

Extension of Martin-Löf's Type Theory with Record Types and Subtyping

Gustavo Betarte and Alvaro Tasistro*

Department of Computing Science

Chalmers University of Technology and University of Gothenburg.

E-mail: gustun@cs.chalmers.se, tato@fing.edu.uy.

1 Introduction.

Our starting point, to which we refer hereafter as type theory, is the formulation of Martin-Löf's set theory using the theory of types as logical framework (Martin-Löf 1987; Nordström *et al.* 1990). The question that we address is that of the representation of systems of structures such as algebraic systems or abstract data types. In order to provide general means to this end, we extend type theory with a new mechanism of type formation, namely that of *dependent record types*. This permits to form types of tuples in such a manner as to allow any arbitrary set —i.e. not restricted to be among those generated by a fixed repertoire of set forming operations— to be used as a component of tuples of those types. Such types of tuples cannot be formed in the original theory.

Moreover, as it is well known from the theory of programming languages, a natural notion of inclusion arises between record types. Given two record types ρ and ρ' , if ρ contains every label declared in ρ' (and possibly more) and the types of the common labels are in the inclusion relation then ρ is included in ρ' , in symbols $\rho \sqsubseteq \rho'$. This is justified because then every object of type ρ is also an object of type ρ' , since it contains components of appropriate types for all the fields specified in ρ' . Our extension contains the form of judgement $\alpha \sqsubseteq \beta$ expressing that the type α is included in the type β and corresponding proof rules, which generalize record type inclusion to dependent record types and propagate it to the rest of the types of the language. In the present formulation, no proper inclusion between ground types is allowed.

Having type inclusion represents a considerable advantage for the formalization of the types of structures in which we are interested. In particular, systems of algebras will be represented as record types and, according to the subtyping rule explained above, any algebraic system obtained by enriching another with additional structure will be a subtype of the original system. Thus, for instance, we can directly express in the formalism that every group is a monoid. As a

* Current address: Instituto de Computación, Facultad de Ingeniería.
Julio Herrera y Reissig 565, Piso 5. Montevideo, Uruguay.

consequence, every function defined on monoids can be applied to any group, which implies that any proof of a property of monoids is itself a proof of exactly the same property of groups. This is precisely the principle that we use in informal reasoning, which is then formalized without further encoding.

The extension preserves the decidability of the formal correctness of judgements, which is a fundamental property of the original theory. Hence, type checking can still be used for verifying the validity of proofs of theorems and the correctness of programs with respect to specifications. A type checking algorithm for the extended theory is exposed in (Tasistro 1997; Betarte 1997).

The structure of the paper is as follows. In the next section we develop in detail the motivations for the extension that have been outlined above by considering a case of formalization of Algebra. It consists of the proof of a basic property of groups, namely that the right identity of the operation of the group is unique. In an appendix we present the complete code of this formalization as verified by a prototype implementation of the type checking algorithm for the extended theory.

In section 3 we present the formal stipulation of the extension. To explain the resulting calculus we proceed according to the syntactico-semantical method exposed in (Martin-Löf 1984) and used in every presentation of Martin-Löf's type theory to which we refer in this work. We introduce the forms of judgements of the extended theory, which are those of the original theory plus four new ones, and explain them semantically. Then we present the whole system of formal rules of inference. Each rule can be justified by showing that the meaning of the conclusion follows from those of the premisses. We give detailed justifications of the most important rules. As for the other rules, their justifications are either immediate or can easily be obtained from the ones that we give.

Finally, in the last section of the paper, we give some conclusions and discuss related work.

2 Formalization of algebraic systems.

2.1 Record Types.

In order to motivate the introduction of record types in type theory we consider the problem of formalizing a proof of a basic property of groups, namely that the right identity for the operation of a group is unique. We will get to the formal definition of group after a process of successively enriching previously defined systems of algebras with further structure. This procedure is common practice in algebra and could be called *incremental definition* of the algebraic systems. A convenient starting point is the notion of a set with an equivalence relation on it, which has elsewhere been called *setoid*. 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 $=$ to be given explicitly as a component of the system being defined. Just for the sake of presentation we shall consider setoids as constructed from a still simpler

notion, namely that of a set with a binary relation on it.

Now, the representation in type theory of systems of algebraic structures presents a difficulty. Such structures are defined as tuples in informal language. So to represent them in type theory we ought to use tuple types. In view of the mechanisms of type formation available, the only way to get tuple types is to introduce *sets* of tuples. But consider now setoids as defined above, i.e. structures composed out of a set X together with an equivalence relation on X . If X can be any set then the type of setoids cannot itself be a set or it would be allowed to form a part of some of its own elements.

Another possibility would be to restrict X in the setoid (X, \sim) to be an element of a previously constructed set of sets that we call a *universe*. This particular type of setoids could be introduced as a set, obviously then not belonging to the universe. Now, if we still want to allow *any* set to be possibly used as component of setoids, then the present approach leads us to the requirement that every set is a member of a certain universe. In such case, the general notion of a setoid would be split in the formalization into several types that could be called of *setoids over U* for each universe U of sets. But, actually, there would be no way to introduce all these types of setoids once and for all unless we know each individual universe U , since there is no such thing as the type of the universes of sets. And in turn, to give rules for forming all possible universes of sets implies to fix once and for all the possible ways to form sets, since each universe must be defined inductively. It would then still be possible to introduce set valued functions, i.e. set operators or predicates and relations on given sets. But each of these would have to be defined in terms of the fixed primitive mechanisms of set formation. They would in general, then, have to be introduced as recursively defined functions giving values in a certain universe of sets.

Besides the one above, there is also the understanding of type theory according to which the notion of a set is indeed an open notion, in the sense that it is at any time possible to introduce new primitive set formers. Further, the version of type theory that we are considering allows to do this in a simple way, i.e. just by declaring primitive constants corresponding to the set former itself and its constructors and defined constants corresponding to the recursion operators. Type theory is in this way understood as a basic framework in which it is possible to express various other theories. Each of these starts simply as a vocabulary of constants and definitions that is determined by one or more set formers in the way explained above. Then it is extended by further definitions corresponding to the various individual theorems. According to this approach there is simply no question of knowing or enumerating all the individual set formers. Therefore, to obtain in this case a formalization of the notion of setoid that allows any set to be a component of setoids one has no alternative but to state this latter condition explicitly. In general, then, one needs types of tuples some of whose components are allowed to be *arbitrary* sets —not just members of sets of sets. As explained above, such a type cannot itself be a set.

This provides a first motivation for introducing the *dependent record types* as a new mechanism of type formation. Dependent record types are just sequences of *fields* in which *labels* are declared as of certain types:

$$\langle L_1 : \alpha_1, \dots, L_n : \alpha_n \rangle.$$

In dependent record types, the type α_i may depend on the preceding labels L_1, \dots, L_{i-1} . In the notation that we are going to use in this section, 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.

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

$$\langle S : Set, R : S \rightarrow S \rightarrow Set \rangle.$$

Let us call this type B . Components of objects of a record type are accessed by *selection* of the labels of the record type in question, which we write using the usual dot notation. Then if r is of type B , $r.S$ is a set and $r.R$ is a binary relation on $r.S$.

Record objects are constructed as sequences of fields that are assignments of objects of appropriate types to labels. For instance, if N is the set of natural numbers and \leq the usual order relation on N , then the following is an object of type B :

$$\langle S = N, R = \leq \rangle.$$

That two objects r and s of type $\langle L_1 : \alpha_1, \dots, L_n : \alpha_n \rangle$ 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.

Now we can show how the example at hand can be formalized in type theory with record types.

2.2 Setoids.

We begin by introducing the type of binary relations on given set:

$$\begin{aligned} B &: type \\ B &= \langle S : Set, R : S \rightarrow S \rightarrow Set \rangle. \end{aligned}$$

From this definition we obtain that of a setoid by adding properties of the binary relation. Thereby we have a first example of incremental definition of systems. In the present extension of type theory, the possibility of incremental definition is given directly by the rules of formation of record types. Formally, record types are constructed up from the record type with no fields by iterating the operation of extension of a record type with one more field. Then setoids can be introduced as follows:

Setoid : type
Setoid = $\langle B,$
 $\text{ref} : (x : S) \text{R}xx,$
 $\text{symm} : (x, y : S) \text{R}xy \rightarrow \text{R}yx,$
 $\text{trans} : (x, y, z : S) \text{R}xy \rightarrow \text{R}yz \rightarrow \text{R}xz \rangle.$

Now we can prove a very simple property which is a consequence of the axioms for setoids. Stated informally it is:

Proposition. For any objects x, y, z, w of a setoid, if $x = y, x = z$ and $y = w$ then $z = w$.

In plain words, this proposition could be called of replacement of equivalents by equivalents. It allows to simultaneously replace both sides of an equation $x = y$ by objects z and w that are, respectively, equivalent to x and y . Now the formal statement will not use $=$ but the equivalence relation explicitly given with the setoid. It is proved by a function:

$\text{eqtoeq} : (S : \text{Setoid}; x, y, z, w : S.S) S.\text{R} x y \rightarrow S.\text{R} x z \rightarrow S.\text{R} y w \rightarrow S.\text{R} z w$

whose definition can be found in the appendix.

2.3 Subtyping.

The formalization continues by introducing several types of algebraic structures by the procedure of incremental definition. Thereby, we get first the definitions of *semigroup* and *monoid*, which can be found in the appendix. Eventually, we formulate the definition of *group*, in the following way:

Group : type
Group = $\langle \text{Monoid},$
 $\text{inv} : (x : S) S,$
 $\text{invcong} : (x, y : S) \text{R}xy \rightarrow \text{R}(\text{inv } x) (\text{inv } y),$
 $\text{invprop} : (x : S) \text{R}(\text{op } x (\text{inv } x)) \text{e} \rangle.$

The informal counterpart to this definition says that groups are monoids with some additional structure, namely the inverse operation. Then every group is a monoid, which is of course straightforwardly used in the informal language. A direct expression of this use in type theory requires that every object of the type *Group* is itself also an object of the type *Monoid*, i.e. a form of polymorphism. Now, both *Group* and *Monoid* are record types and, as it happens, the required form of polymorphism is natural for record objects.

The source of the idea can be traced back at least to (Dahl and Nygaard 1966) and consists in observing that, given a record type ρ it is possible in general to drop and permute fields of ρ and still get a record type ρ' . Moreover, then any object of type ρ satisfies also the requirements imposed by the type ρ' . That is, given $r : \rho$, we are justified in asserting also $r : \rho'$. This is so because what is required to make the latter judgement is that the selections of the labels declared in ρ' from r are defined as objects of the appropriate types. And we have this, since every label declared in ρ' is also declared in ρ and with the same type.

To realize this idea in full generality in the formal language we introduce the form of judgement $\alpha_1 \sqsubseteq \alpha_2$ for types α_1 and α_2 , which is to be read: α_1 is a *subtype* of α_2 . This means that every object of α_1 is also an object of α_2 and that equal objects of α_1 are equal objects of α_2 . We also refer to these judgements as of *type inclusion*. In the case of record types, the condition for $\rho_1 \sqsubseteq \rho_2$ is in words as follows: for each field $L : \alpha_2$ in ρ_2 there must be a field $L : \alpha_1$ in ρ_1 with $\alpha_1 \sqsubseteq \alpha_2$. The formal stipulation of this rule takes care of the dependence of the types occurring in fields on labels, as will be shown in the next section. It also requires that rules of subtyping are given for all the type formers of the language. In the present formulation, this is done by propagating type inclusion to the functional types in the standard manner and without allowing proper inclusion between ground types.

2.4 Unicity of the identity element of a group.

Now we can consider the property of groups in which we were interested from the beginning, namely that the identity element of a group is unique. To prove this, we make use of the following:

Lemma. Let (S, \circ) be a group in which e is a right identity. If x is any element of S such that $x \circ x = x$ then $x = e$.

This is proved as follows. First, see that $(x \circ x) \circ x^{-1} = x \circ e$, using associativity and the property of the right inverse. Further, $(x \circ x) \circ x^{-1} = x \circ x^{-1}$, since $x \circ x = x$ by assumption. So, $x \circ e = x \circ x^{-1}$. But now, we have also that $x \circ e = x$, by the right identity property, and that $x \circ x^{-1} = e$ by the property of the right inverse. So, replacing equals by equals we finally have that $x = e$.

Consider now, for instance, the last step of the proof. There we use the rule of replacement which was proved for setoids. That is, we are using that every group is itself a setoid, which is of course correct. In the formalization of this proof we will then have an application of subtyping. More precisely, the proof *eqtoeq* which was introduced earlier as a function on *Setoid* will be applied to a group. The application is correct because every group has also the type *Setoid* since *Group* \sqsubseteq *Setoid*. Actually, every step in which a property of the equivalence relation associated to the group is used gives rise to an application of subtyping.

Finally, to prove the unicity of e we assume that there exists e_1 in S such that for all element x of S , $x \circ e_1 = x$. In particular, $e_1 \circ e_1 = e_1$. Then by the lemma above $e_1 = e$. The formal proofs are displayed and explained in the appendix.

2.5 Multiple Inheritance.

As a final point, consider the definition of *abelian monoid* as a subtype of the system *monoid*:

$$\begin{aligned} AbMonoid &: type \\ AbMonoid &= \langle Monoid, comm : (x, y : S) R(op\ x\ y)(op\ y\ x) \rangle. \end{aligned}$$

By an analogous procedure, we define *abelian group* as a subtype of *group*:

$$\begin{aligned} AbGroup &: type \\ AbGroup &= \langle Group, comm : (x, y : S) R(op\ x\ y)(op\ y\ x) \rangle. \end{aligned}$$

Of course, abelian groups could also have been introduced as extending abelian monoids, i.e. just in the same way as groups extend monoids. Now, because of the rules of inclusion of record types, the definition above gives us $AbGroup \sqsubseteq AbMonoid$ anyway. A first observation is then that a type may be a subtype of several other types each of which needs not be in the inclusion relation with any of the others. When we use record types to represent systems of algebras, this provides a direct formalization of the principle that a system may inherit properties and proofs from several other systems, themselves defined independently of each other.

3 Formulation of the extension.

We now proceed to give the formal stipulation of the extended theory. We will follow the syntactico-semantical method exposed in (Martin-Löf 1984) and used in every presentation of Martin-Löf's type theory to which we refer in this work. Therefore the first step is to introduce the various forms of judgement of the theory. This is done by exhibiting their syntax and at the same time explaining them semantically, i.e. stating what it is that has to be known in order to assert a judgement of each of the forms in question. In the extended theory, four new forms of judgement are added to those of the original theory. After having introduced the forms of judgement, we set up a system of formal rules of inference. Each individual rule is to be justified by showing that the meaning of the conclusion follows from those of the premisses.

3.1 The forms of judgement.

3.1.1 *The original forms of judgement.*

Let us recall the forms of categorical judgement of type theory:

$$\begin{array}{ll} \alpha : type & \alpha_1 = \alpha_2 \\ \beta : \alpha \rightarrow type & \beta_1 = \beta_2 : \alpha \rightarrow type \\ a : \alpha & a = b : \alpha. \end{array}$$

To know that $\alpha : type$ is to know what it means to be an object of type α as well as what it means for two objects of type α to be the same. That a is an object of type α is written $a : \alpha$. Given $a : \alpha$ and $b : \alpha$, that they are the same

object of type α is written $a=b : \alpha$.

That two types α_1, α_2 are the same—in symbols $\alpha_1=\alpha_2$ —means that to be an object of type α_1 is the same as to be an object of type α_2 and to be the same object of type α_1 is the same as to be the same object of type α_2 .

That β is a family of types over the type α means that for any $a : \alpha$, βa is a type and that for any two objects a, b of type α such that $a=b : \alpha$, βa and βb are the same type. Given type α , that β is a family of types over α is written $\beta : \alpha \rightarrow \text{type}$.

That two families of types β_1 and β_2 over a type α are the same—in symbols $\beta_1=\beta_2 : \alpha \rightarrow \text{type}$ —means that $\beta_1 a = \beta_2 a$ for any $a : \alpha$.

The present notion of a family of types was introduced in the formulation of the calculus of substitutions for type theory (Martin-Löf 1992; Tasistro 1997). It makes it possible to have abstraction as a uniform mechanism of variable binding in the language.

The forms of judgements above are generalized to forms of relative judgements, i.e. of judgements depending on variables $x_1:\alpha_1, \dots, x_n:\alpha_n$. For the sake of brevity, here we consider this as done in the way it was usual in the formulations of type theory prior to the calculus of substitutions, i.e. in for instance (Martin-Löf 1984; Nordström *et al.* 1990).

It may be useful to remark that we make (nominal) definitions of types and families of types in addition to those of objects of the various types which are ordinary in type theory. An (explicit) definition of a type is as follows. Let α be a type and A a name not previously given any meaning. Then we define A as the type α by stating the two axioms:

$$\begin{aligned} A &: \text{type} \\ A &= \alpha. \end{aligned}$$

Then as a consequence of the second axiom, $a : A$ and $a=b : A$ have identical meaning as $a : \alpha$ and $a=b : \alpha$, respectively. We say that A is the *definiendum* and α the *definiens* of the definition. We shall also say that A has α as its definiens.

Definitions of families of types are explained similarly. Let F be a name not yet given any meaning, α a type and α_1 a type depending on variable x of type α . Then we define F as a family of types over α by means of the following two axioms:

$$\begin{aligned} F &: \alpha \rightarrow \text{type} \\ Fx &= \alpha_1 [x:\alpha]. \end{aligned}$$

The second axiom is a relative judgement depending on variable x of type α . Then by virtue of it, Fa turns out to be by definition the type $\alpha_1(x := a)$ for $a : \alpha$ ¹.

¹In definitions of the present form, the dependence of α_1 on x must be uniform. That is to say, families of types cannot be defined by case analysis of the argument.

3.1.2 Judgements of inclusion.

We have now to introduce some new forms of judgement. We consider first those for expressing inclusion of types and of families of types on a given type:

$$\alpha_1 \sqsubseteq \alpha_2 \qquad \beta_1 \sqsubseteq \beta_2 : \alpha \rightarrow \text{type}.$$

Given types α_1 and α_2 , that α_1 is a *subtype* of α_2 —in symbols $\alpha_1 \sqsubseteq \alpha_2$ — means that every object of type α_1 is also an object of type α_2 and equal objects of type α_1 are equal objects of type α_2 .

Given a type α and families β_1 and β_2 over α , that β_1 is a *subfamily* of β_2 —in symbols $\beta_1 \sqsubseteq \beta_2 : \alpha \rightarrow \text{type}$ — means that $\beta_1 a \sqsubseteq \beta_2 a$ for every object a of type α .

3.1.3 Record types and families of record types.

We intend to introduce a new type former, namely that of record types. In principle, all that we would have to do for that is to formulate a number of rules. But in the present case something else has to be considered first. Record types are constructed as lists of fields. We formalize this as it is usual with lists, i.e. from the record type with no fields, by means of an operation of extension of a record type with a further field. And then, as has just been said, the operation of extension must require that what is to be extended is indeed a *record* type. We will express this condition by means of a further form of judgement. This, in turn, is most simply explained as being about types. That is, for type ρ we will have the judgement that ρ is a record type —in symbols, $\rho : \text{record-type}$. Similarly, we need to distinguish *families of record types* on a type α since they give rise to record types when applied to appropriate objects. Therefore we will have also the judgements $\sigma : \alpha \rightarrow \text{record-type}$ for σ a family of types over α . These two new forms of judgement are now to be explained.

For explaining what it is for a type to be a record type we have to distinguish between defined and primitive types. A defined type is a record type if its definiens is a record type. A primitive type is a record type if it is generated by the rules referred to above, namely:

$\langle \rangle$ is a record type.

If ρ is a record type and β a family of types over ρ , then $\langle \rho, L : \beta \rangle$ is a record type, provided L is not already declared in ρ .

We will later justify rules to the effect that there are indeed types generated by the clauses above. 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 will be later 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 given set is formally written. Families of types are formed by abstraction, which we write using square brackets:

$$\langle\langle\rangle, S : [r]Set\rangle, R : [r](r.S)(r.S)Set\rangle.$$

There is a direct way of translating the notation used in the previous section into the present formal notation.

We conclude by explaining what a family of record types is. Given type α and $\sigma : \alpha \rightarrow type$, that $\sigma : \alpha \rightarrow record\text{-}type$ means that σa is a record type for arbitrary $a : \alpha$.

The forms of judgements introduced are all categorical. From now on we consider their generalizations to forms of relative judgements as given in the way indicated at the beginning of this section.

3.2 Inference Rules.

We will now formulate a system of inference rules involving the preceding forms of relative judgement. The rules will be written as of natural deduction, i.e. only the discharged variables will be mentioned. In principle, the rules ought to have enough premisses for them to be completely formal and thereby make it possible to justify each rule individually using only the explanations of the various forms of judgement. We will, for conciseness, often omit premisses. A general principle allowing to recover the omitted premisses of a rule is that they are just those strictly necessary for guaranteeing that every (explicit) premiss and the conclusion of the rule are well formed as instances of the respective forms of judgement. Also, we allow ourselves to mention side conditions to rules. These are of two simple forms, each of them of a purely syntactic nature. We give detailed explanations of rules in the cases in which we think it could be relevant. The entire system corresponding to the extended theory that we are presenting is obtained by adding to the rules below the rule of assumption and the various substitution rules, which are just the same as those of the original theory.

3.2.1 General rules of equality and inclusion.

To begin with, we have that the various equality judgements give rise to equivalence relations. That is, we have rules of:

Reflexivity, symmetry and transitivity of identity of types, identity of objects of a given type and identity of families of types over a given type.

Next we have rules expressing that inclusion follows from identity:

$$\frac{\alpha_1 = \alpha_2}{\alpha_1 \sqsubseteq \alpha_2} \qquad \frac{\beta_1 = \beta_2 : \alpha \rightarrow type}{\beta_1 \sqsubseteq \beta_2 : \alpha \rightarrow type}.$$

Using these two rules it is possible to derive those of *reflexivity of type inclusion and of inclusion of type families*. We also have:

Transitivity of type inclusion and of inclusion of type families.

The following are the rules of *type subsumption*. They are justified immediately in virtue of the explanations of the judgements of inclusion.

$$\frac{a : \alpha_2 \quad \alpha_2 \sqsubseteq \alpha_1}{a : \alpha_1} \qquad \frac{a=b : \alpha_2 \quad \alpha_2 \sqsubseteq \alpha_1}{a=b : \alpha_1}$$

$$\frac{\alpha_1 \sqsubseteq \alpha_2 \quad \beta : \alpha_2 \rightarrow type}{\beta : \alpha_1 \rightarrow type}$$

$$\frac{\alpha_1 \sqsubseteq \alpha_2 \quad \beta_1 = \beta_2 : \alpha_2 \rightarrow type}{\beta_1 = \beta_2 : \alpha_1 \rightarrow type}$$

$$\frac{\alpha_1 \sqsubseteq \alpha_2 \quad \beta_1 \sqsubseteq \beta_2 : \alpha_2 \rightarrow type}{\beta_1 \sqsubseteq \beta_2 : \alpha_1 \rightarrow type}.$$

3.2.2 Remarks.

A number of comments about the preceding rules are now in place. Let us first consider the rules of type subsumption. They replace those called of *type conversion* in the original theory, i.e. for instance the rule:

$$\frac{a : \alpha_2 \quad \alpha_2 = \alpha_1}{a : \alpha_1}.$$

The rules of type conversion can actually be derived from those of type subsumption using the rules expressing that inclusion follows from identity. In the original theory, the rule of type conversion displayed above expresses the part played by definitional identity in the formation of objects of the various types. It is then the formal counterpart of the use of definitions in proofs of theorems. The link between definitional identity and formation of objects obviously subsists in the extended theory, since the rule of type conversion is derivable. On the other hand, the mechanisms of formation of types and objects are in principle generalized by the presence of type inclusion and the rules of type subsumption. That is: the rules for forming types and objects of the various types in the original theory are the following. There is first a rule for each of the various syntactic forms of the theory that states the conditions under which an expression of the form in question denotes or has a type. To these, we have to add the rules of substitution in types and objects. And, finally, there is the rule of type conversion. Exactly the same will be the case for the extended theory, with the rule of type subsumption taking the part of the rule of type conversion.

As another point, notice that we have not given rules to the effect that identity of types and of families of types are equivalent to the respective mutual inclusions. That is, the rules:

$$\frac{\alpha_1 \sqsubseteq \alpha_2 \quad \alpha_2 \sqsubseteq \alpha_1}{\alpha_1 = \alpha_2} \qquad \frac{\beta_1 \sqsubseteq \beta_2 : \alpha \rightarrow type \quad \beta_2 \sqsubseteq \beta_1 : \alpha \rightarrow type}{\beta_1 = \beta_2 : \alpha \rightarrow type}.$$

Now, consider the first of these rules. For justifying it, we ought to have that the two premisses together constituted precisely the meaning of the conclusion. That is, identity of types ought to have been defined as the mutual inclusion of the types in question. This has, however, not been made explicit in our explanations. Defining type identity as mutual inclusion can be defended on the grounds that type inclusion should be understood as intensional, i.e. as having to follow generically from the explanations of what an object is and what identical objects are of the types in question. Then the mutual inclusion of two types α_1 and α_2 would be nothing other than the identity of meaning of $a : \alpha_1$ and $a : \alpha_2$ as well as of the corresponding judgements of identity of objects. That is, it would just coincide with the identity of the two types.

So we have two alternatives here. The corresponding formal systems will differ only as to the presence of the rules above and therefore as to the judgements of the forms $\alpha_1 = \alpha_2$ and $\beta_1 = \beta_2 : \alpha \rightarrow type$ that are derivable. But they will not differ as to the judgements of the forms $\alpha : type$ and $a : \alpha$ that can be derived. This follows from the observation made above about the rules available for making typing judgements and the fact that, clearly, exactly the same judgements of type inclusion can be derived in both systems. We shall consider the theory in which identity of types is not identified with mutual inclusion, which turns then out to be expressive enough for representing (informal) theorems in spite of its weakness in connection to the judgements of identity of types and of families of types that can be proved.

3.2.3 Families of types and function types.

Now we give the rules for using and forming families of types. First come the rules of application, which just express the definition of the notion of family of types:

$$\frac{\beta : \alpha \rightarrow type \quad a : \alpha}{\beta a : type} \qquad \frac{\beta : \alpha \rightarrow type \quad a = b : \alpha}{\beta a = \beta b.}$$

Similarly, the following express the meaning of identity and inclusion of type families:

$$\frac{\beta_1 = \beta_2 : \alpha \rightarrow type \quad a : \alpha}{\beta_1 a = \beta_2 a} \qquad \frac{\beta_1 \sqsubseteq \beta_2 : \alpha \rightarrow type \quad a : \alpha}{\beta_1 a \sqsubseteq \beta_2 a.}$$

Families of types can be formed by abstraction, which is defined by the β -rule. We have a rule of extensionality that is immediately justified from the explanation of what it is for two families of types to be the same:

$$\frac{\alpha_1 : type \ [x:\alpha]}{[x]\alpha_1 : \alpha \rightarrow type}$$

$$\frac{\alpha_1 : type \ [x:\alpha] \quad a : \alpha}{([x]\alpha_1)a = \alpha_1(x := a)} \qquad \frac{\beta_1 x = \beta_2 x [x:\alpha]}{\beta_1 = \beta_2 : \alpha \rightarrow type.}$$

We now introduce the function types. These are explained in the obvious way. We give the rules for proving identity and inclusion of two function types:

$$\frac{\alpha : type \quad \beta : \alpha \rightarrow type}{\alpha \rightarrow \beta : type}$$

$$\frac{\alpha_1 = \alpha_2 \quad \beta_1 = \beta_2 : \alpha_1 \rightarrow type}{\alpha_1 \rightarrow \beta_1 = \alpha_2 \rightarrow \beta_2} \qquad \frac{\alpha_2 \sqsubseteq \alpha_1 \quad \beta_1 \sqsubseteq \beta_2 : \alpha_2 \rightarrow type}{\alpha_1 \rightarrow \beta_1 \sqsubseteq \alpha_2 \rightarrow \beta_2}.$$

By virtue of the first rule we have that $\alpha \rightarrow [x]\alpha'$ is a type if α' is a type depending on $x:\alpha$. This type is usually written $(x:\alpha) \alpha'$. We explain the rule of inclusion of function types. The explanation reduces eventually to that of the case in which the judgements involved are categorical. So we consider only this case. The same will be done for all the rules to be explained in the sequel. Now to see that the conclusion is valid we have first to see that $f : \alpha_2 \rightarrow \beta_2$ for given $f : \alpha_1 \rightarrow \beta_1$. For this, in turn, we have to see that $fa : \beta_2 a$ for $a : \alpha_2$ and that $fa = fb : \beta_2 a$ for any objects a and b of type α such that $a = b : \alpha$. We show only the first of these two parts, the other following in a totally analogous manner. Now if $a : \alpha_2$ then $a : \alpha_1$ by virtue of the first premiss. And, since $f : \alpha_1 \rightarrow \beta_1$, we have that $fa : \beta_1 a$. But then, by virtue of the second premiss, $fa : \beta_2 a$. Also in an analogous way one sees that $f = g : \alpha_2 \rightarrow \beta_2$ for given $f : \alpha_1 \rightarrow \beta_1$ and $g : \alpha_1 \rightarrow \beta_1$ such that $f = g : \alpha_1 \rightarrow \beta_1$. Then the rule is correct.

In a way analogous to that of the case of families of types we formulate the rules of function application, formation of functions by abstraction, the β -rule and rule of extensionality of functions.

3.2.4 Sets.

The ground types are the types of sets and of the elements of given set, as declared by the rules:

$$\frac{}{Set : type} \qquad \frac{A : Set}{A : type}$$

We have also the rule that equal sets give rise to equal types. There are no inclusions between ground types, except for the trivial ones following from the reflexivity of type inclusion.

3.2.5 Record types and families of record types.

We now finally turn to formulating the rules of record types and record objects. The first rules to be given are those of formation of (primitive) record types. These have to be introduced as types and further as record types. So the following four rules have to be understood simultaneously.

$$\frac{}{\langle \rangle : type} \qquad \frac{\rho : record\text{-}type \quad \beta : \rho \rightarrow type}{\langle \rho, L:\beta \rangle : type} \quad (L \text{ fresh in } \rho)$$

$$\frac{}{\langle \rangle : record\text{-}type} \qquad \frac{\rho : record\text{-}type \quad \beta : \rho \rightarrow type}{\langle \rho, L:\beta \rangle : record\text{-}type} \quad (L \text{ fresh in } \rho)$$

From now on we omit side conditions of rules to the effect that labels are declared at most once in record types. The justification of the rules of the second line above follows immediately from the explanation of the form of judgement that $\rho : \text{record-type}$. To justify the rules in the first line we have to explain what an object is and what identical objects are of each of the primitive record types. Let us now make some preliminary remarks that may help to understand the explanations given below. One can interpret the fields that compose a record type as constraints that the objects of the record type must satisfy. More precisely, given a record type ρ , to know $r : \rho$ requires to know that, for every label L declared in ρ , the selection $r.L$ of L out of r is defined as of a type that respects the declaration of the label.

Based on this observation, one first concludes that then the record type with no labels $\langle \rangle$ imposes no constraints on its objects, i.e. there are no conditions that have to be satisfied in order to assert $r : \langle \rangle$ for any expression r . On the other hand, to assert $r : \langle \rho, L:\beta \rangle$ requires to know first that $r : \rho$. Further, the selection $r.L$ must be defined as of appropriate type. This type depends on the values assigned in r to the labels declared in ρ . Formally, this dependence is expressed in the declaration of L by associating the latter to the family of types β over ρ . Correspondingly, the type of $r.L$ is specified as βr . Thus we arrive at the following explanations:

$r : \langle \rangle$ is vacuously satisfied.

$r_1=r_2 : \langle \rangle$ is vacuously satisfied for $r_1 : \langle \rangle$ and $r_2 : \langle \rangle$.

And, under the premisses of the second rule of record type formation:

$r : \langle \rho, L:\beta \rangle$ means that $r : \rho$ and that $r.L : \beta r$.

$r_1=r_2 : \langle \rho, L:\beta \rangle$, where $r_1 : \langle \rho, L:\beta \rangle$ and $r_2 : \langle \rho, L:\beta \rangle$, means that $r_1=r_2 : \rho$ and that $r_1.L=r_2.L : \beta r$.

Record types can also be obtained by applying families of record types. Here are the rules governing these families.

$$\frac{\sigma : \alpha \rightarrow \text{record-type} \quad a : \alpha}{\sigma a : \text{record-type}} \qquad \frac{\rho : \text{record-type} [x:\alpha]}{[x]\rho : \alpha \rightarrow \text{record-type}}$$

Finally, we can also introduce record types by explicit definition. If in an explicit definition of a type R , the definiens is a record type ρ , then we are justified in stating the axiom $R : \text{record-type}$. Also, if in the definition of a family of types F over a type α the definiens of Fx is a record type ρ depending on $x:\alpha$, we are allowed to state the axiom $F : \alpha \rightarrow \text{record-type}$. We will later refer to the *construction* of a record type, meaning the process of its generation by using the rules for forming primitive record types. The construction of a defined record type is then to be understood as the construction of its definiens. The same is the case with respect to the conditions of a field being in a record type and a label being fresh in a record type.

Identical (primitive) record types are constructed by the following rules:

$$\frac{}{\langle \rangle = \langle \rangle} \qquad \frac{\rho_1 = \rho_2 \quad \beta_1 = \beta_2 : \rho_1 \rightarrow \text{type}}{\langle \rho_1, L:\beta_1 \rangle = \langle \rho_2, L:\beta_2 \rangle}.$$

These rules serve only to express that definitional identity is preserved by substitution in record types. Recall that we have chosen a system that is weak in proving definitional identity of types. The expressiveness in typing objects is obtained by the rules of inclusion of record types. Before displaying these, it is convenient to consider the following rules:

$$(1) \frac{}{\langle \rho, L:\beta \rangle \sqsubseteq \rho}$$

$$\frac{}{\beta : \rho \rightarrow \text{type}} (L:\beta \text{ in } \rho) \qquad \frac{r : \rho}{r.L : \beta r.} (L:\beta \text{ in } \rho)$$

Only the latter two require explanation. We refer to them below as the rules of fields. They are explained similarly. The condition that $L:\beta$ is in ρ means that, at one point during the construction of ρ , another record type ρ' was enlarged with the field $L:\beta$. Then it had to be the case that $\beta : \rho' \rightarrow \text{type}$. Also, by repeated use of the rule (1) and transitivity of type inclusion, we conclude $\rho \sqsubseteq \langle \rho', L:\beta \rangle$ and, further, $\rho \sqsubseteq \rho'$. From the latter and $\beta : \rho' \rightarrow \text{type}$ we conclude $\beta : \rho \rightarrow \text{type}$ thereby justifying the first rule of fields. As to the second, its conclusion follows from $r : \langle \rho', L:\beta \rangle$ which is in turn a consequence of the premiss $r : \rho$ and $\rho \sqsubseteq \langle \rho', L:\beta \rangle$.

The second rule of fields serves as a precise direct explanation of the meaning of $r : \rho$ for record type ρ . The three rules just considered are going to be used for explaining the rules of inclusion of record types that we now formulate:

$$\frac{\rho : \text{record-type}}{\rho \sqsubseteq \langle \rangle} \qquad \frac{\rho_1 \sqsubseteq \rho_2 \quad \beta_1 \sqsubseteq \beta_2 : \rho_1 \rightarrow \text{type}}{\rho_1 \sqsubseteq \langle \rho_2, L:\beta_2 \rangle} (L:\beta_1 \text{ in } \rho_1)$$

The rules express that $\rho_1 \sqsubseteq \rho_2$ if ρ_1 contains a field for each label declared in ρ_2 and the (families of) types of the corresponding declarations are in the inclusion relation. The order of the fields within each record type is not relevant for determining whether they are in the inclusion relation. Only the second rule needs to be explained in detail. Assume then the premisses and the side condition. Notice that the condition that L is fresh in ρ_2 has been omitted. This condition is necessary to guarantee the well-formedness of $\langle \rho_2, L:\beta_2 \rangle$ and hence that of the conclusion of the rule. What has to be shown is that every object of type ρ_1 is an object of type $\langle \rho_2, L:\beta_2 \rangle$ and that equal record objects of type ρ_1 are equal objects of type $\langle \rho_2, L:\beta_2 \rangle$. We will now show the first of these, the other one requiring essentially the same reasoning. Assume then $r : \rho_1$. To know that $r : \langle \rho_2, L:\beta_2 \rangle$ is to know that $r : \rho_2$ and that $r.L : \beta_2 r$. Now, from the assumption $r : \rho_1$ and the premiss $\rho_1 \sqsubseteq \rho_2$ it follows that $r : \rho_2$. On the other

hand, using the rules of fields and the side condition that $L:\beta_1$ is in ρ_1 , we see that $\beta_1 : \rho_1 \rightarrow \text{type}$ and that $r.L : \beta_1 r$. Finally, from the latter and the premiss $\beta_1 \sqsubseteq \beta_2 : \rho_1 \rightarrow \text{type}$, we know $r.L : \beta_2 r$.

The next rule is justified in the same manner as the second rule of fields:

$$\frac{r=s : \rho}{r.L=s.L : \beta r} \quad (L:\beta \text{ in } \rho)$$

Record objects are formed as sequences of assignments of objects of appropriate types to labels. We call each of these assignments a field of the record object. Notice that there is no restriction on labels occurring more than once in record objects. This, however, is inessential in the sense that it does not provide any additional expressivity.

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

The first of these rules requires no justification. To justify the second rule, we have to define the selections from $\langle r, L = a \rangle$ of all the labels in $\langle \rho, L:\beta \rangle$. For the labels in ρ , this is done by defining $\langle r, L = a \rangle$ to be the same record object of type ρ as r , which was given. On the other hand, the selection $\langle r, L = a \rangle.L$ is defined in the obvious way, i.e. as a . Thus we arrive at the rules below. Notice that the condition that L is fresh in ρ has been omitted in the rule of extension of record objects. For the sake of clarity, we make it explicit now:

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

The second of these two definitions implies that the rightmost assignment to a label in a record object overrides the preceding ones.

Finally, 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 $r=s : \rho_1$.

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

To understand the second of these rules it might be useful to notice that the premisses that both r and s are of type $\langle \rho, L:\beta \rangle$ have been omitted.

4 Conclusions.

We have formulated and explained an extension of type theory with dependent record types and subtyping. We have shown that dependent record types constitute a general mechanism for the formalization of types of structures such as

algebraic structures or abstract data types. In particular, they make it possible to form types of tuples in such a way that any arbitrary set is allowed to be a component of tuples of the types in question while the possibility is still retained of extending the theory at any time with further set formers. Moreover, the relation of subtyping between record types allows to directly represent the informal principle that every structure of a type T is itself also of type S provided T is defined as an enrichment of S with additional structure.

Type checking in the extended theory is still decidable and has been implemented on machine. In addition to the formalization presented in this paper, the implementation has been used to verify an abstract version of sorting by insertion in (Tasistro 1997). In this latter work, dependent record types are used to express specifications of abstract data types.

The theory here developed is a direct successor of the calculus of substitutions for type theory (Martin-Löf 1992; Tasistro 1997) in the sense that record types can be seen as type constructions corresponding to contexts of variables—record objects becoming then the counterpart to substitutions.

Several theories of records have been developed in the context of systems without dependent types, mainly with the motivation of providing foundations for concepts that appear in object oriented programming. Then, for instance, there is by now a standard way of encoding objects in the sense of object oriented programming as recursively defined records. The general motivation mentioned departs from ours, which, as far as the theory of programming is concerned, is limited to that of providing basic means that allow the use of dependent types for expressing specifications of abstract data types and modules in a general way. The problem of formulating a type system for object oriented programming raises a number of questions that are simply not relevant for our purposes. As to dependent record types, they have been implemented in *PVS* (Owre *et al.* 1993), which is a theorem proving system based on classical higher order logic. The subtyping that record types induce is, however, not a part of this implementation.

In the original type theory, it is possible to encode each particular instance of inclusion between types α and β by using a *coercion* function that injects the objects of type α into the type β . In (Barthe 1996; Bailey 1996; Saïbi 1997) different mechanisms are developed that allow to declare coercions between types or classes of types, which can later be left implicit in expressions. This provides in principle a limited form of the subtyping mechanism that we have formulated here, since only those inclusions hold that can be derived from the explicitly declared coercions. However, the use of coercions allows for further convenient ways of expression, which are not captured by our mechanism of subtyping. One case is the possibility of writing fa when f is not itself a function but from which a function can be obtained in a standard manner. For instance, f could be any structure morphism. Moreover, coercions allow in principle to represent proper

inclusions between sets, which has not been considered in this paper. To extract useful conclusions from a comparison between the two approaches requires to put them to work in extensive case studies. This has not yet been done in the case of the theory that we have here formulated.

Bibliography

- Bailey, A. (1996). *LEGO with implicit coercions*. Available at <ftp.cs.man.ac.uk/pub/baileya/Coercions>.
- Barthe, G. (1996). *Implicit coercions in type systems*. In *Selected Papers from the International Workshop TYPES '95, Torino, Italy, June 1995*. LNCS 1158.
- Betarte, G. (1997). *A type checking algorithm for an extension of Martin-Löf's type theory with record types and subtyping*. Draft paper.
- Dahl, O. and Nygaard K. (1966). *Simula, an Algol-based simulation language*. *Comm. ACM* 9, 671-678.
- Martin-Löf, P. (1984). *Intuitionistic Type Theory*. Bibliopolis, Napoli.
- Martin-Löf, P. (1987). *Philosophical Implications of Type Theory*. Lectures given at the Facoltà de Lettere e Filosofia, Università degli Studi di Firenze, Florence. Privately circulated notes.
- Martin-Löf, P. (1992). *Substitution calculus*. Talks given in Göteborg.
- Nordström, B., Petersson, K. and Smith, J. (1990). *Programming in Martin-Löf's type theory. An introduction*. Oxford Science Publications.
- Owre, S., Shankar, N. and Rushby, J. M. (1993). *User guide for the PVS specification and verification system (Beta release)*. Comp. Sc. Laboratory, SRI International.
- Saïbi, A. (1997). *Typing algorithm in type theory with inheritance*. 24th. Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages.
- Tasistro, A. (1997). *Substitution, record types and subtyping in type theory, with applications to the theory of programming*. PhD thesis, Dpt. of Computing Science, Chalmers University of Technology and University of Gothenburg.

5 Appendix.

We here present code as accepted by a type checker for type theory extended with record types and subtyping that has been implemented.

A script for the type checker looks very much like one for a functional programming language, say ML. The type checker reads *declarations* of various forms. These are *type* declarations (command `type`), *type family* declarations (command `typef`), *value* declarations (command `val`) and *function* declarations (command `fun`).

The notation $[x, \dots, z]e$ stands for the abstraction of the variables x, \dots, z in the expression e . The notation `let {...} e` reads with the usual meaning of a let expression.

We force distinction of labels from constants or variables by writing labels of a record type, when used in subsequent fields of the same record type, preceded by a dot.

5.1 Definition of Group

```

type BinRel = <S:Set,R:(x:.S,y:.S)Set>;

type Setoid = <BinRel,
  ref:(x:.S).R x x,
  symm : (x:.S,y:.S,p:.R x y).R y x,
  trans : (x:.S,y:.S,z:.S,p:.R x y,q:.R y z).R x z>;

type EqToEq = (Sd:Setoid,x:Sd.S,y:Sd.S,z:Sd.S,w:Sd.S,
  p:Sd.R x y, q:Sd.R x z, r:Sd.R y w) Sd.R z w;

val eqtoeq =
  [Sd,x,y,z,w,p,q,r]Sd.trans z y w (Sd.trans z x y (Sd.symm x z q) p) r
    : EqToEq;

type Groupoid = <Setoid,
  op: (x:.S,y:.S).S,
  opcong : (x:.S,y:.S,z:.S,w:.S,p:.R x y,q:.R z w)
    .R (.op x z) (.op y w)>;

type Semigroup = <Groupoid,
  assoc:(x:.S,y:.S,z:.S)
    .R (.op (.op x y) z) (.op x (.op y z))>;

fun symmAssoc(Sg:Semigroup) =
  [x,y,z]Sg.symm (Sg.op (Sg.op x y) z) (Sg.op x (Sg.op y z))
    (Sg.assoc x y z)
  : (x:Sg.S,y:Sg.S,z:Sg.S)
    Sg.R (Sg.op x (Sg.op y z)) (Sg.op (Sg.op x y) z) ;

type Monoid = <Semigroup,
  e:.S,
  id:(x:.S).R (.op x (.e)) x>;

type Group = <Monoid,
  inv:(x:.S).S,
  invcong:(x:.S,y:.S,p:.R x y) .R (.inv x) (.inv y),
  invprop:(x:.S).R (.op x (.inv x)) (.e)>;

```

5.2 Proof of the lemma and unicity of the right identity

The statement of lemma used in the proof of unicity of the right identity of groups and the corresponding proof are defined as follows:

```

type LemmUniq = (G:Group,x:G.S,p:G.R (G.op x x) x) G.R x (G.e);

val lemmuniq =
  [G,x,p]
  let {val lemma = G.trans (G.op x (G.e))
      (G.op (G.op x x) (G.inv x))
      (G.op x (G.inv x))
      (G.trans (G.op x (G.e))
        (G.op x (G.op x (G.inv x)))
        (G.op (G.op x x) (G.inv x))
        (G.opcong x x (G.e) (G.op x (G.inv x))
          (G.ref x)
          (G.symm (G.op x (G.inv x))
            (G.e)
            (G.invprop x))))
      (symmAssoc G x x (G.inv x))
      (G.opcong (G.op x x) x (G.inv x) (G.inv x)
        p
        (G.ref (G.inv x)))
      : G.R (G.op x (G.e)) (G.op x (G.inv x))
  }
  eqtoeq G
  (G.op x (G.e))
  (G.op x (G.inv x))
  x
  (G.e)
  lemma
  (G.id x)
  (G.invprop x)
  : LemmUniq;

```

We use `let` expressions for local lemmas. In this case, the local lemma `lemma` represents the proof that $x \circ e = x \circ x^{-1}$, as is revealed by its type. It is in this proof that the main function's argument `p`, which represents the assumption that $x \circ x = x$, is used. Notice the application in the main function of `eqtoeq` to `G`, which is correct due to the subtyping rules.

Finally, as to the proof of the unicity of the right identity of a group, we get a representation that is a straightforward encoding of the informal argument:

```

fun uniqunit(G:Group) = [e1,id2] lemmuniq G e1 (id2 e1)
  : (e1:G.S, id2:(x:G.S) G.R (G.op x e1) x)
    G.R e1 (G.e);

```