Adequate encodings of proof systems for algebraic specifications in UTT

Nikos Mylonakis
nikos@dcs.ed.ac.uk

Department of Computer Science (University of Edinburgh)
King's Buildings
Mayfield Road
Edinburgh EH9 3JZ
fax: 44-(0)31-667 72 09

Abstract. In this paper, we give an adequate encoding of a sound proof system with respect to the sound and complete proof system for algebraic specifications with structuring mechanisms and observability operators presented in [4]. We use a similar principle of encoding to the one used in LF ([3]), but we use a different type theory. We use the Uniform Theory of Dependent Types (UTT [2], [6]).

1 Introduction

Algebraic specifications are formal specifications of properties which programs have to satisfy. There have been developed many specification languages with the aim to make easier the task of formal specification. We will concentrate on specification languages based on specification operators which allow to build larger specifications from smaller ones. More precisely, we will work with ASL-like specification languages [11].

Type theories were initially used as a logical language for the foundations of mathematics. Since they also include a computational language (in particular a functional language), most of them have also been used as a framework for program development. Some expressive type theories have also been used as logical frameworks like for example the LF type theory [3].

There have been several attempts to design proof systems for algebraic specification languages to develop formal proofs of the form $SP \models \phi$ where SP can in general be a specification expression. Assuming the existence of a model-theoretic semantics of the specification language by which a class of models is associated to any specification expressible in the language, an informal reading of $SP \models \phi$ is as a proposition which is true if and only if all models of the specification SP satisfy the formula ϕ.

One way of defining the proof system is as a single set of rules which roughly speaking state how to derive a proof in a specification in terms of proofs in subspecifications. See [10] to get an idea of these kind of proof systems.

Recently there has been developed a sound and complete proof system for this kind of specification languages with structuring and observability operators

[4]. The proof system is inductively defined by specification expressions, adding to the proof system of the flat specification, different axioms and infinitary rules for each specification expression. Thus, the result is an infinitary proof system and it is well known that it is not possible to give a complete representation of it in a finitary proof system (like for example a type-theoretical framework).

The final goal of this work is to give an adequate encoding of a sound proof system with respect to the sound and complete one presented in [4], in the sense that the proof system in [4] is an extension of the one which we propose. We use a similar principle of encoding to the one used in LF, but not its underlying type theory. Instead we use the Uniform Theory of dependent Types (UTT [6], [2]). Previous works relating algebraic specifications and this type theory can be found in [5] and [8]. In both of them, the proof system of the type theory is used to reason about algebraic specifications. Since this proof system is not sound with respect to the semantics of the kind of specification languages we are treating, we use the type theory as a metalanguage to encode sound proof systems for our language.

In UTT one can define inductive relations, and they 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 syntax and the encoding of proofs of our logical system. See [9] as another work on the use of this type theory as metalanguage.

Using this feature of UTT, a more precise representation of the proof system in the type theory is possible because one can express the least relation generated by a set of constructors, one can define primitive recursive operations associated to the different components of the logic (like for example substitution, renaming, etc) and it is also possible to develop metatheory about the encoding of the object proof system.

Additionally, this type theory has also a decidable type checking (as in LF), and we have an implementation of it in the generic tool Lego [7]. Therefore, we believe that UTT is a good choice as the metalanguage to encode proof systems for algebraic specifications.

2 Proof systems for algebraic specification languages

As we mentioned in the introduction, the proof system which we want to encode is inductively defined by the specification expressions of an ASL-like specification language. We will focus on the proof system associated to a specification constructed with a reachability operator from a basic specification with a signature with at least one sort nat and two operations $zero : nat$ and $succ : nat \rightarrow nat$.This operator restricts the possible values of the carrier set associated to the sort nat of any model of the base specification, to those which are reachable by the ground terms generated by the operations $zero$ and $succ$.This constraint is achieved by adding to the proof system for basic specifications a reachability rule. Because of lack of space, we will not treat the more complicated cases, such as the proof system associated to the observational operators of the language.We just mention that in [8] the problem is treated partially, giving semantics to one of the observational operators using UTT as metalanguage.

The proof system will be formulated as a natural deduction system. See [1] for a formal description of these systems. Basically, these systems are defined by a finite set of rules. *Rules* are defined by a name, 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 an (n+1)-tuple of sequents belonging to this set, and in our case a *sequent* has the form $\Gamma \Rightarrow_X \phi$ where Γ is a set of assumptions (which we will refer as environment), ϕ a first order formula and X a finite set of variables. In next section, we will denote the set of environments by $Environment$, the set of first order formulae by $Formula$, and the set of sets of variables as Var_set. The proof system of the specification expression described above, would have apart from classical rules of first order logic a specific reachability rule *reachn*, which in this case is equivalent to the induction principle on natural numbers. This rule is included just in the proof system for the specification described below and has the form,

$$\frac{\begin{array}{l}\Gamma \Rightarrow_X \phi[zero/x] \\ \Gamma \Rightarrow_{X \cup \{x\}} (\phi \to \phi[succ \quad x/x])\end{array}}{\Gamma \Rightarrow_X \forall x.\phi}$$

Now we make more explicit some definitions which differ a little from [1]. For space reasons, we don't give the formal definitions.

Definition 2.1. A first order formula is closed under a finite set of variables X if every free variable of the formula is in X.

Definition 2.2. A sequent $\Gamma \Rightarrow_X \phi$ is closed iff ϕ and all the formulae in Γ are closed under X.

Definition 2.3. A derivation of a sequent $\Gamma \Rightarrow_X \phi$ in a proof system P is denoted by $\Pi_P(\Gamma \Rightarrow_X \phi)$ and inductively defined as follows:

- $\Gamma \Rightarrow_X \phi$, if $\phi \in \Gamma$.
- $r(\Gamma \Rightarrow_X \phi, [\Pi_P(\Gamma_1 \Rightarrow_{X_1} \phi_1), \ldots, \Pi_P(\Gamma_n \Rightarrow_{X_n} \phi_n)])$, if $r \in P$ and $(\Gamma \Rightarrow_X \phi, \Gamma_1 \Rightarrow_{X_1} \phi_1, \ldots, \Gamma_n \Rightarrow_{X_n} \phi_n)$ is an instance of r

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

3 Adequate encoding of such a proof system

An encoding of a proof system can be considered adequate if there is an exact correspondance between syntax and proofs of the object system and their encodings. As we mentioned before, in [3] additionally to a metalanguage, the principle of judgements-as-types to encode logics is presented.

Our 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. Another difference with respect to the work of [3] and [1] is the encoding of syntax. We won't identify bound variables of formulae 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 in the previous section, the proof of adequacy doesn't need to redefine the object natural deduction 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 (Var_{UTT}), set of variables (Var_set_{UTT}), terms ($Term_{UTT}$), formulae ($Formula_{UTT}$) and environments ($Environment_{UTT}$). 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 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.

The inductive type $Formula_{UTT}$ will be defined with the following constructors for the cases of equality, $\wedge$ and $\Rightarrow$ operators, and universal quantification:

$$\begin{array}{l}\{Eq : \Pi t : Term_{UTT}.\Pi u : Term_{UTT}.Formula_{UTT},\\ and : \Pi A : Formula_{UTT}.\Pi B : Formula_{UTT}.Formula_{UTT},\\ implies : \Pi A : Formula_{UTT}.\Pi B : Formula_{UTT}.Formula_{UTT},\\ forall : \Pi A : Formula_{UTT}.Formula_{UTT}, \ldots\}\end{array}$$

and the substitution operation of a free variable by a term in a formula $subst\_f : Formula_{UTT} \rightarrow Term_{UTT} \rightarrow Var_{UTT} \rightarrow Formula_{UTT}$ can be trivially defined using the primitive recursion operator associated to $Formula_{UTT}$ because we can assume that no α-conversion is needed.

$Term_{UTT}$ for the signature of natural numbers with just the two basic operations would be defined by the constructors $\{fvar\_vt : \Pi var\_t : Var_{UTT}.Term_{UTT}, zero\_t : Term_{UTT}, succ\_t : Term_{UTT} \rightarrow Term_{UTT}\}$ and it is also needed the substitution operation $subst\_t : Term_{UTT} \rightarrow Term_{UTT} \rightarrow Var_{UTT} \rightarrow Term_{UTT}$.

Since we can assume that the infinite set of variables is countable, we can define Var_{UTT} as an inductive type with constructors $\{first\_var : Var_{UTT}, new\_var : Var_{UTT} \rightarrow Var_{UTT}\}$. The type of set of variables Var_set_{UTT} is defined with the constructors $\{empty\_varset : Var\_set_{UTT}, add\_varset : \Pi vs : Var\_set_{UTT}.Var\_set_{UTT}\}$, and predefined operations $get\_new\_varset : Var\_set_{UTT} \rightarrow Var_{UTT}$ (which given a set of variables returns the next fresh variable which is not in the set) and the inductive relation $Is\_in\_varset : \Pi v : Var_{UTT}.\Pi vs : Var\_set_{UTT}.Prop$ (which relates variables $v : Var_{UTT}$ with the sets of variables which contains v). Note that the normal forms of Var_{UTT} and Var_set_{UTT} are isomorphic in the sense that the constructors and their arities are equivalent although they are conceptually different.

Finally, $Environment_{UTT}$ is defined as a sequence of formulae with the constructor names $empty\_env$ and add_env.

For the encoding of formulae, we will assume that we have as a given argument the finite set of free variables of the given formula. Since we will encode variables of the object formulae in the type Var_{UTT}, we define first the function $\epsilon_{ve} : Var\_set \rightarrow Var\_env$ (and its inverse ϵ_{ve}^{-1}). ϵ_{ve} given a variable set returns an environment which associates variable names to variables in Var_{UTT}. The operations for Var_env which we assume predefined are $empty\_varenv : Var\_env$, $update\_varenv : (Var, Var_{UTT}) \rightarrow Var\_env \rightarrow Var\_env$, $get\_varenv : Var \rightarrow Var\_env \rightarrow Var_{UTT}$ (which gets the last updated value in Var_{UTT} for a given variable in Var and environment of type Var_env) and $get\_new\_varenv : Var\_env \rightarrow Var_{UTT}$ (which given a variable environment returns the next fresh variable of type Var_{UTT} of the given environment).

We also define the function $\epsilon_{vs} : Var\_set \rightarrow Var\_set_{UTT}$ which given a variable set returns just the set of variables in Var_{UTT} which are needed to encode the variable set. This set will be used to define well formed formulae with the inductive relation $Wf\_formula_{UTT}$ needed for the proof of adequacy of syntax. ϵ_{ve} and ϵ_{vs} are defined in such a way that the result of the latter can be seen as a projection of the result of the former.

From now on, we will use either Var_env or Var_set_{UTT} as arguments of functions or inductive relations, instead of the original set Var_set.

The encoding function of formulae $\epsilon_F : Var\_env \rightarrow Formula \rightarrow Formula_{UTT}$ is defined for universal quantification as

$$\begin{aligned} \epsilon_F \quad venv \quad \forall x.\phi \quad = \quad & forall \\ & (\epsilon_F \quad (add\_varenv \quad (get\_new\_varenv \quad venv)) \quad \phi) \end{aligned}$$

and the encoding function of terms $\epsilon_T : Var\_env \rightarrow Term \rightarrow Term_{UTT}$, is defined for the case in which the term is a variable as

$$\epsilon_T \quad venv \quad var\_name \quad = \quad get\_varenv \quad var\_name \quad venv$$

Adequacy of the encoding of syntax is guaranteed by the following theorem:

Theorem 3.1. *Let the inductive relation* $Wf\_formula_{UTT} : \Pi vs : Var\_set_{UTT}.\Pi form : Formula_{UTT}.Prop$ *be defined for the cases of equality and universal quantification with the constructors,*

$$\begin{aligned} \{ & Eq\_wff : \Pi vs : Var\_set_{UTT}.\Pi t, u : Term_{UTT}. \\ & \quad \Pi wft : Wf\_term_{UTT} \quad vs \quad t.\Pi wfu : Wf\_term_{UTT} \quad vs \quad u. \\ & \quad Wf\_formula_{UTT} \quad vs \quad (Eq \quad t \quad u), \\ & forall\_wff : \Pi vs : Var\_set_{UTT}.\Pi A : Formula_{UTT}. \\ & \quad \Pi wfA : Wf\_formula_{UTT} \quad (add\_varset\ (getnew\_varset\ vs)\ vs) \quad A. \\ & \quad Wf\_formula_{UTT} \quad vs \quad (forall \quad A), \ldots \} \end{aligned}$$

and let the inductive relation $Wf\_term : \Pi vs : Var\_set_{UTT}.\Pi t : Term_{UTT}.Prop$ *be defined for the variable case with the constructor,*

$$\{\ var\_wft : \Pi vs : Var\_set_{UTT}.\Pi v : Var\_UTT.$$
$$\Pi wfv : Is\_in\_varset \quad v \quad vs.Wf\_term\_UTT \quad vs \quad (var\_t \quad v)\}$$

1. *There is a bijection between the quotient of first order formulae closed under the finite set of free variables* X *by* α *conversion and the normal forms of formulae* $form : Formula_{UTT}$ *which are well formed,i.e that the proposition* $Wf\_formula \quad form \quad (\epsilon_{vs} \quad X)$ *is provable in the type theory and*
2. *the encoding is preserved under the substitution operation*

1. One can trivially prove that the encoding function of syntax is total and injective. To prove the bijection, we define the decoding functions $\epsilon_F^{-1} : Var\_env \rightarrow Formula_{UTT} \rightarrow Formula$ and $\epsilon_T^{-1} : Var\_env \rightarrow Term_{UTT} \rightarrow Term$ for the encoding functions presented above in a similar way to the original encoding functions. We can also prove that the decoding function is total. Note that proving totality is not possible if we take as domain of the decoding function the set of formulae belonging to the inductive type $Formula_{UTT}$. Finally, one can prove for all $form : Formula$ closed under the finite set of free variables X that

   $$[\epsilon_F^{-1} \quad (\epsilon_{ve} \quad X) \quad (\epsilon_F \quad (\epsilon_{ve} \quad X) \quad form)]_\alpha = [form]_\alpha$$

   by induction on $Formula$, 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 formulae $form : Formula$ under X, all terms $t : Term$ and all free variables x in X:

   $$subst\_f \quad (\epsilon_F \quad (\epsilon_{ve} \quad X) \quad form) \quad (\epsilon_T \quad (\epsilon_{ve} \quad X) \quad t)$$
   $$(get\_varenv \quad x \quad (\epsilon_{ve} \quad X)) \quad = \quad \epsilon_F \quad form[t/x]$$

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

One can easily define the encoding function of environments $\epsilon_D : Var\_env \rightarrow Environment \rightarrow Environment_{UTT}$ (and its proof of adequacy), from the encoding of formulae (and its proof of adequacy). Thus, we will assume predefined the inductive relation $Wf\_environment_{UTT} : \Pi vs : Var\_set_{UTT}.\Pi env : Environment_{UTT}.Prop$

The whole logical system is encoded by the inductive relation $FOLr : \Pi Env : Environment_{UTT}.\Pi vs : Var\_set_{UTT}.\Pi form : Formula_{UTT}.Prop$ defining for each rule of the logical system, a constructor in the inductive relation.

For example, the rule *reachn* defined in the previous section will be represented in the inductive relation by the constructor

$$\begin{array}{l} reachn : \Pi vs : Var\_set_{UTT}.\Pi Env : Environment_{UTT}. \\ \quad \Pi \phi : Formula_{UTT}. \\ \quad \Pi base\_case : FOLr \quad Env \quad vs \quad (subst\_f \quad \phi \quad zero \quad newvar). \\ \quad \Pi general\_case : FOLr \quad Env \quad (add\_varset \quad vs) \quad (implies \quad \phi \\ \qquad (subst\_f \quad \phi \quad (succ \quad (fvar\_vt \quad newvar)) \quad newvar)). \\ \qquad FOLr \quad vs \quad Env \quad (forall \quad \phi) \end{array}$$

where $newvar = get\_new\_varset \quad vs$ and as we see in this example, the type of the arguments of the constructor are first the type of the schematic variables which appear in the original rule, and then the type of the proofs of the premises of the rule, which of course depend on the type of the schematic variables.

The encoding of derivations ϵ_D which given a variable environment, a derivation tree of type $\Pi_{FOLr}(\Gamma \Rightarrow_X \phi)$, and a proof of the inductive relation $FOLr \quad (\epsilon_E \quad (\epsilon_{ve} \quad X) \quad \Gamma) \quad (\epsilon_{vs} \quad X) \quad (\epsilon_F \quad (\epsilon_{ve} \quad X) \quad \phi)$ is inductively defined by the first rule of the derivation. The case of $reachn$ as first rule of the derivation will be defined as,

$$\begin{array}{l}
\epsilon_D \quad venv \quad reachn(\Gamma \Rightarrow_X \phi, [\Pi_{FOLr}(\Gamma \Rightarrow_X \phi[zero/x]), \\
\quad \Pi_{FOLr}(\Gamma \Rightarrow_{X,x} \phi[succ\, x/x])]) = \\
\quad\quad reachn \quad (\epsilon_{ve} \quad X) \quad (\epsilon_E \quad (\epsilon_{ve} \quad X) \quad \Gamma) \quad (\epsilon_F \quad (\epsilon_{ve} \quad X) \quad \phi) \\
\quad\quad\quad (\epsilon_D \quad venv \quad (\Pi_{FOLr}(\Gamma \Rightarrow_X \phi[zero/x]))) \\
\quad\quad\quad (\epsilon_D \quad venv \quad (\Pi_{FOLr}(\Gamma \Rightarrow_{X,x} \phi[succ\, x/x])))
\end{array}$$

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

Theorem 3.2. *Let the well formed derivations of $FOLr$ be defined by the inductive relation $Wf\_FOLr : \Pi Env : Environment_{UTT}.\Pi vs : Var\_set_{UTT}.\Pi form : Formula_{UTT}.Prop$ with just the following constructor*

$$\begin{array}{l}
wffolr : \Pi vs : Var_set_{UTT}.\Pi Env : Environment_{UTT}. \\
\quad \Pi form : Formula_{UTT}. \\
\quad \Pi wfenv : Wf_environment_{UTT} \quad vs \quad Env. \\
\quad \Pi wff : Wf_formula_{UTT} \quad vs \quad form. \\
\quad \Pi derp : FOL_r \quad Env \quad vs \quad form. \\
\quad Wf_folr \quad Env \quad vs \quad form
\end{array}$$

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
$Wf\_FOLr \quad (\epsilon_E \quad (\epsilon_{ve} \quad X) \quad \Gamma) \quad (\epsilon_{vs} \quad X) \quad (\epsilon_F \quad (\epsilon_{ve} \quad X) \quad \phi)$

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 prove by induction that the encoding function is injective and total. To prove the bijection we define a decoding function which one can prove that it is total (because the domain is restricted to the inductive relation Wf_FOLr (and not to FOLr)) and it holds by an easy induction that for all closed derivations $deriv$ of the proof system over a variable environment $venv : Var\_env$ that $\epsilon_D^{-1} \quad venv \quad (\epsilon_D\, venv\, deriv) = deriv$ which is necessary to guarantee the bijection.

4 Conclusions

In this paper, we have presented the adequate encoding of a sound proof system with respect to the one associated to a simple specification expression of an ASL-like language presented in [4]. Further research should be done to cope with the more difficult cases such as the observability operators.

This work can be seen as another step forward to use a type theory with inductive types (like for example UTT), as a framework for software design. Thus, it is future work to implement the whole proof system using the Lego tool and use it to develop medium-case studies, like for example the design of a simple concurrent system.

Acknowledgements: I would like to thank Don Sanella for fruitful discussions about the topic and general comments about a draft version. This work has been partially supported by the Cosmos project with reference TIC95-1016-C02-01.

References

1. Philippa Gardner. *Representing Logics in Type Theory.* PhD thesis, University of Edinburgh, July 1992.
2. 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.
3. 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.
4. Rolf Hennicker, Martin Wirsing, and Michel Bidoit. Proof systems for structured specifications with observability operators. *To appear in Theoretical Computer Science*, 1997.
5. Zhaohui Luo. Program specification and data refinement in type theory. *Math. Struct. in Comp. Science*, 3:333–63, 1993.
6. Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science.* Clarendon Press Oxford, 1994.
7. Zhaohui Luo and Randy Pollack. LEGO proof development system: User's manual. Report ECS-LFCS-92-211, Department of Computer Science, University of Edinburgh, May 1992.
8. Nikos Mylonakis. Behavioural specifications in type theory. In Magne Haveraaen, Olaf Owe, and Ole-Johan Dahl, editors, *Recent Trends in Data Type Specification.*, volume ? of *LNCS*. Springer Verlag, 1996. 11th Workshop on Specification of Abstract Data Types joint with the 9th general COMPASS workshop. Oslo, Norway, September 1995. Selected papers.
9. Robert Pollack. *The Theory of LEGO.* PhD thesis, University of Edinburgh, April 1995.
10. Don Sannella and Rod Burstall. Structured theories in LCF. In *Proc. Colloq. on Trees in Algebra and Programming, L'Aquila*, number 159 in Springer LNCS, pages 377–91, 1983.
11. Don Sannella and Martin Wirsing. A kernel language for algebraic specification and implementation. In *Proc. Intl. Conf. on Foundations of Computation Theory, Borgholm, Sweden*, number 158 in Springer LNCS, pages 413–27, 1983.