(\Gamma <: a) \rightarrow (<:\_Ctxt\ (\epsilon_{ctxtSLR}\ \Gamma)\ (\epsilon_a\ a))$
for any $a \in Aspect$,$\Gamma \in [Binding]$

$\epsilon_{<:ctxt}^{-1} : (<:\_Ctxt\ slrc\ a) \rightarrow \Delta_{\Pi_{<:ctxt}}(\epsilon_{ctxtSLR}^{-1}\ slrc) <: (\epsilon_a^{-1}\ a)$
for any $a : SLRaspect$,$slrc : SLRcontext$

where $Binding$ are triples of type $(X, Aspect, T_{SLR})$. The representation of the type system is by the following inductive relation:

Definition 5.25 *The inductive relation*

$$SLRts : SLRctxt \rightarrow SLRterm \rightarrow SLRtype \rightarrow Prop$$

is defined by the following constructors:

$$Tarri : \Pi slrc : SLRcontext.\Pi slrv : SLRvar.\Pi t : SLRterm.$$
$$\quad \Pi pr : SLRts\ (addvar\_SLRctxt\ slrv\ slrc)\ t\ (snd\ (fst\ slrv)).$$
$$\quad\quad SLRts\ slrc\ (abs\_SLRt\ (getvar\_SLRctxt\ (fst\ (fst\ slrv))$$
$$\quad\quad (addvar\_SLRctxt\ slrv\ slrc))\ t)$$

$$Tarre : \Pi slrc, slrc', slrc'' : SLRcontext.\Pi a : SLRaspect.$$
$$\quad \Pi t, t' : SLRterm.\Pi A, B : SLRtype.$$
$$\quad \Pi dpr : Disjoint\_Ctxt\ slrc\ slrc'.\Pi dpr' : Disjoint\_Ctxt\ slrc\ slrc''.$$
$$\quad \Pi dpr'' : Disjoint\_Ctxt\ slrc'\ slrc''.$$
$$\quad \Pi td : SLRts\ (concat\ slrc'\ slrc)\ t\ (lmfunc\ A\ a\ B).$$
$$\quad \Pi td' : SLRts\ (concat\ slrc''\ slrc)\ t'\ A.$$
$$\quad \Pi nlpr : Nonlinear\_Ctxt\ slrc.\Pi rprc :<: \_Ctxt\ (concat\ slrc\ slrc')\ a.$$
$$\quad\quad SLRts\ (concat\ slrc''\ (concat\ slrc''\ slrc))\ (appl\_SLRt\ t\ t')\ B$$

The encoding function of the type theory is as follows:

Definition 5.26 *The encoding function of derivations of SLR* ϵ_{slrtd} *which given a closed derivation in* $\Delta_{\Pi_{SLR}}(\Gamma \vdash e : A)$ *returns a proof of the proposition*

$$SLRts\ (\epsilon_{ctxtSLR}\ \Gamma)\ (\epsilon_{tSLR}\ (\epsilon_{ctxtSLR}\ \Gamma)\ e)(\epsilon_{\tau SLR}\ A)$$

is inductively defined as follows:

$$\epsilon_{slrtd}\ (Tarri(\Gamma \vdash \lambda x : A.e : A \xrightarrow{a} B), [\delta]) = Tarri\ (\epsilon_{ctxtSLR}\ \Gamma)\ encx$$
$$\quad (\epsilon_{tSLR}\ (addvar\_SLRctxt\ encx\ (\epsilon_{ctxtSLR}\ \Gamma))\ e)\ (\epsilon_{slrtd}\ \delta)$$

where
$\delta \in \Delta_{SLR}(\Gamma, \{x \overset{a}{:} A\} \vdash e : B)$

$$encx = mkpair\ SLRvar\ (mkpair\ Varname\ SLRtype\ (\epsilon_{vn}\ x)\ (\epsilon_{\tau SLR}\ \tau))\ (\epsilon_a\ a)$$

$$\begin{array}{l}\epsilon_{slrtd}\ Tarre(\Gamma,\Delta_1,\Delta_2 \vdash e_1\ e_2 : B,[\delta_1,\delta_2,\delta_3,\delta_4,\delta_5,\delta_6,\delta_7]) = \\ \quad tarre\ (\epsilon_{ctxtSLR}\,\Gamma)\ (\epsilon_{ctxtSLR}\,\Delta_1)\ (\epsilon_{ctxtSLR}\,\Delta_2)\ (\epsilon_a\ a) \\ \qquad (\epsilon_{tSLR}\ (\epsilon_{ctxtSLR}\,\Gamma,\Delta_1)\ e_1) \\ \qquad (\epsilon_{tSLR}\ (\epsilon_{ctxtSLR}\,\Gamma,\Delta_1)\ e_2) \\ \qquad (\epsilon_{\tau SLR}\,A)\ (\epsilon_{\tau SLR}\,B) \\ \qquad (\epsilon_{Djctxt}\ \delta_1)\ (\epsilon_{Djctxt}\ \delta_2)\ (\epsilon_{Djctxt}\ \delta_3) \\ \qquad (\epsilon_{SLR}\ \delta_4)\ (\epsilon_{SLR}\ \delta_5)\ (\epsilon_{nlctxt}\ \delta_6)\ (\epsilon_{<:ctxt}\ \delta_7)\end{array}$$

where
$\delta_1 \in \Delta_{\Pi_{Djctxt}}(Disjoint\,\Gamma\ \Delta_1), \delta_2 \in \Delta_{\Pi_{Djctxt}}(Disjoint\,\Gamma\ \Delta_2)$,
$\delta_3 \in \Delta_{\Pi_{Djctxt}}(Disjoint\ \Delta_1\ \Delta_2), \delta_4 \in \Delta_{\Pi_{SLR}}\ (\Gamma,\Delta_1 \vdash e_1 : A \stackrel{a}{\rightarrow} B)$,
$\delta_5 \in \Delta_{\Pi_{SLR}}\ (\Gamma,\Delta_2 \vdash e_2 : A)$,
$\delta_6 \in \Delta_{\Pi_{nlctxt}}(Nonlinear\ \Gamma), \delta_7 \in \Delta_{\Pi_{<:ctxt}}(\Gamma,\Delta_2\ <:\ a)$

5.3. Adequacy of the representation

The adequacy of the representation is stated by the following theorem and its proof:

Theorem 5.27 *For any context Γ,for any term $t \in \Lambda_{SLR}$, for any type $\tau \in \mathcal{T}_{SLR}$, there exists a bijection between the closed derivations of the judgement $(\Gamma \vdash t : \tau)$ and the normal forms of the proofs of the proposition*

$$SLRts\ (\epsilon_{ctxtSLR}\,\Gamma)\ (\epsilon_{tSLR}\ (\epsilon_{ctxtSLR}\,\Gamma)\ t)\ (\epsilon_{\tau SLR}\,\tau)$$

Proof 5.28 *To prove the bijection we define a decoding function with type*

$$\begin{array}{l}\epsilon^{-1}_{slrtd} : (SLRts\ slrc\ t\ \tau) \rightarrow \\ \qquad (\Delta_{\Pi_{SLR}}\ (\epsilon^{-1}_{ctxtSLR}\ slrc)\ (\epsilon^{-1}_{tSLR}\ (\epsilon^{-1}_{ctxtSLR}\ slrc)\ t)\ (\epsilon^{-1}_{\tau SLR}\,\tau))\end{array}$$

for any $slrc : SLRcontext$, $t : SLRterm$,$\tau : SLRtype$ *inductively defined as follows:*

$$\epsilon^{-1}_{slrtd}\ (Tarri\ slrc\ slrv\ t\ pr) =$$
$$Tarri((\epsilon^{-1}_{ctxtSLR}\ slrc) \vdash \lambda\ (\epsilon^{-1}_{vn}\ (fst\ (fst\ slrv)) : (\epsilon^{-1}_{\tau SLR}\ (snd\ (fst\ slrv))).$$
$$(\epsilon^{-1}_{tSLR}\ (addvar\_SLRctx\ slrv\ slrc)\ t), [\epsilon^{-1}_{slrtd}\ pr])$$

$$\epsilon^{-1}_{slrtd}\ (Tarre\ slrc\ slrc'\ slrc''\ a\ t\ t'\ A\ B\ dpr\ dpr'\ dpr''\ td\ td'\ nlpr\ rpr) =$$
$$Tarre((concat\ (\epsilon^{-1}_{ctxtSLR}\ slrc'')\ (concat\ (\epsilon^{-1}_{ctxtSLR}\ slrc')\ (\epsilon^{-1}_{ctxtSLR}\ slrc)))$$
$$(\epsilon^{-1}_{tSLR}\ (concat\ (\epsilon^{-1}_{ctxtSLR}\ slrc')\ (\epsilon^{-1}_{ctxtSLR}\ slrc))\ t)$$
$$(\epsilon^{-1}_{tSLR}\ (concat\ (\epsilon^{-1}_{ctxtSLR}\ slrc'')\ (\epsilon^{-1}_{ctxtSLR}\ slrc))\ t') :$$
$$(\epsilon^{-1}_{tSLR}\ B),$$
$$[\epsilon^{-1}_{Djctxt}\ dpr, \epsilon^{-1}_{Djctxt}\ dpr', \epsilon^{-1}_{Djctxt}\ dpr, \epsilon^{-1}_{SLR}\ td,$$
$$\epsilon^{-1}_{SLR}\ td', \epsilon_{nlctxt}\ nlpr, \epsilon_{<:ctxt}\ rpr])$$

and the rest of the proof follows in the same way as in the typed lambda calculus.

6. Conclusions

In this paper, we have presented how to use the type theory UTT as a logical framework. We have presented the main advantages with respect to LF and how to encode two different kind of proof systems:an adequate encoding of the typed lambda calculus with non-trivial substitution operation on higher-order terms requiring updating of deBruijn indexes and another adequate encoding of a linear type theory which is not possible to encode in LF because it is not possible to identify the variables of LF with the variables with aspect of the linear type theory.

These techniques have been applied to redesign and implement different proof systems for the deduction of properties from algebraic specifications in first-order and higher-order logic and to implement proof systems for refinement of algebraic specifications in (Mylonakis, 2000)

References

Bengt Nordström Kent Petersson and Jan Smith. Programming in Martin-Löf's Type Theory:An Introduction.Oxford University Press,1990

Philippa Gardner. Representing Logics in Type Theory. PhD thesis, University of Edinburgh, July 1992

Healfdene Goguen. A Typed Operational Semantics for Type Theory. PhD thesis, University of Edinburgh, September 1994.

Robert Harper, Furio Honsell and Gordon Plotkin. A framework for defining logics. Journal of the Association for Computing Mchinery,40(1):143-184, January1993.

Rolf Hennicker. Structured Specifications with Behavioural Operators: Semantics, Proof Methods and Applications. Habilitationsschrift. Intitut für Informatik, Ludwig-Maximilians-Universität München, June 1997.

Rolf Hennicker, Martin Wirsing and Michel Bidoit. Proof systems for structured specifications with observability operators. Theoretical Computer Science, 173, February 1997.

Martin Hoffman. Type systems for polynomial-time computation. Habilitation thesis, University of Darmstadt, 1999

LEGO Literature. http://www.dcs.ed.ac.uk/home/lego/html/papers.html

Zhaohui Luo. Computation and Reasoning: A Type Theory for Computer Science, Clarendon Press Oxford, 1994.

N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, Computer Science Laboratory. To appear in D.Gabbay,ed., Handbook of Philosophical Logic,Kluwer Academic Publishers.

Nikos Mylonakis. A type-theoretic approach to proof support for algebraic design frameworks. PhD thesis, Universitat Politècnica de Catalunya. To appear

L.C.Paulson. Isabelle, volume 828 of LNCS. Springer Verlag, 1994

Robert Pollack. The Theory of LEGO. A proof checker for the Extended Calculus of Constructions. University of Edinburgh, April 1995.