PROOF REUTILIZATION IN MARTIN-LÖF'S LOGICAL FRAMEWORK EXTENDED WITH RECORD TYPES AND SUBTYPING ♣

GUSTAVO BETARTE

INSTITUTO DE COMPUTACIÓN
UNIVERSIDAD DE LA REPÚBLICA
MONTEVIDEO, URUGUAY

ABSTRACT. The extension of Martin-Löf's theory of types with record types and subtyping has elsewhere been presented. We give a concise description of that theory and motivate its use for the formalization of systems of algebras. We also give a short account of a proof checker that has been implemented on machine. The logical heart of the checker is constituted by the procedures for the mechanical verification of the forms of judgement of a particular formulation of the extension. The case study that we put forward in this work has been developed and mechanically verified using the implemented system. We illustrate all the features of the extended theory that we consider relevant for the task of formalizing algebraic constructions.

1. INTRODUCTION

The original formulation of Martin-Löf's theory of types, from now on referred to as the logical framework, has been presented in [NPS89, Tas93, CNSvS94]. 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 [BT98, Tas97]. 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 2.1. 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 [Bet98] we have investigated an alternative formulation of the extended theory. In that formulation we make use of parameters, in the sense of [Coq91, Pol94b], to stand for generic objects of the various types. The introduction of the notion of parameter allows us to give a solution to the problems posed by the manipulation of "free names" in the presence of dependent types. We also present in [Bet98] the procedures for the mechanical verification of the forms of judgement of that variant

Date: September 12, 2000.

♣ THIS PAPER IS AN ENHANCED VERSION OF [Bet97]

of the extension. In [Bet00] we make a detailed presentation of the investigations concerned with the design, specification and correctness of those procedures.

In this paper we shall illustrate the use of record types and subtyping for the formalization of systems of algebras. We focus on 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 DeMorgan's laws. This example will allow us to illustrate what we consider to be the relevant features of the extended theory.

The plan for the rest of the article is as follows. In the next section we start by giving a concise description of type theory and its use for carrying out constructive mathematics.Then we give a short account of dependent record types and the mechanism of subtyping they induce, comment on the treatment of free names in systems of dependent types and briefly describe the proof checker we have implemented on machine. 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. We proceed then, in section 3.2, presenting the formal constructions developed in order to formalize the case study. We do not provide the whole code involved in the formalization. Rather we concentrate on the fragments that we consider most interesting, that is to say, those that illustrate how algebraic constructions commonly used in the informal practice are reflected in the formal language.

Finally, in the last section we discuss related work.

2. The system and its implementation

The system of types of the logical framework 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. Type families are expressions of the language that when applied to an individual of the appropriate type yields a type. Moreover, it is possible to introduce arbitrary families of types in the formal language. Families can be constructed by an operation of abstraction, using the notation $[x]\alpha$, which binds the occurrences of the variable x in the type α. 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 type 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, as explained and justified in [Mar87], allows us 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 transferred to propositions about generic individuals. For instance, given a set A, $A{\rightarrow}[x](A{\rightarrow}[y]Set)$ 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 $(x, y : A)Rxy \to Ryx$. 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.

2.1. **Record types.** 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+1} may depend on the preceding labels $\mathsf{L}_1, \ldots, \mathsf{L}_i$. More precisely, α_{i+1} has to be a family of types over the record type $\langle \mathsf{L}_1 : \alpha_1, ..., \mathsf{L}_i : \alpha_i \rangle$. This is formally expressed by the following two rules of record type formation:

$$\frac{}{\langle\rangle : \textit{record-type}} \qquad \frac{\rho : \textit{record-type} \quad \beta : \rho \to \textit{type}}{\langle \rho, L{:}\beta \rangle : \textit{record-type}} \; {\scriptstyle L \text{ fresh in } \rho}$$

We make use of the judgement $\beta : \rho \to type$, which should be read "β is a family of types over the type ρ", to formally reflect that families of types are associated to labels in the formation of record types.

In the case of record types generated by the second clause, $L{:}\beta$ is a field and L a label, which we say to be declared in the field in question. Labels are just identifiers, i.e. names. In the formal notation that we are introducing there will actually arise no situation in which labels can be confused with either constants or variables. Notice that labels may occur at most once in each record type. That a label L is not declared in a record type ρ is referred to as L *fresh in* ρ. Finally, that these are *dependent* record types is expressed in the second clause, in the following way. The "type" declared to the new label is in fact a family β on ρ, i.e. it is allowed to use the labels already present in ρ. In fact, what β is allowed to use is a generic object (i.e. a variable) r of type ρ. Then the labels in ρ will appear in β as taking part in selections from r. Here below we show how the type of binary relations on a given set, which we shall call *BinRel*, is written:

$$\langle \mathsf{S} : Set, \approx : \mathsf{S} \to \mathsf{S} \to Set \rangle$$

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

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

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*:

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

2.2. **Subtyping.** Dependent record types also induce inclusion polymorphism: given a record type ρ_1, it is possible to drop and permute fields of ρ_1 and still get a record type ρ_2. If that is the case, any object of type ρ_1 also satisfies 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 $\beta_1 \sqsubseteq \beta_2 : \alpha \to type$ 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 and β_1 is a *subfamily* of β_2. We shall also refer to the first one as *type inclusion*.

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 $\beta_1 \sqsubseteq \beta_2 : \rho_1 \to type$. It is easy to see that if $\mathsf{L} : \beta_1$ is a field of a record type ρ_1 then, by the subtyping induced on families of types, β_2 can be considered to be a family over ρ_1 and thereby the previous (informal) explanation makes sense.

The formal stipulation of this latter 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 extends the one usually presented in the literature in that it also takes care of the dependencies.

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. Therefore, equality of record objects is based on a kind of extensionality principle. That is, the two rules below can be understood as defining that two objects of a given record type are equal if the selections of every label of the record type in question from the objects are equal. Notice that the type in which two record objects are compared is relevant: suppose namely that r and s are of type ρ_1 and that $\rho_1 \sqsubseteq \rho_2$. Then it may well be the case that $r{=}s : \rho_2$ but not $r{=}s : \rho_1$.

$$\frac{r{:}\langle\rangle \quad s{:}\langle\rangle}{r{=}s : \langle\rangle} \qquad\qquad \frac{r{=}s : \rho \quad r.L{=}s.L : \beta r}{r{=}s : \langle \rho, L{:}\beta \rangle.}$$

To understand the second of these rules notice that the premisses that both r and s are of type $\langle \rho, L{:}\beta \rangle$ have been omitted. We remark that these are rules of equality, they must not be understood as reduction rules.

2.3. **Parameters.** The traditional formulation of the rule for the formation of a function type, for instance as presented in [NPS89], says that if we know that α is a type and that β is a type family under the assumption that a variable x is of type α then we can form the function type $(x : \alpha)\beta$, where all occurrences of x in β become bound. Abstraction then is introduced as an operation of object formation. This is the corresponding rule

$$\frac{\Gamma, x{:}\alpha \vdash b : \beta}{\Gamma \vdash [x]b : (x : \alpha)\beta}$$

The stipulation for the formation of a context $\Gamma, x{:}\alpha$ in [Mar87, NPS89, Mar92], for instance, requires that Γ is a context, α is a type under the context Γ and, further, that the variable x has not already been declared in Γ. This last restriction

is proper of systems of proof rules where an assumption, $x{:}\alpha$ say, may be introduced such that the type α depends on previous assumptions. Therefore, for the premiss of the latter rule of abstraction to be correct it must be the case that x is not already declared in the context Γ.

In [Pol94b] Pollack discusses some consequences of having the restriction above for context formation in the implementation of type checkers for languages with binding operators, and more specifically, with systems of dependent types. The system of proof rules on which the discussion is centered is what has elsewhere been called Pure Type Systems (PTS), as originally presented in [Bar92]. What is shown by Pollack is the impossibility of deriving, using the rules of PTS, the judgement $[x][x]x : (x : A)(y : Px)Px$ under the assumption that A is a type (an object of $*$) and P has kind $A{\to}*$. If one wants to understand the checking of the correctness of instances of the judgement $\Gamma \vdash [x]b : (x : \alpha)\beta$ as the upward reading of the rule of abstraction one should proceed as follows: for checking that $[x][x]x : (x : A)(y : Px)Px$ check that $x{:}A \vdash [x]x : (y : Px)Px$. For this, in turn, we should check that $x : Px$ after extending the context $x{:}A$ with the declaration $x{:}Px$, but we are restrained from doing this by the criterion for context formation above.

Relatively recent works on the construction of proof-checkers for type theories with dependent types have addressed (in a direct manner or not) the problems presented above.

In [Coq91] Coquand investigates the question of checking the formal correctness of judgements of type and object equality in a formulation of Martin-Löf's set theory with generalized cartesian product and one universe.

The notion of context in this theory is that of a list of assumptions of the form $p{:}\alpha$, where p is a parameter and α a type (possibly depending on other parameters). In the formulation of the language of the theory, parameters are understood to play the role of the free variables occurring in the expressions. Consequently, they are used in the system to stand for generic objects of the various types. However, they are defined to be syntactic constructions distinct from the bound variables of the language. The distinction between parameters and bound variables allows to define a simplified operation of substitution on expressions where no mechanism of renaming has to be considered in order to avoid capture. Further, there is no need for an a priori identification of α-convertible terms for the algorithm to be defined. This latter is, we think, quite a relevant point if one wants to describe an actual implementation.

In [Pol94a] Pollack adopts the use of parameters to implement a type checking algorithm for a family of PTS [Bar92]. One of the motivations for introducing the notion of parameter and consequently make use of them in the reformulation of the rules of inference of the formal system is to provide a solution for problems similar to the ones discussed above.

In [Bet98] we formulate a variant of the extended theory presented in [Tas97, BT98]. A first difference is that we consider the rules of inference in their generalized form. Further, we make use of parameters to stand for generic objects of the various types. Thereby, as the stipulation of an assumption will correspond to declare a parameter as of a certain type, the explanation of a relative judgement depends on what are considered to be the permissible assignments of values to the parameters

involved in such judgement. These assignments, in turn, are defined in terms of a particular notion of substitution which, in contrast to the one usually defined for the language of type theory, behaves as the textual replacement of a parameter by an expression.

The algorithms presented in [Bet98, Bet00] implement the mechanical verification of the forms of judgement of the calculus we just described.

2.4. **The implemented system.** The proof checker has been implemented using the programming language Haskell 1.4 [Pet96] and compiled using Chalmers Haskell-B [Aug97].

A very simple XEmacs interface has also been incorporated to the system.Even though it is still in a very primitive stage, we have found its use to be of considerable help to the task of proof construction[1].

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.

$$\begin{array}{rcl} i & ::= & x \mid c \mid [x]i \mid i_1 i_2 \mid \langle\rangle \mid \langle i_1, L = i_2\rangle \mid i.L \\ & & \mathit{let}\ x : i_1 = i_2\ \mathit{in}\ i \mid \mathit{use}\ i_1 : i_2\ \mathit{in}\ i \\ & & i_1 \to i_2 \mid \langle i_1, L{:}i_2\rangle \end{array}$$

FIGURE 1. Syntax of input expressions

The symbol x ranges over a denumerable set V, the set of variables. The symbol c ranges over a countable set C of constants, which is defined to be disjoint with V. There are three distinguished constants *Set*, *type* and *record-type*, from now on called *sorts*. Only the first one may occur in a valid input expression.

Finally, the symbol L ranges over a denumerable set L of labels. This set is defined to be disjoint with the sets V and C.

The expressions $[x]i$ are abstractions, and therefore the occurrences of x are bound in $[x]i$.

With $\langle i_1, L = i_2\rangle$ and $\langle i_1, L{:}i_2\rangle$ we denote (binary) record object and record type formation.

The expression $\mathit{let}\ x : i_1 = i_2\ \mathit{in}\ i$ makes it possible to abbreviate the proof object i_2 of type i_1 as x, which in turn may occur in what is defined as its valid scope, the expression i.

The effect of "using" the expression i_1 of type i_2 in the expression i is almost analogous to the one achieved by the Pascal command *with*, that is to say, all the fields that constitute the object i_1 are made directly available in the expression i.

From now on we use Greek letters α, α_1, ... for expressions intended to denote types and β, β_1, ... for families of types. We sometimes will use the more familiar notation $(x : \alpha)\alpha_1$ instead of $\alpha \to [x]\alpha_1$.

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

[1] At `http://www.fing.edu.uy/~gustun/SUBREC/get.html` it is available a gzipped tar directory containing the source code of the proof checker as well as the instructions for installing both the system and the interface

$$T : \mathtt{type} = \alpha$$
$$F(x : \alpha) : \mathtt{type} = \alpha_1$$
$$c(x_1 : \alpha_1, \ldots, x_n : \alpha_n) : \alpha = i$$

with T, F and c constant names, x,$x_1, \ldots, x_n$ variables and i, α and $\alpha_1, \ldots, \alpha_n$ 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 the constant F as the type family $[x]\alpha_1$ over the 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 $[x_1][x_2]\ldots[x_n]i$ of type $\alpha_1 \to [x_1](\alpha_2 \to \ldots (\alpha_n \to [x_n]\alpha)\ldots)$, with $n \geq 0$.

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. Representation of systems of algebras in type theory

We now consider the formalization of a piece of the theory of Boolean algebras in type theory extended with record types and subtyping. The definitions and propositions introduced in the next section are taken from [BS81] and [Grä71].

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:*

L1 :	$(\vee)$	$x \vee y \approx y \vee x$	
	$(\wedge)$	$x \wedge y \approx y \wedge x$	(commutative laws)
L2 :	$(\vee)$	$x \vee (y \vee z) \approx (x \vee y) \vee z$	
	$(\wedge)$	$x \wedge (y \wedge z) \approx (x \wedge y) \wedge z$	(associative laws)
L3 :	$(\vee)$	$x \vee x \approx x$	
	$(\wedge)$	$x \wedge x \approx x$	(idempotent laws)
L4 :	$(\vee)$	$x \approx x \vee (x \wedge y)$	
	$(\wedge)$	$x \approx x \wedge (x \vee y)$	(absorption laws)

As is well known, it is in the very nature of the above definition that any property Φ valid for all lattices 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 any 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 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 $\{f_1, \ldots, f_k\}$ shall be denoted by $\langle S, f_1, \ldots, f_k \rangle$. 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 the following laws,*

D1 : $x \wedge (y \vee z) \approx (x \wedge y) \vee (x \wedge z)$

D2 : $x \vee (y \wedge z) \approx (x \vee y) \wedge (x \vee z)$

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 $\langle B, \vee, \wedge, \sim, 0, 1 \rangle$ with two binary operations, one unary operation (called* complementation*), and two nullary operations which satisfies:*

B1 : $\langle B, \vee, \wedge \rangle$ *is a distributive lattice*

B2 : $(\vee)$ $x \vee 1 \approx 1$
$(\wedge)$ $x \wedge 0 \approx 0$

B3 : $(\vee)$ $x \vee \sim x \approx 1$
$(\wedge)$ $x \wedge \sim x \approx 0$

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 arbitrary 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.

The following proposition can easily be 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. □

It can readily be verified that using this latter proposition and the postulates B3, any Boolean algebra $\mathbf{B}$ satisfies that $\sim(\sim x) \approx x$, for all elements x of $|\mathbf{B}|$.

One more proposition is introduced before we turn 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 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 the following two propositions: $(x \vee y) \wedge (\sim x \wedge \sim y) \approx 0$ and $(x \vee y) \vee (\sim x \wedge \sim y) \approx 1$. Therefore, Proposition 3.3 can directly be applied to get that $\sim(x \vee y) \approx \sim x \wedge \sim y$. □

3.2. **Formalization.** We shall now proceed to give a formal account of the concepts in section 3.1. Thus, 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. Rather, we shall illustrate the use of the extended type theory in the representation of algebraic constructions. More accurately, what we here mean by type theory is a particular implementation of the system described in section 2.4.

3.2.1. *Preliminary definitions.* For the sake of readability we shall deviate a little from the syntax presented in section 2.4 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

$T : type$	$F : \alpha \to type$	$c : \alpha$
$T = \alpha$	$F\ x = \alpha_1$	$c = \mathsf{e}$

FIGURE 2. Forms of declaration

$$
\begin{array}{l}
binOp : Set \to type \\
binOp\ A = A \to A \to A \\
\\
Rel : Set \to type \\
Rel\ A = A \to A \to Set \\
\\
BinRel : type \\
BinRel = \langle \mathsf{A} : Set, \mathsf{R} : Rel\ \mathsf{A} \rangle \\
\\
RelOp : type \\
RelOp = \langle BinRel, \circ : binOp\ \mathsf{A} \rangle \\
\\
isTrans : BinRel \to type \\
isTrans\ B = (x, y, z : B.\mathsf{A})\ B.\mathsf{R}\ x\ y \to B.\mathsf{R}\ y\ z \to B.\mathsf{R}\ x\ z \\
\\
isCong : RelOp \to type \\
isCong\ Rop = \ \mathbf{use}\ Rop : RelOp \\
\qquad\qquad\quad \mathbf{in}\ (x, y, z, w : A)\ R\ x\ z \to R\ y\ w \to R\ (\circ\ x\ y)\ (\circ\ z\ w)
\end{array}
$$

FIGURE 3. Types and families of types

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 $c : \alpha$.

We consider now, in Figure 3, 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. 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 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 definition of *RelOp*, 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: for all elements x, y and z of A, if R 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.

3.2.2. *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

$isComm : RelOp \rightarrow type$
$isComm\ Rop = \boldsymbol{use}\ \ Rop : RelOp\ \ \boldsymbol{in}\ \ (x, y : A)\ R\ (\circ\ x\ y)\ (\circ\ y\ x)$

$isAssoc : RelOp \rightarrow type$
$isAssoc\ Rop = \boldsymbol{use}\ \ Rop : RelOp\ \ \boldsymbol{in}\ \ (x, y, z : A)\ R\ (\circ\ x\ (\circ\ y\ z))\ (\circ\ (\circ\ x\ y)\ z)$

$isIdemp : RelOp \rightarrow type$
$isIdemp\ Rop = \boldsymbol{use}\ \ Rop : RelOp\ \ \boldsymbol{in}\ \ (x : A)\ R\ (\circ\ x\ x)\ x$

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

$isAbsorb : RelOps \rightarrow type$
$isAbsorb\ Rops = \boldsymbol{use}\ \ Rops : RelOps\ \ \boldsymbol{in}\ \ (x, y : A)\ R\ x\ (\circ\ x\ (*\ x\ y))$

FIGURE 4. Axioms of lattices

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 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.

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, as *Pl* is 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, *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.

$$
\begin{array}{l}
PreLatt : type \\
PreLatt = \langle Setoid, \vee : binOp\ \mathsf{S}, \wedge : binOp\ \mathsf{S} \rangle \\
\\
dualPreLatt : PreLatt \rightarrow PreLatt \\
dualPreLatt\ Pl = \langle Pl, \vee = Pl.\wedge, \wedge = Pl.\vee \rangle \\
\\
opOfLatt : RelOps \rightarrow type \\
opOfLatt\ Rops = \ \langle\ \mathsf{cong} : isCong\ Rops, \\
\qquad\qquad\qquad\quad \mathsf{L1} : isComm\ Rops, \\
\qquad\qquad\qquad\quad \mathsf{L2} : isAssoc\ Rops, \\
\qquad\qquad\qquad\quad \mathsf{L3} : isIdemp\ Rops, \\
\qquad\qquad\qquad\quad \mathsf{L4} : isAbsorb\ Rops \\
\qquad\qquad\qquad \rangle \\
\\
Latt : type \\
Latt = \\
\quad \langle PreLatt, \\
\quad\ \vee\mathsf{Props} : opOfLatt\ \langle \mathsf{A} = \mathsf{S}, \mathsf{R} = \approx, \circ = \vee, * = \wedge \rangle, \\
\quad\ \wedge\mathsf{Props} : opOfLatt\ \langle \mathsf{A} = \mathsf{S}, \mathsf{R} = \approx, \circ = \wedge, * = \vee \rangle \\
\quad \rangle \\
\\
dualLatt : Latt \rightarrow Latt \\
dualLatt\ L = \ \langle dualPreLatt\ L, \\
\qquad\qquad\quad \vee\mathsf{Props} = L.\wedge\mathsf{Props}, \\
\qquad\qquad\quad \wedge\mathsf{Props} = L.\vee\mathsf{Props} \\
\qquad\qquad \rangle
\end{array}
$$

FIGURE 5. Lattice

We then introduce a family of record types *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, 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 $L.\vee\mathsf{Props.L1}$ is the proof that $L.\vee$ is commutative.

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 $\%-\quad formula\quad -\%$ to informally express the property being proved.

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

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

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

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

FIGURE 6. Congruences

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

FIGURE 7. Distributive lattice

The definitions of congR$\vee$ and congL$\wedge$ in Figure 6 illustrate the abbreviation of proof objects and the use of nested selection to access 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.

The type of distributive lattices is shown in Figure 7. We declare as well the function *dualDistrLatt*, which behaves as expected.

3.2.3. *Boolean Algebra.* The representation of the system of boolean algebras, the type *BoolAlg* in Figure 8, 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 $\vee$ and $\wedge$. We illustrate *use* expressions in the definition of the function *dualBoolAlg*.

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

In Figure 9 we first declare the proof of the first part of Proposition 3.2.

Let us now consider the second part of Proposition 3.2. We made, on page 6 of section 3.1, the remark on the duality of the properties enunciated in this latter proposition, and our intention of obtaining the proof of the dual of a given property

$$
\begin{array}{l}
PreBoolAlg : type \\
PreBoolAlg = \ \langle DistrLatt, \\
\qquad\qquad \sim : \mathsf{S} \to \mathsf{S}, \\
\qquad\qquad \mathsf{0} : \mathsf{S}, \\
\qquad\qquad \mathsf{1} : \mathsf{S} \ \rangle
\end{array}
$$

$$
\begin{array}{l}
dualPreBoolAlg : PreBoolAlg \to PreBoolAlg \\
dualPreBoolAlg\ Pba = \ \langle dualDistrLatt\ Pba, \\
\qquad\qquad\qquad\qquad \sim = Pba.\sim, \\
\qquad\qquad\qquad\qquad \mathsf{0} = Pba.\mathsf{1}, \\
\qquad\qquad\qquad\qquad \mathsf{1} = Pba.\mathsf{0} \ \rangle
\end{array}
$$

$$
\begin{array}{l}
BoolAlg : type \\
BoolAlg = \ \langle PreBoolAlg, \\
\qquad\qquad \mathsf{compCong} : (x, y : \mathsf{S}) \approx x\ y \to \approx \sim x\ \sim y, \\
\qquad\qquad \mathsf{B1} : (x : \mathsf{S}) \approx (\vee\ x\ \mathsf{1})\ \mathsf{1}, \\
\qquad\qquad \mathsf{B2} : (x : \mathsf{S}) \approx (\wedge\ x\ \mathsf{0})\ \mathsf{0}, \\
\qquad\qquad \mathsf{B3} : (x : \mathsf{S}) \approx (\vee\ x \sim x)\ \mathsf{1}, \\
\qquad\qquad \mathsf{B4} : (x : \mathsf{S}) \approx (\wedge\ x \sim x)\ \mathsf{0} \ \rangle
\end{array}
$$

$$
\begin{array}{l}
dualBoolAlg : BoolAlg \to BoolAlg \\
dualBoolAlg\ Ba = \ \boldsymbol{use}\ Ba : BoolAlg \\
\qquad\qquad\qquad\quad \boldsymbol{in}\ \ \langle dualPreBoolAlg\ Ba, \\
\qquad\qquad\qquad\qquad \mathsf{compCong} = \mathsf{compCong}, \\
\qquad\qquad\qquad\qquad \mathsf{B1} = \mathsf{B2}, \\
\qquad\qquad\qquad\qquad \mathsf{B2} = \mathsf{B1}, \\
\qquad\qquad\qquad\qquad \mathsf{B3} = \mathsf{B4}, \\
\qquad\qquad\qquad\qquad \mathsf{B4} = \mathsf{B3} \ \rangle
\end{array}
$$

FIGURE 8. Boolean algebra

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

$$
\begin{array}{l}
prop3.2(i) : \ (Ba : BoolAlg)\ (x, y : Ba.\mathsf{S}) \\
\qquad\qquad\quad \boldsymbol{use}\ \ Ba : PreLatt\ \boldsymbol{in}\ \approx (\wedge\ x\ y)\ \mathsf{0} \to \approx (\sim x)\ (\vee\ (\sim x)\ y)
\end{array}
$$

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

$$
\begin{array}{l}
prop3.2(ii) : \ (Ba : BoolAlg)\ (x, y : Ba.\mathsf{S}) \\
\qquad\qquad\quad \boldsymbol{use}\ \ Ba : PreBoolAlg\ \boldsymbol{in}\ \approx (\vee\ x\ y)\ \mathsf{1} \to \approx (\sim x)\ (\wedge\ (\sim x)\ y)
\end{array}
$$

$$prop3.2(ii) \ = \ [Ba]\, prop3.2(i)\ (dualBoolAlg\ Ba)$$

FIGURE 9. Proposition 3.2

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 proposition is constructed by applying —and with this

$prop3.3:\ (Ba : BoolAlg)\ (x, y : Ba.\mathsf{S})$
$\quad \boldsymbol{use}\ \ Ba : PreBoolAlg\ \boldsymbol{in}\ \approx (\wedge\ x\ y)\ \mathsf{0} \rightarrow \approx (\vee\ x\ y)\ \mathsf{1} \rightarrow \approx (\sim x)\ y$

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

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

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

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

FIGURE 10. Propositions 3.3 and 3.4

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

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

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

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

FIGURE 11. DeMorgan laws

we mean function application— the object *prop3.2(i)* to the dual structure of Ba, i.e. *dualBoolAlg* Ba. This construction is shown in that same figure.

In Figure 10 we declare the constants *prop3.3*, *prop3.4(i)* and *prop3.4(ii)* to stand for the proofs of Proposition 3.3 and the two properties of Proposition 3.4. The construction of those proofs is routine.

We end up presenting in Figure 11 the proof objects corresponding to 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

We have motivated the use of an extension of Martin-Löf's logical framework with dependent record types and subtyping for the formalization of algebraic constructions. Dependent record types have been illustrated to constitute an appropriate mechanism for the representation of types of algebraic structures. In addition, the inclusion relation induced by record types allows to represent in a direct manner incremental definition of types of structures. Moreover, the subtyping mechanism made it possible to give a formal account of the fact that a system that conforms to an extension of one previously introduced inherits the constructions associated to the latter.

The experiment reported in this work was mechanically verified using the implemented proof checker. We have, in addition, used the system for developing formalizations concerning Group theory and the algebraic theory of Integral Domains [Bet98].

The formalization of abstract algebra in type theory (in a wide sense) has lately received an increasing amount of attention.

In [Acz94, Acz95], Aczel presents a notion of class and overloaded definitions for predicative type theories. The motivations behind this proposal are mainly concerned with the development of mathematical abstractions for the formalization of algebra in type theory. The key notion that there arises is that of a system of algebras. A crucial condition required from these systems is that they should determine the *type* of algebras of the system. In accordance to this, thus, algebras should be first class objects of the formal language. Furthermore, in order to naturally reflect the usual presentation of these notions in the informal language, it should be feasible, on the one hand, for the systems to be defined in an incremental manner. On the other hand, it is also desirable to be able to reuse notation introduced for a given system S when it comes to consider a system T which has been defined as an extension of it. In other words, T should *inherit* the proof constructions, for instance, developed for S. In this work, the notion of system of algebra is identified with that of a *class* for which *methods* can be defined that in turn may be reused (*overloaded*) on elements of subclasses of the one for which they have been originally defined.

In [Bar95], the ideas above are extended to consider in uniform way the notion that two types are somewhat related in such a way that one can be considered a subtype of the other. This relation is formally reflected by introducing a *coercion* function that indicates how to get an object of the supertype out of one of the subtype. But a mechanism, which is formulated for pure type systems, is introduced that allows to leave the coercions implicit. Applications are then shown using the extended calculus of constructions [Luo94], where the representation of systems of algebras is formulated in terms of Σ types. The relation between types of algebraic structures that we obtain in terms of record inclusion is partially achieved in terms of (the transitive closure of) coercions.

Direct successors of this work are the mechanisms implemented by Bailey [Bai97] and Saïbi [Saï97] for defining coercions between types or classes of types developed for the proof-assistants *LEGO* [Pol94a] and *Coq* [Bar97], respectively. They have also formalized corresponding large-scale case studies on Galois theory and Category theory.

In [Jac95] algebraic structures are formalized in Nuprl's version of type theory [Con86] using sets of unlabeled dependent pairs and subsets. No general solution is given in this work to the problem of representing the inclusion of types of structures that we have been considering.

In [Luo96], 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 concept of a kind K being a principal kind of an object k and that of (proper) kind inclusion. The meaning explanation of the relation of 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 we have 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

[Acz94] P. Aczel. A Notion of Class for Theory Development in Algebra (in a Predicative type theory), 1994. Presented at Workshop of Types for Proofs and Programs, Båstad, Sweden.

[Acz95] P. Aczel. Simple Overloading for Type Theories, 1995. Privately circulated notes.

[Aug97] L. Augustsson. *HBC - The Chalmers Haskell Compiler.* `http://www.cs.chalmers.se/~augustss/hbc/`, 1997.

[Bai97] A. Bailey. Lego with implicit coercions, 1997. Documentation report, available at `ftp.cs.man.ac.uk/pub/baileya/Coercions`.

[Bar92] H. Barendregt. Lambda Calculi with Types. In T. S. E. Maibaum D. M. Gabbay, S. Abramsky, editor, *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.

[Bar95] G. Barthe. Implicit coercions in type systems. In *Selected Papers from the International Workshop TYPES '95, Torino, Italy, LNCS 1158.*, 1995.

[Bar97] B. Barras et al. The Coq Proof Assistant Reference Manual – Version V6.1. Technical Report 0203, INRIA, 1997.

[Bet97] G. Betarte. Dependent record types, subtyping and proof reutilization. In *Online Proceedings of the TYPES Workshop* Subtyping, Inheritance and Modular Development of Proofs, Durham, England, September 1997.

[Bet98] G. Betarte. *Dependent Record Types and Algebraic Structures in Type Theory.* PhD thesis, PMG, Dept. of Computing Science, University of Göteborg and Chalmers University of Technology, 1998.

[Bet00] G. Betarte. Type Checking Dependent (Record) Types and Subtyping. *Journal of Functional Programming*, 10(2):137–166, March 2000.

[BS81] S. Burris and H.P. Sankappanavar. *A Course in Universal Algebra.* Graduate Texts in Mathematics, Springer-Verlag, 1981.

[BT98] G. Betarte and A. Tasistro. *Extension of Martin-Löf's Type Theory with Record Types and Subtyping*, pages 21–39. In Sambin and Smith [SS98], 1998.

[CNSvS94] Th. Coquand, B. Nordström, J.M. Smith, and B. von Sydow. Type theory and programming. In *EATCS 52*, 1994.

[Con86] R. Constable et al. *Implementing mathematics with the Nuprl development sys tem.* Prentice-Hall, 1986.

[Coq91] Th. Coquand. An algorithm for testing conversion in type theory. In *Logical Frameworks, Huet G., Plotkin G. (eds.)*, pages 71–92. Cambridge University Press, 1991.

[Grä71] G. Grätzer. *Lattice Theory. First concepts and Distributive Lattices*. W. H. Freeman and Company, 1971.

[Jac95] P. Jackson. *Enhancing the Nuprl Proof Development System and Applying it to Computational Abstract Algebra*. PhD thesis, Cornell University, 1995.

[Luo94] Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Number 11 in International Series of Monographs on Computer Science. Oxford University Press, 1994.

[Luo96] Z. Luo. Coercive subtyping in type theory. In *CSL'96, the 1996 Annual Conference of the European Association for Computer Science Logic*, Utrech, 1996.

[Mar87] P. Martin-Löf. Philosophical Implications of Type Theory., 1987. Lectures given at the Facoltá de Lettere e Filosofia, Universitá degli Studi di Firenze, Florence, March 15th. - May 15th. Privately circulated notes.

[Mar92] P. Martin-Löf. Substitution calculus., 1992. Talks given in Göteborg.

[NPS89] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory.An Introduction*. Oxford University Press, 1989.

[Pet96] J. Peterson et al. *Report on the Programming Language HASKELL. A Non-strict, Purely Functional Language*, May 1996.

[Pol94a] R. Pollack. *The Theory of LEGO: a proof checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1994.

[Pol94b] R. Pollak. Closure under alpha-conversion. In *Types for Proofs and Programs:International Workshop TYPES'93, Nijmegen, May 1993, Selected Papers, volume 806 of LNCS*, 1994.

[Saï97] A. Saïbi. Typing algorithm in type theory with inheritance. In *24th. Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997.

[SS98] G. Sambin and J.M. Smith, editors. *Twenty-Five Years of Constructive Type Theory*. Oxford University Press, 1998.

[Tas93] A. Tasistro. Formulation of Martin-Löf's theory of types with explicit substitution, 1993. Licenciate thesis.Programming Methodology Group, Dept. of Computer Science, University of Göteborg and Chalmers University of Technology.

[Tas97] A. Tasistro. *Substitution, record types and subtyping in type theory, with applications to the theory of programming*. PhD thesis, PMG, Dept. of Computing Science, University of Göteborg and Chalmers University of Technology, 1997.

E-mail address: `gustun@fing.edu.uy`