DEPENDENT RECORD TYPES, SUBTYPING AND PROOF REUTILIZATION

GUSTAVO BETARTE (DRAFT)

DEPARTMENT OF COMPUTING SCIENCE
CHALMERS UNIVERSITY OF TECHNOLOGY AND UNIVERSITY OF GöTEBORG

Abstract. We present an example of formalization of systems of algebras using an extension of Martin-Löf's theory of types with record types and subtyping. This extension has been presented in [5]. In this paper we intend to illustrate all the features of the extended theory that we consider relevant for the task of formalizing algebraic constructions. We also provide code of the formalization as accepted by a type checker that has been implemented.

1. Introduction

We shall use an extension of Martin-Löf's theory of logical types [14] with dependent record types and subtyping as the formal language in which constructions concerning systems of algebras are going to be represented.

The original formulation of Martin-Löf's theory of types, from now on referred to as the logical framework, has been presented in [15, 7]. The system of types that this calculus embodies are the type *Set* (the type of inductively defined sets), dependent function types and for each set A, the type of the elements of A.

The extension of the logical framework with dependent record types and subtyping is presented in [5, 18]. Dependent record types are just sequences of *fields* in which *labels* are declared as of certain types. These types, in turn, may not only depend on objects but also on labels. How this dependency is obtained is formally introduced in the rules for record types formation that we present in section 2.2. Record objects, as in programming languages, are sequences of assignments of objects of appropriate types to labels. Each of these objects can be accessed by selecting the corresponding label of the record object. The mechanism of subtyping or type inclusion introduced is, in the first place, the one naturally induced by record types. However, once record inclusion is formally stipulated it is also required that rules of subtyping have to be given for the rest of the type formers.

In [5] we illustrate the use of record types and subtyping for the formalization of systems of algebras, namely, we develop a formal definition of group as a record type and provide very simple applications of subtyping. In this paper, we also focus in a simple example: We start out from binary relations, and by successively enriching previously defined notions with further structure, we finally define a *Boolean algebra* as a *distributive lattice* with additional structure. Then, we develop a little piece of the theory of Boolean algebras concerned with the proof of the DeMorgan's laws. This example, however, will allow us to illustrate what we consider are the relevant features of the extended theory.

The rest of the paper proceeds as follows: we give in next section a brief review on type theory, dependent record types and subtyping. Then, in section 3.1 we present the informal formulation of the algebraic notions with which we are concerned. These are literally taken from text books on lattice theory and universal algebra.

A proof checker for the extended theory has been implemented. In section 3.2 we go over the syntax of the input expressions and forms of declaration that the system reads, and give a succinct description of the object and type constructors ***let*** and ***use***. We proceed, then, in section 3.3, presenting the formal constructions developed in order to formalize the case study at hand. We do not provide the whole code involved in the formalization but rather concentrate on the fragments that we consider the most interesting ones.

Finally, in last section we discuss related work.

2. Representation of systems of algebras in type theory

We shall use an extension of Martin-Löf's theory of logical types [14] with dependent record types and subtyping as the formal language in which constructions concerning systems of algebras are going to be represented. A brief description of the main features of the original and the extended theory are given below. For a more comprehensive presentation we refer to [15, 7] and [5], respectively.

2.1. Type theory.

The system of types of the original formulation of the theory is constituted, in the first place, by the type *Set*, the type of *inductively* defined sets. Then, any individual set A, gives rise to the type of its elements. This latter type is denoted in [15] as $El(A)$, where El is a (primitive) family of types over the type *Set*. .Type families are expressions of the language that when applied to individuals of the appropriate type give rise to a type. Moreover, it is possible to introduce arbitrary families of types in the formal language. One such family can be constructed by an operation of abstraction, denoted as $[x]\alpha$, which binds the occurrences of the variable x in the type α. In what follows, we shall omit the family El and write just A for both the object of type set and the type that it determines. Finally, there exists a mechanism for the formation of (dependent) function types: if α is a type, and β is a family of types indexed by objects of types α then $\alpha{\rightarrow}\beta$ is also a type. The application of an object f of this latter type yields an object fa of type βa, if a is an object of type α.

The understanding of propositions as inductively defined by their introduction rules, explained and justified in [14], allows to grasp propositions as sets, and thereby, their proofs as elements of those sets. There is, in principle, no formal distinction in the language of the theory between the type of sets and the type of propositions. Further, in the presence of families of types, this interpretation of propositions can be transfered to propositions about generic individuals. For instance, given a set A, $A{\rightarrow}A{\rightarrow}Set$ is the type of binary relations on A. Then, if R is such a relation, for each element x of A we have a set Rxx. Since each set determines a type, we can form here a family of types over A, namely $[x]Rxx$. Then $A{\rightarrow}[x]Rxx$ is the type of proofs that R is reflexive. This function type is usually written as $(x : A)Rxx$, that can be read: "for any x in A, Rxx".

As another example, consider the type $(x, y : A)Rxy{\rightarrow}Ryx$. A function of this type will produce a proof of Ryx given any two elements x, y of A and a proof of Rxy. In virtue of the given explanations, this is the same as proving that if Rxy holds then so does Ryx, for arbitrary x, y in A, i.e. the symmetry of R.

2.2. Record types.

In order to achieve a formal definition of *Boolean algebra* we will start by introducing the notion of a set with an equivalence relation on it. The reason to have this as the most basic kind of structure is that in formalizing systems of algebras it appears natural to require the relation informally denoted by the equality symbol $\approx$ to be given explicitly as a component of the system being defined. We will call this basic structure *setoid* following [2] and it is formally introduced as a record type. Then, we will proceed by successively enriching the type of setoids with further structure, obtaining definitions for *lattice* and *distributive lattice*, until we get the formalization corresponding to Boolean algebras. Comments shall not abound here on the motivation for using dependent record types as the formal counterpart to the notion of system of algebras. We refer to [5] where a discussion concerning alternatives to this particular mechanism for forming tuple types is carried out.

Dependent record types are just sequences of *fields* in which *labels* are declared as of certain types:

$$\langle \mathsf{L}_1 : \alpha_1, ..., \mathsf{L}_n : \alpha_n \rangle.$$

In dependent record types, the type α_i may depend on the preceding labels L_1,...,L_{i-1}. More precisely, α_i has to be a family of types over the record type $\langle \mathsf{L}_1 : \alpha_1, ..., \mathsf{L}_{i-1} : \alpha_{i-1} \rangle$, for $i \geq 2$. The family α_1, in turn, is indexed by the *empty* record type $\langle\rangle$. This is formally expressed by the following two rules of record type formation:

$$\frac{}{\Gamma \vdash \langle\rangle : \mathit{record\text{-}type}} \qquad \frac{\Gamma \vdash \rho : \mathit{record\text{-}type} \quad \Gamma \vdash \beta : \rho \rightarrow \mathit{type}}{\Gamma \vdash \langle \rho, L{:}\beta \rangle : \mathit{record\text{-}type}} \ {\scriptstyle L \text{ fresh in } \rho}$$

We make use of the judgement $\Gamma \vdash \beta : \rho \rightarrow \mathit{type}$, which should be read as "β is a type family over the type ρ in the context Γ", to formally reflect that families of types are associated to labels in the formation of record types. For the sake of readability, however, in the notation that we are going to use in what follows labels are allowed to participate in the formation of types in the same way as ordinary variables or constants do. So, they have to be syntactically distinguished from the latter, in order to avoid ambiguities. We do this by writing labels in a distinguished font. On the other hand, it is possible to give a mechanical procedure to translate from the informal notation, where labels are singled out by means of a particular notation, to the corresponding formal representation involving families of types.

We can now write the type of binary relations on a set as:

$$\langle \mathsf{S} : \mathit{Set}, \mathsf{R} : \mathsf{S} \rightarrow \mathsf{S} \rightarrow \mathit{Set} \rangle.$$

Let us call this type *BinRel*.

Record objects are constructed as sequences of fields that are assignments of objects of appropriate types to labels:

$$\frac{}{\Gamma \vdash \langle\rangle : \langle\rangle} \qquad \frac{\Gamma \vdash r : \rho \quad \Gamma \vdash e : \beta r}{\Gamma \vdash \langle r, L = e \rangle : \langle \rho, L{:}\beta \rangle} \ {\scriptstyle L \text{ fresh in } \rho}$$

For instance, if N is the set of natural numbers and Id_N the usual propositional equality on N, then the following is an object of type *BinRel*:

$$\langle \mathsf{S} = N, \mathsf{R} = Id_N \rangle.$$

That two objects r and s of type $\langle \mathsf{L}_1 : \alpha_1, ..., \mathsf{L}_n : \alpha_n \rangle$ are the same means that the selection of the labels L_i's from r and s result in equal objects of the corresponding types.

Subtyping. The formalization of the example that we shall study introduces several types of algebraic structures by the procedure of extending previously defined types with further components and axioms. Eventually, we formulate the definition of *Boolean algebra* as that of a *distributive lattice* together with a unary operation $\sim$, two nullary operations $\mathsf{0}$ and $\mathsf{1}$ and the corresponding axioms for universal bounds and complementation.

Thus, we will obtain a formal definition of the system that sustains in a natural way the usual informal reasoning associated to these latter concepts. In other words, one makes use in the informal language, for instance, of the fact that any Boolean algebra is also a distributive lattice. Properties that any lattice satisfies are directly used when reasoning on Boolean algebras. In the formal language this is obtained by the (inclusion) polymorphism induced by record types : given a record type ρ_1, it is possible in general to drop and permute fields of ρ_1 and still get a record type ρ_2. Moreover, then any object of type ρ_1 satisfies also the requirements imposed by the type ρ_2. That is, given $r : \rho_1$, we are justified in asserting also $r : \rho_2$. This is so because what is required to make the latter judgement is that the selections of the labels declared in ρ_2 from r are defined as objects of the appropriate types. And we have this, since every label declared in ρ_2 is also declared in ρ_1 and with the same type.

In the formal language this idea is accomplished by introducing two new forms of judgement, namely, $\alpha_1 \sqsubseteq \alpha_2$ for types α_1 and α_2 and $\beta_1 \sqsubseteq \beta_2 : \alpha \rightarrow \mathit{type}$ for families β_1 and β_2 indexed by the type α. The reading of these forms of judgement is as follows: α_1 is a *subtype* of α_2, also referred sometimes as of *type inclusion*, and β_1 is a *subfamily* of β_2.

In the case of record types, the condition for $\rho_1 \sqsubseteq \rho_2$ is in words as follows: for each field $\mathsf{L} : \beta_2$ in ρ_2 there must be a field $\mathsf{L} : \beta_1$ in ρ_1 with $\beta_1 \sqsubseteq \beta_2 : \rho_1 \rightarrow \mathit{type}$.

The formal stipulation of this rule requires that rules of subtyping are given for all the type formers of the language: *Set* is a subtype only of itself, and if A and B are sets they are in the inclusion relation only if they are convertible. The rule of subtyping for function types departs from the one usually presented in the literature in that it also takes care of the dependencies.

For a detailed presentation of the theory, with the corresponding meaning explanations and justification of the rules of inference, we refer to [5].

3. BOOLEAN ALGEBRAS AND DEMORGAN'S LAWS FORMALIZED

We now consider the formalization of a little piece of the theory of Boolean algebras in type theory extended with record types and subtyping. The definitions introduced in the next section as well as the enunciation of some of the propositions are taken from [6] and [10].

3.1. Informal presentation.

There are two standard ways of defining lattices: one is to grasp them as an algebraic system and the other is based on the notion of order. Here, we shall follow the first approach.

Definition 3.1. A nonempty set L, with an equivalence relation $\approx$ defined on it, together with two binary operations $\vee$ and $\wedge$ (read *join* and *meet* respectively) on L is called a *lattice* if it satisfies the following identities:

$$
\begin{array}{lll}
\mathsf{L1}: & (\vee) & x \vee y \approx y \vee x \\
 & (\wedge) & x \wedge y \approx y \wedge x \qquad \text{(commutative laws)} \\[4pt]
\mathsf{L2}: & (\vee) & x \vee (y \vee z) \approx (x \vee y) \vee z \\
 & (\wedge) & x \wedge (y \wedge z) \approx (x \wedge y) \wedge z \qquad \text{(associative laws)} \\[4pt]
\mathsf{L3}: & (\vee) & x \vee x \approx x \\
 & (\wedge) & x \wedge x \approx x \qquad \text{(idempotent laws)} \\[4pt]
\mathsf{L4}: & (\vee) & x \approx x \vee (x \wedge y) \\
 & (\wedge) & x \approx x \wedge (x \vee y) \qquad \text{(absorption laws)}
\end{array}
$$

As is well known, it is in the very nature of the above definition that any property Φ valid for a lattice L is also valid if all occurrences of the operators $\vee$ and $\wedge$ in the formulation of the property are interchanged. The resulting property is called the *dual* of Φ. This observation can usually be found in text books enunciated as follows

Duality Principle. If a statement Φ is true in all lattices, then its dual is also true in all lattices.

There is nothing profound in this principle, however it gives rise to one of the most used methods of proof reutilization. Moreover, and particularly more convenient for the task we have in mind, the above principle can be equivalently grasped in terms of dual structures. That is to say, once we succeed in constructing a proof ϕ for a certain property Φ of a lattice L it can also be read, if carried out on the dual lattice of L, as a proof of the property dual of Φ.

There are many properties that can be proved to be derivable from the postulates (L1)-(L4). Here, however, we shall only enunciate the one that will manifest itself to be important in the development below.

Proposition 3.1. *A lattice L satisfies the following property*

If $x \approx x \vee y$ *and* $x \approx x \wedge y$ *then* $x \approx y$.

From now on, an algebraic system $\mathbf{S}$, whose carrier is the set S and whose (finite) set of operations (or operation symbols) is $\{f_1, \dots, f_k\}$ shall be denoted by $\langle S, f_1, \dots, f_k \rangle$. We shall also use $|\mathbf{S}|$ to stand for the carrier set of the algebra $\mathbf{S}$.

We now introduce the following

Definition 3.2. A *distributive lattice* is a lattice which satisfies either of the following (distributive) laws,

$$\mathsf{D1}: \quad x \wedge (y \vee z) \approx (x \wedge y) \vee (x \wedge z)$$

$$\mathsf{D2}: \quad x \vee (y \wedge z) \approx (x \vee y) \wedge (x \vee z)$$

The theorem below makes explicit that it suffices to require one of the laws above to be satisfied by a lattice L in order for it to be distributive.

Theorem 3.1. *A lattice L satisfies* D1 *iff it satisfies* D2

Definition 3.3. A *Boolean algebra* is an algebra $\langle B, \vee, \wedge, \sim, 0, 1 \rangle$ which two binary operations, one unary operation (called *complementation*), and two nullary operations which satisfies:

$\mathsf{B1}:\quad \langle B, \vee, \wedge \rangle$ is a distributive lattice

$$\mathsf{B2}:\quad \begin{array}{ll} (\vee) & x \vee 1 \approx 1 \\ (\wedge) & x \wedge 0 \approx 0 \end{array}$$

$$\mathsf{B3}:\quad \begin{array}{ll} (\vee) & x \vee {\sim}x \approx 1 \\ (\wedge) & x \wedge {\sim}x \approx 0 \end{array}$$

3.1.1. *DeMorgan's laws.* To begin with we enunciate some propositions that any boolean algebra satisfies. In what follows $\mathbf{B}$ is used to stand for a boolean algebra and x and y are elements of the carrier $|\mathbf{B}|$ of that algebra.

Proposition 3.2.

i) if $x \wedge y \approx 0$ then ${\sim}x \approx {\sim}x \vee y$

ii) if $x \vee y \approx 1$ then ${\sim}x \approx {\sim}x \wedge y$

Observe that they are dual propositions. It can readily be verified that using this latter lemma and the postulates B3, any boolean algebra $\mathbf{B}$ satisfies that ${\sim}({\sim}x) \approx x$, for all element x of $|\mathbf{B}|$.

The following lemma can easily been proved using Proposition 3.2 and Proposition 3.1.

Proposition 3.3. *If $x \wedge y \approx 0$ and $x \vee y \approx 1$ then ${\sim}x \approx y$*

Proof. We can use that $x \wedge y \approx 0$ and the first property in Proposition 3.2 to obtain that ${\sim}x \approx {\sim}x \vee y$. In a similar manner, from $x \vee y \approx 1$ and applying the second part of that same lemma we get ${\sim}x \approx {\sim}x \wedge y$. Thus, as $\mathbf{B}$ is a lattice, we can finally use Proposition 3.1 to get the desired conclusion. ☐

Two more propositions are introduced before we turn back to the laws with which we are concerned in this section

Proposition 3.4.

i) $(x \vee y) \wedge ({\sim}x \wedge {\sim}y) \approx 0$

ii) $(x \vee y) \vee ({\sim}x \wedge {\sim}y) \approx 1$

Finally, then, we are ready to formulate and prove the DeMorgan's laws for boolean algebras

Theorem (DeMorgan). *Let $\mathbf{B}$ be a boolean algebra, then for all elements x and y of $|\mathbf{B}|$,*

i) ${\sim}(x \vee y) \approx {\sim}x \wedge {\sim}y$

ii) ${\sim}(x \wedge y) \approx {\sim}x \vee {\sim}y$

Proof. We show the proof of the first law. The second follows by duality.

Notice that we know, by Proposition 3.4, that $\mathbf{B}$ satisfies both $(x \vee y) \wedge ({\sim}x \wedge {\sim}y) \approx 0$ and $(x \vee y) \vee ({\sim}x \wedge {\sim}y) \approx 1$. Therefore, Proposition 3.3 can directly be applied to get that ${\sim}(x \vee y) \approx {\sim}x \wedge {\sim}y$. ☐

$$\begin{array}{rll} e & ::= & x \mid c \mid [x]e \mid e_1 e_2 \mid \langle\rangle \mid \langle e_1, L = e_2\rangle \mid e.L \\ & & let\ x : e_1 = e_2\ in\ e \mid use\ e_1 : e_2\ in\ e \\ & & e_1 \to e_2 \mid \langle e_1, L{:}e_2\rangle \end{array}$$

FIGURE 1. Syntax of input expressions

3.2. **The proof checker.** A script for the proof checker looks very much like one for a functional programming language. The syntax of input expressions is given by the grammar in Figure 1.

The proof checker reads (non recursive) *declarations* of the following form:

$$\begin{array}{l} T : \mathtt{type}{=}\alpha \\ F(x : \alpha) : \mathtt{type} = \alpha_1 \\ c(x_1 : \alpha_1, \ldots, x_n : \alpha_n){:}\ \alpha = e \end{array}$$

with T, F and c constant names, x a variable and e, α and α_1 belonging to the language of expressions above.

The first one is called a *type* declaration. It allows to give an explicit definition for the type α.

The second form of declaration is called a *type family* declaration. It expresses the definition of a type family α_1, which is abbreviated by F and indexed by the variable x of type α. The index type has to be made explicit in order for the declaration to be type checked.

The third form of declaration allows the explicit definition, with name c, of an expression e of type α.

The two first are the counterpart in the system to the nominal definitions of types and families of types introduced in [5]. These forms of declaration are not present in the proof-assistant ALF.

The third form of declaration corresponds to the so-called explicit definition of a constant in ALF. We are considering neither definitions of inductive (families of) sets nor the implicit definition of constants, these latter usually defined using a pattern-matching mechanism provided by that proof-assistant.

The system works as an interpreter for a functional language too. Any declaration is checked under a current environment. Once the declaration D is checked to be correct, the environment is extended with it. Thereby, the definiendum of D may occur in any declaration introduced after it.

3.2.1. *Let and Use expressions.* The use of *let* expressions as a mean to introduce *local* declarations of proofs shares the motivation for using these sort of expression formers in functional languages like *ML* or *Haskell*. The possibility of abbreviating a proof object by a name, which, in turn, may occur in what is defined as its valid scope, not only alleviates notation, but also may render the process of proof checking more efficient. The way let expressions are checked in our system is heavily influenced by a proposal by Coquand in [8]. Namely, in order to check that an expression $let\ x : \alpha_1 = e\ in\ e_1$ has a certain type α in a environment $\mathcal{E}$ proceed as follows: check first that $x : \alpha_1 = e_1$ is a valid declaration in $\mathcal{E}$. If this succeeds check then that e is an object of type α in the environment $\mathcal{E}$ locally extended with $x : \alpha_1 = e_1$. Actually, it is possible to define let expressions involving a list of local declarations to the expression e. These, however, can not be mutually defined as they are in the programming languages mentioned above nor in the proof-assistant

Alfa currently being implemented at the Department of Computing Science at Chalmers University.

On the other hand, we have lately been experimenting with *use* expressions. The effect of "using" an expression r of type ρ in a expression e is almost analogous to the one performed by the Pascal-like command *with*. Therefore, in the first place, it does not suffice for ρ to be a type, it has to be a record type. Then, if L is an identifier syntactically equal to a label associated to a type family β in the fields of ρ it is, both for type checking and computation, considered to be definitionally equal to the object $r.L$ of type βr. This is correct if it has previously been checked that $r : \rho$. We can, then, informally explain how the expression $use\ r : \rho\ in\ e$ is checked to have type α in an environment $\mathcal{E}$: see to it first that ρ is a record type in environment $\mathcal{E}$. If this is the case, check whether r is an object of type ρ. Now, as ρ is a record type, it has necessarily to be either of the form $\langle\rangle$ or $\langle \mathsf{L}_1 : \beta_1, ..., \mathsf{L}_n : \beta_n \rangle$. In the first case just proceed by checking that e is an object of type α in $\mathcal{E}$. Otherwise, locally extend the environment $\mathcal{E}$ with declarations $L_i : \beta_i r = r.L_i$, for $i = 1..n$, and proceed by checking that e has type α in this latter environment.

We shall illustrate in next section how *use* expressions allow to amend the notational burden introduced by selections. Further, we think that the combination of *use* expressions and subtyping migth provide a mechanism to prevent accessing fields of a record object, in other words, the type ρ associated to the object r in a use expression may act as a sort of interface to the object. This latter, however, needs to be further investigated.

3.3. **Formalization.** We shall now proceed to give a formal account of the concepts in section 3.1. Thus, in the first place, we will have that the formulation of a property Φ is represented by a type T. Correspondingly, a particular proof ϕ of Φ, then, is introduced as an object of type T. Systems of algebras are formally introduced as record types. The use of type definitions and record types extension allows to naturally reflect the incremental definition of the various systems with which we were concerned in section 3.1.

We do not intend to give a complete presentation of the formalization but rather to illustrate the use of the extended type theory in the representation of algebraic constructions. More accurately, what we here mean by type theory is the particular implementation of the system we described in section 3.2.

$T : type$	$F : \alpha \rightarrow type$	$c : T$
$T = \alpha$	$F\ x = \alpha_1$	$c\ = \mathrm{e}$

FIGURE 2. Forms of declaration

3.3.1. *Preliminary definitions.* For the sake of readability we shall deviate a little from the syntax presented in section 3.2 for the forms of declaration and input expressions that the type checker reads. In Figure 2 we show how we denote in this section the definition of a type, a family of types and the abbreviation of an object of a certain type. At some points, when there is no interest in showing the code that a constant abbreviates, we make use of declarations of the form $c : T$.

We consider now the definition of some useful types and families of types intending, at the same time, to clarify the syntax of type expressions used in what follows. They are given in Figure 3. To begin with, the constant *binOp* is a type family over the type *Set*, whose intended meaning is that when applied to a certain set A it yields the type of the

$$\mathit{binOp} : \mathit{Set} \to \mathit{type}$$
$$\mathit{binOp}\ A = A \to A \to A$$

$$\mathit{Rel} : \mathit{Set} \to \mathit{type}$$
$$\mathit{Rel}\ A = A \to A \to \mathit{Set}$$

$$\mathit{BinRel} : \mathit{type}$$
$$\mathit{BinRel} = \langle \mathsf{A} : \mathit{Set}, \approx\ : \mathit{Rel}\ \mathsf{A} \rangle$$

$$\mathit{RelOp} : \mathit{type}$$
$$\mathit{RelOp} = \langle \mathit{BinRel}, \circ : \mathit{binOp}\ \mathsf{A} \rangle$$

$$\mathit{isTrans} : \mathit{BinRel} \to \mathit{type}$$
$$\mathit{isTrans}\ B = (x, y, z : B.\mathsf{A})\ B.\!\approx x\ y \to B.\!\approx y\ z \to B.\!\approx x\ z$$

$$\mathit{isCong} : \mathit{RelOp} \to \mathit{type}$$
$$\mathit{isCong}\ \mathit{Rop} = \boldsymbol{use}\ \ \mathit{Rop} : \mathit{RelOp}\ \ \boldsymbol{in}\ \ (x, y, z, w : S) \approx x\ z \to \approx y\ w \to \approx (\circ\ x\ y)\ (\circ\ z\ w)$$

FIGURE 3. Types and families of types

binary operations on that set. Observe that we are using that every set A is also a type. As propositions are identified with sets, the constant *Rel*, also a family indexed by *Set*, results in the type of binary relations over the set A if applied to this latter set. The definitions of *BinRel* and *RelOp* illustrate the two possible ways of defining a record type. Labels of records are written using the font label. Notice, particularly in the second of those latter definitions, that when extending a given record type it is possible to make reference to any of its labels in the fields that constitute the extension proper. The definition of *isTrans* shows the use of (functional) dependent types to express propositions. The type *isTrans* B can be read as follows: given a binary relation $\approx$ on a set A, for all elements x, y and z of A, if $\approx$ relates x and y, and y and z, then it also relates x and z. Finally, we show how a type can be defined by means of a ***use*** expression.

3.3.2. *Lattices.* We now turn to introduce the constructions corresponding to the ones presented in section 3.1. Thus, we start by defining the type of lattices. For the representation of this latter notion, and the other systems of algebras there introduced, we adopt the following methodology: we define, first, a record type that acts as the counterpart of the algebra – as defined in section 3.1 – that the system embodies. Then, this latter record type is extended with fields that conform as to the axioms of the system in question. In the case of lattices, in particular, there are two (dual) formulations of each law involved in the axiomatic part of the system. In Figure 4 we give a definition of various families of types indexed by the types *RelOp* and *RelOps*. They express respectively the different laws for lattices as types parameterized by a set, a binary relation defined on it and, in the three first cases, a binary operation over that same set. The last family is further parameterized by a second binary operation.

Now we carry on commenting the definition of lattices we present in Figure 5.

$$\mathit{isComm} : \mathit{RelOp} \to \mathit{type}$$
$$\mathit{isComm}\ \mathit{Rop} = \boldsymbol{use}\ \ \mathit{Rop} : \mathit{RelOp}\ \ \boldsymbol{in}\ \ (x, y : S) \approx (\circ\ x\ y)\ (\circ\ y\ x)$$

$$\mathit{isAssoc} : \mathit{RelOp} \to \mathit{type}$$
$$\mathit{isAssoc}\ \mathit{Rop} = \boldsymbol{use}\ \ \mathit{Rop} : \mathit{RelOp}\ \ \boldsymbol{in}\ \ (x, y, z : S) \approx (\circ\ x\ (\circ\ y\ z))\ (\circ\ (\circ\ x\ y)\ z)$$

$$\mathit{isIdemp} : \mathit{RelOp} \to \mathit{type}$$
$$\mathit{isIdemp}\ \mathit{Rop} = \boldsymbol{use}\ \ \mathit{Rop} : \mathit{RelOp}\ \ \boldsymbol{in}\ \ (x : S) \approx (\circ\ x\ x)\ x$$

$$\mathit{RelOps} : \mathit{type}$$
$$\mathit{RelOps} = \langle \mathit{RelOp}, * : \mathit{binOp}\ \mathsf{A} \rangle$$

$$\mathit{isAbsorb} : \mathit{RelOps} \to \mathit{type}$$
$$\mathit{isAbsorb}\ \mathit{Rops} = \boldsymbol{use}\ \ \mathit{Rops} : \mathit{RelOps}\ \ \boldsymbol{in}\ \ (x, y : S) \approx x\ (\circ\ x\ (*\ x\ y))$$

FIGURE 4. Axioms of lattices

As already anticipated, we first define a record type *PreLatt* as the formal counterpart of the algebra $\langle B, \vee, \wedge \rangle$. Notice that instead of asking just for a set to stand for the carrier of the algebra we consider the structure *Setoid*, which is a set S together with a binary equivalence relation $\approx$ defined on that set. The labels corresponding to the properties of $\approx$ are **refl**, **symm** and **trans** respectively.

Then, we define a function on *PreLatt*, whose intended meaning is to construct the dual out of an object of this latter type. This definition illustrates, on the one hand, how to obtain a record object by extending a given one. Moreover, and most significantly, notice that *Pl* is already an object of type *PreLatt*, however its extension is still considered to be an object of that type. This is correct because, in the first place, being *Pl* an object of type *PreLatt* it is also an object of type *Setoid*, by record inclusion. Furthermore, the objects $Pl.\wedge$ and $Pl.\vee$ are both objects of the appropriate type, namely, $\mathit{binOp}\ S$. On the other hand, by field overriding, the selection of the label $\vee$ (resp. $\wedge$) from the object resulting from the application of *dualPreLatt* to any object *Pl* of type *PreLatt* yields the object $Pl.\wedge$ (resp. $Pl.\vee$) as intended.

We then introduce a family of records *opOfLatt* over the type *RelOps*. This family expresses, principally, the properties that any two binary operations must satisfy in order to constitute, together with a given set, a particular lattice. Observe that the families in the field declarations are all applied to the same variable *Rops* of type *RelOps*. However, only *isAbsorb* was defined as a family over this latter type, the rest being indexed by *RelOp*. Their application to *Rops* is correct nevertheless due to the subtyping induced by record inclusion on families of types.

According to the observation made at the beginning of this section, then, the type of lattices is defined as the record type obtained by extending *PreLatt* with two more fields corresponding to the laws to be satisfied by the operators $\vee$ and $\wedge$ respectively. Thus, for instance, if L is an object of type *Latt*, the object $L.\vee$**Props.L1** is the proof that $L.\vee$ is commutative.

```
PreLatt : type
PreLatt = ⟨Setoid, ∨ : binOp S, ∧ : binOp S⟩

dualPreLatt : PreLatt→PreLatt
dualPreLatt Pl = ⟨Pl, ∨ = Pl.∧, ∧ = Pl.∨⟩

opOfLatt : RelOps→type
opOfLatt Rops =  ⟨ cong : isCong Rops,
                   L1 : isComm Rops,
                   L2 : isAssoc Rops,
                   L3 : isIdemp Rops,
                   L4 : isAbsorb Rops
                 ⟩

Latt : type
Latt =
   ⟨PreLatt,
    ∨Props : opOfLatt ⟨S = S, ≈ = ≈, ∘ = ∨, ∗ = ∧⟩,
    ∧Props : opOfLatt ⟨S = S, ≈ = ≈, ∘ = ∧, ∗ = ∨⟩
   ⟩

dualLatt : Latt→Latt
dualLatt L =  ⟨dualPreLatt L,
               ∨Props = L.∧Props,
               ∧Props = L.∨Props
              ⟩
```

Figure 5. Lattice

As to the definition of the function *dualLatt*, besides having with *dualPreLatt* in common the behaviour commented above, it also illustrates the use of subtyping but now for function objects, namely, the application of *dualPreLatt* to the variable L of type *Latt*.

From now on, we make use of $\%-\ \ \textit{comment}\ \ -\%$ to informally express the property being proved.

The definitions of congR∨ and congL∧ in Figure 6 illustrate the abbreviation of proof objects and the use of nested selection to access to components of record objects. The expression $[L\ x\ y\ \ z\ h]e$ should be read as the abstraction of the variables L, x, y, z and h in the expression e. The variable h corresponds to the hypotheses $L.\!\approx y\ z$ and $L.\!\approx x\ y$ respectively.

In Figure 7 there is the (almost complete) code of the proof that Proposition 3.1 is valid for any lattice L. The notation $\ldots h_i \ldots$ is used to stress the dependency on the hypothesis h_i.

The type of distributive lattices is shown in Figure 8. We declare as well the function *dualDistrLatt*, which behaves as expected.

$\% - \quad \forall\mathbf{B}.\forall x, y, z \in |\mathbf{B}|.y \approx z \supset x \vee y \approx x \vee z \quad - \%$

$\mathit{congR}\vee : (L : \mathit{Latt})\ (x, y, z : L.\mathsf{S})\ L.{\approx}\ y\ z \to L.{\approx}\ (L.\vee\ x\ y)\ (L.\vee\ x\ z)$
$\mathit{congR}\vee\ =\ [L\ x\ y\ z\ h]\ L.\vee\mathsf{Props.cong}\ x\ y\ z\ (L.\mathsf{refl}\ x)\ h$

$\% - \quad \forall\mathbf{B}.\forall x, y, z \in |\mathbf{B}|.x \approx y \supset x \wedge z \approx y \wedge z \quad - \%$

$\mathit{congL}\wedge : (L : \mathit{Latt})\ (x, y, z : L.\mathsf{S})\ L.{\approx}\ x\ y \to L.{\approx}\ (L.\wedge\ x\ z)\ (L.\wedge\ y\ z)$
$\mathit{congL}\wedge\ =\ [L\ x\ y\ z\ h]\ L.\wedge\mathsf{Props.cong}\ x\ y\ z\ h\ (L.\mathsf{refl}\ z)$

FIGURE 6. Congruences

$\% - \quad \forall\mathbf{B}.\forall x, y \in |\mathbf{B}|.(x \approx x \wedge y) \supset (x \approx x \vee y) \supset x \approx y \quad - \%$

$\mathit{antisymmL} :\ (L : \mathit{Latt})\ (x, y : L.\mathsf{S})$
$\qquad L.{\approx}\ x\ (L.\wedge\ x\ y) \to L.{\approx}\ x\ (L.\vee\ x\ y) \to L.{\approx}\ x\ y$
$\mathit{antisymmL}\ =$
$\quad [L\ x\ y\ h_1\ h_2]$
$\quad$ ***let***
$\qquad \mathit{lemm}_1 : L.{\approx}\ x\ (L.\vee\ y\ (L.\wedge\ x\ y)) = \ldots h_1 \ldots$
$\qquad \mathit{lemm}_2 : L.{\approx}\ (L.\vee\ y\ (L.\wedge\ x\ y))\ y = \ldots h_2 \ldots$
$\quad$ ***in***
$\qquad L.\mathsf{trans}\ \ x\ \ (L.\vee\ y\ (L.\wedge\ x\ y))\ \ y\ \ \mathit{lemm}_1\ \ \mathit{lemm}_2$

FIGURE 7. Antisymmetric property of lattices

$\mathit{DistrLatt} : \mathit{type}$
$\mathit{DistrLatt} =\ \langle \mathit{Latt},$
$\qquad \mathsf{D1} : \approx (\vee\ x\ (\wedge\ y\ z))\ (\wedge\ (\vee\ x\ y)\ (\vee\ x\ z)),$
$\qquad \mathsf{D2} : \approx (\wedge\ x\ (\vee\ y\ z))\ (\vee\ (\wedge\ x\ y)\ (\wedge\ x\ z))\ \rangle$
$\mathit{dualDistrLatt} : \mathit{DistrLatt} \to \mathit{DistrLatt}$

FIGURE 8. Distributive lattice

3.3.3. *Boolean Algebra.* The representation of the system of boolean algebras, the type *BoolAlg* in Figure 9, is built up in a similar manner as done for lattices. In order to make the code more legible, however, we chose not to group the axioms corresponding to the operators ∨ and ∧ respectively (notice the duality of B1 (resp. B3) and B2 (resp. B4)). We illustrate ***use*** expressions in the definition of the function *dualBoolAlg*.

3.3.4. *Proof of propositions 3.2-3.4 and DeMorgan laws.* We consider now the presentation of the proofs that were informally sketched in section 3.1 .

```
PreBoolAlg : type
PreBoolAlg =  ⟨DistrLatt,
               ∼ : S → S,
               0 : S,
               1 : S ⟩

dualPreBoolAlg : PreBoolAlg→PreBoolAlg
dualPreBoolAlg Pba =  ⟨dualDistrLatt Pba,
                       ∼ = Pba.∼,
                       0 = Pba.1,
                       1 = Pba.0 ⟩

BoolAlg : type
BoolAlg =  ⟨PreBoolAlg
            compCong : (x, y : S) ≈ x y → ≈ ∼x ∼y,
            B1 : (x : S) ≈ (∨ x 1) 1,
            B2 : (x : S) ≈ (∧ x 0) 0,
            B3 : (x : S) ≈ (∨ x ∼x) 1,
            B4 : (x : S) ≈ (∧ x ∼x) 0 ⟩

dualBoolAlg : BoolAlg→BoolAlg
dualBoolAlg Ba =  use Ba : BoolAlg
                  in  ⟨dualPreBoolAlg Ba,
                       compCong = compCong,
                       B1 = B2,
                       B2 = B1,
                       B3 = B4,
                       B4 = B3 ⟩
```

FIGURE 9. Boolean algebra

In Figure 10 we give the code of the proof of the first part of Proposition 3.2. Notice how ***use*** expressions improve the readability of the code, there is no need for explicit construction of selections. Moreover, notice that in the type of *prop3.2(i)* the variable Ba is used as an object of type *PreLatt*. The objects $trans_2$, $lemm_1$ and $lemm_2$, which are locally declared by means of the ***let*** constructor, are a typical example of local lemmas. Observe that the constants *congR*∨ and *congL*∧ defined in Figure 6 for lattices are applied to the variable Ba of type *BoolAlg*.

The constants *0identR*∨, *1identL*∧ and *commArgsB3* abbreviate the proofs of three properties which can easily be proved to be derivable from the postulates B1-B4. They are declared in Figure 11.

Let us now consider the second part of Proposition 3.2. We made, in section 3.1, the remark on the duality of the properties enunciated in this latter lemma. It was there also manifested our intention of obtaining the proof of the dual of a given property Φ on a certain structure S in terms of a proof ϕ of Φ. Accordingly, then, the proof of the part ii) of the above lemma is constructed by applying, and with this we mean function

```
%-   ∀B.∀x, y ∈ |B|.x ∧ y ≈ 0 ⊃ ∼x ≈ ∼x ∨ y   -%

prop3.2(i) :  (Ba : BoolAlg) (x, y : Ba.S)
              use Ba : PreLatt in ≈ (∧ x y) 0 → ≈ (∼ x) (∨ (∼ x) y)
prop3.2(i) =
  [Ba x y h]
   use Ba : BoolAlg
   in
    let
       trans2 : (x, y, z, w : S) ≈ x y → ≈ y z → ≈ z w → ≈ x z
         = [x y z w h1 h2 h3] trans x z w (trans x y z h1 h2) h3
       lemm1 : ≈ ∼x (∧ (∨ ∼x x) (∨ ∼x y))
         = trans2 ∼x (∨ ∼x 0) (∨ ∼x (∧ x y)) (∧ (∨ ∼x x) (∨ ∼x y))
                  (0identR∨ Ba ∼x)
                  (congR∨ Ba ∼x 0 (∧ x y) (symm (∧ x y) 0 h))
                  (D1 ∼x x y)
       lemm2 : ≈ (∧ (∨ ∼x x) (∨ ∼x y)) (∨ ∼x y)
         = trans (∧ (∨ ∼x x) (∨ ∼x y)) (∧ 1 (∨ ∼x y)) (∨ ∼x y)
                 (congL∧ Ba (∨ ∼x x) 1 (∨ ∼x y) (commArgsB3 Ba x))
                 (1identL∧ Ba (∨ ∼x y))
    in
       trans ∼x  (∧ (∨ ∼x x) (∨ ∼x y)) (∨ ∼x y) lemm1 lemm2
```

Figure 10. Proposition 3.2(i)

```
%-   ∀B.∀x ∈ |B|. x ≈ x ∨ 0   -%

0identR∨ : (Ba : BoolAlg) (x : Ba.S) ≈ x (Ba.∨ x 0)


%-   ∀B.∀x ∈ |B|. 1 ∧ x ≈ x   -%

1identL∧ : (Ba : BoolAlg) (x : Ba.S) ≈ (Ba.∧ 1 x) x


%-   ∀B.∀x ∈ |B|. ∼x ∨ x ≈ 1   -%

commArgsB3 : (Ba : BoolAlg) (x : Ba.S) ≈ (Ba.∨ ∼x x) 1
```

Figure 11. Simple derived properties of Boolean algebras

application, the object *prop3.2(i)* to the dual structure of *Ba*, i.e. *dualBoolAlg Ba*. This construction is shown in Figure 12.

The informal argument given for the validity of Proposition 3.3 is straightforwardly codified, as presented in that same figure. Note the application of *antisymmL* to the variable *Ba* of type *BoolAlg*.

$$\% - \quad \forall \mathbf{B}.\forall x, y \in |\mathbf{B}|.x \vee y \approx 1 \supset {\sim}x \approx {\sim}x \wedge y \quad - \%$$

$$\begin{aligned} \mathit{prop3.2(ii)} :\ & (\mathit{Ba} : \mathit{BoolAlg})\ (x, y : \mathit{Ba}.\mathsf{S})\\ & \boldsymbol{use}\ \ \mathit{Ba} : \mathit{PreBoolAlg}\ \ \boldsymbol{in}\ \approx (\vee\ x\ y)\ 1 \to \approx ({\sim}\ x)\ (\wedge\ ({\sim}\ x)\ y)\\ \mathit{prop3.2(ii)}\ =\ & [\mathit{Ba}]\ \mathit{prop3.2(i)}\ (\mathit{dualBoolAlg}\ \mathit{Ba}) \end{aligned}$$

$$\% - \quad \forall \mathbf{B}.\forall x, y \in |\mathbf{B}|.x \wedge y \approx 0 \supset x \vee y \approx 1 \supset {\sim}x \approx y \quad - \%$$

$$\begin{aligned} \mathit{prop3.3} :\ & (\mathit{Ba} : \mathit{BoolAlg})\ (x, y : \mathit{Ba}.\mathsf{S})\\ & \boldsymbol{use}\ \ \mathit{Ba} : \mathit{PreBoolAlg}\ \ \boldsymbol{in}\ \approx (\wedge\ x\ y)\ 0 \to \approx (\vee\ x\ y)\ 1 \to \approx ({\sim}\ x)\ y \end{aligned}$$

$$\begin{aligned} & \mathit{prop3.3}\ =\\ & \quad [\mathit{Ba}\ x\ y\ h_1\ h_2]\\ & \quad \mathit{antisymmL}\ \mathit{Ba}\ (\mathit{Ba}.{\sim}\ x)\ y\ (\mathit{prop3.2(i)}\ \mathit{Ba}\ x\ y\ h_1)\ \ (\mathit{prop3.2(ii)}\ \mathit{Ba}\ x\ y\ h_2) \end{aligned}$$

FIGURE 12. Propositions 3.2(ii) - 3.3

$$\% - \quad \forall \mathbf{B}.\forall x, y \in |\mathbf{B}|.(x \vee y) \wedge ({\sim}x \wedge {\sim}y) \approx 0 \quad - \%$$

$$\begin{aligned} \mathit{prop3.4(i)} :\ & (\mathit{Ba} : \mathit{BoolAlg})\ (x, y : \mathit{Ba}.\mathsf{S})\\ & \boldsymbol{use}\ \ \mathit{Ba} : \mathit{PreBoolAlg}\ \ \boldsymbol{in}\ \approx (\wedge\ (\vee\ x\ y)\ (\wedge\ ({\sim}\ x)\ ({\sim}\ y)))\ 0 \end{aligned}$$

$$\% - \quad \forall \mathbf{B}.\forall x, y \in |\mathbf{B}|.(x \vee y) \wedge ({\sim}x \vee {\sim}y) \approx 1 \quad - \%$$

$$\begin{aligned} \mathit{prop3.4(ii)} :\ & (\mathit{Ba} : \mathit{BoolAlg})\ (x, y : \mathit{Ba}.\mathsf{S})\\ & \boldsymbol{use}\ \ \mathit{Ba} : \mathit{PreBoolAlg}\ \ \boldsymbol{in}\ \approx (\wedge\ (\vee\ x\ y)\ (\vee\ ({\sim}\ x)\ ({\sim}\ y)))\ 1 \end{aligned}$$

FIGURE 13. Proposition 3.4

In Figure 13 we declare the constants *prop3.4(i)* and *prop3.4(ii)* to stand for the proofs of the two properties of Proposition 3.4. The construction of those proofs is routine.

We end up presenting in Figure 14 the proof objects corresponding to the DeMorgan's laws. Again, the object abbreviated by *DeMorgan(i)* is a direct formalization of the argument given in section 3.1 for showing the validity of this property. As expected, the proof *DeMorgan(ii)* of the second law is obtained by applying *DeMorgan(i)* to the object *dualBoolAlg Ba*.

4. RELATED WORK AND CONCLUSIONS

In the context of variants of type theory, [11] and [4] have considered using (dependent) pair types (Σ types) for expressing tuple types as required by the formalization of algebraic structures. In pairs the components are not labeled —they are accessed by fixed projection operations, usually called *fst* and *snd*. As a consequence, there is no counterpart for Σ types of the inclusion relation that can be defined between record types.

In [4], the inclusion relation between types of algebraic structures that we have been discussing is represented using *coercion* functions between the types that represent those

$\%- \quad \forall\mathbf{B}.\forall x, y \in |\mathbf{B}|.{\sim}(x \vee y) \approx ({\sim}x \wedge {\sim}y) \quad -\%$

$$\begin{aligned}
&\mathit{DeMorgan(i)} : \ (\mathit{Ba} : \mathit{BoolAlg})\ (x, y : \mathit{Ba}.\mathsf{S})\\
&\qquad\qquad\qquad \boldsymbol{use}\ \ \mathit{Ba} : \mathit{PreBoolAlg}\ \ \boldsymbol{in}\ \approx ({\sim}\ (\vee\ x\ y))\ (\wedge\ ({\sim}\ x)\ ({\sim}\ y))\\
&\mathit{DeMorgan(i)} \ =\\
&\quad [\mathit{Ba}\ x\ y]\\
&\quad \boldsymbol{use}\ \mathit{Ba} : \mathit{BoolAlg}\\
&\quad \boldsymbol{in}\ \mathit{prop3.3}\ \mathit{Ba}\ (\vee\ x\ y)\ (\wedge\ ({\sim}\ x)\ ({\sim}\ y))\ (\mathit{prop3.4(i)}\ \mathit{Ba}\ x\ y)\ (\mathit{prop3.4(ii)}\ \mathit{Ba}\ x\ y)
\end{aligned}$$

$\%- \quad \forall\mathbf{B}.\forall x, y \in |\mathbf{B}|.{\sim}(x \wedge y) \approx ({\sim}x \vee {\sim}y) \quad -\%$

$$\begin{aligned}
&\mathit{DeMorgan(ii)} : \ (\mathit{Ba} : \mathit{BoolAlg})\ (x, y : \mathit{Ba}.\mathsf{S})\\
&\qquad\qquad\qquad \boldsymbol{use}\ \ \mathit{Ba} : \mathit{PreBoolAlg}\ \ \boldsymbol{in}\ \approx ({\sim}\ (\wedge\ x\ y))\ (\vee\ ({\sim}\ x)\ ({\sim}\ y))\\
&\mathit{DeMorgan(ii)} \ = \ [\mathit{Ba}]\ \mathit{DeMorgan(i)}\ (\mathit{dualBoolAlg}\ \mathit{Ba})
\end{aligned}$$

FIGURE 14. DeMorgan laws

structures. But a mechanism is introduced that allows to leave the coercions implicit. This is formulated for pure type systems and extends ideas given originally in [1]. Then applications are shown using the extended calculus of constructions [12], which contains Σ types.

In [3] and [17], mechanisms for defining coercions between types or classes of types are developed for the proof-assistants *LEGO* [16] and *Coq* [9], respectively.

In [13], a calculus in the spirit of Martin-Löf's theory of types is presented, where forms of judgement are introduced, among others, that express the concepts of a kind K being a principal kind of an object k and (proper) kind inclusion. The meaning explanation of the relation of (proper) subkinding is given in terms of coercions. This makes it possible to justify the various coercive rules of the calculus which are expressed as judgemental equalities. As a particular example the author illustrates the use of coercions in the formalization of algebraic constructions.

The mechanism of subtyping obtained in all these works is, on the one hand, more limited than the one formulated in [5] and illustrated in this paper, since the inclusions that can be verified to hold are only those induced from the explicitly declared coercions. On the other hand, they all achieve forms of subtyping not coming from record subtyping. We consider this an interesting topic for further research in the framework of Martin-Löf's type theory

REFERENCES

[1] Peter Aczel. A Notion of Class for Theory Development in Algebra (in a Predicative type theory). Presented at Workshop of Types for Proofs and Programs, Båstad, Sweden, June 1994.

[2] Peter Aczel. Galois : A Theory Development Project. A report on work in progress for the Turin meeting on the representation of mathematics in Logical frameworks, 1993.

[3] Anthony Bailey. Lego with implicit coercions. Available at `ftp.cs.man.ac.uk/pub/baileya/Coercions`.

[4] Gilles Barthe. Implicit coercions in type systems. In *Selected Papers from the International Workshop TYPES '95, Torino, Italy, LNCS 1158.*, 1995.

[5] Gustavo Betarte and Alvaro Tasistro. Extension of Martin-Löf 's Type Theory with Record Types and Subtyping. To be published in the proceedings of the conference 25 Years of Constructive Type Theory, Oxford University Press, 1997.

[6] Stanley Burris and H.P. Sankappanavar. *A Course in Universal Algebra.* Graduate Texts in Mathematics, Springer-Verlag, 1981.

[7] Th. Coquand, B. Nordström, J.M. Smith, and B. von Sydow. Type theory and programming. In *EATCS 52*, 1994.

[8] Thierry Coquand. An algorithm for type-checking dependent types. In *Science of Computer Programming 26*, pages 167–177, 1996.

[9] Bruno Barras et al. The Coq Proof Assistant Reference Manual – Version V6.1. Technical Report 0203, INRIA, 1997.

[10] George Grätzer. *Lattice Theory. First concepts and Distributive Lattices.* W. H. Freeman and Company, 1971.

[11] Paul Jackson. Enhancing the Nuprl Proof Development System and Applying it to Computational Abstract Algebra. Ph.D. thesis, Cornell University, 1995.

[12] Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science.* Number 11 in International Series of Monographs on Computer Science. Oxford University Press, 1994.

[13] Zhaoui Luo. Coercive subtyping in type theory. In *CSL'96, the 1996 Annual Conference of the European Association for Computer Science Logic*, Utrech.

[14] Per Martin-Löf. Philosophical Implications of Type Theory. Lectures given at the Facoltá de Lettere e Filosofia, Universitá degli Studi di Firenze, Florence, March 15th. - May 15th., 1987. Privately circulated notes.

[15] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory.An Introduction.* Oxford University Press, 1989.

[16] Robert Pollack. The Theory of LEGO: a proof checker for the Extended Calculus of Constructions. Doctor of Philosophy thesis, University of Edinburgh, 1994.

[17] Amokrane Saïbi. Typing algorithm in type theory with inheritance. In *24th. Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997.

[18] Alvaro Tasistro. Substitution, record types and subtyping in type theory, with applications to the theory of programming. Ph.D. thesis. Programming Methodology Group, Dept. of Computing Science, University of Göteborg and Chalmers University of Technology, 1997.

E-mail address: `gustun@cs.chalmers.se`