Dependent Record Types and Formal Abstract Reasoning: *Theory and practice*

Gustavo Betarte

Department of Computing Science
Chalmers University of Technology and
University of Göteborg

S-412 96 Göteborg, Sweden.

A Dissertation for the Ph.D. Degree in Computing Science
at the University of Göteborg.

Department of Computing Science
S-412 96 Göteborg, Sweden.

ISBN 91-7197-601-9
Göteborg, 1998.

Abstract

This work contains investigations on the formal correctness and use of an extension of Martin-Löf's type theory with dependent record types and subtyping. We put forward the adequacy of dependent record types as a natural type theoretic setting for expressing the notion of abstract data type — in particular we explore the formal representation of systems of algebras — and for the modular development of proofs. By virtue of the mechanism of subtyping available, in addition, it is possible to reutilize code that has been developed for a certain system when reasoning about any other system that conforms to an specialization, or extension, of it. We study the problem of the mechanical verification of the forms of judgement of the extended theory and the outcome is a proof checker that provides assistance in the use of the language of the calculus for the formal development of proofs. The algorithm of type checking, which constitutes the logical heart of the implemented system, is developed for a particular reformulation of the extension. This latter calculus, which is also presented and analysed in this work, incorporates the notion of parameter to stand for that of a free variable of a certain type. We present some experiments on the formalization of algebraic constructions that have been carried out using the proof checker

Acknowledgements

I am deeply indebted to Björn von Sydow, my supervisor. He has always been there for me, enlightening, supportive and patient. I have a profound respect for him and I admire his capacity as a computing scientist. The conception, elaboration and final writing of this thesis immensely benefited from Björn. I have been very fortunate to have him as my advisor, he has also been a caring friend and a good comrade.

I am most grateful to Bengt Nordström, who opened the doors of the Programming Logic group to me. It has been a privilege to partake in the research environment he has greatly contributed to create and develop.

The influence of Alvaro Tasistro, Tato, in my research education has been permanent. He motivated my interest in type theory and taught me to understand it, he also created the subject on which this thesis concentrates. To work with him has been an extremely enriching experience. Besides, Tato together with Ana Bove, Daniel Fridlender, Verónica Gaspes and Nora Szasz are dear friends with whom I shared many enjoyable moments during all these years we have been living in Göteborg.

I want to thank the members of my supervision committee, Thierry Coquand and Jan Smith, for their comments and advice regarding this work.

Many people have contributed to this thesis through fruitful discussions or by reading and commenting previous versions of it. Special thanks to Peter Dybjer, Daniel Fridlender, Verónica Gaspes, Christine Paulin-Mohring, Henrik Persson and Alvaro Tasistro.

To work at this Department has definitely been a very challenging and pleasant experience. The academic environment is thought-provoking, based on solid scientific criteria and friendly. This is complemented with the work of many people who take care of us with great efficiency and a nice disposition. I specially want to thank Marie Larsson, who is also a very good friend, Christer Carlsson and Hans "Hasse" Hellström, for all their help. Among the people that work at the Department I have found very nice persons that also become good friends. Thanks to all of them for making my stay here even more comfortable.

During all these years in Göteborg I have never stopped feeling a member of InCo, the Department of Computing Science at Universidad de la República in Montevideo. I am certainly indebted to Juan José Cabezas, his academic vision sparked and stimulated the attitude that eventually led many of us to come to different places to pursue our research education. I want to express my gratitude to all my friends from InCo, who have helped me in so many occasions. Special thanks to Cristina Cornes and Juan José Prada, for being so cheerful and supportive. I had the opportunity of visiting InCo many times in these last years, and some of these visits were financially supported by the Uruguayan councils of scientific research, PEDECIBA and CONICYT, to which I am grateful.

Finally, I want to acknowledge the constant support and loving care that I have received from my family and friends in Uruguay.

Contents

CHAPTER 1

Introduction

The work we present in this monograph was originally motivated by a development of the formal representation of the arithmetic of integers using the proof-assistant ALF [**Mag95**]. There we study, in the first place, an inductive formulation of the mentioned set, which we denote by Z, and provide the formal proofs that it constitutes an integral domain. A natural next step was to investigate the possibility of giving a formal account inside type theory of the algebraic theory of integral domains, write down proofs of properties that can be derived from its postulates and transfer those results to our implementation of the concrete integral domain Z. This, in turn, was accomplished by making use of the notion of *context* and *substitution* as implemented in the mentioned system:
Let *ID* be a context where assumptions have been introduced to the effect that a certain set and binary operations on that set form an integral domain. To formally reflect that a property, which is formally expressed by the type Φ, is valid for all integral domains, we then construct a proof object ϕ of type Φ under the context *ID*. On the other hand, once a system of algebras has been given a representation in terms of a particular context Γ, for stating that a particular construction conforms a concrete instance of that system we introduce a substitution, also as implemented in ALF, for Γ. Thus, if we have constructed a proof like the one described above, formally represented by the judgement $\phi : \Phi[ID]$, and γ_Z is a substitution for the context *ID*, we can obtain that $\phi\gamma_Z : \Phi\gamma_Z$. In words, if we have a proof that Φ is a property valid for all integral domains, and we know one such structure, then we also have a proof of the property for the latter, namely, it is the object $\phi\gamma_Z$.

This work has been reported in [**Bet93**] and is considered to be a complementary part of the one we shall introduce here.

We, however, found the use of contexts for the formal representation of systems of algebras a quite limited practice. When we started developing a little more involved algebraic constructions than the ones needed for the work described above, like defining the notion of an isomorphism between groups for instance, some disadvantages of this approach emerged rendering the formalization task, and the results, quite unsatisfactory. This led our investigation to considering alternative type theoretic mechanisms better suited for the representation of abstract theories and modular development of proofs. We first investigated an extension of Martin-Löf's logical framework [**Mar87, NPS89**] with dependent pairs (also called Σ types in the literature). A type checking algorithm was implemented for this extension and some case studies were developed using it. Most of the difficulties present in the "context approach" were overcome by using pairs. There remained, however, some drawbacks concerning, in particular, the possibility of making incremental definition of theories and the reutilization of proofs. We then turned our attention to the study of an extension of Martin-Löf's logical framework with dependent record types and subtyping. This extension has been proposed by A. Tasistro and is described in [**Tas97, BT97**]. Hereafter, we shall refer to it as *the extended theory* or sometimes plainly as *the extension.*

We now proceed to describe how records and subtyping can act as the formal counterpart to algebraic constructions consistently used in the informal practice.

Let us consider the problem of formalizing in type theory the notion of a set and an equivalence relation on it. The name *Setoid* has elsewhere been used for this notion. 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. So we start by introducing just binary relations on sets and will obtain the formal definition of setoid by enriching the previous notion with further structure, namely, the components that establish that the relation is reflexive, symmetric and transitive.

The system of types of the original formulation of the theory is constituted, in the first place, by the type *Set*, the type of *inductively* defined sets. Then, any individual set A, gives rise to the type of its elements. Type families are expressions of the language that when applied to individuals of the appropriate type give rise to a type. Moreover, it is possible to introduce arbitrary families of types in the formal language. One such family can be constructed by an operation of abstraction, denoted as $[x]\alpha$, which binds the occurrences of the variable x in the type α. 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 $\alpha{\to}\beta$ is also a type. The application of an object f of this latter type yields an object $f\,a$ 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 transfered to propositions about generic individuals. For instance, given a set A, $A{\to}[x](A{\to}[y]Set)$ 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 $A{\to}[x]Rxx$ 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.

So now let us turn our attention to sets with binary relations on them. We shall call this notion just *binary relation* and define it to be a pair $(S, \approx)$ where S is a set and $\approx$ a binary relation on S. Thus, were we to define the type of binary relations, it should be introduced as a type of tuples.

Now, regarding the mechanisms of type formation we have described above the only way to get tuple types in type theory is to introduce *sets* of tuples. But consider now binary relations as defined above. If S can be any set, then the type of binary relations cannot itself be a set or it would be allowed to form a part of some of its own elements. Circular constructions of the latter kind are not allowed in the *predicative* language of type theory.

Another possibility would be to restrict S in the pair $(S, \approx)$ to be an element of a previously constructed set of sets that we call a *universe*. Then the type of setoids could be introduced as a set, obviously then not belonging to the universe. But now the universe encloses a fixed number of set constructors. And we still want to be able to introduce new set constructors, i.e. new sets that could be carriers of setoids. For this, what we need is types of structures of which *arbitrary* sets may be specified as components.

Dependent record types are just sequences of *fields* in which *labels* are declared as of certain types: $\langle \mathsf{L}_1 : \alpha_1, ..., \mathsf{L}_n : \alpha_n \rangle$. The type α_{i+1} may depend on the preceding labels L_1,...,L_i. We could then introduce the type of binary relations on a set as:

$$\begin{aligned} &\mathit{BinRel} : \mathit{type} \\ &\mathit{BinRel} = \langle \mathsf{S} : \mathit{Set}, \approx : \mathsf{S} \to \mathsf{S} \to \mathit{Set} \rangle \end{aligned}$$

Labels may participate in the formation of types in the same way as ordinary variables or constants do. In order to avoid ambiguities they are syntactically distinguished from the latter. Here we use the font label. A type declaration like the one just introduced is nothing but the explicit definition of a type.

Now we turn to consider the definition of setoid. In the extended theory the possibility of incremental definition is given directly by the rules of formation of record types. This is formally stipulated as the iteration of the operation of extending a record type with one more field, starting from the record type with no fields. Thus:

$$\begin{aligned} &\mathit{Setoid} : \mathit{type} \\ &\mathit{Setoid} = \ \langle \mathit{BinRel}, \\ &\qquad \mathsf{ref} : (x : \mathsf{S}) \approx x\ x, \\ &\qquad \mathsf{symm} : (x, y : \mathsf{S}) \approx x\ y \to \approx y\ x, \\ &\qquad \mathsf{trans} : (x, y, z : \mathsf{S}) \approx x\ y \to \approx y\ z \to \approx x\ z\ \rangle \end{aligned}$$

This example shows how systems of algebras can be represented as record types: in informal language algebraic structures are defined as tuples of elements satisfying certain properties. As was already illustrated, in type theory these properties become in general function types. Therefore, to each property required by the definition of a system of algebras there corresponds a field in the record type that represents the system. Since proofs are objects, we can express this requirement by making proofs actual components of the structures being defined. Formally, then, the distinction between elements of a structure and proofs of the demanded properties disappears.

We have now seen one example of incremental definition of systems. Informally, it would be stated thus: “A setoid is a set with a binary relation in which the latter is an equivalence relation”. Still informally it is then natural to use directly that every setoid is a set with a binary relation. In the formal language, this means that an object of type *Setoid* has also the type *BinRel*, i.e. a form of polymorphism. Now, both *Setoid* and *BinRel* are record types and it is naturally given in the definition of record types that this form of polymorphism should be allowed.

As systems of algebras are represented by record types, the representation of a concrete algebra corresponds to a record object of the type representing the system. Record objects, as usual, are constructed as sequences of fields that are bindings of objects of appropriate types to labels.

An interesting point is that once a derived property has been proved for a system, any concrete algebra for that system should also have this property. In our case, proofs are represented as (functional) objects. Thus, a natural way of obtaining instantiation of properties is by application of the proof object to the representation of the concrete algebra. Let us illustrate this with a simple example.

Suppose that we have introduced the type *Group* as a record type that represents this system of algebras and in addition we have available an object *cancelL* of type $(G : \mathit{Group})\ (x, y, z : G.\mathsf{S})\ G.\approx\ (G.\circ\ z\ x)\ (G.\circ\ z\ y) \to G.\approx\ x\ y$. Thus, in words, *cancelL* proves that the operation of the group G is left cancellative.

Suppose, now, that we have defined the set Z of integers and the propositional equality $=_Z$ on it, as usually done in type theory. Furthermore, we also introduced the binary operation $+_Z$ the unary operation $\sim_Z$ and the distinguished element 0_Z. Suppose then that we have proved all the properties characterising $(Z, =_Z, +_Z, \sim_Z, 0_Z)$ as a group (a formalization of these proofs in type theory is presented in [**Bet93**]). Thus, we could define the algebra Group_Z to be:

$$\begin{aligned}&\mathit{Group}_Z : \mathit{Group}\\ &\mathit{Group}_Z\ = \langle \mathsf{S} = Z, \approx\ = =_Z, \cdots, \circ = +_Z, \cdots, \boldsymbol{e_1} = 0_Z, \cdots, - = \sim_Z \rangle\end{aligned}$$

The labels $\circ$, $\boldsymbol{e_1}$ and $-$ correspond to the operation, the unit and the inverse function of the group, respectively. So, now we could apply the function *cancelL* to the record object Group_Z to obtain the proof that $+_Z$ is cancellative as follows

$$\begin{aligned}&\mathit{cancelL}{+_Z} : (x, y, z : Z) =_Z (+_Z\ z\ x)\ (+_Z\ z\ y) \to =_Z\ x\ y\\ &\mathit{cancelL}{+_Z}\ = \mathit{cancelL}\ \mathit{Group}_Z\end{aligned}$$

We can discern in what we have described different facets of the research connected to type theory. Firstly, its use, as a formal language, to carry out constructive mathematics. There is also the activity of understanding the theory itself, and moreover, its possible extensions. There exists, in addition, one more aspect to be recognized. Type theory is a formal logic, therefore the assertions about mathematical objects that can be expressed in terms of the forms of judgement of the theory can mechanically be verified to hold. This has given rise to a whole area of research concerned with the study, design and implementation of systems that provide assistance in the use of the language. In this direction, we have implemented a proof checker that verifies the formal correctness of the judgements of the extended theory. The outcomes in connection with this latter subject form what we consider the main contribution of this work.

The structure of this monograph

The next chapter summarises to a great extent the mainstream of the investigations that constitute this monograph. We start by giving a concise description of type theory and its use for carrying out constructive mathematics. In particular we focus on the formalization of abstract algebra. We concentrate on a simple case-study, namely, a little portion of the theory of Boolean algebras. Then, we give a succinct description of the proof checker that has been implemented and also of the form of expressions and declarations that it reads. Thereafter, we present (portions of) the formal code, which was checked using the system, representing the algebraic constructions in question. The intension is there to illustrate the features we consider relevant in connection with records and subtyping. This first part of the chapter has almost literally been taken from [**Bet97**]. Finally, we give a detailed account of the extended theory, as originally presented in [**Tas97, BT97**].

In chapter 3 we give an informal discussion concerning the design of the type checking algorithm for the extended theory. As it can be regarded as a quite direct extension of the one for the original theory, we then start by describing this latter algorithm. In addition, this will bring into attention the problems posed by the checking of unlabeled abstractions. This is important because those problems are carried over to the procedures for checking the typing judgements of the extended theory. In another direction, even though still in connection with the checking of abstraction operators, we then confine attention to the treatment of free names. We then motivate the use of parameters, in the sense of [**Coq91, Pol94a**], to stand for the generic values (or free variables) of the various types. As a consequence of this choice, thus, as we are interested in obtaining a final formulation of the type checking algorithm such that it can be easily proven to be correct, we set ourselves to give a formulation of the extended calculus that incorporates the notion of parameters.

The resulting calculus is then presented in chapter 4. The complete formalization of the proof rules that it embodies is presented following the syntactico-semantical method used by P. Martin-Löf in [**Mar84**], and thenceforth consistently exploited in [**Mar87, Mar92, Tas97**]. In principle, there is no need for introducing a notion of reduction for understanding the computational meaning of the calculus, it naturally emerges from the use of definitional equality, which finds its formal counterpart in judgemental equality. Nevertheless, regarding implementation issues, it is convenient to make explicit a procedure that performs the computation of an expression to some normal form. This latter, in turn, can be grasped to be the value of the expression. Thus, we introduce the concept of weak head normal form and define a weak head reduction relation over the expressions of the calculus. Then we prove some metatheoretic results concerning the interplaying of this relation and the judgements of the calculus. Particularly relevant concerning the correctness of the type checking algorithm is the result establishing a sort of subject reduction property.

We then in chapter 5 concentrate in the design, final specification, implementation and correctness of a proof checker, whose logical heart is a type checking algorithm for the forms of judgement of the calculus presented in the previous chapter. In doing this, we maintain the spirit of the informal presentation given in chapter 3. After explaining the procedures that verify the correctness of the various forms of declaration, the input to the system, we then first present the algorithms for the judgements of the original theory and then we show how they are modified to cope with the judgements of the calculus extended with record types and subtyping. We then give an informal proof of the soundness of the algorithm with respect to the calculus in question. In order for this chapter to be an all-embracing presentation of the system we end it up giving a flavour of its implementation, which was developed using the language Haskell [**Pet96**].

In chapter 6 we present some of the experiments we have done concerning the formalization of abstract algebra using the proof checker described in the previous chapter. We show first (parts of) the reformulation of the results on integral domains we presented in [**Bet93**]. The representation of the system integral domain is now given in terms of record types. The incremental definitions of systems of algebras are directly accomplished by using record extension. We also illustrate a simple application of subtyping, namely, the reutilization of proofs developed for groups and commutative rings when reasoning about integral domains. Then we highlight the constructions we needed to develop for the formal representation of Cayley's theorem for group theory, that is to say, that any abstract group is isomorphic to a group of permutations. The formal proof of this theorem per se is not a significant contribution. Nevertheless, it allows to illustrate the adequacy of the extended theory for building up a little more involved algebraic constructions, like isomorphisms between groups, the construction of groups of transformation and permutations over a given space, and morphisms between (these) groups. The corresponding representation of most of these notions using contexts was either inadequate or, in some cases, impossible to achieve.

Finally, in chapter 7 we comment on the connections with related works, give some final conclusions and consider possible further work.

CHAPTER 2

Doing abstract algebra in type theory extended with dependent record types and subtyping

1. Introduction

We shall use an extension of Martin-Löf's theory of logical types [**Mar87**] with dependent record types and subtyping as the formal language in which constructions concerning systems of algebras are going to be represented.

The original formulation of Martin-Löf's theory of types, from now on referred to as the logical framework, has been presented in [**NPS89, Tas93b, 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 [**BT97, 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 2.3. Record objects, as in programming languages, are sequences of assignments of objects of appropriate types to labels. Each of these objects can be accessed by selecting the corresponding label of the record object. The mechanism of subtyping or type inclusion introduced is, in the first place, the one naturally induced by record types. However, once record inclusion is formally stipulated it is also required that rules of subtyping have to be given for the rest of the type formers.

In [**BT97**] is illustrated the use of record types and subtyping for the formalization of systems of algebras by developing a formal definition of group as a record type. A very simple application of subtyping is there provided as well. In this chapter, we also 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 rest of the chapter proceeds as follows: in the next section we give a brief review of type theory and how it can be used to formally represent mathematical constructions. Then we discuss the different alternatives that emerged when trying to represent algebraic constructions in type theory. In particular, we focus on the notions of contexts, Σ-sets and dependent pairs, intending at the same time, to motivate the use of dependent record types as an appropriate mechanism for the formulation of types of tuples. In section 2.3 we summarize the features introduced by the extension of the framework with record types and subtyping. This will allow for better understanding of the formalization of the case study we present.

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. In section 3.2 we go over the syntax of the input expressions and forms of declaration that the implemented proof checker reads. We proceed then, in section 3.3, presenting the formal constructions developed in order to formalize the case study at hand. We do not provide the whole code involved in the formalization but rather concentrate on the fragments that we consider 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 section 4 we present a detailed account of the extension of Martin-Löf's logical framework with dependent record types and subtyping. This extension was first presented by A. Tasistro in the TYPES workshop held at Nijmegen in 1993 and also reported in a draft paper [**Tas93a**]. This formulation of the calculus was subjected to some modifications and its final version is included in Tasistro's thesis [**Tas97**] and in the reference [**BT97**]. The content of section 4 is almost literally taken from the latter, except for some remarks we have introduced in order to help the understanding of the work that follows this chapter.

2. Representation of systems of algebras in type theory

We start this section by giving a brief description of type theory as formulated using the theory of types as logical framework. For a more comprehensive presentation we refer to [**NPS89, CNSvS94, Tas93b**].

2.1. The logical framework. The system of types of the original formulation of the theory is constituted, in the first place, by the type *Set*, the type of *inductively* defined sets. Then, any individual set A, gives rise to the type of its elements. This latter type is denoted in [**NPS89**] as $El(A)$, where El is a (primitive) family of types over the type Set. Type families are expressions of the language that when applied to individuals of the appropriate type give rise to a type. Moreover, it is possible to introduce arbitrary families of types in the formal language. One such family can be constructed by an operation of abstraction, denoted as $[x]\alpha$, which binds the occurrences of the variable x in the type α. In what follows, we shall omit the family El in the notation and write just A for both the object of type set and the type that it determines. Finally, there exists a mechanism for the formation of (dependent) function types: if α is a type, and β is a family of types over the 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, explained and justified in [**Mar87**], allows to grasp propositions as sets, and thereby, their proofs as elements of those sets. There is no formal distinction in the language of the theory between the type of sets and the type of propositions.

Further, in the presence of families of types, this interpretation of propositions can be transfered to propositions about generic individuals. For instance, given a set A, $A \to [x](A \to [y]Set)$ 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 $A \to [x]Rxx$ is the type of proofs that R is reflexive. This function type is usually written as $(x : A)Rxx$, that can be read: "for all 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.

2.2. Formal abstract algebra. In [**Bet93**] is presented a formalization of the arithmetic of integer in Martin-Löf's type theory. The result of the whole work, which was carried out using the proof assistant ALF [**Mag95**], amounts to the formalization of (an inductive definition of) Z, the set of integers, the arithmetical operations $+$ and $*$ and the proofs of the properties establishing that the algebra formed by that particular representation of Z, $+$ and $*$ is an integral domain.

In addition, some of those proofs were also developed for a formalization of Z as a quotient set. As expected, due to the different nature of the respective representations of the set and therefore the corresponding formulation of the mentioned operators, proofs of properties like associativity of the operation $+$ followed quite a different pattern of reasoning in each case. This provided us with interesting insights about the task of formalizing mathematics in type theory. Furthermore, having in mind the properties that can be derived from the postulates of an integral domain, it also motivated the formulation of an abstract notion of algebraic system which could be used to reason about the properties satisfied by the algebra of integers independently of the chosen representation.

To formalize the notion of what an algebraic structure is, whose components are sets and n-ary operations on those sets which satisfy specified axioms, we chose the notion of context as implemented in ALF: let ID be a context where assumptions have been introduced to the effect that a certain set and binary operations on that set form an integral domain. To formally reflect that a property, which is formally expressed by the type Φ, is valid for all integral domains, we then construct a proof object ϕ of type Φ under the context ID.

On the other hand, once a system of algebras has been given a representation in terms of a particular context Γ, for stating that a particular construction conforms a concrete instance of that system we introduce a substitution, also as implemented in ALF, for Γ.

Thus, if we have constructed a proof like the one described above, formally represented by the judgement $\phi : \Phi[ID]$, and γ_Z is a substitution for the context ID, we can obtain that $\phi\gamma_Z : \Phi\gamma_Z$. In words, if we have a proof that Φ is a property valid for all integral domains, and we know one such structure, then we also have a proof of the property for the latter, namely, it is the object $\phi\gamma_Z$.

Using these mechanisms we managed to reflect some of the abstract reasoning usually carried out when doing algebra, and moreover, to transfer those results to the (representation of) concrete algebras.

However, this use of contexts and substitutions as the support for developing formal algebra, could not be further exploited. As soon as one wants to reason about constructions like morphisms between algebraic structures, for instance, the limitations imposed by the very nature of contexts render the process of formalization quite a cumbersome task. Furthermore, notions like that of a functor that when applied to a group returns its underlying monoid, when both systems are represented as contexts, cannot be expressed as an object in the formal language of ALF. This is in accordance, however, with the fact that contexts are not types. We refer to [**Bet93**] for a more detailed discussion on this.

We turned then to investigate alternatives which, most importantly, would allow us to express systems of algebras as types, and therefore, concrete algebras as objects of a certain type. In particular, thus, the definition of a functor as the one mentioned above would conform as to the one of a function between the corresponding types.

We found an adequate starting point in the pioneering work by MacQueen [**Mac86**] on the explanation of the notion of *module* in terms of a ramified system of dependent types with Σ-types. These types, in turn, are there understood as presented by Martin-Löf in, for instance, [**Mar84**]. One such type, usually written down as $\Sigma x \in A.B(x)$, corresponds to the disjoint union of the family of types $B(x)$ with x ranging over the type A. The elements of this type are pairs of the form (a, b) such that a is an object of type A and b has type $B(a)$. Thus, the type of the second component may depend on the first component of the pair. This same understanding of modules, or more precisely of abstract data type in this case, as formally represented by Σ-types is also proposed in [**Luo88**] where an extension of the Calculus of Constructions [**CH88**] with Σ-types (ΣCC) is presented. As a motivating example Luo illustrates the adequacy of this latter calculus to express algebraic constructions.

In the context of Martin-Löf's set theory, a particular methodology for the representation of the above understanding of the notion of module or abstract data types is proposed in [**NPS89**].

A module is in that work grasped as a tuple $\langle A_1, A_2, \ldots, A_n \rangle$, where some A_i are sets and some are elements and functions defined on these sets. An example of the application of these notions to formalize algebra could be the definition of group as the tuple:

$$\langle G, *, u, inv, P_{ass}, P_{unit}, P_{inv} \rangle$$

where G is a set, $* \in G \times G \to G$, $u \in G$, $inv \in G \to G$ and P_{ass}, P_{unit} and P_{inv} express the postulates of groups. To be a group can then be understood as to be an element of the following set:

$$\begin{aligned}
&(\Sigma\ G \in U)\\
&\qquad(\Sigma\ * \in G \times G \to G)\\
&\qquad(\Sigma\ u \in G)\\
&\qquad(\Sigma\ inv \in G \to G)
\end{aligned}$$

$$\begin{array}{l}(\Pi x, y, z \in G)\\ \quad [*(x, *(y, z)) =_G *(*(x, y), z)] \times\\ \quad [*(x, u) =_G x] \times\\ \quad [*(x, inv(x)) =_G u]\end{array}$$

where U is the name for the set of the small sets, as defined in [**NPS89**]. Notice that it is assumed that the set G is equipped with a (propositional) equality.

The need for introducing the set U corresponds, in the first place, to the fact that type theory is a predicative theory, no quantification is allowed over a collection of elements (in this case *Set*) when defining a particular element of that collection. However, we have to formally express the intention that the first component of (the tuple that represents) a concrete group is indeed a set.

This way of formalizing algebraic structures, however, has in our opinion some drawbacks. In the first place, once a set S is defined, in order to be able to express that this set is the carrier of a particular group it has in addition to be an element of the set U. In other words, we must provide the code, and define the corresponding decodification, that allow us to grasp S both as a set and as an element of the set U. Now, sets in type theory have to be inductively defined, so the previous procedure amounts to say that when it comes to define the notion of group as is done above the set of carriers of groups is already closed. This is clearly a more restricted understanding than the one that asks for carriers of groups just to be sets, where this remains an open notion.

In order to achieve a formal representation of systems of algebras as (tuple) types, and therefore classifying a collection of objects that is in principle open, we considered an extension of Martin-Löf's logical framework with a mechanism for forming types of *dependent pairs*.

One possible way of accomplishing this could be extending the calculus with the following rules:

$$\frac{\alpha\ type \quad \beta\ :\ \alpha \rightarrow type}{(\alpha;\beta)\ type} \qquad\qquad \frac{\alpha\ type \quad \beta\ :\ \alpha \rightarrow type \quad a : \alpha \quad b : \beta a}{(a, b) : (\alpha;\beta)}$$

$$\frac{p : (\alpha;\beta)}{\pi_1 p : \alpha} \qquad\qquad \frac{p : (\alpha;\beta)}{\pi_2 p : \beta(\pi_1 p)}$$

The first two rules say what has to be known in order to introduce a type of dependent pairs and how objects of one such type are constructed, respectively. The last two rules, usually called of projection, express (part of) the meaning of being an object p of type $(\alpha;\beta)$; that its first projection is an object of type α and the second one an object of the type that results from applying β to the first projection of p. There must also be rules of equality that conform to the justification of the rule of object construction above.

Observe that now the type of groups defined above can be reformulated to require from a carrier of a group only to be an object of type *Set*, which as already mentioned, is the (open) type of inductively defined sets. And the formulation of the notion of group is still a predicative one.

A type checking algorithm for the logical framework extended with dependent pairs was implemented and some small-size case studies were developed using it. In particular, we focused on transferring the results on integral domains obtained using contexts into a formulation using dependent pairs. Now, in some of the proofs developed using the context approach, we could formally express the following reasoning: once we succeed in proving a property of *monoids* we can directly use that proof as one of *groups*. This was naturally reflected in the formal language by the simple reason that the context *Group* was defined as to be an extension of the context *Monoid*. Thus, "by thinning", any proof developed under the latter context is also valid under the first one. However, this is not the case once the systems of algebras are represented as dependent pairs. In order to be able to reuse the proof obtained for monoids, that now are objects, one must first apply a function that given an object of type *Group* yields the object that forms the underlying monoid. Actually, this function can be grasped as a *coercion* from groups to monoids. There is, in principle, no possible way to obtain in a direct manner, in the formal language , that any object of type *Group* can also be considered as an object of type *Monoid*.

The combination of Σ-types and mechanisms for introducing coercions between types, which once they are declared can be left implicit, has received an increasing amount of attention in recent years. From an original proposal put forward by Aczel [**Acz94**], where a notion of class and method for predicative type theories is proposed, type theoretical explanations and formulations of the notion of coercion [**Bar95, Luo96**] have been laid down. The implementation of coercion mechanisms and their use for the formalization of algebraic constructions has been reported in [**Bai97**].

Close enough to this approach is the work presented in [**Saï97**], which is also an adaptation of the ideas proposed by Aczel, where algebraic systems are represented in terms of class constructors. We shall more extensively comment on this in the chapter on related work.

What we in the following sections intend to do, instead, is to motivate the use of dependent record types as the formal counterpart to the notion of a system of algebras, and more in general, Martin-Löf's logical framework extended with dependent record types and subtyping as the formal language to carry out algebraic constructions.

2.3. Record types. In order to achieve a formal definition of *Boolean algebra* we will start by introducing the notion of a set with an equivalence relation on it. The reason to have this as the most basic kind of structure is that in formalizing systems of algebras it appears natural to require the relation informally denoted by the equality symbol $\approx$ to be given explicitly as a component of the system being defined. We have already introduced this basic structure, which we called *Setoid*. Then we will proceed by successively enriching the type of setoids with further structure, obtaining definitions for *lattice* and *distributive lattice*, until we get the formalization corresponding to Boolean algebras.

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 $\beta : \rho{\to}\textit{type}$, which should be read "$\beta$ 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 is formally written.

$$\langle\langle\langle\rangle, S : [r]Set\rangle, \approx : [r](x, y : r.S)Set\rangle.$$

For the sake of readability, however, in the notation that we are going to use in what follows labels are allowed to participate in the formation of types in the same way as ordinary variables or constants do. Then, they have to be syntactically distinguished from the latter, in order to avoid ambiguities. We do this by writing labels in a distinguished font. This is harmless, since it is possible to give a mechanical procedure to translate from the informal notation, where labels are singled out by means of a particular notation, to the corresponding formal representation involving families of types.

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

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

We have called this type *BinRel*.

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

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

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

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

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

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

We will obtain a formal definition of the system that sustains in a natural way the usual informal reasoning associated to these latter concepts. For example, one makes use in the informal language of the fact that any Boolean algebra is also a distributive lattice; a property ϕ valid for all lattices is directly used as one of Boolean algebras when reasoning about properties of this latter system. In the formal language this is obtained by the (inclusion) polymorphism induced by record types : given a record type ρ_1, it may be 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, also referred sometimes as of *type inclusion*, and β_1 is a *subfamily* of β_2.

In the case of record types, the condition for $\rho_1 \sqsubseteq \rho_2$ is in words as follows: for each field $\mathsf{L} : \beta_2$ in ρ_2 there must be a field $\mathsf{L} : \beta_1$ in ρ_1 with $\beta_1 \sqsubseteq \beta_2 : \rho_1 \to type$. We will show in section 4 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_1 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 departs from the one usually presented in the literature in that it also takes care of the dependencies.

We give a detailed presentation of the extended theory, with the corresponding meaning explanations and justification of the rules of inference, in section 4.

3. Boolean Algebras and DeMorgan's laws formalized

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 introduced in the next section as well as the enunciation of some of the propositions 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 2.1. 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 : (∨) $x \vee y \approx y \vee x$
(∧) $x \wedge y \approx y \wedge x$ (commutative laws)

L2 : (∨) $x \vee (y \vee z) \approx (x \vee y) \vee z$
(∧) $x \wedge (y \wedge z) \approx (x \wedge y) \wedge z$ (associative laws)

L3 : (∨) $x \vee x \approx x$
(∧) $x \wedge x \approx x$ (idempotent laws)

L4 : (∨) $x \approx x \vee (x \wedge y)$
(∧) $x \approx x \wedge (x \vee y)$ (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 to Φ.

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

PROPOSITION 2.1. *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 **S**, whose carrier is the set S and whose (finite) set of operations (or operation symbols) is $\{f_1, \dots , f_k\}$ shall be denoted by $\langle S, f_1, \dots , f_k\rangle$. We shall also use $|\mathbf{S}|$ to stand for the carrier set of the algebra **S**.

We now introduce the following

Definition 2.2. A *distributive lattice* is a lattice which satisfies the following (distributive) laws,

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

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

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

Theorem 2.1. *A lattice L satisfies* D1 *iff it satisfies* D2

Definition 2.3. 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:

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

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

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

3.1.1. *DeMorgan's laws.* To begin with we enunciate some propositions that any Boolean algebra satisfies. In what follows **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 2.2.

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 2.2 and Proposition 2.1.

Proposition 2.3. *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 2.2 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 **B** is a lattice, we can finally use Proposition 2.1 to get the desired conclusion. □

It can readily be verified that using this latter proposition and the postulates B3, any Boolean algebra **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 2.4.

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 2.4, 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 2.3 can directly be applied to get that $\sim(x \vee y) \approx \sim x \wedge \sim y$. $\square$

3.2. The proof checker. A script for the proof checker looks very much like one for a functional programming language. The syntax of input expressions is given by the grammar in Figure 2.1.

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

FIGURE 2.1. Syntax of input expressions

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

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

with T, F and c constant names, x a variable and e, α and α_1,...,α_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 $[x_1][x_2]\ldots[x_n]e$ of type $\alpha_1{\rightarrow}[x_1](\alpha_2{\rightarrow}\ldots(\alpha_n{\rightarrow}[x_n]\alpha)\ldots)$, with $n \geq 0$.

The two first are the counterpart in the system to the nominal definitions of types and families of types introduced in [**BT97**]. The latter form of declaration is not present in the proof-assistant ALF.

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

Any declaration is checked under a current environment. Once the declaration D is checked to be correct, the environment is extended with it. Thereby, the definiendum of D may occur in any declaration introduced after it.

3.2.1. *Let and Use expressions.* The motivation for using *let* expressions as a means to introduce *local* declarations of proofs shares the motivation for using these expression formers in functional languages like *ML* or *Haskell*.

The possibility of abbreviating a proof object by a name, which in turn may occur in what is defined as its valid scope, not only alleviates notation, but may also render the process of proof checking more efficient. The way let expressions are checked in our system is heavily influenced by a proposal by Coquand in [**Coq96**]. Namely, in order to check that an expression *let* $x : \alpha_1 = e_1$ *in* e has a certain type α in a environment $\mathcal{E}$ proceed as follows: check first that $x : \alpha_1 = e_1$ is a valid declaration in $\mathcal{E}$. If this succeeds check then that e is an object of type α in the environment $\mathcal{E}$ locally extended with $x : \alpha_1 = e_1$. The checking of the expression e, in addition to consider that x has type α_1, may also make use of the fact that x is definitionally equal to the expression e_1. This latter is not needed for performing the type checking of a let expression in *ML* or *Haskell*.

Actually, it is possible to define let expressions involving a list of local declarations to the expression e. These, however, can not be mutually defined as they are in the programming languages mentioned above or in the proof-assistant *Alfa* currently being implemented at the Department of Computing Science at Chalmers University.

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

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

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

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

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

FIGURE 2.2. Forms of declaration

3.3.1. *Preliminary definitions.* For the sake of readability we shall deviate a little from the syntax presented in section 3.2 for the forms of declaration and input expressions that the type checker reads. In Figure 2.2 we show how we denote in this section the definition of a type, a family of types and the abbreviation of an object of a certain type. At some points, when there is no interest in showing the code that a constant abbreviates, we make use of declarations of the form $c : \alpha$.

We consider now, in Figure 2.3, 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 *SetRel* 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

$$\begin{aligned}
&\mathit{binOp} : \mathit{Set}{\to}\mathit{type}\\
&\mathit{binOp}\ A = A \to A \to A\\[1ex]
&\mathit{Rel} : \mathit{Set}{\to}\mathit{type}\\
&\mathit{Rel}\ A = A \to A \to \mathit{Set}\\[1ex]
&\mathit{SetRel} : \mathit{type}\\
&\mathit{SetRel} = \langle \mathsf{A} : \mathit{Set}, \mathsf{R} : \mathit{Rel}\ \mathsf{A}\rangle\\[1ex]
&\mathit{RelOp} : \mathit{type}\\
&\mathit{RelOp} = \langle \mathit{SetRel}, \circ : \mathit{binOp}\ \mathsf{A}\rangle\\[1ex]
&\mathit{isTrans} : \mathit{SetRel}{\to}\mathit{type}\\
&\mathit{isTrans}\ B = (x, y, z : B.\mathsf{A})\ B.\mathsf{R}\ x\ y \to B.\mathsf{R}\ y\ z \to B.\mathsf{R}\ x\ z\\[1ex]
&\mathit{isCong} : \mathit{RelOp}{\to}\mathit{type}\\
&\mathit{isCong}\ \mathit{Rop} = \ \boldsymbol{use}\ \mathit{Rop} : \mathit{RelOp}\\
&\qquad\qquad\quad \boldsymbol{in}\ (x, y, z, w : A)\ R\ x\ z \to R\ y\ w \to R\ (\circ\ x\ y)\ (\circ\ z\ w)
\end{aligned}$$

FIGURE 2.3. Types and families of types

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.

$$\begin{aligned}
&\mathit{isComm} : \mathit{RelOp}{\to}\mathit{type}\\
&\mathit{isComm}\ \mathit{Rop} = \boldsymbol{use}\ \ \mathit{Rop} : \mathit{RelOp}\ \ \boldsymbol{in}\ \ (x, y : A)\ R\ (\circ\ x\ y)\ (\circ\ y\ x)\\[1ex]
&\mathit{isAssoc} : \mathit{RelOp}{\to}\mathit{type}\\
&\mathit{isAssoc}\ \mathit{Rop} = \boldsymbol{use}\ \ \mathit{Rop} : \mathit{RelOp}\ \ \boldsymbol{in}\ \ (x, y, z : A)\ R\ (\circ\ x\ (\circ\ y\ z))\ (\circ\ (\circ\ x\ y)\ z)\\[1ex]
&\mathit{isIdemp} : \mathit{RelOp}{\to}\mathit{type}\\
&\mathit{isIdemp}\ \mathit{Rop} = \boldsymbol{use}\ \ \mathit{Rop} : \mathit{RelOp}\ \ \boldsymbol{in}\ \ (x : A)\ R\ (\circ\ x\ x)\ x\\[1ex]
&\mathit{RelOps} : \mathit{type}\\
&\mathit{RelOps} = \langle \mathit{RelOp}, * : \mathit{binOp}\ \mathsf{A}\rangle\\[1ex]
&\mathit{isAbsorb} : \mathit{RelOps}{\to}\mathit{type}\\
&\mathit{isAbsorb}\ \mathit{Rops} = \boldsymbol{use}\ \ \mathit{Rops} : \mathit{RelOps}\ \ \boldsymbol{in}\ \ (x, y : A)\ R\ x\ (\circ\ x\ (*\ x\ y))
\end{aligned}$$

FIGURE 2.4. Axioms of lattices

3.3.2. *Lattices.* We now turn to introduce the constructions corresponding to the ones presented in section 3.1. Thus, we start by defining the type of lattices.

For the representation of this latter notion, and the other systems of algebras there introduced, we adopt the following methodology: we define, first, a record type that acts as the counterpart of the algebra – as defined in section 3.1 – that the system embodies. Then, this latter record type is extended with fields that conform 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 2.4 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 2.5.

$PreLatt : type$

$PreLatt = \langle Setoid, \vee : binOp\ \mathsf{S}, \wedge : binOp\ \mathsf{S}\rangle$

$dualPreLatt : PreLatt \rightarrow PreLatt$

$dualPreLatt\ Pl = \langle Pl, \vee = Pl.\wedge, \wedge = Pl.\vee\rangle$

$opOfLatt : RelOps \rightarrow type$

$$opOfLatt\ Rops = \begin{array}{l} \langle\ \mathsf{cong} : isCong\ Rops, \\ \quad \mathsf{L1} : isComm\ Rops, \\ \quad \mathsf{L2} : isAssoc\ Rops, \\ \quad \mathsf{L3} : isIdemp\ Rops, \\ \quad \mathsf{L4} : isAbsorb\ Rops \\ \rangle \end{array}$$

$Latt : type$

$$\begin{array}{l} Latt = \\ \quad \langle PreLatt, \\ \quad\ \vee\mathsf{Props} : opOfLatt\ \langle \mathsf{A} = \mathsf{S}, \mathsf{R} = \approx, \circ = \vee, * = \wedge\rangle, \\ \quad\ \wedge\mathsf{Props} : opOfLatt\ \langle \mathsf{A} = \mathsf{S}, \mathsf{R} = \approx, \circ = \wedge, * = \vee\rangle \\ \quad \rangle \end{array}$$

$dualLatt : Latt \rightarrow Latt$

$$dualLatt\ L = \begin{array}{l} \langle dualPreLatt\ L, \\ \quad \vee\mathsf{Props} = L.\wedge\mathsf{Props}, \\ \quad \wedge\mathsf{Props} = L.\vee\mathsf{Props} \\ \rangle \end{array}$$

FIGURE 2.5. Lattice

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.

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 $\% - \quad \textit{comment} \quad - \%$ to informally express the property being proved.

The definitions of congR$\vee$ and congL$\wedge$ in Figure 2.6 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.

In Figure 2.7 is the (almost complete) code of the proof that Proposition 2.1 is valid for any lattice L. The notation $\ldots h \ldots$ is used to stress the dependency on the hypothesis h_i.

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

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


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

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

FIGURE 2.6. Congruences

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

$antisymmL$ :  $(L : Latt)\ (x,y : L.\mathsf{S})$
               $L.\approx\ x\ (L.\wedge\ x\ y) \to L.\approx\ x\ (L.\vee\ x\ y) \to L.\approx\ x\ y$
$antisymmL$ =
  $[L\ x\ y\ h_1\ h_2]$
   **let**
        $lemm_1$ : $L.\approx\ x\ (L.\vee\ y\ (L.\wedge\ x\ y)) = \ldots h_1 \ldots$
        $lemm_2$ : $L.\approx\ (L.\vee\ y\ (L.\wedge\ x\ y))\ y = \ldots h_2 \ldots$
   **in**
        $L.\mathsf{trans}\ \ x\ \ (L.\vee\ y\ (L.\wedge\ x\ y))\ \ y\ \ lemm_1\ \ lemm_2$
```

FIGURE 2.7. Antisymmetric property of lattices

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

FIGURE 2.8. Distributive lattice

The type of distributive lattices is shown in Figure 2.8. We declare as well the function *dualDistrLatt*, which behaves as expected.

3.3.3. *Boolean Algebra.* The representation of the system of boolean algebras, the type *BoolAlg* in Figure 2.9, is built up in a similar manner as done for lattices. In order to make the code more legible, however, we chose not to group the axioms corresponding to the operators $\vee$ and $\wedge$. We illustrate *use* expressions in the definition of the function *dualBoolAlg*.

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

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

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

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

FIGURE 2.9. Boolean algebra

3.3.4. *Proof of propositions 2.2-2.4 and DeMorgan laws.* We consider now the presentation of the proofs that were sketched in section 3.1 .

In Figure 2.10 we give the code of the proof of the first part of Proposition 2.2. Notice how *use* expressions improve the readability of the code, there is no need for explicit construction of selections. Moreover, observe that in the type of *prop2.2(i)* the variable *Ba* is used as an object of type *PreLatt*. The objects $trans_2$, $lemm_1$ and $lemm_2$, which are locally declared by means of the *let* constructor, are typical examples of local lemmas. Note that the constants *congR*∨ and *congL*∧ defined in Figure 2.6 for lattices are applied to the variable *Ba* of type *BoolAlg*.

The constants *0identR*∨, *1identL*∧ and *commArgsB3* abbreviate the proofs of three properties which can easily be proved to be derivable from the postulates B1-B4. They are declared in Figure 2.11.

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

prop2.2(i) : (Ba : BoolAlg) (x, y : Ba.S)
             use Ba : PreLatt in ≈ (∧ x y) 0 → ≈ (∼ x) (∨ (∼ x) y)
prop2.2(i) =
  [Ba x y h]
   use Ba : BoolAlg
   in
    let
        trans₂ : (x, y, z, w : S) ≈ x y → ≈ y z → ≈ z w → ≈ x z
          =  [x y z w h₁ h₂ h₃] trans x z w (trans x y z h₁ h₂) h₃
        lemm₁ : ≈ ∼x (∧ (∨ ∼x x) (∨ ∼x y))
          =  trans₂ ∼x (∨ ∼x 0) (∨ ∼x (∧ x y)) (∧ (∨ ∼x x) (∨ ∼x y))
                   (0identR∨ Ba ∼x)
                   (congR∨ Ba ∼x 0 (∧ x y) (symm (∧ x y) 0 h))
                   (D1 ∼x x y)
        lemm₂ : ≈ (∧ (∨ ∼x x) (∨ ∼x y)) (∨ ∼x y)
          =  trans (∧ (∨ ∼x x) (∨ ∼x y)) (∧ 1 (∨ ∼x y)) (∨ ∼x y)
                  (congL∧ Ba (∨ ∼x x) 1 (∨ ∼x y) (commArgsB3 Ba x))
                  (1identL∧ Ba (∨ ∼x y))
    in
        trans ∼x  (∧ (∨ ∼x x) (∨ ∼x y)) (∨ ∼x y) lemm₁ lemm₂
```

FIGURE 2.10. Proposition 2.2(i)

```
%−  $\forall \mathbf{B}.\forall x \in |\mathbf{B}|.\ x \approx x \vee 0$  −%

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

%−  $\forall \mathbf{B}.\forall x \in |\mathbf{B}|.\ 1 \wedge x \approx x$  −%

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

%−  $\forall \mathbf{B}.\forall x \in |\mathbf{B}|.\ {\sim}x \vee x \approx 1$  −%

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

FIGURE 2.11. Simple derived properties of Boolean algebras

Let us now consider the second part of Proposition 2.2. We made, on page 12 of section 3.1, the remark on the duality of the properties enunciated in this latter

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

$\mathit{prop2.2(ii)} : \ (Ba : \mathit{BoolAlg})\ (x, y : Ba.\mathsf{S})$
$\qquad \textbf{\textit{use}}\ Ba : \mathit{PreBoolAlg}\ \textbf{\textit{in}}\ \approx (\vee\ x\ y)\ 1 \to \approx ({\sim}\ x)\ (\wedge\ ({\sim}\ x)\ y)$
$\mathit{prop2.2(ii)} = [Ba]\ \mathit{prop2.2(i)}\ (\mathit{dualBoolAlg}\ Ba)$

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

$\mathit{prop2.3} : \ (Ba : \mathit{BoolAlg})\ (x, y : Ba.\mathsf{S})$
$\qquad \textbf{\textit{use}}\ Ba : \mathit{PreBoolAlg}\ \textbf{\textit{in}}\ \approx (\wedge\ x\ y)\ 0 \to \approx (\vee\ x\ y)\ 1 \to \approx ({\sim}\ x)\ y$
$\mathit{prop2.3} =$
$\quad [Ba\ x\ y\ h_1\ h_2]$
$\quad \mathit{antisymmL}\ Ba\ (Ba.{\sim}\ x)\ y\ (\mathit{prop2.2(i)}\ Ba\ x\ y\ h_1)\ (\mathit{prop2.2(ii)}\ Ba\ x\ y\ h_2)$

FIGURE 2.12. Propositions 2.2(ii) - 2.3

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

$\mathit{prop2.4(i)} : \ (Ba : \mathit{BoolAlg})\ (x, y : Ba.\mathsf{S})$
$\qquad \textbf{\textit{use}}\ Ba : \mathit{PreBoolAlg}\ \textbf{\textit{in}}\ \approx (\wedge\ (\vee\ x\ y)\ (\wedge\ ({\sim}\ x)\ ({\sim}\ y)))\ 0$

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

$\mathit{prop2.4(ii)} : \ (Ba : \mathit{BoolAlg})\ (x, y : Ba.\mathsf{S})$
$\qquad \textbf{\textit{use}}\ Ba : \mathit{PreBoolAlg}\ \textbf{\textit{in}}\ \approx (\wedge\ (\vee\ x\ y)\ (\vee\ ({\sim}\ x)\ ({\sim}\ y)))\ 1$

FIGURE 2.13. Proposition 2.4

proposition, and our intention of obtaining the proof of the dual of a given property Φ on a certain structure S in terms of a proof ϕ of Φ. Accordingly, then, the proof of the part *ii*) of the above proposition is constructed by applying —and with this we mean function application— the object *prop2.2(i)* to the dual structure of *Ba*, i.e. *dualBoolAlg Ba*. This construction is shown in Figure 2.12.

The informal argument given for the validity of Proposition 2.3 is straightforwardly codified, as presented in that same figure. Note the application of *antisymmL* to the variable *Ba* of type *BoolAlg*.

In Figure 2.13 we declare the constants *prop2.4(i)* and *prop2.4(ii)* to stand for the proofs of the two properties of Proposition 2.4. The construction of those proofs is routine.

```
%–   ∀B.∀x, y ∈ |B|.∼(x ∨ y) ≈ (∼x ∧ ∼y)   –%

DeMorgan(i) :  (Ba : BoolAlg) (x, y : Ba.S)
               use Ba : PreBoolAlg in ≈ (∼ (∨ x y)) (∧ (∼ x) (∼ y))
DeMorgan(i) =
  [B x y]
  use B : BoolAlg
  in prop2.3  B (∨ x y) (∧ (∼ x) (∼ y))
              (prop2.4(i) B x y)  (prop2.4(ii) B x y)


%–   ∀B.∀x, y ∈ |B|.∼(x ∧ y) ≈ (∼x ∨ ∼y)   –%

DeMorgan(ii) :  (Ba : BoolAlg) (x, y : Ba.S)
                use Ba : PreBoolAlg in ≈ (∼ (∧ x y)) (∨ (∼ x) (∼ y))
DeMorgan(ii) = [Ba] DeMorgan(i) (dualBoolAlg Ba)
```

FIGURE 2.14. DeMorgan laws

We end up presenting in Figure 2.14 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. The extension

4.1. Formulation of the extension. We now proceed to give the formal stipulation of the extension of type theory with record types and subtyping. We will follow the syntactico-semantical method exposed in [**Mar84**] and used in every formal 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 extended 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, three new forms of judgement are added to those of the original theory. After having introduced them, 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.

The forms of judgement.

4.1.1. *The original forms of judgement.* Let us recall the forms of categorical judgement of type theory:

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

To know that $\alpha : \mathit{type}$ 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 $\alpha_1{=}\alpha_2 : \mathit{type}$ — 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 $\beta : \alpha{\rightarrow}\mathit{type}$.

That two families of types β_1 and β_2 over a type α are the same — in symbols $\beta_1{=}\beta_2 : \alpha{\rightarrow}\mathit{type}$ — means that $\beta_1 a{=}\beta_2 a : \mathit{type}$ 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 [**Mar92, Tas97**]. 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 $x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n$. 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 [**Mar84, Mar87, NPS89**].

It may be useful to remark that we make (nominal) definitions of types and of 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:

$A : \mathit{type}$
$A{=}\alpha : \mathit{type}$.

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 a variable x of type α. Then we define F as a family of types over α by means of the following two axioms:

$F : \alpha{\rightarrow}\mathit{type}$
$Fx{=}\alpha_1 : \mathit{type}\ [x{:}\alpha]$.

Then by virtue of the second axiom, Fa turns out to be the type obtained by substituting a for the occurrences of x in α_1 for $a : \alpha$[1].

[1]In 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.

4.1.2. *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:

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

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 $\beta_1 \sqsubseteq \beta_2 : \alpha {\rightarrow} \mathit{type}$— means that $\beta_1 a \sqsubseteq \beta_2 a$ for every object a of type α.

4.1.3. *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, $\rho : \mathit{record\text{-}type}$. 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 form of judgement $\sigma : \alpha {\rightarrow} \mathit{record\text{-}type}$ 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 a given set is formally written. Families of types are formed by abstraction, which we write using square brackets.

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

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 a type α and $\sigma : \alpha\to type$, that $\sigma : \alpha\to record\text{-}type$ 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.

4.2. 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 [**Mar92, Tas97**].

4.2.1. *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:

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

The first of these rules will be seen later to connect type checking to checking identity of types and thereby identity of objects. 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.

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

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

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

4.2.2. *Informal 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 *type conversion* in the original theory, i.e. for instance the rule:

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

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:

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

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 w.r.t. the presence of the rules above and therefore w.r.t. the judgements of the forms $\alpha_1{=}\alpha_2$:*type* and $\beta_1{=}\beta_2 : \alpha{\rightarrow}\mathit{type}$ that are derivable. But they will not differ w.r.t. the judgements of the forms $\alpha : \mathit{type}$ 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 with the judgements of identity of types and of families of types that can be proved.

4.2.3. *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.

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

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

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

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.

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

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

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.

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

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

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 $f{=}g : \alpha_2{\to}\beta_2$ for given $f : \alpha_1{\to}\beta_1$ and $g : \alpha_1{\to}\beta_1$ such that $f{=}g : \alpha_1{\to}\beta_1$. Then the rule is correct.

In a way analogous to that of the case of families of types we have the following rules.

Rules of function application.
Formation of functions by abstraction.
β-rule and rule of extensionality.

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

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

There are no inclusions between ground types, except for the trivial ones following from the reflexivity of type inclusion.

4.2.5. *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.

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

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

From now on we omit side conditions of rules to the effect that labels are declared at most once in record types. To justify the rules in the first line above 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 $r_1.L{=}r_2.L : \beta r$.

To complete the justification of the rules above it only remains to define what it is to substitute objects for variables in a record type formed by those rules. This is done in the obvious manner, i.e. letting substitution be distributed over the fields of the record type in question. We omit these definitions here.

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

$$\frac{\sigma : \alpha{\to}\textit{record-type} \quad a : \alpha}{\sigma a : \textit{record-type}} \qquad\qquad \frac{\rho : \textit{record-type}\ [x{:}\alpha]}{[x]\rho : \alpha{\to}\textit{record-type}.}$$

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 $R : \textit{record-type}$. 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 $F : \alpha{\to}\textit{record-type}$. 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:

$$\frac{}{\langle\rangle{=}\langle\rangle : \textit{type}} \qquad\qquad \frac{\rho_1{=}\rho_2 : \textit{type} \quad \beta_1{=}\beta_2 : \rho_1{\to}\textit{type}}{\langle\rho_1, L{:}\beta_1\rangle{=}\langle\rho_2, L{:}\beta_2\rangle : \textit{type}}$$

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:

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

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

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 $\beta : \rho'{\to}\textit{type}$. 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 $\beta : \rho'{\to}\textit{type}$ we conclude $\beta : \rho{\to}\textit{type}$ 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:

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

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 $\beta_1 : \rho_1 \to \textit{type}$ and that $r.L : \beta_1 r$. Finally, from the latter and the premiss $\beta_1 \sqsubseteq \beta_2 : \rho_1 \to \textit{type}$, we know $r.L : \beta_2 r$.

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

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

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.

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

The first of these rules requires no justification. The second one will be called of *extension* of record objects. We will also refer to the objects generated by these two rules as *record (object) extensions.* 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:

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

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 $r{=}s : \rho_1$.

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

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.

CHAPTER 3

Type checking: informal explanations and discussion

Type checking in the context of type theory is the task of verifying the formal correctness of a judgement of one of the forms $\alpha : type$ and $a : \alpha$, in general depending on declarations of variables and constants.

We will now describe an algorithm of type checking for the extended theory. For this, it is useful to consider first the problem of type checking in the original theory. For the sake of conciseness, we confine attention to the checking of judgements of the form $a : \alpha$, where we assume that $\alpha : type$.

The formulation of the explanations in sections 1 and 2 are result of joint work with Alvaro Tasistro.

1. Type checking in the original theory

Type checking for systems of typed lambda calculus involves type inference. This is because of applications, whose typing rule (which we show here as it is in type theory),

$$\frac{f : \alpha \to \beta \qquad a : \alpha}{fa : \beta a}$$

is not conservative: information disappears when going from the premisses to the conclusion. Conversely, in order to check the conclusion we need to infer the type $\alpha \to \beta$ of f.

Now, in the presence of dependent types, it is undecidable whether an unlabeled abstraction, i.e. an expression of the form $[x]e$, has a type at all [**Dow93**]. Therefore there is no algorithm for type checking beta redexes. So, in general, for type checking that an expression b has a certain type we have to see to it that b is written in beta normal form. This restriction is inessential in the sense that still every object that can be formed in the theory can be expressed in a way so as to be accepted by the type checking algorithm. For instance, any abstraction $[x]e$ that stands for an object of type $\alpha \to \beta$ can be given a name in type theory, by a definition of the form:

$$\begin{aligned} f &: \alpha \to \beta \\ f &= [x]e \end{aligned}$$

Using this definition we can then express an object $([x]e)a : \beta a$ as fa, which is not anymore a redex.

More precisely, we have that if $b : \alpha$ is valid then there is $b' : \alpha$ such that $b = b' : \alpha$ and $b' : \alpha$ is accepted by the type checking algorithm. In particular, instead of expressions containing beta redexes one has to write their corresponding beta normal forms.

So we have now that applications must be of the form $h\ e_1 \ldots e_m$ where the e_i's are expressions in beta normal form and h, called the *head* of the application, must be either a constant or a variable. We call these expressions *generalized* applications, since they include the constants and the variables as particular cases, i.e. with $m = 0$. The expressions accepted by the type checker are formally specified as follows:

$$\begin{array}{rcl} e & ::= & [x]e \mid f \\ f & ::= & x \mid c \mid fe \end{array}$$

Here x ranges over the variables and c over the constants. Then, the expressions f are the generalized applications. According to the observation made at the beginning, it is for these expressions that type checking links itself with type inference. More precisely, it can be decided whether a generalized application has type or not, by the following

Algorithm (Type inference for generalized applications). To infer the type of a variable x or constant c, just look it up among the declarations.

To infer the type of an expression fe, proceed as follows. First infer a type for the expression f. Supposing the inference is successful, see to it that the type obtained is defined as one of the form $\alpha{\rightarrow}\beta$. Then check whether $e : \alpha$. If this is in turn successful, return the type βe.

Notice that there is at most one declaration for each variable or constant. Then it follows by an inductive argument that a generalized application f has at most one inferred type. As a consequence, if f has type α then it has inferred type α_1 and $\alpha_1{=}\alpha$:*type*. Now the algorithm of type checking is as follows:

Algorithm (Type checking). To check whether $[x]e : \alpha$ see to it first that α is defined as a type of the form $\alpha_1{\rightarrow}\beta_1$. If this is the case, then check whether $e[x := z] : \beta_1 z$, adding $z{:}\alpha_1$ to the declarations of variables, for a fresh variable z.

To check whether $f : \alpha$, infer the type for f. If a type α_1 is obtained, then check whether $\alpha_1{=}\alpha$:*type*.

The last step embodies the use of the rule of type conversion. Thereby type checking is linked with checking judgements of definitional identity. In the next two algorithms, the form of a defined type must be understood to be the form of its ultimate definiens.

Algorithm (Type conversion). Checking type equality proceeds recursively on the form of the types.

For checking the equality of two functional types, $\alpha_1{\rightarrow}\beta_1$ and $\alpha_2{\rightarrow}\beta_2$, α_1 is checked to be equal to α_2 and $\beta_1 x$ is checked to be equal to $\beta_2 x$ adding $x{:}\alpha_1$ to the declarations of variables, for a fresh variable x.

For checking the equality of ground types, check whether they are both the type *Set* or whether they are equal objects of type *Set*.

By virtue of the last step, type checking leads eventually to checking definitional identity of objects, i.e. of judgements $a{=}b : \alpha$.

Algorithm (Object conversion). Checking $a{=}b : \alpha$ proceeds recursively on the type α.

In case α is a ground type, take both a and b to head normal form. Notice that these normal forms cannot be abstractions since they are of ground types. So they must necessarily be generalized applications as defined above. Observe that if an object is in head normal form and its head is a constant, this latter must necessarily be a primitive one. The algorithm proceeds by comparing the heads. In case they are the same constant or variable, h say, it continues by recursively comparing the arguments. For checking the identity of each pair of respective arguments, their (common) type is needed. This is obtained from h, whose type can be recovered from the list of declarations.

For checking $f{=}g : \alpha{\to}\beta$, check whether $fx{=}gx : \beta x$ adding $x{:}\alpha$ to the declarations of variables, for a fresh variable x.

The whole process is guaranteed to terminate if all definitions are well-founded.

The approach used for checking object equality follows the one taken by Magnusson in [**Mag95**]. The process for checking identity of objects having a functional type comprises both α- and η-convertibility. Now, the equality of two objects in the original theory can be checked without using their (common) type, i.e. under the only assumption that they have some type. Concrete algorithms illustrating this are given in [**Coq91, Coq96**]. However, in the presence of record types and subtyping it is not in general possible to check equality of record objects without considering type information.

2. Type checking in the extended theory

Two new forms of expression have to be considered, namely *record extensions* $\langle r, L = a\rangle$ and *selections* $r.L$.

To begin with, notice that the typing rule for selection

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

is also not conservative. In order to check the conclusion we need to infer the type ρ for r. Now, analogously to what is the case for the (unlabeled) abstractions, we cannot decide in general whether an extension $\langle r, L = e\rangle$ has or has not a type. This would in turn require to decide whether the (arbitrary) expression e has a type or not.

There are in addition other difficulties with record object extensions, which we now intend to make clear.

2.1. Type checking of record extensions. Record object extensions are of one of the forms $\langle L_1 = e_1, \dots, L_n = e_n\rangle$ and $\langle f, L_1 = e_1, \dots, L_n = e_n\rangle$, where f is not itself a record extension.

Let us consider first the problem of checking whether $\langle L_1 = e_1, \dots, L_n = e_n\rangle : \rho$. We shall refer to the expression to be checked as r. A possible solution is the following: for checking that r has type ρ, see to it that every label declared in ρ is bound in r to an expression of appropriate type. For this, we can proceed recursively on the components of r that correspond to the labels in ρ.

Now, that a record object extension has a certain type ρ, can be derived in the calculus presented in chapter 2, starting out from $\langle\rangle : \langle\rangle$, by an alternated use, in particular, of the proof rules

$$\frac{r : \rho_1 \quad e : \beta r}{\langle r, L = e\rangle : \langle \rho_1, L{:}\beta\rangle}\ {\scriptstyle L \text{ fresh in } \rho_1} \qquad \frac{r : \rho_1 \quad \rho_1 \sqsubseteq \rho_2}{r : \rho_2}$$

Notice, then, on the one hand, that there could be labels L_i bound to objects in r that do not occur in ρ, due to the use of the subsumption rule. On the other hand, it may also well be the case that some label L occurs bound in r more than once; the rule of record extension allows overriding. Moreover, the objects assigned to the different occurrences of L do not need to be of the same type. From now on we shall call *unreachable* those labels of the object r that either do not occur in ρ or are overridden.

It is clear then that the procedure described above would in general leave components of r unchecked, namely those corresponding to the unreachable labels of r. Since there is no general algorithm for inferring whether an expression has type or not, we cannot by this method ensure the well-formedness of the record object as a whole. However, unchecked components cannot be used without being eventually checked. So, checking only the restrictions imposed by the given type is safe from this point of view. But, on the other hand, the method will still in general violate the principle that correctly typed expressions contain only correctly typed parts and, as a consequence, it would accept expressions that cannot be typed in the theory.

The obvious alternative is just to reject those record objects which contain fields whose labels are not declared in the intended type. This may seem in principle too restrictive, since well formed expression can be rejected by this method. Of especial importance is the case in which we have $\langle L_1 = e_1, \ldots , L_n = e_n\rangle : \rho_1$ but intend to use the record object as of a type ρ_2 with $\rho_1 \sqsubseteq \rho_2$ in the strict sense, i.e. as of a proper supertype of its original type.

These cases can be recovered, however, using auxiliary definitions.
Suppose, for instance, that we want to use $\langle L_1 = e_1, \ldots , L_n = e_n\rangle$ as an object of type ρ_2 and there are labels L unreachable, in the first sense above, in ρ_2. We can give a name r to $\langle L_1 = e_1, \ldots , L_n = e_n\rangle$ and declare it as of a type ρ_1 which, according to the restriction, must contain declarations for all its labels. If this type ρ_1 turns to be a subtype of ρ_2, then we can safely use r as an object of type ρ_2.

Extensions of the form $\langle f, L_1 = e_1, \ldots , L_n = e_n\rangle$, in addition, allow to express a restricted form of overriding, namely, the one that we illustrated in chapter 2 with the definition, for instance, of the function *dualPreLatt*. Notice that it can well be the case that f is a constant that abbreviates a record object extension where some of the labels L_i are bound to objects. But then as f has been defined, we can recover its type and then, as we will show later, we shall not need to inspect the components of f.

There is, then, in principle, a choice between a permissive and a restrictive method. The latter seems to allow for enough expressiveness at the cost of having to introduce additional definitions. This, however, seems not to constitute a problem in practice, especially in the presence of *let* expressions.

For a more detailed discussion of the adequacy of the restrictive method for natural practice we refer to [**Tas97**].

2.2. The algorithm of type checking. We have then pointed out two major problems concerning record extensions. First, it is not possible to decide in general if one such object has a type or not. Therefore, selection redexes of the form $\langle r, L = e\rangle.K$ cannot be accepted as input expressions to the procedure of type checking. But, in a similar manner as suggested for β-redexes in section 1, one can also make use of nominal definitions in order to get rid of redexes as the one above. However, in constrast to the case of abstractions, we must also introduce a restriction on the form of record extensions that can be accepted by a type checking algorithm.

Then, expressions that are not abstractions or record extensions must be of the form $(h\ e_1 \dots e_m).L_1 \dots .L_n$. We call these expressions *generalized selections.* Here the e_i's are expressions in β- and selection-normal form. The L_j's are labels and the head h must now be of one the forms $x.L_1 \dots .L_n$ or $c.L_1 \dots .L_n$.

The syntax of the permissible expressions can be more succinctly formulated as follows:

$$\begin{array}{lcl} e & ::= & [x]e \mid \langle L_1 = e_1, \dots , L_n = e_n\rangle \mid \langle f, L_1 = e_1, \dots , L_n = e_n\rangle \mid f \\ f & ::= & x \mid c \mid (f\ e) \mid f.L \end{array}$$

The expressions f are the generalized selections. The analysis is now the same as for the original theory. We have that it can be decided whether it follows from the declarations of constants and variables that a given generalized selection has type.

Algorithm (Type inference for generalized selections). To infer the type of an expression of any of the forms x, c or (fe) proceed as for the original theory.

For inferring a type for a selection $f.L$, infer first a type for the expression f. If this is successful, see to it that the type obtained is a record type ρ. Then look up for a field $L{:}\beta$ in ρ. If this is found, return the type βf.

Again, generalized selections have at most one inferred type. And then, if a generalized selection f has type α_2, then it has inferred type α_1 and $\alpha_1 \sqsubseteq \alpha_2$. This solves the problem of type checking generalized selections.

We give now an algorithm of type checking based on the restrictive method discussed above.

Algorithm (Type checking). To check whether $[x]e : \alpha$ proceed as for the original theory.

To check $\langle L_1 = e_1, \dots , L_n = e_n\rangle : \alpha$, see to it first that α is a type of the form $\langle L_1 : \beta_1, \dots , L_n : \beta_n\rangle$. If this is the case then, for $i = 1, \dots , n$ check whether $e_i : \beta_i\langle L_1 = e_1, \dots , L_{i-1} = e_{i-1}\rangle$.

For checking $\langle f, L_1 = e_1, \dots , L_n = e_n\rangle : \alpha$, see to it first that α is defined as a type of the form $\langle \rho, L_1 : \beta_1, \dots , L_n : \beta_n\rangle$. If this is the case then check whether $f : \rho$.
In case of a positive answer proceed by checking $e_i : \beta_i\langle f, L_1 = e_1, \dots , L_{i-1} = e_{i-1}\rangle$, for $i = 1, \dots , n$.

We now refer to $\langle f, L_1 = e_1, \ldots, L_n = e_n \rangle$ as r and call the components $L_i = e_i$ the *plain* fields of the extension. Note, first, that for checking that $f : \rho$ we do not need to inspect the components of f. The only condition that we need to require from f is that a type can be inferred for it. In addition, as $\langle \rho, L_1 : \beta_1, \ldots, L_n : \beta_n \rangle$ has been checked to be a valid record type, none of the labels L_i may occur in the record type ρ. Therefore, the selection $r.L_i$ will result in the object bound to L_i in the plain fields of r, which, as it should be, has the type $e_i : \beta\langle f, L_1 = e_1, \ldots, L_{i-1} = e_{i-1} \rangle$.

Finally, to check whether $f : \alpha$ infer the type of f. If a type α_1 is obtained, then check that $\alpha_1 \sqsubseteq \alpha$.

Due to the use of the type subsumption rule, this last step now links type checking with checking judgements of type inclusion.

Algorithm (Type inclusion). The checking of type inclusion proceeds recursively on the form of the types.

For checking that a record type $\langle K_1 : \gamma_1, \ldots, K_n : \gamma_n \rangle$ is included in the record type $\langle L_1 : \beta_1, \ldots, L_m : \beta_m \rangle$ proceed as follows: for $i = 1, \ldots, m$, first look up for a declaration $K_j{:}\gamma_j$ such that $K_j \equiv L_i$. If this turns out to be successful then, for a fresh variable x taken as of type $\langle K_1 : \gamma_1, \ldots, K_{j-1} : \gamma_{j-1} \rangle$, check that $\gamma_j x$ is included in $\beta_i x$.

To check whether a functional type $\alpha_1 \rightarrow \beta_1$ is included in $\alpha_2 \rightarrow \beta_2$, check that α_2 is included in α_1, and $\beta_1 x$ is included in $\beta_2 x$ for a fresh variable x taken as of type α_2.

For checking the inclusion of two ground types, check whether they are both the type *Set* or whether they are equal objects of type *Set*.

We end up with

Algorithm (Object conversion in the extended theory). For checking that r and s are equal objects of type $\langle L_1 : \beta_1, \ldots, L_n : \beta_n \rangle$, check whether $r.L_i{=}s.L_i : \beta_i r$ for $i = 1, \ldots, n$.

In the remaining cases proceed as for the original theory.

3. Towards an implementation of the algorithm

The design of the algorithms informally described in the previous sections, in particular the one for the original theory, closely follows the approach taken by Magnusson in the implementation of the type checking algorithm which is the logical heart of the proof-editor ALF. In addition to the new form of objects, record extensions and selection, the procedure in section 2 for checking that an object has a type also considers the relation of subtyping. This latter modification can be grasped as the replacement of the module for checking type conversion by one which implements the checking of type inclusion. There are, however, some differences in the understanding on how to check whether the type conversion (resp. type inclusion) rule has been applied in the derivation of a judgement $a : \alpha$.

On the other hand, the final implementation of the algorithm, which we present in chapter 5, drastically departs from Magnusson's. We have left unattended some problematic questions in the explanation given for performing the checking that an abstraction has a type. We will adapt ideas by Coquand [**Coq91**] and Pollack [**Pol94a**] to provide a solution to those problems, which we now proceed to discuss.

3.1. Type checking abstractions. In the formulation of the calculus presented in section 4 of chapter 2 the judgements involved in most of the rules are in categorical form, the exception being those rules that introduce binding operators. In these cases, judgements that constitute some of the premisses are made under the assumption that one variable has a certain type. The procedures that constitute the algorithms, however, are formulated as to be performed in the presence of a (valid) list of variable declarations. In other words, what we have in mind is the formal verification of the generalized forms of the judgements of the theory, usually called *hypothetical* or *relative* judgements. What we intend to describe, in particular, with the algorithm for checking that a certain expression a has a type α is how to check the formal correctness of a judgement of the form $\Gamma \vdash a : \alpha$, which says that a is an object of type α under the (valid) context Γ. This observation also applies to the remaining form of judgements involved in the various rules of the calculus.

The way of making sense of the proof rules presented in that same work follows what Martin-Löf has called the syntactico-semantical method. The justification of each individual rule is done by showing that the meaning of the conclusion is contained in the meaning of the premisses. For this, in turn, the semantical explanation of each form of judgement has to be laid down. Let us take for instance the form of judgement above: Let Γ be a context and α a type under Γ. Then $\Gamma \vdash a : \alpha$ means that for any permissible values of the variables in Γ the assignment of these values to the variables in a gives an object of the type obtained from the assignment of those values to the variables in α.

Observe that in this explanation we are assuming that we have already explained what it means to know that Γ is a context, that α is a type under that context, and moreover, what are permissible values of the variables declared in the context Γ. It is precisely the formal treatment of these notions, especially the last one, that differentiates, for instance, the formulation of Martin-Löf's logical framework as presented in [**NPS89**] from the presentation known as the calculus of explicit substitution [**Mar92, Tas97**], from now on referred to as *CES*. In the former work, the notions of context and thereby the explanation of what are permissible values for the variables in the context remain at an informal level. The assignment of values to variables on the expressions of the language is understood as a (meta) operation to be defined over those expressions. On the other hand, one could say that the principal motivation for formulating *CES* is to make sense of these three notions in a completely formal manner. New forms of judgement are introduced to express when a list of variable declarations is a context and what is a construction of a valid assignment of values to the variables of a context. Substitutions then are made explicit in the syntax of the language and the operation of performing a substitution on an expression becomes itself an expression (which is denoted by $e\gamma$ and read as the expression e with the substitution γ). It is for a modified version of *CES* that Magnusson designs and implements the type checking algorithm on top of which ALF's proof-engine is built up.

In both formulations of the framework referred to above a particular system of types is introduced, namely: the function types, the type of sets and for each set, the type of its elements.
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. Thus, as a new way of forming types is introduced in the calculus, the understanding of the judgement $(x:\alpha)\beta$ **:** *type* requires the explanation of what it means to be an object of this type as well as when two such objects are the same: f **:** $(x:\alpha)\beta$ means that fa **:** $\beta[x:=a]$ for any arbitrary object a of type α, and, moreover, $fa{=}fb$ **:** $\beta[x:=a]$ whenever a and b are equal objects of type α. That f and g are equal objects of type $(x:\alpha)\beta$ means that fa and ga are equal objects of type $\beta[x:=a]$ provided that a is an object of type α. In particular, abstraction is introduced as an operation of object formation (of functional types). The corresponding rule

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

is justified by making the following (real) definition: if $x{:}\alpha \vdash b:\beta$ then $[x]b$ is the object of type $(x:\alpha)\beta$ such that if a is an object of type α then $([x]b)a$ is stipulated to be equal to $b[x:=a]$ as object of type $\beta[x:=a]$. This latter stipulation is meaningful because the meaning of the relative form of judgement $x{:}\alpha \vdash b:\beta$ has previously been explained, namely, to know this judgement means to know that $b[x:=a]$ **:** $\beta[x:=a]$ for any object a of type α.

Now, the justification of the generalized formulation of the rule of abstraction

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

can be done in analogous manner as above once we have explained the meaning of the form of judgement $\Gamma \vdash a:\alpha$, and therefore what is the knowledge that we have in the presence of the premiss $\Gamma, x{:}\alpha \vdash b:\beta$. Now, as already said, in particular we should know that $\Gamma, x{:}\alpha$ is a context.

The stipulation for the formation of a context $\Gamma, x{:}\alpha$ in *CES*, 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, in the first place, 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 $A{\to}*$. In [**Mag95**] Magnusson rephrases this example and also shows that the same situation arises in *CES*. 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 proceeds 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. There is no problem, however, in deriving, and also checking, that $[x][y]y : (x : A)(y : Px)Px$. This latter shows that the proof system is not closed with respect to α-conversion.

The decision taken by Magnusson in order to be able to perform the checking that an abstraction has a certain type in the way described above is to restrict the bound variables of the abstraction to be mutually distinct and different from the variables occurring in the context under which the checking is taking place. Therefore, terms like $[x][x]x$ are rejected by the type checking algorithm.

Now, according to the explanation of what it means to be an object of a functional type one could argue that it makes sense to say that $[x][x]x$ is an object of type $(x : A)(y : Px)Px$: let a and p be objects of type A and Pa respectively. Assume now that the substitution of an expression a for a variable x is defined as to have no effect when performed on an expression of the form $[x]b$. Then, the application of $[x][x]x$ to the objects a and p would result in the object p of type Pa. Observe that the restriction imposed on the operation of substitution is respected by both the usual definition of substitution in λ-calculus and the one given for the objects of *CES*. On the other hand, it is clear that in the stipulation of making an assumption, or more precisely, how a context Γ may be extended by a declaration $x{:}\alpha$ to form the context $\Gamma, x{:}\alpha$, the variable x must be required not to already occur in Γ, or as commonly said, it has to be *fresh* for Γ.

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. We believe that this was the first formulation of a calculus in which parameters are used for expressing (relative) judgements about types and objects of certain types. Parameters are chiefly exploited by Coquand in the definition of the algorithm for checking type and object conversion: For checking, for instance, that two abstractions, $[x]e$ and $[y]d$ say, are convertible, first a parameter, p say, is substituted for the variable x (resp. y) in e (resp. d). Then the algorithm proceeds recursively by checking the convertibility of $e[x := p]$ and $d[y := p]$. 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. Due to the inextricable relation between type and object formation and type equality, and therefore object equality, the results of this work have a direct application to the construction of type checking algorithms for theories with dependent types. The algorithms we have sketched in the previous sections of this chapter for checking conversion and inclusion of types as well as conversion of objects are much in the spirit of Coquand's algorithm. There is a difference, however, in that the checking of the conversion of two objects is performed with respect to some (common) type. We have already pointed out that in the presence of record objects and subtyping we may have that the equality of two expressions as objects of a certain type may depend on which is the type being considered.

In [**Pol94a**] Pollack adopts the use of parameters to implement a type checking algorithm for a family of PTS's. In that work, the author starts by presenting the original formulation of the rules of PTS . Then in the strive for obtaining an algorithm out of the inference rules the system is successively modified. 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. The benefit afforded by the use of parameters can be illustrated as follows: let us consider again the question of checking the judgement $[x][x]x : (x : \alpha)(y : Px)Px$. We rephrase the argument given above for the validity of this particular judgement in terms of type checking:

For checking that an expression $[x]e$ has a type $(x : \alpha)\beta$ under a context Γ see to it that $e[x := p]$ has type $\beta[x := p]$ with Γ extended with the declaration $p{:}\alpha$ with p a fresh parameter for Γ. The operation $e_2[x := e_1]$ is defined as textual substitution but it has no effect when performed on an abstraction whose bound variable equals the variable x. Thus, according to the explanation above, we proceed by checking that $([x]x)[x := p]$ has type $((y : Px)Px)[x := p]$ under the context extended with $p{:}\alpha$. Notice that this reduces to checking that $[x]x$ has type $(y : Pp)Pp$. Now we should check that $x[x := q]$ (which is q) has type $Pp[y := q]$ (which is Pp) after extending the context with $q{:}Pp$, which is easily seen to be correct.

It could be argued that this procedure could still be carried out, as we have done, using variables: just choose a fresh variable for the context and then proceed as described above. But this would not be enough, because this variable might at the same time occur as a bound variable in the expression on which the substitution is performed. Therefore, a mechanism of renaming has also to be considered in the definition of the operation of substitution in order to avoid variable capture. This is not needed in the language we are considering because parameters are not subjected to bindings.

The formulation of the rule of abstraction formation presented in [**Pol94a**] is, in spirit, as follows:

$$\frac{\Gamma, p{:}\alpha \vdash b[x := p] : \alpha_1[y := p]}{\Gamma \vdash [x]b : (y : \alpha)\alpha_1}$$

The restriction for this rule is now that the parameter p must not occur neither in Γ, b nor α_1. Observe that the proceedure described above for checking that an abstraction has a certain type conforms with the upward reading of this latter rule. It should also be noticed that the bound variables of the object and the type are not required to be the same. This is a further modification to the original rule of abstraction of PTS's –and also to the corresponding rule in both formulations of Martin-Löf's logical framework– where the two bound variables are required to be the same. The motivation provided by Pollack for this latter change is also influenced by the intended understanding of the type system as closed by α-conversion. We now rephrase the example and the arguments given, which can also be applied to the formulation of type theory in [**NPS89**]:

Let us take, for instance, the judgement $\Gamma \vdash [X][y]y : (X : \mathit{Set})(w : X)X$. With the restriction that the bound variables of the object and type have to be the same, the only possible way of deriving this judgement, up to applications of the thinning rule, would be to derive that $[y]y$ has type $(y : p)p$ under the context $\Gamma, p{:}\mathit{Set}$ and then, provided that $(y : p)p$ and $(w : p)p$ can be proved to be convertible types, apply the rule of type conversion to get that $\Gamma, p{:}\mathit{Set} \vdash [y]y : (w : p)p$. Notice then that this application of type conversion is actually a step of α-conversion, and that this latter can be avoided once the abstraction rule is formulated as above.

In our case, however, there is no need for a formal stipulation of the rule that makes explicit that the bound variables may be distinct. The rule of abstraction we have in mind is the following:

$$\frac{\Gamma, p{:}\alpha \vdash b[x := p] : \alpha_1[x := p]}{\Gamma \vdash [x]b : \alpha \to [x]\alpha_1}$$

In this case the parameter p also must be fresh for Γ and not occur in b nor in α_1. Now the question is whether $\Gamma \vdash [x]b : \alpha \to [y]\alpha_1$ is derivable. We will later show that having the rule of abstraction above it is possible to derive that if $\Gamma, p{:}\alpha \vdash b[x := p] : \beta p$ holds then $\Gamma \vdash [x]b : \alpha \to \beta$ using the η-rule for families of types. Observe, then, that we do not need to make explicit the bound variable of the family β. Therefore, we will obtain a formulation of the procedure for checking an abstraction very close to the one stipulated in the preceding sections, namely: for checking that $[x]b$ has type α, first see to it that α is of the form $\alpha_1 \to \beta_1$. Then proceed by checking that $b[x := p]$ is of type $\beta_1 p$ after extending the context with the declaration that the fresh parameter p is of type α_1.

Now the question is how one can get convinced of the validity of the above rule for abstraction formation, and furthermore, of the whole set of the generalized form of the rules that constitute the calculus. This is what we shall set ourselves to do in the next chapter.

CHAPTER 4

Formulation of the extended theory with parameters

1. Introduction

We shall now proceed to present a variant of the formulation of the logical framework extended with record types and subtyping given in chapter 2. A first difference with this presentation is that we shall consider the rules of inference in their generalized form. Therefore, the corresponding justifications must now be given in accordance to the meaning explanation of the relative forms of judgement. Further, we shall 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.

For the justification of the rules of inference that the calculus embodies, we intend to follow the method adopted by Per Martin-Löf of making sense of the proof rules that constitute the systems presented, for instance, in [**Mar87, NPS89, Tas97**] : each inference rule is justified (or explained) by showing that the meaning expressed by the conclusion of the rule is contained in the meaning of the premisses. For that, all the forms of judgement used in the formulation of the rules have to be made explicit and given their corresponding explanation.

We are, however, especially concerned with the understanding of the interaction between parameters and binders. Particular attention is then paid to the justification of the rules for introducing abstraction operators. There is no counterpart to these explanations in the works by Coquand [**Coq91**] and Pollack [**Pol94a**].

We are interested in the particular system of types introduced in chapter 2, namely: the type of sets, for each individual set the type of its elements, dependent function types and dependent record types. The explanation of what it means to know a certain type α is independent of the particular syntax chosen to formally express such a type. This is also the case for the explanation of what it means that a is an object of a type α. When one knows a type or an object of a type one knows more than the corresponding expression (or syntactic object) used to express it. An expression α becomes a type when it is explained which is the semantical category that α denotes. Correspondingly, it is precisely when an expression is sorted into a semantical category that it becomes an object. Thus, we are allowed to assert that an expression a is an object of type α only if we already know that α is a type and besides we can show that the understanding of the meaning of a is that of an object of type α.

In principle, there should be no knowledge associated to a primitive expression of the language other than the one that allows to sort it into its category, the syntactical category. The primitive objects of the various types are introduced by means of a real definition, that is to say, by a direct meaning explanation. The relation of synonymy or sameness of meaning between linguistic expressions will be understood as identity between objects. Thereby two different expressions are synonymous only if they are equal as objects of a certain type. We will need to stipulate, at some point, the equality of two objects. As this latter equality can be expressed as an instance of a form of judgement of the calculus, these (nominal) definitions can then be formally expressed as rules of the proof system.

Now, for the explanation of the relative forms of judgement of the theory a notion of substitution has to be introduced. As already said, in *CES*, the category of expressions is extended to consider expressions with substitutions as part of the language of the calculus. Due to the introduction of new forms of judgement, whose meaning explanations are precisely laid down, it is possible to give a complete and detailed justification of all the rules that constitute the system of proof rules. Furthermore, there is no need, in general, for the definition of how substitutions are performed on expressions to justify many of the rules involving expressions with substitutions. On the other hand, when a definition is required, it can still be formally expressed as a rule of the calculus. A general and very precise formulation of the system is then obtained. The solution we have in mind for the problems discussed in section 3 of chapter 3 connected with the binding operators, however, strongly relies on the definition of two different operation of substitutions, one for parameters and the other for variables. These two operations will be defined on the elements of the syntactical category, thereby we shall need to give a precise definition of this category. We then will lose some of the generality accomplished in the various formulations of the logical framework we have been making reference to in the sense that we are defining a priori which are the valid expressions of the language of the calculus. Nevertheless, we see this as a natural consequence of the task we have undertaken: the implementation of the mechanical verification of the formal correctness of the forms of judgements and inference rules of a particular formulation of the theory. On the other hand, and this we think may be a more serious drawback of this proposal, the (syntactical) identification of different linguistic expressions introduced by the operations of substitution will render the justification of some of the rules to depend on (meta-) properties proper to the category of expressions. As these properties can not be expressed in terms of the judgements of the theory we are also losing formality in the formulation and justification of some of the rules. In this respect, we think that a formulation of the calculus of explicit substitution which considers the distinction between parameters and variables would remedy this latter situation. This formulation, however, has still to be further investigated.

We now then proceed to introduce the category of expressions and give the corresponding definition of the operations of substitution.

2. The category of expressions

The expressions of the language are given by the grammar in Figure 4.1

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

FIGURE 4.1. Syntax of expressions

The symbol x ranges over a denumerable set V, the set of bound variables. Below we use y as an element of this set too. We also assume there exists a denumerable set P of parameters. The symbol p (and q below) ranges over the set P. The symbol c ranges over a countable set C of constants, which is defined to be disjoint with V. The sort *Set* is a distinguished element of C. Finally, the symbol L ranges over a denumerable set L of labels. This set is defined to be disjoint with the sets P, V and C.

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

We assume that P, V, C and L are equipped with a decidable equivalence relation. Under this assumption we can also define one for expressions. We denote it by $=$, and make an overloaded use of it.

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 $\alpha{\to}[x]\alpha_1$.

2.1. Instantiation and Substitution. We need two kinds of substitution, substitution of an expression for a parameter, that we will call *instantiation* and denote by $e_2[e_1/p]$, and substitution of an expression for a variable, that we will call *substitution* and denote by $e_2[x := e_1]$. The first one is just textual substitution. The latter also behaves as textual substitution but has no effect when performed on an abstraction whose bound variable equals the variable x. It does not prevent capture either. The corresponding definitions are given in Figure 4.2 and Figure 4.3 respectively.

We will introduce a notion of *well-formedness* for expressions. The intuition is that we will consider to be well-formed those expressions where only bound occurrences of variables are allowed. The predicate *wf* on the expressions is inductively defined as shown in Figure 4.4.

What we here define to be well-formed is what in [**Pol94a**] are defined to be *closed* expressions. In [**Coq91**] they are defined as elements of the set EXP. The intuition is exactly the same, in a well-formed expression there are no occurrences of free variables. This is particularly made explicit by the rule *wf-Lda* (x is the only variable that may occur free in e) and the fact that variables are not well-formed (no introduction rule for this case). We shall now enunciate propositions that characterize to some extent the interplay of substitution with well-formed expressions.

$$
\begin{array}{llll}
x[e_1/p] & =_{def} & x & \\
q[e_1/p] & =_{def} & e_1 & \text{if } p = q \\
 & =_{def} & q & \text{if } p \neq q \\
c[e_1/p] & =_{def} & c & \\
([x]e_2)[e_1/p] & =_{def} & [x]e_2[e_1/p] & \\
fe_2[e_1/p] & =_{def} & (f[e_1/p])(e_2[e_1/p]) & \\
(\alpha{\rightarrow}\beta)[e_1/p] & =_{def} & (\alpha[e_1/p]){\rightarrow}\beta[e_1/p] & \\
\langle\rangle[e_1/p] & =_{def} & \langle\rangle & \\
\langle e, L = e'\rangle[e_1/p] & =_{def} & \langle e[e_1/p], L = e'[e_1/p]\rangle & \\
e.L[e_1/p] & =_{def} & (e[e_1/p]).L & \\
\langle e, L{:}e'\rangle[e_1/p] & =_{def} & \langle e[e_1/p], L{:}e'[e_1/p]\rangle &
\end{array}
$$

FIGURE 4.2. Instantiation

$$
\begin{array}{llll}
y[x := e_1] & =_{def} & e_1 & \text{if } x = y \\
 & =_{def} & y & \text{if } x \neq y \\
p[x := e_1] & =_{def} & p & \\
c[x := e_1] & =_{def} & c & \\
([y]e_2)[x := e_1] & =_{def} & [y]e_2 & \text{if } x = y \\
 & =_{def} & [y]e_2[x := e_1] & \text{if } x \neq y \\
fe_2[x := e_1] & =_{def} & (f[x := e_1])(e_2[x := e_1]) & \\
(\alpha{\rightarrow}\beta)[x := e_1] & =_{def} & (\alpha[x := e_1]){\rightarrow}\beta[x := e_1] & \\
\langle\rangle[x := e_1] & =_{def} & \langle\rangle & \\
\langle e, L = e'\rangle[x := e_1] & =_{def} & \langle e[x := e_1], L = e'[x := e_1]\rangle & \\
e.L[x := e_1] & =_{def} & (e[x := e_1]).L & \\
\langle e, L{:}e'\rangle[x := e_1] & =_{def} & \langle e[x := e_1], L{:}e'[x := e_1]\rangle &
\end{array}
$$

FIGURE 4.3. Substitution

(wf-Par): $\dfrac{}{wf\ \ p}$ (wf-Con): $\dfrac{}{wf\ \ c}$

(wf-Lda): $\dfrac{wf\ \ e[x := p]}{wf\ \ [x]e}$ (wf-App): $\dfrac{wf\ \ f \quad wf\ \ e}{wf\ \ fe}$

(wf-ERec): $\dfrac{}{wf\ \ \langle\rangle}$ (wf-RecO): $\dfrac{wf\ \ e \quad wf\ \ e'}{wf\ \ \langle e, L = e'\rangle}$ (wf-Sel): $\dfrac{wf\ \ e}{wf\ \ e.L}$

(wf-Fun): $\dfrac{wf\ \ \alpha \quad wf\ \ \beta}{wf\ \ \alpha{\rightarrow}\beta}$ (wf-RecT): $\dfrac{wf\ \ e \quad wf\ \ e'}{wf\ \ \langle e, L{:}e'\rangle}$

FIGURE 4.4. Well-formed expressions

$lgth\ p$	$=_{def}$	1
$lgth\ x$	$=_{def}$	1
$lgth\ c$	$=_{def}$	1
$lgth\ fe$	$=_{def}$	$lgth\ f + lgth\ e$
$lgth\ [x]e$	$=_{def}$	$1 + lgth\ e$
$lgth\ \alpha{\to}\beta$	$=_{def}$	$lgth\ \alpha + lgth\ \beta$
$lgth\ \langle\rangle$	$=_{def}$	1
$lgth\ \langle e_1, L = e_2\rangle$	$=_{def}$	$lgth\ e_1 + lgth\ e_2$
$lgth\ e.L$	$=_{def}$	$1 + lgth\ e$
$lgth\ \langle e_1, L{:}e_2\rangle$	$=_{def}$	$lgth\ e_1 + lgth\ e_2$

FIGURE 4.5. Length of an expression

2.2. Properties of well-formed expressions. Some of the properties below are proved by complete induction on the length of the expressions. This function, in turn, is defined in Figure 4.5. We shall here enunciate the propositions that we consider relevant for the understanding of the work that follows. Their proofs, as well as those of some auxiliary lemmas, can be found in Appendix A.

PROPOSITION 4.1. *Given expressions e_1 and e_2, such that wf e_2, and any variable x and parameter p, then*

1) $e_2[x := e_1] = e_2$.

2) if wf $(e_1[x := p])$ *then wf* $(e_1[x := e_2])$.

The intuition behind the first proposition is that well-formed expressions are not affected by substitution. The second one says that if the result of substituting in an expression a parameter for a variable is a well-formed expression, this will also be the case if the variable is replaced by any well-formed expression.

2.3. Closed expressions. In the following we will talk of *closed* expressions. As anticipated, the valid open expressions that participate in a relative judgement will depend on parameters not on variables. Therefore, we shall need a notion of closed expression that says more than the one traditionally used in languages with binding operators. For doing that, we first introduce the notion of *independence* of an expression e of a parameter p. The inductive definition of this predicate on expressions is given in Figure 4.6.

2.3.1. *Independence, substitution and instantiation.* Now we enunciate a proposition about the interaction of these three notions.

PROPOSITION 4.2. *Let e_1 and e_2 be expressions and p be a parameter such that e_1 indep p, then*

1) $e_1[e_2/p] = e_1$.

2) for any variable x, $e_1[x := p][e_2/p] = e_1[x := e_2]$.

(indep-Par): $$\frac{q \neq p}{q\ \mathit{indep}\ p}$$
(indep-Var): $$\frac{}{x\ \mathit{indep}\ p}$$

(indep-Con): $$\frac{}{c\ \mathit{indep}\ p}$$

(indep-Lda): $$\frac{e\ \mathit{indep}\ p}{[x]e\ \mathit{indep}\ p}$$
(indep-App): $$\frac{f\ \mathit{indep}\ p \quad e\ \mathit{indep}\ p}{fe\ \mathit{indep}\ p}$$

(indep-ERec): $$\frac{}{\langle\rangle\ \mathit{indep}\ p}$$
(indep-RecO): $$\frac{e\ \mathit{indep}\ p \quad e'\ \mathit{indep}\ p}{\langle e, L = e'\rangle\ \mathit{indep}\ p}$$

(indep-Sel): $$\frac{e\ \mathit{indep}\ p}{e.L\ \mathit{indep}\ p}$$

(indep-Fun): $$\frac{\alpha\ \mathit{indep}\ p \quad \beta\ \mathit{indep}\ p}{\alpha{\to}\beta\ \mathit{indep}\ p}$$
(indep-RecT): $$\frac{e\ \mathit{indep}\ p \quad e'\ \mathit{indep}\ p}{\langle e, L{:}e'\rangle\ \mathit{indep}\ p}$$

Figure 4.6. Independence

Thus, the instantiation of a parameter that does not occur in an expression does not affect that expression.

Finally, a closed expression is then defined as follows:

Definition (Closed expression). An expression e is closed if and only if e is well-formed and for all parameters p the expression e is also independent of p .

2.4. Properties of closed expressions. To begin with, we enunciate the proposition that says that closed expressions are affected neither by substitution nor instantiation.

Proposition 4.3. *Given expressions e_1 and e_2 variable x and parameter p. If e_1 is a closed expression then $e_1[x := e_2] = e_1$ and $e_1[e_2/p] = e_1$.*

We end up this section with some very useful properties relating closed expressions, substitution and instantiation.

Proposition 4.4. *Let e_1, e_2 and a be expressions, and x any variable. Further, let a_i and p, p_i, with $i = 1..n$, be n closed expressions and $n+1$ mutually distinct parameters, respectively.*
Then

1) $e_1[a/p][a_1/p_1, \ldots, a_n/p_n] = e_1[a_1/p_1, \ldots, a_n/p_n][a[a_1/p_1, \ldots, a_n/p_n]/p]$.
2) $e_1[x := e_2][a_1/p_1, \ldots, a_n/p_n] = e_1[a_1/p_1, \ldots, a_n/p_n][x := e_2[a_1/p_1, \ldots, a_n/p_n]]$.
3) $e_1[x := p][a_1/p_1, \ldots, a_n/p_n] = e_1[a_1/p_1, \ldots, a_n/p_n][x := p]$.

We now turn to introduce the various forms of judgement and give the corresponding meaning explanations

3. Forms of judgement

The categorical forms of judgement of the calculus are the following:

α **:** *type*, to be read "α is a type"
$\alpha{=}\beta$ **:***type*, to be read "α and β are equal types"
a **:** α, to be read "a is an object of type α"
$a{=}b$ **:** α, to be read "a and b are equal objects of type α"
β **:** $\alpha{\rightarrow}$*type*, to be read "β is a family of types over α"
$\beta_1{=}\beta_2$ **:** $\alpha{\rightarrow}$*type*, to be read "β_1 and β_2 are the same family of types over α"
ρ **:** *record-type*, to be read "ρ is a record type"
σ **:** $\alpha{\rightarrow}$*record-type*, to be read "σ is a family of record types over α"
$\alpha \sqsubseteq \beta$, to be read "α is a subtype of β"
$\beta_1 \sqsubseteq \beta_2$ **:** $\alpha{\rightarrow}$*type*, to be read "β_1 is a subfamily of β_2"

The meaning of each of these forms of judgement has already been explained in chapter 2.

3.1. The relative forms of judgement. The basic forms of judgements above are generalized in order to express also hypothetical judgements, i.e. judgements which are made under assumptions. From now on we will refer to them as *relative* judgements. Making an assumption is formally reflected by the introduction of a parameter, and the stipulation of how parameters may be introduced gives rise to the notion of a context. It is possible, then, to make judgements involving open expressions, namely, expressions which depend on the parameters of a context.

We start then by the notion of a context and that of being a type under a given context. These two concepts have to be simultaneously explained because contexts are extended by assumptions of the form $p{:}\alpha$, where α has to be a type under a shorter context. The *empty context* (denoted by $[]$) is the context with no assumptions. Let Γ be a context, if α is a type under Γ and p is a parameter not occurring in Γ then $\Gamma, p{:}\alpha$ is the *non-empty* context which results from extending Γ with the assumption that p is a generic object of type α.

Assume now that Γ is a context, the relative judgement $\Gamma \vdash \alpha : \mathit{type}$ says that α is a type under the context Γ. The meaning of this form of judgement when Γ is the empty context is the same as the one given for the corresponding categorical one. When Γ is of the form $[p_1{:}\alpha_1, \ldots, p_n{:}\alpha_n]$, with $n > 0$, a judgement of the form

$$[p_1{:}\alpha_1, \ldots, p_n{:}\alpha_n] \vdash \alpha : \mathit{type}$$

means that $\alpha[a_1/p_1, \ldots, a_n/p_n]$ is a type whenever a_1 is a closed object of type α_1, a_2 of type $\alpha_2[a_1/p_1]$,..., a_n of type $\alpha_n[a_1/p_1, \ldots, a_{n-1}/p_{n-1}]$ and $\alpha[a_1/p_1, \ldots, a_n/p_n]$ and $\alpha[b_1/p_1, \ldots, b_n/p_n]$ are equal types whenever a_1 and b_1 are equal closed objects of type α_1, ..., a_n and b_n of type $\alpha_n[a_1/p_1, \ldots, a_{n-1}/p_{n-1}]$.

In general, then, a context will be of the form

$$[p_1{:}\alpha_1, \dots, p_n{:}\alpha_n]$$

where

- α_1 is a type,
- $\alpha_2[a_1/p_1]$ is a type for any object a_1 of type α_1, and $\alpha_2[a_1/p_1]$ and $\alpha_2[b_1/p_1]$ are equal types whenever a_1 and b_1 are equal objects of type α_1,

 ...
- $\alpha_n[a_1/p_1, \dots, a_{n-1}/p_{n-1}]$ is a type for arbitrary objects a_1 of type α_1, a_2 of type $\alpha_2[a_1/p_1]$,..., a_{n-1} of type $\alpha_{n-1}[a_1/p_1, \dots, a_{n-2}/p_{n-2}]$. Moreover, it is also the case that $\alpha_n[a_1/p_1, \dots, a_{n-1}/p_{n-1}]$ and $\alpha_n[b_1/p_1, \dots, b_{n-1}/p_{n-1}]$ are equal types for equal objects a_1 and b_1 of type α_1, ... , a_{n-1} and b_{n-1} of type $\alpha_{n-1}[a_1/p_1, \dots, a_{n-2}/p_{n-2}]$

Notice that any initial segment of a context is itself a context.

The expression $\alpha[a_1/p_1, \dots, a_n/p_n]$ denotes the result of performing the instantiation of a_n for p_n in the expression $\alpha[a_1/p_1, \dots, a_{n-1}/p_{n-1}]$, for $n \geq 1$. Observe, however, that the objects $a_1, a_2 \dots, a_n$ are respectively closed expressions of type $\alpha_1, \alpha_2[a_1/p_1], \dots,$ $\alpha_n[a_1/p_1, \dots, a_{n-1}/p_{n-1}]$. Therefore, as all the parameters are mutually distinct, the order in which the instantiation of the parameter p_i by the expression a_i is performed is not relevant.

We shall now introduce a notational device : if $[p_1{:}\alpha_1, \dots, p_n{:}\alpha_n]$ is a context, and $a_1, a_2 \dots, a_n$ are respectively closed expressions of type $\alpha_1, \alpha_2[a_1/p_1], \dots,$ $\alpha_n[a_1/p_1, \dots, a_{n-1}/p_{n-1}]$ we shall say that γ is an assignment for the variables in the context Γ and denote by $e\gamma$ the expression $e[a_1/p_1, \dots, a_n/p_n]$. Thus, γ is not itself a construction of the language, it only makes sense when occurring in an expression of the form above. We principally want with this to alleviate the meta notation to be used in the rest of the work. On the other hand, we would like to think of γ as an environment for the context $[p_1{:}\alpha_1, \dots, p_n{:}\alpha_n]$, and then to use this intuition when presenting the explanation of the remaining forms of judgements as well as when providing the justification of the rules we shall present below. In addition to this, if a_1 and b_1 are equal objects of type α_1, ... , a_{n-1} and b_{n-1} of type $\alpha_n[a_1/p_1, \dots, a_{n-1}/p_{n-1}]$ we shall say that γ and δ are the equal assignments $[a_1/p_1, \dots, a_{n-1}/p_{n-1}]$ and $[b_1/p_1, \dots, b_{n-1}/p_{n-1}]$ respectively.

We reformulate now the explanation of the form of judgement $\Gamma \vdash a : \alpha$ which says that a is an object of type α under the context Γ. Assume $\Gamma \vdash \alpha : type$. Then, the meaning of a judgement $\Gamma \vdash a : \alpha$ when Γ is the empty context is the same as the one given for the corresponding categorical one. When Γ is of the form $[p_1{:}\alpha_1, \dots, p_n{:}\alpha_n]$, with $n > 0$, a judgement of the form

$$[p_1{:}\alpha_1, \dots, p_n{:}\alpha_n] \vdash a : \alpha$$

means that $a\gamma : \alpha\gamma$ and that $a\gamma$ and $a\delta$ are equal objects of type $\alpha\gamma$ whenever γ and δ are equal assignments for the context Γ.

The meaning of the remaining forms of relative judgement can now be explained in analogous manner as done for the form of judgements above.

4. Rules of inference

We shall now build up a system of rules that assembles the concepts we have previously explained. The various forms of judgement of the calculus are the following:

Γ *context*
$\Gamma \vdash \alpha : type$
$\Gamma \vdash \alpha_1 = \alpha_2 : type$
$\Gamma \vdash a : \alpha$
$\Gamma \vdash a = b : \alpha$
$\Gamma \vdash \beta : \alpha \rightarrow type$
$\Gamma \vdash \beta_1 = \beta_2 : \alpha \rightarrow type$
$\Gamma \vdash \rho : record\text{-}type$
$\Gamma \vdash \sigma : \alpha \rightarrow record\text{-}type$
$\Gamma \vdash \alpha_1 \sqsubseteq \alpha_2$
$\Gamma \vdash \beta_1 \sqsubseteq \beta_2 : \alpha \rightarrow type$

The rules of the calculus are classified as follows: general rules, instantiation rules and the rules for families of types and types. We shall principally concentrate on the presentation and justification of this latter group of rules, which introduce the particular system of types we want to consider. The general rules correspond to a great extent to the ones presented in the formulation of *CES* in [**Tas97**]. In addition, there are the generalized form of the rules presented in chapter 2 concerning the new form of judgements introduced by the extension of the theory with subtyping. The instantiation rules express in a formal manner the meaning of the relative forms of judgement.

We do not intend to make a comprehensive presentation of the rules of the calculus in this section, the formulation of the whole system can be found in Appendix B.

4.1. General rules.

context formation:

$$\frac{}{[]\ context} \qquad\qquad \frac{\Gamma\ context \quad \Gamma \vdash \alpha : type}{\Gamma, p{:}\alpha\ context}\ {\scriptstyle p \text{ fresh in } \Gamma}$$

Let Γ and Δ be contexts. If every parameter declaration in Γ is also a parameter declaration in Δ then we will say that Γ is a *subcontext* of Δ. This relation is written $\Gamma \preceq \Delta$. Following the terminology in [**Tas97**] we shall also say that Δ is an *extension* of Γ. There are rules then expressing that if we know a relative judgement under a given context we also know it under any extension of the context. Instead of presenting the complete set of rules we just formulate the rule schema:

thinning:

$$\frac{\Gamma \vdash\ J}{\Delta \vdash\ J}\ {\scriptstyle \Gamma \preceq \Delta}$$

where J stands for any of the categorical forms of judgement introduced in the section above.

From the semantical explanations of Δ being a context, an expression being an object of a type under a context and the definition of the relation $\Gamma \preceq \Delta$ the following rules of object and type formation are obtained:

assumption:

$$\frac{}{\Gamma \vdash p : \alpha}\ {\scriptstyle p\,:\,\alpha \text{ in } \Gamma} \qquad\qquad \frac{}{\Gamma \vdash \alpha : \mathit{type}}\ {\scriptstyle p\,:\,\alpha \text{ in } \Gamma}$$

4.1.1. *Equality rules.* These are the general rules for the equality relation of types, objects of a type and families of types. Their justification is done as in previous formulations of type theory. We have then *reflexivity, symmetry and transitivity* rules for identity of types, objects of types and families of types under a given context.

4.1.2. *Rules of inclusion.* We have also rules for expressing that the inclusion of two types and two families of types follows from their identity, as well as the *reflexivity* and *transitivity* of inclusion of types. The following rules are immediately justified from the meaning explanation of the judgement of inclusion:

subsumption:

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

4.2. Rules of instantiation. The rules of instantiations are given in Figure 4.7. They can all be justified in a similar manner. Let us take for instance the first rule of instantiation of types:

$$\frac{\Gamma, p{:}\alpha \vdash \alpha_1 : \mathit{type} \quad \Gamma \vdash a : \alpha}{\Gamma \vdash \alpha_1[a/p] : \mathit{type}}$$

For justifying this rule what has to be shown is that if γ is an assignment for the context Γ then $\alpha_1[a/p]\gamma$ is a type and that if γ and δ are equal assignment for the context Γ then $\alpha_1[a/p]\gamma$ and $\alpha_1[a/p]\delta$ are equal types. We will show only the first of these conditions, the second follows by a similar reasoning. Now, assume the premisses of the rule. That a is an object of type α under Γ gives that $a\gamma : \alpha\gamma$. Then $\gamma[a\gamma/p]$ is an assignment for the context $\Gamma, p{:}\alpha$. The first premiss of the rule then gives that $\alpha_1\gamma[a\gamma/p] : \mathit{type}$. From this same premiss we get that p does not belong to Γ, thus it is different from all the parameters of γ.We can apply then Proposition 4.4 to get that $\alpha_1[a/p]\gamma$ is equal to the expression $\alpha_1\gamma[a\gamma/p]$, hence $\alpha_1[a/p]\gamma : \mathit{type}$.

All the remaining rules can be explained analogously.

4.3. Rules for families of types and types. We shall now give the rules for using and forming families of types.

The rules of application and those expressing the meaning of identity and inclusion of type families are the generalized form of the ones presented in chapter 2.

instantiation of types:

$$\frac{\Gamma, p{:}\alpha \vdash \alpha_1 : \mathit{type} \quad \Gamma \vdash a : \alpha}{\Gamma \vdash \alpha_1[a/p] : \mathit{type}} \qquad \frac{\Gamma, p{:}\alpha \vdash \alpha_1 = \alpha_2 : \mathit{type} \quad \Gamma \vdash a = b : \alpha}{\Gamma \vdash \alpha_1[a/p] = \alpha_2[b/p] : \mathit{type}}$$

$$\frac{\Gamma, p{:}\alpha \vdash \alpha_1 \sqsubseteq \alpha_2 \quad \Gamma \vdash a = b : \alpha}{\Gamma \vdash \alpha_1[a/p] \sqsubseteq \alpha_2[b/p]}$$

instantiation of objects:

$$\frac{\Gamma, p{:}\alpha \vdash b : \alpha_1 \quad \Gamma \vdash a : \alpha}{\Gamma \vdash b[a/p] : \alpha_1[a/p]} \qquad \frac{\Gamma, p{:}\alpha \vdash b_1 = b_2 : \alpha_1 \quad \Gamma \vdash a = c : \alpha}{\Gamma \vdash b_1[a/p] = b_2[c/p] : \alpha_1[a/p]}$$

instantiation of families of types:

$$\frac{\Gamma, p{:}\alpha \vdash \beta \mathbin{\mathbf{:}} \alpha_1 \rightarrow \mathit{type} \quad \Gamma \vdash a : \alpha}{\Gamma \vdash \beta[a/p] \mathbin{\mathbf{:}} \alpha_1[a/p] \rightarrow \mathit{type}} \qquad \frac{\Gamma, p{:}\alpha \vdash \beta_1 = \beta_2 : \alpha_1 \rightarrow \mathit{type} \quad \Gamma \vdash a = b : \alpha}{\Gamma \vdash \beta_1[a/p] = \beta_2[b/p] : \alpha_1[a/p] \rightarrow \mathit{type}}$$

$$\frac{\Gamma, p{:}\alpha \vdash \beta_1 \sqsubseteq \beta_2 : \alpha_1 \rightarrow \mathit{type} \quad \Gamma \vdash a = b : \alpha}{\Gamma \vdash \beta_1[a/p] \sqsubseteq \beta_2[b/p] : \alpha_1[a/p] \rightarrow \mathit{type}}$$

instantiation of record types and record families:

$$\frac{\Gamma, p{:}\alpha \vdash \rho : \mathit{record\text{-}type} \quad \Gamma \vdash a : \alpha}{\Gamma \vdash \rho[a/p] : \mathit{record\text{-}type}} \qquad \frac{\Gamma, p{:}\alpha \vdash \sigma \mathbin{\mathbf{:}} \alpha_1 \rightarrow \mathit{record\text{-}type} \quad \Gamma \vdash a : \alpha}{\Gamma \vdash \sigma[a/p] \mathbin{\mathbf{:}} \alpha_1[a/p] \rightarrow \mathit{record\text{-}type}}$$

FIGURE 4.7. Rules of instantiation

The formal treatment we make in this work of families of types was first presented in the formulation of *CES*. In particular the abstraction operator is introduced as a type family former. As discussed in section 3, we shall make use of the distinction between parameters and bound variables to provide a solution to the problems connected with the binding operators. The rule for family formation then is formulated as:

$$\frac{\Gamma, p{:}\alpha \vdash \alpha_1[x := p] : \mathit{type}}{\Gamma \vdash [x]\alpha_1 \mathbin{\mathbf{:}} \alpha \rightarrow \mathit{type}} \; {\scriptstyle \alpha_1 \; \mathit{indep} \; p}$$

The intuition is the same: for $[x]\alpha_1$ to be a family with index α then α_1 has to be a type for any possible value of type α. The difference is that the notion of a generic object of certain type, the parameter p of type α, is not identified with the notion of variable of that type.

We need to differentiate the cases of Γ being empty and non-empty for the justification of this rule. In the first case we have to show that $[x]\alpha_1 \mathbin{\mathbf{:}} \alpha \rightarrow \mathit{type}$. This in turn requires that for any object a of type α, $([x]\alpha_1)a$ is a type. From the premiss we know then that $\alpha_1[x := p][a/p]$ is a type, and by Proposition 4.2, as α_1 *indep* p we obtain that $\alpha_1[x := a]$ is a type. We can then make the following (meaningful) definition: If $p{:}\alpha \vdash \alpha_1[x := p] : \mathit{type}$ and α_1 *indep* p then $[x]\alpha_1$ is a type family over α such that $([x]\alpha_1)a$ is equal to $\alpha_1[x := a]$ as type if $a : \alpha$.

Now, if Γ is non-empty what has to be shown is that if γ is an assignment for the context Γ then $([x]\alpha_1)\gamma : \alpha\gamma{\rightarrow}\mathit{type}$. For this, in turn, we have to show that if $a : \alpha\gamma$ then $(([x]\alpha_1)\gamma)a$ is a type. Notice first that by definition of instantiation the expression $([x]\alpha_1)\gamma$ is equal to $[x]\alpha_1\gamma$. Observe now that if γ is an assignment for Γ then by the meaning explanation of the relative form of judgement $\Gamma \vdash \alpha : \mathit{type}$ it follows from the premiss that $p{:}\alpha\gamma \vdash \alpha_1[x := p]\gamma : \mathit{type}$. Then, by Proposition 4.4 as p is fresh for Γ we get that $\alpha_1[x := p]\gamma$ is equal to $\alpha_1\gamma[x := p]$. Observe that if α_1 *indep* p the expression $\alpha_1\gamma$ is also independent of p. Therefore, we can use the definition above to get that $[x]\alpha_1\gamma$ is a type family over $\alpha\gamma$ and, moreover, that if a is an object of type $\alpha\gamma$ then $([x]\alpha_1\gamma)a$ is the type $\alpha_1\gamma[x := a]$.

For the justification of the β-rule:

$$\frac{\Gamma, p{:}\alpha \vdash \alpha_1[x := p] : \mathit{type} \quad \Gamma \vdash a : \alpha}{\Gamma \vdash ([x]\alpha_1)a = \alpha_1[x := a] : \mathit{type}} \ {\scriptstyle \alpha_1\ \mathit{indep}\ p}$$

we must, in addition to the constructions above, use Proposition 4.4 to show that $\alpha_1[x := a]\gamma$ is equal to $\alpha_1\gamma[x := a\gamma]$.

From the explanation of what it is for two families of types to be identical the following rule of extensionality can also be justified:

$$\frac{\Gamma, p{:}\alpha \vdash \beta_1 p = \beta_2 p : \mathit{type}}{\Gamma \vdash \beta_1 = \beta_2 : \alpha{\rightarrow}\mathit{type}} \ {\scriptstyle \beta_1, \beta_2\ \mathit{indep}\ p}$$

Its explanation is analogous to the one given below for the rule of extensionality of objects of a function type.

Using the two latter rules it is possible to derive the following formulation of the η-rule for families of types:

$$\frac{\Gamma \vdash \beta : \alpha{\rightarrow}\mathit{type}}{\Gamma \vdash \beta = [x]\beta x : \alpha{\rightarrow}\mathit{type}} \ {\scriptstyle \mathit{wf}\ \beta}$$

That the family β is required to be well-formed is equivalent to ask for the variable x not to occur free in β. We show now a tree-like derivation of the rule:

$$\cfrac{\cfrac{\cfrac{\cfrac{\Gamma \vdash \beta : \alpha{\rightarrow}\mathit{type}}{\Gamma, q{:}\alpha, p{:}\alpha \vdash (\beta x)[x := p] : \mathit{type}} \quad \cfrac{}{\Gamma, q{:}\alpha \vdash q : \alpha}}{\Gamma, q{:}\alpha \vdash ([x]\beta x)q = \beta q : \mathit{type}}}{\Gamma \vdash [x]\beta x = \beta : \alpha{\rightarrow}\mathit{type}}}{}$$

From the premiss $\Gamma \vdash \beta : \alpha{\rightarrow}\mathit{type}$ we can infer by thinning and the first rule of application that $\Gamma, q{:}\alpha, p{:}\alpha \vdash \beta p : \mathit{type}$. Now, observe that by definition of substitution and the side condition we know, using Proposition 4.1, that the expression βp is equal to the expression $(\beta x)[x := p]$. A similar reasoning is applied when the expressions involve the parameter q instead of p. The two final steps are applications of the β-rule and the rule of extensionality for objects of functional type respectively. The conclusion of the η-rule then is obtained by the rule of symmetry for the identity of families of types. Actually, we could drop the side condition on the well-formedness of the expression β, it is possible to show that if $\Gamma \vdash \beta : \alpha \to type$ then both β and α are well-formed. For the application of the β-rule we need the expression βx to be independent of the parameter p. This also holds because of the premiss $\Gamma \vdash \beta : \alpha \to type$ and $\Gamma, q{:}\alpha, p{:}\alpha$ being a context.

4.4. Sets and elements of sets. We introduce the type of (inductively defined) sets, the rule that expresses that each set gives rise to the type of its elements as well as the one stating that equal sets give rise to equal types without further comment:

$$\frac{}{\vdash Set : type} \qquad \frac{\Gamma \vdash A : Set}{\Gamma \vdash A : type} \qquad \frac{\Gamma \vdash A = B : Set}{\Gamma \vdash A = B : type}$$

We remark that the inclusion between ground types A and B is the trivial one following from their identity.

4.5. Function types. Function types now are introduced and explained in the following way: If α is a type and β is a family of types indexed by α we can then form the type of functions that when applied to an object a of type α results in an object of type βa. This function type is denoted by $\alpha \to \beta$. In addition, for $f : \alpha \to \beta$, fa and fb have to be equal objects of type βa if a and b are equal objects of type α. Observe, then, that in contrast to the explanation given in chapter 3 for function types of the form $(x : \alpha)\beta$, this explanation of what it means for f to be an object of type $\alpha \to \beta$ needs no explicit mentioning the index of the family.

These are the rules for forming function types and the associated equality and inclusion rules:

formation of $\alpha \to \beta$:

$$\frac{\Gamma \vdash \alpha : type \quad \Gamma \vdash \beta : \alpha \to type}{\Gamma \vdash \alpha \to \beta : type}$$

equality and inclusion of $\alpha \to \beta$:

$$\frac{\Gamma \vdash \alpha_1 = \alpha_2 : type \quad \Gamma \vdash \beta_1 = \beta_2 : \alpha_1 \to type}{\Gamma \vdash \alpha_1 \to \beta_1 = \alpha_2 \to \beta_2 : type}$$

$$\frac{\Gamma \vdash \alpha_2 \sqsubseteq \alpha_1 \quad \Gamma \vdash \beta_1 \sqsubseteq \beta_2 : \alpha_2 \to type}{\Gamma \vdash \alpha_1 \to \beta_1 \sqsubseteq \alpha_2 \to \beta_2}$$

The justification of these rules reduces to the case in which the judgements involved are categorical. These explanations, in turn, have already been given in chapter 2.

We proceed now to give the rules of application, object formation and the associated identity rules:

application:

$$\frac{\Gamma \vdash f : \alpha \to \beta \quad \Gamma \vdash a : \alpha}{\Gamma \vdash fa : \beta a} \qquad \frac{\Gamma \vdash f = g : \alpha \to \beta \quad \Gamma \vdash a = b : \alpha}{\Gamma \vdash fa = gb : \beta a}$$

It suffices to remember the meaning of the first premiss to get convinced of the validity of these two rules.

We now introduce the rule for abstraction:

abstraction:

$$\frac{\Gamma, p{:}\alpha \vdash b[x := p] : \alpha_1[x := p]}{\Gamma \vdash [x]b : \alpha \to [x]\alpha_1} \; {\scriptstyle b,\, \alpha_1 \; indep \; p}$$

The justification of this rule follows a similar reasoning as the one used for the formation of a family of types. We will only look at the case of Γ being empty. What has to be shown, then, is that $[x]b$ is an object of type $\alpha \to [x]\alpha_1$. So, let a be an object of type α, thus we have to show that $([x]b)a$ is of type $([x]\alpha_1)a$. Observe that from the premiss we get that $[x]\alpha_1$ is a family over α. Thus we know that $([x]\alpha_1)a{=}\alpha_1[x := a]$ **:***type* and also that $b[x := p][a/p]$ **:** $\alpha_1[x := p][a/p]$. Therefore, as both b and α_1 are independent of p, we can apply Proposition 4.2 to get that $b[x := p][a/p]$ (resp. $\alpha_1[x := p][a/p]$) is equal to $b[x := a]$ (resp. $\alpha_1[x := a]$), and then we have that $b[x := a]$ **:** $\alpha_1[x := a]$. So we make the following definition:
if $p{:}\alpha \vdash b[x := p] : \alpha_1[x := p]$ then the expression $[x]b$ is an object of type $\alpha \to [x]\alpha_1$, and moreover, if a is an object of type α we stipulate that $([x]b)a$ is equal to $b[x := a]$ as objects of type $\alpha_1[x := a]$.

The former constructions are expressed by the following formulation of the β-rule

β-conversion:

$$\frac{\Gamma, p{:}\alpha \vdash b[x := p] : \alpha_1[x := p] \quad \Gamma \vdash a : \alpha}{\Gamma \vdash ([x]b)a = b[x := a] : \alpha_1[y := a]}$$

The following rule of extensionality can also be justified:

extensionality:

$$\frac{\Gamma, p{:}\alpha \vdash fp = gp : \beta p}{\Gamma \vdash f = g : \alpha \to \beta} \; {\scriptstyle f,\, g,\, \beta \; indep \; p}$$

Observe that the premisses to the effect that f and g are objects of type $\alpha \to \beta$ under Γ have been omitted. To see that the conclusion is valid we must convince ourselves that $f\gamma$ and $g\gamma$ are equal objects of type $\alpha \to \beta\gamma$ if γ is an assignment for Γ. This type, in turn, is by definition of instantiation equal to $\alpha\gamma \to \beta\gamma$. We have, then, to show that if a is an arbitrary object of type $\alpha\gamma$ then $(f\gamma)a$ and $(g\gamma)a$ are equal objects of type $(\beta\gamma)a$. Notice that $\gamma[a/p]$ is an assignment for the context $\Gamma, p{:}\alpha$ then the premiss gives that $(fp)\gamma[a/p]$ and $(gp)\gamma[a/p]$ are equal objects of type $(\beta p)\gamma[a/p]$. By definition of instantiation, the application $(fp)\gamma[a/p]$ is equal to $(f\gamma[a/p])(p\gamma[a/p])$ and similarly with $(gp)\gamma[a/p]$ and $(\beta p)\gamma[a/p]$. Finally, as f, g and β are independent of the parameter p and the assignment γ does not affect p, we get that $(f\gamma)a{=}(g\gamma)a$ **:** $(\beta\gamma)a$.

Using the two rules above is then possible to show that the following formulations of the η and ξ-rule are derivable:

η and ξ-rules:

$$\frac{\Gamma \vdash b : \alpha \to \beta}{\Gamma \vdash [x]bx = b : \alpha \to \beta}\ \mathit{wf}\ b \qquad \frac{\Gamma, p{:}\alpha \vdash f[x := p] = g[x := p] : \alpha_1[x := p]}{\Gamma \vdash [x]f = [x]g : \alpha \to [x]\alpha_1}\ f, g, \alpha_1\ \mathit{indep}\ p$$

4.6. Record Types. The formulation of the rules for record types and record objects remains almost identical to the one presented in chapter 2. In most of the cases, the only difference is that now the judgements involved in the rules are generalized to their relative form. For each of these rules its justification reduces to the one given in that chapter for the corresponding categorical formulation. This is in accordance with the fact that no binding operators are introduced in the formation of record types and record objects. The exception are the rules concerned with the formation and use of families of record types, but their explanation are analogous to the already given above for families of types.

We then start by giving the formation rules of (primitive) record types:

formation of record-types:

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

type formation:

$$\frac{\Gamma \vdash \rho : \mathit{record\text{-}type}}{\Gamma \vdash \rho : \mathit{type}}$$

A remark is in place concerning the three rules above. In chapter 2 for the formation of record types four rules are introduced which have to be simultaneously understood. In these rules it is expressed that every (primitive) record type has to be introduced as a type and further as a record type. The motivation for doing that was to give a uniform explanation for nominal definitions of types. For the sake of conciseness we here obviate those considerations. However, there must still be a rule that says that every record type gives rise to a type, because, for instance, in the right premiss of the second formation rule above ρ has to be a type in order for that judgement to be correct.

The rules for constructing identical (primitive) record types are as follows:

record types equality:

$$\frac{}{\vdash \langle\rangle = \langle\rangle : \mathit{type}} \qquad \frac{\Gamma \vdash \rho_1 = \rho_2 : \mathit{type} \quad \Gamma \vdash \beta_1 = \beta_2 : \rho_1 \to \mathit{type}}{\Gamma \vdash \langle \rho_1, L{:}\beta_1 \rangle = \langle \rho_2, L{:}\beta_2 \rangle : \mathit{type}}$$

Their justification is routine.

In order to justify the rules of inclusion of record types the following three rules are needed:

$$\frac{}{\Gamma \vdash \langle \rho, L{:}\beta \rangle \sqsubseteq \rho} \qquad \frac{}{\Gamma \vdash \beta : \rho \to \mathit{type}}\ L : \beta \text{ in } \rho \qquad \frac{\Gamma \vdash r : \rho}{\Gamma \vdash r.L : \beta r}\ L : \beta \text{ in } \rho$$

We now proceed to formulate and explain the rules of inclusion of record types:

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

We will justify only the second rule, the first one can easily be seen to be valid. Now for the conclusion of the second rule to be valid we have to show that if γ is an assignment for the context Γ then $\rho_1\gamma$ is a subtype of $\langle \rho_2, L{:}\beta_2 \rangle\gamma$. For this, in turn, it has to be shown that if r is an object of type $\rho_1\gamma$ then it is also an object of type $\langle \rho_2, L{:}\beta_2 \rangle\gamma$, and if r and s are equal objects of type $\rho_1\gamma$ then they are also equal objects of type $\langle \rho_2, L{:}\beta_2 \rangle\gamma$. Now, by definition of instantiation on expressions of the form $\langle \rho, L{:}\beta \rangle$ we have that $\langle \rho_2, L{:}\beta_2 \rangle\gamma$ is equal to $\langle \rho_2\gamma, L{:}\beta_2\gamma \rangle$. Moreover the first premiss gives that $\rho_1\gamma \sqsubseteq \rho_2\gamma$ and the second that $\beta_1\gamma \sqsubseteq \beta_2\gamma : \rho_1\gamma \rightarrow \mathit{type}$. The justification proceeds now as the one given for the categorical formulation of the rule in chapter 2. The second condition is explained in a similar manner.

Finally we introduce the various rules of record object formation and the associated equality rules:

record object extension:

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

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

equality rules:

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

$$\frac{\Gamma \vdash r = s : \rho}{\Gamma \vdash r.L = s.L : \beta r}\ {\scriptstyle L\,:\,\beta \text{ in } \rho}$$

5. Weak head reduction

No notion of reduction for the expressions of the calculus has been introduced so far. The meaning explanations of the forms of judgement of the framework do not depend on any such notion. However, in order to define an algorithm for checking the formal correctness of judgements of the theory we shall introduce one such notion. Its use will render the checking process more efficient, but on the other hand, a proof of subject reduction for the forms of judgement $\Gamma \vdash \alpha : \mathit{type}$ and $\Gamma \vdash a : \alpha$ shall be required when proving the correctness of the algorithm.

Following [**Coq91**], we introduce a *weak head reduction* relation, which is inductively defined as shown in Figure 4.8. A remark is in place concerning the notion of reduction we have introduced. In next section, when we say that $e_1 \Rightarrow e_2$, we shall know that e_1 has already been proved to be a type family, a type or an object of a type. Therefore, in the first place, by Proposition 4.8 in next section, we know that

e_1 is a well-formed expression. Furthermore, by Proposition 4.5 below we shall also know that e_2 is well-formed.

$$p \Rightarrow p$$

$$Set \Rightarrow Set$$

$$\alpha{\to}\beta \Rightarrow \alpha{\to}\beta$$

$$\langle\rangle \Rightarrow \langle\rangle$$

$$\langle\rho, L{:}\beta\rangle \Rightarrow \langle\rho, L{:}\beta\rangle$$

$$[x]e \Rightarrow [x]e$$

$$\langle r, L = e\rangle \Rightarrow \langle r, L = e\rangle$$

$$\frac{f \Rightarrow [x]e \quad e[x := a] \Rightarrow v}{fa \Rightarrow v}$$

$$\frac{f \Rightarrow f_1}{fa \Rightarrow f_1 a}\ {\scriptstyle f_1 \neq [x]e}$$

$$\frac{r \Rightarrow \langle r_1, L_1 = e\rangle \quad e \Rightarrow v}{r.L \Rightarrow v}\ {\scriptstyle L = L_1}$$

$$\frac{r \Rightarrow \langle r_1, L_1 = e\rangle \quad r_1.L \Rightarrow v}{r.L \Rightarrow v}\ {\scriptstyle L \neq L_1}$$

$$\frac{r \Rightarrow r_1}{r.L \Rightarrow r_1.L}\ {\scriptstyle r_1 \neq \langle r_2, L_2 = e\rangle}$$

FIGURE 4.8. Weak head reduction

We shall say that e_1 has *weak head normal form* e_2 iff $e_1 \Rightarrow e_2$.

It is clear from the definition of $\Rightarrow$ that an expression can have at most one weak head normal form.

PROPOSITION 4.5. *The relation $\Rightarrow$ preserves well-formedness, i.e. if wf e_1 and $e_1 \Rightarrow e_2$ then wf e_2.*

PROOF. This proposition can easily been proved by nested structural induction, first on $e_1 \Rightarrow e_2$ and then on *wf* e_1.

The interesting case is when e_1 is of the form fa, $f \Rightarrow [x]e$ and $e[x := a] \Rightarrow v$. That *wf* fa gives that both, *wf* f and *wf* a. Then, by induction hypothesis we obtain, first, that *wf* $([x]e)$. Thus, by definition of well-formedness we can assume that *wf* $(e[x := p])$, for any parameter p. Therefore, as *wf* a, we can finally apply Proposition 4.1 to obtain that *wf* $e[x := a]$. Induction finally gives that *wf* v. $\square$

6. Basic meta-properties of the calculus

PROPOSITION 4.6. *Let* Γ *be a valid context and* $\Gamma \vdash J$. *If a parameter* p *occurs in* J *then there exists a declaration* $p{:}\alpha$ *in* Γ.

PROPOSITION 4.7. *Let* Γ *be a valid context and* p *be a fresh parameter for* Γ

$th_1 \diamond$ *If* $\Gamma \vdash \alpha : type$ *then* α *indep* p.

$th_2 \diamond$ *If* $\Gamma \vdash \beta : \alpha{\to}type$ *then* β *indep* p.

$th_3 \diamond$ *If* $\Gamma \vdash a : \alpha$ *then* a *indep* p.

PROOF. The proof proceeds by simultaneous induction on the derivation of $\Gamma \vdash \alpha : type$ and $\Gamma \vdash \beta : \alpha{\to}type$ for the cases th_1 and th_2, and induction on the derivation of $\Gamma \vdash a : \alpha$ in case th_3. $\square$

PROPOSITION 4.8. *Let* Γ *be a valid context,*

$th_1 \diamond$ *If* $\Gamma \vdash \alpha : type$ *then wf* α.

$th_2 \diamond$ *If* $\Gamma \vdash \beta : \alpha{\to}type$ *then wf* β.

$th_3 \diamond$ *If* $\Gamma \vdash a : \alpha$ *then wf* a.

PROOF. These are also proved by induction on the derivations of the judgements $\Gamma \vdash \alpha : type$, $\Gamma \vdash \beta : \alpha{\to}type$ and $\Gamma \vdash a : \alpha$ respectively. $\square$

PROPOSITION 4.9. *Let* Γ *be a valid context.*
Then,

$th_1 \diamond$ *If* $\Gamma \vdash \alpha : type$ *and* $\alpha \Rightarrow \alpha_1$
then $\Gamma \vdash \alpha_1 : type$ *and* $\Gamma \vdash \alpha = \alpha_1 : type$.

$th_2 \diamond$ *If* $\Gamma \vdash \beta : \alpha{\to}type$ *and* $\beta \Rightarrow \beta_1$
then $\Gamma \vdash \beta_1 : \alpha{\to}type$ *and* $\Gamma \vdash \beta = \beta_1 : \alpha{\to}type$.

$th_3 \diamond$ *If* $\Gamma \vdash \rho : record\text{-}type$ *and* $\rho \Rightarrow \rho_1$
then $\Gamma \vdash \rho_1 : record\text{-}type$ *and* $\Gamma \vdash \rho = \rho_1 : type$.

$th_4 \diamond$ *If* $\Gamma \vdash \sigma : \alpha{\to}record\text{-}type$ *and* $\sigma \Rightarrow \sigma_1$
then $\Gamma \vdash \sigma_1 : \alpha{\to}record\text{-}type$ *and* $\Gamma \vdash \sigma = \sigma_1 : \alpha{\to}type$.

PROOF. We simultaneously prove the proposition by induction on the definition of the relation $\Rightarrow$. The cases when α is the type *Set* or a function type, and ρ is a primitive record type, are trivial, because no reduction is performed, by definition of $\Rightarrow$. Then, reflexivity of type equality is used to prove the second part of the proposition. For families the proof is also straigthforward because there are only two possible ways of deriving a judgement of that form: either $\beta(\sigma)$ is introduced as an abstraction of the form $[x]\alpha_1$, and thereby no reduction is performed, or by an application of subtyping, and in this case induction allows to prove the desired conclusion. The interesting cases are when the type(record) $\alpha(\rho)$ is the application of a type family β to an object of the appropriate type. The reasoning used to prove the proposition, however, is analogous to the one for applications when proving subject reduction for objects of a type, which is Proposition 4.10 below.

□

PROPOSITION 4.10. *Let* Γ *be a valid context,* $\Gamma \vdash \alpha : type$ *and* $\Gamma \vdash a : \alpha$.
If $a \Rightarrow a_1$ *then* $\Gamma \vdash a_1 : \alpha$ *and* $\Gamma \vdash a = a_1 : \alpha$.

PROOF. The proof proceeds by structural induction on the derivation of $a \Rightarrow a_1$. If a is either a parameter p, a record object or an abstraction the proof is trivial, because in this cases $a \Rightarrow a$. Then hypothesis and reflexivity of object equality give the desired result. As function types and record types cannot be objects of any type they are not considered. The interesting cases are when a is either an abstraction or a selection.

$a = fb$ ▷ We have to consider two cases

1) $a_1 = v$, with $f \Rightarrow [x]f_2$ and $f_2[x := b] \Rightarrow v$.

2) $a_1 = f_1 b$, with $f \Rightarrow f_1$ and f_1 is not an abstraction.

Now, we have that if $\Gamma \vdash fb : \alpha$ then

i) $\Gamma \vdash f : \alpha_1 \rightarrow \beta$

ii) $\Gamma \vdash b : \alpha_1$, and

iii) $\Gamma \vdash \beta b \sqsubseteq \alpha$

Let us now consider the first case above.
Induction gives that $\Gamma \vdash [x]f_2 : \alpha_1 \rightarrow \beta$ and that $\Gamma \vdash f = [x]f_2 : \alpha_1 \rightarrow \beta$. The former, in turn, gives that for any fresh parameter p, $\Gamma, p{:}\alpha_1 \vdash f_2[x := p] : \beta p$. Thus, we get by instantiation rule that $\Gamma \vdash f_2[x := p][b/p] : \beta p[b/p]$. Further, by Proposition 4.7 we also know that f_2 and β are independent of p. Therefore, by Proposition 4.2 we get then that $f_2[x := p][b/p] = f_2[x := b]$ and, moreover, by definition of instantiation $\beta p[b/p] = \beta b$.
Then $\Gamma \vdash f_2[x := b] : \beta b$.

We can now apply induction using that $f_2[x := b] \Rightarrow v$ to get $\Gamma \vdash v : \beta b$ and $\Gamma \vdash f_2[x := b] = v : \beta b$. From $\Gamma \vdash f = [x]f_2 : \alpha_1 \rightarrow \beta$ and $\Gamma \vdash b : \alpha_1$, we obtain, by application rule, that $\Gamma \vdash fb = ([x]f_2)b : \beta b$. The rule of β-conversion and transitivity of equal objects then give that $\Gamma \vdash fb = v : \beta b$. The application of the corresponding rules of subsumption lead to the desired conclusions.

As to the second case, by induction we get that $\Gamma \vdash f_1 : \alpha_1 \rightarrow \beta$. Therefore, we directly get the proof of the property by using the rules of application.

$a = r.L$ ▷ There are three possible values for a_1

1) $a_1 = v$, with $r \Rightarrow \langle r_1, L_1 = e \rangle$, $e \Rightarrow v$ and $L = L_1$

2) $a_1 = v$, with $r \Rightarrow \langle r_1, L_1 = e\rangle$, $r_1.L \Rightarrow v$ and $L \neq L_1$

3) $a_1 = r_1.L$, with $r \Rightarrow r_1$ and r is not a record object

If $\Gamma \vdash r.L : \alpha$ then

i) $\Gamma \vdash r : \rho$

ii) $L : \beta$ in ρ

iii) $\Gamma \vdash \beta r \sqsubseteq \alpha$

Let us take the first case. We know by induction that

iv) $\Gamma \vdash \langle r_1, L = e\rangle : \rho$, and

v) $\Gamma \vdash r = \langle r_1, L = e\rangle : \rho$.

By iv) and Proposition 4.11 below we get $\Gamma \vdash e : \beta\langle r_1, L = e\rangle$ and that $\Gamma \vdash \langle r_1, L = e\rangle.L = e : \beta\langle r_1, L = e\rangle$. So, induction gives that $\Gamma \vdash v : \beta\langle r_1, L = e\rangle$, and then it is also of type βr . On the other hand, we can apply the second rule of selection to v) to get that $\Gamma \vdash r.L = \langle r_1, L = e\rangle.L : \beta r$. Finally, then, using transitivity we get that $\Gamma \vdash r.L = v : \beta r$.

As to the second case, we now know by induction that

vi) $\Gamma \vdash \langle r_1, L_1 = e\rangle : \rho$, and

vii) $\Gamma \vdash \langle r_1, L_1 = e\rangle = r : \rho$.

An analogous reasoning proves, but now using Proposition 4.12 below, that $\Gamma \vdash r_1.L : \beta r$ and that $\Gamma \vdash r.L = v : \beta r$.

Finally, the third case follows by induction and applying the rules of selection.

□

Proposition 4.11. *If* $\Gamma \vdash \langle r, L = e\rangle : \rho$ *and* $L : \beta$ *in* ρ *then*

$th_1 \diamond \Gamma \vdash e : \beta\langle r, L = e\rangle$

$th_2 \diamond \Gamma \vdash \langle r, L = e\rangle.L = e : \beta\langle r, L = e\rangle$

Proof. The proof proceeds by induction on the derivation of $\Gamma \vdash \langle r, L = e\rangle : \rho$. There are then two cases to be considered.

1) $\Gamma \vdash \langle r, L = e\rangle : \langle \rho_1, L{:}\beta\rangle$

We are allowed to assume that

i) $\Gamma \vdash r : \rho_1$

ii) $\Gamma \vdash \beta : \rho_1 \rightarrow type$, and

iii) $\Gamma \vdash e : \beta r$.

We can use i) and iii) to get by rule of record object equality that

iv) $\Gamma \vdash \langle r, L = e\rangle = r : \rho_1$.

Then $\Gamma \vdash e : \beta\langle r, L = e\rangle$. Now, observe that the label L must necessarily be fresh in ρ_1. Thus, we show the following derivation that gives the proof of th_2:

$$\dfrac{\dfrac{\Gamma \vdash r : \rho_1 \quad \Gamma \vdash e : \beta r}{\Gamma \vdash \langle r, L = e\rangle.L = e : \beta r}\ {\scriptstyle L \text{ fresh in } \rho_1} \quad \dfrac{\dfrac{\text{ii)} \qquad \text{iv)}}{\Gamma \vdash \beta r = \beta\langle r, L = e\rangle : type}}{\Gamma \vdash \beta r \sqsubseteq \beta\langle r, L = e\rangle}}{\Gamma \vdash \langle r, L = e\rangle.L = e : \beta\langle r, L = e\rangle}$$

2) $\Gamma \vdash \langle r, L = e\rangle : \rho_1$ and $\Gamma \vdash \rho_1 \sqsubseteq \rho$

Then, there exists a field $L{:}\beta_1$ in ρ_1 and $\Gamma \vdash \beta_1 \sqsubseteq \beta : \rho_1 \rightarrow type$. The induction hypothesis gives that

i) $\Gamma \vdash e : \beta_1\langle r, L = e\rangle$

ii) $\Gamma \vdash \langle r, L = e\rangle.L = e : \beta_1\langle r, L = e\rangle$.

Using the rules of subsumption we get the following derivations of th_1 and th_2:

$$\dfrac{\text{i)} \qquad \Gamma \vdash \beta_1\langle r, L = e\rangle \sqsubseteq \beta\langle r, L = e\rangle}{\Gamma \vdash e : \beta\langle r, L = e\rangle}$$

$$\dfrac{\text{ii)} \qquad \Gamma \vdash \beta_1\langle r, L = e\rangle \sqsubseteq \beta\langle r, L = e\rangle}{\Gamma \vdash \langle r, L = e\rangle.L = e : \beta\langle r, L = e\rangle}$$

□

PROPOSITION 4.12. *If* $\Gamma \vdash \langle r, L_1 = e\rangle : \rho$, $L : \beta$ *in* ρ *and* $L_1 \neq L$ *then*

$th_1 \diamond \Gamma \vdash r.L : \beta\langle r, L_1 = e\rangle$

$th_2 \diamond \Gamma \vdash \langle r, L_1 = e\rangle.L = r.L : \beta\langle r, L_1 = e\rangle$

PROOF. The proof proceeds by induction on the derivation of $\Gamma \vdash \langle r, L_1 = e\rangle : \rho$. There are, then, two cases to be considered:

1) $\Gamma \vdash \langle r, L_1 = e\rangle : \langle \rho_1, L_1{:}\beta\rangle$

We are allowed to assume that

i) $\Gamma \vdash r : \rho_1$

ii) $\Gamma \vdash e : \beta_1 r$.

We can use i) and ii) to get, by record extension, that

iii) $\Gamma \vdash \langle r, L_1 = e\rangle = r : \rho_1$.

Now, observe that if $L : \beta$ in ρ and $L \neq L_1$, then we have that $L : \beta$ in ρ_1. Therefore, by the rule of fields we get that $\Gamma \vdash \beta : \rho_1 \rightarrow type$.

Thus, we can get the proofs of th_1 and th_2 by application of the rules of selection and subsumption as follows:

i)

$$\dfrac{\dfrac{}{\Gamma \vdash r.L : \beta r}\ {\scriptstyle L : \beta \text{ in } \rho_1} \qquad \Gamma \vdash \beta r \sqsubseteq \beta\langle r, L_1 = e\rangle}{\Gamma \vdash r.L : \beta\langle r, L_1 = e\rangle}$$

iii)

$$\dfrac{}{\Gamma \vdash \langle r, L_1 = e\rangle.L = r.L : \beta_1\langle r, L_1 = e\rangle}\ {\scriptstyle L : \beta \text{ in } \rho_1}$$

2) $\Gamma \vdash \langle r, L_1 = e\rangle : \rho_1$ and $\Gamma \vdash \rho_1 \sqsubseteq \rho$

The proof is complete analogy to the one given for 2) in Proposition 4.11.

$\square$

CHAPTER 5

The proof checker

1. Introduction

In this chapter we shall present an implementation of the algorithms for checking the formal correctness of judgements of the calculus presented in the previous chapter. Here we shall concentrate on the implementation of that basic framework, we do not consider neither inductively defined sets nor the *let* and *use* expressions described in chapter 2.

In order to provide a complete description of the implemented system we explain again the input expressions and forms of declaration that it accepts. Then the algorithms are presented and motivated. We enunciate and informally prove the soundness of the algorithms under very strong assumptions, namely, that the reduction of a (well-typed, in a general sense) expression is normalising. The decidability of the algorithm, therefore, also depends on this assumption. We have not proved this property of normalization, but we are convinced that it could be done adapting the proof by Coquand for the system presented in [**Coq91**] or the one presented by Goguen in [**Gog94**].

2. The system

A script for the type checker looks very much like one for a functional programming language. The syntax of input expressions is given by the grammar in Figure 5.1.

$$\begin{array}{rcl} e & ::= & x \mid\ c \mid\ [x]e \mid\ e_1 e_2 \mid\ \langle\rangle \mid\ \langle e_1, L = e_2\rangle \mid\ e.L \\ & & e_1 \rightarrow e_2 \mid\ \langle e_1, L{:}e_2\rangle \end{array}$$

FIGURE 5.1. Syntax of input expressions

Observe then that parameters are not valid objects of the category of input expressions.

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

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

with T, F and c constant names, x,$x_1, \ldots, x_n$ variables and e, 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 $[x_1][x_2]\ldots[x_n]e$ of type $\alpha_1 \rightarrow [x_1](\alpha_2 \rightarrow \ldots (\alpha_n \rightarrow [x_n]\alpha)\ldots)$, with $n \geq 0$.

The two first are the counterpart in the system to the nominal definitions of types and families of types introduced in chapter 2. The latter form of declaration is not present in the proof-assistant ALF.

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

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.

2.1. Valid declarations. In the following we will make explicit that declarations are checked in a given environment. We use for this a form of judgement $\mathcal{E} \vdash D$, where $\mathcal{E}$ is a checking environment and D is one of the forms of declaration introduced above.

Definition 5.1 (Checking environment).

A checking environment ($\mathcal{E}$) is defined as a pair formed by a typed environment (Σ) and a context (Γ). A typed environment Σ is a dictionary of pairs of expressions indexed by names of constants. A context Γ is a dictionary of expressions indexed by parameters.

The environment part of a checking environment shall be denoted by $\mathcal{E}_{\Sigma}$ and the context part by $\mathcal{E}_{\Gamma}$.

We introduce now some operations for a checking environment $\mathcal{E}$.

Definition 5.2.

– the function *Dom* returns all definienda from $\mathcal{E}_{\Sigma}$.
– $\mathcal{E}, p{:}\alpha$ is defined to be the updating of $\mathcal{E}_{\Gamma}$ with index p and expression α.
– $\mathcal{E} + d : \tau = e$ is the updating of $\mathcal{E}_{\Sigma}$ with index d and the pair (e, τ).

The verification of the formal correctness of a declaration $\mathcal{E} \vdash D$ is defined by cases in D as follows:

A form of declaration $\mathcal{E} \vdash T : \texttt{type}{=}\alpha$ is valid if
- T does not occur in α nor belongs to *Dom* $\mathcal{E}$.[1]
- `checkType` $\mathcal{E}$ α succeeds.

We say that $\mathcal{E} \vdash F(x : \alpha) : \texttt{type} = \alpha_1$ is a valid declaration if

[1]Actually, as α is checked to be a type in $\mathcal{E}$ it suffices with controling the second condition to hold.

- F does not belong to $Dom\ \mathcal{E}$.
- $\mathsf{checkType}\ \ \mathcal{E}\ \ \alpha$ and $\mathsf{checkTypeFam}\ \ \mathcal{E}\ \ [x]\alpha_1\ \ \alpha$ succeed.

For a declaration $\mathcal{E} \vdash c(x_1 : \alpha_1, \ldots , x_n : \alpha_n)$: $\alpha = e$ to be valid it must hold that

- The constant c does not belong to $Dom\ \mathcal{E}$.
- $\mathsf{checkType}\ \ \mathcal{E}\ \ \alpha_1 \!\rightarrow\! [x_1](\alpha_2 \!\rightarrow\! \ldots (\alpha_n \!\rightarrow\! [x_n]\alpha) \ldots)$ succeeds.
- $\mathsf{checkExp}\ \ \mathcal{E}\ \ [x_1][x_2] \ldots [x_n]e\ \ \alpha_1 \!\rightarrow\! [x_1](\alpha_2 \!\rightarrow\! \ldots (\alpha_n \!\rightarrow\! [x_n]\alpha) \ldots)$ succeeds.

The procedures $\mathsf{checkType}$, $\mathsf{checkTypeFam}$ and $\mathsf{checkExp}$ above perform the checking, in the environment $\mathcal{E}$, that α is a type, $[x]\alpha_1$ is a type family over α and the expression a is an object of type α respectively. They are defined in section 3. After a declaration D is checked, the updating of the checking environment $\mathcal{E}$, if D is valid, is respectively defined to be

- $\mathcal{E} + T : type = \alpha$,
- $\mathcal{E} + F : \alpha \!\rightarrow\! [x]type = [x]\alpha_1$, and
- $\mathcal{E} + c : \alpha_1 \!\rightarrow\! [x_1](\alpha_2 \!\rightarrow\! \ldots (\alpha_n \!\rightarrow\! [x_n]\alpha) \ldots) = [x_1][x_2] \ldots [x_n]e$

These we call *valid updatings* of $\mathcal{E}_{\sum}$.

3. The type checking algorithm

We now intend to give a precise formulation of the informal explanations in chapter 3 for checking the correctness of the judgement $a : \alpha$. Recall that in those explanations we were assuming that α was already known to be a type. Thus, we shall also formulate the algorithm for checking judgements of the form $\alpha : type$ and thereby also for judgements of the form $\beta : \alpha \!\rightarrow\! type$.

In contrast to the input expressions accepted by the proof checker, the arguments to the programs we shall define may contain parameters. As anticipated, for checking that an abstraction $[x]\alpha_1$ is a type family over α, for instance, a fresh parameter, p say, is introduced in the context part of the enviroment in which the checking is taking place and then we shall proceed by checking that $\alpha_1[x := p]$ is a type. The language of expressions is then the one defined in chapter 4 with the following extension: we shall use s to range over the set $\mathcal{S}$ whose elements are, for the time being, the distinguished constants *Set* and *type*, from now on called *sorts*. Both are considered well-formed expressions, but only the first may occur in a valid input expression.

Now we proceed to introduce a function for computing the weak-head normal form of a well-formed expression. It could be grasped as the function implementing the relation $\Rightarrow$ presented in chapter 4. In addition, it shall, when needed, also unfold constants which have been introduced in the environment $\mathcal{E}$. We make extensive use of this function in the algorithms we present below.

3.1. Weak-head normalization. The definition of the function $\Downarrow$ is given in Figure 5.2. Due to the presence of constants in expressions, it also takes as argument the typed environment Σ of a checking environment $\mathcal{E}$. We use $e \Downarrow \Sigma$ to denote its application to expression e and environment Σ.

$$\begin{array}{lll}
p \Downarrow \Sigma & =_{def} & p \\
s \Downarrow \Sigma & =_{def} & s \\
c \Downarrow \Sigma & =_{def} & red_\delta\ c\ \Sigma \\
[x]e \Downarrow \Sigma & =_{def} & [x]e \\
\alpha\to\beta \Downarrow \Sigma & =_{def} & \alpha\to\beta \\
fe \Downarrow \Sigma & =_{def} & red_\beta\ f\ e\ \Sigma \\
& \text{where} &
\end{array}$$

$$\begin{array}{llll}
red_\delta\ c\ \Sigma & =_{def} & e \Downarrow \Sigma & \text{if } c : e = \alpha \text{ in } \Sigma \\
red_\beta\ f\ e\ \Sigma & =_{def} & \text{let } f' = f \Downarrow \Sigma & \\
& & \text{in } \ \text{if } f' = [x]f'' & \\
& & \qquad \text{then } f''[x := e] \Downarrow \Sigma & \\
& & \qquad \text{else } f'e &
\end{array}$$

FIGURE 5.2. Weak-head normalization

DEFINITION 5.3.

- An expression is in *weak head normal form* if it is either of the form $[x]e$, $\alpha\to\beta$ or $(ha_1 \ldots a_n)$, with $n \geq 0$ and h a parameter or a sort.
- A *top-level redex*[2] is an expression of the form $(fa_1 \ldots a_n)$ where f is either an abstraction $[x]e$ and $n \geq 1$ or a constant c (of arity n) and $n \geq 0$.

The intuition is that if the value of $e \Downarrow \Sigma$ is the expression e', then e' is the result of performing contractions of top-level redexes (if any) starting from e until a weak-head normal form is reached. Observe that due to the fact that we are going to apply the function $\Downarrow$ to well-typed (in a wide sense) expressions we refine the characterization of weak head normal to the effect that h can only be a parameter or a sort.

Notice that the order of evaluation is normal and no reduction is performed under binders.

3.2. Type checking in the original theory. We will now present algorithms for checking that an expression α is a type ($\mathsf{checkType}\ \mathcal{E}\ \alpha$) and that an expression a is an object of type α ($\mathsf{checkExp}\ \mathcal{E}\ a\ \alpha$). As explained in chapter 3, the construction of these algorithms is intertwined with that of the algorithms for inferring the type of an expression($\mathsf{inferExp}\ \mathcal{E}\ f \gg \alpha$), checking conversion of types ($\mathsf{typeConv}\ \mathcal{E}\ \alpha\ \alpha_1$) and conversion of objects ($\mathsf{objConv}\ \mathcal{E}\ a\ b\ \alpha$).

Each program below is presented by a set of rules of the form $\frac{P_1\ \ldots\ P_n}{Q}$, where the premisses and the conclusion are either of the form P or $P \gg v$. The form P should be read as "the program P succeeds" and the form $P \gg v$ as "P succeeds with value v". The general explanation of a rule is as follows: to compute the program Q, compute the premisses $P_1 \ldots P_n$ from left to right. For the computation of conclusion Q to succeed the computation of all the premisses must succeed. This

[2]We borrowed this terminology from [**Pey87**].

approach for presenting the semantics of a program follows the one taken by B. Nordström in [**Nor**].

Some of the rules will also have a side condition, which can either be p fresh in $\mathcal{E}$, $p : \alpha$ in $\mathcal{E}$, $c : e = \alpha$ in $\mathcal{E}$ or $s \in \mathcal{S}$. The three first should be read as "there is no entry for the parameter p in the context $\mathcal{E}_\Gamma$", "the lookup of p in $\mathcal{E}_\Gamma$ yields α" and "the lookup of c in $\mathcal{E}_\sum$ yields (e, α)".

The success of the conclusion naturally also depends on the success of the side condition.

To begin with, we shall now introduce the mutually defined programs $\mathsf{checkType}$ and $\mathsf{checkTypeFam}$. We recall that they check whether the expression α is a type and whether the expression β is a family of types over α, respectively.

Program ($\mathsf{checkType}\ \mathcal{E}\ \alpha$). This program is recursively defined by cases on the form of the expression α.

$$\frac{}{\mathsf{checkType}\ \ \mathcal{E}\ \ Set}$$

$$\frac{\mathsf{checkType}\ \ \mathcal{E}\ \ \alpha \quad \mathsf{checkTypeFam}\ \ \mathcal{E}\ \ \beta\ \ \alpha}{\mathsf{checkType}\ \ \mathcal{E}\ \ \alpha{\to}\beta}$$

$$\frac{\mathsf{inferExp}\ \ \mathcal{E}\ \ f\ \gg s}{\mathsf{checkType}\ \ \mathcal{E}\ \ f}\ {\scriptstyle s\,\in\,\mathcal{S}}$$

Observe that in the last rule, f can only be a generalized application (the sort s is inferred) for the program to succeed. The checking of an object of type *Set* using (sort) inference is a modification of Magnusson's algorithm that works for sets defined à la ALF as well. What really makes a difference is that the same procedure also allows to check the correctness of type expressions formed out of constants introduced by type and type family declarations.

Program ($\mathsf{checkTypeFam}\ \mathcal{E}\ \beta\ \alpha$). This program is defined by cases on the expression β. There exists only two possible forms of expression for a type family: it is either an abstraction or a constant introduced by a type family declaration. It is assumed that $\mathcal{E}$ is a valid environment and that α has already been checked to be a type in this environment.

$$\frac{\mathsf{checkType}\ \ \mathcal{E}, p{:}\alpha\ \ \alpha_1[x := p]}{\mathsf{checkTypeFam}\ \ \mathcal{E}\ \ [x]\alpha_1\ \ \alpha}\ {\scriptstyle p \text{ fresh in } \mathcal{E}}$$

We are assuming that the set P of parameters is infinite; as expressions and contexts are finite we can always choose a fresh one. Moreover, since parameters are not allowed in the input expressions, the expression α_1 is independent of the parameter p.

$$\frac{\mathsf{typeConv}\ \ \mathcal{E}\ \ \alpha_1\ \ \alpha}{\mathsf{checkTypeFam}\ \ \mathcal{E}\ \ F\ \ \alpha}\ {\scriptstyle F\,:\,e\,=\,\alpha_1{\to}[x]s \text{ in } \mathcal{E},\ s\,\in\,\mathcal{S}}$$

This rule expresses that once we know that an expression is a type family over a certain type α_1 it is also the case that it is a family over any type α if α_1 and α are convertible types.

PROGRAM (checkExp $\mathcal{E}$ e α). This program is recursively defined by cases on the expression e. It is assumed that $\mathcal{E}$ is valid and checkType $\mathcal{E}$ α succeeded.

$$\frac{\alpha \Downarrow \Sigma \ \gg \alpha_1 \to \beta \quad \mathsf{checkExp}\ \mathcal{E}, p{:}\alpha_1\ \ e[x := p]\ \ \beta p}{\mathsf{checkExp}\ \mathcal{E}\ \ [x]e\ \ \alpha}\ {\scriptstyle p \text{ fresh in } \mathcal{E}}$$

$$\frac{\mathsf{inferExp}\ \mathcal{E}\ \ f\ \gg \alpha_1 \quad \mathsf{typeConv}\ \mathcal{E}\ \ \alpha_1\ \ \alpha}{\mathsf{checkExp}\ \mathcal{E}\ \ f\ \ \alpha}\ {\scriptstyle \alpha_1 \neq type,\ \alpha_1 \neq \alpha_2 \to [x]type}$$

As explained in the informal presentation of the algorithm, for checking that a generalized application has type α, we infer its type, α_1 say, and then check whether α_1 and α are convertible types. The side condition prevents unnecessary conversion checkings. This control will become more clear after we present the definition of the function inferExp.

PROGRAM (inferExp $\mathcal{E}$ f $\gg \alpha$). We recall the reading of this form of program: (the computation of the function) inferExp when applied to inputs $\mathcal{E}$ and f (if succeeds) yields the expression α. This function is defined by cases on the expression f.

$$\frac{}{\mathsf{inferExp}\ \mathcal{E}\ \ p\ \gg \alpha}\ {\scriptstyle p\,:\,\alpha \text{ in } \mathcal{E}}$$

For inferring the type of a parameter a lookup operation on the dictionary $\mathcal{E}_\Gamma$ is performed.

$$\frac{}{\mathsf{inferExp}\ \mathcal{E}\ \ c\ \gg \alpha}\ {\scriptstyle c\,:\,e = \alpha \text{ in } \mathcal{E}}$$

For constants the lookup is performed on $\mathcal{E}_\Sigma$. Notice that the sort *type* and expressions of the form $\alpha \to [x]type$ are possible results.

$$\frac{\mathsf{inferExp}\ \mathcal{E}\ \ f\ \gg \alpha \quad \alpha \Downarrow \Sigma \ \gg \alpha_1 \to \beta \quad \mathsf{checkExp}\ \mathcal{E}\ \ e\ \ \alpha_1}{\mathsf{inferExp}\ \mathcal{E}\ \ fe\ \gg \beta e}$$

In this rule there is a subtle point, already present in some rules above, that we consider worth remarking. The type inferred for fe is βe. As β is a family of types, its ultimate definiens will be of the form $[x]\alpha_2$. Then, when the weak-head normal form of the expression βe is computed , the substitution $\alpha_2[x := e]$ will be performed. Now, recall that the substitution we are using does not prevent capture of variables. One may wonder then whether it is safe to use this operation in the programs we are presenting . Actually, it is possible to prove that for expressions α and e if checkType $\mathcal{E}$ α then α is a well-formed expression, and further, if checkExp $\mathcal{E}$ e α, the expressions e is also well-formed. Thus, no variable can be captured.

PROGRAM (typeConv $\mathcal{E}$ α_1 α_2). This program is simultaneously defined with the program whTypeConv.

A remark is in place before we provide the rules defining the program typeConv. Observe first that the computation of typeConv $\mathcal{E}$ α_1 α_2 is triggered, for instance, by the rule that checks whether a generalized application f has a type α_2. At that point it is already known that α_2 is a type, since this is a precondition for the program checkExp. However, since the expression α_1 is obtained as output of the function inferExp, it could also be the sort *type* or an expression of the form $\alpha \to [x]type$. However, the possibility has been ruled out for these forms of expressions to be arguments of the program typeConv.

Let us now turn back to the definition of the program.

First, it is checked whether the expressions are syntactically equal:

$$\frac{}{\mathsf{typeConv}\ \mathcal{E}\ \ \alpha\ \ \alpha}$$

If this is not the case, both α_1 and α_2 are reduced to their corresponding weak-head normal forms, which are in turn the input to the program whTypeConv. This latter is recursively defined by case analysis on the form of its arguments.

$$\frac{\alpha_1 \Downarrow \Sigma\ \gg \alpha_1{}' \quad \alpha_2 \Downarrow \Sigma\ \gg \alpha_2{}' \quad \mathsf{whTypeConv}\ \mathcal{E}\ \ \alpha_1{}'\ \ \alpha_2{}'}{\mathsf{typeConv}\ \mathcal{E}\ \ \alpha_1\ \ \alpha_2}$$

$$\frac{}{\mathsf{whTypeConv}\ \mathcal{E}\ \ Set\ \ Set}$$

For checking the convertibility of two function types it must be checked that the types and type families out of which they are formed are respectively convertible:

$$\frac{\mathsf{typeConv}\ \mathcal{E}\ \ \alpha_1\ \ \alpha_2 \quad \mathsf{typeConv}\ \mathcal{E}, p{:}\alpha_1\ \ \beta_1 p\ \ \beta_2 p}{\mathsf{whTypeConv}\ \mathcal{E}\ \ \alpha_1 \to \beta_1\ \ \alpha_2 \to \beta_2}\ p \text{ fresh in } \mathcal{E}$$

Notice that once it is checked that α_1 and α_2 are convertible types it is correct to apply the family β_2 to the parameter p which is declared as a generic object of type α_1.

The following (and last) rule expresses that two ground types different from the type *Set* are convertible if they are as objects of type *Set*.

$$\frac{\mathsf{objConv}\ \mathcal{E}\ \ \alpha_1\ \ \alpha_2\ \ Set}{\mathsf{whTypeConv}\ \mathcal{E}\ \ \alpha_1\ \ \alpha_2}$$

Program (objConv $\mathcal{E}$ a b α). According to the informal formulation of this algorithm, the checking that two objects of a certain type are convertible is recursively defined by cases on the form of the type. First, then, the weak-head normal form α_1 of the type α is computed. Therefore, α_1 must either be a function type of the form $\alpha \to \beta$ or a ground type. This expression is, in turn, together with the objects a and b, the input for the program whObjConv, which is simultaneously defined with objConv.

$$\frac{\alpha \Downarrow \Sigma\ \gg \alpha_1 \quad \mathsf{whObjConv}\ \mathcal{E}\ \ a\ \ b\ \ \alpha_1}{\mathsf{objConv}\ \mathcal{E}\ \ a\ \ b\ \ \alpha}$$

For checking that two objects of type $\alpha \to \beta$ are convertible, check whether when applied to a fresh parameter p of type α they are convertible objects of type βp. Notice that this checking comprises both α- and η-conversion.

$$\frac{\mathsf{objConv}\ \mathcal{E}, p{:}\alpha\ \ fp\ \ gp\ \ \beta p}{\mathsf{whObjConv}\ \mathcal{E}\ \ f\ \ g\ \ \alpha \to \beta}\ p \text{ fresh in } \mathcal{E}$$

Objects of a ground type are checked to be convertible as follows

$$\frac{a \Downarrow \Sigma \ \gg a_1 \quad b \Downarrow \Sigma \ \gg b_2 \quad \mathsf{headConv}\ \mathcal{E}\ \ a_1\ \ b_2\ \gg \alpha_1 \quad \mathsf{typeConv}\ \mathcal{E}\ \ \alpha_1\ \ \alpha}{\mathsf{whObjConv}\ \mathcal{E}\ \ a\ \ b\ \ \alpha}$$

This rule merits some more discussion. It should be read as: two objects of ground type α are convertible if their corresponding weak-head normal forms are head-convertible objects of a type α_1, which in turn has to be convertible to the type α.

Now, observe that a_1 and b_1 are objects of a ground type, therefore they must necessarily be of the form of a generalized application. Furthermore, as both are in weak-head normal form, either they are parameters or applications whose heads are parameters. Thus, for checking the convertibility of a_1 and b_1 two cases must be considered: either they are both the same parameter or in the case they are objects of the form fa_2 and gb_2, respectively, f and g are convertible objects of a function type $\alpha_2 \rightarrow \beta$ and a_2 and b_2 are convertible objects of type α_2. We can perform this latter checking using the object conversion program only if we infer the type of one of f and g (what we can do because they are generalized applications). We prefer instead to follow Magnusson's presentation for checking typed conversion of objects and define a function headConv which implements the procedure described above.

Before we proceed with the formulation of headConv, we introduce a further rule for the program objConv. Notice that the definition given so far does not consider, in principle, the form of the expressions a and b but of their common type α. This could entail that in the case that a and b are syntactically equal a considerable amount of computations might be unnecessarily performed. Think, for instance, of the case when both a and b are the same constant f of type $\alpha \rightarrow \beta$. In order to improve the efficiency of the whole procedure of object conversion checking we then also formulate the rule

$$\frac{}{\mathsf{objConv}\ \mathcal{E}\ \ a\ \ a\ \ \alpha}$$

which acts as the formal counterpart of the reflexivity rule of the equality of objects of a certain type. Actually, this case should be the first considered in the definition of the program objConv.

Program (headConv $\mathcal{E}$ a b $\gg \alpha$).

$$\frac{}{\mathsf{headConv}\ \mathcal{E}\ \ p\ \ p\ \gg \alpha}\ {\scriptstyle p\,:\,\alpha\ \text{in}\ \mathcal{E}}$$

$$\frac{\mathsf{headConv}\ \mathcal{E}\ \ f\ \ g\ \gg \alpha \quad \alpha \Downarrow \Sigma\ \gg \alpha_1 \rightarrow \beta \quad \mathsf{objConv}\ \mathcal{E}\ \ a\ \ b\ \ \alpha_1}{\mathsf{headConv}\ \mathcal{E}\ \ fa\ \ gb\ \gg \beta a}$$

Remark . The whole procedure of conversion checking is efficient in the sense that in the case that objects are not convertible there is no need, in general, for their complete normalization. On the other hand, it will accept as convertible those objects whose (full) normal forms are identical, up to α- and η-convertibility.

The definition of this latter program ends the formulation of the type checking algorithm for the original theory.

3.3. Type checking in the extended theory. We shall now present the formulation of the type checking algorithm for the extended theory. In correspondence to the explanations in section 2 of chapter 3 we will focus on the changes to be made for considering those extensions in the definition of the programs.

3.3.1. *Valid Declarations.* The only modification to be introduced concerns the updating of a checking environment $\mathcal{E}$ after a declaration D of one of the forms $\mathcal{E} \vdash T : \texttt{type}{=}\alpha$ and $\mathcal{E} \vdash F(x : \alpha) : \texttt{type} = \alpha_1$ has been checked to be valid. In the case that the ultimate definiens of α (in the first declaration) and α_1 is a type of the form $\langle\rho, L{:}\beta\rangle$ the (valid) extensions are defined to be $\mathcal{E} + T : \textit{record-type} = \alpha$ and $\mathcal{E} + F : \alpha{\to}[x]\textit{record-type} = [x]\alpha_1$ respectively.

The set $\mathcal{S}$ is now extended with the sort *record-type*.

3.3.2. *Weak-head normalization.* The notions of weak-head normal form and top-level redex are now defined as follows:

Definition 5.4.

- An expression is in *weak-head normal form* if it is either of the form $[x]e$, $\alpha{\to}\alpha'$, $\langle\rangle$, $\langle\rho, L{:}\beta\rangle$, $\langle e, L = e'\rangle$ or $(h.L_1 \dots L_l\; a_1 \dots a_m).K_1 \dots K_n$ with l, m and $n \geq 0$ and h a parameter or a sort.
- A *top-level redex* is an expression of the form $(f a_1 \dots a_n)$ where f is either an abstraction $[x]e$ and $n \geq 1$, a constant c (of arity n) and $n \geq 0$ or a selection $\langle e, L_1 = e'\rangle.L_2$.

The reformulation of the function $\Downarrow$ is given in Figure 5.3.

Two new forms of side condition are now involved in the rules: L fresh in ρ and $L : \beta$ in ρ. They should be read as "the label L does not occur in the fields of ρ" and "there exists a field declaration $L{:}\beta$ in the record type ρ" respectively.

For the sake of comprehensiveness we repeat (for each program) the rules presented in section 3.2, making, in most of the cases, no further comment.

Program (checkType $\mathcal{E}$ α).

$$\frac{}{\mathsf{checkType}\ \mathcal{E}\ \ \mathit{Set}}$$

$$\frac{\mathsf{checkType}\ \mathcal{E}\ \ \alpha \quad \mathsf{checkTypeFam}\ \mathcal{E}\ \ \beta\ \ \alpha}{\mathsf{checkType}\ \mathcal{E}\ \ \alpha{\to}\beta}$$

$$\frac{\mathsf{inferExp}\ \mathcal{E}\ \ f\ \gg s}{\mathsf{checkType}\ \mathcal{E}\ \ f}\ {\scriptstyle s\,\in\,\mathcal{S}}$$

Due to the new form of valid extensions introduced above and considering that the set $\mathcal{S}$ was extended with the sort *record-type*, if the ultimate definiens of a generalized application f is a record form then it is a valid type expression. As to record types we now introduce the following rules:

$$\frac{}{\mathsf{checkType}\ \mathcal{E}\ \ \langle\rangle}$$

```
$p \Downarrow \Sigma$                 $=_{def}$  $p$
$s \Downarrow \Sigma$                 $=_{def}$  $s$
$c \Downarrow \Sigma$                 $=_{def}$  $red_\delta\ c\ \Sigma$
$[x]e \Downarrow \Sigma$              $=_{def}$  $[x]e$
$\langle\rangle \Downarrow \Sigma$    $=_{def}$  $\langle\rangle$
$\langle e, L = e'\rangle \Downarrow \Sigma$  $=_{def}$  $\langle e, L = e'\rangle$
$\alpha{\to}\beta \Downarrow \Sigma$  $=_{def}$  $\alpha{\to}\beta$
$\langle \rho, L{:}\beta\rangle \Downarrow \Sigma$  $=_{def}$  $\langle \rho, L{:}\beta\rangle$
$fe \Downarrow \Sigma$                $=_{def}$  $red_\beta\ f\ e\ \Sigma$
$r.L_1 \Downarrow \Sigma$             $=_{def}$  $red_\sigma\ r\ L_1\ \Sigma$
                                     where
        $red_\delta\ c\ \Sigma$        $=_{def}$  $e \Downarrow \Sigma$        if $c : e = \alpha$ in $\Sigma$
        $red_\beta\ f\ e\ \Sigma$      $=_{def}$  let $f' = f \Downarrow \Sigma$
                                          in  if $f' = [x]f''$
                                              then $f''[x := e] \Downarrow \Sigma$
                                              else $f'e$
        $red_\sigma\ r\ L_1\ \Sigma$   $=_{def}$  let $r' = r \Downarrow \Sigma$
                                          in  if $r' = \langle r'', L_2 = e\rangle$
                                              then if $L_1 = L_2$
                                                   then $e \Downarrow \Sigma$
                                                   else $red_\sigma\ r''\ L_1\ \Sigma$
                                              else $r'.L_1$
```

FIGURE 5.3. Weak-head normalization revisited

$$\frac{\mathsf{checkRecType}\ \ \mathcal{E}\ \ \langle \rho, L{:}\beta\rangle}{\mathsf{checkType}\ \ \mathcal{E}\ \ \langle \rho, L{:}\beta\rangle}$$

As previously explained in section 4 of chapter 2, it is in the nature of a record type for its formation to be explained both as a type and as a record formation. For $\langle \rho, L{:}\beta\rangle$ to be a record type it has to be checked that ρ is also a record type, not just a type. Now, in the presence of type and type family declarations ρ may also either be a defined constant R or the result of applying a record family to an object of the index type of this family. The preceding considerations then give rise to the following formulation of the procedure checkRecType:

PROGRAM (checkRecType $\mathcal{E}$ ρ). This program is recursively defined on the form of ρ and it is assumed that $\mathcal{E}$ is valid.

$$\frac{}{\mathsf{checkRecType}\ \ \mathcal{E}\ \ \langle\rangle}$$

$$\frac{\mathsf{checkRecType}\ \ \mathcal{E}\ \ \rho \qquad \mathsf{checkTypeFam}\ \ \mathcal{E}\ \ \beta\ \ \rho}{\mathsf{checkRecType}\ \ \mathcal{E}\ \ \langle \rho, L{:}\beta\rangle}\ {\scriptstyle L\ \text{fresh in}\ \rho}$$

$$\frac{\mathsf{inferExp}\ \ \mathcal{E}\ \ \rho\ \gg \mathit{record\text{-}type}}{\mathsf{checkRecType}\ \ \mathcal{E}\ \ \rho}$$

PROGRAM (checkTypeFam $\mathcal{E}$ β α).

$$\frac{\mathsf{checkType}\ \ \mathcal{E}, p{:}\alpha\ \ \alpha_1[x := p]}{\mathsf{checkTypeFam}\ \ \mathcal{E}\ \ [x]\alpha_1\ \ \alpha}\ {\scriptstyle p \text{ fresh in } \mathcal{E}}$$

$$\frac{\mathsf{typeIncl}\ \ \mathcal{E}\ \ \alpha\ \ \alpha_1}{\mathsf{checkTypeFam}\ \ \mathcal{E}\ \ F\ \ \alpha}\ {\scriptstyle F\,:\,e\,=\,\alpha_1 \to [x]s \text{ in } \mathcal{E},\ s \in \mathcal{S}}$$

Notice that now, because of the extension of the set $\mathcal{S}$, this rule subsumes the case of record families. Besides, it is checked whether α is a subtype of α_1 instead of checking for their convertibility.

PROGRAM (checkExp $\mathcal{E}$ a α).

$$\frac{\alpha \Downarrow \Sigma\ \gg \alpha_1 \to \beta\quad \mathsf{checkExp}\ \ \mathcal{E}, p{:}\alpha_1\ \ e[x := p]\ \ \beta p}{\mathsf{checkExp}\ \ \mathcal{E}\ \ [x]e\ \ \alpha}\ {\scriptstyle p \text{ fresh in } \mathcal{E}}$$

$$\frac{\rho \Downarrow \Sigma\ \gg \langle\rangle}{\mathsf{checkExp}\ \ \mathcal{E}\ \ \langle\rangle\ \ \rho}$$

We make an overloaded use of $\langle\rangle$ to denote both the empty record object and record type.

$$\frac{\rho \Downarrow \Sigma\ \gg \langle \rho_1, L{:}\beta\rangle\quad \mathsf{checkExp}\ \ \mathcal{E}\ \ r\ \ \rho_1\quad \mathsf{checkExp}\ \ \mathcal{E}\ \ e\ \ \beta r}{\mathsf{checkExp}\ \ \mathcal{E}\ \ \langle r, L = e\rangle\ \ \rho}$$

According to the explanations in section 2, for checking that a generalized selection has type α, we first infer its type, α_1 say, and then check whether it is a subtype of α.

$$\frac{\mathsf{inferExp}\ \ \mathcal{E}\ \ f\ \gg \alpha_1\quad \mathsf{typeIncl}\ \ \mathcal{E}\ \ \alpha_1\ \ \alpha}{\mathsf{checkExp}\ \ \mathcal{E}\ \ f\ \ \alpha}\ {\scriptstyle \alpha_1 \neq t,\ \alpha_1 \neq \alpha_2 \to [x]t,\ t \in \{\mathit{type}, \mathit{record\text{-}type}\}}$$

We explain now why this definition of checkExp corresponds to the restrictive method formulated in section 2 of chapter 3. Observe first, that the rule above for checking record object extensions rejects undeclared labels. It requires that the labels of the plain fields of an object are declared in the intended type. As this latter, in turn, has previously been checked to be a record type, there is no risk for multiple declarations of the same label in it. On the other hand, notice that checking whether an extension of the form $\langle f, L_1 = e_1, \ldots, L_n = e_n\rangle$ has a certain type ρ is implemented by n applications of that same rule and then the rule for checking generalized selections (the last one) is applied. The whole procedure for checking record extensions can, naturally, be implemented in a much more efficient way. For the sake of clarity, however, we prefer this presentation which, in addition, will allow to simplify the proofs when reasoning about the correctness of the algorithm.

Program (inferExp $\mathcal{E}$ $a \gg \alpha$).

$$\frac{}{\mathsf{inferExp}\ \ \mathcal{E}\ \ p\ \gg \alpha}\ {\scriptstyle p\,:\,\alpha \text{ in } \mathcal{E}}$$

$$\frac{}{\mathsf{inferExp}\ \ \mathcal{E}\ \ c\ \gg \alpha}\ {\scriptstyle c\,:\,e\,=\,\alpha \text{ in } \mathcal{E}}$$

$$\frac{\mathsf{inferExp}\ \ \mathcal{E}\ \ f\ \gg \alpha \quad \alpha \Downarrow \Sigma\ \gg \alpha_1 \to \beta \quad \mathsf{checkExp}\ \ \mathcal{E}\ \ e\ \ \alpha_1}{\mathsf{inferExp}\ \ \mathcal{E}\ \ fe\ \gg \beta e}$$

This function is extended with the following rule in order to consider selections

$$\frac{\mathsf{inferExp}\ \ \mathcal{E}\ \ r\ \gg \rho}{\mathsf{inferExp}\ \ \mathcal{E}\ \ r.L\ \gg \beta r}\ {\scriptstyle L\,:\,\beta \text{ in } \rho}$$

Program (typeIncl $\mathcal{E}$ α α'). The program typeConv is now replaced by the program typeIncl, which is simultaneously defined with whTypeIncl.

$$\frac{}{\mathsf{typeIncl}\ \ \mathcal{E}\ \ \alpha\ \ \alpha}$$

$$\frac{\alpha_1 \Downarrow \Sigma\ \gg \alpha_1{}' \quad \alpha_2 \Downarrow \Sigma\ \gg \alpha_2{}' \quad \mathsf{whTypeIncl}\ \ \mathcal{E}\ \ \alpha_1{}'\ \ \alpha_2{}'}{\mathsf{typeIncl}\ \ \mathcal{E}\ \ \alpha_1\ \ \alpha_2}$$

The explanation of these rules is analogous to the one given for the two first rules in the definition of typeConv

$$\frac{}{\mathsf{whTypeIncl}\ \ \mathcal{E}\ \ Set\ \ Set}$$

The rule for checking the inclusion of two function types is also similar to the one for checking whether they are convertible. However, it must also take into account that the type former $\to$ is contravariant on the index type.

$$\frac{\mathsf{typeIncl}\ \ \mathcal{E}\ \ \alpha_2\ \ \alpha_1 \quad \mathsf{typeIncl}\ \ \mathcal{E}, p{:}\alpha_2\ \ \beta_1 p\ \ \beta_2 p}{\mathsf{whTypeIncl}\ \ \mathcal{E}\ \ \alpha_1 \to \beta_1\ \ \alpha_2 \to \beta_2}\ {\scriptstyle p \text{ fresh in } \mathcal{E}}$$

The following two rules directly implement the explanation for two record forms to be in the inclusion relation:

$$\frac{}{\mathsf{whTypeIncl}\ \ \mathcal{E}\ \ \rho\ \ \langle\rangle}$$

$$\frac{\mathsf{typeIncl}\ \ \mathcal{E}\ \ \rho_1\ \ \rho_2 \quad \mathsf{typeIncl}\ \ \mathcal{E}, p{:}\rho_1\ \ \beta_1 p\ \ \beta_2 p}{\mathsf{whTypeIncl}\ \ \mathcal{E}\ \ \rho_1\ \ \langle \rho_2, L{:}\beta_2 \rangle}\ {\scriptstyle L\,:\,\beta_1 \text{ in } \rho_1}$$

Finally, for two ground types different from the type *Set* it is checked whether they are convertible objects of this latter type

$$\frac{\mathsf{objConv}\ \ \mathcal{E}\ \ \alpha_1\ \ \alpha_2\ \ Set}{\mathsf{whTypeIncl}\ \ \mathcal{E}\ \ \alpha_1\ \ \alpha_2}$$

It is clear from the former rule that two ground types are accepted to be in the inclusion relation only if they are definitionally equal. We have not explored a more sophisticated treatment for this case. Yet, it seems quite reasonable to expect that a mechanism of subtyping for ground types could, in a modular way, be incorporated to typeIncl by just modifying the premiss of the last rule above.

The whole definition of the program typeIncl follows exactly the informal explanations for checking that two types are in the inclusion relation. It is not difficult to see that type conversion is subsumed by type inclusion. Moreover, it can be proved that checking for type inclusion instead of type conversion is conservative in the sense that if α_1 and α_2 belong to the language of expressions defined in section 3.2 and typeIncl $\mathcal{E}\ \ \alpha_1\ \ \alpha_2$ succeeds then so does typeConv $\mathcal{E}\ \ \alpha_1\ \ \alpha_2$.

PROGRAM (objConv $\mathcal{E}\ \ a\ \ b\ \ \alpha$).

$$\frac{}{\mathsf{objConv}\ \mathcal{E}\ \ a\ \ a\ \ \alpha}$$

$$\frac{\alpha \Downarrow \Sigma\ \gg \alpha_1 \quad \mathsf{whObjConv}\ \mathcal{E}\ \ a\ \ b\ \ \alpha_1}{\mathsf{objConv}\ \mathcal{E}\ \ a\ \ b\ \ \alpha}$$

$$\frac{\mathsf{objConv}\ \mathcal{E}, p{:}\alpha\ \ fp\ \ gp\ \ \beta p}{\mathsf{whObjConv}\ \mathcal{E}\ \ f\ \ g\ \ \alpha{\to}\beta}\ {\scriptstyle p \text{ fresh in } \mathcal{E}}$$

The rules below for checking that two objects of a record type are convertible are, also, a direct implementation of the informal procedure described in chapter 3 for checking the equality of two objects of a given record type: for checking that two objects of a record type are convertible, check whether the selections of every label of the record type in question from the objects are convertible.

$$\frac{}{\mathsf{whObjConv}\ \mathcal{E}\ \ r\ \ s\ \ \langle\rangle}$$

$$\frac{\mathsf{objConv}\ \mathcal{E}\ \ r\ \ s\ \ \rho \quad \mathsf{objConv}\ \mathcal{E}\ \ r.L\ \ s.L\ \ \beta r}{\mathsf{whObjConv}\ \mathcal{E}\ \ r\ \ s\ \ \langle\rho, L{:}\beta\rangle}$$

$$\frac{a \Downarrow \Sigma\ \gg a_1 \quad b \Downarrow \Sigma\ \gg b_2 \quad \mathsf{headConv}\ \mathcal{E}\ \ a_1\ \ b_2\ \gg \alpha_1 \quad \mathsf{typeIncl}\ \mathcal{E}\ \ \alpha_1\ \ \alpha}{\mathsf{whObjConv}\ \mathcal{E}\ \ a\ \ b\ \ \alpha}$$

For checking that two generalized selections a_1 and b_1 in weak-head normal form are convertible objects of a ground type , we must now consider that they can also be of the form $r.L_1$ and $s.L_2$ respectively. What must be checked then is whether $L_1 = L_2$ and that r and s are head-convertible objects of some type ρ. Observe also that α_1 is checked to be a subtype of α.

PROGRAM (headConv $\mathcal{E}\ \ a\ \ b\ \gg \alpha$).

$$\frac{}{\mathsf{headConv}\ \mathcal{E}\ \ p\ \ p\ \gg \alpha}\ {\scriptstyle p\,:\,\alpha \text{ in } \mathcal{E}}$$

$$\frac{\mathsf{headConv}\ \mathcal{E}\ \ f\ \ g\ \gg \alpha \quad \alpha \Downarrow \Sigma\ \gg \alpha_1{\to}\beta \quad \mathsf{objConv}\ \mathcal{E}\ \ a\ \ b\ \ \alpha_1}{\mathsf{headConv}\ \mathcal{E}\ \ fa\ \ gb\ \gg \beta a}$$

Finally, we extend the program headConv with the rule

$$\frac{\mathsf{headConv}\ \mathcal{E}\ \ r\ \ s\ \gg \rho}{\mathsf{headConv}\ \mathcal{E}\ \ r.L\ \ s.L\ \gg \beta r}\ {\scriptstyle L\,:\,\beta \text{ in } \rho}$$

4. Correctness of the algorithm

We now proceed to give an informal proof of the soundness of the algorithm with respect to the calculus presented in chapter 4. The programs and functions that the whole algorithm embodies are defined to work on a checking environment. However, the forms of judgement of the calculus are not defined as to explicitly consider that judgements can be made under a set of constant declarations, or more formally, in the presence of nominal definition of constants. This is the approach taken by Severi in [**Sev96**] where a formulation of PTS with definitions is presented. Magnusson, on the other hand, for the correctness proofs of the algorithms presented in [**Mag95**] just assume that such a set of declarations has a formal counterpart in *CES*, the calculus whose forms of judgement are mechanically verified by those algorithms.

We preferred to follow the tradition of presenting the calculus without explicitly introducing the notion of a set of nominal definitions. However, we do not want to leave unattended the role played by the typed environment when reasoning on the correctness of the procedures we have defined to check the formal correctness of judgements of the calculus in question. Thus, we shall prove, for instance that if $\mathsf{checkType}\ (\Sigma, \Gamma)\ \ \alpha$ succeeded then it holds that $\Gamma^*_\Sigma \vdash \alpha^*_\Sigma : \mathit{type}$. The function $[-]^*_\Sigma$ performs the unfolding of the constants declared in Σ which occur in its argument (in the case of a context Γ, recursively unfolds the (type) expressions associated to the parameters declared in it).

We now first turn to define this latter function. A few propositions, in addition, are also enunciated and some of them proved.

4.1. Unfolding and basic properties. The definition of the function , which we show in Figure 5.4, is much in the spirit of the *projection mapping* introduced in chapter 11 of Severi's thesis [**Sev96**].
It is possible to prove that the unfolding function on expressions terminates. For this it is crucial the fact that no recursive declarations of constants are allowed in the typed environment Σ. A measure yielding a natural number can be defined, $\mathcal{C}(\Sigma, e)$ say, which decreases when the function is used. This measure computes the number of constants replaced in the expression e when the unfolding of this expression is performed with environment Σ.

We now introduce the following

Definition 5.5.

- a typed environment is valid if it is either {} or the result of performing a valid updating on a valid environment Σ.
- a context is valid w.r.t. a typed environment Σ if it is either [] or the result of updating a valid context Γ w.r.t. Σ with index p and expressions α, p is a fresh parameter for Γ and $\Gamma^*_\Sigma \vdash \alpha^*_\Sigma : \mathit{type}$.
- A checking environment $\mathcal{E}$ is valid if $\mathcal{E}_\Sigma$ is valid and $\mathcal{E}_\Gamma$ is valid w.r.t. $\mathcal{E}_\Sigma$.

By being fresh we mean that there is no entry corresponding to the index p in Γ.

Remark . When the system starts, its checking environment $\mathcal{E}$ is initialized to be the pair ({},[]). By construction then $\mathcal{E}$ is valid. When the checking of a *unfolding of contexts:*

$$[]^*_\Sigma =_{def} [] \qquad\qquad (\Gamma, p{:}\alpha)^*_\Sigma =_{def} \Gamma^*_\Sigma, p{:}\alpha^*_\Sigma$$

unfolding of expressions:

$$\begin{array}{lll}
x^*_\Sigma & =_{def} & x \\
p^*_\Sigma & =_{def} & p \\
s^*_\Sigma & =_{def} & s \\
c^*_\Sigma & =_{def} & e^*_{\Sigma_1} \quad \textit{with } \Sigma = \Sigma_1; c : \alpha = e; \Sigma_2 \\
 & =_{def} & c \quad \textit{otherwise} \\
([x]e)^*_\Sigma & =_{def} & [x]e^*_\Sigma \\
\langle\rangle^*_\Sigma & =_{def} & \langle\rangle \\
\langle e_1, L = e_2\rangle^*_\Sigma & =_{def} & \langle {e_1}^*_\Sigma, L = {e_2}^*_\Sigma\rangle \\
(\alpha\to\beta)^*_\Sigma & =_{def} & \alpha^*_\Sigma\to\beta^*_\Sigma \\
\langle\rho, L{:}\beta\rangle^*_\Sigma & =_{def} & \langle\rho^*_\Sigma, L{:}\beta^*_\Sigma\rangle \\
(f\,e)^*_\Sigma & =_{def} & f^*_\Sigma e^*_\Sigma \\
(r.L)^*_\Sigma & =_{def} & r^*_\Sigma.L
\end{array}$$

FIGURE 5.4. Unfolding

declaration begins $\mathcal{E}_\Gamma$ is always []. We shall see that in the algorithms presented in the previous section, all the extensions we have made of the context preserve its validity, as above defined.

PROPOSITION 5.1. *Let $\mathcal{T}$ be $\mathcal{S} - \{Set\}$ and Σ be a valid typed environment. If d is a constant such that $\Sigma = \Sigma_1; d : \tau = e; \Sigma_2$ then either*

1) $\tau \in \mathcal{T}$ and checkType $(\Sigma_1, [])$ e *succeeded,*

2) $\tau = \alpha\to[x]t$, $t \in \mathcal{T}$, checkType $(\Sigma_1, [])$ α *and* checkTypeFam $(\Sigma_1, [])$ e α *succeeded, or*

3) checkType $(\Sigma_1, [])$ τ *and* checkExp $(\Sigma_1, [])$ e τ *succeeded.*

PROOF. This follows by definition of valid typed environment □

PROPOSITION 5.2. *Let Σ be a valid typed environment, e_1 and e_2 two expressions and x any variable.*
Then $(e_1[x := e_2])^*_\Sigma = {e_1}^*_\Sigma[x := {e_2}^*_\Sigma]$.

PROOF. The proof is done by structural induction on the expression e_1. We show here the cases when it is either a constant, a variable or an abstraction.

$e_1 = y$ ▷. We now proceed by cases on $y = x$.

$y = x$ ▷ Both expressions reduce to ${e_2}^*_\Sigma$.

$y \neq x$ ▷ Both expressions reduce to y.

$e_1 = c$ ▷ We first consider the case when c is declared in the environment $\mathcal{E}$. By definition of substitution we have first that $(c[x := e_2])^*_\Sigma$ is equal to

c^*_Σ. Thus, by definition of unfolding, this expression in turn reduces to e^*_Σ, if $c = e : \alpha$ in $\mathcal{E}$. On the other hand, $c^*_\Sigma[x := e_2]$ is equal to $e^*_\Sigma[x := e_2]$. Now, by Proposition 5.1 and definition of the checking programs we know that e is well-formed, therefore e^*_Σ is also well-formed. Thus, by Proposition 4.1, the substitution $e^*_\Sigma[x := e_2]$ can not have effect. Therefore, $e^*_\Sigma[x := e_2]$ is also equal to e^*_Σ.

The case in that c does not belong to $\mathcal{E}$ is trivial.

$e_1 = [y]f$ ▷ The proof proceed by cases on $y = x$.

$y = x$ ▷ By definition of substitution we first have that $(([x]f)[x := e_2])^*_\Sigma$ is equal to $([x]f)^*_\Sigma$, which in turn is equal to $[x]f^*_\Sigma$. On the other hand, it is not difficult to see that the expression $([x]f^*_\Sigma)[x := e_2{}^*_\Sigma]$ also reduces to $[x]f^*_\Sigma$.

$y \neq x$ ▷ By definition of substitution and unfolding we have that the expression $(([y]f)[x := e_2])^*_\Sigma$ reduces to $[y](f[x := e_2])^*_\Sigma$. On the other hand, $([y]f)^*_\Sigma[x := e_2{}^*_\Sigma]$ reduces to $[y](f^*_\Sigma[x := e_2{}^*_\Sigma])$. Then, we can apply induction to get that both abstractions are equal.

□

We now digress to discuss the issue of the termination of the algorithm of type checking presented in the previous section.

In contrast to the usual presentation of this kind of algorithms, we have used a recursively defined function to compute the weak head normal form of well-typed expressions. Actually, the definition given in Figure 4.8 of section 3.3 is a slightly modified version of its corresponding definition in Haskell [**Pet96**]. In accordance with this, then, when defining the semantics of our programs we explicitly introduced the termination requirement for the whole checking procedure to succeed.

There is, in principle, no need for proving that the function $\Downarrow$ is normalizing on types and objects of certain types, for being able to give a proof of soundness of the algorithm. This is not the case if we want to establish its decidability. We have already pointed out in chapter 3, and it is also clear from the definition of the programs, that the whole process of type checking is ultimately reduced to the checking of object conversion, which in turn, for being succesful, needs to completely normalize its arguments.

We could have, on the other hand, defined an inductive reduction relation, which we show in Figure 5.5, which extends the relation $\Rightarrow$ introduced in chapter 4 to consider the unfolding of constants present in the typed environment. Therefore, any premiss of the form $e_1 \gg e_2$ would be replaced by one of the form $e_1 \overset{\Sigma}{\Rightarrow} e_2$. But then, we would place ourselves in the situation that what we are defining, when introducing $\mathsf{checkExp}\ \mathcal{E}\ e\ f$ for instance, is closer to an inductively defined predicate on expressions than a program. This latter approach is particularly useful if one wants to carry out proofs as the one we shall present in the next section, because then we can apply the natural induction principles that can be obtained from the definition of the relations in question.

We shall need, in particular, to characterize the interplay of the function $\Downarrow$ with the relation $\Rightarrow$. More precisely, we need the following

CLAIM 1. Given a well-formed expression e_1, and a valid typed environment Σ, If $e_1 \Downarrow \Sigma \gg e_2$ then $e_1 \overset{\Sigma}{\Rightarrow} e_2$.

It is quite easy to prove, on the other hand, by induction on the derivation of $e_1 \overset{\Sigma}{\Rightarrow} e_2$, that if $e_1 \overset{\Sigma}{\Rightarrow} e_2$ then ${e_1}^*_\Sigma \Rightarrow {e_2}^*_\Sigma$. Therefore, we will understand, in the proofs that follows, that an assumption of the form $e_1 \Downarrow \Sigma \gg e_2$ amounts to one of the form ${e_1}^*_\Sigma \Rightarrow {e_2}^*_\Sigma$.

4.2. Soundness.

PROPOSITION 5.3. *Let $\mathcal{E}$ be the valid checking environment (Σ, Γ). Then it holds that Γ^*_Σ context.*

PROOF. By induction on the definition of valid typed environment.

$\Gamma = []$ $\rhd$ Trivial

$\Gamma = \Gamma_1, p{:}\alpha$ $\rhd$ By definition of valid checking environment we know that (Σ,Γ_1) is also valid, that p is fresh for the latter context and that ${\Gamma_1}^*_\Sigma \vdash \alpha^*_\Sigma : \mathit{type}$. Then, by induction, ${\Gamma_1}^*_\Sigma$ *context* and we also have that p is also fresh for it by definition of unfolding. Thus, by rule of context formation we get that ${\Gamma_1}^*_\Sigma, p{:}\alpha^*_\Sigma$ is a context.

□

PROPOSITION 5.4. *Let $\mathcal{E}$ be the valid checking environment (Σ, Γ),*

$th_1 \diamond$ *if* $\mathsf{checkType}\ \mathcal{E}\ \alpha$ *then* $\Gamma^*_\Sigma \vdash \alpha^*_\Sigma : \mathit{type}$

$th_2 \diamond$ *if* $\mathsf{checkRecType}\ \mathcal{E}\ \rho$ *then* $\Gamma^*_\Sigma \vdash \rho^*_\Sigma : \mathit{record\text{-}type}$

$th_3 \diamond$ *if* $\mathsf{checkType}\ \mathcal{E}\ \alpha$ *and* $\mathsf{checkTypeFam}\ \mathcal{E}\ \beta\ \alpha$
then $\Gamma^*_\Sigma \vdash \beta^*_\Sigma : \alpha^*_\Sigma \to \mathit{type}$.

$th_4 \diamond$ *if* $\Gamma^*_\Sigma \vdash \alpha^*_\Sigma : \mathit{type}$ *and* $\mathsf{checkExp}\ \mathcal{E}\ a\ \alpha$
then $\Gamma^*_\Sigma \vdash a^*_\Sigma : \alpha^*_\Sigma$.

$th_5 \diamond$ *if* $\mathsf{inferExp}\ \mathcal{E}\ f \gg \alpha$ *then either*

i) $\Gamma^*_\Sigma \vdash \alpha^*_\Sigma : \mathit{type}$ *and* $\Gamma^*_\Sigma \vdash f^*_\Sigma : \alpha^*_\Sigma$,

ii) $\alpha \in \mathcal{T}$ *and* $\Gamma^*_\Sigma \vdash f^*_\Sigma : \alpha$ *or*

iii) $\alpha = \alpha_1 \to [x]t$, $t \in \mathcal{T}$ *and* $\Gamma^*_\Sigma \vdash f^*_\Sigma : {\alpha_1}^*_\Sigma \to [x]t$

PROOF. The proof is done by simultaneous induction on the definitions of the programs involved in the proposition.

th_1 $\rhd$

$\alpha = \mathit{Set}$ $\rhd$ Assume $\mathsf{checkType}\ \mathcal{E}\ \mathit{Set}$.

$$p \overset{\Sigma}{\Rightarrow} p$$

$$Set \overset{\Sigma}{\Rightarrow} Set$$

$$\alpha{\rightarrow}\beta \overset{\Sigma}{\Rightarrow} \alpha{\rightarrow}\beta$$

$$\langle\rangle \overset{\Sigma}{\Rightarrow} \langle\rangle$$

$$\langle\rho, L{:}\beta\rangle \overset{\Sigma}{\Rightarrow} \langle\rho, L{:}\beta\rangle$$

$$[x]e \overset{\Sigma}{\Rightarrow} [x]e$$

$$\langle r, L = e\rangle \overset{\Sigma}{\Rightarrow} \langle r, L = e\rangle$$

$$\frac{e \overset{\Sigma_1}{\Rightarrow} v}{c \overset{\Sigma}{\Rightarrow} v}\ {\scriptstyle \Sigma = \Sigma_1;\, c\,:\,\tau = e;\, \Sigma_2}$$

$$\frac{f \overset{\Sigma}{\Rightarrow} [x]e \quad e[x := a] \overset{\Sigma}{\Rightarrow} v}{f a \overset{\Sigma}{\Rightarrow} v}$$

$$\frac{f \overset{\Sigma}{\Rightarrow} f_1}{f a \overset{\Sigma}{\Rightarrow} f_1 a}\ {\scriptstyle f_1 \neq [x]e}$$

$$\frac{r \overset{\Sigma}{\Rightarrow} \langle r_1, L_1 = e\rangle \quad e \overset{\Sigma}{\Rightarrow} v}{r.L \overset{\Sigma}{\Rightarrow} v}\ {\scriptstyle L = L_1}$$

$$\frac{r \overset{\Sigma}{\Rightarrow} \langle r_1, L_1 = e\rangle \quad r_1.L \overset{\Sigma}{\Rightarrow} v}{r.L \overset{\Sigma}{\Rightarrow} v}\ {\scriptstyle L \neq L_1}$$

$$\frac{r \overset{\Sigma}{\Rightarrow} r_1}{r.L \overset{\Sigma}{\Rightarrow} r_1.L}\ {\scriptstyle r_1 \neq \langle r_2, L_2 = e\rangle}$$

FIGURE 5.5. Weak head reduction relation in a typed environment

Then, as Γ^*_Σ *context* we have that $\Gamma^*_\Sigma \vdash Set : type$ by type formation and thinning.

$\alpha = \alpha_1{\rightarrow}\beta$ $\triangleright$ Assume checkType $\mathcal{E}$ $\alpha_1{\rightarrow}\beta$.
Then we know that

i) checkType $\mathcal{E}$ α_1

ii) checkTypeFam $\mathcal{E}$ β α_1

We can apply induction to get both that $\Gamma^*_\Sigma \vdash \alpha^*_\Sigma : type$ and also that $\Gamma^*_\Sigma \vdash \beta^*_\Sigma : \alpha^*_\Sigma \to type$. Then by function type formation we can derive that $\Gamma^*_\Sigma \vdash {\alpha_1}^*_\Sigma \to \beta^*_\Sigma : type$.

$\alpha = f$ $\triangleright$ Assume checkType $\mathcal{E}$ f.
We know then that inferExp $\mathcal{E}$ f $\gg s$ and $s \in \mathcal{S}$. Thus, in the case that s is the sort *Set*, by th_5 we know that $\Gamma^*_\Sigma \vdash f^*_\Sigma : Set$. Therefore, by type formation we obtain that $\Gamma^*_\Sigma \vdash f^*_\Sigma : type$. Otherwise, s is either the sort *type* or *record-type*. Thus, th_5 gives either $\Gamma^*_\Sigma \vdash f^*_\Sigma : type$ or $\Gamma^*_\Sigma \vdash f^*_\Sigma : record\text{-}type$ respectively.

$\alpha = \langle\rangle$ $\triangleright$ Analogous to the case *Set*.

$\alpha = \langle \rho, L{:}\beta \rangle$ $\triangleright$ Assume checkType $\mathcal{E}$ $\langle \rho, L{:}\beta \rangle$.
Then we know that checkRecType $\mathcal{E}$ $\langle \rho, L{:}\beta \rangle$. By th_2 we know that $\Gamma^*_\Sigma \vdash \langle \rho^*_\Sigma, L{:}\beta^*_\Sigma \rangle : record\text{-}type$. Thus, from type formation for record types follows the desired conclusion.

th_2 $\triangleright$

$\rho = \langle\rangle$ $\triangleright$ Trivial.

$\rho = \langle \rho, L{:}\beta \rangle$ $\triangleright$ Assume that checkRecType $\mathcal{E}$ $\langle \rho, L{:}\beta \rangle$.
Then we know that checkRecType $\mathcal{E}$ ρ and checkTypeFam $\mathcal{E}$ β ρ, and moreover, L fresh in ρ. .
Induction gives that $\Gamma^*_\Sigma \vdash \rho^*_\Sigma : record\text{-}type$ and, furthermore, th_2 gives that $\Gamma^*_\Sigma \vdash \beta^*_\Sigma : \rho^*_\Sigma \to type$. The second rule of record type formation can then be applied because L is also fresh for ρ^*_Σ by definition of unfolding.

$\rho = f$ $\triangleright$ Assume checkRecType $\mathcal{E}$ f.
We know then that inferExp $\mathcal{E}$ f $\gg record\text{-}type$. By th_6 we get that $\Gamma^*_\Sigma \vdash f^*_\Sigma : record\text{-}type$.

th_3 $\triangleright$

$\beta = [x]\alpha_1$ $\triangleright$ Assume checkType $\mathcal{E}$ α and checkTypeFam $\mathcal{E}$ $[x]\alpha_1$ α.
We know then that checkType $\mathcal{E}, p{:}\alpha$ $\alpha_1[x := p]$. In the first place, as p is a fresh parameter and checkType $\mathcal{E}$ α by hypothesis we have that $\mathcal{E}, p{:}\alpha$ is a valid environment. Observe that this latter checking environment is equal to $(\Sigma, \Gamma, p{:}\alpha)$. Thus, by Proposition 5.3 , definition of unfolding for contexts and th_1 we get that $\Gamma^*_\Sigma, p{:}\alpha^*_\Sigma \vdash \alpha_1[x := p]^*_\Sigma : type$. We can use Proposition 5.2 to get that the latter type is equal to ${\alpha_1}^*_\Sigma[x := p]$. Finally, the rule for type family formation can then be applied to get that $\Gamma^*_\Sigma \vdash [x]{\alpha_1}^*_\Sigma : \alpha^*_\Sigma \to type$.

$\beta = F$ $\triangleright$ Assume that $\mathsf{checkTypeFam}\ \ \mathcal{E}\ \ F\ \ \alpha$.
We know then $\mathsf{typeIncl}\ \ \mathcal{E}\ \ \alpha_1\ \ \alpha$ and further, that $F : \alpha_1 \to [x]s = e$ is a declaration in $\mathcal{E}$. Thus, by Proposition 5.1 we have that for a typed environment Σ_1 included in Σ. $\mathsf{checkTypeFam}\ \ (\Sigma_1, [])\ \ e\ \ \alpha_1$, On the other hand by Proposition 5.5 we also know that $\Gamma^*_\Sigma \vdash {\alpha_1}^*_\Sigma \sqsubseteq \alpha^*_\Sigma$. We can then apply induction to get, first, that $\Gamma^*_\Sigma \vdash e^*_\Sigma : {\alpha_1}^*_\Sigma \to \mathit{type}$. Finally, the rule of subtyping for families gives that $\Gamma^*_\Sigma \vdash e^*_\Sigma : \alpha^*_\Sigma \to \mathit{type}$

th_4 $\triangleright$

$a = [x]e$ $\triangleright$ Assume that $\mathsf{checkExp}\ \ \mathcal{E}\ \ [x]e\ \ \alpha$.
Then we know that

i) $\alpha \Downarrow \mathcal{E} = \gamma \to \beta$

ii) $\mathsf{checkExp}\ \ \mathcal{E}, p{:}\gamma\ \ e[x := p]\ \ \beta p$, with p a fresh parameter.

By claim in the previous section we have that $\alpha^*_\Sigma \Rightarrow \gamma^*_\Sigma \to \beta^*_\Sigma$. Now, by Proposition 4.9 we have that $\Gamma^*_\Sigma \vdash \gamma^*_\Sigma \to \beta^*_\Sigma : \mathit{type}$, and moreover, that $\Gamma^*_\Sigma \vdash \alpha^*_\Sigma = \gamma^*_\Sigma \to \beta^*_\Sigma : \mathit{type}$ From the former, in turn, we get that $\Gamma^*_\Sigma \vdash \gamma^*_\Sigma : \mathit{type}$ and also that $\Gamma^*_\Sigma \vdash \beta^*_\Sigma : \gamma^*_\Sigma \to \mathit{type}$. We can then use induction hypothesis to get that $\Gamma^*_\Sigma \vdash e[x := p]^*_\Sigma : \beta p^*_\Sigma$. We show below the derivation that $\Gamma^*_\Sigma \vdash [x]e^*_\Sigma : \gamma^*_\Sigma \to \beta^*_\Sigma$.

$$\dfrac{\dfrac{\dfrac{\Gamma^*_\Sigma, p{:}\gamma^*_\Sigma \vdash e[x := p]^*_\Sigma : \beta p^*_\Sigma}{\Gamma^*_\Sigma, p{:}\gamma^*_\Sigma \vdash e^*_\Sigma[x := p] : \beta^*_\Sigma x[x := p]}}{\Gamma^*_\Sigma \vdash [x]e^*_\Sigma : \gamma^*_\Sigma \to [x]\beta x} \qquad \dfrac{\Gamma^*_\Sigma \vdash \gamma^*_\Sigma \to [x]\beta^*_\Sigma x = \gamma^*_\Sigma \to \beta^*_\Sigma : \mathit{type}}{\Gamma^*_\Sigma \vdash \gamma^*_\Sigma \to [x]\beta^*_\Sigma x \sqsubseteq \gamma^*_\Sigma \to \beta^*_\Sigma}}{\Gamma^*_\Sigma \vdash [x]e^*_\Sigma : \gamma^*_\Sigma \to \beta^*_\Sigma}$$

In the uppermost step of derivation in the left branch we have used Proposition 5.2 for obtaining the object and that β is independent of p and Proposition 4.2 for obtaining the type. The leaf judgement of the right branch was shown to hold chapter 4. Finally, then, we can use subsumption, due to the equality of α^*_Σ and $\gamma^*_\Sigma \to \beta^*_\Sigma$ to obtain that $\Gamma^*_\Sigma \vdash [x]e^*_\Sigma : \alpha^*_\Sigma$.

$a = \langle\rangle$ $\triangleright$ Trivial.

$a = \langle r, L = e\rangle$ $\triangleright$ Assume that $\mathsf{checkExp}\ \ \mathcal{E}\ \ \langle r, L = e\rangle$. Then we have that

i) $\alpha \Downarrow \mathcal{E} = \langle \rho, L{:}\beta\rangle$

ii) $\mathsf{checkExp}\ \ \mathcal{E}\ \ r\ \ \rho$

iii) $\mathsf{checkExp}\ \ \mathcal{E}\ \ e\ \ \beta r$

We can use the claim again to get that $\alpha^*_\Sigma \Rightarrow \langle \rho^*_\Sigma, L{:}\beta^*_\Sigma\rangle$. By a similar reasoning as above we get that $\Gamma^*_\Sigma \vdash \rho^*_\Sigma : \mathit{type}$ and $\Gamma^*_\Sigma \vdash \beta^*_\Sigma : \rho^*_\Sigma \to \mathit{type}$.

Induction then gives that $\Gamma^*_\Sigma \vdash r^*_\Sigma : \rho^*_\Sigma$ and therefore $\Gamma^*_\Sigma \vdash \beta^*_\Sigma r^*_\Sigma : \mathit{type}$. Thus, we can again use induction to get $\Gamma^*_\Sigma \vdash e^*_\Sigma : \beta^*_\Sigma r^*_\Sigma$. Finally, we can apply the rule of record object extension to get the desired conclusion.

$a = f$ $\triangleright$ With the expresion f we mean here a generalized selection. Assume then that $\mathsf{checkExp}\ \ \mathcal{E}\ \ f\ \ \alpha$.
Therefore we can assume that

i) $\mathsf{inferExp}\ \ \mathcal{E}\ \ f\ \gg \alpha_1$

ii) $\mathsf{typeIncl}\ \ \mathcal{E}\ \ \alpha_1\ \ \alpha$

iii) $\alpha_1 \notin \mathcal{S}$and $\alpha_1 \neq \alpha_2 \rightarrow [x]t$

Then th_6 says that it must be the case that $\Gamma^*_\Sigma \vdash {\alpha_1}^*_\Sigma : \mathit{type}$, and moreover, that $\Gamma^*_\Sigma \vdash f^*_\Sigma : {\alpha_1}^*_\Sigma$. Then, as Proposition 5.5 below gives that $\Gamma^*_\Sigma \vdash {\alpha_1}^*_\Sigma \sqsubseteq \alpha^*_\Sigma$ we finally can apply the rule of subsumption to get that $\Gamma^*_\Sigma \vdash f^*_\Sigma : \alpha^*_\Sigma$.

th_5 $\triangleright$

$f = p$ $\triangleright$ Assume that $\mathsf{inferExp}\ \ \mathcal{E}\ \ p\ \gg \alpha$.
We have the side condition $p : \alpha$ in Γ. On the one hand, then, we have by Proposition 5.3 that Γ^*_Σ *context*. It is easy to see, by definition of unfolding, that $p : \alpha^*_\Sigma$ in Γ^*_Σ. Thus, by general rule and rule of assumption we obtain that $\Gamma^*_\Sigma \vdash \alpha^*_\Sigma : \mathit{type}$ and $\Gamma^*_\Sigma \vdash p : \alpha^*_\Sigma$.

$f = c$ $\triangleright$ Assume that $\mathsf{inferExp}\ \ \mathcal{E}\ \ p\ \gg \alpha$. We know then that $\Sigma = \Sigma_1; c : \tau = e; \Sigma_1$. By definition of valid environment we know that Σ_1 is also valid, then we can use Proposition 5.1 and induction.

$f = ge$ $\triangleright$ Assume $\mathsf{inferExp}\ \ \mathcal{E}\ \ ge\ \gg \beta e$.
We know then that

i) $\mathsf{inferExp}\ \ \mathcal{E}\ \ ge\ \gg \alpha$

ii) $\alpha \Downarrow \Sigma = \alpha_1 \rightarrow \beta$

iii) $\mathsf{checkExp}\ \ \mathcal{E}\ \ e\ \ \alpha_1$

We will assume that $\Gamma^*_\Sigma \vdash {\alpha_1}^*_\Sigma : \mathit{type}$ in order to apply th_5 using iii) and have at hand that $\Gamma^*_\Sigma \vdash a^*_\Sigma : {\alpha_1}^*_\Sigma$. We shall make sure that this assumption is valid. We then first apply induction with i), and now proceed by case analysis

1) We have that $\Gamma^*_\Sigma \vdash \alpha^*_\Sigma : \mathit{type}$ and $\Gamma^*_\Sigma \vdash f^*_\Sigma : \alpha^*_\Sigma$.
By claim then we get that $\alpha^*_\Sigma \Rightarrow {\alpha_1}^*_\Sigma \rightarrow \beta^*_\Sigma$, and further, by Proposition 4.9 $\Gamma^*_\Sigma \vdash {\alpha_1}^*_\Sigma \rightarrow \beta^*_\Sigma : \mathit{type}$. From the latter we also know that $\Gamma^*_\Sigma \vdash {\alpha_1}^*_\Sigma : \mathit{type}$ and $\Gamma^*_\Sigma \vdash \beta^*_\Sigma : {\alpha_1}^*_\Sigma \rightarrow \mathit{type}$. Thus, by application rules we finally get that $\Gamma^*_\Sigma \vdash \beta^*_\Sigma a^*_\Sigma : \mathit{type}$ and $\Gamma^*_\Sigma \vdash f^*_\Sigma a^*_\Sigma : \beta^*_\Sigma a^*_\Sigma$.

2) This is an impossible case. It cannot be both that $\alpha \in \mathcal{T}$ and $\alpha \Downarrow \Sigma = \alpha_1 \to \beta$.

3) We have that $\alpha = \alpha_2 \to [x]t$ and $\Gamma^*_\Sigma \vdash f^*_\Sigma : {\alpha_2}^*_\Sigma \to type$. From this we know that $\Gamma^*_\Sigma \vdash {\alpha_2}^*_\Sigma : type$. By ii) and definition of $\Downarrow$ we get then that $\alpha_1 = \alpha_2$ and that $\beta = [x]t$. From the former we also get then that $\Gamma^*_\Sigma \vdash {\alpha_1}^*_\Sigma : type$. Thus, $\beta a = t$, and thereby it belongs to $\mathcal{T}$. Finally, application rule for families gives that $\Gamma^*_\Sigma \vdash f^*_\Sigma a^*_\Sigma : type$.

$f = r.L$ $\triangleright$ Assume $\mathsf{inferExp}\ \mathcal{E}\ \ r.L\ \gg \beta r$.
We then know that

i) $\mathsf{inferExp}\ \mathcal{E}\ \ r\ \gg \rho$

ii) $L : \beta$ in ρ

Induction on i) gives only one possible case, namely, that $\Gamma^*_\Sigma \vdash \rho^*_\Sigma : type$ and $\Gamma^*_\Sigma \vdash r^*_\Sigma : \rho^*_\Sigma$. This is because we know that for ii) to hold, ρ must be a record form, and so must ρ^*_Σ, by definition of unfolding. Moreover we also know that $L : \beta^*_\Sigma$ in ρ^*_Σ (unfolding preserves the structure of the expressions). Then by rule of fields we have that $\Gamma^*_\Sigma \vdash \beta^*_\Sigma : \rho^*_\Sigma \to type$, we can then apply the rule of selection to achieve that $\Gamma^*_\Sigma \vdash r^*_\Sigma.L : \beta^*_\Sigma r^*_\Sigma$.

□

PROPOSITION 5.5. *Let $\mathcal{E}$ be the valid environment (Σ, Γ) and let us assume*

- $\Gamma^*_\Sigma \vdash {\alpha_1}^*_\Sigma : type$ *and* $\Gamma^*_\Sigma \vdash {\alpha_2}^*_\Sigma : type$ *for the cases* th_1 *and* th_2.
- $\Gamma^*_\Sigma \vdash \alpha^*_\Sigma : type$, $\Gamma^*_\Sigma \vdash a^*_\Sigma : \alpha^*_\Sigma$ *and* $\Gamma^*_\Sigma \vdash b^*_\Sigma : \alpha^*_\Sigma$ *for the cases* th_3 *and* th_4.

th_1 ◇ *If* $\mathsf{typeIncl}\ \mathcal{E}\ \ \alpha_1\ \ \alpha_2$ *then* $\Gamma^*_\Sigma \vdash {\alpha_1}^*_\Sigma \sqsubseteq {\alpha_2}^*_\Sigma$

th_2 ◇ *If* $\mathsf{whTypeIncl}\ \mathcal{E}\ \ \alpha_1\ \ \alpha_2$ *then* $\Gamma^*_\Sigma \vdash {\alpha_1}^*_\Sigma \sqsubseteq {\alpha_2}^*_\Sigma$

th_3 ◇ *If* $\mathsf{objConv}\ \mathcal{E}\ \ a\ \ b\ \ \alpha$ *then* $\Gamma^*_\Sigma \vdash a^*_\Sigma = b^*_\Sigma : \alpha^*_\Sigma$

th_4 ◇ *If* $\mathsf{whObjConv}\ \mathcal{E}\ \ a\ \ b\ \ \alpha$ *then* $\Gamma^*_\Sigma \vdash a^*_\Sigma = b^*_\Sigma : \alpha^*_\Sigma$

th_5 ◇ *If* $\mathsf{headConv}\ \mathcal{E}\ \ a\ \ b\ \gg \tau$
then $\Gamma^*_\Sigma \vdash \tau^*_\Sigma : type$ *and* $\Gamma^*_\Sigma \vdash a^*_\Sigma = b^*_\Sigma : \tau^*_\Sigma$

PROOF. This proof is by simultaneous induction on the definition of the programs and functions above.

th_1 $\triangleright$

$\triangleright$ Assume $\mathsf{typeIncl}\ \mathcal{E}\ \ \alpha\ \ \alpha$.
Then $\Gamma^*_\Sigma \vdash \alpha^*_\Sigma \sqsubseteq \alpha^*_\Sigma$ by reflexivity.

$\triangleright$ Assume $\mathsf{typeIncl}\ \mathcal{E}\ \ \alpha_1\ \ \alpha_2$. We know then that

i) $\alpha_1 \Downarrow \mathcal{E} = {\alpha_1}'$

ii) $\alpha_2 \Downarrow \mathcal{E} = {\alpha_2}'$

iii) $\mathsf{whTypeIncl}\ \mathcal{E}\ \ {\alpha_1}'\ \ {\alpha_2}'$

By the claim we have then that ${\alpha_1}^*_\Sigma \Rightarrow ({\alpha_1}')^*_\Sigma$ and also that ${\alpha_2}^*_\Sigma \Rightarrow ({\alpha_2}')^*_\Sigma$. Thus by Proposition 4.9 we have that

iv) $\Gamma^*_\Sigma \vdash ({\alpha_1}')^*_\Sigma : \mathit{type}$ and $\Gamma^*_\Sigma \vdash \alpha_1 = ({\alpha_1}')^*_\Sigma : \mathit{type}$

v) $\Gamma^*_\Sigma \vdash ({\alpha_2}')^*_\Sigma : \mathit{type}$ and $\Gamma^*_\Sigma \vdash \alpha_2 = ({\alpha_2}')^*_\Sigma : \mathit{type}$

and induction gives that

vi) $\Gamma^*_\Sigma \vdash ({\alpha_1}')^*_\Sigma \sqsubseteq ({\alpha_2}')^*_\Sigma$

From iv) we get that $\Gamma^*_\Sigma \vdash {\alpha_1}^*_\Sigma \sqsubseteq ({\alpha_1}')^*_\Sigma$. This latter and vi) give that $\Gamma^*_\Sigma \vdash {\alpha_1}^*_\Sigma \sqsubseteq ({\alpha_2}')^*_\Sigma$ by transitivity. From v), we have, first, that $\Gamma^*_\Sigma \vdash ({\alpha_2}')^*_\Sigma = {\alpha_2}^*_\Sigma : \mathit{type}$, by symmetry of type equality, and thus $\Gamma^*_\Sigma \vdash ({\alpha_2}')^*_\Sigma \sqsubseteq {\alpha_2}^*_\Sigma$.
Finally, then, again using transitivity of type inclusion, we get that $\Gamma^*_\Sigma \vdash {\alpha_1}^*_\Sigma \sqsubseteq {\alpha_2}^*_\Sigma$.

th_2 ▷

▷ Assume $\mathsf{whTypeIncl}\ \mathcal{E}\ \ \alpha_1 \to \beta_1\ \ \alpha_2 \to \beta_2$.
We are allowed to assume then that

i) $\mathsf{typeIncl}\ \mathcal{E}\ \ \alpha_2\ \ \alpha_1$

ii) $\mathsf{typeIncl}\ \mathcal{E}, p{:}\alpha_2\ \ \beta_1 p\ \ \beta_2 p$, with p a fresh parameter.

We know by hypothesis that $\Gamma^*_\Sigma \vdash {\alpha_1}^*_\Sigma \to {\beta_1}^*_\Sigma : \mathit{type}$ and also that $\Gamma^*_\Sigma \vdash {\alpha_2}^*_\Sigma \to {\beta_2}^*_\Sigma : \mathit{type}$. Then, in particular we know that $\Gamma^*_\Sigma \vdash {\alpha_1}^*_\Sigma : \mathit{type}$, $\Gamma^*_\Sigma \vdash {\alpha_2}^*_\Sigma : \mathit{type}$, $\Gamma^*_\Sigma \vdash {\beta_1}^*_\Sigma : {\alpha_1}^*_\Sigma \to \mathit{type}$ and $\Gamma^*_\Sigma \vdash {\beta_2}^*_\Sigma : {\alpha_2}^*_\Sigma \to \mathit{type}$. Therefore, in the first place, the environment $\mathcal{E}, p{:}\alpha_2$ is valid, and further, ${\beta_1}^*_\Sigma p$ and ${\beta_2}^*_\Sigma p$ are types under Γ^*_Σ. Thus we can apply induction to get that

iii) $\Gamma^*_\Sigma \vdash {\alpha_2}^*_\Sigma \sqsubseteq {\alpha_1}^*_\Sigma$

iv) $\Gamma^*_\Sigma, p{:}{\alpha_2}^*_\Sigma \vdash {\beta_1}^*_\Sigma p \sqsubseteq {\beta_2}^*_\Sigma p$.

Actually, that ${\beta_1}^*_\Sigma p$ is a type can be derived after iii) and subtyping for families. As done for equality of families of types, it is possible to justify a rule of "extensionality" for the inclusion of family of types. Thereby, from iv) we get that $\Gamma^*_\Sigma \vdash {\beta_1}^*_\Sigma \sqsubseteq {\beta_2}^*_\Sigma : {\alpha_2}^*_\Sigma \to \mathit{type}$. Then, we can apply the rule of function type inclusion to get that $\Gamma^*_\Sigma \vdash {\alpha_1}^*_\Sigma \to {\beta_1}^*_\Sigma \sqsubseteq {\alpha_2}^*_\Sigma \to {\beta_2}^*_\Sigma$.

▷ Assume $\mathsf{whTypeIncl}\ \mathcal{E}\ \ \rho\ \ \langle\rangle$.
Then we apply the first rule of record types inclusion.

▷ Assume $\mathsf{whTypeIncl}\ \mathcal{E}\ \ \rho_1\ \ \langle \rho_2, L{:}\beta_2 \rangle$.
We know then that

i) $\mathsf{typeIncl}\ \mathcal{E}\ \ \rho_1\ \ \rho_2$

ii) $\mathsf{typeIncl}\ \ \mathcal{E}, p{:}\rho_1\ \ \beta_1 p\ \ \beta_2 p$, with p fresh and $L : \beta_1$ in ρ_1

The rest of the proof is completely analogous to the one for function types but using as the last step of derivation the second rule of record types inclusion.

$\triangleright$ Assume $\mathsf{whTypeIncl}\ \ \mathcal{E}\ \ \alpha_1\ \ \alpha_2$.
We know then that $\mathsf{objConv}\ \ \mathcal{E}\ \ \alpha_1\ \ \alpha_2\ \ Set$.
Induction [3] then gives that $\Gamma^*_\Sigma \vdash {\alpha_1}^*_\Sigma = {\alpha_2}^*_\Sigma : Set$. We can then construct the following derivation:

$$\cfrac{\cfrac{\Gamma^*_\Sigma \vdash {\alpha_1}^*_\Sigma = {\alpha_2}^*_\Sigma : Set}{\Gamma^*_\Sigma \vdash {\alpha_1}^*_\Sigma = {\alpha_2}^*_\Sigma : type}}{\Gamma^*_\Sigma \vdash {\alpha_1}^*_\Sigma \sqsubseteq {\alpha_2}^*_\Sigma}$$

th_3 $\triangleright$

$\triangleright$ Assume $\mathsf{objConv}\ \ \mathcal{E}\ \ a\ \ a\ \ \alpha$.
Then $\Gamma^*_\Sigma \vdash a^*_\Sigma = a^*_\Sigma : \alpha^*_\Sigma$ by reflexivity.

$\triangleright$ Assume $\mathsf{objConv}\ \ \mathcal{E}\ \ a\ \ b\ \ \alpha$. We know then that

i) $\alpha \Downarrow \mathcal{E} = \alpha_1$

ii) $\mathsf{whObjConv}\ \ \mathcal{E}\ \ a\ \ b\ \ \alpha_1$

By i), claim and Proposition 4.9 we know that $\Gamma^*_\Sigma \vdash \alpha^*_\Sigma = {\alpha_1}^*_\Sigma : type$. Thus, by symmetry of type equality and inclusion from identity we have that $\Gamma^*_\Sigma \vdash {\alpha_1}^*_\Sigma \sqsubseteq \alpha^*_\Sigma$. Induction on ii) gives that $\Gamma^*_\Sigma \vdash a^*_\Sigma = b^*_\Sigma : {\alpha_1}^*_\Sigma$. Finally, the second rule of subsumption gives that $\Gamma^*_\Sigma \vdash a^*_\Sigma = b^*_\Sigma : \alpha^*_\Sigma$.

th_4 $\triangleright$

$\triangleright$ Assume $\mathsf{whObjConv}\ \ \mathcal{E}\ \ f\ \ g\ \ \alpha{\to}\beta$.
We can, then, in turn, assume that $\mathsf{objConv}\ \ \mathcal{E}, p{:}\alpha\ \ fp\ \ gp$, with p a fresh parameter. As $\Gamma^*_\Sigma \vdash \alpha^*_\Sigma{\to}\beta^*_\Sigma : type$, we know that β^*_Σ is a type family over the type α^*_Σ. Therefore we have that $\Gamma^*_\Sigma, p{:}\alpha^*_\Sigma \vdash \beta^*_\Sigma p : type$. Moreover, we also have that $\Gamma^*_\Sigma \vdash f^*_\Sigma p : \beta^*_\Sigma p$ and $\Gamma^*_\Sigma \vdash g^*_\Sigma p : \beta^*_\Sigma p$. We can then use induction to obtain that $\Gamma^*_\Sigma, p{:}\alpha^*_\Sigma \vdash f^*_\Sigma p = g^*_\Sigma p : \beta^*_\Sigma p$. Thereby, the rule of extensionality for function objects can be applied to finally get that $\Gamma^*_\Sigma \vdash f^*_\Sigma = g^*_\Sigma : \alpha^*_\Sigma{\to}\beta^*_\Sigma$.

$\triangleright$ Assume $\mathsf{whObjConv}\ \ \mathcal{E}\ \ r\ \ s\ \ \langle\rangle$.
The first rule of record objects equality gives directly $\Gamma^*_\Sigma \vdash r^*_\Sigma = s^*_\Sigma : \langle\rangle$

$\triangleright$ Assume $\mathsf{whObjConv}\ \ \mathcal{E}\ \ r\ \ s\ \ \langle\rho, L{:}\beta\rangle$. We have that

i) $\mathsf{objConv}\ \ \mathcal{E}\ \ r\ \ s\ \ \rho$

[3] In order to apply induction we need to know that both are objects of type Set! Observe that both α_1 and α_2 are in weak head normal form and we know that their respective unfoldings are types different from a record and a function type, thus necessarily they have to be objects of type Set.

ii) $\mathsf{objConv}\ \mathcal{E}\ \ r.L\ \ s.L\ \ \beta r$

As we know that $\Gamma^*_\Sigma \vdash \langle \rho^*_\Sigma, L{:}\beta^*_\Sigma\rangle : \mathit{record\text{-}type}$ we are allowed to assume that $\Gamma^*_\Sigma \vdash \rho^*_\Sigma : \mathit{record\text{-}type}$ and $\Gamma^*_\Sigma \vdash \beta^*_\Sigma : \rho^*_\Sigma \to \mathit{type}$. Then, in particular we know that $\Gamma^*_\Sigma \vdash \beta^*_\Sigma r^*_\Sigma : \mathit{type}$. By hypothesis and rule of selection we also know that both $r^*_\Sigma.L$ and $s^*_\Sigma.L$ are objects of type $\beta^*_\Sigma r$ under Γ^*_Σ. Now, by induction then we get that $\Gamma^*_\Sigma \vdash r^*_\Sigma = s^*_\Sigma : \rho^*_\Sigma$ and $\Gamma^*_\Sigma \vdash r^*_\Sigma.L = s^*_\Sigma.L : \beta^*_\Sigma r^*_\Sigma$.
These latter are the premisses needed to apply the second rule of record object equality to derive that $\Gamma^*_\Sigma \vdash r^*_\Sigma = s^*_\Sigma : \langle \rho^*_\Sigma, L{:}\beta^*_\Sigma\rangle$.

$\triangleright$ Assume $\mathsf{whObjConv}\ \mathcal{E}\ \ a\ \ b\ \ \alpha$.
We then know that

i) $a \Downarrow \mathcal{E} = a_1$

ii) $b \Downarrow \mathcal{E} = b_1$

iii) $\mathsf{headConv}\ \mathcal{E}\ \ a_1\ \ b_1 \gg \alpha_1$

iv) $\mathsf{typeIncl}\ \mathcal{E}\ \ \alpha_1\ \ \alpha$

We now use the claim and Proposition 4.10 to get that

v) $\Gamma^*_\Sigma \vdash {a_1}^*_\Sigma : \alpha^*_\Sigma$ and $\Gamma^*_\Sigma \vdash a^*_\Sigma = {a_1}^*_\Sigma : \alpha^*_\Sigma$

vi) $\Gamma^*_\Sigma \vdash {b_1}^*_\Sigma : \alpha^*_\Sigma$ and $\Gamma^*_\Sigma \vdash b^*_\Sigma = {b_1}^*_\Sigma : \alpha^*_\Sigma$

From iii) and induction we get that

vii) $\Gamma^*_\Sigma \vdash {\alpha_1}^*_\Sigma : \mathit{type}$ and $\Gamma^*_\Sigma \vdash {a_1}^*_\Sigma = {b_1}^*_\Sigma : {\alpha_1}^*_\Sigma$

As both α^*_Σ and ${\alpha_1}^*_\Sigma$ are types under Γ^*_Σ we get also by and iv) and th_1 that

viii) $\Gamma^*_\Sigma \vdash {\alpha_1}^*_\Sigma \sqsubseteq \alpha^*_\Sigma$

We can then use the second rule of subsumption with premisses vii) and viii) to get as result that $\Gamma^*_\Sigma \vdash {a_1}^*_\Sigma = {b_1}^*_\Sigma : \alpha^*_\Sigma$. Finally then, by symmetry and transitivity of object equality, v) and vi) we get that $\Gamma^*_\Sigma \vdash a^*_\Sigma = b^*_\Sigma : \alpha^*_\Sigma$.

th_5 $\triangleright$

$\triangleright$ Assume $\mathsf{headConv}\ \mathcal{E}\ \ p\ \ p \gg \alpha$. Then we have that $p : \alpha$ in Γ. A similar argument as in th_5 in Proposition 5.4 gives that $\Gamma^*_\Sigma \vdash \alpha^*_\Sigma : \mathit{type}$ and $\Gamma^*_\Sigma \vdash p : \alpha^*_\Sigma$. From the latter, in turn, we obtain that $\Gamma^*_\Sigma \vdash p = p : \alpha^*_\Sigma$.

$\triangleright$ Assume $\mathsf{headConv}\ \mathcal{E}\ \ fa\ \ gb \gg \beta a$. Then we know that

i) $\mathsf{headConv}\ \mathcal{E}\ \ f\ \ g \gg \alpha$

ii) $\alpha \Downarrow \mathcal{E} = \alpha_1 \to \beta$

iii) $\mathsf{objConv}\ \mathcal{E}\ \ a\ \ b\ \ \alpha_1$

By i) and induction we obtain that

iv) $\Gamma^*_\Sigma \vdash \alpha^*_\Sigma : \mathit{type}$

v) $\Gamma^*_\Sigma \vdash f^*_\Sigma = g^*_\Sigma : \alpha^*_\Sigma$

Now, ii) and Proposition 4.9 give that

vi) $\Gamma^*_\Sigma \vdash {\alpha_1}^*_\Sigma \to \beta^*_\Sigma : \mathit{type}$ and $\Gamma^*_\Sigma \vdash \alpha^*_\Sigma = {\alpha_1}^*_\Sigma \to \beta^*_\Sigma : \mathit{type}$

Therefore, we are allowed to assume that $\Gamma^*_\Sigma \vdash {\alpha_1}^*_\Sigma : \mathit{type}$ and $\Gamma^*_\Sigma \vdash \beta^*_\Sigma : {\alpha_1}^*_\Sigma \to \mathit{type}$. Moreover, from v), vi) and subsumption we obtain that

vii) $\Gamma^*_\Sigma \vdash f^*_\Sigma = g^*_\Sigma : \alpha_1 \to \beta$

Induction hypothesis and iii) give that $\Gamma^*_\Sigma \vdash a^*_\Sigma = b^*_\Sigma : {\alpha_1}^*_\Sigma$. Thus, by the rule of application , $\Gamma^*_\Sigma \vdash f^*_\Sigma a^*_\Sigma = g^*_\Sigma b^*_\Sigma : \beta^*_\Sigma a^*_\Sigma$. We also have that $\Gamma^*_\Sigma \vdash \beta^*_\Sigma a^*_\Sigma : \mathit{type}$.

$\triangleright$ Assume headConv $\mathcal{E}$ $r.L$ $s.L \gg \beta r$. Then we know that

i) headConv $\mathcal{E}$ r $s \gg \rho$

ii) $L : \beta$ in ρ

Induction hypothesis gives that $\Gamma^*_\Sigma \vdash \rho^*_\Sigma : \mathit{type}$ and $\Gamma^*_\Sigma \vdash r^*_\Sigma = s^*_\Sigma : \rho^*_\Sigma$. As $L : \beta$ in ρ, the type ρ^*_Σ must necessarily be a record type. Moreover, by the rule of fields, we get that $\Gamma^*_\Sigma \vdash \beta^*_\Sigma : \rho^*_\Sigma \to \mathit{type}$. Then $\beta^*_\Sigma r^*_\Sigma$ is a type under Γ^*_Σ.
Finally rule of selection gives that $\Gamma^*_\Sigma \vdash r^*_\Sigma.L = s^*_\Sigma.L : \beta^*_\Sigma r^*_\Sigma$.

□

5. Implementation of the proof checker

The proof checker described in previous sections has been implemented on machine. The programming language used is Haskell 1.3, and the code has been compiled using Chalmers Haskell-B, the compiler implemented by L. Augustsson at Chalmers University of Technology [**Aug97**].

The general design and implementation of the system follows the approach taken in the recent years by the implementation group of the Programming Logic group at the Department of Computing Science at the same university. That is to say, there is a basic kernel constituted by the type checking algorithm, and on top of that an interactive system is built up that helps the user in the process of proof construction. In our case the help amounts to very simple commands mostly oriented to obtain information from the proof environment and to the checking of declarations. It is also possible to type check files of declaration in a batch fashion. Furthermore, we have also adopted the methodology of developing a completely pure functional code. In particular, the state of the system is implemented as a simple monad, in the sense of [**Wad92**], which has associated a basic set of combinators that allow to access and update the state components. The type checking monad, is just a combination of a state and error monad, which also interacts with a parsing monad, this latter implemented along the lines of [**Bur75, Hut92, Röj95**]. Our code greatly benefited from the one developed for an early implementation of Half, a successor of ALF, by Thierry Coquand and Björn von Sydow, and also from an experimental type checker for the framework extended with dependent pairs implemented in collaboration with Daniel Fridlender.

In what follows we shall give a flavour of the implemented code, (partially) describing the monads referred to above to finally end up showing the part of the type checking function, `checkExp`, concerned with the checking of abstractions.

```
data  E a = Error String | Val a

instance Monad E where

  (Val x)    >>= k  = k x
  (Error s)  >>= _  = Error s
  return            = Val
```

FIGURE 5.6. Error monad

```
data STE s a = STE (s -> E (a,s))

funOfSTE (STE f) = f

instance Monad (STE s) where
   (STE f) >>= g  =  STE (\s -> f s >>= \(a,s') -> funOfSTE (g a) s')
   return         =  \a -> STE (\s -> return (a,s))

getSTE :: STE s s
getSTE = STE (\s -> return (s,s))

putSTE :: s -> STE s ()
putSTE s = STE (\_ -> return ((),s))

updateSTE :: (s -> s) -> STE s ()
updateSTE f = getSTE >>= (putSTE . f)
```

FIGURE 5.7. State-error monad

```
type CheckState = (Ctxt,Env,Par,...)

getCtxtParamChSt (c,_,i,_,_) = (c,i)

putParamChSt i (c,e,_,...) = (c,e,i,...)

putCtxtChSt c (_,e,i,...) = (c,e,i,...)
```

FIGURE 5.8. Checking state

```
type ChM a = STE CheckState a

selectChM :: (CheckState -> a) -> ChM a
selectChM acc =
  do
    chst <- getSTE
    return (acc chst)

getCtxtParamChM =
  do
    (c,i) <- selectChM getCtxtParamChSt
    putParamChM (i+1)
    return (c,i)

updateChM :: (a -> CheckState -> CheckState) -> a -> ChM ()
updateChM f = updateSTE . f

putCtxtChM = updateChM putCtxChSt
```

FIGURE 5.9. Checking monad

In Figure 5.6 we show the definition of the monad `E`, which is used for handling and raising errors. First a parameterized data type with two constructors is defined. It is almost the same as the primitive type `Maybe`, with the difference that in case of errors a string for providing a message is also considered. Then, this data type is declared as an instance of the primitive class Monad, by providing the implementation for the two basic combinators `>>=` and `return` , which are respectively called `bind` and `unit` in [**Wad92**].

In Figure 5.7 the definition of the state-error monad is given. The combinators `getSTE`, `putSTE` and `updateSTE` allow to recover the state, to initialize it (the argument `s`), and to update it using a given function, respectively. The operator `.` is function composition.

Then, the definition of the checking state, which is partially presented in Figure 5.8, is introduced as a tuple type, where the components `Ctxt`, `Env` and `Par` are the types of contexts, typed environments and the source of the "gensym" function, in this case an integer, respectively. The combinators `getCtxtParamChSt`, `putParamChSt` and `putParamChSt` allow to access and update the context and "fresh" parameter of the checking state, respectively.

The monad `ChM` is just a partial refinement of the monad `STE` where the state is instantiated to be the checking state `CheckState`. Its definition is given in Figure 5.9.

We illustrate the use of `do` expressions, as provided by Haskell, which allow to express monadic programming by means of a more readable syntax. The semantics of the combinator `selectChM`, which takes as argument a (selection) function on the checking state and returns the computation of that selection, is understood as follows: first compute `getSTE`, whose result, the state component, is bound to the pattern `chst`, and then return the computation resulting from applying the argument function `acc` to it. The combinator `getCtxtParamChM` selects the context and the counter of the state and also increments, as a "side-effect", the value of the latter. The behaviour of the rest of the combinators in the same figure is quite direct to grasp.

Finally, we show in Figure 5.10, the code of the function `checkExp` in the case that the expression to be checked is an abstraction. We also include the corresponding rule of computation as presented in section 3.2. The abstract syntax for the expressions $[x]e$ and $\alpha \to \beta$ is `EAbs x e` and `EPi alfa beta` respectively.

As already explained, the program is defined by case analysis on the object expression, which in this case is an abstraction. Thus, first the weak head normal form of the type expressions is computed by the function `whnf`, which is a direct implementation of the one in Figure 5.3. If it is a function type, then the context and the available parameter are recovered from the state. For the sake of readability, we use a let expression to abbreviate the expression obtained from substituting the fresh parameter for the variable `x` in the expression `e`, the body of the abstraction, and for building up the application of the type family β to the same parameter. Further, we also abbreviate by `c'` the new context obtained by extending `c` with the declaration of the parameter `p` as of type `alfa`. The effect of `mkParam` is to generate a parameter (which, as additional information, carries over the name of the bound variable being substituted) out of an integer value. Finally, then, the program is recursively performed on the expressions `e'` and `beta'` in the updated checking state.

A very simple XEmacs interface has also been incorporated to the system. The basic kernel was implemented by Guillermo Calderón, a researcher at the Department of Computing Science (InCo) at Montevideo, Uruguay. We then extended it to consider all the commands that were already present in the checking engine. 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 using the system.

$$\frac{\alpha \Downarrow \Sigma \;\gg \alpha_1 \!\to\! \beta \quad \mathsf{checkExp}\;\; \mathcal{E}, p{:}\alpha_1 \;\; e[x := p] \;\; \beta p}{\mathsf{checkExp}\;\; \mathcal{E} \;\; [x]e \;\; \alpha}\; p \text{ fresh in } \mathcal{E}$$

```
checkExp :: Exp -> Exp -> ChM ()
checkExp obj alfa =

    case obj  of

          ... ->

          EAbs x e  ->
              do
                whalfa <- whnf alfa
                case whalfa of
                 EPi alfa1 beta ->
                   do
                    (c,i) <- getCtxtParamChM
                    let p      = mkParam i x
                        e'     = substVar p x e
                        beta' = EApp beta p
                        c'     = addCtxt (i,alfa1) c
                    putCtxtChM c'
                    checkExp e' beta'
                 _ -> errorChM "checkExp, Function type expected"

          ... ->
```

FIGURE 5.10. Type checking abstractions

CHAPTER 6

Applications: Integral domains and Cayley's theorem

1. Introduction

In this chapter we shall comment on some experiments we have done concerning the formalization of abstract algebra using the proof checker described in the previous chapter.

In chapter 2 we gave a brief discussion on the process that led our work to considering the use of dependent record types as an appropriate mechanism for the representation of abstract constructions. The starting point was the work described in [**Bet93**], where we present a formalization of the notion of integral domain and the representation of the properties that can be derived from the postulates of such system of algebras. In addition, we also illustrate the possibility of formally establishing that a particular construction conforms a concrete algebra satisfying those postulates. All this was achieved using the notions of context and substitution and making use of the language of the logical framework which ALF implements. The incremental definition of systems of algebras, like "a group is a monoid with an inverse operation such that...", was naturally reflected by the extension of a context *Monoid* with new assumptions corresponding to the operation and the properties that such operation must satisfy. Thus, a proof of a derived property for monoids, formally a proof developed under the context *Monoid*, would naturally remain valid under the above extension of the context.

It was also remarked in chapter 2 that the "context approach" for the representation of algebraic constructions, however, soon revealed itself to have many drawbacks. Already when trying to represent simple higher order algebraic constructions, like the notion of morphism between algebras for instance, the formal counterpart to these notions in terms of contexts has many shortcomings.

We will show in section 2 the reformulation of the representation of the system integral domain in terms of record types. The incremental definitions now are directly accomplished by using record extension. We also provide a simple application of subtyping, namely, the reutilization of proofs developed groups and commutative rings when reasoning about integral domains. In section 3 we highlight the constructions we needed to develop for the formal representation of Cayley's theorem for group theory, which says that any abstract group is isomorphic to a group of permutations. The formal proof of this theorem per se is not a significant contribution. Nevertheless, it allows to illustrate the adequacy of the extended theory for building up a little more involved algebraic constructions, like isomorphims between groups, the construction of groups of transformation and permutations over a given space, and morphisms between (these) groups.

The complete code of the case studies presented here can be found at `http://www.cs.chalmers.se/~gustun/algebra/[IntDom,Cayley]`, respectively.

2. Integral domains

We shall reuse many of the definitions introduced in chapter 2. In principle the methodology that we shall follow to achieve the formal definition of the system of algebras Integral domain is the same as the one for Boolean algebras in that same chapter, namely, we start from the notion of *Setoid* and then successively enrich this structure with new components and axioms. This is also the approach followed in the formalization using contexts. However, using record types we shall be able of explicitly consider an algebraic system as formed out of an algebraic part, the carrier set, the equivalence relation, and the operation symbols and, on the other hand, the axioms that any such algebra must satisfy. Furthermore, we shall naturally maintain this structure when we perform the consecutive extensions.

In Figure 6.1 we show the definition of *Monoid*

$RelOpElem : type$
$RelOpElem = \langle RelOp, \boldsymbol{e_1} : \mathsf{A}\rangle$

$isUnitLeft : RelOpElem \to type$
$isUnitLeft\ Roe = \boldsymbol{use}\ \ Roe : RelOpElem\ \ \boldsymbol{in}\ \ (x : \mathsf{A}) \approx (\circ\ e_1\ x)\ x$

$isUnitRight : RelOpElem \to type$
$isUnitRight\ Roe = \boldsymbol{use}\ \ Roe : RelOpElem\ \ \boldsymbol{in}\ \ (x : \mathsf{A}) \approx (\circ\ x\ e_1)\ x$

$PreMonoid : type$
$PreMonoid = \langle Setoid, + : binOp\ \mathsf{S}, \mathbf{0} : \mathsf{S}\rangle$

$AxsOfMonoid : RelOpElem \to type$
$$AxsOfMonoid\ Roe = \begin{array}{l}\langle\ \mathsf{cong} : isCong\ Roe,\\ \quad \mathsf{assoc} : isAssoc\ Roe,\\ \quad \mathsf{unitL} : isUnitLeft\ Roe,\\ \quad \mathsf{unitR} : isUnitRight\ Roe\\ \rangle\end{array}$$

$Monoid : type$
$$\begin{array}{l}Monoid =\\ \quad \langle PreMonoid,\\ \quad\ \ \mathsf{props} : AxsOfMonoid\ \langle \mathsf{A} = \mathsf{S}, \mathsf{R} = \approx, \circ = +, \boldsymbol{e_1} = \mathbf{0}\rangle\\ \quad \rangle\end{array}$$

FIGURE 6.1. Monoid

The system *Monoid*, then, is defined as a record type, which is the result of extending the record type *PreMonoid* with a new field corresponding to the axioms that any monoid must satisfy. The type associated to the label props is obtained by applying the family of record types *AxsOfMonoid*, whose definition is parameterized by a set, a binary relation, a binary operation on the set and a distinguished element, to the appropriate record object.

We proceed by introducing in Figure 6.2 the definition of the system *Group*.

$\mathit{RelOpElUn} : \mathit{type}$
$\mathit{RelOpElUn} = \langle \mathit{RelOpEl}, - : \mathsf{A}\to\mathsf{A}\rangle$

$\mathit{isInvLeft} : \mathit{RelOpElUn}\to\mathit{type}$
$\mathit{isInvLeft}\ \mathit{Roeu} = \boldsymbol{use}\ \ \mathit{Roeu} : \mathit{RelOpElUn}\ \ \boldsymbol{in}\ \ (x : \mathsf{A}) \approx (\circ\ (-\ x\ )\ x)\ e_1$

$\mathit{isInvRight} : \mathit{RelOpElUn}\to\mathit{type}$
$\mathit{isInvRight}\ \mathit{Roeu} = \boldsymbol{use}\ \ \mathit{Roeu} : \mathit{RelOpElUn}\ \ \boldsymbol{in}\ \ (x : \mathsf{A}) \approx (\circ\ x\ (-\ x\ ))\ e_1$

$\mathit{AxsOfGroup} : \mathit{RelOpElUn}\to\mathit{type}$
$\mathit{AxsOfGroup}\ \mathit{Roeu} = \ \langle\ \mathit{AxsOfMonoid}\ \mathit{Roeu},$
$\qquad\qquad\qquad\qquad\qquad \mathsf{invL} : \mathit{isInvLeft}\ \mathit{Roeu},$
$\qquad\qquad\qquad\qquad\qquad \mathsf{invR} : \mathit{isInvRight}\ \mathit{Roeu}$
$\qquad\qquad\qquad\qquad\quad \rangle$
$\mathit{PreGroup} : \mathit{type}$
$\mathit{PreGroup} = \langle \mathit{PreMonoid}, \sim\ : \mathsf{S}\to\mathsf{S}\rangle$

$\mathit{Group} : \mathit{type}$
$\mathit{Group} =$
$\quad\langle \mathit{PreGroup},$
$\quad\ \ \mathsf{props} : \mathit{AxsOfGroup}\ \langle \mathsf{A} = \mathsf{S}, \mathsf{R} = \approx, \circ = +, \boldsymbol{e_1} = \mathbf{0}, - = \sim\rangle$
$\quad\rangle$

FIGURE 6.2. Group

Observe that the axioms of *Group* are defined as a family of record types which is obtained by extending the axioms of *Monoid* with two new fields corresponding to the axioms of the inverse operation of the group. The family *AxsOfGroup* is further parameterized with a unary operation. The application of the family *AxsOfMonoid* to the variable *Roeu* is correct because of the subtyping induced for families of types. Thus, we can understand the definition of *Group* as the respective extension of the algebraic and axiomatic part of the previously defined system *Monoid*.

We will not show here the whole sequence of definitions we made to obtain the one corresponding to the system integral domain. Instead we shall illustrate some features of the record approach that we consider interesting. After introducing the definitions of the systems *AbGroup* (for Abelian group) and *Ring* (with the multiplicative binary operation $\times$and its identity $\mathbf{1}$), we can then define the record type *CommRing* as shown in Figure 6.3

$\mathit{RelOpsElUns} : \mathit{type}$
$\mathit{RelOpsElUns} = \langle \mathit{RelOpElUn}, * : \mathit{binOp}\ \mathsf{A}, \boldsymbol{e_2} : \mathsf{A}\rangle$

$\mathit{AxsOfCRing} : \mathit{RelOpsElUns} \rightarrow \mathit{type}$
$\mathit{AxsOfCRing}\ \mathit{Roeus} =$
$\quad$ ***use*** $\mathit{Roeus} : \mathit{RelOpsElUns}$
$\quad$ ***in*** $\langle\ \mathit{AxsOfAbGroup}\ \mathit{Roeus},$
$\qquad \mathsf{multmon} : \mathit{AxsOfCommMonoid}\ \langle \mathsf{A} = A, \mathsf{R} = R, \circ = *, \boldsymbol{e_1} = e_2\rangle,$
$\qquad \mathsf{diffunits} : \mathit{Not}\ (R\ e_1 e_2),$
$\qquad \mathsf{distlft} : (x, y, z : \mathsf{A}) \approx (\times\ x\ (+\ y\ z))\ (+\ (\times\ x\ y)\ (\times\ x\ z))\ \rangle$

$\mathit{PreRing} : \mathit{type}$
$\mathit{PreRing} = \langle \mathit{PreGroup}, \times : \mathit{binOp}\ \mathsf{S}, \boldsymbol{e_2} : \mathsf{S}\rangle$

$\mathit{CommRing} : \mathit{type}$
$\mathit{CommRing} =$
$\quad \langle \mathit{PreRing},$
$\quad\ \mathsf{props} : \mathit{AxsOfCRing}\ \langle \mathsf{A} = \mathsf{S}, \mathsf{R} = \approx, \circ = +, \boldsymbol{e_1} = \mathbf{0}, - = \sim, * = \times, \boldsymbol{e_2} = \mathbf{1}\rangle$
$\quad \rangle$

FIGURE 6.3. Commutative ring

Thus, a *PreRing* forms a commutative ring if it is formed out of an additive Abelian group and a multiplicative commutative monoid such that the identities of both operations are different, and multiplication is distributive (to the left) with respect to the addition operator. Observe that if we had available an operation that allowed the concatenation of two record types to form a new one it might not be necessary to stratify the axioms for multiplication. We have been experimenting with these kind of record operations (both for types and objects) but no satisfactory formulation of them has yet been obtained. However, even if such a concatenation operation were available, notice that in this particular case we should still have to deal with duplicated labels.

Another example in the spirit of the function *dualPreLatt* presented in chapter 2 is the definition of the function *extractMonoid*, that extracts the multiplicative monoid out of a commutative ring. We show its definition in Figure 6.4

Observe, in the first place, that the variable *Cr* is used in the definiendum both as of type *CommRing* and as of type *PreMonoid*. The first requirement is clear from the form of the ***use*** expression. On the other hand, for the record object that is the result of the function to be an object of type *Monoid*, *Cr* must have type *PreMonoid*. This requires an application of subtyping. Moreover, that *Cr* is, as a component of

```
extractMonoid : CommRing→Monoid
extractMonoid Cr =  use Cr : CommRing
                    in ⟨Cr, props = props.multmon⟩
```

FIGURE 6.4. Extraction of the multiplicative monoid of a commutative ring

the record being constructed, considered as an object of the latter type will prevent access to the information proper of a commutative ring. One more application of subtyping is illustrated by the fact that the object *props*.multmon is accepted as a proof of the axioms of a monoid when it actually is one of a commutative monoid.

Finally, the definition of *IntDom* is given in Figure 6.5.

```
AxsOfIntDom : RelOpsElUns→type
AxsOfIntDom Roeus =
   use Roeus : RelOpsElUns
   in  ⟨AxsOfCRing Roeus
        mcancel : $(x, y, z : A)\ R\ (\times\ z\ x)\ (\times zy) \to Not\ (R\ z\ \boldsymbol{e_1}) \to R\ x\ y$
        ⟩

IntDom : type
IntDom =
  ⟨PreRing,
   props : AxsOfIntDom $\langle \mathsf{A} = \mathsf{S}, \mathsf{R} = \approx, \circ = +, \boldsymbol{e_1} = \mathbf{0}, - = \sim, * = \times, \boldsymbol{e_2} = \mathbf{1}\rangle$
  ⟩
```

FIGURE 6.5. Integral domain

2.1. Derived properties. We now show the proofs of three simple properties, namely, that the operation of a group is left cancellative, that the identity of the additive operation of a commutative ring is absorbent with respect to multiplication, and finally, that in an integral domain no element divides the identity of the additive operation. We intend to illustrate with these examples, on the one hand, the kind of expressions that are obtained. They do not look much different from programs expressed in a functional language like, say, Haskell. On the other hand, we shall see the reutilization we can make of proofs due to the use of subtyping.

To begin with, we show in Figure 6.6 a (skeleton of the) proof of the first of the properties above.

There is not much to say about this proof. The constant *eqtoeq* allows the replacement of equivalents objects in an equality proof. The ***use*** of G makes available all of its fields. The proof of *lemma* is trivial.

```
cancelL :  (G : Group) (x, y, z : G.S)
              use G : Group in ≈ (+ z x) (+ z y) → (≈ x y)
cancelL =
  [G x y z h]
   use G : Group
   in
    let
        lemma : (x, y : S) ≈ (+ (∼ x) (+ x y)) y
          = ...
     in
         eqtoeq G (+ (∼ z) (+ z x)) (+ (∼ z) (+ z y)) x y
                  (props.cong (∼ z) (+ z x) (∼ z) (+ z y) (refl (∼ z)) h)
                  (lemma z x)
                  (lemma z y)
```

Figure 6.6. Cancellation on the left

In Figure 6.7 we show the skeleton of the proof of the property on commutative rings mentioned above.

```
multAddUnit :  (Cr : CommRing) (x : Cr.S)
                  use Cr : CommRing in ≈ (× x e₁) e₁
multAddUnit =
  [Cr x]
   use Cr : CommRing
   in
    let
        cancelprop : ≈ (+ (× x x) (× x e₁)) (+ (× x x) e₁)
          = ...
     in
        cancelL Cr (× x e₁) e₁ (× x x) cancelprop;
```

Figure 6.7. Multiplication by identity of addition

Observe that we apply *cancelL* to the variable *Cr* of type *CommRing*.

We end up this section showing in Figure 6.8 the proof of the property of "non zero divisors", as it is usually called in the literature, for integral domains. The proof *multaddUnit* is used as one of integral domains.

3. Transformations and Cayley's theorem

In this section we will review the implementation we have done of the proof of Cayley's theorem presented in [**MB67**]. Actually, the original aim was to formalize

```
nonZeroDiv : (Id : IntDom) (x, y : Id.S)
             use Cr : IntDom in ≈ (× x y) e₁ → Not (≈ x e₁) → ≈ x e₁
nonZeroDiv =
  [Id x y h₁ h₂]
   use Id : IntDom
   in
        mcancel y e₁ x
                (trans (× x y) e₁ (× x e₁)
                       h₁
                       (symm (× x e₁) e₁ (multAddUnit Id x)))
                h₂
```

FIGURE 6.8. Non zero divisors

in type theory the whole chapter on basic group theory included in the book by the same authors [**BM53**]. This is one of the reasons why we use the term *transformation*, the old fashioned terminology used in this latter book to mean functions. The first reference above is a revised edition of this book, which uses the language of category theory to introduce most of the basic notions and provides new insights to modern algebra.

We managed to give a formal representation to the contents of sections 1 to 5 in the mentioned chapter, which ends up with a (very informal and succinct) proof of Cayley's theorem. In [**MB67**], however, this theorem is given a more detailed and clear proof, even though there are still constructions that remain informally treated. The idea there outlined is the one that we followed to build up the formal proof. We shall further comment on this in section 3.3.

Just for the sake of completeness we include here the definitions (as given in either of the books above) of some of the notions involved in the whole development. At the same time we shall also provide insights on how they were grasped and codified in the language of the type theory with which we are concerned.

3.1. Transformations.

DEFINITION (Transformation). A *transformation* $\phi : S \rightarrow T$ from a (non-empty) set S into a set T is a rule which assigns to each element $p \in S$ a unique image element $\phi(p)$.

The notion of a transformation of S into T is thus the same as that of a function defined on the elements of S, with values in T, and as that of a many-one correspondence of the elements of S to those of T. The set S is called the *domain* of ϕ, and T its *codomain*.

3.1.1. *Algebra of transformations.* Two transformations $\phi : S \rightarrow T$ and $\phi' : S \rightarrow T$ are called *equal* if they have the same effect upon every point of S; that is,

$$\phi = \phi' \text{ means that } \phi(p) = \phi'(p) \qquad \text{for every } p \in S \tag{1}$$

The *product* $\phi\psi$ of two transformations is defined as the result of performing them in succession.

The *identity* transformation I on the set S can be defined as that transformation $I : S \to S$ which leaves every point of S fixed.

PROPOSITION 6.1 (Associative law). *Whenever the products involved are defined multiplication of transformations conforms to the law* $(\phi\psi)\theta = \phi(\psi\theta)$.

PROOF. Follows straightforwardly by equality of transformations and definition of product of transformations. □

PROPOSITION 6.2 (Identity law). $I\phi = \phi I = \phi$ *for all* ϕ.

PROOF. Follows by definition of I and definition of product of transformations. □

THEOREM 6.1. *A transformation* $\phi : S \to S$ *is one-one if and only if it has a right-inverse; it is onto if and only if it has a left-inverse.*

DEFINITION (Group of transformations). By a *group of transformations* on a "space" S is meant any set G of one-one transformations ϕ of S onto S such that

i) the identity transformation of S is in G

ii) if ϕ is in G, so is its inverse

iii) if ϕ and ψ are in G, so is their product $\phi\psi$

THEOREM 6.2. *The set* G *of all one-one transformations of any space* S *onto itself is a group of transformations*

3.1.2. *Formalization.* We introduce the notions of a non-empty setoid (instead of sets we work with setoids) and equality of transformations on non-empty setoids as shown in Figure 6.9.

$$\begin{array}{ll}
\mathit{NESetoid} : \mathit{type} & \\
\mathit{NESetoid} = \langle \mathit{Setoid}, \boldsymbol{el} : \mathsf{S} \rangle & \\
& \\
\mathit{eqTS} : & (T, U : \mathit{NESetoid}) \\
& (\mathit{phi}, \mathit{xi} : (x : T.\mathsf{S})\ U.\mathsf{S})\ \mathit{Set} \\
\multicolumn{2}{l}{\mathit{eqTS}\ T\ U\ \mathit{phi}\ \mathit{xi} = (x : T.\mathsf{S})\ U.\!\approx (\mathit{phi}\ x)\ (\mathit{xi}\ x)}
\end{array}$$

FIGURE 6.9. Non empty setoids and equality of transformations

Then it is proved that *eqTS* is an equivalence relation on transformations.

The proof of Theorem 6.1 provided in the references makes use of the axiom of choice. This is needed for the construction of the right inverse of ϕ. In order to construct the proof of Theorem 6.2 we shall first define the notion of injective transformation of a space onto any other space. Then, we define the family of sets *Perms* as the set of bijections of a space into itself. This is shown in Figure 6.10.

```
Bijec : (X, Y : Setoid) Set
Bijec X Y =  Σfun ∈ (x : X.S)→Y.S.
             Σmap ∈ (x, y : X.S)→X.≈ x y→Y.≈ (fun x) (fun y).
              ((x, y : X.S)→Y.≈ (fun x) (fun y)→X.≈ x y)  ×
              ((y : Y.S)→Σx ∈ X.S.Y.≈ (fun x) y)

Perms : (X : Setoid) Set
Perms X = Bijec X X
```

FIGURE 6.10. Permutations on a setoid

Thus, *Perms X* is the set of all bijective functions with domain and codomain the carrier set of X. We use sigma sets to define this family because they are intended to constitute the carrier of the group of permutations, which, by definition of *NESetoid*, has to be an object of type *Set*.

Then, if the product (*prodP*) of objects of the set *Perms X* is defined as composition of functions, it is quite straightforward to prove that it satisfies the properties of the operation of a group, with identity *permId*. It is also quite direct to define a function *permInv*, that given any permutation p returns its inverse. Therefore, we can define the function *permGroup* over non empty setoids which is shown in Figure 6.11.

```
permGroup : (X : NESetoid) Group
permGroup X =
   ⟨setoidPerm X
   + = prodP X
   0 = permId X
   ∼ = permInv X
   props = ⟨ cong = congprodP X
             assoc = assocprodP X
             unitL = permIdunitL X
             unitR = permIdunitR X
             invL = permInvinvL X
             invR = permInvinvR X
             ⟩
   ⟩
```

FIGURE 6.11. Group of permutations on a setoid

3.2. Isomorphisms.

DEFINITION . By an isomorphism between two groups *G1* and *G2* is meant a one-one correspondence $a \leftrightarrow a'$ between their elements which preserves group multiplication – i.e., which is such that if $a \leftrightarrow a'$ and $b \leftrightarrow b'$, then $ab \leftrightarrow a'b'$.

The notion of isomorphism between any two groups is represented by a (record) type family indexed by two groups. This family has two fields, a bijection defined on the carrier of the groups (bijec), and the one expressing the morphism property to be satisfied by the function that constitutes the bijection (morphprop). This is shown in Figure 6.12

```
isoGroup : ⟨G1 : Group, G2 : Group⟩→type
isoGroup gs =
    use gs : ⟨G1 : Group, G2 : Group⟩
    in  ⟨ bijec : Bijec G1 G2
          morphprop : (x, y : G1.S)   G2.≈ (fst bijec (G1.+ x y))
                                           (G1.+ (fst bijec x) (fst bijec y))
        ⟩
```

FIGURE 6.12. Isomorphism of groups

The two following theorems were also proved.

THEOREM 6.3. *The relation "G1 is isomorphic to G2" is an equivalence relation between groups.*

Remark. It is worth observing that Theorem 6.3 and its proof hold equally for isomorphisms between integral domains, and indeed for isomorphisms between algebraic systems which are extension of the system *Group*.

THEOREM 6.4. *Under an isomorphism between two groups, the identity elements correspond and the inverses of corresponding elements correspond.*

3.3. Cayley's theorem. We now proceed to discuss the formal proof carried out for the following result by Cayley

THEOREM 6.5. *Any abstract group G is isomorphic with a group of permutations.*

The final presentation of this proof benefited from discussions with Gilles Barthe and Daniel Fridlender.

In [**BM53**] the theorem is enunciated and some ideas of the proof are laid down. There, it is suggested and justified what should be the morphism ϕ between the elements of a given group G and the elements of a group of transformations T: "define ϕ as a function that given an element a of the carrier set of G yields a function ϕ_a, such that $\phi_a(x) = ax$ for each $x \in G$". Thus, the function ϕ is a mapping from G to a transformation that operates on elements of G to return elements of G.

Therefore, in the first place, we have to construct the group T out of G, so we must define a sort of functor that for any given group yields the intended group of transformations, such that ϕ is an isomorphism between the carrier sets of G and T. Thus, the first question was to define what should be the carrier set of the group T. We chose to define it as the sigma set $\Sigma a \in G.\Sigma f \in G \to G.\forall x \in G.f(x) = ax$. The definition of ϕ is given in Figure 6.13

$$
\begin{aligned}
&\mathit{Phi} : (\mathit{Group})\ \mathit{Set}\\
&\mathit{Phi}\ G = \ \Sigma a \in G.\mathsf{S}.\\
&\qquad\quad \Sigma \mathit{phi} \in G \to G.\\
&\qquad\quad (x : G.\mathsf{S}) \to G.\!\approx (\mathit{phi}\ x)\ (G.\!+\ x\ a)
\end{aligned}
$$

FIGURE 6.13. Definition of the family ϕ

With this set at hand, thus, it is possible to define, in the first place, an equivalence relation over its elements in terms of the equality *eqTS* introduced in section 3.1.

The product of two transformations ϕ and ξ *prodTG*, can also be easily defined in terms of the product of transformations. It remains then to define the identity element of the set *Phi G* and the inverse of any element of this set, which is done as shown in Figure 6.14.

$$
\begin{aligned}
&\mathit{phiId} : (G : \mathit{Group})\ \mathit{Phi}\ G\\
&\mathit{phiId}\ G = \{G.0,\ \ [x]G.\!+\ x\ G.0,\ \ \ [x]G.\mathsf{ref}\ (G.\!+\ x\ G.0)\ \}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{phiInv} : (G : \mathit{Group})\ (\mathit{phi} : \mathit{Phi}\ G)\ \mathit{Phi}\ G\\
&\mathit{phiInv}\ G\ \mathit{phi} =\\
&\ \ \{G.\!\sim (\mathit{fst}\ \mathit{phi}),\ \ [x]G.\!+\ x\ (G.\!\sim (\mathit{fst}\ \mathit{phi})),\ \ [x]G.\mathsf{ref}\ (G.\!+\ x\ (G.\!\sim (\mathit{fst}\ \mathit{phi})))\ \}
\end{aligned}
$$

FIGURE 6.14. Identity and inverse of *Phi G*

The proofs corresponding to the postulates that say that given any group G, $\langle \mathit{Phi}\ G, \mathit{prodTG}, \mathit{phiId}, \mathit{phiInv} \rangle$ form a group are quite direct. Thus, we define a functor *transfGroup* in Figure 6.15, that given any group G returns the corresponding group of transformations described above.

In order to complete the proof, then, it has to be shown that given any group G:

i) *transfGroup G* is a group of permutations,

ii) it is possible to define an isomorphism between G and *transfGroup G*

This was done as follows: first we proved that it is possible to construct a monomorphism of groups between *transfGroup G* and *permsGroup G*. This is equivalent to say that the first is a subgroup (and therefore a group) of the latter. Then, we constructed the isomorphism between G and *transfGroup G*. The skeleton of the latter is shown in Figure 6.16.

Thus, for any group G, *fst* (*Cayley G*.bijec) is the isomorphism that can be constructed between G and its corresponding group of permutations.

```
transfGroup : (G : Group) Group
transfGroup G =
  ⟨setoidPhi G
   + = prodTG G
   0 = phiId G
   ∼ = phiInv G
   props = ⟨ cong = congprodTG G
             assoc = assocprodTG G
             unitL = phiIdunitL G
             unitR = phiIdunitR G
             invL = phiInvinvL G
             invR = phiInvinvR G
             ⟩
   ⟩
```

FIGURE 6.15. The group of transformations out of G

```
Cayley : (G : Group) isoGroup G (transfGroup G)
Cayley G =
  use G : Group
  in
   let
      transf : S→Phi G = [a]{a, [x]+ x a, [x]ref (+ x a)}

      ismaptransf : (x, y : S)→≈ x y→eqTG G (transf x) (transf y) = ...

      oneonetransf : (x, y : S)→eqTG G (transf x) (transf y)→≈ x y = ...

      ontotransf : (f : Phi G)→Σx ∈ S.eqTG G (transf x) f = ...

      isMorphtransf : (a, b : S)  eqTG G (transf (+ a b))
                                        (prodTG G (transf a) (transf b))

            = ...

   in
      ⟨ bijec = {transf, ismaptransf, {oneonetransf, ontotransf}},
        morphprop = isMorphtransf ⟩
```

FIGURE 6.16. Any group *G* is isomorphic to *transfGroup G*

CHAPTER 7

Related Work and Conclusions

Related work

Formal algebra in type theory. 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 achieve 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. Since these are set formers and the theory is predicative, one has that sets that are components of structures have to be restricted to be elements of a universe set, as indicated in chapter 2. 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 chapters 2 and 6, 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.

Abstraction and modularization. The explanation of the notions of abstract data type and module in terms of type-theoretic constructions has been extensively explored since the beginning of the last decade.

In [**MP85**] the authors present a functional language (SOL) which incorporates existential types of the form $\exists t.\sigma(t)$, where t is a type variable which may occur free in the type expression $\sigma(t)$. Values of such types are also introduced and are intended to model abstract data types. They are called *data algebras*. The intuitionistic explanation of the existential quantifier and what it means to be a proof of an existential proposition together with the Curry-Howard correspondence of propositions with types provide the conceptual basis for this understanding of abstract data types.

In [**Mac86**] MacQueen proposes the use of dependent function types and Σ-types to described the notion of structures, functors and signatures as provided by the language SML [**MTH90, MTH87**]. A language (DL) with a ramified type system in the spirit of the language of Martin-Löf's type theory is presented and examples concerned with the definition and use of modules are discussed. In [**Luo88**] Luo presents a higher order calculus (ΣCC). The language includes a Σ-type constructor together with the corresponding projection operations. In these two latter works the adequacy of Σ-types as a basic mechanism to express abstract structures is analysed and put forward as a safe methodology to deal with the manipulation of independently developed theories. Our understanding of systems of algebras (and abstract data types in general) as types of tuples has to a great extent been inspired by these two works.

The formulation of module mechanisms that, as a formal counterpart, make use either of existential types or Σ-types have been presented (and some of them implemented) in, among others, [**HMM90, Luo94, HL94, Ler94**].

On the other hand, closer to the proposal we make in this work of regarding modules as record types are the works by [**Apo93**] and [**Jon96**]. In the former, the author proposes an extension to the ML-records to express modules, but functors are only first order. The language analysed in the second of these works supports higher order functors, but no notion of subtyping is introduced. It is, nevertheless, acknowledged that it could be useful to have available one such notion.

In the context of variants of type theory, Coquand [**Coq96**] gives a type-theoretic formulation of the notion of *theory*, as provided by the *PVS* system. The combination of (parameterized) theories and Σ-types, both mechanisms implemented in *Alfa*, provides a powerful tool for the task of abstract and modular development of proofs. Coquand's ideas were adopted by Pollack and included, with some modifications, into the system *LEGO* [**Pol97**].

A module calculus for pure type systems is investigated in [**Cou97**]. Meta-theoretic results are discussed, but, as far as we know, no implementation of this calculus has been carried out.

Records and subtyping. The study of record types and subtyping have found a natural setting in object oriented programming. Starting with the pioneering work of Dahl and the team behind the language Simula [**DN**] these two notions have principally been exploited for providing foundations to the mentioned programming paradigm. Our work, however, is not intended to provide new insights and mechanisms for the theory of object oriented languages. It is primarily concerned with the use of dependent types for expressing specifications of abstract data types and modules in a general way. In principle we do not reject the idea of applying the formalism presented in this work to the study of objects, but it should be clear at this point that not even the ground notions of the paradigm, like that of *self* for instance, can be given a natural formulation in it.

Several extensions of the Hindley-Milner type system have been proposed to deal with records. A seminal work by Wand is [**Wan87**]. This work has been corrected and extended in [**Rem89**], [**Wan89**] and others. In these systems the only notion of polymorphism is generic polymorphism. But record types schemes are used that can be assigned to any record object in which certain labels are bound to objects of appropriate types, no matter whether or not other labels are present in the record object in question. This allows to express some of the (inclusion) polymorphism that we introduce using subtyping.

A variant of Girard's system F [**Gir72**] with record types and subtyping is given in [**CM91**] which extends earlier proposals in [**CW85, Car88**]. This system, which in the literature is referred to as $F_{\leq}$, also includes impredicative bounded quantification. One of the motivation for the latter is to assign a type to functions that update records in such a way that fields that are not mentioned in the function are preserved from input type to output type. This formalism has been shown by Pierce [**Pie94**] to have undecidable type checking.

In the language we have presented, record objects are extensible and thus update functions can be written. Extension is actually overriding, as in [**Wan87**], i.e. a record object may contain multiple occurrences of a label, the latest overriding the previous ones. An overriding operator is shown to be derivable from the basic record operations introduced in [**CM91**], which do not include this latter mechanism as a primitive one.

The type systems considered in the works cited above (most of them included as a chapter in [**GM94**]) do not embody types depending on individuals. As a consequence their record types are non dependent as opposed to the ones in our work.

Dependent record types have been implemented in *PVS* [**OSR**], which is a theorem proving system based on classical higher order logic. The subtyping that record types induce is, however, not a part of the implementation.

Type checking dependent types. The type checking algorithm we have presented in chapter 5 is much influenced by the one presented in [**Mag95**] for complete terms. This latter algorithm, in turn, makes use of ideas presented by Th. Coquand in [**Coq91**]. As already discussed in chapter 3, however, in addition to the fact that we also define the type checker to deal with record types and subtyping, our algorithm implements the formal verification of the judgement of a calculus that, to some extent, deviates from Martin-Löf's calculus of explicit substitutions. The calculus that we consider, instead, is a modified version of the one originally proposed by Tasistro and presented in [**Tas97, BT97**] which incorporates the notion of parameters to represent the notion of "free names". In that respect, we have situated ourselves closer to the spirit of the calculus presented by Coquand in the work we reference above. The work by McKinna and Pollack, presented in [**MP93, Pol94a**], concerning the type checking of PTS has also been quite influential in the development of our work.

In another direction, Coquand [**Coq96**] has recently proposed an algorithm for type checking dependent types that, to some extent, conceptually departs from the spirit of the ones above mentioned. The notion of the closure of an expression with the environment under which it has been introduced plays a principal role in the procedure that describes the checking of the typing judgements of a system of proof rules there introduced. Regarding this latter observation, the algorithm shares some of the principles used by Magnusson in the definition of her algorithm. However, a notion of generic value is introduced by Coquand that allows to cope with the checking of abstraction operators without the restrictions that have to be imposed for Magnusson's algorithm to work. The methodology used by Coquand, that relies on a model theoretic understanding of the type system, is shown in that same work to smoothly accommodate to provide explanation for extensions of the original system, like let expressions and the theory mechanisms mentioned in the previous section.

The problems posed by the type checking of languages with dependent types which incorporate mechanisms of subtyping have been studied in [**Car87**] and [**AC96**]. The latter work presents an extension to λP, an abstract version of the Edinburgh Logical Framework *LF*. A type checking algorithm for the extended system is there proposed and some meta-theoretic properties are shown to hold both for the calculus and the algorithm in question. The notion of subtyping introduced, however, applies only to (dependent) function types and constant type constructors. A more recent work is the one presented in [**JLS97**]. The study of a type checking algorithm for Luo's logical framework with coercive subtyping above mentioned is carried out in that work.

Conclusions and further work

We have presented investigations concerned with the understanding, implementation and use of an extension of Martin-Löf's logical framework with dependent record types and subtyping.

We have motivated the use of this system of proof rules 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.

Our main concern, however, was to design and implement an algorithm for the formal verification of the forms of judgement of the extended theory. We then had to face the problems inherent in the formal language when considering the process of type checking. There is no general algorithm for inferring the type of the (unlabeled) abstractions of the original framework. This restriction is transferred to the objects of the extension. Further, there arises an analogous situation with the type checking of record objects. The decision was taken then of restricting the forms of expression that constitute a valid input to the algorithm. We have shown, however, that the shortcomings resulting from that restriction seem to be harmless for the natural practice.

When considering the formal verification of the relative forms of judgement of the calculus there also appeared the well known problems posed by the manipulation of free names in the presence of dependent types. We adopted the technique of using parameters to stand for the notion of free variables of the various types. This decision led to a reformulation of the proof system. To have available this particular formulation of the calculus facilitated the task of reasoning about the correctness of the implemented algorithm.

The experiments reported in this work were all mechanically verified using the proof checker. This provided us with interesting feedbacks concerning the new mechanisms introduced. In particular, the incorporation of *use* expressions pursues, in the first place, to alleviate notation. We think, however, that the latter expression former combined with subtyping might provide a uniform mechanism for hiding implementations of abstract data types. This we consider merits to be further investigated.

Besides the case studies presented here, the system has been used to verify an abstract version of sorting by insertion [**Tas97**], which uses record types to express specifications of abstract data types. As a continuation of this latter work, the formal derivation of different implementations of insertion sort using the system has been reported in [**Gas98**].

No mechanism for the definition of inductive sets has been discussed in this work. We would like to formulate one that extends the relation of subtyping to consider inclusion between sets.

Bibliography

[AC96] D. Aspinall and A. Compagnoni. Subtyping dependent types. In *Proceedings of the 11th. IEEE Symposium on Logic in Computer Science*, 1996.

[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.

[Apo93] M. V. Aponte. Extending record typing to type parametric modules with sharing. In *Twentieth Annual ACM Symp. on Principles of Prog. Languages*, pages 465–478. ACM Press, 1993.

[Aug97] L. Augustsson. *HBC - The Chalmers Haskell Compiler.* Documentation report, available at `http://www.cs.chalmers.se/ augustss/hbc.html`, 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.

[Bet93] G. Betarte. A case-study in machine assisted proofs: The Integers form an Integral Domain, 1993. Licenciate thesis, Dpt. of Computer Sciences, Chalmers University of Technology and University of Göteborg.

[Bet97] G. Betarte. Dependent record types, subtyping and proof reutilization. In *online Proc. of the working group TYPES workshop Inheritance, subtyping and modular development of proofs*, Durham, England, September 1997.

[BM53] G. Birkhoff and S. MacLane. *A Survey of Modern Algebra.* Macmillan, 1953.

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

[BT97] G. Betarte and A. Tasistro. Extension of Martin-Löf's Type Theory with Record Types and Subtyping. To appear in *25 Years of Constructive Type Theory*, Oxford University Press, 1997.

[Bur75] W. H. Burge. *Recursive Programming Techniques.* Addison-Wesley Publishing Company, 1975.

[Car87] L. Cardelli. Typechecking dependent types and subtypes. In L.C. Aiello M. Boscarol and G. Levi, editors, *Proc. of the Workshop on Foundations of Logic and Functional programming*, number 306 in Lectures Notes in Computer Science. Springer, 1987.

[Car88] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76, 1988.

[CH88] Th. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76((2/3)), 1988.

[CM91] L. Cardelli and J. Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1, 1991.

[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.

[Coq96] Th. Coquand. An algorithm for type-checking dependent types. In *Science of Computer Programming 26*, pages 167–177, 1996.

[Cou97] J. Courant. A module calculus for pure type systems. In *Typed Lambda Calculi and Applications 97*, LNCS. Springer, 1997.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17(4), 1985.

[DN] O. Dahl and K. Nygaard. Simula, an algol-based simulation language. Comm. ACM 9, 671-678, 1966.

[Dow93] G. Dowek. The undecidability of typability in the lambda-pi-calculus. In *TLCA, LNCS 664, Bezem M., Groote J.F. (eds.)*, 1993.

[Gas98] V. Gaspes. Deriving instances of Abstract Insertion Sort in an implementation of Martin-Löf's type theory extended with dependent record types and subtyping. Talk given at *The Winter Meeting 1998*, Dept. of Computing Science, Chalmers University of Technology., 1998.

[Gir72] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures de lárithmétique dórdre superiéur.* PhD thesis, Université Paris VII, 1972.

[GM94] C.A. Gunter and J.C. Mitchell, editors. *Theoretical Aspects of Object Oriented Programming.* MIT, 1994.

[Gog94] H. Goguen. *A Typed Operational Semantics for Type Theory.* PhD thesis, University of Edinburgh, 1994.

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

[HL94] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Twenty First Annual Symp. on Principles on Prog. Languages.* ACM Press, 1994.

[HMM90] R. Harper, J. Mitchell, and E. Moggi. Higher-order modules and the phase distinction. In *Seventeenth Annual Symp. on Principles on Prog. Languages.* ACM Press, 1990.

[Hut92] G. Hutton. Higher-order functions for parsing. *Functional Programming*, 2:323–343, 1992.

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

[JLS97] A. Jones, Z. Luo, and S. Soloviev. Some algorithmic and proof-theoreticsl aspects of coercive subtyping. In E. Giménez and C. Paulin-Mohring, editors, *Proceedings of TYPES '96*, LNCS, 1997. To appear.

[Jon96] M. Jones. Using parameterized signatures to express modular structures. In *Twenty Third Annual Symp. on Principles on Prog. Languages.* ACM Press, 1996.

[Ler94] X. Leroy. Manifest types, modules, and separate compilation. In *Twenty First Annual Symp. on Principles on Prog. Languages.* ACM Press, 1994.

[Luo88] Z. Luo. A Higher-order Calculus and Theory of Abstractions. Technical report, LFCS, Department of Computer Science, University of Edinbu rgh, 1988.

[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.

[Mac86] D. MacQueen. Using Dependent Types to Express Modular Structures. In *Proceedings of the 13th POPL*, 1986.

[Mag95] L. Magnusson. The Implementation of ALF - a Proof Editor based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution, 1995. Ph.D. thesis. Programming Methodology Group, Dept. of Computing Science, University of Göteborg and Chalmers University of Technology.

[Mar84] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.

[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.

[MB67] S. MacLane and G. Birkhoff. *Algebra.* MacMillan, 1967.

[MP85] John Mitchell and Gordon Plotkin. Abstract types have existential type. In *Proc. of the 12th ACM Symposium on Principles of Programming Languages*, pages 37–51, New York, 1985.

[MP93] J. McKinna and R. Pollack. Pure type systems formalized. In M. Bezem and J.F.Groote, editors, *Proc. of the International Conference on Typed Lambda Calculi ans Applications*, number 664 in LNCS. Springer-Verlag, 1993.

[MTH87] R. Milner, M. Tofte, and R. Harper. A type discipline for program modules. In *TAPSOFT 87*, volume 250 of *LNCS*. Springer, 1987.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[Nor] B. Nordström. The typechecking algorithm. Document in preparation.

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

[OSR] S. Owre, N. Shankar, and J. M. Rushby. User guide for the PVS specification and verification system (Beta release). Comp. Sc. Laboratory, SRI International, 1993.

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

[Pey87] S. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[Pie94] B. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1), 1994.

[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.

[Pol97] R. Pollack. Theories in type theory. In *Online Proc. of the TYPES working group workshop Subtyping, inheritance and modular development of proofs*, Durham, England, Sep. 1997.

[Rem89] D. Remy. Typechecking records and variants in a natural extension of ml. In *Conf. Rec. of the 16th. Ann. ACM. Symp. on Principles of Programming Languages*, 1989.

[Röj95] N. Röjemo. *Garbage collection, and memory efficiency, in lazy functional languages.* PhD thesis, Dept. of Computing Science, University of Göteborg and Chalmers University of Technologyf, 1995.

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

[Sev96] P. Severi. *Normalisation in Lambda Calculus and its relation to type inference.* PhD thesis, Eindhoven University of Technology, 1996.

[Tas93a] A. Tasistro. Extension of Martin-Löf's Theory of Types with Record Types and Subtyping, 1993. Privately circulated notes.

[Tas93b] 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, 1997. Ph.D. thesis. Programming Methodology Group, Dept. of Computing Science, University of Göteborg and Chalmers University of Technology.

[Wad92] P. Wadler. The essence of functional programming. In *1992 Symposium on principles of Programming Languages*, pages 1–14, 1992.

[Wan87] M. Wand. Complete type inference for simple objects. In *2nd. Symposium on Logic in Computer Science*, 1987.

[Wan89] M. Wand. Type inference for record concatenation and multiple inheritance. In *4th. Symposium on Logic in Computer Science*, 1989.

Appendices

A - Category of expressions, substitutions and properties

The category of expressions. The expressions of the language are given by the grammar in Figure A.1

$$\begin{array}{lll} e & ::= & x \mid p \mid c \mid [x]e \mid e_1e_2 \mid \langle\rangle \mid \langle e_1, L = e_2\rangle \mid e.L \\ & & e_1{\rightarrow}e_2 \mid \langle e_1, L{:}e_2\rangle \end{array}$$

FIGURE A.1. Syntax of expressions

Instantiation and Substitution. We show in Figure A.3 and Figure A.2 the definition of the functions that perform the substitution of a expression for a variable and the instantiation of a parameter by an expression respectively.

$x[e_1/p]$	$=_{def}$	x	
$q[e_1/p]$	$=_{def}$	e_1	if $p = q$
	$=_{def}$	q	if $p \neq q$
$c[e_1/p]$	$=_{def}$	c	
$([x]e_2)[e_1/p]$	$=_{def}$	$[x]e_2[e_1/p]$	
$f e_2[e_1/p]$	$=_{def}$	$(f[e_1/p])(e_2[e_1/p])$	
$(\alpha{\rightarrow}\beta)[e_1/p]$	$=_{def}$	$(\alpha[e_1/p]){\rightarrow}\beta[e_1/p]$	
$\langle\rangle[e_1/p]$	$=_{def}$	$\langle\rangle$	
$\langle e, L = e'\rangle[e_1/p]$	$=_{def}$	$\langle e[e_1/p], L = e'[e_1/p]\rangle$	
$e.L[e_1/p]$	$=_{def}$	$(e[e_1/p]).L$	
$\langle e, L{:}e'\rangle[e_1/p]$	$=_{def}$	$\langle e[e_1/p], L{:}e'[e_1/p]\rangle$	

FIGURE A.2. Instantiation

$y[x := e_1]$	$=_{def}$	e_1	if $x = y$
	$=_{def}$	y	if $x \neq y$
$p[x := e_1]$	$=_{def}$	p	
$c[x := e_1]$	$=_{def}$	c	
$([y]e_2)[x := e_1]$	$=_{def}$	$[y]e_2$	if $x = y$
	$=_{def}$	$[y]e_2[x := e_1]$	if $x \neq y$
$f e_2[x := e_1]$	$=_{def}$	$(f[x := e_1])(e_2[x := e_1])$	
$(\alpha{\rightarrow}\beta)[x := e_1]$	$=_{def}$	$(\alpha[x := e_1]){\rightarrow}\beta[x := e_1]$	
$\langle\rangle[x := e_1]$	$=_{def}$	$\langle\rangle$	
$\langle e, L = e'\rangle[x := e_1]$	$=_{def}$	$\langle e[x := e_1], L = e'[x := e_1]\rangle$	
$e.L[x := e_1]$	$=_{def}$	$(e[x := e_1]).L$	
$\langle e, L{:}e'\rangle[x := e_1]$	$=_{def}$	$\langle e[x := e_1], L{:}e'[x := e_1]\rangle$	

FIGURE A.3. Substitution

$$\text{(wf-Par):}\ \frac{}{\mathit{wf}\ p} \qquad \text{(wf-Con):}\ \frac{}{\mathit{wf}\ c}$$

$$\text{(wf-Lda):}\ \frac{\mathit{wf}\ e[x := p]}{\mathit{wf}\ [x]e} \qquad \text{(wf-App):}\ \frac{\mathit{wf}\ f \quad \mathit{wf}\ e}{\mathit{wf}\ fe}$$

$$\text{(wf-ERec):}\ \frac{}{\mathit{wf}\ \langle\rangle} \qquad \text{(wf-RecO):}\ \frac{\mathit{wf}\ e \quad \mathit{wf}\ e'}{\mathit{wf}\ \langle e, L = e'\rangle} \qquad \text{(wf-Sel):}\ \frac{\mathit{wf}\ e}{\mathit{wf}\ e.L}$$

$$\text{(wf-Fun):}\ \frac{\mathit{wf}\ \alpha \quad \mathit{wf}\ \beta}{\mathit{wf}\ \alpha{\to}\beta} \qquad \text{(wf-RecT):}\ \frac{\mathit{wf}\ e \quad \mathit{wf}\ e'}{\mathit{wf}\ \langle e, L{:}e'\rangle}$$

FIGURE A.4. Well-formed expressions

$$\begin{aligned}
\mathit{lgth}\ p &= 1\\
\mathit{lgth}\ x &= 1\\
\mathit{lgth}\ c &= 1\\
\mathit{lgth}\ fe &= \mathit{lgth}\ f + \mathit{lgth}\ e\\
\mathit{lgth}\ [x]e &= 1 + \mathit{lgth}\ e\\
\mathit{lgth}\ \alpha{\to}\beta &= \mathit{lgth}\ \alpha + \mathit{lgth}\ \beta\\
\mathit{lgth}\ \langle\rangle &= 1\\
\mathit{lgth}\ \langle e_1, L = e_2\rangle &= \mathit{lgth}\ e_1 + \mathit{lgth}\ e_2\\
\mathit{lgth}\ e.L &= 1 + \mathit{lgth}\ e\\
\mathit{lgth}\ \langle e_1, L{:}e_2\rangle &= \mathit{lgth}\ e_1 + \mathit{lgth}\ e_2
\end{aligned}$$

FIGURE A.5. Length of an expression

Properties of well-formed expressions. The notion of being a well-formed expression is inductively defined as shown in Figure A.4. Some of the properties below are proved by complete induction on the length of expressions. This function, in turn, is defined in Figure A.5.

PROPOSITION A.1. *Given expressions e and e_1, a parameter p, variables x and y, if $x \neq y$ and $e[y := p][x := e_1] = e[y := p]$ then $e[x := e_1] = e$.*

PROOF. This proposition can be proved by complete induction on the length of e. The interesting case is when e is an abstraction.

□

PROPOSITION A.2. *Given expressions e_1 and e_2, such that $\mathit{wf}\ e_2$, and any variable x, then $e_2[x := e_1] = e_2$.*

PROOF. This proposition is also proved by complete induction, in this case on the length of the expression e_2. The interesting case is when e_2 is an abstraction.

lgth $e_2 = 1$ $\triangleright$ If e_2 is either a parameter, a constant or a sort the proof is direct by def. of substitution. As to a variable y, notice that to assume that *wf* y leads to contradiction.

$e_2 = [y]e$ $\triangleright$
Assume that for all expression e', such that *lgth* $e' <$ *lgth* $[y]e$, if *wf* e' then $e'[x := e_1] = e'$, and also that *wf* $[y]e$. We have to prove that $([y]e)[x := e_1] = [y]e$. We now proceed by cases on the equality of x and y:

$x = y$ $\triangleright$ By definition the substitution has no effect on $[y]e$, thereby the equality holds.

$x \neq y$ $\triangleright$ The goal becomes now to prove $[y]e[x := e_1] = [y]e$. Notice that the assumption that $[y]e$ is well-formed allows ourselves to assume that *wf* $e[y := p]$ for any parameter p. Furthermore, the substitution of p for y does not change the length of e, which is less that the one of $[y]e$. Thus, we can apply induction with $e[y := p]$ to obtain that $e[y := p][x := e_1] = e[y := p]$. Now, by Proposition A.1, we get that $e[x := e_1] = e$, and then so are $[y]e[x := e_1]$ and $[y]e$.

The rest of the cases follow by definition of substitution and induction.

□

REMARK . The intuition behind this proposition is that well-formed expressions are not affected by substitution.

Using the proposition above is then quite direct to prove the following

PROPOSITION A.3. *Given expressions e,e_1 and e_2, variables x and y, if $x \neq y$, wf e_1 and wf e_2 then* $e[y := e_2][x := e_1] = e[x := e_1][y := e_2]$

PROOF. Surprisingly enough, at least to us, the proof can be done by structural induction on the expression e. We show the cases where e is either a variable or an abstraction.

$e = z$ $\triangleright$ First we perform case analysis on $z = y$:

$z = y$ $\triangleright$ $y[y := e_2][x := e_1] = e_2[x := e_1]$ and $y[x := e_1][y := e_2] = e_2$. Now, as e_2 is a well-formed expression, we can apply Proposition A.2 to get that $e_2[x := e_1] = e_2$. Transitivity and symmetry of the equality on expressions do the rest.

$z \neq y$ $\triangleright$ Now we make case analisys on $z = x$:

$z = x$ $\triangleright$ We repeat the former argument but now we use that e_1 is well-formed in order to apply Proposition A.2.

$z \neq x$ $\triangleright$ Both expressions reduce to z.

$e = [z]f$ $\triangleright$ First we perform case analysis on $z = y$

$z = y$ $\triangleright$ Both expressions reduce to $[y]f[x := e_1]$

$y \neq z \triangleright$ Now we make case analysis on $x = z$

$x = z$ $\triangleright$ Both expressions reduce to $[x]f[y := e_2]$

$x \neq z$ $\triangleright$ By definition of substitution the expressions $([z]f)[y := e_2][x := e_1]$ and $([z]f)[x := e_1][y := e_2]$ are equal to the expressions $[z]f[y := e_2][x := e_1]$ and $[z]f[x := e_1][y := e_2]$ respectively. Thus it suffices to prove that $f[y := e_2][x := e_1]$ and $f[x := e_1][y := e_2]$ are equal expressions, which we get from the induction hypothesis on f.

□

PROPOSITION A.4. *Given an expression e, a parameter p and variable x, then for all parameter q if wf $(e[x := p])$ it also holds that wf $(e[x := q])$.*

PROOF. The proof of this proposition is by complete induction on the length of the expression e. □

Now we can prove a very useful property

PROPOSITION A.5. *Given expressions e_1 and e_2, a variable x and parameter p if wf $(e_1[x := p])$ and wf e_2 then wf $(e_1[x := e_2])$.*

PROOF. The proof proceeds by complete induction on the length of the expression e_1. Thus we will have to prove that for all expression e, if for all expression e' such that *lgth* $e' <$ *lgth* e, *wf* $e'[x := p]$ and *wf* e_2 implies *wf* $e'[x := e_2]$ then *wf* $e[x := p]$ and *wf* e_2 implies *wf* $e[x := e_2]$. We show the proof for the cases p, x, fe', $[y]e'$, the cases when e is a constant or a sort are identical to p, the rest to the application case.

$e = p$ $\triangleright$ Follows directly by definition of substitution and well-formedness.

$e = y$ $\triangleright$ We proceed by case analysis on $y = x$

$y = x$ $\triangleright$ By def. of substitution $x[x := e_2] = e_2$, then the goal follows by the assumption that *wf* e_2

$x \neq y$ $\triangleright$ Notice that $y[x := p] = y$, then to assume that it is *wf* $y[x := p]$ leads to contradiction, due to definition of well-formedness.

$e = fe'$ $\triangleright$ Assume that it is *wf* $fe'[x := p]$ and *wf* e_2. By definition of substitution and *wf* fe', we are allowed to assume that both *wf* $f[x := p]$ and *wf* $e'[x := p]$. Clearly *lgth* $f <$ *lgth* fe' and *lgth* $e' <$ *lgth* fe'. We can now use the ind. hyp. to obtain that *wf* $f[x := e_2]$ and *wf* $e'[x := e_2]$. Applying rule *(wf-App)* in Figure A.4 we get *wf* $f[x := e_2]e'[x := e_2]$.

$e = [y]f$ $\triangleright$ Assume that it is *wf* $(([y]f)[x := p])$ and *wf* e_2. Now we proceed by cases on $y = x$

$y = x$ $\triangleright$. The goal is to prove that the expression $([y]f)[x := e_2]$ is well-formed. Now, by definition of substitution this latter expression is equal to $[y]f$. The first assumption gives that *wf* $[y]f$ (the substitution has no effect).

$x \neq y$ $\triangleright$ The first assumption gives, by def. of well-formedness, that *wf* $f[x := p][y := q]$ for any parameter q. Now, the goal to prove in this case is *wf* $[y]f[x := e_2]$ (after performing the substitution). Thus, it suffices to prove that *wf* $f[x := e_2][y := r]$ for any parameter r, and then apply rule *(wf-Lda)* in Figure A.4. If we now assume r we get

- *wf* $f[x := p][y := r]$, instantiating the parameter q in the proposition above with r
- *wf* $f[y := r][x := p]$, by Proposition A.3, *wf* r and *wf* p

and we have that *lgth* $f[y := r] <$ *lgth* $[y]f$. Thus, by induction we get that *wf* $f[y := r][x := e_2]$. Finally, as e_2 is also well-formed , we can use again Proposition A.3 to get *wf* $f[x := e_2][y := r]$.

$\square$

Closed expressions. In the following we will talk of *closed* expressions. As anticipated, the valid open expressions that participate in a relative judgement will depend on parameters not on variables. Therefore, we shall need a notion of closed expression that says more than the one traditionally used in languages with binding operators. For doing that, we first introduce the notion of *independence* of an expression e of a parameter p. The inductive definition of this predicate on expressions is given in Figure A.6.

We can now formulate an important property of expressions:

PROPOSITION A.6. *Given expressions e_1 and e_2, let p be a parameter such that e_1 indep p, then $e_1[e_2/p] = e_1$.*

PROOF. The proof of this lemma is by structural induction on the proof of e_1 *indep* p. We show only the cases where e_1 is either a parameter or an abstraction.

q *indep* p $\triangleright$ Thus, $p \neq q$ and then by definition of instantiation $q[e_2/p] = q$.

$[x]e$ *indep* p $\triangleright$ By definition of substitution we have $([x]e)[e_2/p] = [x]e[e_2/p]$. We know that e *indep* p, thus by induction hypothesis we get $e[e_2/p] = e$. Then $[x]e[e_2/p] = [x]e$.

$\square$

PROPOSITION A.7. *Given expressions e, e_1 and e_2, a parameter p and a variable x, if e indep p then $e[x := e_1][e_2/p] = e[e_2/p][x := e_1[e_2/p]]$.*

PROOF. The proof of this lemma is by structural induction on the expression e and nested structural induction on e *indep* p. $\square$

As a corollary of the two propositions above we obtain the following:

$$\text{(indep-Par): } \frac{q \neq p}{q \ \textit{indep } p} \qquad \text{(indep-Var): } \frac{}{x \ \textit{indep } p}$$

$$\text{(indep-Con): } \frac{}{c \ \textit{indep } p}$$

$$\text{(indep-Lda): } \frac{e \ \textit{indep } p}{[x]e \ \textit{indep } p} \qquad \text{(indep-App): } \frac{f \ \textit{indep } p \quad e \ \textit{indep } p}{fe \ \textit{indep } p}$$

$$\text{(indep-ERec): } \frac{}{\langle\rangle \ \textit{indep } p} \qquad \text{(indep-RecO): } \frac{e \ \textit{indep } p \quad e' \ \textit{indep } p}{\langle e, L = e' \rangle \ \textit{indep } p}$$

$$\text{(indep-Sel): } \frac{e \ \textit{indep } p}{e.L \ \textit{indep } p}$$

$$\text{(indep-Fun): } \frac{\alpha \ \textit{indep } p \quad \beta \ \textit{indep } p}{\alpha \to \beta \ \textit{indep } p} \qquad \text{(indep-RecT): } \frac{e \ \textit{indep } p \quad e' \ \textit{indep } p}{\langle e, L{:}e' \rangle \ \textit{indep } p}$$

Figure A.6. Independence

Proposition A.8. *Given expressions e_1 and e_2, a parameter p and a variable x, if e_1 indep p then $e_1[x := p][e_2/p] = e_1[x := e_2]$.*

Proof. By Proposition A.7 we know $e_1[x := p][e_2/p] = e_1[e_2/p][x := p[e_2/p]]$. Now, as e_1 *indep* p Proposition A.6 says that $e_1[e_2/p] = e_1$. Finally, the expression $p[e_2/p]$ is equal to e_1.

□

Definition 1.1. [Closed expression] An expression e is closed if and only if e is well-formed and for all parameter p the expression e is also independent of p .

Proposition A.9. *Given expressions e_1 and e_2 variable x and parameter p. If e_1 is a closed expression then $e_1[x := e_2] = e_1$ and $e_1[e_2/p] = e_1$.*

Proof. This proposition is a corollary of both Proposition A.2 and Proposition A.6. By definition e_1 is both well-formed and independent of p.

□

Proposition A.10. *Let e_1 and e_2 be any two expressions and p and p_1 be two parameters such that $p \neq p_1$.*
If a_1 is closed then $e_1[e_2/p][a_1/p_1] = e_1[a_1/p_1][e_2[a_1/p_1]/p]$.

Proof. This can easily be proved by structural induction on expression e_1. We show here the cases that e_1 is either a parameter or an abstraction.

$e_1 = q$ $\triangleright$ We perform case analysis on the equality of q and p:

$q = p$ ▷ Both expressions reduce to $e_2[a_1/p_1]$

$q \neq p$ ▷ We perform case analysis on the equality of q and p_1:

$q = p_1$ ▷ The first expression reduces directly to the expression a_1 by definition of instantiation. The right-hand side expressions, on the other hand, reduces to $a_1[e_2[a_1/p_1]/p]$. However, by hypothesis we know that a_1 is closed, then the latter instantiation has no effect on a_1.

$q \neq p_1$ ▷ Both expressions are equal to the parameter q.

$e_1 = [x]e$ ▷ By definition of instantiation we have that the lhs expression reduces to $[x]e[e_2/p][a_1/p_1]$. Moreover, we also know that the rhs expression is equal to $[x]e[a_1/p_1][e_2[a_1/p_1]/p]$. We can then apply induction to show that these expressions are equal.

□

The above proposition can be generalized as follows

PROPOSITION A.11. *Given expressions e and a, and closed expressions a_i with $i = 1..n$, let p,$p_1, \ldots, p_n$ be $n + 1$ mutually distinct parameters.*
Then $e[a/p][a_1/p_1, \ldots, a_n/p_n] = e[a_1/p_1, \ldots, a_n/p_n][a[a_1/p_1, \ldots, a_n/p_n]/p]$.

PROOF. By induction on n and using Proposition A.10. □

PROPOSITION A.12. *Given expressions e_1,e_2, a variable x and a parameter p_1. If a_1 is a closed expression then $e_1[x := e_2][a_1/p_1] = e_1[a_1/p_1][x := e_2[a_1/p_1]]$*

PROOF. By structural induction on expression e_1. We here show the proof for the cases where e_1 is either a variable, a parameter or an abstraction.

$e_1 = y$ ▷ We make case analysis on the equality of x and y:

$y = x$ ▷ Both the rhs and the lhs are equal to the expression $e_2[a_1/p_1]$.

$y \neq x$ ▷ Both the rhs and the lhs are equal to the variable y.

$e_1 = q$ ▷ We make case analysis on the equality of parameters q and p_1:

$q = p_1$ ▷ The lhs is equal to a_1 by definition of substitution and instantiation. By definition of instantiation the rhs is equal to $a_1[x := e_2[a_1/p_1]]$. Now, as the expression a_1 is closed by hypothesis, this latter substitution has no effect.

$e_1 = [y]e$ ▷ We make case analysis on the equality of x and y:

$y = x$ ▷ Both the lhs and the rhs are equal to the expression $[x]e[a_1/p_1]$.

$y \neq x$ ▷ The lhs reduces to $[y]e[x := e_2][a_1/p_1]$ and the rhs to the expression $[y]e[a_1/p_1][x := e_2[a_1/p_1]]$. We can then apply the induction hypothesis to show that these two expressions are equal.

□

The above proposition can be generalized as follows

PROPOSITION A.13. *Given expressions e_1,e_2, a variable x, closed expressions a_i and mutually distinct parameters p_i, with $i = 1..n$. Then, the expressions $e_1[x := e_2][a_1/p_1, \ldots , a_n/p_n]$ and $e_1[a_1/p_1, \ldots , a_n/p_n][x := e_2[a_1/p_1, \ldots , a_n/p_n]]$ are equal.*

PROOF. By induction on n and using Proposition A.12.

□

As a corollary of the former proposition we can obtain the following:

PROPOSITION A.14. *Given expression e_1 a variable x, closed expressions a_i and parameters $p_i$with $i = 1..n$. If p is a parameter distinct from p_i for $i = 1..n$ then $e_1[x := p][a_1/p_1, \ldots , a_n/p_n] = e_1[a_1/p_1, \ldots , a_n/p_n][x := p]$.*

PROOF. By Proposition A.13 we get that $e_1[x := e_2][a_1/p_1, \ldots , a_n/p_n]$ is equal to $e_1[a_1/p_1, \ldots , a_n/p_n][x := p[a_1/p_1, \ldots , a_n/p_n]]$. Now, as p is different from all the parameters p_i the instantiations $p[a_1/p_1, \ldots , a_n/p_n]$ have no effect on p. □

B - The calculus

General rules. In Figure B.1 we present the rules for formation of contexts, the (schematic) rule of thinning and the rules of assumption and type formation.

context formation, thinning and assumption:

$$\frac{}{[]\ \mathit{context}} \qquad \frac{\Gamma\ \mathit{context} \quad \Gamma \vdash \alpha : \mathit{type}}{\Gamma, p{:}\alpha\ \mathit{context}}\ {\scriptstyle p \text{ fresh in } \Gamma}$$

$$\frac{\Gamma \vdash \mathrm{J}}{\Delta \vdash \mathrm{J}}\ {\scriptstyle \Gamma \preceq \Delta}$$

$$\frac{}{\Gamma \vdash p : \alpha}\ {\scriptstyle p : \alpha \text{ in } \Gamma} \qquad \frac{}{\Gamma \vdash \alpha : \mathit{type}}\ {\scriptstyle p : \alpha \text{ in } \Gamma}$$

Figure B.1. Context formation, thinning and assumption

Equality rules. These are the general rules for the equality relation of types, objects of a type and families of types. Their justification is done as in previous formulations of type theory. We have then *reflexivity, symmetry and transitivity* rules for identity of types, objects of types and families of types under a given context.

inclusion from identity:

$$\frac{\Gamma \vdash \alpha_1 = \alpha_2 : \mathit{type}}{\Gamma \vdash \alpha_1 \sqsubseteq \alpha_2} \qquad \frac{\Gamma \vdash \beta_1 = \beta_2 : \alpha \to \mathit{type}}{\Gamma \vdash \beta_1 \sqsubseteq \beta_2 : \alpha \to \mathit{type}}$$

subsumption:

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

$$\frac{\Gamma \vdash \alpha_1 \sqsubseteq \alpha_2 \quad \Gamma \vdash \beta : \alpha_2 \to \mathit{type}}{\Gamma \vdash \beta : \alpha_1 \to \mathit{type}}$$

$$\frac{\Gamma \vdash \alpha_1 \sqsubseteq \alpha_2 \quad \Gamma \vdash \beta_1 = \beta_2 : \alpha_2 \to \mathit{type}}{\Gamma \vdash \beta_1 = \beta_2 : \alpha_1 \to \mathit{type}} \qquad \frac{\Gamma \vdash \alpha_1 \sqsubseteq \alpha_2 \quad \Gamma \vdash \beta_1 \sqsubseteq \beta_2 : \alpha_2 \to \mathit{type}}{\Gamma \vdash \beta_1 \sqsubseteq \beta_2 : \alpha_1 \to \mathit{type}}$$

Figure B.2. Rules of subsumption and inclusion of families of types

Rules of inclusion. We have also rules for expressing that the inclusion of two types and two families of types follows from their identity, as well as the *reflexivity* and *transitivity* of inclusion of types. The rules in Figure B.2 are immediately justified from the meaning explanation of the judgement of inclusion.

Rules of instantiation. The various rules of instantiation are presented in Figure B.3

instantiation of types:

$$\frac{\Gamma, p{:}\alpha \vdash \alpha_1 : \mathit{type} \quad \Gamma \vdash a : \alpha}{\Gamma \vdash \alpha_1[a/p] : \mathit{type}} \qquad \frac{\Gamma, p{:}\alpha \vdash \alpha_1 = \alpha_2 : \mathit{type} \quad \Gamma \vdash a = b : \alpha}{\Gamma \vdash \alpha_1[a/p] = \alpha_2[b/p] : \mathit{type}}$$

$$\frac{\Gamma, p{:}\alpha \vdash \alpha_1 \sqsubseteq \alpha_2 \quad \Gamma \vdash a = b : \alpha}{\Gamma \vdash \alpha_1[a/p] \sqsubseteq \alpha_2[b/p]}$$

instantiation of objects:

$$\frac{\Gamma, p{:}\alpha \vdash b : \alpha_1 \quad \Gamma \vdash a : \alpha}{\Gamma \vdash b[a/p] : \alpha_1[a/p]} \qquad \frac{\Gamma, p{:}\alpha \vdash b_1 = b_2 : \alpha_1 \quad \Gamma \vdash a = c : \alpha}{\Gamma \vdash b_1[a/p] = b_2[c/p] : \alpha_1[a/p]}$$

instantiation of families of types:

$$\frac{\Gamma, p{:}\alpha \vdash \beta : \alpha_1 \to \mathit{type} \quad \Gamma \vdash a : \alpha}{\Gamma \vdash \beta[a/p] : \alpha_1[a/p] \to \mathit{type}} \qquad \frac{\Gamma, p{:}\alpha \vdash \beta_1 = \beta_2 : \alpha_1 \to \mathit{type} \quad \Gamma \vdash a = b : \alpha}{\Gamma \vdash \beta_1[a/p] = \beta_2[b/p] : \alpha_1[a/p] \to \mathit{type}}$$

$$\frac{\Gamma, p{:}\alpha \vdash \beta_1 \sqsubseteq \beta_2 : \alpha_1 \to \mathit{type} \quad \Gamma \vdash a = b : \alpha}{\Gamma \vdash \beta_1[a/p] \sqsubseteq \beta_2[b/p] : \alpha_1[a/p] \to \mathit{type}}$$

instantiation of record types and record families:

$$\frac{\Gamma, p{:}\alpha \vdash \rho : \mathit{record\text{-}type} \quad \Gamma \vdash a : \alpha}{\Gamma \vdash \rho[a/p] : \mathit{record\text{-}type}} \qquad \frac{\Gamma, p{:}\alpha \vdash \sigma : \alpha_1 \to \mathit{record\text{-}type} \quad \Gamma \vdash a : \alpha}{\Gamma \vdash \sigma[a/p] : \alpha_1[a/p] \to \mathit{record\text{-}type}}$$

FIGURE B.3. Rules of instantiation

Rules for families of types and types. The rules of application for families of types and the associated equality and inclusion rules are shown in Figure B.4.

Then, in Figure B.5 we show the rule for formation of families, the corresponding β-rule and the various rules of equality.

Sets and elements of sets. The rules in Figure B.6 introduce the type of (inductively) defined set, the rule saying that any set gives rise to a type and the associated equality rule.

application:

$$\frac{\Gamma \vdash \beta : \alpha \to type \quad \Gamma \vdash a : \alpha}{\Gamma \vdash \beta a : type} \qquad \frac{\Gamma \vdash \beta : \alpha \to type \quad \Gamma \vdash a = b : \alpha}{\Gamma \vdash \beta a = \beta b : type}$$

equality and inclusion:

$$\frac{\Gamma \vdash \beta_1 = \beta_2 : \alpha \to type \quad \Gamma \vdash a : \alpha}{\Gamma \vdash \beta_1 a = \beta_2 a : type} \qquad \frac{\Gamma \vdash \beta_1 \sqsubseteq \beta_2 : \alpha \to type \quad \Gamma \vdash a : \alpha}{\Gamma \vdash \beta_1 a \sqsubseteq \beta_2 a}$$

FIGURE B.4. Family application, equality and inclusion

abstraction and β:

$$\frac{\Gamma, p{:}\alpha \vdash \alpha_1[x := p] : type}{\Gamma \vdash [x]\alpha_1 : \alpha \to type}\ {\scriptstyle \alpha_1\ indep\ p} \qquad \frac{\Gamma, p{:}\alpha \vdash \alpha_1[x := p] : type \quad \Gamma \vdash a : \alpha}{\Gamma \vdash ([x]\alpha_1)a = \alpha_1[x := a] : type}\ {\scriptstyle \alpha_1\ indep\ p}$$

extensionality and η-rule:

$$\frac{\Gamma, p{:}\alpha \vdash \beta_1 p = \beta_2 p : type}{\Gamma \vdash \beta_1 = \beta_2 : \alpha \to type}\ {\scriptstyle \beta_1, \beta_2\ indep\ p} \qquad \frac{\Gamma \vdash \beta : \alpha \to type}{\Gamma \vdash \beta = [x]\beta x : \alpha \to type}\ {\scriptstyle wf\ \beta}$$

FIGURE B.5. Family formation, β-conversion and equality rules

$$\frac{}{\vdash Set : type} \qquad \frac{\Gamma \vdash A : Set}{\Gamma \vdash A : type} \qquad \frac{\Gamma \vdash A = B : Set}{\Gamma \vdash A = B : type}$$

FIGURE B.6. The type of sets

Function types. The rule of formation of function types and the corresponding equality and inclusion rules are shown in Figure B.7

Then it comes the rules of (function) object application (in Figure B.8) and in Figure B.9 the formation of opbjects of funtional types together with the various equality rules.

Record Types. We then turn to present the rules of record types and record objects. In Figure B.10 there are the rules for record type formation and record type equality.

Formation and application of families of record types are given in Figure B.11

The so-called rule of fields and a rule of record inclusion are shown in Figure B.12.

Then, in Figure B.13 we show the rules saying when any two record types are in the inclusion relation.

formation of $\alpha{\rightarrow}\beta$:

$$\frac{\Gamma \vdash \alpha : \mathit{type} \quad \Gamma \vdash \beta : \alpha{\rightarrow}\mathit{type}}{\Gamma \vdash \alpha{\rightarrow}\beta : \mathit{type}}$$

equality and inclusion of $\alpha{\rightarrow}\beta$:

$$\frac{\Gamma \vdash \alpha_1 = \alpha_2 : \mathit{type} \quad \Gamma \vdash \beta_1 = \beta_2 : \alpha_1{\rightarrow}\mathit{type}}{\Gamma \vdash \alpha_1{\rightarrow}\beta_1 = \alpha_2{\rightarrow}\beta_2 : \mathit{type}}$$

$$\frac{\Gamma \vdash \alpha_2 \sqsubseteq \alpha_1 \quad \Gamma \vdash \beta_1 \sqsubseteq \beta_2 : \alpha_2{\rightarrow}\mathit{type}}{\Gamma \vdash \alpha_1{\rightarrow}\beta_1 \sqsubseteq \alpha_2{\rightarrow}\beta_2}$$

FIGURE B.7. Function types formation, equality and inclusion

application:

$$\frac{\Gamma \vdash f : \alpha{\rightarrow}\beta \quad \Gamma \vdash a : \alpha}{\Gamma \vdash fa : \beta a} \qquad \frac{\Gamma \vdash f = g : \alpha{\rightarrow}\beta \quad \Gamma \vdash a = b : \alpha}{\Gamma \vdash fa = gb : \beta a}$$

FIGURE B.8. Application of function objects

abstraction:

$$\frac{\Gamma, p{:}\alpha \vdash b[x := p] : \alpha_1[x := p]}{\Gamma \vdash [x]b : \alpha{\rightarrow}[x]\alpha_1}\ {\scriptstyle b,\alpha_1\ \mathit{indep}\ p}$$

β-conversion and extensionality:

$$\frac{\Gamma, p{:}\alpha \vdash b[x := p] : \alpha_1[x := p] \quad \Gamma \vdash a : \alpha}{\Gamma \vdash ([x]b)a = b[x := a] : \alpha_1[x := a]} \qquad \frac{\Gamma, p{:}\alpha \vdash fp = gp : \beta p}{\Gamma \vdash f = g : \alpha{\rightarrow}\beta}\ {\scriptstyle f,g,\beta\ \mathit{indep}\ p}$$

η and ξ-rules:

$$\frac{\Gamma \vdash b : \alpha{\rightarrow}\beta}{\Gamma \vdash [x]bx = b : \alpha{\rightarrow}\beta}\ {\scriptstyle \mathit{wf}\ b} \qquad \frac{\Gamma, p{:}\alpha \vdash f[x := p] = g[x := p] : \alpha_1[x := p]}{\Gamma \vdash [x]f = [x]g : \alpha{\rightarrow}[x]\alpha_1}\ {\scriptstyle f,g,\alpha_1\ \mathit{indep}\ p}$$

FIGURE B.9. Function object formation and equality rules

We end up showing in Figure B.14, first, the rules of record object extension. Then, the rule governing the selection of a label from a record object and the one which says that if two record objects are equal then the result of selectings the same *formation of record-types:*

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

type formation:

$$\frac{\Gamma \vdash \rho : \textit{record-type}}{\Gamma \vdash \rho : \textit{type}}$$

record types equality:

$$\frac{}{\vdash \langle\rangle = \langle\rangle : \textit{type}} \qquad \frac{\Gamma \vdash \rho_1 = \rho_2 : \textit{type} \quad \Gamma \vdash \beta_1 = \beta_2 : \rho_1\to\textit{type}}{\Gamma \vdash \langle \rho_1, L{:}\beta_1\rangle = \langle \rho_2, L{:}\beta_2\rangle : \textit{type}}$$

FIGURE B.10. Record types formation and equality rules

$$\frac{\Gamma \vdash \sigma \mathbin{\mathbf{:}} \alpha\to\textit{record-type} \quad \Gamma \vdash a : \alpha}{\Gamma \vdash \sigma a : \textit{record-type}} \qquad \frac{\Gamma, p{:}\alpha \vdash \rho[x := p] : \textit{record-type}}{\Gamma \vdash [x]\rho \mathbin{\mathbf{:}} \alpha\to\textit{record-type}}\ {\scriptstyle \alpha_1\ \textit{indep}\ p}$$

FIGURE B.11. Record types families

$$\frac{}{\Gamma \vdash \langle \rho, L{:}\beta\rangle \sqsubseteq \rho} \qquad \frac{}{\Gamma \vdash \beta \mathbin{\mathbf{:}} \rho\to\textit{type}}\ {\scriptstyle L\,:\,\beta \text{ in } \rho}$$

FIGURE B.12. Rules of fields

$$\frac{\Gamma \vdash \rho : \textit{record-type}}{\Gamma \vdash \rho \sqsubseteq \langle\rangle} \qquad \frac{\Gamma \vdash \rho_1 \sqsubseteq \rho_2 \quad \Gamma \vdash \beta_1 \sqsubseteq \beta_2 : \rho_1\to\textit{type}}{\Gamma \vdash \rho_1 \sqsubseteq \langle \rho_2, L{:}\beta_2\rangle}\ {\scriptstyle L\,:\,\beta_1 \text{ in } \rho_1}$$

FIGURE B.13. Inclusion of record types

label from them must be equal objects. Finally there are the equality rules for record objects.

record object extension:

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

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

selection:

$$\frac{\Gamma \vdash r : \rho}{\Gamma \vdash r.L : \beta r}\ {\scriptstyle L : \beta \text{ in } \rho} \qquad \frac{\Gamma \vdash r = s : \rho}{\Gamma \vdash r.L = s.L : \beta r}\ {\scriptstyle L : \beta \text{ in } \rho}$$

equality rules:

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

FIGURE B.14. Record object extension, selection and equality rules