Chapter3:UTT as a design framework

1 Introduction

As we mentioned in the previous chapter, UTT was originally designed as an object language for the specification, development and verification of functional programs. As we explained also, since UTT is an expressive type theory, it can also be used as a formalism to represent other formalisms and therefore to represent design frameworks.

Some examples of the different uses of this type theory as a framework for software design are deliverables [BM91], [McK92], which is a methodology for the development of functional programs from specifications using a precondition-postcondition style like the specifications of imperative programs in Hoare logics, the representation of Z specifications by [Mah95], a framework for the development of operational semantics by [Kha97] and the representation of VDM and Hoare logics for verification of imperative programs by [Kle98]. In the rest of the chapter, we will first present the modular design framework for functional programs developed by Luo since it is the most related to our work, then we will present the verification of a functional program with respect to a basic specification and the refinement of an algebraic specification in Luo's framework as examples of two of the main tasks of design frameworks for functional programs.

Since in Luo's work there exists no sound and/or complete result between the semantics of this design framework and the semantics of a design framework for algebraic specifications, we will finally explain how to represent natural deduction systems in *UTT* following the basic ideas of *LF* but using the more expressive type theory *UTT*. None of the previous works have developed this representation technique which will be the basis to represent adequately the complete proof systems for deduction and refinement of algebraic specifications in ASL presented in [BCH] and in [Hen97] using first-order and higher-order as specification logics.

The main objective of this work is to give a way to reuse current and future generations of theorem provers of the type theory *UTT* or a similar one following the same philosophy as LF, to develop theorem provers for design frameworks for algebraic specifications like the one presented in this thesis or others a little bit more complex designed for industrial use.

2 UTT as an algebraic design framework

In this section, we represent as in [Luo94], the basic components of this algebraic design framework such as specifications, refinement of specifications and implementation of specifications. We concentrate on specifications of abstract data types with a first- order signature and we explain how the implementations of a specification of these abstract data types can be seen as functional programs.

Definition 2.1 *Specifications are defined as elements of the following type:*

$$SPEC \;=\; \sum[Str : Type, Ax : Str \;\rightarrow\; Prop]$$

Notation: *The first component of a specification $SP : SPEC$ will be denoted by $Str[SP]$ and its inhabitants will be referred as SP-structures and the second component of a specification will be denoted by $Ax[SP]$. For any type $S : Type$, $Spec(S)$ will denote the "subclass" of specifications such that $Str[SP] \equiv S$ and the axioms are of the type $Ax[SP] : S \to Prop$. A possible type of this "subclass" of specifications is just $S \to Prop$ but as in [Luo94], we will consider them as specifications and therefore with structures as inhabitants.*

Since we are specially interested in specification of abstract data types using the algebraic approach, in our examples the first component of the Σ-type will be normally a first-order signature also denoted as Σ, its sorts denoted as $Sorts(\Sigma)$ and its operations denoted as $Op(\Sigma)$. The sorts of the signature of a specification will be represented by a type and a binary relation on this type as in [Luo94] similar to the ultraloose approach of algebraic specifications. The binary relation of a sort will be normally required to be a congruence and therefore this will be normally axiomatised in the second component of the specification and operations will be represented as functions.

Definition 2.2 *The representation in UTT of a first-order signature Σ of the following form:*

$$\Sigma = (s_1, \ldots, s_n, op_1 : s_{11} \times \ldots \times s_{1m_1} \to s_{m_1}, \ldots, op_n : s_{p1} \times \ldots \times s_{pm_p} \to s_{m_p})$$

is the following Σ type:

$$SIG = \sum[s_1 : Setoid, \ldots, s_n : Setoid,$$

$$op_1 : Dom(s_{11}) \times \ldots \times Dom(s_{1m_1}) \to Dom(s_{m_1}), \ldots,$$

$$op_n : Dom(s_{p1}) \times \ldots \times Dom(s_{pm_p}) \to Dom(s_{m_p})]$$

where a Setoid is again a Σ type of the form

$$Setoid = \Sigma[Dom : Type, Eq : Dom \to Dom \to Prop]$$

Note that the structures of these specifications are closely related to the notion of Σ-algebra but they are not the same because the "carrier sets" of these structures are types of the type theory and not arbitrary sets and the functions of these structures are not set-theoretical functions but functions which can be represented in the type theory using normally primitive recursion. See the examples of this chapter for the specifications of some abstract data types in the type theory following this representation and the axiomatisation that the binary relations associated to the sorts are congruences.

Definition 2.3 *Let SP be a specification of type $SPEC$. A realisation or an implementation of SP is an SP-structure $str : Str[SP]$ such that the proposition $Ax[SP]\ str$ is derivable in the type theory.*

Remark and notation: *When the first component of the specification SP is the representation of a first-order signature Σ as specified above, then a SP-structures $str : Str[SP]$ can be seen as a functional program or module defined by a set of types (which are represented as setoids) and a set of functions. The first component of the setoids of these structures will be normally denoted by $Dom[s]_{str}$ or just by s_{str} where $s \in Sorts(\Sigma)$ and the functions will be normally denoted by op_{str} where $op \in Op(\Sigma)$.*

Definition 2.4 *Let SP be a specification. The satisfaction relation $str \models \phi$ between a SP-structure $str : Str[SP]$ and the sentence over SP-structures $\phi : Str[SP] \rightarrow Prop$ holds if the proposition $(\phi\ str)$ is derivable in UTT.*

Definition 2.5 *Let SP and SP' be specifications. A refinement map from SP' to SP is a function $\rho : Str[SP'] \rightarrow Str[SP]$ such that the following condition, called satisfaction condition, is provable:*

$$Sat(\rho) \ = \ \forall s' : Str[SP'].Ax[SP'](s') \ \supset \ Ax[SP](\rho(s'))$$

Notation: *A refinement relation between two specifications SP, SP' by a refinement map $\rho : Str[SP'] \rightarrow Str[SP]$ will be denoted by $SP \Longrightarrow_{\rho} SP'$ and we will say that SP refines to SP'.*

Proposition 2.6 Vertical composition: *If $SP \Longrightarrow_{\rho} SP'$ and $SP' \Longrightarrow_{\rho'} SP''$ then $SP \Longrightarrow_{\rho\ o\ \rho'} SP''$.*

Notation: *The composition function $f\ o\ g$ is defined for any $f : A \rightarrow B$, $g : C \rightarrow A$, $x : B$ as $f\ o\ g\ x \ = \ f\ (g\ x)$.*

3 An observational equality

One of the congruences which has been most frequently used in algebraic specifications is an observational equality which was originally defined for first-order signatures with a distinguished set of observable sorts, which in the following will be referred to as Obs. Intuitively, two elements of the carrier set of a sort s are related if they are indistinguishable by a set of observations which are represented by contexts, and contexts are represented by terms of observable sort with a distinguished context variable of sort s. See below for a type-theoretic representation of contexts and the chapter of ASL for the most general definition of contexts in an algebraic setting which includes also a distinguished set of input sorts. These sorts can denote unreachable input in the context since only free variables of input sort can appear in the context. For simplicity, in this setting we will assume that the set of input sorts is empty since in our examples and normally in this framework, we will be interested in reachable SP-structures which implement a specification SP. In these cases, the repesentation of contexts with and without input sorts is equivalent since they denote the same set of observations.

In this section, we present a type-theoretic definition of the observational equality proving that is a congruence and then we present a context induction principle which is useful to reason about observational equality and some simplification techniques. Finally, we present an alternative definition given in [HS96] based on the property of this relation which is the greatest congruence which coincides with the Leibniz equality for the observable sorts. Both representantions of the observational equality and the different proof techniques which are presented in this section were not in the original work of Luo.

Definition 3.1 *Assuming a structure str which inhabits the encoding of a signature $\Sigma = (S, Op)$, the inductive relation which defines contexts from sort r to s of the signature Σ restricted to the operation symbols $Op_{r,s} \subseteq Op$ is written $C^{r\to s}_{(str,Op_{r,s})}$ and has type $(r_{str} \to s_{str}) \to Prop$. It is generated by the following set of constructors:*

$$Ctr(C^{r\to s}_{(str,Op_{r,s})}) = \begin{cases} KC(C^{r\to s}_{(str,Op_{r,s})}) & , r \neq s \\ KC(C^{r\to s}_{(str,Op_{r,s})}) \cup & \\ \quad \{ctriv\_r\_s : C^{r\to s}_{(str,Op_{r,s})}\ \lambda x : r_{str}.x\} & , r = s \end{cases}$$

where

$$KC(C^{r\to s}_{(str,Op_{r,s})}) =$$

$$\left\{ \begin{array}{l|l} cc\_r\_s\_op_i : \Pi v_1 : s_{1\,str}. & \\ \vdots & \\ \Pi c_i : r_{str} \to s_{i_{str}}. & op : s_1 \times \ldots \times s_n \to s \in Op_{r,s} \\ \Pi r_i :\ C^{r\to s_i}_{(str,Op)}\ c_i. & i \in [1..n] \\ \vdots & \\ \Pi v_n : s_{n\,str}. & \\ C^{r\to s}_{(str,Op_{r,s})}\ \lambda x : r_{str}.op_{str}(v_1, \ldots, c_i\ x, \ldots, v_n) & \end{array} \right\}$$

Remarks:

We consider contexts with just one occurrence of the context variable. We are able to restrict the operation symbols appearing in the outermost position of the context depending on the sort of the context variable and the result sort of the context we are defining. As we will see, these sets of contexts will be valid to define observational equalities.

To make the presentation and the encoding in type theory easier, we parameterise contexts by a structure (to make the interpretation of contexts more implicit) and abstract by the context variable (to make context application to a value just functional application). Therefore, contexts can be seen as functions from the type associated to the sort of the context variable to the type associated to the result sort of the context. Since not all functions of this type are contexts, it is necessary to define contexts via an inductive relation which determines the functions which are contexts. Since $C^{r\to s}_{(str,Op_{r,s})}$ has type $(r_{str} \to s_{str}) \to Prop$, the application $C^{r\to s}_{(str,Op_{r,s})}\ c$ where $c : r_{str} \to s_{str}$ is a valid UTT_term, and intuitively denotes the proposition one has to prove to show that the function c is a context.

Notation: *Given a function c of type $r_{str} \rightarrow s_{str}$ where $C^{r\rightarrow s}_{(str,Op_{r,s})}$ c holds (which from now on will be referred to as c is a context in $C^{r\rightarrow s}_{(str,Op_{r,s})}$), we use $c(c' : r_{str} \rightarrow s'_{str})$ to distinguish the proper greatest subcontext c' of c and its associated type.*

We consider **depth** *a predefined function which given a context returns the depth of the occurrence of the context variable. (The depth of the context variable of the trivial context is 0). We will denote by Cb_{str}, the $S\times S$-family of inductive relations defined for every pair $(r, s) \in S \times S$ as $C^{r\rightarrow s}_{(str,Op)}$. In general, given an $S \times S$-family of inductive relations I_{str} of type $(r_{str} \rightarrow s_{str}) \rightarrow Prop$ for every pair $(r, s) \in S \times S$ we will denote by $I^{(r\rightarrow s)}_{str}$ the inductive relation associated to the pair $(r, s) \in S \times S$.*

Definition 3.2 *Given a signature $\Sigma = (S, Op)$, the S-family of observational equalities associated to a set of observable sorts $Obs \subseteq S$, a structure str which inhabits the representation in UTT of Σ, and an $S \times S$-family of inductive relations I_{str} of type $(r_{str} \rightarrow s_{str}) \rightarrow Prop$ for every pair $(r, s) \in S \times S$, is defined as:*

$$\approx^{I_{str}}_{str,Obs} =_{\mathbf{def}}$$

$$\begin{cases} \lambda x : r_{str}.\lambda y : r_{str}. \bigwedge\limits_{obs\in Obs} \forall cx : r_{str} \rightarrow obs_{str}.(I^{r\rightarrow obs}_{str}\ cx) \supset \\ \qquad\qquad (cx\ x) =_{obs_{str}} (cx\ y) & ,if\ r \notin Obs \\ \lambda x : r_{str}.\lambda y : r_{str}.x =_{r_{str}} y & ,if\ r \in Obs \end{cases}$$

Notation: *We will denote by $\approx^{r,I_{str}}_{str,Obs}$: $r_{str} \rightarrow r_{str} \rightarrow Prop$ the relation associated to $r \in S$ in this S-family.*

Remark: *Roughly speaking, this S-family of indistinguishability relations is defined for observable sorts as Leibniz equality and for non-observable sorts as the equality which relates two elements if they are indistinguishable by a set of observable contexts. For a given $r \notin Obs$, this set is determined by the inductive relations $C^{r\rightarrow obs}_{str}$ where $obs \in Obs$.*

Definition 3.3 *Given a first-order signature $\Sigma = (S, Op)$, an inhabitant str of the representation in UTT of Σ, the S-family of equivalence relations $(=_{str})$ with respect to the structure str is an UTT-congruence if for all $f : s_1 \times \ldots \times s_n \rightarrow s \in Op$, the following proposition is provable in UTT:*

$$\vdash_{UTT} \forall x_1 : s_{1_{str}}.\ldots.\forall x_n : s_{n_{str}}.\forall y_1 : s_{1_{str}}.\ldots.\forall y_n : s_{n_{str}}.$$

$$x_1 =^{s_1}_{str} y_1 \wedge \ldots \wedge x_n =^{s_n}_{str} y_n \supset f_{str}(x_1, \ldots , x_n) =^{s}_{str} f_{str}(y_1, \ldots , y_n)$$

Notation: *We will denote by $Cong_{str}(=_{str})$ the conjunction of $Equiv_{str}(=_{str})$ (which determines that family of equivalence relations is reflexive, symmetric and transitive as specified in previous chapter) and the proposition below for all $f : s_1 \times \ldots \times s_n \rightarrow s \in Op$ for a given sort $s \in S$.*

Remark: From now on, we will use the notation $\vdash_{UTT} \phi$ where ϕ is a term of UTT, to express that ϕ is provable in UTT.

Proposition 3.4 *For any first-order signature $\Sigma = (S, Op)$ and for any inhabitant str of the representation in UTT of Σ, the S-family of Leibniz equalities is an UTT-congruence.*

Proposition 3.5 *For any first-order signature $\Sigma = (S, Op)$ and for any inhabitant str of the representation in UTT of Σ, the S-family of observational equalities $\approx^{Cb_{str}}_{str,Obs}$ is an UTT-congruence.*

Since the definition of the observational equality includes universal quantification over contexts, proofs about observational equality will normally use context induction. The formulation of context induction which appears in [Hen91] is based on the induction principle generated by a well-founded set.

Since our definition of contexts is by means of a set of constructors, the new formulation of context induction that we propose is just the inductive principle generated by the set of constructors which define contexts oriented to proofs of observational equality. Since we have modified a little bit the general formulation of the induction principle associated to a set of constructors, we present also a proof of correctness of the context induction principle.

Definition 3.6 *The new formulation of context induction is defined as:*

$$Ind(C^{r\to s}_{(str,Op_{r,s})}\ , Op) = \quad \forall P : (r_{str} \to s_{str}) \to Prop.$$

$$\bigwedge_{cstr \in Ctr(C^{r\to s}_{(str,Op_{r,s})}\ )} Ind\_pr(cstr, C^{r\to s}_{(str,Op_{r,s})}\ , P) \supset$$

$$\forall c : (r_{str} \to s_{str}).\ C^{r\to s}_{(str,Op)}\ \ c \supset P\ c$$

where $Ind\_pr(cstr, C^{r\to s}_{(str,Op_{r,s})}\ , P)$ returns the inductive premise associated to the constructor cstr of $C^{r\to s}_{(str,Op_{r,s})}$ given a proposition $P : (r_{str} \to s_{str}) \to Prop$. For example, if $cstr \in KC(C^{r\to s}_{(str,Op_{r,s})}\ )$ the inductive premise would be defined as

$$\forall v_1 : s_{1str}. \ldots .\forall c_i : r_{str} \to s_{i_{str}}. \ldots .\forall v_n : s_{nstr}.$$

$$C^{r\to s_i}_{(str,Op_{r,s_i})}\ \ c_i \supset P \quad c_i \supset P \quad \lambda x : r_{str}.op_{str}(v_1, \ldots, c_i \quad x, \ldots, v_n)$$

provided that $s_i = s$.

Theorem 3.7 *$Ind(C^{r\to s}_{(str,Op)}, Op)$ is derivable in UTT.*

Proof:
The proof is by induction on natural numbers over the proposition
$\forall n : N.\forall cx : (r_{str} \to s_{str}) \to Prop.(C^{r\to s}_{(str,Op)} cx) \supset depth(cx) = n + 1 \supset P\ cx$

where $P : (r_{str} \rightarrow s_{str}) \rightarrow Prop$. The base case is trivial and the general case is easily provable using the induction hypotheses and the inductive premises of the context induction principle.

3.1 Making easier proofs about observational equality

The main drawback of our formulation of context induction for formal reasoning is that it has in general a considerable number of premises (one for every constructor). If we define this indistinguishability relation with a smaller but still adequate set of contexts, then we can reduce the number of premises of the induction principle associated to the inductive definition of contexts. We have to guarantee that the new formulation of indistinguishability is equivalent to the one above. First, we will formulate a theorem which allows us to reduce the set of contexts provided that the indistinguishability relation associated to this new set is a congruence. A proof of a similar theorem can be found in [BH95]. After that, we will discuss how to make a choice of this set of contexts.

Theorem 3.8 *Let $\Sigma = (S, Op)$ be a signature, let str be a structure which inhabits the representation in UTT of Σ and let Ir_{str} be a $S \times S$-family of inductive relations such that for every pair (r,s), the following proposition holds:*

$$\forall c : r_{str} \rightarrow s_{str}.\ Ir^{r\rightarrow s}_{(str)}\ c \supset C^{r\rightarrow s}_{(str,Op_{r,s})}\ c$$

If the indistinguishability relation $\approx^{C_{str}}_{str,Obs}$ is a UTT-congruence then for all $s \in S$

$$\vdash_{UTT}\ \forall x : s_{str}.\forall y : s_{str}.x\ \approx^{s,Ir_{str}}_{str,Obs}\ y \Leftrightarrow x\ \approx^{s,Cb_{str}}_{str,Obs}\ y$$

Now let's see how one can choose the set of contexts so as to make formal proofs simpler. Although in concrete cases one can develop more interesting simplifications, we present these results because we think that they give an idea of how to reason to make a good choice of contexts. As we mentioned in the definition of contexts, we introduced the parameter $Op_{r,s}$, a distinguished subset of operations of the original signature. This parameter allows us to define subtypes of the original set of contexts with type $C^{r\rightarrow s}_{(str,Op)}$ where the outermost operation symbol of the context has to belong to $Op_{r,s}$. This new set of contexts are enough to get a context induction principle that is simpler to use in practice. Besides, proofs which guarantee that the induced indistinguishability relation is a congruence are especially easier to develop in concrete cases.

Let's see with an example how to choose this set $Op_{r,s}$ and how formal proofs are simplified. Imagine the specification of natural numbers with operations $0 : nat$, $succ : nat \rightarrow nat$ and $+ : nat \times nat \rightarrow nat$. It is clear that contexts generated by the operation $+$ can easily be transformed into contexts with *succ* outermost. Therefore, the operation $+$ would not be included in the subset $Op_{nat,nat}$ and the simplified induction principle would have two premises less.

To guarantee that the indistinguishability relation induced with this smaller set of contexts is a congruence, it is necessary to assume that all the contexts which we have excluded (contexts in $C^{r\to s}_{(str,Op-Op_{r,s})}$) are extensionally equal to some of those that we have chosen (contexts in $C^{r\to s}_{(str,Op_{r,s})}$). Let's now show formally our results:

Theorem 3.9 *Let $\Sigma = (S, Op)$ be a signature, let str be a structure which inhabits the representation in UTT of Σ and let C_{str} be the $S \times S$-family of inductive relations defined for every pair (r,s) as $C^{r\to s}_{(str,Op_{r,s})}$. Assume that the following judgement is derivable in UTT for every $(r, s) \in S \times S$:*

$$\vdash_{UTT} Hyp: \quad \forall c : r_{str} \to s_{str}.\ C^{r\to s}_{(str,Op-Op_{r,s})}\ \ c(sc : r_{str} \to s'_{str}\ ) \supset$$

$$\exists c' : r_{str} \to s_{str}.\ C^{r\to s}_{(str,Op_{r,s})}\ \ c'(sc' : r_{str} \to s'_{str}\ ) \wedge$$

$$\forall v : r_{str}.Eqs_{str}\ (c\ v)\ (c'\ v)$$

Then $\approx^{C_{str}}_{str,Obs}$ is a UTT-congruence.

Proof:
We have to prove that for any $x_1 : s_{1_{str}}, \dots , x_n : s_{n_{str}}$ and any $y_1 : s_{1_{str}}, \dots , y_n : s_{n_{str}}$ assuming that $x_i \approx^{s_i,C_{str}}_{str,Obs} y_i$ for all $i \in [1..n]$ then $f_{str}(x_1, \dots , x_n) \approx^{s,C_{str}}_{str,Obs} f_{str}(y_1, \dots , y_n)$. The idea of the proof is based on the fact that we can easily get a proof by transitivity provided that the following elements are indistinguishable:

$$f_{str}(x_1, \dots , x_n) \approx^{s,C_{str}}_{str,Obs} f_{str}(y_1, x_2, \dots , x_n),$$

$$f_{str}(y_1, x_2, \dots , x_n) \approx^{s,C_{str}}_{str,Obs} f_{str}(y_1, y_2, \dots , y_n),$$

$$\dots , f_{str}(y_1, \dots , y_{n-1}, x_n) \approx^{s,C_{str}}_{str,Obs} f_{str}(y_1, \dots , y_n)$$

To prove for example that
$f_{str}(y_1, \dots , y_{i-1}, x_i, x_{i+1}, \dots , x_n) \approx^{s,C_{str}}_{str,Obs} f_{str}(y_1, \dots , y_{i-1}, y_i, x_{i+1}, \dots , x_n)$ holds it is necessary to differentiate the cases $s \in Obs$ and $s \notin Obs$. Since both proofs are quite similar, we will present just the case $s \notin Obs$.

We have to prove that

$$\bigwedge_{obs\in Obs} \forall cx : s_{istr} \to obs_{str}.(\ C^{s_i\to obs}_{(str,Op_{s_i,obs})}\ \ cx) \supset$$

$$(Eqs_{str}\ (cx\ f(y_1, \dots , y_{i-1}, x_i, x_{i+1}, \dots , x_n))\ (cx\ f(y_1, \dots , y_{i-1}, y_i, x_{i+1}, \dots , x_n)))$$

Let's fix *obs* and *cx*. Again, we have to differentiate the cases $s_i \in Obs$ and $s_i \notin Obs$. We will show just the second case.

Since $x_i \approx^{s_i,C_{str}}_{str,Obs} y_i$, if we guarantee that
$\lambda x : s_{istr}.cx\ f(y_1, \dots , y_{i-1}, x, x_{i+1}, \dots , x_n)$ is in $C^{s_i\to obs}_{(str,Op_{s_i,obs})}$ we get what we

wanted. To show that $\lambda x : s_{i\,str}.cx\, f(y_1, \ldots, y_{i-1}, x, x_{i+1}, \ldots, x_n)$ is in $C^{s_i \to obs}_{(str, Op_{s_i, obs})}$, let $scx : (s_{i\,str} \to s_{j\,str}) \to Prop$ be the greatest subcontext of cx. If $s_i \neq s_j$ there is no problem, but if $s_i = s_j$ and scx is the trivial context we have to apply the assumption to convert the context $\lambda x : s_{i\,str}.f(y_1, \ldots, y_{i-1}, x, x_{i+1}, \ldots, x_n)$ which in general is not in $C^{s_i \to s_j}_{(str, Op_{s_i, s_j})}$ to a context which is in this relation.

As we have said, in concrete cases we can often make a better choice of the subset of observable contexts. Normally, it is a good idea to take the minimum set of operations which characterises the intuitive notion of behaviour of the specification. For example, two sets are indistinguishable if they have the same elements or for the specification of trees, two trees are indistinguishable if after applying to them the same ordering operation, the resulting sequences are indistinguishable. Since in general the definition of observable contexts uses contexts with result sort a non-observable sort, it is also useful to think of reductions of these set of contexts. When the result sort coincides with the sort of the context variable, it is normally a good choice to take just the operation constructors associated to the sort of the context and in some cases just the trivial context.

Finally, we just mention that we do not need to obtain a finite set of contexts as in [BH95], since our formalism is powerful enough to represent infinite sets of observable contexts using inductive definitions. On the other hand, since our type theory includes a higher-order logic, it is possible to give an alternative representation in UTT of the same observational equality based on the definition of [HS96]. We will refer to this new definition as indistinguishability relation (as in [HS96]). This alternative definition is given below and it does not require in general the use of context induction to relate two elements of a given carrier set although it is always possible to use the previous representation in the new one since the representation of the latter is existentially quantified by a congruence.

Definition 3.10 *Given a first-order signature Σ, a set of observable sorts $Obs \subseteq Sorts(\Sigma)$ and a structure str which inhabits the representation in UTT of Σ, the indistinguishability relation for any sort $r \in Sorts(\Sigma)$ is defined as follows:*

$$Indrel^{r}_{str,Obs} = \lambda x : r_{str}.\lambda y : r_{str}.\ \exists R_{s \in S} : s_{str} \to s_{str} \to Prop.R_r(x, y) \wedge$$

$$\Big(\bigwedge_{obs \in Obs} \forall x', y' : obs_{str}.R_{obs}(x', y') \Leftrightarrow x' = y'\Big) \wedge Cong_{str}(R)$$

Proposition 3.11 *For any signature $\Sigma = (S, Op)$, for any set of observable sorts $Obs \subseteq Sorts(\Sigma)$ and for any inhabitant str of the representation in UTT of Σ, the indistinguishability relation is an UTT-congruence.*

One can prove the equivalence of the indistinguishability relation and the observational equality in the same way as in [HS96]:

Proposition 3.12 *Let $\Sigma = (S, Op)$ be a signature, let str be a structure which inhabits the representation in UTT of Σ, let $\approx^{Cb_{str}}_{str,Obs}$ be the observational equal-*

ity and let $Indrel_{str,Obs}$ be the indistinguishability relation. The following proposition holds for any sort $s \in Sorts(\Sigma)$:

$$\forall v, w : s_{str}.v \approx^{s,Cb_{str}}_{str,Obs} w \Leftrightarrow Indrel^{s}_{str,Obs}\; v\; w$$

4 An Example

In this section, we show with a simple example how to prove that a certain structure satisfies a specification. The example is the specification of an abstract data type to store elements called $CONTAINER$. The example appears in [BH95] and we use their simpler version of the observational equality for this concrete signature to prove satisfaction and we also mention how to prove satisfaction using the indistinguishability relation. We also develop the refinement of stacks of elements by lists of elements with a pointer in a similar way as in [Luo94].

4.1 The Container Example

The specification $CONTAINER$ is defined by $\mathbf{Str}[CONTAINER]$ and $\mathbf{Ax}[CONTAINER]$ as follows:

$$\mathbf{Str}[CONTAINER] = \textstyle\sum[Container : Setoid, Elem : Setoid, Nat : Setoid, Bool : Setoid,$$

$$\begin{array}{l}
\emptyset : Dom(Container) \\
insert : Dom(Elem) \times Dom(Container) \rightarrow Dom(Container) \\
union : Dom(Container) \times Dom(Container) \rightarrow Dom(Container) \\
remove : Dom(Elem) \times Dom(Container) \rightarrow Dom(Container) \\
inset : Dom(Elem) \times Dom(Container) \rightarrow Dom(Bool) \\
card : Dom(Container) \rightarrow Dom(Nat) \\
subset : Dom(Container) \times Dom(Container) \rightarrow Dom(Bool) \\
zero : Dom(Nat) \\
succ : Dom(Nat) \rightarrow Dom(Nat) \\
false : Dom(Bool) \\
true : Dom(Bool)
\end{array}$$

and for any $CONTAINER$-structure $str : \mathbf{Str}[CONTAINER]$, $\mathbf{Ax}[CONTAINER]\; str$ is defined as follows:

$$Cong_{str}(Eq[Container]_{str}) \wedge Cong_{str}(Eq[Elem]_{str}) \wedge$$

$$Cong_{str}(Eq[Nat]_{str}) \wedge Cong_{str}(Eq[Bool]_{str}) \wedge$$

$$\forall S, S' : Dom[Container]_{str}.\forall e, e' : Dom[Elem]_{str}.$$

$$Eq[Container]_{str}\; (union_{str}\; \emptyset_{str}\; S)\; S \wedge$$

$Eq[Container]_{str}\ (union_{str}\ (insert_{str}\ e\ S)\ S')\ insert_{str}\ e\ (union_{str}\ S\ S')) \wedge$

$Eq[Container]_{str}\ (remove_{str}\ e\ \emptyset_{str})\ \emptyset_{str} \wedge$

$Eq[Container]_{str}\ (remove_{str}\ e\ (insert_{str}\ e\, S))\ (remove_{str}\ e\ S) \wedge$

$\neg(Eq[Elem]_{str}\ e\ e') \supset$

$\quad Eq[Container]_{str}\ (remove_{str}\ e\ (insert_{str}\ e'\ S))\ (insert_{str}\ e'\ (remove_{str}\ e\ S)) \wedge$

$(inset_{str}\ e\ \emptyset_{str})\ =\ false_{str} \wedge$

$(Eq[Bool]_{str}\ (inset_{str}\ e\ (insert_{str}\ e'\ S))\ true) \Leftrightarrow$

$\quad ((Eq[Elem]_{str}\ e\ e') \vee (Eq[Bool]_{str}\ (inset_{str}\ e\ S)\ true_{str}) \wedge$

$Eq[Nat]_{str}\ card_{str}(\emptyset_{str})\ zero_{str} \wedge$

$(Eq[Bool]_{str}\ (inset_{str}\ e\ S)\ true_{str}) \supset (Eq[Nat]_{str}\ (card_{str}\ (insert_{str}\ e\ S))\ (card_{str}\ S)) \wedge$

$(Eq[Bool]_{str}\ (inset_{str}\ e\ S)\ false_{str}) \supset$

$\quad (Eq[Nat]_{str}\ (card_{str}\ (insert_{str}\ e\ S))\ (succ_{str}\ (card_{str}\ S))) \wedge$

$(Eq[Bool]_{str}\ (subset_{str}\ S\ S')\ true_{str}) \Leftrightarrow (\forall e : Elem.(Eq[Bool]_{str}\ (inset_{str}\ e\ S)\ true_{str}) \supset$

$\quad (Eq[Bool]_{str}\ (inset_{str}\ e\ S')\ true_{str}))$

What we want to prove is that the structure list of natural numbers denoted by $Listn$ is a $CONTAINER$-structure. This structure is defined by a tuple with the following setoids and functions:

$$Listn = (LN, N, N, B, empty\_ln, insert\_ln, union\_ln, remove\_ln, is\_in\_ln,$$

$$card\_ln, subset\_ln, zero\_ln, succ\_ln, true\_ln, false\_ln)$$

where the setoids N and B are defined by the type Nat and $Bool$ respectively and with the predefined Leibniz equality $=_{Nat}$ and $=_{Bool}$, and the setoid list of natural numbers (LN) is defined by the type $List\ Nat$ and the congruence associated to this type is defined as the observational equality $\approx^{Container, Cb'_{Listn}}_{Listn,\{nat,bool\}}$ where assuming that S_c is the set of sorts of the specification $CONTAINER$ and Op_c is the set of operations of the specification $CONTAINER$, Cb'_{Listn} is the $S_c \times S_c$-family of inductive relations defined for $(Container, Nat)$ as $C^{Container \to Nat}_{(\{\}, Listn)} : (LN \to N) \to Prop$, for $(Container, Bool)$ as $C^{Container \to Bool}_{(\{\in\}, Listn)}$: $(LN \to B) \to Prop$, for $(Container, Container)$ as $C^{Container \to Container}_{(\{\}, Listn)}$:

$(LN \to LN) \to Prop$, and for the rest of the cases (r, s) as $C^{r\to s}_{(\{Op_c\}, Listn)}$: $(r_{Listn} \to s_{Listn}) \to Prop$.

One can prove that the $\approx^{Container, Cb'_{Listn}}_{Listn,\{Nat,Bool\}}$ is an UTT-congruence.

As examples of the definition of the functions of this structure, the function *empty_ln* would be defined as *nil Nat*, the function *insert_ln* as *cons Nat*, and the function *remove_ln* would be defined as follows:

$$remove\_ln\; n\; l1 \;=\; Prim\_rec(LN)\; l1\; (ins\_if\_neq\; n)\; l$$

$$where$$

$$ins\_if\_neq\; n\; m\; l\; lf \;=\; Prim\_rec\; Bool\; lf\; (cons\; m\; lf)\; (Eqbool\_nat\; n\; m)$$

To prove that the structure *Listn* is a *CONTAINER*-structure, one has to prove that *Listn* satisfies the axioms of the specification. All the equational subexpressions of the form $Eq[CONTAINER]_{Listn}\; t\; t'$ where $t, t' : List\ Nat$ are transformed into formulas of the form:

$$\forall cxb : (List\ Nat) \to Bool.(\; C^{setn\to bool}_{(str,\{\in\})}\; cxb) \supset Eq_{Bool} \quad (cxb\, t) \quad (cxb\, t')$$

After applying context induction over *cxb*, which is necessary for all equations of our example, one has to provide a proof of the premises of the induction principle, which is:

$$\forall n : Nat.\forall cxl : (List\ Nat) \to (List\ Nat).(\; C^{setn\to setn}_{(str,\{\})}\; cxl) \supset$$

$$Eq_{Bool} \quad is\_in\_ln(n, cxl\, t) \quad is\_in\_ln(n, cxl\, t')$$

Since we have just the trivial context in $C^{setn\to setn}_{(str,\{\})}$, the previous proposition is equivalent to

$$\forall n : Nat.Eq_{Bool} \quad is\_in\_ln(n, t) \quad is\_in\_ln(n, t')$$

which is the proposition we have to proof for every equation $t = t'$ where $t, t' : List\ Nat$. For the rest of subformulas of our specification where the equality is interpreted by Leibniz equality, classical proof techniques can be applied.

If we had chosen as equivalence relation of the setoid LN the indistinguishability relation $Indrel^{Container}_{Listn,\{Nat,Bool\}}$, to prove that the equational subexpressions of the form $Eq[CONTAINER]_{Listn}\; t\; t'$ where t,t':LN, we could use the S_c-family of equivalence relations defined for *Elem* and *Nat* as the Leibniz equality for the type *Nat*, for *Bool* the Leibniz equality for the type *Bool* and for *Container*

$$\lambda t, t' : List\ Nat.\forall n : Nat.Eq_{Bool} \quad is\_in\_ln(n, t) \quad is\_in\_ln(n, t')$$

In this case, the proposition that we must prove for every equation $t = t'$ where $t, t' : ListNat$ is also

$$\forall n : Nat.Eq_{BOOL} \quad is\_in\_ln(n, t) \quad is\_in\_ln(n, t')$$

because we can prove generally that the S_c-family of equivalence relations is an UTT-congruence and it coincides with the Leibniz equality for Nat and $Bool$.

4.2 The refinement example

As we mentioned in the introduction, we develop here the refinement of stacks of natural numbers by a list of natural numbers with a pointer.

Definition 4.1 *The specification of stack of elements is defined by* $\mathbf{Str}[STACK]$ *and* $\mathbf{Ax}[STACK]$ *as follows:*

$$\begin{aligned} Str[STACKN] \ = \ & \textstyle\sum[Stack : Setoid, Elem : Setoid,\\ & empty \ : \ Dom(Stack)\\ & push \ : \ Dom(Elem) \times Dom(Stack) \to Dom(Stack)\\ & pop \ : \ Dom(Stack) \ \to \ Dom(Stack)\\ & top \ : \ Dom(Stack) \ \to \ Dom(Elem)\\ & errelem \ : \ Dom(Elem) \end{aligned}$$

and for any $STACKN$*-structure* $str : \mathbf{Str}[STACKN]$*,* $\mathbf{Ax}[STACKN]\ str$ *is defined as follows:*

$Cong_{str}(Eq[Stackn]_{str}) \ \wedge \ Cong_{str}(Eq[Elem]_{str}) \ \wedge$

$Eq[Stackn]_{str} \ (pop_{str} \ empty_{str}) \ empty_{str} \ \wedge$

$\forall el : Dom[Elem]_{str}.\forall st : Dom[Stackn]_{str}.Eq[Stackn]_{str} \ (pop_{str} \ (push_{str} \ el \ st)) \ st \ \wedge$

$Eq[Elem]_{str} \ (top_{str} \ empty_{str}) \ errelem_{str} \ \wedge$

$\forall el : Dom[Elem]_{str}.\forall st : Dom[Stackn]_{str}.Eq[Elem]_{str} \ (top_{str} \ (push_{str} \ el \ st)) \ el$

Definition 4.2 *The specification of list of elements with pointer is defined by* $\mathbf{Str}[PLIST]$ *and* $\mathbf{Ax}[PLIST]$ *as follows:*

$$\begin{aligned} Str[PLIST] \ = \ & \textstyle\sum[Plist : Setoid, Elem : Setoid,\\ & emptyl \ : \ Dom(Plist)\\ & addl \ : \ Dom(Elem) \times Dom(Plist) \to Dom(Plist)\\ & access \ : \ Nat \ \times \ Dom(Plist) \ \to \ Dom(Elem)\\ & assigninp \ : \ Dom(Elem) \times Dom(Plist) \ \to \ Dom(Plist)\\ & getdim \ : \ Dom(Plist) \ \to \ Nat\\ & initpointer \ : \ Dom(Plist) \ \to \ Dom(Plist)\\ & getpointer \ : \ Dom(Plist) \ \to \ Nat\\ & shiftpl \ : \ Dom(Plist) \ \to \ Dom(Plist)\\ & shiftpr \ : \ Dom(Plist) \ \to \ Dom(Plist)\\ & errelem \ : \ Dom(Elem) \end{aligned}$$

and for any PLIST-structure str : $\mathbf{Str}[PLIST]$, $\mathbf{Ax}[PLIST]$ *str is defined as follows:*

$$Cong_{str}(Eq[Plist]_{str}) \ \wedge\ Cong_{str}(Eq[Elem]_{str}) \wedge$$

$$\forall l : Dom[Plist]_{str}.((Eqbool\_Nat\ (getpointer_{str}\ l)\ (getdim_{str}\ l)) \ =_{Bool}\ true) \supset$$

$$(Eq[Plist]_{str}\ (shiftpr\ l)\ l) \wedge$$

$$\forall el : Dom[Elem].\forall i : Nat.\forall l : Dom[Plist]_{str}.$$

$$((LtBool\_Nat\ (getdim_{str}\ l)\ i) \ =_{Bool}\ true) \supset (Eq[Elem]_{str}\ (access_{str}\ (i, l))\ errelem) \wedge$$

$$(Eq[Elem]_{str}\ (access_{str}\ (i, emptyl_{str}))\ errelem) \wedge$$

$$((EqBool\_Nat\ i\ (succ\ (getdim_{str}\ l))) \ =_{Bool}\ true) \supset$$

$$(Eq[Elem]_{str}\ (access_{str}\ (i, (addl_{str}\ el\ l)))\ el) \wedge$$

$$((LeqBool\_Nat\ i\ (getdim_{str}\ l)) \ =_{Bool}\ true) \supset$$

$$(Eq[Elem]_{str}\ (access_{str}\ (i, (addl_{str}\ el\ l)))\ (access_{str}\ (i, l))) \wedge$$

$$(Eq[Elem]_{str}\ (access_{str}\ (i, (shiftpr_{str}\ l)))\ (access_{str}\ (i, l))) \wedge$$

$$\forall el, el' : Dom[Elem].\forall i : Nat.\forall l : Dom[Plist]_{str}.$$

$$(Eq[Plist]_{str}\ (assigninp_{str}\ (el, emptyl_{str}))\ (shiftpr_{str}\ (addl_{str}\ el\ emptyl_{str}))) \wedge$$

$$(Eq[Plist]_{str}\ (assigninp_{str}\ (el, (addl_{str}\ el'\ l)))\ (addl_{str}\ el'\ (assigninp_{str}\ (el, l)))) \wedge$$

$$(Eqbool\_Nat\ (getdim_{str}\ l)\ (getpointer_{str}\ l) \ =_{Bool}\ true) \supset$$

$$(Eq[Plist]_{str}\ (assigninp_{str}\ (el, (shiftpr\ (add\ el'\ l))))\ (addl_{str}\ el\ l)) \wedge$$

$$((LtBool\_Nat\ (getpointer_{str}\ l)\ (getdim_{str}\ l)) \ =_{Bool}\ true) \supset$$

$$(Eq[Plist]_{str}\ (assigninp_{str}\ (el, (shiftpr_{str}\ (addl_{str}\ el'\ l))))$$

$$(addl_{str}\ el'\ (assigninp_{str}\ (el, (shiftpr_{str}\ l))))) \wedge$$

$\forall el : Dom[Elem].\forall l : Dom[Plist]_{str}.$

$(Eq[Elem]_{str}\ (access_{str}\ ((getpointer_{str}\ l), (assigninp_{str}\ (el, l))))\ el) \wedge$

$\forall el, el' : Dom[Elem].\forall l : Dom[Plist]_{str}.$

$(getpointer_{str}\ emptyl_{str})\ =_{Nat}\ zero \wedge$

$(getpointer_{str}\ (addl_{str}\ el\ emptyl_{str}))\ =_{Nat}\ (succ\ zero) \wedge$

$(getpointer_{str}\ (addl_{str}\ el\ (addl_{str}\ el'\ l))\ =_{Nat}\ (getpointer_{str}\ l) \wedge$

$((LtBool\_Nat\ (getpointer_{str}\ l)\ (getdim_{str}\ l))\ =_{Bool}\ true)\ \supset$

$(getpointer_{str}\ (shiftpr\ l))\ =_{Nat}\ (succ\ (getpointer_{str}\ l)) \wedge$

$((EqBool\_Nat\ (getpointer_{str}\ l)\ (getdim_{str}\ l))\ =_{Bool}\ true)\ \supset$

$(getpointer_{str}\ (shiftpr\ l))\ =_{Nat}\ (getpointer_{str}\ l) \wedge$

$\forall el : Dom[Elem].\forall l : Dom[Plist]_{str}.$

$(getdim_{str}\ emptyl_{str})\ =_{Nat}\ zero \wedge$

$(getdim_{str}\ (addl_{str}\ el\ l)\ =_{Nat}\ (succ\ (getdim_{str}\ l)) \wedge$

$(getdim_{str}\ (shiftpr\ l))\ =_{Nat}\ (getdim_{str}\ l) \wedge$

$\forall el : Dom[Elem].\forall l : Dom[Plist]_{str}.$

$(Eq[Listn]_{str}\ (shiftpl_{str}\ emptyl_{str})\ emptyl_{str}) \wedge$

$(Eq[Listn]_{str}\ (shiftpl_{str}\ (addl_{str}\ el\ l))\ (addl_{str}\ el\ (shiftpl_{str}\ l))) \wedge$

$((EqBool\_Nat\ (getpointer_{str}\ l)\ (getdim_{str}\ l))\ =_{Bool}\ true) \wedge$

$((EqBool\_Nat\ (getdim_{str}\ l)\ (succ\ zero))\ =_{Bool}\ true)\ \supset$

$(Eq[Listn]_{str}\ (shiftpl_{str}\ (shiftpr_{str}\ l))\ l)$

$$((EqBool\_Nat\ (getpointer_{str}\ l)\ (getdim_{str}\ l))\ =_{Bool}\ true) \wedge$$

$$((LtBool\_Nat\ (succ\ zero)\ (getdim_{str}\ l))\ =_{Bool}\ true)\ \supset$$

$$(Eq[Listn]_{str}\ (shiftpl_{str}\ (shiftpr_{str}\ l))\ (shiftpl_{str}\ l))$$

$$((LtBool\_Nat\ (getpointer_{str}\ l)\ (getdim_{str}\ l))\ =_{Bool}\ true)\ \supset$$

$$(Eq[Listn]_{str}\ (shiftpl_{str}\ (shiftpr_{str}\ l))\ l)$$

$$\forall i : Nat.\forall el : Dom[Elem].\forall l : Dom[Plist]_{str}.$$

$$((LtBool\_Nat\ (succ\ zero)\ (getpointer_{str}\ l))\ =_{Bool}\ true) \supset$$

$$(getpointer_{str}\ (shiftpl\ l))\ =_{Nat}\ (decr\ (getpointer_{str}\ l)) \wedge$$

$$(getpointer_{str}\ (assigninp_{str}\ (el, l)))\ =_{Nat}\ (getpointer_{str}\ l) \wedge$$

$$(LtBool\_Nat\ i\ (getpointer_{str}\ l))\ =_{Bool}\ true)\ \supset$$

$$(Eq[Elem]_{str}\ (access_{str}\ (i, (assigninp_{str}\ (n, l))))\ (access_{str}\ (i, l)))$$

Definition 4.3 *The refinement map ρ : $\mathbf{Str[PLIST]} \to \mathbf{Str[STACK]}$, given a structure Pl : $\mathbf{Str[PLIST]}$ returns the following $STACKN$-structure:*

$$Dom[Stack]_{\rho(Pl)}\ =\ Dom[Plist]_{Pl}$$

$$Eq[Stack]_{\rho(Pl)}\ st\ st'\ =\ (getpointer_{Pl}\ st)\ =_{Nat}\ (getpointer_{Pl}\ st') \wedge$$

$$\forall i : Nat.(LeqBool\_Nat\ i\ (getpointer\_Pl\ st)\ =_{Bool}\ true)\ \supset$$

$$(Eq[Elem]_{Pl}\ (acces_{Pl}\ i\ st)\ \ (acces_{Pl}\ i\ st'))$$

$$Dom[Elem]_{\rho(Pl)}\ =\ Dom[Elem]_{Pl}$$

$$Eq[Elem]_{\rho(Pl)}\ =\ Eq[Elem]_{Pl}$$

$$empty_{\rho(Pl)}\ =\ emptyl_{Pl}$$

$$push_{\rho(Pl)}\ el\ st = \ Primrec\ Bool\ (EqBool\_Nat\ (getpointer_{Pl}\ st)\ (getdim_{Pl}\ st))$$

$$(shiftpr_{Pl}\ (addl_{Pl}\ el\ st))\ (assigninp_{Pl}\ (el, (shiftpr_{Pl}\ st)))$$

$$pop_{\rho(Pl)}\ st\ = \ (shiftpl_{Pl}\ st)$$

$$top_{\rho(Pl)}\ st\ = \ (access_{Pl}\ ((getpointer_{Pl}\ st), st))$$

$$errelem_{\rho(Pl)}\ = \ errelem_{Pl}$$

and one can prove that the function ρ is a refinement map in a similar way as in [Luo94].

5 Structuring operators

In [Luo94],different structuring operators to structure specifications have been developed in a similar way as the specification language ASL. One of these operators is defined to put together two subspecifications and other operators are defined to rename, extend or modify a given specification. These operators are related to the basic set of operators of the specification language ASL which we present in the following chapter but we do not think that it is possible to obtain any soundness and/or completeness between the semantics of the two sets of operators. As we mentioned in the introduction, the resulting framework is an expressible and acceptable framework for software design although it has some drawbacks. In this section, we finish to present some of the operators of this framework and in the rest of this thesis we present how to represent an algebraic design frameworks for ASL in a generic way proving soundness and/or completenes between the formal semantics of the original frameworks and its representation.

Definition 5.1 *Let SP and SP' be specifications. Then, the infix specification operation $\bigotimes : SPEC \rightarrow SPEC \rightarrow SPEC$ is defined as follows:*

$$Str[SP\ \bigotimes\ SP']\ =\ Str[SP]\ \times\ Str[SP']$$

and for any s of type $Str[SP\ \bigotimes\ SP']$,

$$Ax[SP\ \bigotimes\ SP'](s)\ =\ Ax[SP](\pi_1(s))\ \wedge\ Ax[SP](\pi_2(s))$$

Definition 5.2 *Given a specification SP, an extension function of $Str[SP]$ of type $Ext\_Str\ :\ Str[SP]\ \rightarrow\ Type$ and some extra axioms on the extended structure $\Sigma s : Str[SP].Ext\_Str(s)$ $Ext\_Ax : \Sigma s : Str[SP].Ext\_Str(s) \rightarrow Prop$, the specification $E \equiv Extend(SP, Ext\_Str, Ext\_Ax)$ with arity*

$$E : \Pi SP : SPEC.\Pi f : Str[SP] \rightarrow Type.\Pi g : \Sigma s : Str[SP].f(s) \rightarrow Prop.SPEC$$

is defined by

$$Str[E] \;=\; \Sigma s : Str[SP].Ext\_Str(s)$$

and for any s of type $s : Str[SP].Ext\_Str(s) \rightarrow Prop$, *Ax[E](s) is defined as*

$$Ax[E](s) \;=\; Ax[SP](\pi_1(s')) \;\wedge\; Ext\_Ax(s')$$

Definition 5.3 *Assume that* $S, S' : Type$ *and* $\rho : S \rightarrow S'$. *The specification operation* $Con_\rho : Spec(S') \rightarrow Spec(S)$ *is defined for any specification* SP' *such that* $S' \equiv Str[SP']$ *as follows:*

$$Str[Con_\rho(SP')] \;=\; S$$

$$Ax[Con_\rho(SP')] \;=\; \exists s' : S'.Ax[SP'](s') \wedge \rho(s') \;=\; s$$

In [Luo94], you can find the proofs that these operators are monotonic with respect to the refinement relation, the definition of others and the definition of parameterised specifications, their instantiation and their refinement.

6 Adequate encodings of logical systems in UTT

The main tasks of software design in an algebraic framework are the the deduction of properties from algebraic specifications, the refinement of specifications or the verification of programs from algebraic specifications. To assist these tasks, sound (and in some cases complete) proof systems with respect to the formal semantics of these tasks have been developed. For the case of design frameworks for ASL, in [HS96], [BCH] and in [Hen97] sound and complete proof systems for the tasks of deduction and refinement have been developed. The main difficulty to give a representation of these proof systems is that some of them are infinitary proof systems and for some cases it is not possible to give a sound and complete representation in a finitary proof system, like for example the type theory we use, since there exist some incompleteness results. In order to be able to give a representation of this kind of proof systems, we redefine the infinitary proof systems as finitary proof systems which are sound with respect to the semantics of the design tasks and then we give adequate encodings of the finitary proof systems in type theory. Some of the finitary proof systems are formulated as natural deduction systems but we will allow in general the use of different sequents to define proof systems like for example all the sequents which are used to define all the type theories of the previous chapter.

Before presenting the redefinition of the proof systems for the deduction of properties of ASL specifications and the refinement of ASL specifications for first-order and higher-order logic in Chapter 5 and the adequate encodings of these proof systems in the appendix and Chapter 6, we present in this chapter the basic representation techniques of natural deduction systems in UTT. First we give the adequate encoding of the fragment of first-order logic presented in the previous chapter in LF, then we give the adequate encoding of the typed lambda calculus and its substitution operation and after that we give the adequate encoding of a functional fragment of a linear type system in UTT which, as we will explain, is not possible to represent with the principle of encoding of LF.

As we mentioned in the introduction, the main objective of this work is to reuse theorem provers of type theories with inductive types to develop theorem provers for design frameworks for algebraic specifications and logical formalisms in general.

6.1 Adequate encoding of first-order logic

In this subsection, we present the adequate encoding in UTT of the following fragment of first-order logic which we will refer to as FOL and which was also encoded in LF in the previous chapter:

$$\frac{\Gamma \cup \phi \Rightarrow_X \phi'}{\Gamma \Rightarrow_X \phi \supset \phi'} \; (\supset i) \qquad \frac{\Gamma \Rightarrow_X \phi \supset \phi' \quad \Gamma \Rightarrow_X \phi}{\Gamma \Rightarrow_X \phi'} \; (\supset e)$$

$$\frac{\Gamma \Rightarrow_X \phi\{t/x\}}{\Gamma \Rightarrow_X \exists x.\phi} \; (\exists I)$$

$$\frac{\Gamma \Rightarrow_X \exists x.\phi \quad \Gamma \cup \{\phi\} \Rightarrow_{X \cup \{x\}} \psi}{\Gamma \Rightarrow_X \psi} \; (\exists E)$$

$$\frac{\Gamma \Rightarrow_{X \cup \{x\}} \phi}{\Gamma \Rightarrow_X \forall x.\phi} \; (\forall\, I)$$

$$\frac{\Gamma \Rightarrow_X \forall x.\phi}{\Gamma \Rightarrow_X \phi\{t/x\}} \; (\forall\, E)$$

Before presenting the encoding, we give some extra definitions of logical systems which are needed for the presentation of the encoding and its proof of adequacy.

Definition 6.1 *The sequent $\Gamma \Rightarrow_X \phi$ is closed iff ϕ and all the formulas in Γ are closed under X.*

Notation: *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 6.2 *A rule is closed if the sequents of the premises and the sequent of its conclusion are closed.*

Definition 6.3 *The set of derivations of a sequent $\Gamma \Rightarrow_X \phi$ in FOL is denoted by $\Delta_{FOL}(\Gamma \Rightarrow_X \phi)$ and recursively defined as follows:*

- *if $\phi \in \Gamma$ then $\Gamma \Rightarrow_X \phi \in \Delta_{FOL}(\Gamma \Rightarrow_X \phi)$.*
- *if $r \in FOL$, $(\Gamma \Rightarrow_X \phi, \Gamma_1 \Rightarrow_{X_1} \phi_1, \ldots, \Gamma_n \Rightarrow_{X_n} \phi_n)$ is an instance of the rule r, $\delta_1 \in \Delta_{FOL}(\Gamma_1 \Rightarrow_{X_1} \phi_1)$, $\ldots$ and $\delta_n \in \Delta_{FOL}(\Gamma_n \Rightarrow_{X_n} \phi_n)$, then $r(\Gamma \Rightarrow_X \phi, [\delta_1, \ldots, \delta_n]) \in \Delta_{FOL}(\Gamma \Rightarrow_X \phi)$.*

Notation: *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 Δ_Π.*

Definition 6.4 *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.*

As we explained in chapter 2, an encoding of a proof system can be considered adequate if there is an exact correspondence between syntax and proofs of the object system and their encodings. Our new encoding is based on the principle of proof systems as inductive relations. Using inductive relations and this encoding principle will allow us to make a more precise representation of syntax, proof systems and derivations of logical systems, and what is more important, we will be able to use primitive recursive operations to describe the whole encoding of our proof system. See also chapter 2 for a general comparison between our principle of encoding and the principle of encoding of LF.

A technical difference with respect to the work of [HHP93] and [Gar92] is the encoding of syntax. We won't identify bound variables of formulas with variables of the type theory. Instead, we will encode them using the underlying idea of deBruijn indexes which will allow us to make a trivial conversion between free and bound variables. Finally, note that because of the general definition of derivation which we give above, the proof of adequacy does not need to redefine the object logical system making more explicit in the sequent some notion of correct proof expression.

The encoding of the syntax of our logic requires the representation in the type theory of variables, contexts, terms, formulas and environments (set of assumptions). All of them are defined as inductive types. Besides, it is necessary to define some functions over these inductive types, like for example the substitution of a term by a variable in a term or a formula. Just remember that in this type theory we have no operator for general recursion, but instead one can use the operators of primitive recursion associated to each inductive type to define this kind of operations.

In order to avoid the necessity of α-conversion in the substitution operation on formulas, we need a non-trivial encoding of variables. Thus, the inductive type which define variables is defined as a product type including always its variable name, a variable index and in some logics additional information like for example its associated sort or type. Variable names are defined as non-empty sequences of characters, numbers or the symbols \$ or _. Since we can assume that the infinite set of variables is countable, variable indexes are trivially defined as inductive types. These indexes are assigned during the encoding of terms and formulas using the underlying idea of deBruijn indexes, and the deBruijn indexes for bounded variables start from the greatest index assigned to the free variables of the formula. It is obvious that using the variable names together with the variable indexes as identifiers no name clashes can occur during substitution and therefore no α-conversion is needed to define this operation. Note also that we do not lose readability because we preserve the original names of the variables.

In the following, we proceed with the whole encoding of the proof system presenting first the encoding of variables, encoding of terms and formulas, encoding of well-formed terms and formulas and after that the adequacy of the representation of synstax and finally the adequate encoding of the proof system.

6.1.1 Encoding of variables

In the following we present the definitions of the inductive types for variable symbols, variable names, variable indexes, indexed variable and set of indexed variables. First, we define variable symbols which determine the symbols of variable names.

Definition 6.5 *The type Var_symbol is inductively defined by the following set of constructors:*

$$a, \ldots, z : Var\_symbol$$

$$A, \ldots, Z : Var\_symbol$$

$$\_, ', \$: Var\_symbol$$

Remark: *We assume predefined the equality function $Eqbool\_Vs : Var\_symbol \rightarrow Var\_symbol \rightarrow Bool$.*

Next, we define variable names as non-empty sequences of variable symbols.

Definition 6.6 *For any type $T : Type_0$, the inductive type $Nelist\ T$ is defined by the following constructors:*

$$first\_Nel\ :\ T\ \rightarrow\ (Nelist\ T)$$

$$cons\_Nel\ :\ T\ \rightarrow\ (Nelist\ T)\ \rightarrow\ (Nelist\ T)$$

Definition 6.7 *The type $Varname$ is defined as follows:*

$$Varname\ =\ Ne\_list\ Var\_symbol$$

Remark: *We assume predefined the equality function $Eqbool\_Vn : Var\_name \rightarrow Var\_name \rightarrow Bool$.*

Next, we define variables with indexes as a pair of a variable name and variable index.

Definition 6.8 *The type Var_index is inductively defined by the following set of constructors:*

$$first\_Vi : Var\_index$$

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

Remark: *We assume predefined the equality function $Eqbool\_Vi : Var\_index \rightarrow Var\_index \rightarrow Bool$.*

Definition 6.9 *For any $\Sigma \in |AlgSig|$, the type $Invarn$ is defined as:*

$$Invarn \;=\; Pair\ Varname\ Var\_index$$

Remark: *We assume predefined the equality function $Eqbool\_Ivarn : Invarn \rightarrow Invarn \rightarrow Bool$.*

Finally, we define set of variables and some operations on it. The basic operations on set of variables are to define the empty set, to add a variable with a new index to the variable set, to get the indexed variable with the greatest index of a given variable name from a variable set, and an inductive relation to check whether a given variable is in the variable set.

Definition 6.10 *The type Var_set is defined as:*

$$Var\_set = pair\ Var\_index\ (List\ Invarn)$$

Definition 6.11 *The function*

$$empty\_Vst : Var\_set$$

is defined as follows:

$$empty\_Vst \;=\; mkpair\ Var\_set\ first\_Vi\ (nil\ Invarn)$$

Definition 6.12 *The function*

$$addvar\_Vst : Varname \rightarrow Var\_set \rightarrow Var\_set$$

is defined as follows:

$$addvar\_Vst\ vn\ vs \;=\; mkpair\ Var\_set$$

$$(next\_Vi\ (fst\ vs))\ (cons\ Invarn\ (mkpair\ Invarn\ vn\ (fst\ vs))\ (snd\ vs))$$

Definition 6.13 *The function*

$$getvar\_Vst : Varname \rightarrow Var\_set \rightarrow Invarn$$

is defined as follows:

$$getvar\_Vst\ vn\ vs\ =$$

$$Primrec\ (List\ Invarn)\ (mkpair\ Invarn\ v\ (fst\ vs))$$

$$(get\_if\_eq\ vn)\ (snd\ vs)$$

where

$$get\_if\_eq\ vn\ v'\ ivl\ vf =$$

$$Prim\_rec\ Bool\ v'\ vf\ (Eqbool\_Vn\ vn\ (fst\ v'))$$

Definition 6.14 *The inductive relation*

$$Is\_in\_Vst : \Pi v : Varname.\Pi vs : Var\_set.Prop$$

is defined by the following set of constructors:

$$ctr\_Invs : \Pi hv : Varname.\Pi vs : Var\_set.$$

$$\Pi isinpr : Is\_in\_ivl\ v\ (snd\ hvs).Isin\_Vst\ v\ vs$$

Definition 6.15 *The inductive relation*

$$Is\_in\_ivl : \Pi v : Varname.\Pi vs : List\ Invarn.Prop$$

is defined by the following set of constructors:

$$base\_Invs : \Pi v, v' : Varname.\Pi ivl : List\ Invarn.$$

$$\Pi eqpr : Eqbool\_Vn\ v\ v'\ =_{bool}\ true.Is\_in\_ivl\ v$$

$$(cons\ Invarn\ (getvar\_Vst\ v'(addvar\_Vst\ v'\ vs))\ ivl)$$

$$genc\_Invs : \Pi v : Varname.\Pi iv : Invarn.\Pi ivl : List\ Invarn.$$

$$\Pi pr : Is\_in\_ivl\ v\ ivl.$$

$$Is\_in\_ivl\ v\ (cons\ Invarn\ iv\ ivl)$$

6.1.2 Encoding of terms and formulas

And now we define the inductive types of terms, formulas and their associated substitution operation. The inductive type for terms is defined with one constructor to define variables and one constructor for each function symbol. The constructor names of the overloaded function symbols have the arity of the function symbols as component of the constructor names.

Definition 6.16 *For any single sorted first-order signature, the inductive type Term is defined by the following set of constructors:*

$$\{var\_Trms : Invarn \ \rightarrow \ Term\}\cup$$

$$\{f\_Trm : Term \rightarrow \ldots \rightarrow Term \rightarrow Term \mid$$

$$f : n \in \Sigma \ \textit{and} \ f \ \textit{is not overloaded in} \ \Sigma\}\cup$$

$$\{f\_n\_Trm : Term \rightarrow \ldots \rightarrow Term \rightarrow Term \mid$$

$$f : n \in \Sigma \ \textit{and} \ f \ \textit{is overloaded in} \ \Sigma\}$$

The substitution operation on terms is trivially defined using primitive recursion replacing the ocurrences of a given variable in a term by another term.

Definition 6.17 *For any signature* Σ

$$subst\_Trm : Term \rightarrow Term \rightarrow Invarn \rightarrow Term$$

is defined as follows:

$$subst\_Trm\; trm\; trm'\; v \;=\; Primrec\; Term\; (varc\; trm'\; v)\; (func\_1\; trm'\; v)\; \ldots$$

$$(func\_n\; trm'\; v)\; (funovc\_1\; trm'\; v)\; \ldots\; (funovc\_m\; trm'\; v)\; trm$$

where

$$varc\; trm\; v\; v' \;=\; Primrec\; Bool\;\; trm\; v'\; (Eqbool\_Ivarn\; v\; v')$$

$$func\_1\; trm\; v\; trm\_11\; \ldots\; trm\_1n_1\; trms\_11 \ldots trms\_1n_1 \;=$$

$$f_1\_Trms\ trms\_11\ \ldots\ trms\_1n_1$$

$$\vdots$$

$$func\_n\ trm\ v\ trm\_n1\ \ldots\ trm\_nn_n\ trms\_n1\ \ldots\ trms\_nn_n =$$

$$f_n\_Trms\ trms\_n1\ \ldots\ trms\_nn_n$$

$$funovc\_1\ trm\ v\ trm\_11\ \ldots\ trm\_1m_1\ trms\_11\ \ldots\ trms\_1m_1 =$$

$$g_1\_m_1\_Trms\ trms\_11\ \ldots\ trms\_1m_1$$

$$\vdots$$

$$funovc\_m\ trm\ v\ trm\_m1\ \ldots\ trm\_mm_m\ trms\_m1 \ldots\ trms\_mm_m =$$

$$g_m\_m_m\_Trms\ trms\_m1\ \ldots\ trms\_mm_m$$

where $f_1 : n_1, \ldots, f_n : n_n, g_1 : m_1, \ldots, g_m : m_m$.

The inductive types for formulas is defined with one constructor for each logical operator.

Definition 6.18 *For any signature, the type Formula is defined by the following set of constructors:*

$$equal\_Form : Term \to Term \to Formula$$

$$implies\_Form : Formula \to Formula \to Formula$$

$$exists\_Form : Invarn \to Formula \to Formula$$

$$forall\_Form : Invarn \to Formula \to Formula$$

And the substitution operation of a free variable by a term in a given term is trivially defined by primitive recursion as follows:

Definition 6.19 *The function*

$$subst\_Form : Formula \to Term \to Invarn \to Formula$$

is defined as follows:

$subst\_Form\ form\ trm\ iv\ =$

$Primrec\ Formula\ (equalfc\ trm\ iv)\ (impliesc\ trm\ iv)$

$(existsc\ trm\ iv)\ (forallc\ trm\ iv)\ form$

$where$

$equalfc\ trm\ iv\ strm\ strm' =$

$equal\_Htrm\ (subst\_Trm\ strm\ trm\ iv)\ (subst\_Trm\ strm'\ trm\ iv)$

$impliesc\ trm\ iv\ form\ sform\ form'\ sform'\ =\ implies\_Htrm\ sform\ sform'$

$forallc\ trm\ iv\ iv'\ strm\ strmf\ =\ forall\_Htrm\ iv'\ strmf$

$existsc\ trm\ iv\ iv'\ strm\ strmf\ =\ exists\_Htrm\ iv'\ strmf$

6.1.3 Encoding of well-formed terms and formulas

For the adequacy of syntax, it is required to represent well-formed terms and well-formed formulas closed by a set of variables. As we explained before, the bound variables of well-formed formulas are indexed with deBruijn indexes starting from the greatest index assigned to the set of free variables.

Definition 6.20 *The inductive relation*

$$Wfterm : Var\_set \to Term \to Prop$$

is defined by the following set of constructors:

$\{ass\_tr : \Pi vs : Var\_set.\Pi v : Invarn.\Pi pr : Is\_in\_Vst\ (fst\ v)\ vs.$

$Wfterm\ \ vs\ \ (var\_s\_Trms\ v)\ \ \} \cup$

$\{appl\_f\_tr : \Pi vs : Var\_set.\Pi t_1 : Term. \dots .\Pi t_n : Term.$

$\Pi wft_1 : Wfterm\ vs\ t_1 \dots .\Pi wft_n : Wfterm\ vs\ \ t_n.$

$Wfterm\ vs\ (f\_Trms\ t_1\ \dots\ t_n)$

$f : n\ and\ f\ \ is\ not\ overloaded\ in\ \Sigma\} \cup$

$$\{appl\_f\_n\_tr : \Pi vs : Var\_set.\Pi t_1 : Term.\ldots.\Pi t_n : Term.$$

$$\Pi wft_1 : Wfterm\ vs\ t_1.\Pi wft_n : Wfterm\ vs\ t_n.$$

$$Wfhterm\ vs\ (f\_n\_Trm\ t_1 \ldots t_n)) \mid$$

$$f : n\ and\ f\ \ is\ overloaded\ in \Sigma\}$$

Definition 6.21 *The inductive relation*

$$Wfform : Var\_set \to Formula \to Prop$$

is defined by the following set of constructors:

$eq\_fcr : \Pi vs : Var\_set.\Pi t, r : Term.\Pi wft : Wfterm\ vs\ t.\Pi wfr : Wfterm\ vs\ r.$

$\quad Wfform\ vs\ (equal\_Form\ t\ r)$

$implies\_fcr : \Pi vs : Var\_set.\Pi \phi_1, \phi_2 : Formula.$

$\quad \Pi wfpr_1 : Wfform\ vs\ \phi_1.\Pi wfpr_2 : Wfform\ vs\ \phi_2.$

$\quad\quad Wfform\ vs\ (implies\_Form\ \phi_1\ \phi_2)$

$forall\_fcr : \Pi vs : Var\_set.\Pi vn : Varname.\Pi \phi : Formula.$

$\quad \Pi prp : Wfform\ (addvar\_Vst\ vn\ vs)\ \phi.$

$\quad Wfform\ vs\ (forall\_Form\ (getvar\_Vst\ vn\ (addvar\_Vst\ vn\ vs))\ \phi)$

$exists\_fcr : \Pi vs : Var\_set.\Pi vn : Varname.\Pi \phi : Formula.$

$\quad \Pi prp : Wfform\ (addvar\_Vst\ vn\ vs)\ \phi.$

$\quad\quad Wfform\ vs\ (exists\_Form\ (getvar\_Vst\ vn\ (addvar\_Vst\ vn\ vs))\ \phi)$

Definition 6.22 *The inductive relation*

$$Wfforml : Var\_set \to (List\ \ Formula) \to Prop$$

is defined by the following set of constructors:

$$nil\_Wffl : \Pi vs : Var\_set.Wfforml\ vs\ (nil\ Formula)$$

$$cons\_Wffl : \Pi vs : Var\_set.\Pi f : Formula.\Pi fl : List\ Formula.$$

$$\Pi wffp : Wfform\ vs\ f.\Pi wfflp : Wfforml\ vs\ fl.$$

$$Wfforml\ vs\ (cons\ Formula\ f\ fl)$$

6.1.4 Adequate representation of syntax

One can easily define encoding and decoding functions of variable names, variable sets, terms, formula and lists of formula with the following arities:

$$\epsilon_{vn} : X \to Varname$$

$$\epsilon_{vn}^{-1} : Varname \to X$$

$$\epsilon_{vs} : [X] \to (Var\_set)$$

$$\epsilon_{vs}^{-1} : (Var\_set) \to [X]$$

$$\epsilon_{hivl}^{-1} : (List\ \ Invarn) \to [X]$$

$$\epsilon_{t} : Var\_set \to T_{\Sigma}(X) \to Term$$

$$\epsilon_{t}^{-1} : Var\_set \to Term \to T_{\Sigma}(X)$$

$$\epsilon_{f} : Var\_set \to Sen_{FOL}(\Sigma, X) \to Formula$$

$$\epsilon_{f}^{-1} : Var\_set \to Formula \to Sen_{FOL}(\Sigma, X)$$

$$\epsilon_{fl} : Var\_set \to [Sen_{FOL}(\Sigma, X)] \to (List\ Formula)$$

$$\epsilon_{fl}^{-1} : Var\_set \to (List\ Formula) \to [Sen_{FOL}(\Sigma, X)]$$

and the encoding and decoding functions of well-formed terms, well-formed formulas and list of well-formed formulas with the following arities:

- ϵ_{wft} which given $vs : Var\_set$, $t \in T_{\Sigma}(X)$ returns an element of the following set:

  $$\{\ t : Term \mid (Wfterm\ vs\ t)\ \}$$

- ϵ_{wft}^{-1} which given $vs : Var\_set$ and an element of the set

  $$\{\ t : Term \mid (Wfterm\ vs\ t)\ \}$$

  returns an element $t \in T_{\Sigma}(X)$.

- ϵ_{wff} which given $vs : Var\_set$, $f \in Sen_{FOL}(\Sigma, X)$ returns an element of the following set:

$$\{\ f : Formula \mid (Wfform\ vs\ f)\ \}$$

- ϵ^{-1}_{wff} which given $vs : Var\_set$ and an element of the set

$$\{\ f : Formula \mid (Wfform\ vs\ f)\ \}$$

  returns an element $f \in Sen_{FOL}(\Sigma, X)$.

where $T_\Sigma(X)$ denotes the term algebra generated by the signature Σ with a finite set of free variables X and in this case, $Sen_{FOL}(\Sigma, X)$ denotes the set of closed first-order fomulas generated just by the operators $=, \supset, \exists, \forall$. See the full encoding of a higher-order logic in the appendix for the complete definition of similar encoding and decoding functions.

Definition 6.23 *The substitution operation on terms* $\_\{\_/\_\} : T_\Sigma(X) \rightarrow T_\Sigma(X) \rightarrow X \rightarrow T_\Sigma(X)$ *for any signature* Σ *is inductively defined as follows:*

$$\begin{array}{lll} y\ \{t\ /\ x\} & = y & , if x\ =\ y \\ & = t & , otherwise \\ & & \\ f\ (t_1, \ldots, t_n)\ \{t\ /\ x\}\ = & f\ (t_1\ \{\ t\ /\ x\ \}, \ldots, t_n\{\ t\ /\ x\ \}) & \end{array}$$

where

$$t, t_1, \ldots, t_n \in T_{\Sigma,r}(X),\quad f\ : n\ \in\ \Sigma$$

Definition 6.24 *The substitution operation on formulas* $\_\{\_/\_\} : Sen_{FOL}(\Sigma, X) \rightarrow T_\Sigma(X) \rightarrow X$ *for any signature* $\Sigma\ \in\ AlgSig$ *is inductively defined as follows:*

$$\begin{array}{lll} y\ \{t\ /\ x\} & = y & , if x\ =\ y \\ & = t & , otherwise \end{array}$$

$$t_1\ =\ t_2\ \{\ t\ /\ x\ \}\ =\ \ (t_1\ \{\ t\ /\ x\ \}\ =\ t_2\{\ t\ /\ x\ \})$$

$$\begin{array}{lll} \exists x.\phi\ \{t\ /\ y\} & =\ \exists x'.((\phi\{x'/x\})\ \{t\ /\ y\}) & , if\ y \neq x \\ & =\ \exists x.\phi\ \{t\ /\ y\} & , if\ y = x \end{array}$$

where $\qquad l$

$$(x \notin FV(t))\ \Rightarrow\ (x'\ =\ x) \wedge$$

$$(x \in FV(t))\ \Rightarrow\ ((x' \notin FV(t)) \wedge (x' \notin FV(\phi)) \wedge (x' \notin BV(\phi)))\ ,$$

$$\forall x.\phi \; \{t \,/\, y\} \;=\; \forall x'.((\phi\{x'/x\})\; \{t \,/\, y\}) \qquad , if y \neq x$$

$$\;=\; \forall x.\phi \qquad , if y = x$$

where

$$(x \notin FV(t)) \;\Rightarrow\; (x' \;=\; x) \wedge$$

$$(x \in FV(t)) \;\Rightarrow\; ((x' \notin FV(t)) \wedge (x' \notin FV(\phi)) \wedge (x' \notin BV(\phi))$$

$$\phi \supset \phi' \; \{t \,/\, x\} \;=\; \phi \; \{t \,/\, x\} \;\supset\; \phi' \; \{t \,/\, x\}$$

where $FV(t)$ and $FV(\phi)$ are the free variables of term t and formula ϕ respectively, and $BV(t)$ and $BV(\phi)$ are the bound variables of term t and formula ϕ calculated in the obvious way.

We can prove the following adequacy theorem of syntax:

Theorem 6.25 • *There exists a bijection between first-order formulas closed under the finite set of free variables X and the normal forms of formulas $form$: $Formula$ which are well formed, i.e. that the proposition $Wfform$ $(\epsilon_{vs} \quad X)$ $form$ is provable in the type theory and*

• *this bijection is preserved under the substitution operation*

Proof:

1. One can trivially prove that the encoding function of syntax ϵ_{wft} and ϵ_{wff} are total and injective. To prove the bijection, we use the decoding functions ϵ^{-1}_{wff} and ϵ^{-1}_{wft} defined in a similar way as the ones of higher-order logic. We can also prove that these decoding functions are injective and total. Note that this would not be possible if we take as domain of the decoding function the set of formulas belonging to the inductive type *Formula*. Finally, one can prove for all $form \in Sen_{FOL}(\Sigma, X)$ that

$$\epsilon^{-1}_{wff} \quad (\epsilon_{vs} \quad X) \quad (\epsilon_{wff} \quad (\epsilon_{vs} \quad X) \quad form) = form$$

   by induction on $form$, which guarantees the bijection stated in 1).

2. This is guaranteed by proving that encoding and substitution commutes, which can be formulated by the following equation for all closed formulas $form \in Sen_{FOL}(\Sigma, X)$, all term $t \in T_\Sigma(X)$ and all free variables x in X:

$$subst\_f \quad (\epsilon_{wff} \quad (\epsilon_{vs} \quad X) \quad form) \quad (\epsilon_{wft} \quad (\epsilon_{vs} \quad X) \quad t)$$
$$(getvar\_Vst \quad (\epsilon_{vn} \quad x) \quad (\epsilon_{vs} \quad X)) \quad = \quad (\epsilon_{wff} \, (\epsilon_{vs} \quad X) \, (form \; \{t/x\}))$$

   This equation is trivially provable by induction over $form$.

6.1.5 Adequate encoding of the proof system

The fragment of first-order logic presented in this section is encoded in UTT as the inductive relation with type $FOL : \Pi Env : List\ Formula.\Pi vs : Var\_set.\Pi form : Formula.Prop$ defining for each rule of the logical system, a constructor in the inductive relation as follows:

$$
\begin{array}{l}
impl\_i : \Pi env : List\ Formula.\Pi vs : Var\_set.\Pi \phi, \phi' : Formula. \\
\quad \Pi wfenv : Wfforml\ vs\ env. \\
\quad\quad \Pi wff : Wfform\ vs\ \phi.\Pi wff' : Wfform\ vs\ \phi'. \\
\quad\quad\quad \Pi prd : FOL\ (cons\ Formula\ \phi\ env)\ vs\ \phi'. \\
\quad\quad\quad\quad FOL\ env\ vs\ (implies\_Form\ \phi\ \phi')
\end{array}
$$

$$
\begin{array}{l}
impl\_e : \Pi env : List\ Formula.\Pi vs : Var\_set.\Pi \phi, \phi' : Formula. \\
\quad \Pi wfenv : Wfforml\ vs\ env. \\
\quad \Pi wff : Wfform\ vs\ \phi.\Pi wff' : Wfform\ vs\ \phi'. \\
\quad\quad \Pi prd : FOL\ env\ vs\ (implies\_Form\ \phi\ \phi').\Pi prd' : FOL\ env\ vs\ \phi. \\
\quad\quad FOL\ env\ vs\ \phi'
\end{array}
$$

$$
\begin{array}{l}
forall\_i : \Pi env : List\ Formula.\Pi vs : Var\_set.\Pi vn : Varname.\Pi \phi : Formula. \\
\quad \Pi wfenv : Wfforml\ vs\ env.\Pi wff : Wfform\ (addvar\_Vst\ vn\ vs)\ \phi. \\
\quad\quad \Pi dpr : FOL\ env\ (addvar\_Vst\ vn\ vs)\ \phi. \\
\quad\quad\quad FOL\ env\ vs\ (forall\_Htrm\ (getvar\_Vst\ vn\ (addvar\_Vst\ vn\ vs))\ \phi)
\end{array}
$$

$forall\_e : \Pi env : List\ Formula.\Pi vs : Var\_set.\Pi vn : Varname.\Pi \phi : Formula.\Pi t : Term.$

$\Pi wfp : Wfforml\ vs\ env.$

$\Pi wff : Wfform\ (addvar\_Vst\ vn\ vs)\ \phi.\Pi wft : Wfterm\ vs\ t.$

$\Pi dpr : FOL\ env\ vs\ (forall\_Form\ (getvar\_Vst\ vn\ (addvar\_Vst\ vn\ vs))\ \phi).$

$FOL\ env\ vs\ (subst\_Form\ \phi\ t\ (getvar\_Vst\ vn\ (addvar\_Vst\ vn\ vs)))$

$exists\_i : \Pi env : List\ Formula.\Pi vs : Var\_set.\Pi vn : Varname.\Pi \phi : Formula.\Pi t : Term.$

$\Pi wfp : Wfforml\ vs\ env.$

$\Pi wff : Wfform\ (addvar\_Vst\ vn\ vs)\ \phi.\Pi wft : Wfterm\ vs\ t.$

$\Pi dpr : FOL\ env\ vs\ (subst\_Form\ \phi\ t\ (getvar\_Vst\ vn\ (addvar\_Vst\ vn\ vs))).$

$FOL\ env\ vs\ (exists\_Form\ (getvar\_Vst\ vn\ (addvar\_Vst\ vn\ vs))\ \phi)$

$exists\_e : \Pi env : List\ Formula.\Pi vs : Var\_set.\Pi vn : Varname.\Pi \phi, \phi' : Formula.$

$\Pi wfenv : Wfforml\ vs\ env.\Pi wff : Wfform\ env\ (addvar\_Vst\ vn\ vs)\ \phi.$

$\Pi wff' : Wfform\ env\ \phi'.$

$\Pi dpr : FOL\ (cons\ env\ \phi)\ (addvar\_Vst\ vn\ vs)\ \phi'.$

$\Pi dpr' : FOL\ env\ vs\ (exists\_Form\ (getvar\_Vst\ vn\ (addvar\_Vst\ vn\ vs))\ \phi).$

$FOL\ env\ vs\ \phi'$

For the encoding and decoding functions of the proof system for first-order logic, we will also assume predefined the following functions:

- ϵ_{wftp} which given a variable set $vs : Var\_set$, $t \in T_\Sigma$ returns an inhabitat of $(Wfterm\ vs\ (\epsilon_t\ (\epsilon_{vs}\ vs)\ t))$
- ϵ_{wffp} which given a variable set $vs : Var\_set$, $f \in Sen_{FOL}(\Sigma)$ returns an inhabitat of $(Wfform\ vs\ (\epsilon_f\ (\epsilon_{vs}\ vs)\ f))$
- ϵ_{wfflp} which given a variable set $vs : Var\_set$, $fl \in [Sen_{FOL}(\Sigma)]$ returns an inhabitat of $(Wfforml\ vs\ (\epsilon_{fl}\ (\epsilon_{vs}\ vs)\ fl))$

The encoding function of derivations of FOL ϵ_{fd} which given a closed derivation in $\Delta_{\Pi_{FOL}}(\Gamma \Rightarrow_X \phi)$ returns a proof of the proposition

$$FOL\ (\epsilon_{fl}\ (\epsilon_{vs}\ X)\ \Gamma)\ \ (\epsilon_{vs}\ X)\ (\epsilon_f\ (\epsilon_{vs}\ X)\ \phi)$$

is inductively defined by closed derivations as follows:

$\epsilon_{fd}\ (impl\_i\ (\Gamma \Rightarrow_X \phi \supset \phi', [\delta]))\ =$

$impl\_i\ (\epsilon_{fl}\ (\epsilon_{vs}\ X)\ \Gamma)\ (\epsilon_{vs}\ X)\ (\epsilon_f\ (\epsilon_{vs}\ X)\ \phi)\ (\epsilon_f\ (\epsilon_{vs}\ X)\ \phi')$

$(\epsilon_{wfflp}\ (\epsilon_{vs}\ X)\ \Gamma)\ (\epsilon_{wffp}\ (\epsilon_{vs}\ X)\ \phi)\ (\epsilon_{wffp}\ (\epsilon_{vs}\ X)\ \phi')\ \ (\epsilon_{fd}\ \delta)$

$where\ \delta \in \Delta_{\Pi_{HOL}}(\Gamma \cup \phi \Rightarrow_X \phi').$

$\epsilon_{fd}\ (impl\_e\ (\Gamma \Rightarrow_X \phi', [\delta_1, \delta_2]))\ =$

$impl\_e\ (\epsilon_{fl}\ (\epsilon_{vs}\ X)\ \Gamma)\ (\epsilon_{vs}\ X)\ (\epsilon_f\ (\epsilon_{vs}\ X)\ \phi)\ (\epsilon_f\ (\epsilon_{vs}\ X)\ \phi')$

$(\epsilon_{wfflp}\ (\epsilon_{vs}\ X)\ \Gamma)\ (\epsilon_{wffp}\ (\epsilon_{vs}\ X)\ \phi)\ (\epsilon_{wffp}\ (\epsilon_{vs}\ X)\ \phi')\ \ (\epsilon_{fd}\ \delta_1)\ (\epsilon_{fd}\ \delta_2)$

$where\ \delta_1 \in \Delta_{\Pi_{FOL}}(\Gamma \Rightarrow_X \phi \supset \phi'), \delta_2 \in \Delta_{\Pi_{FOL}}(\Gamma \Rightarrow_X \phi).$

$\epsilon_{fd}\ (forall\_i(\Gamma \Rightarrow_X \forall x.\phi, [\delta]))\ =$

$forall\_i\ (\epsilon_{fl}\ (\epsilon_{vs}\ X)\ \Gamma)\ (\epsilon_{vs}\ X)\ (\epsilon_{vn}\ x)(\epsilon_f\ (addvar\_Vst\ (\epsilon_{vn}\ x)(\epsilon_{vs}\ X))\ \phi)$

$(\epsilon_{wfflp}\ (\epsilon_{vs}\ X)\ \Gamma)\ (\epsilon_{wffp}\ (addvar\_Vst\ (\epsilon_{vn}\ x)(\epsilon_{vs}\ X))\ \phi)\ \ (\epsilon_{fd}\ \delta)$

$where\ \delta \in \Delta_{\Pi_{FOL}}(\Gamma \Rightarrow_{X \cup \{x\}} \phi)$

$\epsilon_{fd}\ forall\_e(\Gamma \Rightarrow_X \phi\{t/x\}, [\delta])\ =$

$forall\_e\ (\epsilon_{fl}\ (\epsilon_{vs}\ X)\ \Gamma)\ (\epsilon_{vs}\ X)\ (\epsilon_{vn}\ x)(\epsilon_f\ (addvar\_Vst\ (\epsilon_{vn}\ x)(\epsilon_{vs}\ X))\ \phi)\ (\epsilon_t\ (\epsilon_{vs}\ X)\ t)$

$(\epsilon_{wfflp}\ (\epsilon_{vs}\ X)\ \Gamma)\ (\epsilon_{wffp}\ (addvar\_Vst\ (\epsilon_{vn}\ x)(\epsilon_{vs}\ X))\ \phi)\ (\epsilon_{wft}\ (\epsilon_{vs}\ X)\ t)\ (\epsilon_{fd}\ \delta)$

$where\ \delta \in \Delta_{\Pi_{HOL}}(\Gamma \Rightarrow_X \forall x.\phi),$

$\epsilon_{fd}\ exists\_i(\Gamma \Rightarrow_X \exists x.\phi, [\delta])) \ = \ exists\_i$

$(\epsilon_{fl}\ (\epsilon_{vs}\ X)\ \Gamma)\ (\epsilon_{vs}\ X)\ (\epsilon_{vn}\ x)(\epsilon_f\ (addvar\_Vst\ (\epsilon_{vn}\ x)(\epsilon_{vs}\ X))\ \phi)\ (\epsilon_t\ (\epsilon_{vs}\ X)\ t)$

$(\epsilon_{wfflp}\ (\epsilon_{vs}\ X)\ \Gamma)\ (\epsilon_{wffp}\ (addvar\_Vst\ (\epsilon_{vn}\ x)(\epsilon_{vs}\ X))\ \phi)\ (\epsilon_{wftp}\ (\epsilon_{vs}\ X)\ t)\ (\epsilon_{fd}\ \delta)$

$where\ \delta \in \Delta_{\Pi_{FOL}}(\Gamma \Rightarrow_{X \cup \{x\}} \phi)$

$\epsilon_{fd}\ exists\_e(\Gamma \Rightarrow_X \phi', [\delta_1, \delta_2])) \ = \ exists\_e$

$(\epsilon_{fl}\ (\epsilon_{vs}\ X)\ \Gamma)\ (\epsilon_{vs}\ X)\ (\epsilon_{vn}\ x)(\epsilon_f\ (addvar\_Vst\ (\epsilon_{vn}\ x)(\epsilon_{vs}\ X))\ \phi)\ (\epsilon_f\ (\epsilon_{vs}\ X)\ \phi')$

$(\epsilon_{wfflp}\ (\epsilon_{vs}\ X)\ \Gamma)\ (\epsilon_{wffp}\ (addvar\_Vst\ (\epsilon_{vn}\ x)(\epsilon_{vs}\ X))\ \phi)\ (\epsilon_{wffp}\ (\epsilon_{vs}\ X)\ \phi')$

$(\epsilon_{fd}\ \delta_1)\ (\epsilon_{fd}\ \delta_2)$

$where \delta_1 \in \Delta_{\Pi_{FOL}}(\Gamma \cup \{\phi\} \Rightarrow_{X \cup \{x\}} \phi'), \delta_2 \in \Delta_{\Pi_{FOL}}(\Gamma \Rightarrow_X \exists x.\phi)$

Adequacy of the encoding of the proof system is guaranteed by the following theorem:

Theorem 6.26 *There is a bijection between closed derivation trees of a sequent* $\Gamma \Rightarrow_X \phi$ *and the normal forms of the proofs of the inductive relation* $FOL\ \ (\epsilon_{fl}\ \ (\epsilon_{vs}\ \ X)\ \ \Gamma)\ \ (\epsilon_{vs}\ \ X)\ \ (\epsilon_f\ \ (\epsilon_{vs}\ \ X)\ \ \phi)$.

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. One can proof by induction that the encoding function is injective and total. To prove the bijection we define a decoding function with arity

$$\epsilon_{fd}^{-1} : FOL\ env\ vs\ form\ \rightarrow$$

$$\Delta_{\Pi_{FOL}}((\epsilon_{fl}^{-1}\ (\epsilon_{vs}^{-1}\ vs)\ env) \Rightarrow_{\epsilon_{vs}^{-1}\ vs} (\epsilon_f^{-1}\ (\epsilon_{vs}^{-1}\ vs)\ form))$$

for any $env : List\ Formula, vs : Var\_set, form : Formula$ inductively defined as follows:

$\epsilon_{fd}^{-1}\ (impl\_i\ env\ vs\ \phi\ \phi'\ wfenv\ wff\ wff'\ prd)\ =$

$$impl\_i((\epsilon_{fl}^{-1}\ vs\ env)\ \Rightarrow_{\epsilon_{vs}^{-1}\ vs}\ (\epsilon_{f}^{-1}\ vs\ \phi) \supset (\epsilon_{f}^{-1}\ vs\ \phi'), [\epsilon_{fd}^{-1}\ prd])$$

$\epsilon_{fd}^{-1}\ (impl\_e\ env\ vs\ \phi\ \phi'\ wfenv\ wff\ wff'\ prd\ prd')\ =$

$$impl\_e((\epsilon_{fl}^{-1}\ vs\ env) \Rightarrow_{\epsilon_{vs}^{-1}\ vs}\ (\epsilon_{f}^{-1}\ vs\ \phi'), [\epsilon_{fd}^{-1}\ prd, \epsilon_{fd}^{-1}\ prd']$$

$\epsilon_{fd}^{-1}\ (forall\_i\ env\ vs\ vn\ \phi\ wfenv\ wff\ dpr)\ =$

$$forall\_i\ ((\epsilon_{fl}^{-1}\ vs\ env)\ \Rightarrow_{(\epsilon_{vs}^{-1}\ vs)}\ \forall \epsilon_{vn}^{-1}\ vn.(\epsilon_{f}^{-1}\ (addvar\_Vst\ vn\ vs)\ \phi), [\epsilon_{fd}^{-1}\ dpr])$$

$\epsilon_{fd}^{-1}\ (forall\_e\ env\ vs\ vn\ \phi\ t\ wfenv\ wff\ wft\ dpr)\ =$

$$forall\_e\ ((\epsilon_{fl}^{-1}\ vs\ env)\ \Rightarrow_{\epsilon_{vs}^{-1}\ vs}\ (\epsilon_{f}^{-1}\ (addvar\_Vst\ vn\ vs)\ \phi)$$
$$\{(\epsilon_{t}^{-1}\ vs\ t)\ /\ (\epsilon_{vn}^{-1}\ vn)\}, [\epsilon_{fd}^{-1}\ dpr])$$

$\epsilon_{fd}^{-1}\ (exists\_i\ env\ vs\ vn\ \phi\ t\ wfenv\ wff\ wft\ dpr)\ =$

$$exists\_i\ ((\epsilon_{fl}^{-1}\ vs\ env)\ \Rightarrow_{(\epsilon_{vs}^{-1}\ vs)}\ \exists \epsilon_{vn}^{-1}\ vn.(\epsilon_{f}^{-1}\ (addvar\_Vst\ vn\ vs)\ \phi), [\epsilon_{fd}^{-1}\ dpr])$$

$\epsilon_{fd}^{-1}\ (exists\_e\ env\ vs\ vn\ \phi\ \phi'\ wfenv\ wff\ wff'\ dpr\ dpr')\ =$

$$exists\_e\ ((\epsilon_{fl}^{-1}\ vs\ env)\ \Rightarrow_{(\epsilon_{vs}^{-1}\ vs)}\ (\epsilon_{f}^{-1}\ vs\ \phi'), [\epsilon_{fd}^{-1}\ dpr, \epsilon_{fd}^{-1}\ dpr'])$$

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

6.2 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:

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

$$where$$

$$x \notin FV(e')\ \Rightarrow\ x'\ =\ x\ \wedge$$

$$x \in FV(e')\ \Rightarrow\ x' \notin FV(e')\ \wedge\ x' \notin FV(e)\ \wedge\ x' \notin BV(e),$$

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

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 for the case of first-order logic, the encoding of variables is not trivial but in this case additionally to the variable name and its type requires two variable indexes: one to denote the DeBruijn index as in first-order logic and the other to denote the bound level of the variable. Variable names are variable indexes are defined as in first-order logic . Both indexes are assigned during the encoding of terms. But in this case, 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 (the additional variable index associated to variables). 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 either because we always preserve the original names of the variables.

6.2.1 Encoding of variables

The encoding of variable symbols, variable names and variable indexes is the same as in first-order logic. We assume predefined the following functions on variable indexes which can be found in the appendix where a full encoding of a higher-order logic is given:

- $Eqbool\_Vi : Var\_index\ \rightarrow\ Var\_index\ \rightarrow\ Bool$ 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.

- $add\_Vi : Var\_index \to Var\_index \to Var\_index$ which adds two variable indexes like they were naturals.
- $decr\_Vi : Var\_index \to Var\_index \to Var\_index$ which decrements a variable index like it was a natural.
- $substract\_Vi : Var\_index \to Var\_index \to Var\_index$ which substracts two variable indexes like they were naturals.

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

Definition 6.27 *The inductive types Holtype for a given set of base types B is defined by the following set of constructors:*

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

$$\{\, func\_Holt : Holtype \to Holtype \to Holtype\}$$

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

Definition 6.28 *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 6.29 *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.

- $assindex\_Hiv : Holinvar \rightarrow Var\_index \rightarrow Holinvar$ which given a higher-order variable with indexes and a variable index, assigns the variable index as deBruijn index to the variable.

- $assblevel\_Iv : Holinvar \rightarrow Var\_index \rightarrow Holinvar$ which given a higher-order variable with indexes and a variable index, assigns the variable index as bound level to the variable.

- $addindex\_Hiv : Holinvar \rightarrow Var\_index \rightarrow Holinvar$ which given a higher-order variable with indexes and a variable index, adds the variable index with the deBruijn index of the variable

- $addblevel\_Hiv : Holinvar \rightarrow Var\_index \rightarrow Holinvar$ which given a higher-order variable with indexes and a variable index, adds the variable index with the bound level of the variable.

6.2.2 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 6.30 *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 $Holvar\_set$ which can be found in the appendix where a full encoding of a higher-order logic is given:

- $empty\_Hvst : Holvar\_set$ which returns the empty variable set

- $addfvar\_Hvst : Holvar \rightarrow Holvar\_set \rightarrow Holvar\_set$ 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.

- $addbvar\_Hvst : Holvar \rightarrow Holvar\_set \rightarrow Holvar\_set$ 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.

- $getblevel\_Hvst : Holvar\_set \rightarrow Var\_index$ 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)

- $getvar\_Hvst : Holvar \rightarrow Holvar\_set \rightarrow Holinvar$ which given a higher-order variable hv and a variable set hvs, returns the higher-order variable with variable indexes with the greatest deBruijn 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 6.31 *The inductive relation*

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

is defined by the following set of constructors:

$base\_Inhivl : \Pi hv : Holvar.\Pi hiv : Holinvar.\Pi hivl : List\ Holinvar.$

$\Pi eqpr : (Eqbool\_Hvar\ hv\ (fst\ hiv)) =_{bool}\ true.$

$Is\_in\_Hivl\ hv\ (cons\ hiv\ hivl)$

$genc\_Inhivl : \Pi hv : Holvar.\Pi hiv : Holinvar.\Pi hivl : list\ Holinvar.$

$\Pi pr : Is\_in\_Hivl\ hv\ hivl.$

$Is\_in\_Hivl\ hv\ (cons\ Holinvar\ hiv\ hivl)$

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

Definition 6.32 *The inductive relation*

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

is defined by the following set of constructors:

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

$genc\_Ninhivl : \Pi hv : Holvar.\Pi hiv : Holinvar.\Pi hivl : list\ Holinvar.$

$\Pi eqpr : (Eqbool\_Hvar\ hv\ (fst\ hiv)) =_{bool}\ false.$

$\Pi pr : Notisin\_Hivl\ hv\ hivl.$

$Notisin\_Hivl\ hv\ (cons\ Holinvar\ hiv\ hivl)$

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 6.33 *The inductive relation*

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

is defined by the following set of constructors:

$$ctr\_Inbhvs : \Pi hv : Holvar.\Pi hvs : Holvar\_set.$$

$$\Pi isinpr : Is\_in\_hivl\ hv\ (snd\ (snd\ hvs)).Isin\_boundv\_Hvs\ hv\ hvs$$

Definition 6.34 *The inductive relation*

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

is defined by the following set of constructors:

$$ctr\_Ninbhvs : \Pi hv : Holvar.\Pi hvs : Holvar\_set.$$

$$\Pi isinpr : Notisin\_hivl\ hv\ (snd\ (snd\ hvs)).Notisin\_boundv\_Hvs\ hv\ hvs$$

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

Definition 6.35 *The inductive relation*

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

is defined by the following set of constructors:

$$ctr\_Inbhvs : \Pi hv : Holvar.\Pi hvs : Holvar\_set.$$

$$\Pi isinpr : Is\_in\_hivl\ hv\ (snd\ (fst\ hvs)).Isin\_freev\_Hvs\ hv\ hvs$$

6.2.3 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 6.36 *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 6.37 *The function*

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

are defined as follows:

$subst\_Htrm\ vi\ htrm\ htrm'\ hiv\ =$

$\quad Prim\_rec\ Holterm\ (holvarc\ vi\ htrm'\ hiv)\ (abstrc\ htrm'\ hiv)$

$\quad\quad (applc\ htrm'\ hiv)\ htrm$

$\quad where$

$\quad holvarc\ vi\ htrm'\ hiv\ hiv'\ =$

$\quad\quad Primrec\ Bool\ (update\_index\_Htrm\ vi\ (getblevel\_Hiv\ hiv')\ htrm')$

$\quad\quad\quad (holvar\_Htrm\ hiv')\ (Eqbool\_Hivar\ hiv\ hiv')$

$\quad abstrc\ htrm'\ hiv\ hiv'\ htrm\ htrmf\ =\ (abstr\_Htrm\ hiv'\ htrmf)$

$\quad applc\ htrm'\ hiv\ htrm\ htrm''\ htrmf\ htrmf''\ =$

$\quad\quad appl\_Htrm\ htrmf\ htrmf''$

Definition 6.38 *The function* $update\_index\_Htrm : Var\_index\ \rightarrow\ Var\_index$

$\rightarrow Holterm \ \rightarrow Holterm$ *is defined as follows:*

$$update\_index\_Htrm\ vi\ bl\ htrm\ =\ Primrec\ Holterm$$

$$(holvarc\ vi\ bl)\ (abstrc\ bl)\ applc\ htrm$$

$$where$$

$$holvarc\ vi\ bl\ hiv\ =\ Primrec\ bool\ (addblevel\_Hiv\ bl\ hiv)$$

$$(addblevel\_Hiv\ bl\ (addindex\_Hiv\ bl\ hiv))$$

$$Ltbool\_Vi\ (getindex\_Hiv\ hiv)\ vi)$$

$$abstrc\ bl\ hiv\ ht\ htf\ =$$

$$abstr\_Htrm\ (addblevel\_Hivl\ bl\ (addindex\_Hivl\ bl\ hiv))\ htf$$

$$applc\ ht\ ht'\ htf\ htf'\ =\ appl\_Htrm\ htf\ htf'$$

6.2.4 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 6.39 *The inductive relation*

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

is defined by the following set of constructors:

$$\{ass1\_tr : \Pi vs : Holvar\_set. \Pi hv : Holvar.$$

$$\Pi pr : Notisin\_boundv\_Hvs\ hv\ vs. \Pi prin : Isin\_freev\_Hvs\ hv\ vs.$$

$$Wfhterm\ \ vs\ \ (holvar\_Htrm\ \ (getvar\_Hvst\ hv\ vs))\ \ (snd\ hv)\} \cup$$

$$\{ass2\_tr : \Pi vs : Holvar\_set. \Pi hv : Holvar. \Pi pr : Isin\_boundv\_Hvs\ hv\ vs.$$

$$Wfhterm\ \ vs\ \ (holvar\_Htrm\ (getvar\_Hvst\ hv\ vs))\ \ (snd\ hv)\} \cup$$

$$\{abs\_tr : \Pi vs : Holvar\_set. \Pi hv : Holvar. \Pi ht : Holterm. \Pi hty : Holtype.$$

$$\Pi wft : Wfhterm\ (addbvar\_Hvst\ hv\ vs)\ ht\ hty.$$

$$Wfhterm\ vs\ (abstr\_Htrm\ (getvar\_Hvst\ hv$$

$$(addbvar\_Hvst\ hv\ vs))\ ht)\ (func\_Holt\ (snd\ hv)\ hty),$$

$$appl\_tr : \Pi vs : Holvar\_set. \Pi ht, ht' : Holterm. \Pi hty, hty' : Holtype$$

$$\Pi prt : Wfhterm\ vs\ ht\ (func\_Holt\ hty\ hty').$$

$$\Pi prt' : Wfhterm\ vs\ ht'\ hty.$$

$$Wfhterm\ vs\ (appl\_Ht\ ht\ ht')\ hty'\}$$

One can easily define encoding and decoding functions of types, variable names, list of variables, variable sets and higher-order terms. See the appendix of a full encoding of a higher-order logic for the definitions of similar functions. These functions have the following arities:

$$\epsilon_\tau : Types(B) \rightarrow Holtype$$

$$\epsilon_\tau^{-1} : Holtype \rightarrow Types(B)$$

$$\epsilon_{vn} : X \rightarrow Var\_name$$

$$\epsilon_{vn}^{-1} : Var\_name \rightarrow X$$

$$\epsilon_{hvl} : [(X, Types(B))] \rightarrow (List\ Holvar)$$

$$\epsilon_{hvl}^{-1} : (List\ Holvar) \rightarrow [(X, Types(B))]$$

$$\epsilon_{vs} : ([X], [X]) \rightarrow (Holvar\_set)$$

$$\epsilon_{vs}^{-1} : (Holvar\_set) \rightarrow ([X], [X])$$

$$\epsilon_{ht} : Holvar\_set \rightarrow Term(X) \rightarrow Holterm$$

$$\epsilon_{ht}^{-1} : Holvar\_set \rightarrow Holterm \rightarrow Term(X)$$

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

$$\epsilon_{td}\ Ass1((X, X') \blacktriangleright x : \tau) =$$

$$ass\_tr(\epsilon_{vs}\ (X, X'))\ encx$$

$$(ctr\_Ninbhvs\ encx\ (\epsilon_{vs}\ (X, X'))$$

$$(genc\_Ninhivl\ encx\ (getvar\_Hvst\ enchv'_n$$

$$(\epsilon_{vs}\ [hv'_1, \ldots, hv'_n]))$$

$$(snd\ (snd\ (\epsilon_{vs}\ [hv'_1, \ldots, hv'_{n-1}])))$$

$$\lambda P : bool \rightarrow Prop.\lambda pr : P\ false.pr$$

$$(\ldots (genc\_Ninhivl\ encx\ (getvar\_Hvst\ \ enchv'_n\ \epsilon_{vs}\ [])$$

$$(snd\ (snd\ (\epsilon_{vs}\ [])))$$

$$(base\_Ninhivl\ encx))\ldots)))$$

$$(ctr\_Infhvs\ encx\ (\epsilon_{vs}\ (X, X'))$$

$$(genc\_Inhivl\ encx\ (getvar\_Hvst\ enchv_n$$

$$(\epsilon_{vs}\ [hv_1, \ldots, hv_{i-1}, (x, \tau), hv_i, \ldots, hv_n]))$$

$$(snd\ (fst\ (\epsilon_{vs}\ [hv_1, \ldots, hv_{i-1}, (x, \tau), hv_i, \ldots, hv_{n-1}])))$$

$$(\ldots (genc\_Inhivl\ encx\ (getvar\_Hvst\ enchv_i$$

$$(\epsilon_{vs}\ [hv_1, \ldots, hv_{i-1}, (x, \tau), hv_i]))$$

$$(snd\ (fst\ (\epsilon_{vs}\ [hv_1, \ldots, hv_{i-1}, (x, \tau)])))$$

$$(base\_Hivs\ encx\ (getvar\_Hvst\ encx$$

$$(\epsilon_{vs}\ [hv_1, \ldots, hv_{i-1}, (x, \tau)]))$$

$$(snd\ (fst\ (\epsilon_{vs}\ [hv_1, \ldots, hv_{i-1}, (x, \tau)])))$$

$$(\lambda P : bool \rightarrow Prop.\lambda pr : P\ true.pr)))\ldots)))$$

$$where\ X = [hv_1, \ldots, hv_{i-1}, (x, \tau), hv_i, \ldots, hv_n]$$

$$encx\ =\ mkpair\ Holvar\ (\epsilon_{vn}\ x)\ (\epsilon_\tau\ \tau)$$

$$enchv_n\ =\ mkpair\ Holvar\ (fst\ hv_n)\ (snd\ hv_n)$$

$$enchv_i\ =\ mkpair\ Holvar\ (fst\ hv_i)\ (snd\ hv_i) \qquad X' = [hv'_1, \ldots, hv'_n]$$

$$enchv'_n\ =\ mkpair\ Holvar\ (fst\ hv'_n)\ (snd\ hv'_n)$$

$$\vdots$$

$$enchv'_1\ =\ mkpair\ Holvar\ (fst\ hv'_1)\ (snd\ hv'_1)$$

$$\epsilon_{td}\ Ass2((X, X') \blacktriangleright x : \tau) =$$

$$\quad ass2\_tr(\epsilon_{vs}\ (X, X'))\ encx$$

$$\quad\quad (ctr\_Inbhvs\ encx\ (\epsilon_{vs}\ (X, X'))$$

$$\quad\quad\quad (genc\_Inhivl\ encx\ (getvar\_Hvst\ enchv'_n$$

$$\quad\quad\quad\quad (\epsilon_{vs}\ [hv'_1, \ldots, hv'_{i-1}, (x, \tau), hv'_i, \ldots, hv'_n]))$$

$$\quad\quad\quad (snd\ (snd\ (\epsilon_{vs}\ [hv'_1, \ldots, hv'_{i-1}, (x, \tau), hv'_i, \ldots, hv'_{n-1}])))$$

$$\quad\quad\quad (\ldots (genc\_Inhivl\ encx\ (getvar\_Hvst\ enchv'_i$$

$$\quad\quad\quad\quad (\epsilon_{vs}\ [hv'_1, \ldots, hv'_{i-1}, (x, \tau), hv'_i]))$$

$$\quad\quad\quad (snd\ (snd\ (\epsilon_{vs}\ [hv'_1, \ldots, hv'_{i-1}, (x, \tau)])))$$

$$\quad\quad\quad (base\_Hivs\ encx\ (getvar\_Hvst\ encx$$

$$\quad\quad\quad\quad (\epsilon_{vs}\ [hv'_1, \ldots, hv'_{i-1}, (x, \tau)]))$$

$$\quad\quad\quad (snd\ (snd\ (\epsilon_{vs}\ [hv'_1, \ldots, hv'_{i-1}, (x, \tau)])))$$

$$\quad\quad\quad (\lambda P : bool \to Prop.\lambda pr : P\ true.pr))) \ldots)))$$

$$where\ X' = [hv'_1, \ldots, hv'_{i-1}, (x, \tau), hv'_i, \ldots, hv'_n]$$

$$\quad encx\ =\ mkpair\ Holvar\ (\epsilon_{vn}\ x)\ (\epsilon_\tau\ \tau)$$

$$\quad enchv'_n\ =\ mkpair\ Holvar\ (fst\ hv'_n)\ (snd\ hv'_n)$$

$$\quad enchv'_i\ =\ mkpair\ Holvar\ (fst\ hv'_i)\ (snd\ hv'_i)$$

$$\epsilon_{td}\ Abs((X,X') \blacktriangleright \lambda x:\tau.e : \tau \to \tau', [\delta]) =$$

$$abs\_tr\ (\epsilon_{vs}\ (X,X'))\ (\epsilon_{vn}\ x, \epsilon_\tau\ \tau)$$

$$(\epsilon_{ht}\ (addbvar\_Hvst\ (\epsilon_{vn}\ x, \epsilon_\tau\ \tau)(\epsilon_{vs}\ (X,X')))\ e)$$

$$(\epsilon_{td}\ \delta)$$

$$where$$

$$\delta \in \Delta_{\Pi_{TLC}}((X,X') \cup \{x:\tau\} \blacktriangleright e)$$

$$\epsilon_{td}\ Appl((X,X') \blacktriangleright e\ e' : \tau', [\delta,\delta']) =$$

$$appl\_tr\ (\epsilon_{vs}\ (X,X'))\ (\epsilon_{ht}\ (\epsilon_{vs}\ (X,X'))\ e)$$

$$(\epsilon_{ht}\ (\epsilon_{vs}\ (X,X'))\ e')\ (\epsilon_\tau\ \tau)\ (\epsilon_\tau\ \tau')$$

$$(\epsilon_{td}\ \delta)\ (\epsilon_{td}\delta')$$

$$where$$

$$\delta' \in \Delta_{\Pi_{TLC}}((X,X') \blacktriangleright e : \tau),$$

$$\delta \in \Delta_{\Pi_{TLC}}((X,X') \blacktriangleright e : \tau \to \tau')$$

6.2.5 Adequacy of the representation

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

Theorem 6.40 *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

$$\epsilon_{td}^{-1} : Wfhterm\ (\epsilon_{vs}\ (X,[]))\ (\epsilon_{ht}\ (\epsilon_{vs}\ (X,[]))\ e)\ (\epsilon_\tau\ \tau) \to$$

$$\Delta_{\Pi_{TLC}}((X,[]) \ \blacktriangleright\ e : \tau)$$

inductively defined as follows:

$$\epsilon_{td}^{-1}\ (ass1\_tr\ vs\ hv\ pr\ prin)\ =$$

$$ASS1((\epsilon_{vs}^{-1}\ vs) \blacktriangleright (\epsilon_{vn}^{-1}\ (fst\ hv) : (\epsilon_{\tau}^{-1}\ (snd\ hv)))$$

$$\epsilon_{td}^{-1}\ (ass2\_tr\ vs\ hv\ pr)\ =$$

$$ASS2((\epsilon_{vs}^{-1}\ vs) \blacktriangleright (\epsilon_{vn}^{-1}\ (fst\ hv) : (\epsilon_{\tau}^{-1}\ (snd\ hv)))$$

$$\epsilon_{td}^{-1}\ (abs\_tr\ vs\ hv\ ht\ hty\ dpr) =$$

$$\lambda ABS((\epsilon_{vs}^{-1}\ vs) \blacktriangleright \lambda(\epsilon_{vn}^{-1}\ (fst\ hv) : \epsilon_{\tau}^{-1}\ (snd\ hv)).$$

$$(\epsilon_{ht}^{-1}\ (addbvar\_Hvst\ hv\ vs)\ ht) :$$

$$((fst\ hv) \to hty), [(\epsilon_{td}^{-1}(dpr))])$$

$$\epsilon_{td}^{-1}(appl\_tr\ vs\ ht\ ht'\ hty\ hty'\ wftpr\ wftpr') =$$

$$APPL\ (\epsilon_{vs}^{-1}\ vs)\ \blacktriangleright\ (\epsilon_{ht}^{-1}\ vs\ ht)\ (\epsilon_{ht}^{-1}\ vs\ ht') : hty',$$

$$[(\epsilon_{td}^{-1}(wftpr)), (\epsilon_{td}^{-1}\ wftpr')]$$

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

6.3 Adequate encoding of a fragment of a linear type system

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 [Hof99]. 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, this is possible in our framework and not in LF since we are able to define non-standard contexts and manipulate them because we do not have to identify the variables of the object logic with the variables of LF. 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 [Hof99] and we do not detail it here because it is not necessary for our purposes. The fragment of SLR which we are going to encode adequately in UTT is the following:

Definition 6.41 *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 6.42 *The type expressions which we will consider are the following:*

$\mathcal{T}_{SLR}$ ::=	N	*natural numbers*
	$L(\mathcal{T}_{SLR})$	*lists over* $\mathcal{T}_{SLR}$
	$T(\mathcal{T}_{SLR})$	*binary trees labelled over* $\mathcal{T}_{SLR}$
	$\mathcal{T}_{SLR} \overset{a}{\to} \mathcal{T}_{SLR}$	*function space of aspect* a.

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

$$\mathcal{T}_{SLR} \overset{a}{\to} \mathcal{T}_{SLR} \textit{ is } \mathcal{T}_{SLR} \multimap \mathcal{T}_{SLR} \textit{ when } a = \{linear, nonmodal\}$$

$$\mathcal{T}_{SLR} \overset{a}{\to} \mathcal{T}_{SLR} \textit{ is } \mathcal{T}_{SLR} \to \mathcal{T}_{SLR} \textit{ when } a = \{nonlinear, nonmodal\}$$

$$\mathcal{T}_{SLR} \overset{a}{\to} \mathcal{T}_{SLR} \textit{ is } \Box\mathcal{T}_{SLR} \to \mathcal{T}_{SLR} \textit{ when } a = \{nonlinear, modal\}$$

Definition 6.43 *The expressions which we will consider are the following:*

Λ_{SLR} ::=	x	*(variable)*
	$\Lambda_{SLR}\,\Lambda_{SLR}$	*(application)*
	$\lambda x : \mathcal{T}_{SLR}.\Lambda_{SLR}$	*(abstraction)*

Definition 6.44 *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 6.45 *The judgement $\Gamma \ <: \ a$ for any context Γ and any aspect a is inductively defined by the following rules:*

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

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

Definition 6.46 *The judgement Disjoint Γ Δ for any context Γ, Δ is inductively defined by the following rules:*

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

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

Definition 6.47 *The judgement Γ 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 6.48 *The functional fragment of the type system SLR is inductively defined by the following rules:*

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

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

And now we proceed with the encoding of the type theory. To give the encoding and proof of adequacy of this type type theory, we first give the representation of aspects, the order relation between aspects, types, variables, contexts and terms with some predefined operations. Then, we encode the type theory and finally we give the adequacy of the representation.

6.3.1 Encoding of variables, contexts and terms

First, we respresent aspects and their relation operation.

Definition 6.49 *The inductive type Linearity is defined by the following constructors:*

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

Definition 6.50 *The inductive relation* $<: \_Lin : Linearity \rightarrow Linearity \rightarrow Prop$ *is inductively defined by the following constructor:*

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

Definition 6.51 *The inductive type Modality is defined by the following constructors:*

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

Definition 6.52 *The inductive relation* $<: \_Mod : Modality \rightarrow Modality \rightarrow Prop$ *is inductively defined by the following constructor:*

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

Definition 6.53 *The type SLRaspect is defined as Pair Linearity Modality.*

Definition 6.54 *The inductive relation* $<: \_Asp : SLRaspect \rightarrow SLRaspect \rightarrow Prop$ *is defined by the following set of constructors:*

$refl : \Pi l : Linearity.\Pi m : Modality.$

$$<: \_Asp\ (mkpair\ SLRaspect\ l\ m)\ (mkpair\ SLRaspect\ l\ m)$$

$compw :: \Pi l, l' : Linearity.\Pi m, m' : Modality.\Pi linr :<: \_lin\ l\ l'.\Pi modr :<: \_mod\ m\ m'.$

$$<: \_Asp\ (mkpair\ SLRaspect\ l\ m)\ (mkpair\ SLRaspect\ l'\ m')$$

Next, we define the types of the type theory.

Definition 6.55 *The inductive type SLRtype is defined by the following set of constructors:*

$$nat\ :\ SLRtype$$

$$list\ :\ SLRtype \rightarrow SLRtype$$

$$tree\ :\ SLRtype \rightarrow SLRtype$$

$$lmfunc\ :\ SLRtype \rightarrow SLRaspect \rightarrow SLRtype$$

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

Definition 6.56 *The type* $SLRvar$ *is defined as* $Pair\ (Pair\ Varname\ SLRtype)\ SLRaspect$

Definition 6.57 *The function* $getaspect\_SLRv : SLRvar \rightarrow SLRaspect$ *is defined as follows:*

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

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

Definition 6.58 *The inductive relation* $Nonlin\_Asp : SLRaspect \rightarrow Prop$ *is defined by the following constructor:*

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

Definition 6.59 *The type* $SLRivar$ *is defined as* $Pair\ SLRvar\ Varindex$*.*

Definition 6.60 *The type* $SLRcontext$ *is defined as* $Pair\ Varindex\ (List\ SLRivar)$

Definition 6.61 *The inductive type* $SLRterm$ *is defined by the following constructors:*

$$var\_SLRt : SLRivar \rightarrow SLRterm$$

$$appl\_SLRt : SLRterm \rightarrow SLRterm \rightarrow SLRterm$$

$$abs\_SLRt : SLRivar \rightarrow SLRterm \rightarrow SLRterm$$

We assume predefined the following functions and inductive relations of $SLRvar$, $SLRivar$, $SLRcontext$ and $SLRcontext$ which are defined in a very similar way as the equivalent operations of Var, $Invarn$, Var_set and $Formula$ in first-order logic:

$$Eqbool\_SLRv\ :\ SLRvar \rightarrow SLRvar \rightarrow Bool$$

$$Eqbool\_SLRiv\ :\ SLRivar \rightarrow SLRivar \rightarrow Bool$$

$$empty\_SLRctxt\ :\ SLRcontext$$

$$addvar\_SLRctxt\ :\ SLRvar \rightarrow SLRcontext \rightarrow SLRcontext$$

$$getvar\_SLRctxt\ :\ Varname \rightarrow SLRcontext \rightarrow SLRivar$$

$$concat\_SLRctxt\ :\ SLRcontext \rightarrow SLRcontext \rightarrow SLRcontext$$

$$Is\_in\_SLRctxt\ :\ Varname \rightarrow SLRcontext \rightarrow Prop$$

$$Not\_is\_in\_SLRctxt\ :\ Varname \rightarrow SLRcontext \rightarrow Prop$$

We also need the following representations of the judgements $\Gamma \ <: \ a$, $Disjoint\ \Gamma\ \Delta$ and $\Gamma\ nonlinear$ used in the definition of this type theory.

Definition 6.62 *The inductive relation* $<:\_Ctxt : SLRcontext \ \rightarrow\ SLRaspect\ \rightarrow$ *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 6.63 *The inductive relation* $Nonlinear\_Ctxt : SLRcontext \ \rightarrow$ *Prop is defined by the following constructors:*

$$bcnl : Nonlinear\_Ctxt\ empty\_SLRctxt$$

$$gcnl : \Pi slrc : SLRcontext.\Pi slrv : SLRvar.\Pi nlpr : Nonlinear\_Ctxt\ slrc.$$

$$\Pi ninpr : \neg\ Not\_is\_in\_SLRctxt\ (fst\ (fst\ slrv))\ (snd\ slrc).$$

$$\Pi nlapr : Nonlin\_Asp\ (getaspect\_SLRv\ slrv).$$

$$Nonlinear\_Ctxt\ (addvar\_SLRctxt\ slrv\ slrc)$$

Definition 6.64 *The inductive relation* $Disjoint\_Ctxt : SLRcontext \ \rightarrow\ SLRcontext\ \rightarrow$ *Prop is defined by the following constructors:*

$$bcdisj : \Pi slrc : SLRcontext.Disjoint\_Ctxt\ empty\_SLRctxt\ slrc$$

$$gcdisj : \Pi slrc, slrc' : SLRcontext.\Pi slrv : SLRvar.$$

$$\Pi ninpr : \neg\ Not\_is\_in\_SLRctxt\ (fst\ (fst\ slrv))\ slrc.$$

$$\Pi ninpr : \neg\ Not\_is\_in\_SLRctxt\ (fst\ (fst\ slrv))\ slrc'.$$

$$\Pi disjpr : Disjoint\_Ctxt\ slrc\ slrc'.$$

$$Disjoint\_Ctxt\ (addvar\_SLRctxt\ slrv\ slrc)\ slrc'$$

6.3.2 Encoding of the type theory

We will also assume predefined the following encoding and decoding functions which are very similar to the ones of first-order logic:

$$\epsilon_a : Aspect \;\rightarrow\; SLRaspect$$

$$\epsilon_a^{-1} : SLRaspect \;\rightarrow\; Aspect$$

$$\epsilon_{\mathcal{T}SLR} : \mathcal{T}_{SLR} \;\rightarrow\; SLRtype$$

$$\epsilon_{\mathcal{T}SLR}^{-1} : SLRtype \;\rightarrow\; \mathcal{T}_{SLR}$$

$$\epsilon_{ctxtSLR} : [Binding] \;\rightarrow\; SLRcontext$$

$$\epsilon_{ctxtSLR}^{-1} : SLRcontext \;\rightarrow\; [Binding]$$

$$\epsilon_{tSLR} : SLRcontext \;\rightarrow\; \Lambda_{SLR} \;\rightarrow\; SLRterm$$

$\epsilon_{tSLR}^{-1} : SLRcontext \;\rightarrow\; SLRterm \;\rightarrow\; \Lambda_{SLR}$

$\epsilon_{Djctxt} : (\Delta_{\Pi_{Disjoint}}(Disjoint\ \Delta_1\ \Delta_2)) \;\rightarrow$

$(Disjoint\_Ctxt\ (\epsilon_{ctxtSLR}\ \Delta_1)\ (\epsilon_{ctxtSLR}\ \Delta_2))$

for any $\Delta_1, \Delta_2 \in [Binding]$

$\epsilon_{Djctxt}^{-1} : (Disjoint\_Ctxt\ sclr\ sclr') \;\rightarrow$

$\Delta_{\Pi_{Disjoint}}(Disjoint\ (\epsilon_{ctxtSLR}^{-1}\ sclr)\ (\epsilon_{ctxtSLR}^{-1}\ sclr'))$

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 $a \in Aspect$,$\Gamma \in [Binding]$

$$\epsilon^{-1}_{<:ctxt} : (<: \_Ctxt\ slrc\ a) \rightarrow \Delta_{\Pi_{<:ctxt}}(\epsilon^{-1}_{ctxtSLR}\ slrc) <: (\epsilon^{-1}_{a}\ a)$$

for any $s : SLRaspect$,$slrc : SLRcontext$

where *Binding* are triples of type $(X, Aspect, T_{SLR})$. See the full encoding of higher-order logic for similar encoding and decoding functions for similar structures.

And finally, we give the representation of the proof system of the fragment of the linear type theory, its encoding functions and its proof of adequacy.

Definition 6.65 *The inductive relation*

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

is defined by the following constructors:

$Tarri : \Pi slrc : SLRcontext.\Pi slrv : SLRvar.\Pi t : SLRterm.$

$\Pi pr : SLRts\ (addvar\_SLRctxt\ slrv\ slrc)\ t\ (snd\ (fst\ slrv)).$

$SLRts\ slrc\ (abs\_SLRt\ (getvar\_SLRctxt\ (fst\ (fst\ slrv))\ (addvar\_SLRctxt\ slrv\ slrc))\ t)$

$Tarre : \Pi slrc, slrc', slrc'' : SLRcontext.\Pi a : SLRaspect.\Pi t, t' : SLRterm.\Pi A, B : SLRtype.$

$\Pi dpr : Disjoint\_Ctxt\ slrc\ slrc'.\Pi dpr' : Disjoint\_Ctxt\ slrc\ slrc''.$

$\Pi dpr'' : Disjoint\_Ctxt\ slrc'\ slrc''.$

$\Pi td : SLRts\ (concat\_SLRctxt\ slrc'\ slrc)\ t\ (lmfunc\ A\ a\ B).$

$\Pi td' : SLRts\ (concat\_SLRctxt\ slrc''\ slrc)\ t'\ A.$

$\Pi nlpr : Nonlinear\_Ctxt\ slrc.\Pi rprc :<: \_Ctxt\ (concat\_SLRctxt\ slrc\ slrc')\ a.$

$SLRts\ (concat\_SLRctxt\ slrc''\ (concat\_SLRctxt\ slrc''\ slrc))\ (appl\_SLRt\ t\ t')\ B$

where Binding are triples of type $(X, Aspect, T_{SLR})$.

The representation of the type system is by the following inductive relation:

Definition 6.66 *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:

$\epsilon_{slrtd}\ (Tarri(\Gamma\ \vdash\ \lambda x : A.e\ :\ A \overset{a}{\rightarrow} B), [\delta])\ =\ Tarri\ (\epsilon_{ctxtSLR}\, \Gamma)\ encx$

$(\epsilon_{tSLR}\ (addvar\_SLRctxt\ encx\ (\epsilon_{ctxtSLR}\, \Gamma))\ e)\ (\epsilon_{slrtd}\ \delta)$

where

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

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

$\epsilon_{slrtd}\ Tarre(\Gamma, \Delta_1, \Delta_2\ \vdash\ e_1\, e_2\ : B, [\delta_1, \delta_2, \delta_3, \delta_4, \delta_5, \delta_6, \delta_7])\ =$

$tarre\ (\epsilon_{ctxtSLR}\, \Gamma)\ (\epsilon_{ctxtSLR}\, \Delta_1)\ (\epsilon_{ctxtSLR}\, \Delta_2)\ (\epsilon_a\ a)$

$(\epsilon_{tSLR}\ (\epsilon_{ctxtSLR}\, \Gamma, \Delta_1)\ e_1)$

$(\epsilon_{tSLR}\ (\epsilon_{ctxtSLR}\, \Gamma, \Delta_1)\ e_2)$

$(\epsilon_{\tau SLR}\ A)\ (\epsilon_{\tau SLR}\ B)$

$(\epsilon_{Djctxt}\ \delta_1)\ (\epsilon_{Djctxt}\ \delta_2)\ (\epsilon_{Djctxt}\ \delta_3)$

$(\epsilon_{SLR}\ \delta_4)\ (\epsilon_{SLR}\ \delta_5)\ (\epsilon_{nlctxt}\ \delta_6)\ (\epsilon_{<:ctxt}\ \delta_7)$

where

$\delta_1 \in \Delta_{\Pi_{Djctxt}}(Disjoint\ \Gamma\ \Delta_1), \delta_2 \in \Delta_{\Pi_{Djctxt}}(Disjoint\ \Gamma\ \Delta_2),$
$\delta_3 \in \Delta_{\Pi_{Djctxt}}(Disjoint\ \Delta_1\ \Delta_2), \delta_4 \in \Delta_{\Pi_{SLR}}\ (\Gamma, \Delta_1\ \vdash\ e_1\ :\ A \overset{a}{\rightarrow} B),$

$\delta_5 \in \Delta_{\Pi_{SLR}}\ (\Gamma, \Delta_2\ \vdash\ e_2\ :\ A),$
$\delta_6 \in \Delta_{\Pi_{nlctxt}}(Nonlinear\ \Gamma), \delta_7 \in \Delta_{\Pi_{<:ctxt}}(\Gamma, \Delta_2\ <:\ a)$

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

Theorem 6.67 *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:

To prove the bijection we define a decoding function with type

$$\epsilon^{-1}_{slrtd} : (SLRts\ slrc\ t\ \tau)\ \rightarrow$$

$$(\Delta_{\Pi_{SLR}}\ (\epsilon^{-1}_{ctxtSLR}\ slrc)\ (\epsilon^{-1}_{tSLR}\ (\epsilon^{-1}_{ctxtSLR}\ slrc)\ t)\ (\epsilon^{-1}_{\tau SLR}\ \tau))$$

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

$\epsilon^{-1}_{slrtd}\ (Tarri\ slrc\ slrv\ t\ pr)\ =$

$Tarri((\epsilon^{-1}_{ctxtSLR}\ slrc)\ \vdash\ \lambda\ (\epsilon^{-1}_{vn}\ (fst\ (fst\ slrv))\ :\ (\epsilon^{-1}_{\tau SLR}\ (snd\ (fst\ slrv))).$

$(\epsilon^{-1}_{tSLR}\ (addvar\_SLRctx\ slrv\ slrc)\ t), [\epsilon^{-1}_{slrtd}\ pr])$

$\epsilon^{-1}_{slrtd}\ (Tarre\ slrc\ slrc'\ slrc''\ a\ t\ t'\ A\ B\ dpr\ dpr'\ dpr''\ td\ td'\ nlpr\ rpr)\ =$

$Tarre(\ (\epsilon^{-1}_{ctxtSLR}\ (concat\ (\epsilon^{-1}_{ctxtSLR} slrc'')\ (concat(\epsilon^{-1}_{ctxtSLR} slrc')\ (\epsilon^{-1}_{ctxtSLR}\ slrc)))\ \vdash$

$(\epsilon^{-1}_{tSLR}\ (concat\_ctxtSLR\ slrc'\ slrc)\ t)$

$(\epsilon^{-1}_{tSLR}\ (concat\_ctxtSLR\ slrc''\ slrc)\ t')\ :$

$(\epsilon^{-1}_{\tau SLR}\ B),$

$[\epsilon^{-1}_{Djctxt}\ dpr, \epsilon^{-1}_{Djctxt}\ dpr', \epsilon^{-1}_{Djctxt}\ dpr, \epsilon^{-1}_{SLR}\ td,$

$\epsilon^{-1}_{SLR}\ td', \epsilon^{-1}_{nlctxt}\ nlpr, \epsilon^{-1}_{<:ctxt}\ rpr])$

and the rest of the proof follows in the same way as the first-order case.

References

[BCH] Michel Bidoit, María Victoria Cengarle, and Rolf Hennicker. Proof systems for structured specifications and their refinements. Chapter 11 of the book Algebraic Foundations of Systems Specification.

[BH95] Michel Bidoit and Rolf Hennicker. Behavioural theories and the proof of behavioural properties. Report LIENS-95-5, Ecole Normale Supérieure, 1995.

[BM91] Rod Burstall and James McKinna. Deliverables: An approach to program development in the calculus of constructions. *ECS-LFCS-91-133*, January 1991.

[Gar92] Philippa Gardner. *Representing Logics in Type Theory.* PhD thesis, University of Edinburgh, July 1992.

[Hen97] Rolf Hennicker. *Structured Specifications with Behavioural Operators: Semantics, Proof Methods and Applications.* Habilitationsschrift, Institut für Informatik, Ludwig-Maximilians-Universität München, June 1997.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[Hof99] Martin Hoffman. *Type systems for polynomial-time computation.* Habilitation thesis, University of Darmstadt, 1999.

[HS96] Martin Hofmann and Donald Sannella. On behavioural abstraction and behavioural satisfaction in higher-order logic. *Theoretical Computer Science*, 167:3–45, 1996.

[Kha97] Saif Ullah Khan. *Machine Assisted Proofs for Generic Semantics to Compiler Transformation Correctness Theorems.* PhD thesis, University of Edinburgh, 1997.

[Kle98] Thomas Kleymann. *Hoare logic and VDM:Machine-Checked Soundness and Completeness Proofs.* PhD thesis, University of Edinburgh, 1998.

[Luo94] Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science.* Clarendon Press Oxford, 1994.

[Mah95] Savitri Maharaj. *A Type-Theoretic Analysis of Modular Specifications.* PhD thesis, University of Edinburgh, 1995.

[McK92] James Hugh McKinna. *Deliverables: A Categorical Approach to Program Development in Type Theory.* PhD thesis, University of Edinburgh, November 1992.