Chapter2:Type theory

1 Introduction

Several formalisms have been used as metalanguages to define other formalisms, which are normally referred as object formalisms. Two main principles to guide the design of a metalanguage are expressibility and decidability issues. One should be able to express several different object formalisms in a natural way preferably, and the metalanguage should have some decidable properties such as type checking to offer machine assistance.

Some object formalisms which have been encoded in different metalanguages are for example type theories, semantics of programming languages or proof systems for logics. Two different metalanguages which have been used to encode logics are LF [HHP93] and higher-order logic (used as a core formalism of several generic theorem provers like for example Isabelle [Pau]).

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

UTT (Uniform Theory of dependent types ([Luo94],[Gog94])) is a type theory which adds to the Extended Calculus of Constructions the possibility to define inductive types. The whole type theory is encoded in the Martin-Löf Logical Framework [BNS90]. A more refined view of UTT can differentiate two different universes:

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

UTT has been used in several ways for the specification and development of software and it was originally designed as a framework for the development of functional programs from modular algebraic-like specifications.

As we mentioned in the previous chapter, the main drawbacks of the resulting framework are lack of generality, expressibility at the computational level and additionally it is difficult to relate with the algebraic frameworks by which it has been inspired. In a first attempt to improve this framework, we present general techniques to encode adequately different proof systems for deduction and refinement of algebraic specifications which are sound and in some cases complete with respect to the sound and complete but infinitary proof systems presented in [BHW95] and more exhaustively in [Hen97]. We represent some of these proof systems with first-order or higher-order logic as specification logic. The implementation is presented with the same level of formality as the encodings of logics in LF but using the more expressible metalanguage *UTT* with a similar expressibility as the metalanguages of theorem provers like *HOL* or *Isabelle*.

The main construction which we will use for our encodings are inductive relations. In this type theory, inductive relations can be seen as functional types which given some arguments of the appropriate type, return the proposition one has to prove to guarantee that the tuple formed by the given arguments belongs to the relation. We will use these constructions both for the encoding of formulae and the encoding of proof systems.

In the rest of the chapter, first we will formally present LF showing with an example how it has been used as metalanguage to encode logics and then we will present the extended calculus (ECC) and UTT. ECC is an extended version by Luo in order to use this new type theory as an object language for software design. UTT is an extension of ECC with inductive types defined in the Martin Löf logical framework. We will also present in the section of ECC and UTT the notation that we will use to define basic types and functions, inductive types and inductive relations in UTT, which will be used in the next chapter and chapter 6. Finally, the meta-theoretic properties of both type theories will be summarised. In the next chapter we will present how this type theory has been previously used in software design with special emphasis in the algebraic approach and how to define adequate encodings of logical systems, and in chapter 6 we will present fully adequate encodings of proof systems for deduction and refinement of *ASL* specifications with higher-order logic as specification logic.

2 LF

As we mentioned in the introduction, the type theory of LF can be seen as a typed lambda calculus with dependent types. The type theory was chosen as weak as possible in order to get efficient decidability features. It is quite limited from a computational point of view since not even the primitive recursive functions are representable in the type theory and from a logical point of view, it has the expressive power of intuitionistic first-order logic. Even with these limitations it is possible to make adequate encodings of different logics including higher-order and modal logics using its higher-order features. Bounded quantification is encoded as λ-abstraction and therefore substitution can be implemented as β-reduction. Judgements and rules of the logic are represented as dependent functional types

In this section, first we will explain the basic rules of dependent types of LF and then we will present how to develop adequate encodings with a fragment of first-order logic.

Dependent functional types are defined by a type formation rule and introduction and elimination rules which determine the inhabitants of the dependent functional types: abstraction and application.

Whereas the formation rule of dependent functional types can vary between type theories, the introduction and elimination rules are always the same.

For the case of the formation rules of a dependent functional type of the form $\Pi x : A.B$ the formation rules vary depending on the universes of the given type theory which the types A, B and $\Pi x : A.B$ inhabit.

In the case of one formation rule of dependent functional types of LF, all these types inhabit the basic universe of the type theory which is denoted as $Type$.

The formation rules of these dependent types is as follows:

$$\frac{\Gamma \vdash A : Type \quad \Gamma, x : A \vdash B : Type}{\Gamma \vdash \Pi x : A.B : Type}$$

and the introduction and elimination rules are of the following form:

$$\frac{\Gamma, x : A \vdash_\Sigma M : B}{\Gamma \vdash \lambda x : A.M : \Pi x : A.B} \quad (\lambda)$$

$$\frac{\Gamma \vdash_\Sigma M : \Pi x : A.B \quad \Gamma \vdash_\Sigma N : B}{\Gamma \vdash_\Sigma M\ N : B\{N/x\}} \quad (app)$$

where as we can observe, the type of the application $M\ N$ depends on the term N and is the result of the substitution of the free variable x by N in the type B.

Dependent functional types are the main families of types of this type theory, but appart from families of types and their associated objects, this type theory has got contexts to represent the variables of the type theory, signatures to represent all the components of a logic (terms, formulae, judgements and rules) and kinds to classify the different families of types. See the appendix on LF for the full definition of the type theory

The proof systems which are encoded in LF are usually formulated as natural deduction systems. See [Gar92] for a formal description of these systems. Basically, these systems are defined by a finite set of natural deduction rules. These kind of rules are defined by a set of n premises, a conclusion, and side conditions are allowed. *Premises* and *conclusions* are defined by sequents with schematic variables and therefore a rule denotes in general a set of (n+1)-tuples of sequents. An *instance* of a rule is a (n+1)-tuple of sequents of this set. In general, sequents are defined with judgements and for example the only judgement which is used to define first-order logic is $\phi \ \ true$ which means that the formula ϕ is derivable. In the definition of first-order logic below we will denote this judgement just by ϕ.

The sequent to define natural deduction systems in [Gar92] is $\Gamma \Rightarrow_X J$ where Γ is a set of judgements (normally referred as environment), J is a judgement and X a finite set of variables.

Some of the structural rules which natural deduction systems with the previous sequent usually satisfy are the following:

$$\frac{}{\Gamma \cup \{J\} \Rightarrow_X J} \quad (ASS) \qquad \frac{\Gamma' \Rightarrow_X J}{\Gamma \Rightarrow_X J}\ \Gamma' \subseteq \Gamma \quad (MON)$$

$$\frac{\Gamma \Rightarrow_X J \quad \Delta \cup J \Rightarrow_X J'}{\Gamma \cup \Delta \Rightarrow_X J'} \quad (CUT)$$

As an example, we present the encoding in LF of the following fragment of first-order logic:

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

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

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

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

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

Terms and formulae are represented in LF by the following constants:

$$i : \mathit{Type}$$

$$o : \mathit{Type}$$

Terms are formed over a signature, and for the signature of natural numbers with just the operations $0, succ, +$, the inhabitants of i are defined by the following constants:

$$zero\ :\ i$$

$$succ\ :\ i \to i$$

$$sum\ :\ i \to i \to i$$

To encode formulae with the operators $=, \supset, \forall, \exists$, we need to define the following constants:

$$equal\ :\ i \to i \to o$$

$$implies\ :\ o \to o \to o$$

$$forall\ :\ (i \to o) \to o$$

$$exists\ :\ (i \to o) \to i$$

Thus, the following terms are inhabitants of i and o assuming that we have the context $\Gamma\ =\ \{x : i, y : i\}$ and the signature FOL contains the constants

defined above:

$$\Gamma \vdash_{FOL} sum\ zero\ (succ\ x)\ :\ i$$

$$\Gamma \vdash_{FOL} equal\ (sum\ zero\ (succ\ x))\ y\ :\ o$$

$$\Gamma \vdash_{FOL} forall\ \lambda\ x : i.(exists\ \lambda\ y : i.\ (equal\ x\ y))\ :\ o$$

To represent the rules of a natural deduction system, first we have to define its judgements as types of LF and then the rules by LF constants which inhabit those types parameterizing the schematic variables which appear in the rule.

Since for the case of first-order logic we just have the judgement $\phi\ true$, the only type which we have to add to the signature FOL is the following:

$$true : o \rightarrow Type$$

And the constants which we have to add to FOL to encode the rules of first-order logic defined above are:

$$implies\_i\ :\ \Pi\ \phi, \psi : o.((true\ \phi)\ \rightarrow\ (true\ \psi)) \rightarrow (true\ (implies\ \phi\ \psi))$$

$$implies\_e\ :\ \Pi\ \phi, \psi : o.(true\ (implies\ \phi\ \psi)) \rightarrow (true\ \phi) \rightarrow (true\ \psi)$$

$$forall\_i\ :\ \Pi\ \Phi : i \rightarrow o.(\Pi x : i.true\ (\Phi\ x)) \rightarrow (true\ (Forall\ \Phi))$$

$$forall\_e\ :\ \Pi\ \Phi : i \rightarrow o.\Pi t : i.(true\ (forall\ \Phi)) \rightarrow (true\ (\Phi\ t))$$

$$exists\_i\ :\ \Pi\ \Phi : i \rightarrow o.\Pi t : i.(true\ (\Phi\ t)) \rightarrow (true\ (exists\ \Phi))$$

$$exists\_e\ :\ \Pi\ \Phi : i \rightarrow o.\Pi\ \psi : o.(true\ (exists\ \Phi)) \rightarrow$$

$$((\Pi x : i.true\ (\Phi\ x))\ \rightarrow (true\ \psi)) \rightarrow (true\ \psi))$$

The encoding of terms and formulae is defined by the functions

$$\epsilon_{T,X}\ :\ \mathcal{T}(X)\ \rightarrow\ i_{\Gamma_X}$$

$$\epsilon_{F,X}\ :\mathcal{F}(X)\ \rightarrow\ o_{\Gamma_X}$$

where X is a finite sequence of free variables $X\ =\ \{x_1, \ldots, x_n\}$, $\mathcal{T}(X)$ and $\mathcal{F}(X)$ denote the set of terms and formulae generated by the signature of natural numbers, Γ_X denote the context $\Gamma_X\ =\ \{x'_1 : i, \ldots, x'_n : i\}$ and i_{Γ_X} and o_{Γ_X} denote any inhabitant of $i : Type$ and $o : Type$ generated by the signature FOL with context Γ. The function $\epsilon_{T,X}\ :\ \mathcal{T}(X)\ \rightarrow\ i_{\Gamma_X}$ is inductively defined as follows:

$$\epsilon_{T,X}\ x\ =\ x'$$

$$\epsilon_{T,X}\ (succ\ t)\ =\ succ\ (\epsilon_{T,X}\ t)$$

$$\epsilon_{T,X}\ (t_1\ +\ t_2)\ =\ (sum\ (\epsilon_{T,X}\ t_1)\ (\epsilon_{T,X}\ t_2))$$

and the function $\epsilon_{F,X} : \mathcal{F}(X) \rightarrow o_{\Gamma_X}$ is inductively defined as follows:

$$\epsilon_{F,X}\ (t\ =\ r)\ =\ (equal\ (\epsilon_{T,X}\ t)\ (\epsilon_{T,X}\ r))$$

$$\epsilon_{F,X}\ (\phi\ \supset\ \psi)\ =\ implies\ (\epsilon_{F,X}\ \phi)\ (\epsilon_{F,X}\ \psi)$$

$$\epsilon_{F,X}\ (\forall x.\phi)\ =\ forall\ \lambda x' : i.(\epsilon_{F,X \cup \{x\}}\ \phi)$$

$$\epsilon_{F,X}\ (\exists x.\phi)\ =\ exists\ \lambda x' : i.(\epsilon_{F,X \cup \{x\}}\ \phi)$$

And to prove the adequacy of the presentation, we need to prove the following theorems the proofs of which can be found in [HHP93]:

Theorem 2.1 *For any finite sequence of variables X, there exists a bijection between $\mathcal{T}(X)$ and the β-normal forms of the inhabitants of i_{Γ_X}, and $\mathcal{F}(X)$ and the β-normal forms of the inhabitants of o_{Γ_X}. The encoding functions of terms and formulae commutes with the substitution operation for any finite sequence of variables X, for any terms $t, t \in \mathcal{T}(X)$, for any formula $\phi \in \mathcal{F}(X)$ as follows:*

$$\epsilon_{T,X}\ t\ \{\ t'\ /\ x\ \}\ =\ (\epsilon_{T,X}\ t)\ \{\ (\epsilon_{T,X}\ t')\ /\ x\ \}$$

$$\epsilon_{F,X}\ \ \phi\ \{\ t'\ /\ x\ \}\ =\ (\epsilon_{T,X}\ \phi)\ \{\ (\epsilon_{T,X}\ t')\ /\ x\ \}$$

Theorem 2.2 *For any finite sequence of variables X, for any sequence of assumptions $\Delta\ = [\phi_1, \ldots, \phi_n]$ and formula ϕ closed in X, the formula ϕ is derivable in the proof system FOL if and only if there exists an inhabitant of*

$$true\ (\epsilon_{F,X}\ \phi)$$

in the context

$$\Gamma_X, a_1 : true\ (\epsilon_{F,X}\ \phi_1), \ldots, a_n : true\ (\epsilon_{F,X}\ \phi_n)$$

We are going to define a new principle of encoding for natural deduction systems in UTT because it solves the following limitations of the principle of encoding of LF.

First, in LF it is not possible to develop metatheory of the encoded logics. Since we will encode logics as inductive relations, we will be able to have inductive principles to formalize metatheory. Terms and formulae will be encoded as inductive types for the same reason and in such a way that functions on terms and formulae (and specially substitution) will be easily encodable using primitive recursion.

Second, in LF the consequence relation of the object logic must be intuitionistic because the object logic inherits the basic meta-properties of the type theory. See for example [Gar92] for a more formal explanation. Since we will also encode variables and contexts of the sequent as inductive types (as a parameter of the inductive relation which encodes the logic), the properties of the

type theory like for example weakening are not inherited by the encoded logic. These properties have to be proven for the concrete object logic using for example the induction principles associated to the inductive relation which encodes the object logic.

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

As a conclusion, we believe that from a practical point of view the increase of the expressivity of the type theory is rellevant and on the other hand, it does not affect significatively the efficiency of its associated proof checkers. Note that most of their decidability features have been developed generically for pure type systems. See [Pol95] for details.

As we mentioned in the introduction, in the next sections we will formally present the type theory UTT and we leave for the end of the next chapter the formal explanation on how to encode natural deduction systems in UTT.

3 ECC and UTT

The calculus of constructions ([CH88]) is the most expressive type theory of the λ-cube [Bar92]. From a logical point of view, it can be seen as an intuitionistic higher-order logic in which Leibniz equality and several logical operators can be encoded such as conjunction, disjunction and an existential operator. From a computational point of view, different inductive types can be encoded provided that their constructors satisfy some syntactic constraints, and primitive recursion functions on these inductive types can also be encoded. Luo proposes an extension of this type theory (the Extended Calculus of Constructions (ECC) which is more suitable for the development of functional programs from modular specifications. See [Luo94] for an explanation of the design decisions and next chapter for how to use this type theory with these purposes.

In ECC and UTT, there exist an impredicative universe $Prop$ and a predicative hierarchy of universes $Type_i$ $i \in \omega$. There are two formation rules of dependent functional types: one for the impredicative universe and the other for the predicative hierarchy. They are defined as follows:

$$\frac{\Gamma, x : A \vdash P : Prop}{\Gamma \vdash \Pi x : A : P : Prop} \quad (\Pi_1)$$

$$\frac{\Gamma \vdash A : Type_j \quad \Gamma, x : A \vdash B : Type_j}{\Gamma \vdash \Pi x : A : B : Type_j} \quad (\Pi_2)$$

The impredicative one allows quantification over the universe of propositions. This means that the term of the form $\Pi x : Prop.x$ inhabits the universe $Prop$, and this gives a great expressive power to the type theory because in this impredicative universe a higher-order intuitionistic logic resides. See the appendix on ECC for the full definition of the logical operators of this logic which is also possible to perform in the calculus of constructitions. Note that in the predicative hierarchy of universes this is not the case, since we are forced to quantify over types which are inhabitants of the universe $Type_i$, but not the universe $Type_i$ itself. In this predicative hierarchy, all the computational types of the type theory reside: dependent functional types, dependent product types and in UTT, inductive types also.

The introduction and elimination rules of dependent functional types are as in LF and dependent product types has a similar formation rule as the predicative formation rule of dependent functional types:

$$\frac{\Gamma \vdash A : Type_j \quad \Gamma, x : A \vdash B : Type_j}{\Gamma \vdash \Sigma x : A.B : Type_j} \quad (\Sigma)$$

The inhabitants of dependent product types are pairs and projections determined by the following introduction and elimination rules

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B\{M / x\} \quad \Gamma, x : A \vdash B : Type_j}{\Gamma \vdash < M, N >_{\Sigma x:A.B} : \Sigma x : A.B} \quad (pair)$$

$$\frac{\Gamma \vdash M : \Sigma x : A.B}{\Gamma \vdash \pi_1(M) : A} \quad (\pi_1) \qquad \frac{\Gamma \vdash M : \Sigma x : A.B}{\Gamma \vdash \pi_2(M) : B\{\pi_1(M) / x\}} \quad (\pi_2)$$

UTT extends ECC with inductive types. Inductive types can be defined in this type theory by a set of constructors. They have to satisfy the usual positive syntactic restrictions in order to be definable in the type theory. In [Gog94], in [Luo94] and in the appendix of this thesis for UTT one can find the formal definition of UTT including the schema which inductive types have to follow in order to be representable in UTT. These formal definitions are given in the Martin Löf logical framework, a metalanguage for defining type theories.

In the following we present some examples on how to define inductive types and inductive relations using the notation which we will use in the following chapters. We will also explicit the induction principles and the computation rules associated to inductive definitions following a similar notation. In the rest of the thesis we will use most of the examples presented below and for the new definitions we will normally assume predefined the induction principles and computation rules associated to inductive definitions by a set of constructors. See the appendixes of UTT for these examples using the notation of [Luo94] and [Gog94] and for a list of predefined functions for some of the inductive types presented here using a similar notation to the one used in this chapter.

- **Booleans:**

The inductive type $Bool : Type_0$ is defined by the following set of constructors:

$$true : Bool$$

$$false : Bool$$

The induction principle $Ind(Bool)$ which we will use to reason about propositions of type $Bool \to Prop$ is the following:

$$\Pi P : Bool \ \to \ Prop.(P\ true) \ \supset \ (P\ false) \supset (\forall b : Bool.P\ b)$$

and the primitive recursion principle $Primrec\ Bool$ with arity

$$Primrec\ Bool \ : T \to \ T \ \to \ Bool \ \to \ T$$

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

$$Primrec\ Bool\ bct\ bcf\ true \ \to \ bct$$

$$Primrec\ Bool\ bct\ bcf\ false \ \to \ bcf$$

- **Naturals:**

  The inductive type $Nat : Type_0$ is defined by the following set of constructors:

  $$zero : Nat$$

  $$succ : Nat \to Nat$$

  The induction principle $Ind(Nat)$ which we will use to reason about propositions of type $Nat \to Prop$ is the following:

  $$\Pi P : Nat \ \to \ Prop.(P\ zero) \ \supset \ (\forall n : Nat.(P\ n) \ \supset \ (P\ (succ\ n))) \supset (\forall n : Nat.P\ n)$$

  and the primitive recursion principle $Primrec\ Nat$ with arity

  $$Primrec\ Nat \ : T \to \ (Nat \ \to \ T \ \to \ T) \ \to \ Nat \ \to \ T$$

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

  $$Primrec\ Nat\ bcn\ gcn\ zero \ \to \ bcn$$

  $$Primrec\ Nat\ bcn\ gcn\ (succ\ n) \ \to \ gcn\ n\ (Primrec\ Nat\ bcn\ gcn\ n)$$

- **Pairs of elements:**

  The inductive type $Pair : Type_0 \to Type_0 \to Type_0$ is defined by the following constructor:

  $$mkpair : \Pi A : Type_0.\Pi B : Type_0.A \ \to \ B \ \to (Pair\ A\ B)$$

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

$$\Pi P : (Pair\ A\ B)\ \rightarrow\ Prop.\forall a : A.\forall b : B.(P\ (mkpair\ A\ B\ a\ b)) \supset (\forall p : Pair\ A\ B.P\ p)$$

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

$$Primrec\ (Pair\ A\ B)\ : (A\ \rightarrow\ B \rightarrow\ T)\ \rightarrow\ (Pair\ A\ B)\ \rightarrow\ T$$

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

$$Primrec\ (Pair\ A\ B)\ bcp\ (mkpair\ A\ B\ a\ b)\ \rightarrow\ (bcp\ a\ b)$$

Notation: In some cases we will denote the type $Pair\ A\ B$ by the infix operator $A\ \times\ B$ for any types $A, B : Type_0$, and normailly we will denote the expression $mkpair\ A\ B\ a\ b$ just by (a, b) or we will omit the types A and B of the expression.

- **List of elements:**

  The inductive type $List : Type_0 \rightarrow Type_0$ is defined by the following set of constructors:

  $$nil : \Pi A : Type_0.List\ A$$

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

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

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

  $$(\forall l : List\ A.P\ l)$$

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

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

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

  $$Primrec\ (List\ A)\ bcl\ gcl\ (nil\ A)\ \rightarrow\ bcl$$

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

As an example of inductive relations we give an inductive relation which could be used to encode the following fragment of propositional logic:

$$\frac{\Gamma \Rightarrow \phi_1 \wedge \phi_2}{\Gamma \Rightarrow \phi_1}\quad (\wedge El) \qquad\qquad \frac{\Gamma \Rightarrow \phi_1 \wedge \phi_2}{\Gamma \Rightarrow \phi_2}\quad (\wedge Er)$$

$$\frac{\Gamma \Rightarrow \phi_1 \quad \Gamma \Rightarrow \phi_2}{\Gamma \Rightarrow \phi_1 \wedge \phi_2} \qquad (\wedge I)$$

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

Definition 3.1 *Assume that the inductive type Propos is defined by just the following two constructors:*

$$and : Propos \rightarrow Propos \rightarrow Propos$$

$$implies : Propos \rightarrow Propos \rightarrow Propos$$

The inductive relation $PC : (List\ Propos) \rightarrow Propos \rightarrow Prop$ *is defined by the following constructors:*

$andle$: $\Pi env : List\ Propos.\Pi\phi_1 : Propos.\Pi\phi_2 : Propos.$

$\Pi pr : PC\ env\ (and\ \phi_1\ \phi_2).PC\ env\ \phi_1$

$andre$: $\Pi env : List\ Propos.\Pi\phi_1 : Propos.\Pi\phi_2 : Propos.$

$\Pi pr : PC\ env\ (and\ \phi_1\ \phi_2).PC\ env\ \phi_2$

$andi$: $\Pi env : List\ Propos.\Pi\phi : Propos.$

$\Pi pr_1 : PC\ env\ \phi_1.\Pi pr_2 : PC\ env\ \phi_2.PC\ env\ (and\ \phi_1\ \phi_2)$

$impli$: $\Pi env : List\ Propos.\Pi\phi_1 : Propos.\Pi\phi_2 : Propos.$

$\Pi pr_1 : PC\ (cons\ Propos\ env\ \phi_1)\ \phi_2.PC\ env\ (implies\phi_1\ \phi_2)$

$imple$: $\Pi env : List\ Propos.\Pi\phi_1 : Propos.\Pi\phi_2 : Propos.$

$\Pi pr_1 : PC\ env\ (implies\ \phi_1\ \phi_2).\Pi pr_2 : PC\ env\ \phi_1.PC\ env\ \phi_2$

and the induction principle $Ind(PC)$ *to reason about propositions of type* P :

$(List\ Propos) \rightarrow Propos \rightarrow Prop$ *is defined as follows:*

$\Pi P : (List\ Propos) \rightarrow Propos \rightarrow Propos.$

$\Pi andle\ : \Pi env : List\ Propos.\Pi\phi_1 : Propos.\Pi\phi_2 : Propos.$

$\Pi pr : PC\ env\ (and\ \phi_1\ \phi_2).\Pi pr' : P\ env\ (and\ \phi_1\ \phi_2).P\ env\ \phi_1.$

$\Pi andre\ : \Pi env : List\ Propos.\Pi\phi_1 : Propos.\Pi\phi_2 : Propos.$

$\Pi pr : PC\ env\ (and\ \phi_1\ \phi_2).\Pi pr : P\ env\ (and\ \phi_1\ \phi_2).P\ env\ \phi_2.$

$\Pi andi\ : \Pi env : List\ Propos.\Pi\phi : Propos.$

$\Pi pr_1 : PC\ env\ \phi_1.\Pi pr_2 : PC\ env\ \phi_2.\Pi pr'_1 : P\ env\ \phi_1.\Pi pr'_2 : P\ env\ \phi_2.$

$P\ env\ (and\ \phi_1\ \phi_2).$

$\Pi impli\ : \Pi env : List\ Propos.\Pi\phi_1 : Propos.\Pi\phi_2 : Propos.$

$\Pi pr_1 : PC\ (cons\ Propos\ env\ \phi_1)\ \phi_2\ .\Pi pr'_1 : P\ (cons\ Propos\ env\ \phi_1)\ \phi_2.$

$P\ env\ (implies\ \phi_1\ \phi_2).$

$\Pi\ imple\ : \Pi env : List\ Propos.\Pi\phi_1 : Propos.\Pi\phi_2 : Propos.$

$\Pi pr_1 : PC\ env\ (implies\ \phi_1\ \phi_2).\Pi pr_2 : PC\ env\ \phi_1.$

$\Pi pr_1 : P\ env\ (implies\ \phi_1\ \phi_2).\Pi pr_2 : P\ env\ \phi_1.$

$P\ env\ \phi_2.$

$\Pi env : List\ Propos.\Pi\phi : Propos.P\ env\ \phi$

Finally, we present an example of mutually recursive inductive types. The example is the type which is used for the encoding of the higher-order types of the higher-order specification logic which we will define in the chapter of the semantics of ASL. The definition is as follows:

Definition 3.2 *The mutually recursive inductive types Holtype and Holtype_list for a given signature* Σ *are defined by the following set of construc-*

tors:

$$\{\, s\_Holt : Holtype \mid s \in Sorts(\Sigma)\,\} \cup$$

$$\{\, prop\_Holt : Holtype,$$

$$holrel\_Holt : Holtype\_list \to Holtype\ ,$$

$$nil\_Holt : Holtype\_list$$

$$cons\_Holt : Holtype \to Holtype\_list \to Holtype\_list\}$$

The induction principle associated to Holtype is defined as follows:

$$\forall\ P : Holtype\ \to\ Prop.\forall\ P' : Holtype\_list\ \to\ Prop.$$

$$(P\ s_1\_Holt)\ \supset\ \ldots\ \supset\ (P\ s_n\_Holt)\ \supset\ (P\ prop\_Holt)\ \supset$$

$$(\forall\ htl : List\ Holtype.(P'\ (holrel\_Holt\ htl))\ \supset\ (P\ (holrel\_Holt\ htl)))$$

$$\supset\ (P\ nil\_Holt)\ \supset\ (\forall ht : Holtype.\forall htl : Holtype\_list.$$

$$(P\ ht)\ \supset\ (P'\ htl)\ \supset\ (P'\ (cons\_Holt\ ht\ htl)))\ \supset$$

$$(\forall ht : Holtype.P\ ht)$$

and the induction principle associated to Holtype_list is defined as follows:

$$\forall\ P : Holtype\ \to\ Prop.\forall\ P' : Holtype\_list\ \to\ Prop.$$

$$(P\ s_1\_Holt)\ \supset\ \ldots\ \supset\ (P\ s_n\_Holt)\ \supset\ (P\ prop\_Holt)$$

$$\supset\ (\forall\ htl : List\ Holtype.(P'\ (holrel\_Holt\ htl))\ \supset\ (P\ (holrel\_Holt\ htl)))$$

$$\supset\ (P\ nil\_Holt)\ \supset\ (\forall ht : Holtype.\forall htl : Holtype\_list.$$

$$(P\ ht)\ \supset\ (P'\ htl)\ \supset\ (P'\ (cons\_Holt\ ht\ htl)))\ \supset$$

$$(\forall htl : Holtype\_list.P'\ htl)$$

Note that the induction premises of both induction principles (one for each constructror of both inductive types) are both universally quantified by propositions on Holtype and propositions on Holtype_list. The induction principles just differ on the conclusions.

The arity of the primitive recursive operators for any type $T, T' : Type_0$ are

the following :

$$Primrec\ Holtype : T \supset \ldots \supset T \supset T \supset (Holtype\_list \to T' \to T) \to T \to$$
$$(Holtype \to Holtype\_list \to T \to T' \to T') \to Holtype \to T$$

$$Primrec\ Holtype\_list : T \supset \ldots \supset T \supset T \supset (Holtype\_list \to T' \to T) \to T \to$$
$$(Holtype \to Holtype\_list \to T \to T' \to T') \to Holtype\_list \to T'$$

and the computational rules associated to the type Holtype and the type Holtype_list are the following:

$$\{\ Primrec\ Holtype\ s1c\ \ldots\ snc\ propc\ holrelc\ nilhtlc\ conshtlc\ s_i\_Holt \to sic \mid$$
$$s_i \in Sorts(\Sigma)\ \} \cup$$

$$\{Primrec\ Holtype\ s1c\ \ldots\ snc\ propc\ holrelc\ nilhtlc\ conshtlc\ prop\_Holt \to propc,$$

$$Primrec\ Holtype\ s1c\ \ldots\ snc\ propc\ holrelc\ nilhtlc\ conshtlc\ (holrel\_Holt\ htl) \to$$
$$holrelc\ htl\ (Primrec\ (List\ Holtype)\ s1c\ \ldots\ snc\ propc\ holrelc\ nilhtrlc\ conshtlc\ htl),$$

$$Primrec\ (List\ Holtype)\ s1c\ \ldots\ snc\ propc\ holrelc\ nilhtlc\ conshtlc\ nil\_Holt \to nilhtlc,$$

$$Primrec\ (List\ Holtype)\ s1c\ \ldots\ snc\ propc\ holrelc\ nilhtlc\ conshtlc\ (cons\_Holt\ ht\ htl) \to$$
$$conshtlc\ ht\ htl\ (Primrec\ Holtype\ s1c\ \ldots\ snc\ propc\ holrelc\ nilhtlc\ conshtlc\ ht)$$
$$(Primrec\ (List\ Holtype)\ s1c\ \ldots\ snc\ propc\ holrelc\ nilhtlc\ conshtlc\ htl)$$
$$\}$$

4 Metatheory and decidability features of the previous type theory

The metatheory and decidability features of LF, ECC and UTT are quite similar. See [HHP93] for the metatheory of LF and [Bar92] for a generic presentation of the metatheory of pure type systems and how it can be instantiated to determine the metatheory of LF. In [Luo94] and [Gog94] you can find the metatheory of ECC and UTT respectively and in this section we will summarize the metatheory of ECC which is very similar to UTT although the latter

is formulated in a more complicated way using typed operational semantics and the Martin Löf logical framework to handle with inductive types. The main difference with respect to the the metatheory of pure type systems is the property of uniqueness of types. This property is more complicated to formulate because of the predicative hierarchy of types of ECC:

Theorem 4.1 Thinning lemma:

For any valid contexts Γ, Δ *such that* $\Gamma \subseteq \Delta$, *for any terms* A, B, *if* $\Gamma \vdash A : B$ *then* $\Delta \vdash A : B$

Theorem 4.2 Substitution lemma:
For any context Γ, Δ, *for any term* A, B, C, *if* $\Gamma, x : A, \Delta \vdash B : C$ *and* $\Gamma \vdash D : A$ *then*

$$\Gamma, \Delta\{ D / x\} \vdash B \{ D / x\} : C \{ D / x\}$$

Theorem 4.3 Church Rosser theorem:
For any terms M_1, M_2 *such that* $M_1 \equiv M_2$ *there exists a term* M *such that* M_1 *and* M_2 *reduces to* M *by a finite sequence of reductions.*

Theorem 4.4 Type reflection:
For any context Γ *and terms* M, A, *if* $\Gamma \vdash M : A$ *then* $\Gamma \vdash A : U$ *for some universe* U.

Definition 4.5 Principal types:
A *is a principal type of a term* M *in the context* Γ *iff:*

- $\Gamma \vdash M : A$.
- *For any term* A', $\Gamma \vdash M : A'$ *if and only if* $A \leq A'$ *and* $\Gamma \vdash A' : U$ *for some universe* U.

Theorem 4.6 Uniqueness of principal types:

For any context Γ *and for any terms* M, A *such that* $\Gamma \vdash M : A$ *there exists* A' *such that* A' *is a principal type of* M.

Theorem 4.7 Subject reduction:
For any context Γ *and for any terms* A, A', B, *if* $\Gamma \vdash A : B$ *and* $A \equiv A'$ *then* $\Gamma \vdash A' : B$

Theorem 4.8 Strong normalisation:
For any context Γ *and for any terms* A, B *such that* $\Gamma \vdash A : B$, *there exists no infinite sequence of reductions starting from* A *and* B.

Theorem 4.9 Decidability of type checking and type inference:

- **Type checking:** *For any context* Γ *and terms* A, B *it is decidable whether* $\Gamma \vdash A : B$ *holds.*
- **Type inference:** *For any context* Γ *and terms* A *it is decidable whether* $\Gamma \vdash A : B$ *holds for some term* B*.*

References

[Bar92] H.P. Barendregt. *Lambda Calculi with Types. In Abramsky, Gabbai, and Maibaum editors, Handbook of Logic in Computer Science*, volume II. Oxford University Press, 1992.

[BHW95] Michel Bidoit, Rolf Hennicker, and Martin Wirsing. Behavioural and abstractor specifications. *Science of Computer Programming*, 25(2-3):149–186, December 1995.

[BNS90] Kent Petersson Bengt Nordström and Jan Smith. *Programming in Martin-Löf's Type Theory: An Introduction.* Oxford University Press, 1990.

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

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

[Gog94] Healfdene Goguen. *A Typed Operational Semantics for Type Theory.* PhD thesis, University of Edinburgh, September 1994. Also published as Technical Report CST-110-94, Department of Computer Science.

[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.

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

[Pau] Lawrence C. Paulson. Introduction to isabelle. 25 October 1998 (Computer laboratory of University of Cambridge).

[Pol95] Robert Pollack. *The Theory of LEGO.* PhD thesis, University of Edinburgh, April 1995.