J. Functional Programming 10 (2): 137–166, March 2000. Printed in the United Kingdom
© 2000 Cambridge University Press

Type checking dependent (record) types and subtyping

GUSTAVO BETARTE

Instituto de Computación, Universidad de la República, Montevideo, Uruguay
(*e-mail:* gustun@fing.edu.uy)

Abstract

In this work we put forward an algorithm for the mechanical verification of an extension of Martin-Löf's theory of types with dependent record types and subtyping. We first give a concise description of that theory and motivate its use for the formalization of algebraic constructions. Then we concentrate on the informal explanation and specification of a proof checker that we have implemented. The logical heart of this proof checker is a type checking algorithm for the forms of judgement of a particular formulation of the extended theory which incorporates a notion of parameter. The algorithm has been proven sound with respect to the latter calculus. We include a discussion on that proof in the present work.

Capsule Review

This paper describes a specification and implementation of a proof-checker for an extension of Martin-Löf's theory of types. This extension is motivated by the need for structuring formal developments: to provide an effective mechanism for packaging and handling related definitions, it introduces a notion of records, dependent record-types and subtyping as inclusion of record fields. In this way one can, for instance, form the type of binary relation on a given set as well as the type of partial equivalence relation on a given set, the latter being a subtype of the former.

Knowing whether a term has a type or not is undecidable in the proposed extension. However, the paper describes a subset of terms (generalized selections) for which typing can be decided. In practice, any well-typed term can be rewritten to fit in that set.

The chosen formulation uses parameters to handle free names, thus giving an elegant solution to renaming problems. Therefore, the implementation of the proposed algorithm is quite straightforward, as the reader will figure – a Haskell implementation is actually available.

1 Introduction

The subject of this paper is the specification and implementation of a proof checker for an extension of Martin-Löf's theory of logical types (Martin-Löf, 1987) with dependent record types and subtyping.

The original formulation of Martin-Löf's theory of types, from now on referred to as the logical framework, has been presented elsewhere (Nordström *et al.*, 1989;

https://doi.org/10.1017/S0956796899003627 Published online by Cambridge University Press

Coquand *et al*., 1994; Tasistro, 1997). 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 has been presented by several authors (Tasistro, 1997; Betarte and Tasistro, 1998). Dependent record types are just sequences of *fields* in which *labels* are declared as being of certain types. These types, in turn, may not only depend upon 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. 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.

We have investigated an alternative formulation of the extended theory (Betarte, 1998). In that formulation we make use of parameters, in the sense of Coquand (1991) and Pollak (1994), to stand for generic objects of the various types. The introduction of the notion of parameter allows us to give a solution to the problems posed by the manipulation of "free names" in the presence of dependent types. We also present (Betarte, 1998) the procedures for the mechanical verification of the forms of judgement of that variant of the extension. In this paper, we are mainly concerned with the design and specification of those procedures.

The plan for the rest of the paper is as follows. In the next section we start by giving a concise description of type theory and its use for carrying out constructive mathematics. Then we confine attention to the treatment of free names in systems of dependent types. We also briefly describe in this section dependent record types and the mechanism of subtyping they induce, as well as the proof checker we have implemented.

In section 3 we informally discuss the design of the type checking algorithm for the extended theory. We bring to attention the problems the checking of unlabelled abstractions presents, and how they are carried over to the procedures for checking the typing judgements of the extension.

In section 4 we concentrate on the final specification and implementation of the proof checker. In section 5 we comment on the decidability and correctness issues of the algorithms developed. Finally, we give some final conclusions and discuss related work.

2 The system and its implementation

The system of types of the logical framework is constituted, in the first place, by the type *Set*, the type of *inductively* defined sets. Then, any individual set A gives rise to the type of its elements. Type families are expressions of the language that, when applied to an individual of the appropriate type, yields a type. Moreover, it is possible to introduce arbitrary families of types in the formal language. Families can

https://doi.org/10.1017/S0956796899003627 Published online by Cambridge University Press

be constructed by an operation of abstraction, using the notation $[x]\alpha$, which binds the occurrences of the variable x in the type α. Finally, there exists a mechanism for the formation of (dependent) function types: if α is a type, and β is a family of types indexed by objects of type α then $\alpha\rightarrow\beta$ is also a type. The application of an object f of this latter type yields an object fa of type βa, if a is an object of type α.

The understanding of propositions as inductively defined by their introduction rules, as explained and justified by Martin-Löf (1987), allows us to grasp propositions as sets, and thereby their proofs as elements of those sets. There is, in principle, no formal distinction in the language of the theory between the type of sets and the type of propositions. Further, in the presence of families of types, this interpretation of propositions can be transferred to propositions about generic individuals. For instance, given a set A, $A\rightarrow[x](A\rightarrow[y]Set)$ is the type of binary relations on A. Then, if R is such a relation, for each element x of A we have a set Rxx. Since each set determines a type, we can form here a family of types over A, namely $[x]Rxx$. Then $A\rightarrow[x]Rxx$ is the type of proofs that R is reflexive. This function type is usually written as $(x : A)Rxx$, which can be read: "for any x in A, Rxx".

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

2.1 Parameters

The traditional formulation of the rule for the formation of a function type, for instance as presented by Nordström *et al.* (1989), says that if we know that α is a type and that β is a type family under the assumption that a variable x is of type α then we can form the function type $(x : \alpha)\beta$, where all occurrences of x in β become bound. Abstraction is then introduced as an operation of object formation. This is the corresponding rule

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

The stipulation for the formation of a context $\Gamma, x{:}\alpha$ (in Martin-Löf (1987, 1992) and Nordström *et al.* (1989)), for instance, requires that Γ is a context, α is a type under the context Γ and further, that the variable x has not already been declared in Γ. This last restriction is proper of systems of proof rules where an assumption, $x{:}\alpha$ say, may be introduced such that the type α depends on previous assumptions. Therefore, for the premiss of the latter rule of abstraction to be correct, it must be the case that x is not already declared in the context Γ.

Pollack (1994) 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 by Barendregt (1992). What is shown by Pollack is the impossibility of deriving, using the rules of PTS, the

https://doi.org/10.1017/S0956796899003627 Published online by Cambridge University Press

judgement $[x][x]x : (x : A)(y : Px)Px$ under the assumption that A is a type (an object of $*$) and P has kind $A \rightarrow *$. If one wants to understand the checking of the correctness of instances of the judgement $\Gamma \vdash [x]b : (x : \alpha)\beta$ as the upward reading of the rule of abstraction, one should proceed as follows: for checking that $[x][x]x : (x : A)(y : Px)Px$ check that $x{:}A \vdash [x]x : (y : Px)Px$. For this, in turn, we should check that $x : Px$ after extending the context $x{:}A$ with the declaration $x{:}Px$, but we are restrained from doing this by the criterion for context formation above.

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

Coquand (1991) investigates the question of checking the formal correctness of judgements of type and object equality in a formulation of Martin-Löf's set theory with generalized cartesian product and one universe.

The notion of context in this theory is that of a list of assumptions of the form $p{:}\alpha$, where p is a parameter and α a type (possibly depending on other parameters). In the formulation of the language of the theory, parameters are understood to play the role of the free variables occurring in the expressions. Consequently, they are used in the system to stand for generic objects of the various types. However, they are defined to be syntactic constructions distinct from the bound variables of the language. The distinction between parameters and bound variables allows us 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.

Pollack (1994) adopts the use of parameters to implement a type checking algorithm for a family of PTS (Barendregt, 1992). One of the motivations for introducing the notion of parameter, and consequently making use of them in the reformulation of the rules of inference of the formal system, is to provide a solution for problems similar to those 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 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

https://doi.org/10.1017/S0956796899003627 Published online by Cambridge University Press

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.

We formulate a variant of the extension (Betarte, 1998) presented by Tasistro (1997) and Betarte and Tasistro, 1998). A first difference is that we consider the rules of inference in their generalized form. Further, we make use of parameters to stand for generic objects of the various types. Thus, as the stipulation of an assumption will correspond to declare a parameter as of a certain type, the explanation of a relative judgement depends upon 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 that usually defined for the language of type theory, behaves as the textual replacement of a parameter by an expression.

The algorithms we shall present in this work implement the mechanical verification of the form of judgements of the calculus in Betarte (1998).

2.2 Record types

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

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

In dependent record types, the type α_{i+1} may depend upon 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 $\langle \mathsf{L}_1 : \alpha_1, \ldots, \mathsf{L}_i : \alpha_i \rangle$. This is formally expressed by the following two rules of record type formation:

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

We make use of the judgement $\beta : \rho \rightarrow type$, 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

https://doi.org/10.1017/S0956796899003627 Published online by Cambridge University Press

set, which we shall call *BinRel*, is written:

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

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

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

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

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

2.3 Subtyping

Dependent record types also induce inclusion polymorphism: given a record type ρ_1, it is possible to drop and permute fields of ρ_1 and still get a record type ρ_2. If that is the case, any object of type ρ_1 also satisfies the requirements imposed by the type ρ_2. That is, given $r : \rho_1$, we are justified in asserting also $r : \rho_2$. This is so because what is required to make the latter judgement is that the selections of the labels declared in ρ_2 from r are defined as objects of the appropriate types. And we have this, since every label declared in ρ_2 is also declared in ρ_1 and with the same type.

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

In the case of record types, the condition for $\rho_1 \sqsubseteq \rho_2$ is in words as follows: for each field $\mathsf{L} : \beta_2$ in ρ_2 there must be a field $\mathsf{L} : \beta_1$ in ρ_1 with $\beta_1 \sqsubseteq \beta_2 : \rho_1 \to \mathit{type}$. It is easy to see that if $\mathsf{L} : \beta_1$ is a field of a record type ρ_1 then, by the subtyping induced on families of types, β_2 can be considered to be a family over ρ_1, and thereby the previous (informal) explanation makes sense.

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

That two objects r and s of type $\langle \mathsf{L}_1 : \alpha_1, \ldots, \mathsf{L}_n : \alpha_n \rangle$ are the same means that the selection of the labels L_i's from r and s result in equal objects of the corresponding types. Therefore, equality of record objects is based on a kind of extensionality principle. That is, the two rules below can be understood as defining that two objects of a given record type are equal if the selections of every label of the record type in question from the objects are equal. Notice that the type in which two record objects are compared is relevant: suppose that r and s are of type ρ_1 and that

https://doi.org/10.1017/S0956796899003627 Published online by Cambridge University Press

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

Fig. 1. Syntax of input expressions.

$\rho_1 \sqsubseteq \rho_2$. Then it may well be the case that $r{=}s : \rho_2$ but not $r{=}s : \rho_1$:

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

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

2.4 The implemented system

The proof checker we shall describe has been implemented. The programming language used is Haskell 1.4, and the code has been compiled using Chalmers Haskell-B, the compiler implemented by Lennart Augustsson at Chalmers University of Technology (Augustsson, 1997).

The general design and implementation of the system follows the approach in which 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 task of checking declarations. 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 Wadler (1992), 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, and it interacts with a parsing monad. The latter is implemented following the ideas of several other authors (Burge, 1975; Hutton, 1992; Röjemo, 1995).

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

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

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

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

https://doi.org/10.1017/S0956796899003627 Published online by Cambridge University Press

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

The expressions $[x]i$ are abstractions, and therefore the occurrences of x are bound in $[x]i$. With $\langle i_1, L = i_2\rangle$ and $\langle i_1, L{:}i_2\rangle$ we denote (binary) record object and record type formation. We shall later comment on *let* and *use* expressions.

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

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

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

with T, F and c as constant names, x, $x_1, \ldots, x_n$ as variables and i, α and $\alpha_1, \ldots, \alpha_n$ belonging to the language of expressions above.

The former is called a *type* declaration. It allows us 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 the name c, of an expression $[x_1][x_2]\ldots[x_n]i$ of type $\alpha_1{\rightarrow}[x_1](\alpha_2{\rightarrow}\ldots(\alpha_n{\rightarrow}[x_n]\alpha)\ldots)$, with $n \geq 0$. This form of declaration corresponds to the so-called explicit definition of a constant in ALF Magnusson (1995). We are considering neither definitions of inductive (families of) sets, nor the implicit definition of constants – the latter are usually introduced 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. Thus, the definiendum of D may occur in any declaration introduced after it.

3 Informal explanations

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

We now describe a type checking algorithm for the extended theory. For the sake of conciseness, we confine our attention to the checking of judgements of the form $a : \alpha$, where we assume that $\alpha : \mathit{type}$.

3.1 Informal description of type checking

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

https://doi.org/10.1017/S0956796899003627 Published online by Cambridge University Press

theory)

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

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

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 (Dowek, 1993). Therefore, we choose not to try to type check 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 every object that can still be formed in the theory can be expressed in such a way as to be accepted by the type checking algorithm. For instance, any abstraction $[x]b$ that stands for an object of type $\alpha \to \beta$ can be given a name in type theory, by a definition of the form:

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

Using this definition, we can then express an object $([x]b)a : \beta a$ as $f\,a$, which is no longer a redex.

More precisely, we have that if $c : \alpha$ is valid, then there is $c' : \alpha$ such that $c{=}c' : \alpha$ and $c' : \alpha$ is accepted by the type checking algorithm. In particular, instead of expressions containing beta redexes, one has to write their corresponding beta normal forms. Indeed, the resulting syntax is not only sufficiently expressive in the sense indicated above, but also natural, since in practice one does not write down β-redexes.

For 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. To check the conclusion we need to infer the type ρ for r. Now, analogous to the case for the (unlabeled) abstractions, we cannot decide in general whether an extension $\langle r, L = a\rangle$ does or does not have a type. This would in turn require us to decide whether or not the (arbitrary) expression a has a type.

There are other difficulties with record object extensions, which we now make clear.

3.1.1 *Type checking of record extensions*

Record object extensions are of one of the forms $\langle L_1 = a_1, \ldots, L_n = a_n\rangle$ and $\langle f, L_1 = a_1, \ldots, L_n = a_n\rangle$, where f is not itself a record extension.

A record object extension which has a certain type ρ can be derived in the calculi presented by Tasistro (1997), Betarte and Tasistro (1998) and Betarte (1998), starting

https://doi.org/10.1017/S0956796899003627 Published online by Cambridge University Press

from $\langle\rangle : \langle\rangle$, by alternate use of the proof rules

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

Let us first consider the problem of checking whether $\langle L_1 = a_1, \ldots, L_n = a_n\rangle : \rho$. We 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 an appropriate type. For this, we can proceed recursively on the components of r that correspond to the labels in ρ.

Notice, 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 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 those labels of the object r *unreachable* that either do not occur in ρ or are overridden.

It is clear 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 a 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 eventually being checked. So, checking only the restrictions imposed by the given type is safe from this point of view. 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 expressions can be rejected by this method. Of particular importance is the case in which we have $\langle L_1 = a_1, \ldots, L_n = a_n\rangle : \rho_1$ 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 $\langle L_1 = a_1, \ldots, L_n = a_n\rangle$ 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 $\langle L_1 = a_1, \ldots, L_n = a_n\rangle$ 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 $\langle f, L_1 = a_1, \ldots, L_n = a_n\rangle$, in addition, allow us to express a restricted form of overriding. Notice that it can 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 will not need to inspect the components of f.

There is, then, in principle, a choice between a permissive and a restrictive method for dealing with the type checking of record extensions. The latter seems to allow for

https://doi.org/10.1017/S0956796899003627 Published online by Cambridge University Press

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 Tasistro (1997).

3.2 The algorithms

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

So expressions that are not abstractions or record extensions must be of the form $(h\ a_1 \dots a_m).L_1 \dots .L_n$, with m and $n \geq 0$. We call these expressions *generalized selections*. Here the a_is are expressions in β- and selection-normal form. The L_js are labels and the head h must now be of one of the form $x.L_1 \dots .L_n$ or $c.L_1 \dots .L_n$.

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

$$\begin{array}{lcl} e & ::= & [x]e \mid \langle L_1 = e_1, \dots, L_n = e_n\rangle \mid \langle f, L_1 = e_1, \dots, L_n = e_n\rangle \mid \\ & & e_1 \rightarrow e_2 \mid \langle e_1, L{:}e_2\rangle \mid f \\ f & ::= & x \mid c \mid (f\ e) \mid f.L \end{array}$$

The expressions f are the generalized selections. According to the observation made at the beginning, it is for these expressions that type checking links itself with type inference. More precisely, the following algorithm can decide whether or not a generalized selection has a type:

Type inference for generalized selections

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.

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.

Notice that there is at most one declaration for each variable or constant. Then it follows by an inductive argument that a generalized selection f has at most one inferred type. As a consequence, if f has type α then it has inferred type α_1 and $\alpha_1 \sqsubseteq \alpha$. This solves the problem of inferring the type of generalized selections.

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

https://doi.org/10.1017/S0956796899003627 Published online by Cambridge University Press

Typing terms

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 $e[x := p] : \beta_1 p$, adding $p{:}\alpha_1$ to the declarations of parameters, for a fresh parameter p.

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

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

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 $f : \rho$ we do not need to inspect the components of f. The only condition that we need 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. In the next two algorithms, the form of a defined type must be understood to be the form of its ultimate definiens.

Type inclusion

The checking of type inclusion proceeds recursively on the form of the types.

First, any record type ρ is a subtype of the empty record. For checking that a record type ρ_1 is included in the record type $\langle \rho_2, L{:}\beta_2 \rangle$, proceed as follows: check that ρ_1 is included in ρ_2. Then look for a declaration $L{:}\beta_1$ in ρ_1. If this turns out to be successful, then for a fresh parameter p taken as of type ρ_1, check that $\beta_1 p$ is included in $\beta_2 p$.

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 $\beta_1 p$ is included in $\beta_2 p$ for a fresh parameter p 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*.

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

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 selections as defined above. Observe that if an object is in head normal form and its head is a constant, this latter must necessarily be

https://doi.org/10.1017/S0956796899003627 Published online by Cambridge University Press

a primitive one. The algorithm proceeds by comparing the heads. In case they are the same constant or parameter, c 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 c, whose type can be recovered from the list of declarations.

For checking $g{=}h : \alpha{\to}\beta$, check whether $gp{=}hp : \beta p$ adding $p{:}\alpha$ to the declarations of parameters, for a fresh parameter p.

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 $i = 1, \ldots, n$.

The approach used for checking object equality follows that taken by Magnusson (1995). The process for checking the 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 elsewhere (Coquand, 1991; Coquand, 1996). However, in the presence of record types and subtyping, it is not in general possible to check the equality of record objects without considering type information.

3.3 Let and Use expressions

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 (1996), i.e. to check that an expression *let* $x : \alpha_1 = e_1$ *in* e has a certain type α in an environment $\mathscr{E}$ proceed as follows: check first that $x : \alpha_1 = e_1$ is a valid declaration in $\mathscr{E}$. If this succeeds, check that e is an object of type α in the environment $\mathscr{E}$ locally extended with $x : \alpha_1 = e_1$. The checking of the expression e, in addition to considering 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*.

On the other hand, we have recently been experimenting with *use* expressions. The effect of "using" an expression r of type ρ in an expression e is almost analogous to that achieved by the Pascal command *with*, i.e. 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 with 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 $\mathscr{E}$: check first that ρ is a record type in the environment $\mathscr{E}$. 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 of the form $\langle \mathsf{L}_1 : \beta_1, \ldots, \mathsf{L}_n : \beta_n \rangle$, with $n \geq 0$. Then, locally extend the environment $\mathscr{E}$ 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.

https://doi.org/10.1017/S0956796899003627 Published online by Cambridge University Press

4 Formalization of the algorithms

We now intend to give a precise formulation of the informal explanations in section 3.1 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. We shall also then formulate the algorithm for checking judgements of the form $\alpha : \mathit{type}$, and thereby also for judgements of the form $\beta : \alpha \rightarrow \mathit{type}$. For the sake of conciseness, we consider here neither the checking of *let* nor *use* expressions.

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 environment, and then we proceed by checking that $\alpha_1[x := p]$ is a type.

4.1 Valid declarations

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

Definition 4.1 (*Checking environment*)
A checking environment ($\mathscr{E}$) 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 $\mathscr{E}_\Sigma$ and the context part by $\mathscr{E}_\Gamma$.

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

Definition 4.2

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

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

$$\frac{T \notin \mathit{Dom}\ \mathscr{E} \quad \mathsf{checkType}\ \mathscr{E}\ \alpha}{\mathscr{E} \vdash T : \mathtt{type} = \alpha \quad \mathsf{valid}}$$

$$\frac{F \notin \mathit{Dom}\ \mathscr{E} \quad \mathsf{checkType}\ \mathscr{E}\ \alpha \quad \mathsf{checkTypeFam}\ \mathscr{E}\ ([x]\alpha_1)\ \alpha}{\mathscr{E} \vdash F(x : \alpha) : \mathtt{type} = \alpha_1 \quad \mathsf{valid}}$$

$$\frac{c \notin \mathit{Dom}\ \mathscr{E} \quad \mathsf{checkType}\ \mathscr{E}\ T \quad \mathsf{checkExp}\ \mathscr{E}\ ([x_1][x_2]\ldots[x_n]e)\ T}{\mathscr{E} \vdash c(x_1 : \alpha_1, \ldots, x_n : \alpha_n){:}\ \alpha = e \quad \mathsf{valid}}$$

where T in the last rule abbreviates the type $\alpha_1 \rightarrow [x_1](\alpha_2 \rightarrow \ldots(\alpha_n \rightarrow [x_n]\alpha)\ldots)$.

The procedures checkType, checkTypeFam and checkExp above perform the checking, in the environment $\mathscr{E}$, that α is a type, $[x]\alpha_1$ is a type family over α and

https://doi.org/10.1017/S0956796899003627 Published online by Cambridge University Press

the expression $[x_1][x_2]\ldots[x_n]e$ is an object of type T, respectively. They are defined in section 4.3.

Definition 4.3 (*Valid updating*)
After a declaration D is checked, the updating of the checking environment $\mathscr{E}$, if D is valid, is respectively defined to be

- $\mathscr{E} + T : type = \alpha$ or, if α is of the form $\langle \rho, L{:}\beta \rangle$, $\mathscr{E} + T : record\text{-}type = \alpha$
- $\mathscr{E} + F : \alpha \rightarrow [x]type = [x]\alpha_1$ or, if α_1 is of the form $\langle \rho, L{:}\beta \rangle$, $\mathscr{E} + F : \alpha \rightarrow [x]record\text{-}type = [x]\alpha_1$
- $\mathscr{E} + c : \alpha_1 \rightarrow [x_1](\alpha_2 \rightarrow \ldots(\alpha_n \rightarrow [x_n]\alpha)\ldots) = [x_1][x_2]\ldots[x_n]e$

Now we introduce a function for computing the weak-head normal form of a well-formed expression. It shall, when needed, also unfold constants which have been introduced in the environment $\mathscr{E}$. We make extensive use of this function in the algorithms we present below.

4.2 Weak-head normalization

We start by introducing the notions of *weak-head normal form* and *top level redex* for expressions.

Definition 4.4

- An expression is in weak-head normal form if it is either of the form $[x]e$, $\alpha \rightarrow \alpha'$, $\langle\rangle$, $\langle \rho, L{:}\beta \rangle$, $\langle e, L = e' \rangle$ or $(h.L_1 \ldots L_l\ a_1 \ldots a_m).K_1 \ldots K_n$ with l, m and $n \geq 0$ and h a parameter or a sort.
- A top-level redex is an expression of the form $(f a_1 \ldots a_n)$ 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 definition of the function $\Downarrow$ is given in figure 2. Due to the presence of constants in expressions, it also takes as argument the typed environment Σ of a checking environment $\mathscr{E}$. We use $e \Downarrow \Sigma$ to denote its application to expression e and environment Σ.

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 also that no reduction is performed under binders.

Remark
The operation of substitution we are considering in the definition of $\Downarrow$ does not perform renaming. However, the normalization of an expression will take place only when it is a well-typed one. Therefore, it is always the case that there are no free occurrences of variables in the expression e that is substituted for the variable x.

https://doi.org/10.1017/S0956796899003627 Published online by Cambridge University Press

$$
\begin{array}{lll}
p \Downarrow \Sigma & =_{def} & p \\
s \Downarrow \Sigma & =_{def} & s \\
c \Downarrow \Sigma & =_{def} & red_{\delta}\ c\ \Sigma \\
[x]e \Downarrow \Sigma & =_{def} & [x]e \\
\langle\rangle \Downarrow \Sigma & =_{def} & \langle\rangle \\
\langle e, L = e'\rangle \Downarrow \Sigma & =_{def} & \langle e, L = e'\rangle \\
\alpha{\rightarrow}\beta \Downarrow \Sigma & =_{def} & \alpha{\rightarrow}\beta \\
\langle \rho, L{:}\beta\rangle \Downarrow \Sigma & =_{def} & \langle \rho, L{:}\beta\rangle \\
f\,e \Downarrow \Sigma & =_{def} & red_{\beta}\ f\ e\ \Sigma \\
r.L_1 \Downarrow \Sigma & =_{def} & red_{\sigma}\ r\ L_1\ \Sigma
\end{array}
$$

where

$$
\begin{array}{lll}
red_{\delta}\ c\ \Sigma & =_{def} & e \Downarrow \Sigma \qquad \text{if } c : \alpha = e \text{ in } \Sigma \\
red_{\beta}\ f\ e\ \Sigma & =_{def} & \text{let } f' = f \Downarrow \Sigma \\
 & & \text{in if } f' = [x]f'' \\
 & & \quad \text{then } f''[x := e] \Downarrow \Sigma \\
 & & \quad \text{else } f'e \\
red_{\sigma}\ r\ L_1\ \Sigma & =_{def} & \text{let } r' = r \Downarrow \Sigma \\
 & & \text{in if } r' = \langle r'', L_2 = e\rangle \\
 & & \quad \text{then if } L_1 = L_2 \\
 & & \qquad \text{then } e \Downarrow \Sigma \\
 & & \qquad \text{else } red_{\sigma}\ r''\ L_1\ \Sigma \\
 & & \quad \text{else } r'.L_1
\end{array}
$$

Fig. 2. Weak-head normalization.

Parameters, on the other hand, might occur in e, but they are not captured by the abstraction operator.

4.3 The programs

We shall now present programs for checking that an expression α is a type (checkType $\mathscr{E}$ α) and that an expression a is an object of type α (checkExp $\mathscr{E}$ a α). As explained in section 3.1, the construction of these algorithms is intertwined with that of the algorithms for inferring the type of a generalized selection (inferExp $\mathscr{E}$ f $\gg \alpha$), checking inclusion of types (typeIncl $\mathscr{E}$ α α_1) and conversion of objects (objConv $\mathscr{E}$ a b α).

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 approach for presenting the semantics of a program follows the one taken by B. Nordström (in preparation).

Some of the rules will also have a side condition, which can either be

- p fresh in $\mathscr{E}$
- $p : \alpha$ in $\mathscr{E}$
- $c : e = \alpha$ in $\mathscr{E}$

https://doi.org/10.1017/S0956796899003627 Published online by Cambridge University Press

- L fresh in ρ
- $L : \beta$ in ρ
- $s \in \mathscr{S}$.

They should respectively be read as: "there is no entry for the parameter p in the context $\mathscr{E}_\Gamma$", "the lookup of p in $\mathscr{E}_\Gamma$ yields α", "the lookup of c in $\mathscr{E}_\Sigma$ yields (e, α)", "the label L does not occur in the fields of the (weak-head normal form of) ρ", "there exists a field declaration $L{:}\beta$ in the (weak-head normal form of the) record type ρ" and "s is a sort".

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

In most of the programs we shall use the non-terminals e and f defined in section 3.2. In particular, the intended use we make of f in the rules below is that it will match any expression that is a generalized selection, that is to say, it is either a variable, a constant, a permissible application or a permissible selection.

In addition, the absence in a program of a rule for a particular form of expression indicates that the program will fail if provided with such argument.

To begin, we now introduce the mutually defined programs checkType, checkRecType and checkTypeFam. We recall that they check whether the expression α is a type, ρ is a record type and the expression β is a family of types over α, respectively.

Checking types (checkType $\mathscr{E}$ α)
This program is recursively defined by cases on the form of the expression α.

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

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

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

Observe that in the last rule we perform the checking of an object of type *Set* using (sort) inference. The same procedure also allows us to check the correctness of type expressions formed out of constants introduced by type and type family declarations.

As for record types, we now introduce the following rules:

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

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

As has already been explained (Tasistro, 1997; Betarte and Tasistro, 1998), 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,

https://doi.org/10.1017/S0956796899003627 Published online by Cambridge University Press

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:

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

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

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

$$\frac{\mathsf{inferExp}\ \mathscr{E}\ f \gg \mathit{record\text{-}type}}{\mathsf{checkRecType}\ \mathscr{E}\ f}$$

Checking type families (checkTypeFam $\mathscr{E}$ β α)
This program is defined by cases on the expression β. There exist 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 $\mathscr{E}$ is a valid environment, and that α has already been checked to be a type in this environment.

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

We are assuming an infinite set P of parameters; as expressions and contexts are finite we can always choose a fresh one.

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

Notice that this rule subsumes the case of record families, and that it is checked whether α is a subtype of α_1.

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

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

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

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

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

According to the explanations in section 3.1 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

https://doi.org/10.1017/S0956796899003627 Published online by Cambridge University Press

of α. The side condition prevents unnecessary conversion checking. This control will become more clear after we present the definition of the function inferExp.

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

We now explain why this definition of checkExp corresponds to the restrictive method formulated in section 3.1. Observe, first, that the rule above for checking record object extensions rejects unreachable 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 of 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 through an n-ary version of the last but one rule. For the sake of clarity, however, we prefer this presentation which, in addition, allows us to simplify the proofs when reasoning about the correctness of the algorithm.

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

$$\frac{}{\mathsf{inferExp}\ \mathscr{E}\ p \gg \alpha}\ {\scriptstyle p : \alpha\ \mathrm{in}\ \mathscr{E}}$$

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

$$\frac{}{\mathsf{inferExp}\ \mathscr{E}\ c \gg \alpha}\ {\scriptstyle c : \alpha = e\ \mathrm{in}\ \mathscr{E}}$$

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

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

We end up the definition of this function by considering the case for selections:

$$\frac{\mathsf{inferExp}\ \mathscr{E}\ r \gg \rho}{\mathsf{inferExp}\ \mathscr{E}\ r.L \gg \beta r}\ {\scriptstyle L : \beta\ \mathrm{in}\ \rho}$$

Checking type inclusion (typeIncl $\mathscr{E}\ \alpha_1\ \alpha_2$)
This program is simultaneously defined with the program whTypeIncl.

A remark is in order before we provide the rules defining the program typeIncl. Observe first that the computation of typeIncl $\mathscr{E}\ \alpha_1\ \alpha_2$ is triggered, for instance, by the rule that checks whether a generalized selection 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

https://doi.org/10.1017/S0956796899003627 Published online by Cambridge University Press

inferExp, it could also be the sort t, with $t \in \{type, record\text{-}type\}$, or an expression of the form $\alpha \rightarrow [x]t$. However, the possibility has been ruled out for these forms of expressions to be arguments of the program typeIncl.

Let us now turn back to the definition of the program. First, it is checked whether the expressions are syntactically equal:

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

Observe that this subsumes the case in that α is *Set*. If the type expressions are not syntactically equal, both α_1 and α_2 are reduced to their corresponding weak-head normal forms, which are in turn the input to the program whTypeIncl. This latter is recursively defined by case analysis on the form of its arguments.

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

For checking that two function types are in the inclusion relation, it must be checked in turn that the types and type families out of which they are formed are respectively related. As usual the type former $\rightarrow$ is contravariant on the index type. Notice, in addition, that care is taken of type dependency when checking the (covariant) relation of result types.

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

Remark
Once it is checked that α_2 is a subtype of α_1, 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 two rules directly implement the explanation for two record forms to be in the inclusion relation:

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

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

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

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

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.

https://doi.org/10.1017/S0956796899003627 Published online by Cambridge University Press

Object conversion (objConv $\mathscr{E}$ 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 $\alpha \to \beta$, a record type 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.

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

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.

$$\frac{\mathsf{objConv}\ \mathscr{E}, p{:}\alpha\ gp\ hp\ \beta p}{\mathsf{whObjConv}\ \mathscr{E}\ g\ h\ \alpha \to \beta}\ {}^{p \text{ fresh in } \mathscr{E}}$$

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 section 3.1 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.

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

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

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

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

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 a subtype of 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 selection. Furthermore, as both are in weak-head normal form, either they are parameters or otherwise applications or selections whose heads are parameters. Thus, for checking the convertibility of a_1 and b_1, three cases must be considered:

1. either they are both the same parameter;
2. they are objects of the form ga_2 and hb_2, respectively, g and h are convertible objects of a function type $\alpha_2 \to \beta$ and a_2 and b_2 are convertible objects of type α_2; or
3. they are of the form $r.L_1$ and $s.L_2$, respectively, and what must be checked then is whether $L_1 = L_2$ and that r and s are head-convertible objects of some type ρ

https://doi.org/10.1017/S0956796899003627 Published online by Cambridge University Press

We can perform the two last procedures using the object conversion program only if we infer the type of one of g and h and the type of one of r and s, respectively (what we can do because they are generalized selections). 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 discuss a possible 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 number of computations might be unnecessarily performed. Think, for instance, of the case when both a and b are the same constant c of type $\alpha\rightarrow\beta$. As an heuristic to improve the efficiency of the whole procedure of object conversion checking we could consider introducing the following rule:

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

It would act as the formal counterpart of the reflexivity rule of the equality of objects of a certain type. This rule then should be the first considered in the definition of the program objConv. The heuristic, however, relies on the fact that the cost of comparison is smaller than the cost of recursively applying the algorithm. Although no precise benchmark has been performed, our experience is that the performance of the system is improved if the syntactical equality is checked in the first place.

The definition of the next program ends the formulation of the type checking algorithm for the extended theory.

Head conversion (headConv $\mathscr{E}\ a\ b \gg \alpha$)

$$\frac{}{\mathsf{headConv}\ \mathscr{E}\ p\ p \gg \alpha}\ {\scriptstyle p:\alpha\ \mathrm{in}\ \mathscr{E}}$$

$$\frac{\mathsf{headConv}\ \mathscr{E}\ g\ h \gg \alpha \quad \alpha \Downarrow \Sigma \gg \alpha_1\rightarrow\beta \quad \mathsf{objConv}\ \mathscr{E}\ a\ b\ \alpha_1}{\mathsf{headConv}\ \mathscr{E}\ ga\ hb \gg \beta a}$$

$$\frac{\mathsf{headConv}\ \mathscr{E}\ r\ s \gg \rho}{\mathsf{headConv}\ \mathscr{E}\ r.L\ s.L \gg \beta r}\ {\scriptstyle L:\beta\ \mathrm{in}\ \rho}$$

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.

5 Correctness of the algorithm

We have proven the algorithm presented in the previous section to be sound with respect to the calculus presented in Betarte (1998, Chapter 4). In this section we shall concentrate on this soundness result.

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

https://doi.org/10.1017/S0956796899003627 Published online by Cambridge University Press

unfolding of contexts:

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

unfolding of expressions:

$$\begin{array}{lll}
x^*_\Sigma & =_{def} & x \\
p^*_\Sigma & =_{def} & p \\
s^*_\Sigma & =_{def} & s \\
c^*_\Sigma & =_{def} & e^*_{\Sigma_1} \qquad \textit{with} \;\; \Sigma = \Sigma_1; c : \alpha = e; \Sigma_2 \\
([x]e)^*_\Sigma & =_{def} & [x]e^*_\Sigma \\
\langle\rangle^*_\Sigma & =_{def} & \langle\rangle \\
\langle e_1, L = e_2\rangle^*_\Sigma & =_{def} & \langle e_{1\Sigma}^*, L = e_{2\Sigma}^*\rangle \\
(\alpha{\to}\beta)^*_\Sigma & =_{def} & \alpha^*_\Sigma{\to}\beta^*_\Sigma \\
\langle\rho, L{:}\beta\rangle^*_\Sigma & =_{def} & \langle\rho^*_\Sigma, L{:}\beta^*_\Sigma\rangle \\
(f\,e)^*_\Sigma & =_{def} & f^*_\Sigma e^*_\Sigma \\
(r.L)^*_\Sigma & =_{def} & r^*_\Sigma.L
\end{array}$$

Fig. 3. Unfolding.

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 (1996) where a formulation of PTS with definitions is presented. Magnusson (1995), on the other hand, for the correctness proofs of the algorithms presented just assumes that such a set of declarations has a formal counterpart in Martin-Löf's calculus of explicit substitution (Martin-Löf, 1992; Tasistro, 1997), which is 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. In particular, when reasoning on the correctness of the procedures we have defined to check the formal correctness of judgements of the calculus in question we prove, for instance, that if checkType (Σ, Γ) α succeeds then it holds that $\Gamma^*_\Sigma \vdash \alpha^*_\Sigma : \textit{type}$. 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).

5.1 Unfolding and basic properties

The definition of the unfolding function, which we show in figure 3, is much in the spirit of the *projection mapping* introduced in chapter 11 of Severi's thesis (Severi, 1996).

It is possible to prove that the unfolding of an expression 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, $\mathscr{C}(\Sigma, e)$ say, which decreases when the function is used. This measure computes the number

https://doi.org/10.1017/S0956796899003627 Published online by Cambridge University Press

of constants replaced in the expression e when the unfolding of this expression is performed with environment Σ.

We now introduce the following:

Definition 5.1

- 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 expression α, p is a fresh parameter for Γ and $\Gamma^*_\Sigma \vdash \alpha^*_\Sigma : \mathit{type}$.
- A checking environment $\mathscr{E}$ is valid if $\mathscr{E}_\Sigma$ is valid and $\mathscr{E}_\Gamma$ is valid w.r.t. $\mathscr{E}_\Sigma$.

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 $\mathscr{E}$ is initialized to be the pair $(\{\},[])$. By construction then $\mathscr{E}$ is valid. When the checking of a declaration begins $\mathscr{E}_\Gamma$ is always []. It can be proved that in the algorithms presented in the previous section all the extensions we have made of the context preserve its validity, as above defined.

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

In Betarte (1998, Chapter 4) a calculus that incorporates the notion of parameter is put forward as a variant of that presented by Betarte and Tasistro (1998) and Tasistro (1997). A weak head reduction relation $\Rightarrow$ is defined for the expressions of the calculus. There is in principle no need for introducing one such relation – the corresponding meaning explanations of the forms of judgement that the calculus embodies do not depend upon any such notion. However, to define an algorithm for checking the formal correctness of judgements of the theory, we chose to introduce the relation mentioned. Its use renders the checking process more efficient. We prove (Betarte, 1998) the subject reduction property for the forms of judgement $\Gamma \vdash \alpha : \mathit{type}$ and $\Gamma \vdash a : \alpha$. These properties are crucial when proving the correctness of the algorithm.

In the previous section, and in contrast to the usual presentation of this kind of algorithm, we have used a recursively defined function to compute the weak head normal form of well-typed expressions. 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 section 3.1, 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 successful, needs to completely normalize its arguments.

https://doi.org/10.1017/S0956796899003627 Published online by Cambridge University Press

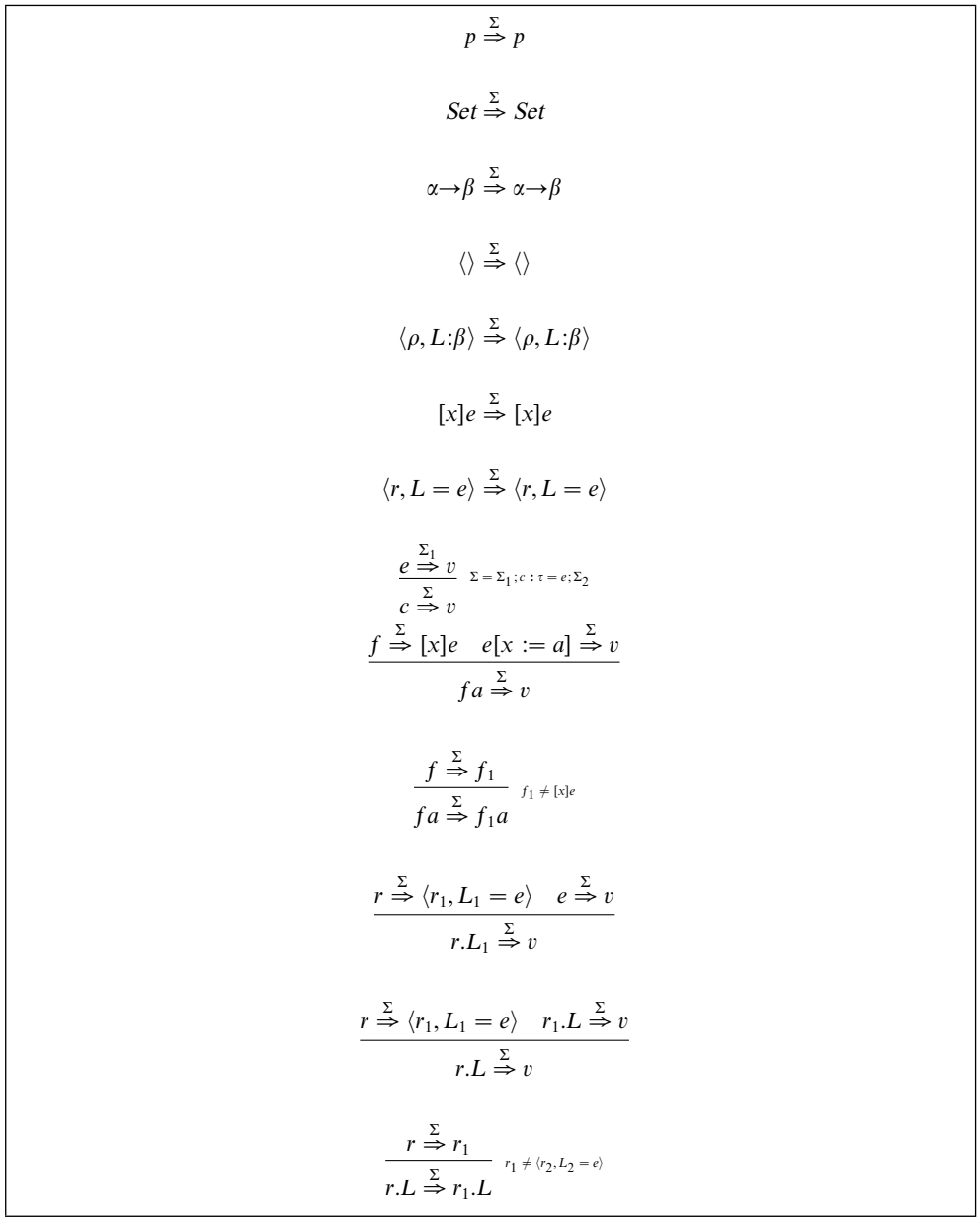

Fig. 4. Weak head reduction relation in a typed environment.

On the other hand, we could have defined an inductive reduction relation $\overset{\Sigma}{\Rightarrow}$, which we show in figure 4, which extends the relation $\Rightarrow$ to consider the unfolding of constants present in the typed environment. Therefore, any premiss of the form $e_1 \Downarrow \Sigma \gg e_2$ would be replaced by one of the form $e_1 \overset{\Sigma}{\Rightarrow} e_2$. However, we would then place ourselves in the situation that what we are defining, when introducing checkExp $\mathscr{E}$ e, $f$or instance, is closer to an inductively defined predicate on ex-

https://doi.org/10.1017/S0956796899003627 Published online by Cambridge University Press

pressions than a program. This latter approach is particularly useful if one wants to carry out proofs of properties as those we shall formulate 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 $e_1 \overset{\Sigma}{\Rightarrow} e_2$.

On the other hand, it is quite easy to prove 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 understand in the proofs of the properties that follows, that an assumption of the form $e_1 \Downarrow \Sigma \gg e_2$ amounts to one of the form $e_{1\Sigma}^* \Rightarrow e_{2\Sigma}^*$.

5.2 Soundness

The proofs of the propositions here introduced can be found in Betarte (1998, Chapter 5). We define $\mathscr{T}$ to be the set $\mathscr{S} - \{Set\}$.

Proposition 5.1
Let $\mathscr{E}$ be the valid checking environment (Σ, Γ). Then it holds that Γ_Σ^* *context*.

Proof
By induction on the definition of valid typed environment.

Proposition 5.2
Let $\mathscr{E}$ be the valid checking environment (Σ, Γ),

$th_1 \diamond$ if checkType $\mathscr{E}$ α then $\Gamma_\Sigma^* \vdash \alpha_\Sigma^* : type$

$th_2 \diamond$ if checkRecType $\mathscr{E}$ ρ then $\Gamma_\Sigma^* \vdash \rho_\Sigma^* : record\text{-}type$

$th_3 \diamond$ if checkType $\mathscr{E}$ α and checkTypeFam $\mathscr{E}$ β α
then $\Gamma_\Sigma^* \vdash \beta_\Sigma^* : \alpha_\Sigma^* \rightarrow type$.

$th_4 \diamond$ if $\Gamma_\Sigma^* \vdash \alpha_\Sigma^* : type$ and checkExp $\mathscr{E}$ a α
then $\Gamma_\Sigma^* \vdash a_\Sigma^* : \alpha_\Sigma^*$.

$th_5 \diamond$ if inferExp $\mathscr{E}$ $f \gg \alpha$ then either

i) $\Gamma_\Sigma^* \vdash \alpha_\Sigma^* : type$ and $\Gamma_\Sigma^* \vdash f_\Sigma^* : \alpha_\Sigma^*$,

ii) $\alpha \in \mathscr{T}$ and $\Gamma_\Sigma^* \vdash f_\Sigma^* : \alpha$ or

iii) $\alpha = \alpha_1 \rightarrow [x]t$, $t \in \mathscr{T}$ and $\Gamma_\Sigma^* \vdash f_\Sigma^* : \alpha_{1\Sigma}^* \rightarrow [x]t$

Proof
The proof proceeds by simultaneous induction on the definitions of the programs involved in the proposition.

https://doi.org/10.1017/S0956796899003627 Published online by Cambridge University Press

Proposition 5.3
Let $\mathscr{E}$ be the valid environment (Σ, Γ) and let us assume

- $\Gamma^*_\Sigma \vdash \alpha_{1\Sigma}^* : \mathit{type}$ and $\Gamma^*_\Sigma \vdash \alpha_{2\Sigma}^* : \mathit{type}$ for the cases th_1 and th_2.
- $\Gamma^*_\Sigma \vdash \alpha^*_\Sigma : \mathit{type}$, $\Gamma^*_\Sigma \vdash a^*_\Sigma : \alpha^*_\Sigma$ and $\Gamma^*_\Sigma \vdash b^*_\Sigma : \alpha^*_\Sigma$ for the cases th_3 and th_4.

$th_1 \diamond$ If typeIncl $\mathscr{E}$ α_1 α_2 then
$\Gamma^*_\Sigma \vdash \alpha_{1\Sigma}^* \sqsubseteq \alpha_{2\Sigma}^*$

$th_2 \diamond$ If whTypeIncl $\mathscr{E}$ α_1 α_2 then
$\Gamma^*_\Sigma \vdash \alpha_{1\Sigma}^* \sqsubseteq \alpha_{2\Sigma}^*$

$th_3 \diamond$ If objConv $\mathscr{E}$ a b α then
$\Gamma^*_\Sigma \vdash a^*_\Sigma = b^*_\Sigma : \alpha^*_\Sigma$

$th_4 \diamond$ If whObjConv $\mathscr{E}$ a b α then
$\Gamma^*_\Sigma \vdash a^*_\Sigma = b^*_\Sigma : \alpha^*_\Sigma$

$th_5 \diamond$ If headConv $\mathscr{E}$ a b $\gg \tau$
then $\Gamma^*_\Sigma \vdash \tau^*_\Sigma : \mathit{type}$ and
$\Gamma^*_\Sigma \vdash a^*_\Sigma = b^*_\Sigma : \tau^*_\Sigma$

Proof
This proof proceeds by simultaneous induction on the definition of the programs and functions above.

The completeness of the algorithm can be argued following the ideas exposed in Magnusson (1995). As to the problems put forward by the subtyping relation, in particular by having its rule of transitivity in the calculus, we think we should be able to address them as in Aspinall and Compagnoni (1996). Actually, in our case the proof should be simpler due to the absence of (bounded) type variables in our calculus. The idea would be to show that the rule of transitivity is "admissible" as a program. In our case, that amounts to prove that if whTypeIncl $\mathscr{E}$ α_1 α_2 and whTypeIncl $\mathscr{E}$ α_2 α_3 succeed so does whTypeIncl $\mathscr{E}$ α_1 α_3, where α_1, α_2 and α_3 are in weak-head normal form. In addition, we should define a well- founded ordering on pairs of types, relying on the normalization property of the calculus, so as to be able to show that the program typeIncl $\mathscr{E}$ α_1 α_2 terminates for all types α_1 and α_2.

6 Conclusions and related work

Our main concern in this work was the design and implementation of 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.

https://doi.org/10.1017/S0956796899003627 Published online by Cambridge University Press

The type checking algorithm we have presented in section 4.3 is much influenced by the one presented by Magnusson (1995) for complete terms. This latter algorithm, in turn, makes use of ideas presented by Coquand (1991). As already discussed in section 2, 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 presented in Tasistro (1997) and Betarte and Tasistro (1998), 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 (1993) and Pollack (1994) concerning the type checking of PTS has also been quite influential in the development of our work.

In another direction, Coquand (1996) 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 an appropriate environment 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 a theory mechanism.

The problems posed by the type checking of languages with dependent types which incorporate mechanisms of subtyping have been studied by Cardelli (1987) and Aspinall and Compagnoni (1996). 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 by Jones *et al.* (1997). There is carried out the study of a type checking algorithm for a modified version of Martin-Löf's logical framework, proposed by Luo (1996), which introduces a notion of coercive subtyping.

Recently, Augustsson (1998) put forward a new functional language, *Cayenne*, which incorporates dependent types. Record types are there proposed as the basic mechanism to achieve modularization. However, no inclusion relation is induced by Cayenne's records.

We have reported various experiments (Betarte, 1998) on the formalization of algebraic constructions using the proof checker. Those experiments 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

https://doi.org/10.1017/S0956796899003627 Published online by Cambridge University Press

think, however, that the latter expression construct combined with subtyping might provide a uniform mechanism for hiding implementations of abstract data types. This we consider merits to be further investigated.

The system has also been used to verify an abstract version of sorting by insertion (Tasistro, 1997), 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 elsewhere (Gaspes, 1998).

Acknowledgements

The author is grateful to Cristina Cornes and Alvaro Tasistro who made helpful comments, and to the anonymous referee whose remarks and suggestions helped to improve previous versions of this paper.

References

Aspinall, D. and Compagnoni, A. (1996) Subtyping dependent types. *Proceedings of the 11th IEEE Symposium on Logic in Computer Science*.

Augustsson, L. (1997) *HBC – The Chalmers Haskell Compiler*. `http://www.cs.chalmers.se/augustss/hbc/hbc.html`.

Augustsson, L. (1998) Cayenne – a language with dependent types. *Proceedings of ICFP 98*.

Barendregt, H. (1992) Lambda Calculi with Types. In: D. M. Gabbay, S. Abramsky and T. S. E. Maibaum (eds.), *Handbook of Logic in Computer Science*, pp. 117–309. Oxford University Press.

Betarte, G. (1998) *Dependent record types and algebraic structures in type theory*. PhD thesis, PMG, Department of Computing Science, University of Göteborg and Chalmers University of Technology.

Betarte, G. and Tasistro, A. (1998) *Extension of Martin-Löf's Type Theory with Record Types and Subtyping*. In: Sambin, G. and Smith, J. M. (eds.), *Twenty-five Years of Constructive Type Theory*, pp. 21–39. Oxford University Press.

Burge, W. H. (1975) *Recursive Programming Techniques*. Addison-Wesley.

Cardelli, L. (1987) Typechecking dependent types and subtypes. In: M. Boscarol, L. C. Aiello and Levi, G. (eds.), *Proc. Workshop on Foundations of Logic and Functional Programming. Lectures Notes in Computer Science*, **306**. Springer-Verlag.

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

Coquand, Th. (1996) An algorithm for type-checking dependent types. *Science of Computer Programming 26*, pp. 167–177.

Coquand, Th., Nordström, B., Smith, J. M. and von Sydow, B. (1994) Type theory and programming. *EATCS 52*.

Dowek, G. (1993) The undecidability of typability in the lambda-pi-calculus. In: M. Bezem and J. F. Groote (eds.), *TLCA*, *Lecture Notes in Computer Science*, **664**. Springer-Verlag.

Gaspes, V. (1998) *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*, Department of Computing Science, Chalmers University of Technology.

Hutton, G. (1992) Higher-order functions for parsing. *J. Functional Programming*, **2**, 323–343.

https://doi.org/10.1017/S0956796899003627 Published online by Cambridge University Press

Jones, A., Luo, Z. and Soloviev, S. (1997) Some algorithmic and proof-theoretical aspects of coercive subtyping. In: *Proceedings of Types '96*, *Lecture Notes in Computer Science*, **1512**. Springer-Verlag.

Luo, Z. (1996) Coercive subtyping in type theory. *CSL'96, 1996 Annual Conference of the European Association for Computer Science Logic*.

Magnusson, L. (1995) *The Implementation of ALF – a Proof Editor based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution*. PhD thesis, Programming Methodology Group, Department of Computing Science, University of Göteborg and Chalmers University of Technology.

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

Martin-Löf, P. (1992) *Substitution calculus*. Talks given in Göteborg.

McKinna, J. and Pollack, R. (1993) Pure type systems formalized. In: M. Bezem and J. F. Groote (eds.), *Proc. International Conference on Typed Lambda Calculi and Applications. Lecture Notes in Computer Science*. **664**. Springer-Verlag.

Nordström, B. *The typechecking algorithm*. In preparation.

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

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

Pollak, R. (1994) Closure under alpha-conversion. *Types for Proofs and Programs: International Workshop TYPES'93*, Nijmegen. *Lecture Notes in Computer Science*, **806**. Springer-Verlag.

Röjemo, N. (1995) *Garbage collection, and memory efficiency, in lazy functional languages*. PhD thesis, Department of Computing Science, University of Göteborg and Chalmers University of Technology.

Sambin, G. and Smith, J. M. (eds). (1998) *Twenty-five Years of Constructive Type Theory*. Oxford University Press.

Severi, P. (1996) *Normalisation in Lambda Calculus and its relation to type inference*. PhD thesis, Eindhoven University of Technology.

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

Wadler, P. (1992) The essence of functional programming. *1992 Symposium on Principles of Programming Languages*, pp. 1–14.

https://doi.org/10.1017/S0956796899003627 Published online by Cambridge University Press