# The Implementation of ALF - a Proof Editor based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution

Lena Magnusson

Department of Computing Science,
Chalmers University of Technology/Göteborg University

January 2, 1995

The painter Willem de Kooning once suggested to a friend that he spend some time trying to sculpture a sphere, for example in plaster - but without using a pair of compasses or any other measuring instrument. The task is impossible, de Kooning said. You can never know if you really have created a sphere; if you start staring at it some part of it will sooner or later seem incorrect, that something has to be added or taken away. It is only by using an instrument that one can be sure, since the instrument has a social proof-value which the testimony of one's own eyes lacks; when you are certain that what you have created 'is' a sphere, since it has been measured, then the sphere has become a social phenomenon.

From *Giacometti - Moderna Museet*, exhibition catalogue no. 145 of Stockholm's Museum of Modern Art, 1977.

# Abstract

This thesis describes the implementation of ALF, which is an interactive proof editor based on Martin-Löf's type theory with explicit substitutions. ALF is a general purpose proof assistant, in which different logics can be represented. Proof objects are manipulated directly, by the usual editing operations. A partial proof is represented as an *incomplete proof object*, i.e., a proof object containing *placeholders*. A modular *type/proof checking algorithm* for complete proof objects is presented, and it is proved sound and complete assuming some basic meta theory properties of the substitution calculus. The algorithm is extended to handle incomplete objects in such a way that the type checking problem is reduced to a unification problem, i.e., the problem of finding instantiations to the placeholders in the object. Placeholders are represented together with their expected type and local context. We show that checking the correctness of instantiations can be *localised*, which means that it is enough to check an instantiation of a placeholder relative its expected type and local context. Instantiations of placeholders are either given by the user, as refinements, or is found automatically by unification. We present a unification algorithm which partially solves a unification problem, and we apply this unification algorithm to the type checking algorithm. We show that the type checking algorithm with unification and with localisation is sound, and hence when a proof object is completed, we do not have to type check the completed object again. Finally, we define two basic operations on a type checking problem, *insert* and *delete*, and we show that the basic tactics "intro" and "refine" can be defined in terms of *insert*. The *delete* operation provides a local undo mechanism which is unique for ALF. The operations are showed to preserve the validity of a partially solved type checking problem, and hence the proof editing facilities are proved to construct valid proofs.

# Acknowledgements

First and most of all, I want to thank my supervisor Thierry Coquand. He is an endless source of inspiration. His most important quality as a supervisor, I think, is that he seriously listens to all ideas and intuitions even when they are unclear and not well thought out. He has an ability of always asking the right questions which point out the essence of the idea or the gap in the reasoning. Furthermore, he is very interested in my subject, so I could not have wished for a better supervisor.

I am also grateful to Bengt Nordström for all stimulating discussions and arguments about and around ALF, and for reading and commenting this thesis. I want to thank Jan Smith, for first suggesting that I should implement a type checker, since that is how it all started. I also want to thank him for patiently reading all the version during the last period of writing this thesis, and in particular for providing the moral support I desperately needed during a week of panic. Lennart Augustsson also have read and commented the thesis, and I want to thank him for that. Also, I enjoyed his and many other night workers company during long working nights. The constructive criticism I got from Randy Pollack, I appreciate since it certainly improved the presentation of incomplete type checking.

I want to thank all ALF-users, and in particular Catarina Coquand for being the first and the most enthusiastic user of all. I have certainly enjoyed having many users of my implementation, even though from time to time some of you were a bit demanding... The total lack of bug reports and suggestions during this past year, I anticipate will not last now when I finished writing. I did appreciate the silence, though.

Pelle Lundgren and Görgen Olofsson always have a helping hand when it comes to machine related problems - thanks for the help and for answering all my stupid questions. Hans Hellström, I know you are waiting right now to get the last few pages, thank for being so patient with all my delays. I want to thank Marie Larsson for being a good friend and for all non-technical discussion, they are always welcome breaks. Staffan Truvè I want to thank for all the encouragement during these years, for being a good friend and for reading and commenting this thesis. The pictures would not have been what they are without Magnus Carlsson, who knows all the tricks.

Finally, I thank my parents and family for supporting me whatever crazy ideas I come up with, and for always being around when I needed you. Niklas, I must have been horrible to live with this past year. I hope I sometime can repay you for this, and be as tolerant and caring as you are.

I apologies to everyone I ought to thank but forgot, but now I am so tired I don't even know my name anymore.

# Contents

# Chapter 1

# Introduction

## 1.1 Background

A proof assistant is a tool which provides an environment for developing machine checked proofs and theorems. The areas of our focus are mainly formal mathematics, program verification, logic and meta theory. There are specialised proof assistants in which proofs in a particular theory can be developed, and there are general purpose proof assistants which are *frameworks* in which different theories can be defined. A general purpose proof assistant supports a *meta logic* in which particular theories can be represented as *object logics*, and then proofs can be developed within the particular theory. A framework is clearly more suitable for some theories than others, and depending on the meta logic, the representation of a particular theory is more or less straightforward. Some theories may require a lot of encoding.

ALF is a general purpose proof assistant, which is based on Martin-Löf's type theory. The basic idea behind type theory for developing proofs and programs is the Curry-Howard isomorphism between propositions and types [How80]. For instance, consider the expression

$$A \to (B \to A).$$

Interpreted as a type, it is for example the type of the $K$ combinator, and in general it is the type of a computable function which when supplied with an argument of type $A$ and an argument of type $B$, returns a value of type $A$. On the other hand, viewed as a proposition, it can be interpreted as $A \supset (B \supset A)$.

In type theory, the judgement

$$K : A \to (B \to A)$$

can either be read as the program $K$ of the function type $A \to (B \to A)$ or the proof $K$ which proves the proposition $A \supset (B \supset A)$.

The purpose of types in a programming language is to classify how objects are meant to be used, so the type of a program gives a partial specification of the program. The strength of type theory as a specification language comes from the generalisation from simply typed to *dependently typed* $\lambda$-calculus. The result type of a dependently typed function may not only depend on the type of its arguments (as in polymorphic functional programming languages such as ML and Haskell) but also on the argument itself. This is important for the expressiveness of propositions, since quantifiers such as $\forall$ and $\exists$ can be represented. Proposition can be interpreted as problems, as suggested by Kolmogorov [Kol32] already in 1932, and since a specification of a program can be seen as the problem of finding a program which satisfies the specification, type theory is both a programming language and a specification language. For instance, the specification of a sorting program which sorts a list of natural numbers can be expressed by

$$\forall l \in List(N). \exists l' \in List(N).\ Permutation(l, l') \wedge Sorted(l')$$

for suitable specifications of $Permutation$ and $Sorted$. Any program of the above type is a sorting program, but different programs may represent different sorting algorithms such as, for example, insertion sort, merge sort or quick sort. However, the program may contain parts which are computationally irrelevant, that is parts which are only present to verify that the resulting list is really a permutation of the input list and that it is sorted. There are methods to *extract* pure programs without these computationally irrelevant parts from the programs derived from the specification ([PM89]), which are provably correct with respect to their specification. Without dependent function types, the specification of a sorting program would be the type $List(N) \to List(N)$, which is not precise enough to only contain sorting programs. Since type theory is at the same time a specification language and a programming language, an alternative approach to extracting pure programs from programs satisfying specifications, is to define the pure program directly and then prove properties of this program.

In type theory, programs and proofs can be identified, so the problem of proof checking coincides with the problem of type checking. In Martin-Löf's monomorphic[1] type theory [NPS90], the typing relation is a decidable property. Clearly, this is a desirable property for the meta logic of a proof development system, since it is then possible to construct a decision procedure for proof

[1] or explicitly polymorphic

checking.

The level of assistance provided by the proof development tool can vary. On one extreme we have a proof checker, which verifies if a given object is a proof of a given proposition. At the other extreme we have an automatic theorem prover, which given a proposition tries to produce a proof of the proposition, or simply gives the answer yes or no. Clearly, a proof checker is of little or no help in the process of constructing the proof, since the proof must be supplied by the user. The advantage is that the user will have a mechanical check that all details are correct. On the other hand, when an automatic theorem prover succeeds in proving a theorem, the user may have no understanding of why the theorem holds. Even if the prover constructs a proof, it may be extremely difficult to understand. Another problem is that automated proof search in expressive languages such as type theory and higher order logics is a difficult problem, in particular for practical applications (see [BBKM93], [Dow93], [Ell89], [Pym92], [PW90], [TS94]). There is a tradeoff between a meta logic with powerful expressiveness and the possibilities of designing practical proof search strategies. The simplicity of representation ought to be a major concern. Type theory is an expressive language, which is closely related to $\lambda$-calculus and functional programming languages, and in which inductive definitions and inductive proofs are naturally represented. Therefore, we think that type theory is a suitable meta logic for a proof assistant.

The author believes that the optimal proof assistant is an *interactive* proof assistant, which is guided by the user, but is capable of proving simple problems automatically. The ideal would be for the user to perform the clever and creative steps, and for the proof assistant to fill in the gaps in-between. The user guidance is important, since finding proofs is a hard problem. Moreover, for the purpose of program extraction, we may want to find proofs which correspond to a particular algorithm. Thus, it is not only necessary for the user to interact with the proof assistant, it is also desirable.

## 1.2 Interactive proof assistants for type theory

Most interactive proof assistants today work as goal-directed state transition machines. The user states a theorem to prove, and then the state of the machine is transformed by a sequence of commands until a final state is reached, in which the theorem is proved. Basic commands correspond to inference rules in the underlying meta logic, and a successful application of such a basic command replaces the conclusion of the inference rule by the premisses of the rule. Thus, the proving process corresponds to constructing (implicitly or explicitly) a derivation of the theorem to be proved. Usually, the proving process is recorded as a proof script, which in essence is the sequence of commands used to reach the final state.

In type theory the inference rules are decorated by *proof objects*, which are $\lambda$-terms extended with constants. In an inference rule, the proof object of the conclusion is constructed from the proof objects of the premisses. Hence, the structure of a proof object of the theorem corresponds to the structure of the derivation of the theorem.

In [CKT94], it is claimed that proof objects are important for the following reasons:

- proof objects are far less dependent of the proof assistant than proof scripts,
- proof objects form a better basis for understanding and displaying the intellectual content of a proof,
- if proof objects can be built incrementally as in ALF, they provide a useful interactive feedback on what is going on in the proof.

Several modern interactive proof assistants, such as ALF, Coq [$D^+$91], HOL [GM92], LEGO [LP92] and NuPRL [Con86] construct proof objects during the proving process. Coq, HOL, LEGO and NuPRL are all tactic based proof assistants, where a tactic is a simple program which combines basic commands (inference rules) in a certain way. Users of these systems manipulate partial proof trees by applying tactics. The proof object can be extracted from the derivation, but is rarely used in the process. In ALF the proof object is more emphasised: the user manipulate the proof object directly.

Since the idea of a proof assistant is to help the user to construct formal proofs, we must be able to represent proofs which are not yet finished. When proofs are represented as derivations, a partial derivation is usually a derivation tree of which some branches are still open, i.e. all branches in the tree does not yet end in a closed assumption or an axiom. Some proof assistants have a notion of meta variables which represents sub-goals left to prove. Among these are ALF, the Constructor system [HA90], Coq, Elf [Pfe89], Isabelle [PN90] and LEGO.

In this thesis we will mainly compare ALF to Coq and LEGO, since these are all proof assistants based on type theory, they support inductive definitions and they have a notion of meta variables. Hence, these two systems are most closely related to ALF.

### 1.2.1 ALF

ALF is an interactive proof assistant based on Martin-Löf's monomorphic type theory. ALF is being developed at Chalmers ([Nor93], [CNSvS94], [MN94]) and there has been several implementations ([ACN90], [Mag91]) leading to the current implementation ([Mag92]). In ALF, proofs are treated as objects, i.e. the representation of a proof is the proof object itself together with the proposition of which it is a proof. A notion of *placeholders* represents the "holes" in the proof object which are not yet filled in. Thus the language is extended with placeholders, and placeholders are distinguished from the variables in the language. Already ancient Greek mathematicians made this distinction between a thing *sought* and a thing *given* (which may be a variable) [Mäe93]. The difference is that the sought for thing is *intended* to be filled in whereas a variable remains arbitrary. The introduction of an explicit notation for sought entities is due to Viete in 1591 [Mäe93], and it has been essential for the development of mathematics.

An incomplete proof is a proof object containing placeholders. Thus, the placeholders in a proof object specify sub-problems which will give a solution to the original problem when solutions are found for these sub-problems, that is, instantiations are found for the placeholders. Then the process of finding a solution to a problem is to successively *refine* the sub-problems until they are simple enough to be filled in directly. In ALF, users manipulate incomplete proof objects directly by editing the proof with the common editing operations such as insert, delete, copy and paste, rather than by giving commands in terms of tactics. There are only two basic operations on an incomplete proof object, the *insert* operation and the *delete* operation. We will show that the basic tactics "intro" and "refine" can both be defined in terms of the operation insert. The delete operation is a new operation, which enables the user to delete any sub-part of the proof object. Since there are dependencies between different parts of a proof, the delete operation has to be more sophisticated than simply deleting a sub-proof. In other systems we know of, global or chronological undo operations are provided, but there is no operation corresponding to local undo. The two operations are dual to each other, so the user can freely edit the incomplete proof object in an unrestricted way. Hence, the proving process become the well-known activity of editing a program (proof) of a given type (proposition). This is the reason we call ALF a *proof editor* rather than a proof assistant.

## 1.3 Thesis overview

The thesis contains a description of the implementation of ALF. We start by giving an informal presentation of type theory and describe how constant definitions and theories can be represented in this language. Then we introduce placeholders, and show how partial proofs can be represented as incomplete proof objects. The following chapter gives an introduction to ALF as a proof editing system, and a small example is carried out step by step to illustrate how a proof can be constructed by successively building a proof object.

In chapter 4 we give a formal presentation of the substitution calculus, which is a version of Martin-Löf's monomorphic type theory extended with explicit substitution. The calculus completely formalises the notion of and computation with substitutions, and hence it is much closer to an implementation then other formalisations of type theory without explicit substitution. The calculus is previously presented in [Tas93], together with semantic justifications of all the rules. In our presentation we try instead to describe the calculus from the point of view as a formalisation of an implementation. Since substitutions are part of the language, explicit substitutions give the possibility of binding terms to variables without actually performing the substitution. Hence, we can choose to perform a substitution immediately or to delay it until a later occasion. This possibility of delaying a substitution is important for computing with incomplete objects, since we cannot perform the substitution to the incomplete parts of an object, that is the placeholders, before they are completed.

Chapter 5 starts the description of the implementation, by presenting the type checking algorithm for complete terms, that is terms without placeholders. Recall that proof checking coincide with type checking, so the algorithm is the core of a proof checker. The algorithm is proved sound and complete assuming some basic meta theory of the substitution calculus, which is described in section 4.3. One of the points of this algorithm is that it can be extended to handle incomplete objects with a few modifications.

In chapter 6, placeholders are formally introduced, and the extension of the type checking algorithm is described here. Our representation of placeholders is such that it includes type and scope information of the placeholder, so that it is enough to check that an instantiation of a placeholder is correct relative this information. Hence, we can separate a placeholder and its information from the actual occurrence in the proof object, and the correctness of placeholder instantiations can be localised to the placeholder information, which is the main result of this chapter.

Instantiations of placeholders can be given by a user, but sometimes instantiations can be found automatically by unification. It was first showed in [Ell89] and [Pym92] that in $\lambda\Pi$, which is a type theory with dependent types, the well-typedness condition was also a unification problem. Then it was noticed that in systems with dependent types, unification algorithms and proof searching algorithms use the same techniques (see [Dow93], [Hag91] and [Pfe89]), and "instantiation" was proposed as the kernel of proof building systems. In chapter 7, we will present a unification algorithm which tries to find partial solutions to our unification problem. Our algorithm deals with Martin-Löf's type theory, which is a more complex system than $\lambda\Pi$ since it has other computation rules than $\beta$- and $\eta$-conversion. The algorithms presented in [Ell89] and [Pym92] are complete higher order unification algorithms, whereas our algorithm only solves first order unification problems and it leaves the difficult equations as *constraints* for future instantiations. The main result of this chapter is the soundness proof of our unification algorithm.

The next chapter shows how the unification algorithm is used in connection with proof refinement. We show that with our representation of partial proofs as incomplete proof objects, checking proof refinements and proof checking can be unified in one algorithm. Hence we do not need a separate machinery which handles incremental construction of proofs. The representation of a partial proof is the incomplete proof object together with a unification problem. A unification problem is a collection of *placeholders declarations* which gives the expected types and contexts of the placeholders, together with the constraints of the placeholders. The unification problem contains placeholders occurring in the proof object. A valid state of a partial proof is such that for any instantiation of the placeholders which satisfies the constraints, these instantiations completes the proof correctly. We also define the two basic operations *insert* and *delete* on incomplete proofs, and show that they preserves the validity of a proof state. Hence, when all placeholders of a proof object are filled in, we know that the completed object is a type correct proof, so it needs not be checked again.

Finally, we come back to ALF again, and explain how the previous algorithms are used in the proof engine of ALF, to achieve flexible ways of editing proofs. We show that our representation of incomplete proof objects provides a possibility of hiding arguments. Thus, we have a notion of implicit arguments, that is monomorphic functions can be treated as if they were polymorphic, which highly increase the readability of proof objects. The last chapter contains a short summary and some related works.

# Chapter 2

# Informal presentation of Martin-Löf's type theory

Type theory is a small functional language with a type system which allows dependent function types. The pure language contains the ordinary $\lambda$-calculus constructions abstraction, application and variables. The version of type theory we will be using is enriched with the notion of explicit substitution, and is described in more detail in chapter 4. It is possibe to extend the language with new *constant* declarations, as in most standard functional language. The purpose of a proof editor is not merely to represent programs but also to reason about them. Therefore we need a logic to represent properties of the programs. We believe that there is not one programming logic which is suitable for all kinds of reasoning we may want to do. In Martin-Löf's monomorphic type theory we can represent different logics (with various degree of coding) as well as data types and programs, since a logic is represented as a collection of constant definitions. The dependent function types give a greater expressiveness than common functional programming languages, and this is crucial for representing the universal and existential quantifiers, for example. Thus, we can represent a logic in type theory and we can reason about programs by stating and proving properties using the logic of our choice. This type theory is sometimes also referred to as Martin-Löf's logical framework.

Specifications (problems) and propositions are represented as *types*. Programs (solutions) and proofs are represented as objects in their corresponding type. Thus, a false proposition is represented by a type with no inhabitants, and an object of a type represents the proof of the proposition corresponding to that type. Axioms and rules of a logic are represented by typed constants, where the types correspond to the statements of the axioms or rules, respectively. Rules with premisses are represented by constants with functional types, where the argument types correspond to the premisses.

A proof object represents the proof of a statement. The process of proving a proposition $A$ corresponds to the process of building a proof object of type $A$. There is a close connection between the individual steps in proving $A$ and the steps in building the proof object. For instance, the act of applying a rule is done by building an application of the corresponding constant, assuming a proposition $A$ corresponds to abstracting a variable of type $A$, and the assumption is referred to by using the variable in question. Since we are interested in successively building an object of a given type, we must be able to handle *incomplete* proof objects, i.e. objects which represent incomplete proofs. Incomplete objects are represented by an object which contains *place holders*, which are temporary constant declarations, where the constant is intended to eventually be replaced by a complete object. An object which has been built by the proof editor is always meaningful in the sense that it is well typed, which means that it really represents a proof.

We will start by presenting how types, objects and definitions are formed in Martin-Löf's type theory. Thereafter we will describe how a logic is represented in the type theory and the representation of incomplete proofs. The remainder of this chapter is an extended version of a presentation of ALF earlier published in [Nor93] and [MN94].

## 2.1 Type and object formation

There are four judgement forms in Martin-Löf's type theory:

| | |
|---|---|
| $A : \mathit{Type}$ | $A$ is a type. |
| $A = B : \mathit{Type}$ | $A$ and $B$ are equal types. |
| $a : A$ | $a$ is an object in type $A$. |
| $a = b : A$ | $a$ and $b$ are equal objects in type $A$. |

In general, all judgements can be hypothetical, i.e. they may depend on a list of assumptions; a *context*. A context is a list of typed variables

$$[x_1 : A_1, \ldots, x_n : A_n]$$

where $A_i$ may depend on the previous variables $x_1, \ldots, x_{i-1}$. In the version of type theory with explicit substitution, here referred to as the substitution calculus, there are several more judgement forms, since contexts and substitutions are completely formalised as well. Here we will use the notation of type theory as it is presented in [NPS90], since it is this notation which is used in ALF.

In later chapters we will convert to the notation of the substitution calculus, since there it is more convenient for our purposes.

### 2.1.1 Type formation

There are two ways of forming ground types, and function types are built up from these ground types. We have the following three type constructors:

*Set* **formation** *Set* is a type. This is the type in which objects are inductively defined sets.

$$\frac{}{\mathit{Set} : \mathit{Type}}$$

*El* **formation** If $A$ is a set, then $\mathit{El}(A)$ is a type. The objects in this type are the objects in the (inductively defined) set $A$.

$$\frac{A : \mathit{Set}}{\mathit{El}(A) : \mathit{Type}}$$

We will often omit the type constructor *El* and simply write $A$ when it is clear from the context that we refer to the type $A$ rather than the object $A$ (in type *Set*).

**Fun formation** The dependent function type can be constructed from a type $A$ and a family of types $B$ depending on a variable $x$ in type $A$. A family of types is a type which is parameterised over a variable, and for each substitution of that variable we get a type. The function constructor is a binding operation, so every free occurrence of $x$ in the family $B$ becomes bound in the type $(x : A)B$.

$$\frac{A : \mathit{Type} \quad B : \mathit{Type}\ [x : A]}{(x : A)B : \mathit{Type}}$$

When we have a function $f$ in the type $(x : A)B$, $f$ can be applied to any object $a$ in type $A$. The result is the object $f(a)$ in the type $B\{x{:=}a\}$, where $\{x{:=}a\}$ denotes the substitution of $a$ for $x$. The substitution can at this point be seen either as a meta operation as in the traditional type theory, or as a true construction in the language as in the substitution calculus. We will sometimes write $(A)B$ when $B$ does not depend on $A$, i.e. the non dependent function type, and $(x_1{:}A_1\,;\,\ldots\,;x_n{:}A_n)B$ instead of $(x_1{:}A_1)\cdots(x_n{:}A_n)B$. For readability, we write $(a, b{:}A)B$ for $(a{:}A\,;b{:}A)B$.

Note that the only built in types are the ground type *Set* and functions built up from *Set*. We can only construct types by the type constructor *El* first after some ground type is defined. We get new types by defining a constant in the type *Set*, such as for instance the data type of natural numbers

$N : Set$

with the two constructors

$0 : El(N)$ which we will write $0 : N$
$s : (El(N))El(N)$ or simply $s : (N)N$

which defines the objects in the type $N$. The type $(N)N$ corresponds to $N \to N$ in usual functional programming languages. The type of lists, which for any set $A$ constructs a type of lists of $A$, is defined by the set constructor

$List : (A{:}Set)Set$

with the list constructors

$nil : (A{:}Set)List(A)$
$cons : (A{:}Set\,;\, a{:}A\,;\, l{:}List(A))List(A)$.

The type of the constructor specifies the intended meaning of the constructor. For instance, the constructor *cons* is a function which given an arbitrary set, an element and a list of the corresponding set, constructs a new list element. In the list definition we can see that the monomorphic type information (the parameter $A : Set$) occurs in the type of each constructor as an explicit argument. It can be contrasted with the polymorphic list type in standard ML, for instance, where we have an implicit binding of the type variables (on the outermost level). The monomorphic type information has a tendency to clutter the objects with redundant information, and therefore we have a way of hiding these kinds of arguments in ALF. The monomorphic type information can almost always be deduced automatically by ALF, and with the hiding facility we need neither see nor fill in that information, and it is therefore not a problem in practice.

### 2.1.2 Object formation

Objects are built up from variables and constants by abstraction and application. Variables must be declared in the context and constants in the *environment* (the table of all current constant definitions). The formation rules correspond to the constructions:

**Assumption** As mentioned, all judgements are in general relative to a context, and we will identify non-hypothetical judgements with judgements relative to the empty context. Thus, if $\Gamma$ denotes a context, we have the following rule of assumption

$$\frac{}{x : A \quad \Gamma}\; {\scriptstyle x{:}A \,\in\, \Gamma}$$

which says that if we have assumed a variable $x$ of type $A$, we can derive that $x$ is of type $A$, i.e. we can use our assumption. Note that we may assume any variable in the context, not only the last one. This rule can be derived in a system where only the last variable in a context may be assumed, by successively performing thinning (extending the context with new assumptions) after the variable is assumed. In such a system where only the last variable can be assumed, there are no side conditions as in the above rule.

**Constant** Constants must be defined before they are used, which means that they must occur in the current environment of constant definitions. We denote the constant environment by $\Sigma$, so the following rule says that we may use the constants in the environment:

$$\frac{}{c : A}\; {\scriptstyle c{:}A \,\in\, \Sigma}$$

The above rule is an informal rule and we will explain later in section 9.1 how constants can be introduced and what a valid environment is.

**Abstraction** If $b$ is an object of type $B$ which depends on the variable $x$ of type $A$, we can form the function $[x]b$ (usually denoted $\lambda x.b$) of type $(x{:}A)B$. All free occurrences of $x$ become bound in both $b$ and $B$. If $b$ depends on several assumptions, we can only create the abstraction with respect to the last variable, since $x$ is discharged from the list of assumptions and otherwise we may create an invalid context. For instance, we can not create the abstraction $[x]b$ from $b$ if the object is defined in the context $[x : A, y : B(x)]$, since $[y : B(x)]$ is not a valid context.

$$\frac{b : B \quad [\Gamma, x : A]}{[x]b : (x{:}A)B \quad \Gamma}$$

We will write $[x_1, \ldots, x_n]b$ instead of $[x_1] \cdots [x_n]b$ for a function with more than one argument, to improve readability.

**Application** If $f$ is a function (an object of function type), we can apply the function to an object of the appropriate type:

$$\frac{f : (x{:}A)B \quad \Gamma \qquad a : A \quad \Gamma}{f(a) : B\{x{:=}a\} \quad \Gamma}$$

The notation $B\{x{:=}a\}$ means that $a$ is substituted for $x$ in $B$.

For instance, if we have defined the type of natural numbers and lists from before in an environment $\Sigma$, we can construct the object which is the list containing *0* as its single element

$$cons(N, 0, nil(N)) : List(N)$$

and we can form the function

$$[x]s(s(x)) : (N)N$$

which adds 2 to it argument.

## 2.2 Definitions

As mentioned, the strength of the language comes from the possibility of introducing new constants. We can introduce constants representing all the usual inductive data types such as natural numbers, lists, trees etc. Logical connectives such as disjunction and implication or quantifiers are represented as constants as well. We can also define sets which represent inductively defined predicates. Objects in these families of sets represent the different ways of justifying the corresponding predicate, that is the objects are themselves proof objects.

We will distinguish between *primitive* constants and *defined* constants, where defined constants can be *explicitly* or *implicitly* defined.

### 2.2.1 Primitive constants

Primitive constants (also called constructors) have a type but no definition. They are *canonical* objects, and their meaning is justified by the semantics of the theory. An inductively defined set is represented by a set constructor (corresponding to the formation rule) and a collection of constructors of the set (corresponding to the introduction rules). We have already seen examples of the data types for natural numbers and lists where $N$ and *List* are the set constructors, *0* and $s$ are the constructors for the set $N$ and *nil* and *cons* are the constructors which build up objects in the set *List*. The identity predicate can be represented as a set depending on two elements of a given type, which has only one constructor representing reflexivity. Thus, the objects in the set $Id(A, a, b)$ are proof objects denoting that $a$ and $b$ are identical objects in the set $A$:

$$Id : (A{:}Set\,; a{:}A\,; b{:}A)Set$$

$refl : (A{:}Set\,;a{:}A)Id(A,a,a)$

We will use the convention to write primitive definitions in the form

$$\begin{array}{l} A : Set \\ \quad c_1 : A_1 \\ \qquad \vdots \\ \quad c_n : A_n \end{array}$$

where $A$ is the defined set and $c_1, \ldots, c_n$ are the constructors of the set.

### 2.2.2 Explicitly defined constants

Defined constants have a type and a definition. Explicit constants have no intrinsic meaning, they are simply abbreviations such as

$1 = s(0) : N$
$consN = cons(N) : (n{:}N\,; l{:}List(N))List(N)$

where the latter is the cons constructor specialised to natural numbers. An explicit constant computes to its definition in one step, which is always an object of the same type as declared for the constant. It can be checked to be meaningful (type correct) inside the theory. This step of computation will often be referred to as *unfolding*.

### 2.2.3 Implicitly defined constants

The definition of an implicitly defined constant is a collection of *pattern matching equations* following the functional language tradition. Implicit constants may be recursively defined. ALF generates an exhaustive, non-overlapping collection of patterns, which is an algorithm developed by Thierry Coquand [Coq92]. Since the collection of pattern equations are generated this way, we are guaranteed that for any closed, well-typed object which is an implicit constant applied to all its arguments, there is exactly one pattern which matches the object. Exhaustiveness guarantees that at least one pattern matches and the non-overlapping requirement that at most one pattern matches.

Since in a proof editor we are not mainly interested in computing programs but rather in reasoning about programs, we will have to deal with objects which are *not closed*, i.e. objects which depend on free variables. The following example illustrates the need for computing with open objects. We can define the addition function by

$add : (N\,;N)N$

$$add(x, 0) = x$$
$$add(x, s(y)) = s(add(x, y)).$$

Assume we want to show that associativity holds for addition, which means we want to show that

$$Id(N, add(add(m, n), k), add(m, add(n, k)))$$

holds for *arbitrary* $n$,$m$ and $k$, i.e. $n$,$m$ and $k$ are free variables. If we do induction on $k$, we need to show the two cases

$$(1)\ Id(N, add(add(m, n), 0), add(m, add(n, 0)))$$
$$(2)\ Id(N, add(add(m, n), s(k')), add(m, add(n, s(k'))))$$

where (1) is solved directly by reflexivity, since

$$add(add(m, n), 0) = add(m, n), \text{ and}$$
$$add(m, add(n, 0)) = add(m, n)$$

by the first pattern equation (1) which in the latter case is applied to the second argument. In (2) we see that

$$add(add(m, n), s(k')) = s(add(add(m, n), k')), \text{ and}$$
$$add(m, add(n, s(k'))) = add(m, s(add(n, k'))) = s(add(m, add(n, k')))$$

by repeated use of the second pattern equation, and (2) can be proved by a congruence rule and the induction hypothesis.

On the other hand, any closed instance of the associativity property is proved directly by reflexivity, since both sides of the identity will compute to the same natural number. For open objects, such as $add(m, z)$ where $z$ is a variable, we have an object which is neither *canonical*, nor can be computed any further since no pattern equation applies. A canonical object is an object which is built up by constructors only. Thus, only closed objects can be guaranteed to be reduced by a pattern equation.

Naturally, it is not enough to have exhaustive and non-overlapping patterns to guarantee a well-defined computation to a canonical object, the definition must also be well-founded, to assure termination. At present, this is not checked in ALF, and it is mainly since ALF is still an experimental system. This matter will be discuss more in section 9.1.

Implicit constants defined by pattern-matching can also be used to prove properties. For instance, we can prove symmetry of the identity predicate defined above, by the definition

$$Id\_symm : (A{:}Set\,;\, a, b{:}A\,;\, hyp : Id(A, a, b))Id(A, b, a)$$
$$\quad Id\_symm(A, a, \_, refl(\_, \_)) = refl(A, a)$$

The "_" sign denotes arguments which are redundant, since they are uniquely determined from the remaining information. This is explained further below.

The definition is read in the following way: First, the type of $Id\_symm$ states the property to be proven. Thus, given a set $A$, two objects $a$ and $b$ in $A$ and a proof that $a$ and $b$ are identical ($hyp$ represents the hypothetical proof), then we can conclude that also $b$ and $a$ are identical.

To prove the property $Id\_symm$, we need to construct an object in the type $Id(A, b, a)$. The proof is done by case analysis (pattern-matching) on the variable $hyp$, i.e. we analyse the possible proofs for $Id(A, a, b)$. The only possible canonical proof is of the form $\mathit{refl}(A, a)$, since $\mathit{refl}$ is the only constructor of the set. But if $\mathit{refl}(A, a)$ is the proof of (has the type) $Id(A, a, b)$, then $b$ must be the same as $a$ since the type of $\mathit{refl}(A, a)$ is $Id(A, a, a)$ which is the same type as $Id(A, a, b)$ if and only if $a = b$. Hence, $\mathit{refl}(A, a)$ also proves (has the type) $Id(A, b, a)$.

The underscores in the pattern denote arguments which are *uniquely determined* by the type of the constant $Id\_symm$ and the remaining arguments. In the example above the full pattern would be

$$Id\_symm(A, a, a, \mathit{refl}(A, a))$$

or

$$Id\_symm(A, a, a_1, \mathit{refl}(A_1, a_2)), \text{ where } A = A_1 \text{ and } a = a_1 = a_2.$$

However, for any well-typed object $Id\_symm(A, a, a_1, \mathit{refl}(A_1, a_2))$, we know that these equalities must hold due to the type of $Id\_symm$, or else the object would not be well-typed. Therefore, these equalities are guaranteed to hold by type-correctness, and need not be checked again. In that sense, these arguments are redundant information and motivates the choice of the underscore sign. All patterns can be made linear with the use of underscores, which is an advantage in the matching algorithm since no equality checks are needed after the bindings of pattern variables are established. This is also the main motivation of requiring linear patterns in most functional languages which support pattern matching.

## 2.3 The representation of theories

A theory is a collection of constant definitions. Primitive definitions are mainly used to represent inductive datatypes, logical connectives and inductively defined predicates.

### 2.3.1 Datatypes and logical constants

For example, the datatype *List* which we have seen in the previous section is explained by the two schematic rules

$$\frac{}{\mathit{nil} : \mathit{List}(A)} \qquad \frac{a : A \quad l : \mathit{List}(A)}{\mathit{cons}(a, l) : \mathit{List}(A)}$$

where $A$ is any set. As we have seen, these two rules are represented as two constructors of the datatype *List*. The type of the *List* constant corresponds to how the datatype is formed, i.e. the formation rule, whereas the constructors correspond to the ways an element in the datatype is introduced, i.e. the introduction rules:

$$\begin{array}{l} \mathit{List} : (A{:}\mathit{Set})\mathit{Set} \\ \quad \mathit{nil} : (A{:}\mathit{Set})\mathit{List}(A) \\ \quad \mathit{cons} : (A{:}\mathit{Set}\,; a{:}A\,; l{:}\mathit{List}(A))\mathit{List}(A) \end{array}$$

Note that since we are in a monomorphic setting, all schematic parameters become arguments to the constants. Similarly we can define a (complete) binary tree by the introduction rules

$$\frac{a : A}{\mathit{leaf}(a) : \mathit{BinTree}(A)} \qquad \frac{\mathit{left} : \mathit{BinTree}(A) \quad a : A \quad \mathit{right} : \mathit{BinTree}(A)}{\mathit{node}(\mathit{left}, a, \mathit{right}) : \mathit{BinTree}(A)}$$

again for any type $A$. The datatype of these binary trees is represented as

$$\begin{array}{l} \mathit{BinTree} : (A{:}\mathit{Set})\mathit{Set} \\ \quad \mathit{leaf} : (A{:}\mathit{Set}\,; a{:}A)\mathit{BinTree}(A) \\ \quad \mathit{node} : (A{:}\mathit{Set}\,; \mathit{left}{:}\mathit{BinTree}(A)\,; a{:}A\,; \mathit{right}{:}\mathit{BinTree}(A))\mathit{BinTree}(A) \end{array}$$

which says for instance that *node* is a constructing function which takes four arguments, the first being the parameter $A$ and the other three the premisses of the rules above.

Thus, a general rule of the form above where even implicit premisses are present is directly translated into a constant of function type, where each argument of the function corresponds to one premiss of the rule.

In a natural deduction setting, where a true proposition $A$ is denoted by $a : A$ (we have a proof of the proposition) and a well-formed proposition is denoted by $A$ prop, the logical connective $\vee$ is explained by the rules

$$\frac{A \text{ prop} \quad B \text{ prop}}{A \vee B \text{ prop}} \qquad \frac{a : A \quad B \text{ prop}}{\mathit{inl}(a) : A \vee B} \qquad \frac{b : B \quad A \text{ prop}}{\mathit{inr}(b) : A \vee B}$$

The first rule is the formation rule and the other two the introduction rules, where *inl* and *inr* denotes the left and right injection, respectively. In Martin-

Löf's type theory propositions and sets are identified, so we will use $A : Set$ to express that $A$ is a proposition. Thus, a constant $\vee$ can be defined to represent the $\vee$ connective by the primitive definition

$$
\begin{array}{l}
\vee : (A, B{:}Set)Set \\
\quad inl : (A, B{:}Set\,; a{:}A) \vee (A, B) \\
\quad inr : (A, B{:}Set\,; b{:}B) \vee (A, B)
\end{array}
$$

In a similar manner we can define constants representing the quantifiers $\forall$ and $\exists$. Consider the rule of $\forall$-introduction (to the left) and the same rule decorated by proof objects (to the right)

$$
\frac{\begin{array}{c}[x : A] \\ \vdots \\ B(x)\end{array}}{\forall x{:}A.B(x)}
\qquad\qquad
\frac{\left.\begin{array}{c}[x : A] \\ \vdots \\ b(x) : B(x)\end{array}\right\} \; b : (x{:}A)B(x)}{\forall_I(b) : \forall x{:}A.B(x)}
$$

where the hypothetical proof of $B(x)$ depending on the assumption $x : A$ is represented as a function $b$ which for every element in $A$ produces a proof of $B(a)$. Thus the $\forall_I$ constructor will take a meta-logic *function* $b$ as argument and this way we can benefit from the function formation and application which is already implemented in the meta logic. The corresponding definition in ALF would be

$$
\begin{array}{l}
\forall : (A{:}Set\,; B{:}(A)Set)Set \\
\quad \forall_I : (A{:}Set\,; B{:}(A)Set\,; b{:}(x{:}A)B(x))\forall(A, B)
\end{array}
$$

Note that we need the dependent function type for representing the quantifiers, since the proposition $B(x)$ may depend on the argument $x$.

### 2.3.2 Inductively defined predicates

We can use primitive definitions to represent inductively defined predicates. Here, the constructors correspond to the different ways of justifying the predicate, and thus elements in these sets are proof objects of the predicate. For instance, we can define the relation $n < m$ as a predicate over two natural numbers by the rules

$$
\frac{}{0 < s(n)} \qquad\qquad \frac{n < m}{s(n) < s(m)}
$$

which is represented as a constant $Lt$ of type $(n, m : N)Set$ with two constructors corresponding to the rules above (with all typing information stated explicitly):

$Lt : (n, m{:}N)Set$
  $Lt0 : (n{:}N)Lt(0, s(n))$
  $Lts : (n, m : N\,; h{:}Lt(n, m))Lt(s(n), s(m))$.

A predicate can also be defined in objects of already known predicates. We could either give a constant definition for the relation $\leq$ very similar to the one for $<$, or we can define it as an abbreviation of $(n < m) \vee (n = m)$, which in ALF would also be an abbreviation

$Lte = [n, m] \vee (Lt(n, m), Id(N, n, m)) : (n, m : N)Set$

Another example is the predicate stating that an element is a member of a list, which can be defined by the rules

$$\frac{}{Mem(a, cons(a, l))} \qquad\qquad \frac{Mem(a, l)}{Mem(a, cons(b, l))}$$

Considering the obvious typing constraints of $a$ and $l$, we get the following ALF definition:

$Mem : (A{:}Set\,; a{:}A\,; l{:}List(A))Set$
  $Mem1 : (A{:}Set\,; a{:}A\,; l{:}List(A))Mem(A, a, cons(a, l))$
  $Mem2 : (A{:}Set\,; a, b{:}A\,; l{:}List(A)\,; h{:}Mem(A, a, l))Mem(A, a, cons(b, l))$

### 2.3.3 Functions and elimination rules

Functions and elimination rules are often represented as implicit constants. Functions are defined in the same way as in an ordinary functional language, where the equations are computation rules defining the function. We have already seen the addition function, and another simple example is the *append* function which takes two lists as arguments and returns the lists concatenated.

$append : (A{:}Set\,; l1, l2 : List(A))List(A)$
  $append(A, nil(\_), l2) = l2$
  $append(A, cons(\_, a, l1), l2) = cons(A, a, append(A, l1, l2))$

Elimination rules are also represented as implicit constants, where the equations are the corresponding contraction rules. For datatypes, elimination rules can be seen as a general rule of structural induction over the datatype. Consider for example the induction schema of natural numbers

$$\frac{P(0) \qquad \begin{matrix}[x : N, P(x)] \\ \vdots \\ P(s(x))\end{matrix}}{P(n)}$$

where $P$ is any predicate over natural numbers, i.e. $P$ is of type $(N)Set$. Thus, we get the translation to a constant *natrec* by representing the rule above as the type of *natrec*, and the equations show how a closed proof of $P(n)$ can be transformed into a canonical proof by the contraction rules:

$$\begin{array}{l} natrec : (P{:}(N)Set\,;\, d{:}P(0)\,;\, e{:}(x{:}N\,;\, h{:}P(x))P(s(x))\,;\, n{:}N)P(n) \\ \quad natrec(P, d, e, 0) = d \\ \quad natrec(P, d, e, s(n)) = e(n, natrec(P, d, e, n)) \end{array}$$

Functions can also be defined in terms of the elimination rules. For instance, addition can be defined by *natrec* in the following way

$$add = [n, m]natrec([n]N, n, [m', h]s(h), m) : (n, m : N)N$$

which reads as follows; the induction is over $m$ so the second argument of *natrec* is the result when $m$ is $0$, which means $n$. In the other case when $m = s(m')$ we have the third argument which is a function of $m'$ and $h$, where $h$ is the result of the previous induction step i.e. the result of $add(n, m')$, and gives as result $s(h)$.

Thus, a function or a proof can either be an implicit constant or an explicit constant as the one above, that is it can either be defined directly by pattern matching or by an elimination rule. The two approaches differ in that elimination rules represent general schemas of primitive recursion as well as structural induction over inductively defined sets. They are justified by reflection on the constructors of the corresponding sets. Once an elimination rule is defined, any primitive recursive function or inductive proof can be defined in terms of this rule, as an explicitly defined constant. The advantage with this approach is that structural induction and primitive recursion is justified once and for all in the elimination rule. When a function or a proof is defined directly by pattern matching, the reflection on the corresponding constructors is performed for each particular proof or function. The general pattern matching approach is not present in Martin-Löf's type theory, and proof theoretically the two approaches are not equivalent [Hof93].

Another example of an elimination rule is the rule corresponding to the $\vee$

connective:

$$\frac{\begin{array}{ccc}[a:A] & [b:B] & \\ \vdots & \vdots & \\ P(inl(a)) & P(inr(b)) & h:\vee(A,B)\end{array}}{P(h)}$$

which is translated into the implicit constant

$$\begin{array}{ll}\vee\_elim: & (P:(\vee(A,B))Set\,;\ f:(a{:}A)P(inl(a))\,; \\ & \ \ g:(b{:}B)P(inr(b))\,;\ h:\vee(A,B) \\ & )\ P(h) \\ \multicolumn{2}{l}{\ \ \vee\_elim(P,f,g,inl(a)) = f(a)} \\ \multicolumn{2}{l}{\ \ \vee\_elim(P,f,g,inr(b)) = g(b)}\end{array}$$

where the parameters $A$ and $B$ are omitted for readability.

Elimination rules can be derived automatically from the introductions rules by an elimination schema, as is shown in [CP90],[Dyb91]. These kinds of schemas are implemented in Coq and LEGO. In ALF, elimination rules can be defined by using pattern matching, but they are not derived automatically, instead we have implemented the general mechanism of defining constants by pattern matching [Coq92].

### 2.3.4 Proofs and theorems

Theorems are represented as either an implicit constant definition or an explicit constant definition. The type of the constant states the property to be proven, and the definition represents the proof. When we want to apply a previously proven theorem, we simply apply the name of the constant, and if the theorem has any assumptions (i.e., the constant is of function type), we must apply the constant to proofs of the assumptions.

With the above definition of $<$, we may have to prove that

$$n < s(n)$$

for instance. This could be done in either of the two ways. The case with an implicit definition we would get

$$\begin{array}{l} n\_lt\_sn1 : (n:N)Lt(n,s(n)) \\ \quad n\_lt\_sn1(0) = Lt0(0) \\ \quad n\_lt\_sn1(s(n_1)) = Lts(n_1, s(n_1), n\_lt\_sn1(n_1)) \end{array}$$

Since the recursive call is clearly structurally smaller, this is a proof of the property. The other case is when we prove properties with explicit constants, and then we use the elimination rules corresponding to the case analysis done in the patterns. In this case

$$n\_lt\_sn2 : (n : N)Lt(n, s(n))$$
$$n\_lt\_sn2 = [n_1]natrec([n]Lt(n, s(n)), Lt0(0), [n_2, ih]Lts(n_2, s(n_2), ih), n_1)$$

Explicit constants may not be recursive, but according to the elimination rule we only have to show the induction step under the assumption that the property holds for the smaller argument ($ih$ denotes a proof of the induction hypothesis, i.e. a proof of $Lt(n_2, s(n_2))$).

Usually, the proof done by pattern matching is much shorter than the corresponding proof with elimination rules. One reason is that "impossible" cases can be eliminated when the matching is done over several variables. Consider for example the proof of transitivity for $<$, i.e. an object in the type

$$(n, m, k : N\,; h{:}Lt(n, m)\,; h_1{:}Lt(m, k))Lt(n, k)$$

This can be done by analysing the possible cases of $n, m$ and $k$, but then we must handle "impossible" cases such as when $m$ is $0$. Obviously there cannot exist a proof of $n < 0$. If we instead analyse the possible cases of the proofs of $n < m$ and $m < k$ (the variables $h$ and $h_1$), we can see that $m$ cannot be 0 since then these proof objects can not exists. This is detected by the pattern generation algorithm, and these cases are eliminated. Thus, the proof only consists of the cases

(1) $0 < s(m_1)$ and $s(m_1) < s(k_1)$ implies $0 < s(k_1)$, and
(2) $s(n_1) < s(m_1)$ and $s(m_1) < s(k_1)$ implies $s(n_1) < s(k_1)$

which correspond to the two pattern equations

$$Lt\_trans(\_, \_, \_, Lt0(n_1), Lts(, k_1, ih)) = Lt0(k_1)$$
$$Lt\_trans(\_, \_, \_, Lts(m_1, n_1, ih_1), Lts(\_, k_1, ih_2)) =$$
$$\qquad Lts(n_1, k_1, Lt\_trans(n_1, m_1, k_1, ih_1, ih_2))$$

This concludes the description of representation of theories, and in the next section we will describe how these proofs can be built up incrementally.

## 2.4 The representation of incomplete objects

A temporary constant declaration is a name together with an expected type and a local context. It is temporary since it is intended to be replaced by an object of the expected type which may contain variables from the local context. The constant itself is referred to as a placeholder. It is denoted by a prefix '?'

to indicate that it stands for a yet unknown object, as in

$$?_0 : \alpha \quad \Gamma$$

where $\alpha$ is the expected type and $\Gamma$ the local context of the placeholder $?_0$. An incomplete proof object is an object which contains placeholders.

We build proof objects (top-down) in ALF, by successively replacing a placeholder with a (possibly incomplete) object, which is required to match the type of the placeholder. The placeholder $?_0$ can be replaced by the following constructs, corresponding to the different ways of forming an object:

**A constant** The placeholder may be replaced by a constant $c$, if $c$ is defined in the environment of constant definitions, and $c$ is of type $\alpha$.

**A variable** The placeholder may be replaced by a variable $x$, if $x$ occurs in the local context $\Gamma$ and is of proper type, i.e. is equal to $\alpha$.

**An abstraction** $?_0$ may also be replaced by an abstraction $[x]?_1$, if $\alpha$ is a function type $(x : A)B$, $x$ is of type $A$ and $?_1$ of type $B$ in the context $[x : A]$. The type and the local context of the new placeholder $?_1$ can be computed from the type $(x : A)B$, since the type must be $B$ and the local context is the one of $?_0$ extended by the variable declaration $x : A$. This information need not be given to ALF, and therefore the new placeholder

$$?_1 : B \quad [\Gamma, x : A]$$

with its declaration is generated automatically.

**An application** The placeholder $?_0$ of type $\alpha$ may be replaced by an application

$$b(?_1, \ldots, ?_n)$$

where $b$ is a variable or a constant, and $b$ is of type

$$b : (x_1 : A_1, \ldots, x_n : A_n)B.$$

Thus, the problem of finding a proof object for $?_0$ is reduced to the subproblems of finding proof objects for

$$\begin{array}{l} ?_1 : A_1 \\ ?_2 : A_2\{x_1 := ?_1\} \\ \vdots \\ ?_n : A_n\{x_1 := ?_1, \ldots, x_{n-1} := ?_{n-1}\} \end{array}$$

The number of new placeholders can be computed from the type of $b$ and the expected type of the replaced placeholder. Their expected types and local context can also be computed, provided that $b$ is a constant or a variable from the local context of $?_0$. In this case, the type of the constant or variable is known. If $b$ is an abstraction, we can not compute the type of $b$. The reason is the generalisation of type theory compared to simply typed $\lambda$-calculus, since in type theory functions may be dependently typed. This result is shown in [Sal88].

Therefore, we require the *head* of the application to be a variable or constant. Also, since the head of an application determines how the original problem is divided into sub-problems, it is natural to try to solve the problem by a rule (a constant) or by an assumption (a variable). However, not every constant or variable can be used to refine the placeholder, since its resulting type (i.e. its conclusion) must match the type of the placeholder. Therefore, for this replacement to be correct, we need to make sure that the *constraint*

$$\alpha = B\{x_1 := ?_1, \ldots, x_n := ?_n\}$$

holds.

For example, let us consider the associativity for addition from the previous section once again, where we have performed the case analysis on the last variable. Then we are in the situation where we have two placeholders

$$?_1 : Id(N, add(add(m,n),0), add(m, add(n,0))) \quad [n,m:N]$$
$$?_2 : Id(N, add(add(m,n),s(k')), add(m, add(n,s(k')))) \quad [n,m,k':N]$$

which must be filled in. The first case is solved directly by reflexivity, so we refine $?_1$ with the constant *refl* which means that two new placeholders

$$?_A : Set \quad [n,m:N]$$
$$?_a : ?_A \quad [n,m:N]$$

are created, and the placeholder $?_1$ is replaced by

$$refl(?_A, ?_a)$$

which created the type checking constraint

$$Id(?_A, ?_a, ?_a) = Id(N, add(add(m,n),0), add(m, add(n,0))) \quad [n,m:N].$$

The constraint can be simplified since $Id$ is a constructor which is assumed to be one-to-one, to the following simpler constraints

$$?_A = N \quad [n,m:N]$$
$$?_a = add(add(m,n),0) \quad [n,m:N]$$
$$?_a = add(m, add(n,0))) \quad [n,m:N].$$

These equations hold if

$$?_A = N$$
$$?_a = add(m,n)$$

since both simple constraints involving the placeholder $?_a$ above can be simplified to $add(m,n)$. In this simple situation, the placeholders will be instantiated automatically.

To conclude, an incomplete proof is represented by a partial definition, a list of placeholders with their expected type and local context and a list of constraints. The constraints then become a unification problem, i.e. we want to find instantiations of all placeholders such that all constraints hold. However, since placeholders may be of higher order it is not always possible to solve the unification problem completely, since we might have equations such as

$$?_0(a_1, \ldots, a_n) = b$$

which do not have a unique solution. Therefore, the simple constraints are solved whereas the others are kept as constraints and future refinements are checked not to violate these constraints. Thus, incomplete proof objects are only correct relative to unsolved constraints.

# Chapter 3

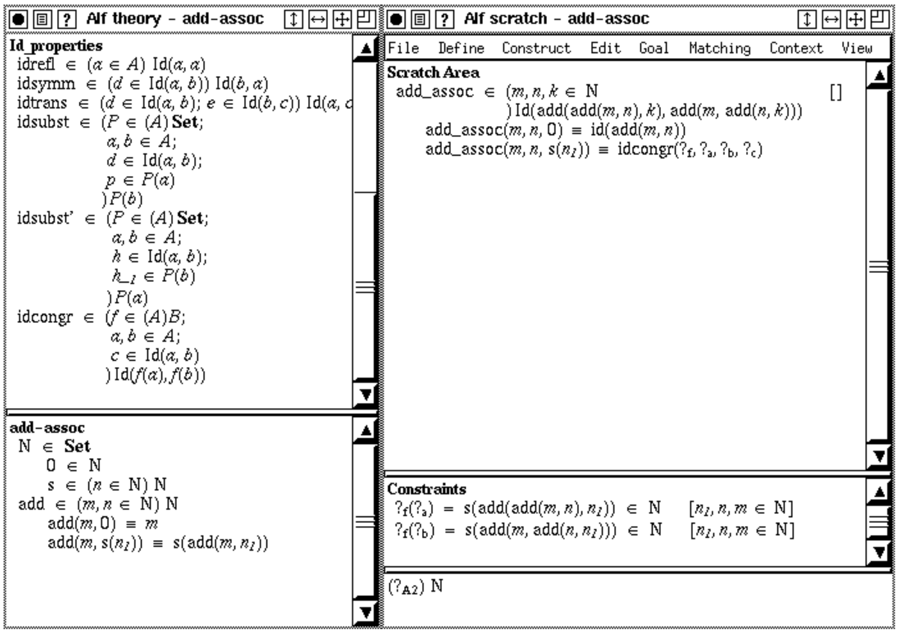
# Introduction to the proof editor ALF

ALF is an *interactive, structure-oriented proof editor* based on Martin-Löf's *monomorphic type theory* with explicit substitution. The system consists of a window interface and a proof engine. The notion of a proof object is important, since all operations are performed *directly* on the proof object. The interactiveness means that an object is built up stepwise, and that each step is checked immediately. Objects are manipulated by ordinary editing commands such as insertion, copying and deletion. Only syntactically correct entities can be selected and operated on, and thus the object will always maintain the structure of a proper proof object, following the tradition in [TR81] and [DGKLM84]. Inserted sub-objects are checked to be syntactically correct as well as of the proper type. The user interface of ALF administrates the commands of the user, such as mouse clicks and menu choices. The object is altered by selecting a part of it, thereafter a command is chosen from one of the menus. The interface invokes the proof engine with the chosen command and the current selection, and the proof engine replies with the changes of the object in question. Thus, the result of the operation is immediately shown on the screen, or an error message is reported if the operation is illegal. The user may work on several proof objects simultaneously, and there are few restrictions on what order operations must be performed. The aim is to have a system where proof objects can be built and changed as flexibly as possible.

## 3.1 The system

The proof editor ALF consists of two parts - the proof engine and the user interface. They are two separate processes, and are implemented in different programming languages. The proof engine is implemented by the author, and is written in Standard ML [MTH90]. The user interface is implemented by Johan Nordlander at Chalmers, and it is developed using the tool kit InterViews for window-based applications.

The purpose of ALF is to interactively prove theorems or construct programs. Since ALF is a framework for different logics, the particular theory of interest must be specified, before the statement to be proved can be expressed. However, theories as well as theorems are constant definitions, so the same facilities which are used to edit proofs of theorems are also used to construct theories. Previously defined theories can be loaded and used. All new definitions may depend on constants declared in the current theory. The user interface consists of two windows, one for the (completed) theory definitions and one for definitions which are not yet completed. All editing of objects take place in the latter window which is called the *scratch area*. When all placeholders in a scratch area definition are completely filled in, the definition is complete and can be moved to the theory. Below is a snapshot from the construction of the associativity of addition from the previous section:



A session in ALF will be presented in the next section, where the different parts of the windows will be explained.

The window interface provides a graphical representation of the theory and the scratch area. It is structure-oriented, so only proper sub-parts of the different syntactic categories can be selected and edited. The window interface makes the selection of placeholders, pattern variables and sub-objects very natural, and the only applicable operations are selectable from the menus. Without an interface, especially the selection of sub-objects and pattern variables become rather awkward and cumbersome.

All layout features are naturally parts of the interface. When an operation is demanded by the user, the user interface orders the proof engine to check and perform the operation in question, and the proof engine replies with the new changes. These changes invoke the updating of the windows, and the operation is completed.

The user interface provides the possibility of actually hiding arguments, but these arguments can only be ignored by the user if they can be inferred by the proof engine. The interface record information about hidden arguments, for displaying the objects with these arguments hidden. The user chooses for each constant which arguments should be hidden when that constant is used. The hiding can be switched on and off.

The user demands various actions of ALF by using the mouse and menus of the interface, or by using a text editing window for input from the keyboard. The communication between the processes is governed by the interface, which sends commands to the proof engine, as is shown in figure 3.1 below.

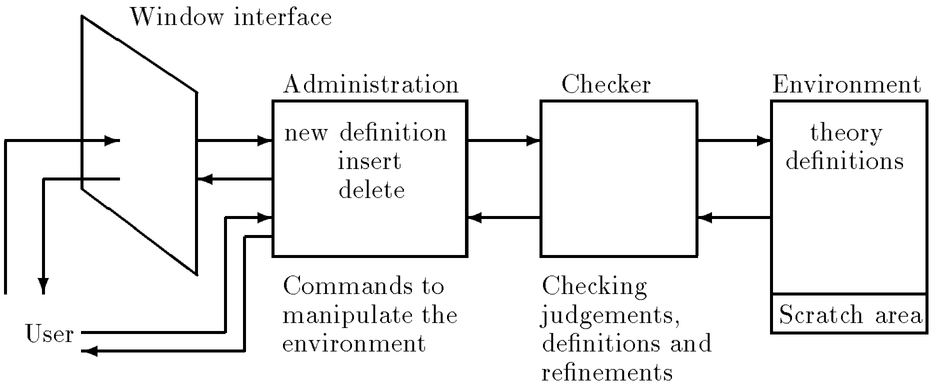



Figure 3.1: System overview

The proof engine performs the requested command, and sends back the changes in the state of the *environment*. The environment contains all the constant definitions in the theory and in the scratch area. Positions in the graphical representation of the definitions in the interface is communicated to the proof engine via search paths to the position in question.

Most of the remaining part of this thesis will describe the proof engine, and in particular the type checking which is the core of the proof engine. However, before we start the description of the proof engine, we will give an example of how a session in ALF may proceed, to illustrate the use of the system.

## 3.2 An ALF session

We will show how a session in ALF may appear for the user by doing a small example step by step. The example is about the list function *map*, which takes as arguments a function and a list, and applies the function to each element in the list. The property we want to show is that if we first map a function $f$ over a list and then map a function $g$ over the resulting list, it is the same as mapping the composition of the two functions over the list; that is,

$$map(g, map(f, l)) = map(f \circ g, l)$$

for any function $f$ and $g$ and any list $l$. For instance, this could be a rule of transformation in a compiler, since the right-hand side only traverses the list once, whereas the left-hand side must traverse the list twice. Hence, the right-hand side can be seen as an optimisation of the left-hand side.

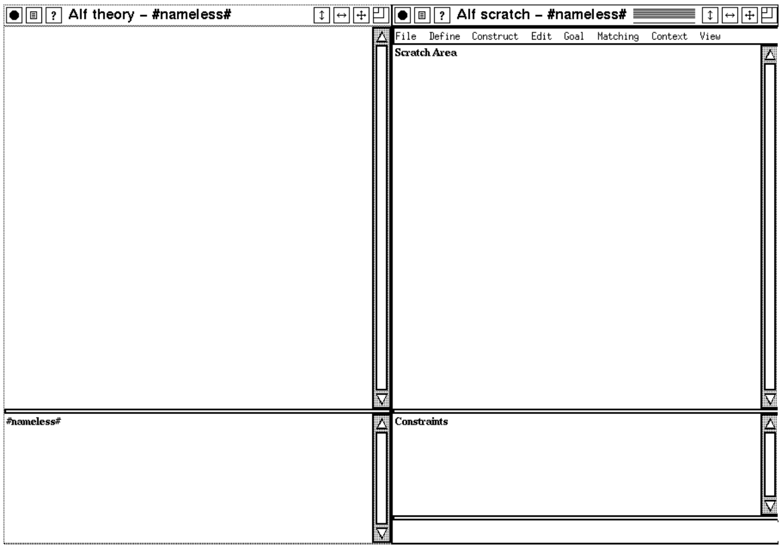
We will assume the datatype *List* and the relation *Id* from the previous sections are already defined in the theory. We will also use a congruence rule for the *Id*-relation, which says that if two elements $a$ and $b$ are equal in $A$ and $f$ is a function from $A$ to $B$, then the elements $f(a)$ and $f(b)$ are equal in $B$. The theory of our example is thus

$$
\begin{array}{l}
List : (A{:}Set)Set \\
\quad nil : (A{:}Set)List(A) \\
\quad cons : (A{:}Set\,;\, a{:}A\,;\, l{:}List(A))List(A)
\end{array}
$$

$$
\begin{array}{l}
Id : (A{:}Set\,;\, a{:}A\,;\, b{:}A)Set \\
\quad id : (A{:}Set\,;\, a{:}A)Id(A, a, a)
\end{array}
$$

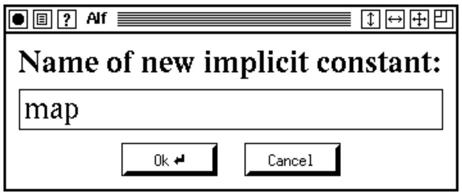
$$Idcongr : (A, B{:}Set\,;\, f{:}(A)B\,;\, a, b{:}A\,;\, c{:}Id(A, a, b))Id(B, f(a), f(b))$$

When ALF is first started, the following two windows appear on the screen:

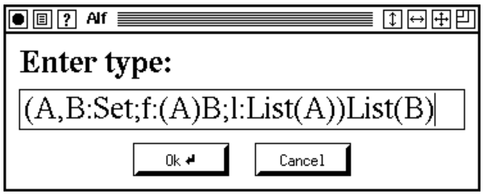



The theory window contains the complete definition, the theory, and the scratch area window the incomplete definitions. The scratch area window consists of three parts, the actual scratch area, the constraint window and the bottom part which shows the type of the current selection, if any. There are menu bars in the top, from which different commands can be selected.

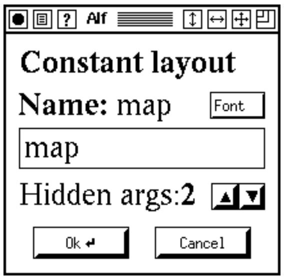
We will start by defining the function *map*, and we choose to represent it as an implicit constant, i.e. defining the function by pattern matching. When we start to define a new constant, a new window pops up asking for the name of the constant.




The next thing we must fill in, is the type of the constant. Here we chose to use the text edit window to enter the type directly as text. We could also build up the type incrementally in a similar manner as for terms.

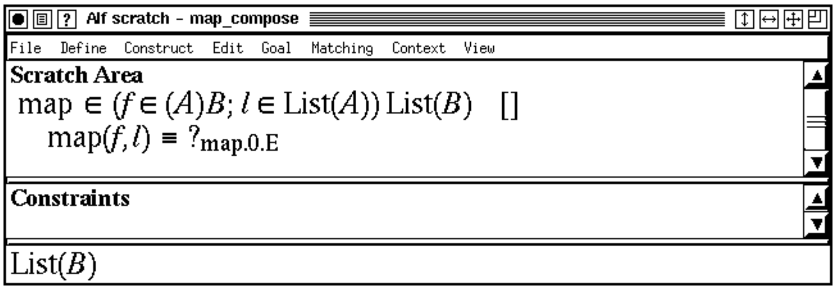



The type of the *map* function has two additional arguments, $A$ and $B$ of type *Set*, compared to the corresponding function in a polymorphic language such as SML, since the language of ALF is monomorphic. However, we can hide these two arguments by the layout mechanism, and the *map* function will appear as if it was polymorphic. Only if the hidden arguments could not be inferred by the proof engine, we would have to consider these arguments again.

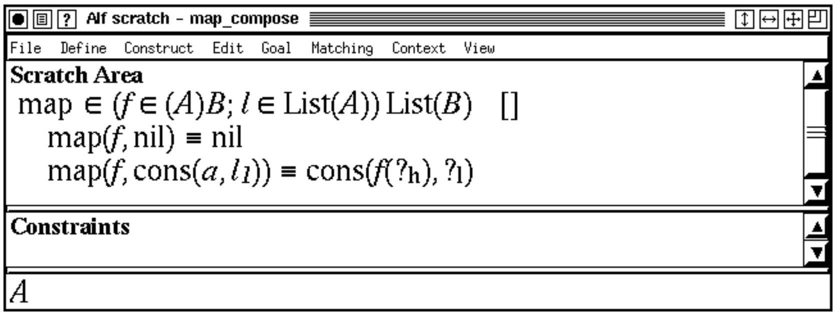



At present, one can only hide the first consecutive arguments, since a more general hiding mechanism is not yet implemented in the interface.

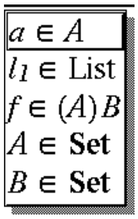
Now when the type of *map* is defined, the next step is to define the body of the function. The "make-pattern" command produces a pattern equation with a pattern containing only pattern variables and a right-hand side which is a new placeholder. In the bottom part of the scratch window below the constraint window, there is a small window which shows the type of the current selection. Unfortunately, the selection is not visible on the screen dumps, but in the below figure the placeholder is selected. We can see that the type of it is *List(B)*:




The *map* function is defined by case analysis on the structure of the list, so we will select the pattern variable $l$ and ask ALF to expand the patterns with respect to this variable. This command can only be applied if the right-hand side of the pattern equation is a placeholder. The result of this operation is two new pattern equations, one for the *nil* case and one for the *cons* case. The monomorphic list parameter is hidden in the constructors. The base case is solved by the *nil* constructor, but here the parameter is inferred to $B$, whereas the *nil* in the pattern is of type $List(A)$. *map* on a non-empty list is defined by the *cons* constructor, where the first element of the list is applied to the function $f$:



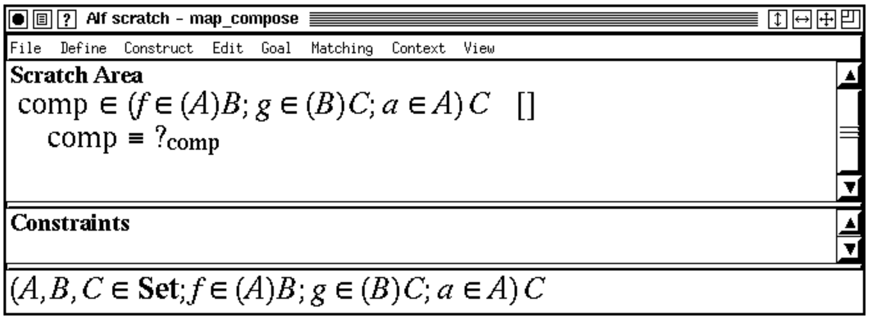

The current selection in the above picture is the placeholder $?_h$, and we can see its type in the bottom window. If we look in the context menu, we will find the variables of the local context of the current selection. In this case, it is the pattern variables in the left-hand side, which are:




We could also have looked in the matching menu, which contains the variables from the local context and the constants from the theory which matches the current selection, if it is a placeholder. However, it does not contain every constant or variable which is possible to apply, since this would require that the type of the placeholder should be unified with respect to the type of every constant in the theory, and if the theory is large this could take some time. Therefore, we have chosen a simple kind of matching which is efficient and often enough gives the desired choice. For our current selection, there was only one possible choice, the variable $a$. Naturally, we could also choose to give the term directly, by selecting the edit as text command.

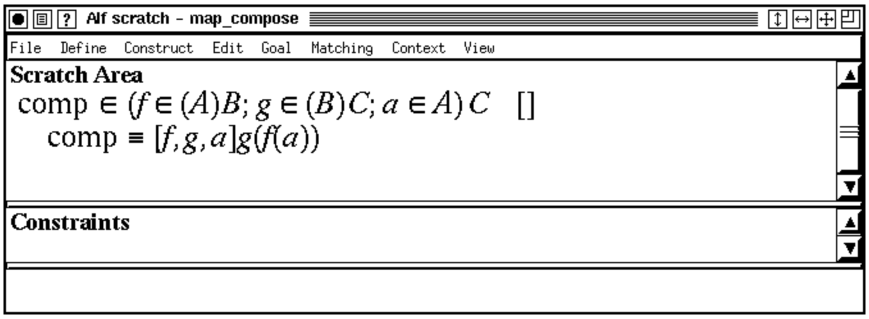
Finally, we can complete the function by recursively call the *map* function on the smaller argument $l_1$. As mentioned, we have to convince ourselves that the recursion in the implicit constants is well-founded, and in this case $l_1$ is a proper sub-term of the original list $l = cons(a, l_1)$.

Next, we need to define composition of two functions. It will be defined as an explicit constant since it is not a recursive function. As before, we give the type of the constant first, and we choose to hide the parameters $A, B$ and $C$ which are the domain and range of the respective functions. In the bottom window we can see the expanded type of the function, since the placeholder $?_{comp}$ is selected:

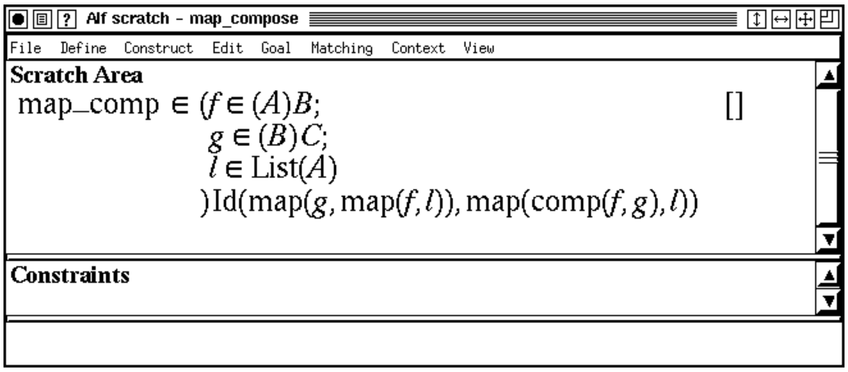


The type of *comp* may not look exactly as expected, since it might not be apparent that it returns a function from $A$ to $C$. Despite the notation in ALF, the functions in ALF are curried as in other functional languages, so given two functions $f$ and $g$, $comp(f, g)$ is a function of type $(A)C$.

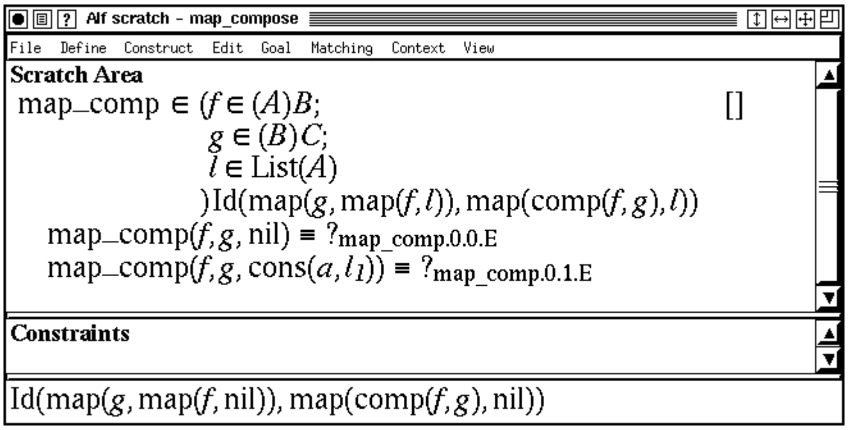
Now the definiens of $comp(f, g)$ can be filled in, and since it is an explicit constant it is a $\lambda$-term. Hence, *comp* is an abbreviation of the term $\lambda f \lambda g \lambda a.g(f(a))$ (in $\lambda$-calculus notation), which in ALF is written as $[f, g, a]g(f(a))$:



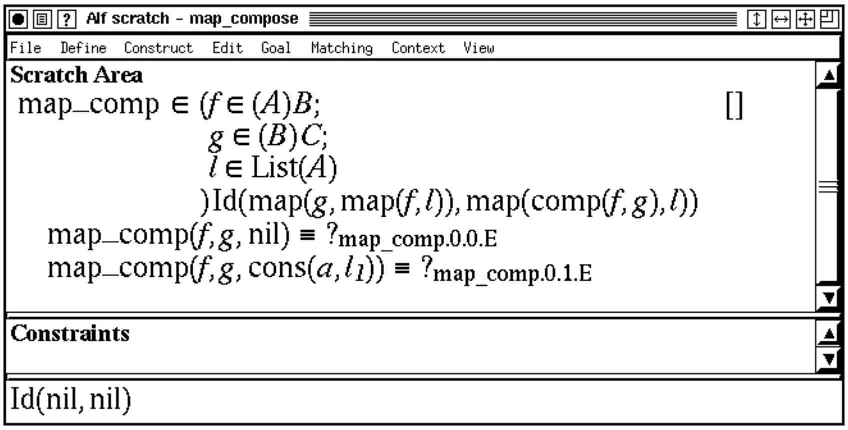

Everything we have done so far could have been done in any ordinary functional programming language. However, the next step is to define the property of *map*, and here we need the power of dependent types, since the arguments to the identity relation *Id* depends on the arguments $f, g$ and $l$:




Just as for the *map* function, we will define *map_comp* by pattern matching over the list, i.e. we do induction on the structure of the list, which results in the two cases corresponding to the two constructors of lists. In the following figure we have selected the placeholder corresponding to the case when the list is empty:

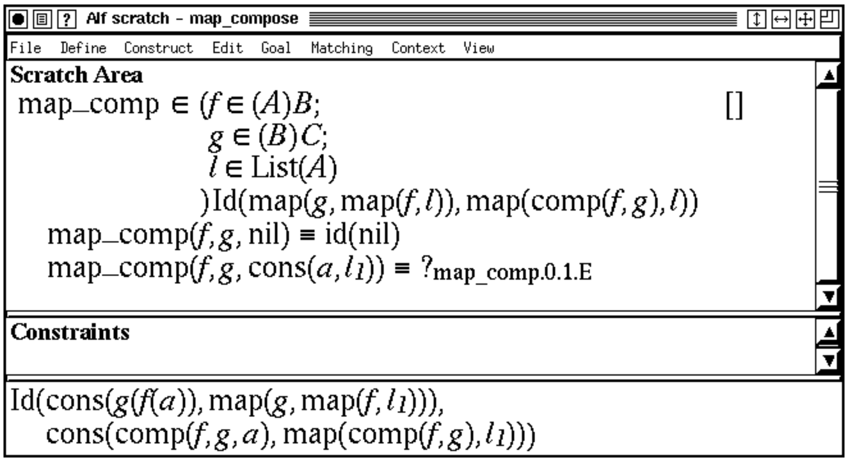



We can see the type of the placeholder in the bottom window, which is the property we have to show for this case. If we look closely at the arguments to *Id*, we can see that both arguments can be reduced and are in fact both the empty list, i.e. the constructor *nil*. However, it is not always so easy to see what the type of the goal states, so we can ask ALF to reduce the type of the placeholder for us which usually results in a more understandable type:

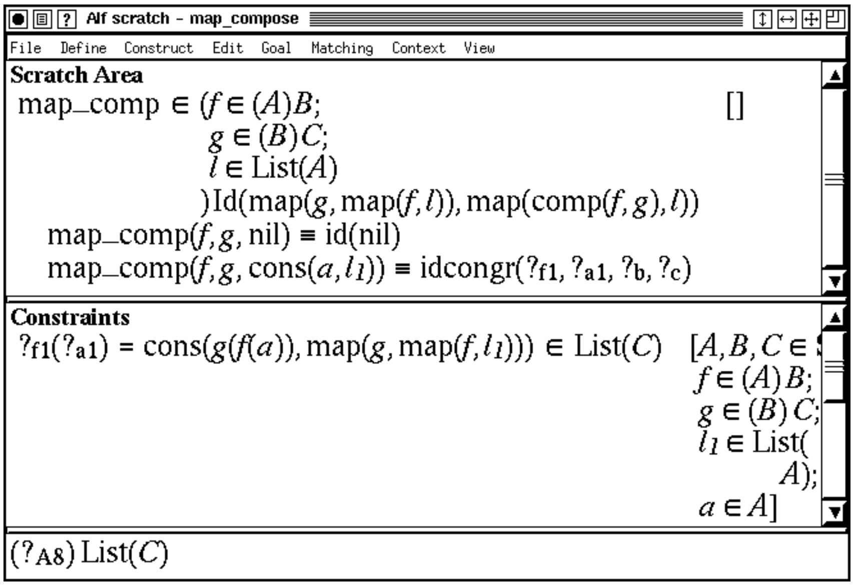



Now it clear that we can solve this case with the only constructor of the *Id* relation, which denotes reflexivity.

For the case when the list is non-empty, we do the same thing and after the type of this case is massaged to a more readable form, we can see that both arguments to *Id* is on *cons*-form:

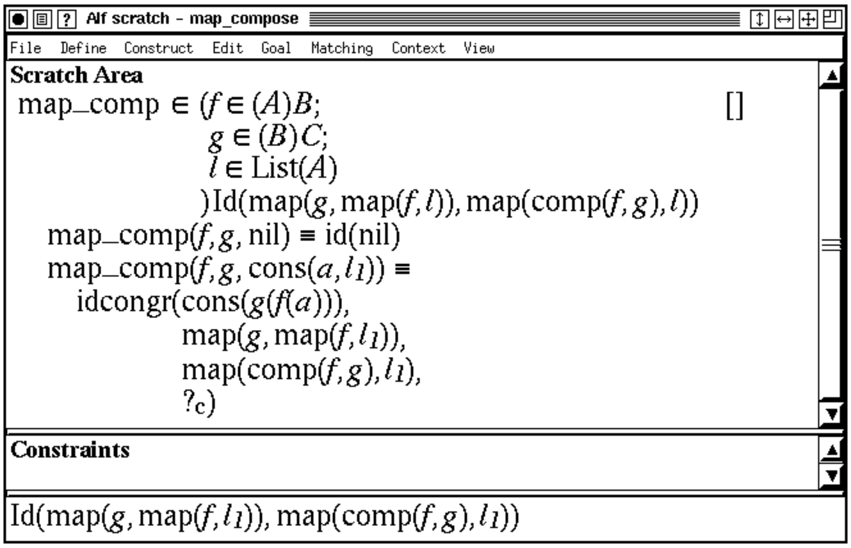



Hence we can apply the congruence rule from the theory:



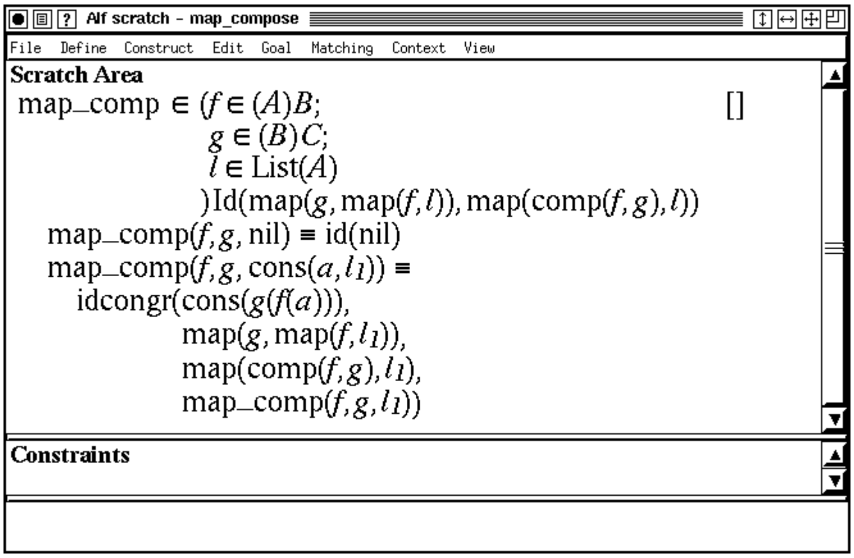
For the first time we can see the use of the constraint window. When the constant *idcongr* was applied to the second case, the type of *idcongr* was unified with the type of the remaining placeholder. Here the unification did not manage to solve all equations, since the placeholder $?_{f1}$ is a higher-order placeholder, that is it is of function type. Difficult equations like this one, are kept as constraints, restricting the future instantiations of the involved placeholders.

If we now give the function we intended, we replace $?_{f1}$ by *cons*, then the constraint can be resolved and the second and third arguments to *idcongr* are inferred by unification.

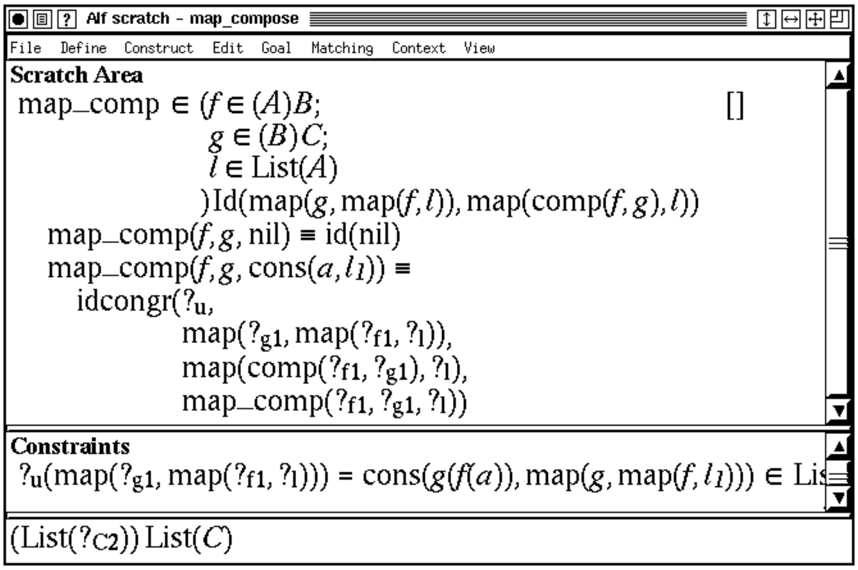



Now we have only one placeholder left, and looking at its type we can see that it is an instance of the property we are currently proving. Moreover, *map_comp* is applied to the list $l_1$ which is strictly smaller than $l$, and hence we can use *map_comp* recursively to complete the second case. The recursive call corresponds to using the induction hypothesis in a proof of structural induction over the list. Then one would start by assuming that the property holds for the sub-list $l_1$ and from there construct the proof for the list $l$. Here we do the other way around, the proof is constructed in a top-down manner. We start by proving the property for the entire list when it is of the form $cons(a, l_1)$, and when we have reached a sub-goal which is an instance of our property which is structurally smaller, we are allowed to use the constant recursively.

Hence, we have completed the proof and it can be moved to the theory:

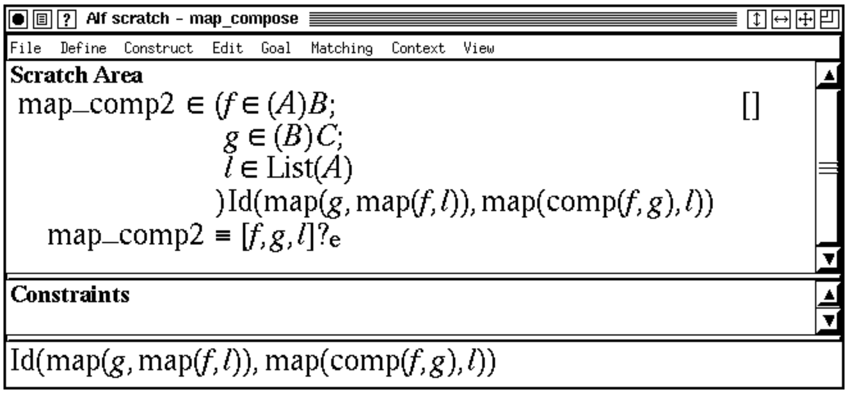



Just to illustrate the local undo operation, which is a unique feature of ALF, we will delete a sub-term in the completed proof. If we delete the first argument to *idcongr*, we get the following incomplete proof object:



Note that this is a completely new state, since we never had the fourth argument of *idcongr* instantiated when the first argument was not. We can also see that the placeholders $?_{f1}, ?_{g1}$ and $?_l$ are not inferred anymore, since they were inferred from the equation we now see as a constraint. When the placeholder $?_u$ is again instantiated to *cons*, the equation can be simplified further which gives the instantiations to $?_{f1}, ?_{g1}$ and $?_l$. What is important is that this is exactly the same incomplete proof object we would have got if we started to solve the fourth argument directly with the recursive call. Hence, the local undo operation is really the dual operation to the refine operation, and the user can freely edit the proof object and recover from mistakes.

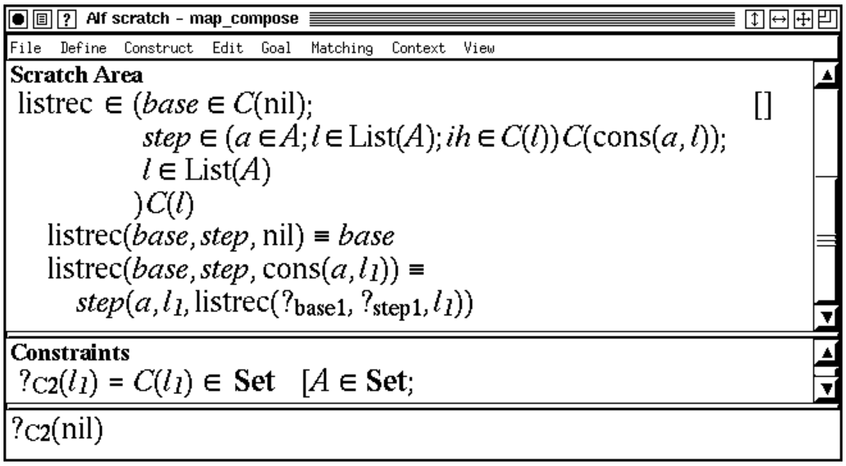
To give a complete description of ALF, we will shortly present how the above proof can be made using the elimination rule of lists, that is the rule of structural induction over lists. For instance, if one wants to stay within the monomorphic set theory as it is presented in [NPS90], and not use the pattern matching facility (except for defining the elimination rules), then the proof will be a named $\lambda$-term, i.e. an explicit constant. Here we have started by abstracting the variables $f, g$ and $l$, and the type of our sub-goal $?_e$ is in the bottom window:




Here we need to use the elimination rule for lists, corresponding to the rule of structural induction:

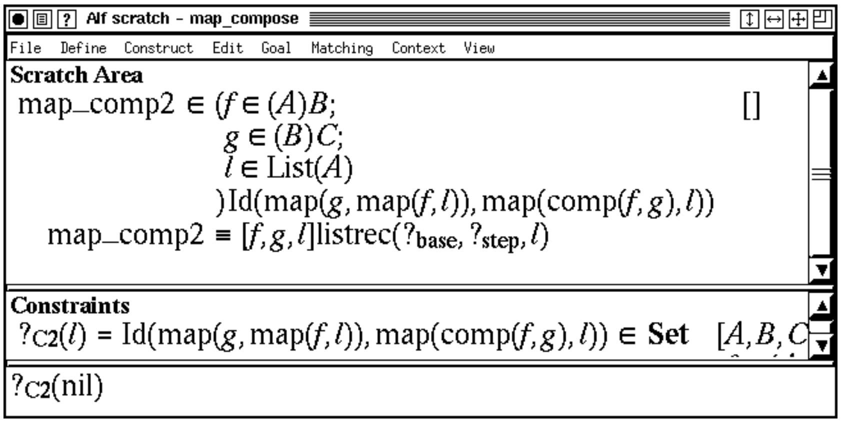
$$\frac{C(nil) \qquad \begin{array}{c}[a : A, l_1 : List(A), C(l_1)] \\ \vdots \\ C(cons(a, l_1))\end{array}}{C(l)}$$

If we do not already have this rule in our theory, we can simply add a new implicit constant and start defining it since we can have several incomplete definitions in the scratch area simultaneously. The elimination constant *listrec* thus takes as arguments a predicate $C$ over lists, a proof of the base case and a method (function) which takes a proof of $C(l_1)$ to a proof of $C(cons(a, l_1))$ and a list $l$. The contraction rules shows how a general induction proof (for any list) can be reduced to a particular proof for a given list. Hence, if we have a general induction proof (which consists of a base case and a step function) and want a proof for the empty list, we simply take the proof of the base case. Similarly, in the non-empty case we use the step function, and apply the elimination rule recursively on the smaller list $l_1$:

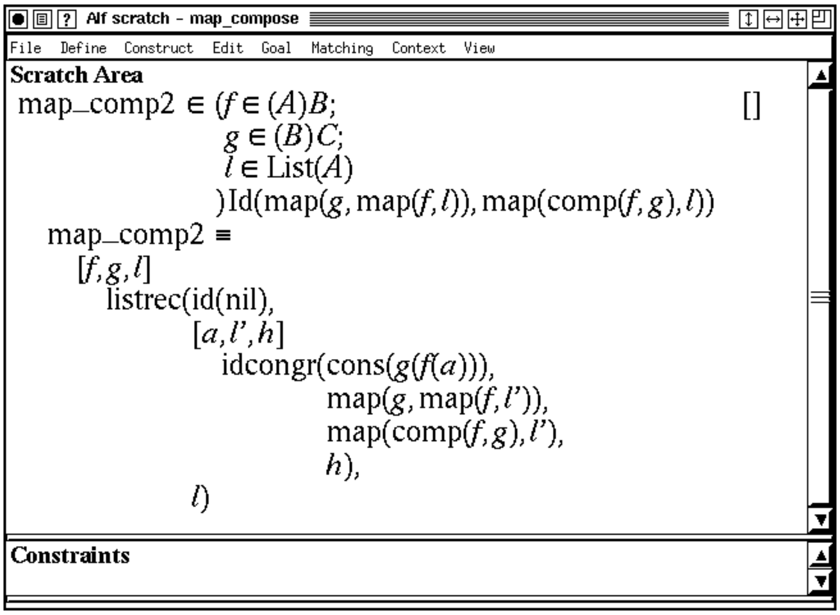



In the above definition of *listrec*, we have hidden the first two arguments, that is the set $A$ and the predicate $C$. The placeholder $?_{C2}$ which occur in the constraint thus corresponds to the second (hidden) argument to *listrec* in the recursive call. Since it is hidden, it does not appear anywhere in the incomplete definition. To instantiate it, we can temporarily change the mode in the view menu such that all hidden variables are visible in the definition. However, in this case we can also use a constraint-tactic, which solves constraints of some simple forms. Here we want $?_{C2}$ to be instantiated to $[l_1]C(l_1)$, so we can simply click on the constraint and choose the command solve, which will do exactly this. This constraint could have been solved automatically if the extension of first-order unification presented in [Mil89] was implemented. The last two placeholders are simply filled in with $base$ and $step$, respectively.

Now we can come back to our proof of *map_comp2* again, and refine the goal with *listrec*. When we have chosen the induction variable $l$, we can solve the constraint in the same way as before.




Finally, we can solve the two cases: the base case when the list is empty and the induction case when the list is non-empty just as we did with the pattern matching method. In the induction case, the variable $h$ is the assumption of the induction hypothesis, and it is used where the recursive call is used in the pattern matching method.




Here we conclude the general description of ALF. Before we start the detailed description of the proof engine we need to present the substitution calculus which is the version of type theory that ALF is based on.

# Chapter 4

# The substitution calculus of type theory

Martin-Löf's monomorphic type theory [NPS90], also referred to as Martin-Löf's logical framework, is a typed $\lambda$-calculus with dependent function types. The assertions which can be made in the theory, are *judgements* stating for instance that an object is of a given type, i.e. the typing judgement

$$a : A.$$

This judgement form is the most important for our purposes since proof checking (type checking) is a procedure which given a term and a type, checks whether the typing relation holds between the term and the type. For the above judgement to be meaningful we must know that $A$ is a type, hence there is also a type formation judgement $A : Type$. The formalism of type theory provides rules for how judgements are formed. We will present the most important rules below. Besides type formation and typing rules there are equality rules stating that two types are equal types and that two objects in a type are equal.

All the above judgement forms can be hypothetical, that is the assertion can depend on assumptions which correspond to that the judgement is stated relative a context $\Gamma$, which describes the types of variables occurring in the judgement

$$a : A \quad \Gamma.$$

Thus we have the following judgement forms:

| | |
|---|---|
| $\Gamma \vdash \alpha : Type$ | $\alpha$ is a type in $\Gamma$ |
| $\Gamma \vdash \alpha = \beta : Type$ | $\alpha$ and $\beta$ are equal types in $\Gamma$ |
| $\Gamma \vdash a : \alpha$ | $a$ is a term of type $\alpha$, in context $\Gamma$ |
| $\Gamma \vdash a = b : \alpha$ | $a$ and $b$ are equal terms of type $\alpha$, in $\Gamma$ |

where $\Gamma \vdash \mathcal{J}$ denotes that the assertion $\mathcal{J}$ relative $\Gamma$ is derivable according to the judgement rules in the theory. The judgement rules will be defined below.

The purpose of the substitution calculus of type theory is to formalise the notion of a context and substitution, which are usually presented informally. For instance, substitution is usually a meta notation

$$b[a/x]$$

which is explained as "replacing all free occurrences of $x$ in $b$ by $a$". In the substitution calculus this operation is explained in detail, and is therefore much closer to an actual implementation. Moreover, the informality of '...' in the explanation of a context

$$[x_1 : \alpha_1, \ldots, x_n : \alpha_n]$$

or a substitution

$$\{x_1{:=}a_1, \ldots, x_n{:=}a_n\}$$

is eliminated. Substitutions now become part of the term language instead of a meta notation, which motivates the name *explicit substitutions*.

The substitution calculus can be seen as a calculus which formalises the way substitutions are usually handled in an implementation of a functional programming language, or any implementation of function application. The intuition is that an explicit substitution is like a (non-recursive) local environment of variables bound to terms, that is a *closure* in the functional programming language terminology. Hence a term applied to a substitution is a term which carry around its own closure.

The order of the variable bindings is important, since we will use closures during computation to bind arguments to formal parameters (bound variables) of a function when it is applied. A substitution behaves like a stack which means that in the substitution $\gamma$ extended by an new binding

$$\{\gamma, x{:=}a\},$$

the binding of $x$ to $a$ hides other bindings of $x$ in $\gamma$. Explicit substitutions are used for defining $\beta$-reduction,

$$([x]b)a \rightarrow_\beta b\{x{:=}a\}$$

just as in implementations an environment is often used for binding terms or values to variables. The environment is extended with the new binding of $x$ to $a$ during the computation of the function body $b$. In both cases, the substitution is not performed in the term, instead the assignment to a variable is looked up in the substitution. One can also achieve different kinds of evaluation orders depending on how the $\beta$-rule and the look-up rules are defined. For instance the $\beta$-rule together with lookup rules which simply copies the assigned term for every occurrence of the variable in the function body is the call-by-name (normal-order) strategy. If instead the assigned term is reduced and the explicit substitution updated with the reduced term, then we get laziness with some sharing, that is call-by-need. Finally, if the assigned term $a$ is reduced before the explicit substitution is created, then we get the call-by-value strategy. The strategy used in the below presentation of the substitution calculus, corresponds to the normal-order evaluation strategy.

Hence, one advantage of the substitution calculus is that depending on what set of computation rules we use for handling $\beta$-reduction and the substitutions, we can explain different evaluation orders. Another advantage is that $\alpha$-conversion, that is renaming of bound variables, is not needed at all if we do not reduce under $\lambda$ [CHL]. This is because we can postpone substitutions, and therefore a substitution must never be applied inside a bound variable where the problem of capturing variables occur. For instance, the term

$$((([x,y](xy))\ y)\ z$$

is computed to

$$(xy)\{x{:=}y, y{:=}z\}$$

which reduces to

$$(yz).$$

Without explicit substitutions, the bound variable $y$ must be renamed so that the argument $y$ is not captured when it is substituted for $x$.

There will be new judgement forms expressing that a context or a substitution are properly formed. Moreover, there are assertions corresponding to the equality judgements above which states that two substitutions are equal. For contexts, equality will not be the basic judgement form but it can be defined. The most convenient relation on contexts is the sub-context relation, denoted

$$\Gamma \leq \Delta,$$

which means that $\Delta$ is an extension of $\Gamma$. This relation is chosen since it fits with the thinning (or weakening) rule, which says that if an assertion holds relative a context it also holds relative any extension of this context. The logical reading is that if a statement is true under a set of assumptions, it is also true if more assumptions are added.

Substitutions are "typed" by contexts. We will say that a substitution *fits a context* if all assignments in the substitution are well-typed relative the corresponding type of the variables in the context. Analogously to other hypothetical judgements, we will use the notation

$$\Delta \vdash \gamma : \Gamma$$

to mean that $\gamma$ fits $\Gamma$ relative the context $\Delta$. Intuitively, the substitution $\gamma$ is of the form

$$\{x_1{:=}a_1, \ldots, x_n{:=}a_n\}$$

if $\Gamma$ is the context

$$[x_1 : \alpha_1, \ldots, x_n : \alpha_n]$$

and where the assigned terms have the correct types, i.e.

$$\begin{array}{c} a_1 : \alpha_1 \quad \Delta \\ \vdots \\ a_n : \alpha_n \quad \Delta \end{array}$$

For example, from the $\beta$-rule

$$([x]b)a \rightarrow_\beta b\{x{:=}a\}$$

we get the substitution which is well-typed by

$$[\,] \vdash \{x{:=}a\} : [x : \alpha]$$

if

$$[\,] \vdash [x]b : \alpha{\rightarrow}\beta \text{ and } [\,] \vdash a : \alpha$$

If a substitution does not assign terms to all variables in its context, it can always be extended by adding the identity assignment $x{:=}x$ for these variables. We believe it is easier to think of a substitution in its expanded form, that is when all variables of its context are implicitly assigned. When a substitution is applied to a term, it is clear that assignments of the form $x{:=}x$ has the same effect as no assignment, that is no effect at all. However, when we think of the judgement

$$\Delta \vdash \gamma : \Gamma$$

and if $\gamma$ is expanded with the identity assignments, it is clear that all variables assigned by themselves must also be in $\Delta$, since $\Delta$ must contain all free variables of the assigned terms in $\gamma$.

The new judgements of the substitution calculus thus have the following forms:

| | |
|---|---|
| $\Gamma : Context$ | $\Gamma$ is a context |
| $\Gamma \preceq \Delta$ | $\Gamma$ is a sub-context of $\Delta$ |
| $\Gamma \vdash \gamma : \Delta$ | $\gamma$ is a substitution fitting context $\Delta$ in context $\Gamma$ |
| $\Gamma \vdash \gamma = \delta : \Delta$ | $\gamma$ and $\delta$ are equal substitutions of $\Delta$ in $\Gamma$ |

The substitution calculus is not crucial for the work presented here: the type checking of incomplete terms could be performed without explicit substitutions. However, it effects the possibility of reducing incomplete terms, that is "how far" the incomplete term could be reduced. For example, assume we want to reduce the term

$$([x]f(x, ?_1))a$$

where $a$ is the argument to the function $[x]f(x, ?_1)$ which contains a placeholder $?_1$. Since $?_1$ is within the scope of the binder $x$, it may depend on $x$, that is its local context is $[x : A]$ for some type $A$. Without explicit substitutions, this term could not be reduced any further, since what would we do with the second argument, i.e. the placeholder? We cannot forget that once the placeholder $?_1$ is instantiated, say to $x$, then $x$ should be replaced by $a$. With explicit substitution we can safely reduce the term to

$$f(a, ?_1\{x{:=}a\})$$

and when the placeholder $?_1$ is instantiated to $x$ we have the term

$$f(a, x\{x{:=}a\}) = f(a, a)$$

since the term $a$ can simply be looked up in the substitution.

The possibility of reducing terms as far as possible is important since it improves the unification algorithm. If we are interested in finding instantiations to the placeholders in the equation

$$([x]f(0, ?_1))a = f(?_2, a)$$

where we can reduce the left-hand side expression yielding

$$f(0, ?_1\{x{:=}a\}) = f(?_2, a)$$

which can be simplified to

$$0 = ?_2$$
$$?_1\{x{:=}a\} = a$$

Hence, the unification found an instantiation of the placeholder $?_2$ which would not have been possible without explicit substitutions.

Some of the rules of the substitution calculus will be explained below, after we have formally introduced the syntax. The complete set of rules which are used in the correctness proof can be found in appendix A.

## 4.1 Syntax

A term $e$ is defined by the following grammar

$$e ::= \ x \mid c \mid [x]e \mid (ee) \mid e\gamma$$

where $x$ is a variable and $c$ a constant. Later, we will extend the notion of a term to include incomplete terms, that is when the term also may contain placeholders. Application is sometimes written $f(e)$, and $f(e_1, \ldots, e_n)$ is an abbreviation of $((\cdots(fe_1)\cdots)e_n)$. Terms and substitutions become mutually dependent when we extend the language of $\lambda$-calculus with explicit substitutions. A substitution $\gamma$ is of the form

$$\gamma ::= \{\} \mid \{\gamma, x{:=}e\} \mid \gamma\gamma.$$

where $\gamma\gamma$ denotes composition of substitutions. We write $\{x_1{:=}a_1, \ldots, x_n{:=}a_n\}$ as an abbreviation of $\{\{\cdots\{x_1{:=}a_1\}\cdots\}, \ldots, x_n{:=}a_n\}$.

Composing substitutions $\gamma$ with $\delta$ is valid if

$$\Psi \vdash \delta : \Delta, \text{ and}$$
$$\Delta \vdash \gamma : \Gamma.$$

Hence, the substitutions must fit, that is the variables in $\Delta$ are the ones which are assigned by $\delta$. The composition is computed in the following way; $\delta$ is applied to each assigned term in $\gamma$ and the substitution $\delta$ is "pushed in front of" the substitution $\gamma$:

$$\{\}\delta = \delta$$
$$\{\gamma, x{:=}e\}\delta = \{\gamma\delta, x{:=}e\delta\}.$$

Composition of substitutions appear for instance in the following example:

$$([y](([x]f(x,y))\ y))b$$

where we have a $\beta$-redex outermost which is reduced to

$$(([x]f(x,y))\ y)\{y{:=}b\}.$$

Reducing the other $\beta$-redex yields the new term

$$(f(x,y)\{x{:=}y\})\{y{:=}b\}.$$

If we first perform the inner substitution we get $f(y,y)\{y{:=}b\}$. The other possibility is to compose the two substitutions, and here we see that the outer substitution must be applied to the assigned term in the inner substitution to get the proper result:

$$f(x,y)\{y{:=}b, x{:=}b\}.$$

Furthermore, the result of composition contains all variable assignments, otherwise we would have lost the assignment to $y$.

Substitutions can be applied to types as well as to terms, but can only effect the terms in the types since the variables in the substitutions range over terms and not types. Despite this, they are the reason why we need to change the notation for types. Consider for instance the function type applied to a substitution:

$$((x : A)B)\gamma$$

In this type $x$ is a binder for free occurrences of $x$ that may appear in $B$. The argument type $A$ on the other hand, may not depend on the variable $x$. If we allow a substitution to be distributed inside this function type, we could get the following situation:

$$\begin{aligned}&((x : N)P(x))\{x{:=}true\} =\\ &(x : N\{x{:=}true\})(P(x)\{x{:=}true\}) =\\ &(x : N)P(true)\end{aligned}$$

which is not even well-typed, and even if the assignment was of type $N$ it is not any longer the same function. Hence, we have no way of distributing the substitution inside this type constructor. Therefore, the type constructor of a function type will instead be denoted by

$$A \rightarrow [x]B$$

where $B$ is a family of types which depends on the variable $x$. The advantage of this notation is that the binder $x$ appears in the position of its binding occurrence. Hence, we can distribute the substitution inside the type constructor without problem

$$A\gamma \rightarrow ([x]B)\gamma.$$

A type $\alpha$ and a family of types $\beta$ have the following forms

$$\begin{aligned}\alpha &::= \ Set \mid \alpha \rightarrow \beta \mid \beta e \mid \alpha\gamma\\ \beta &::= \ El \mid [x]\alpha \mid \beta\gamma.\end{aligned}$$

We will write $\alpha \rightarrow \beta$ as an abbreviation of $\alpha \rightarrow [x]\beta$ when $\beta$ does not depend on $x$. Also, *El* is omitted when it is clear from the context that the type is meant rather than the term. The syntax of a context $\Gamma$ is

$$\Gamma ::= [\,] \mid [\Gamma, x : \alpha].$$

We will denote terms by $a, b, d, e$, and reserve $f$ for functional terms and $c$ for constants. Occasionally, $A$ and $B$ stands for terms of type *Set*. Types are denoted by $\alpha$ and families of types by $\beta$. Substitutions are called $\gamma$ or $\delta$ and contexts $\Gamma, \Delta$ or $\Psi$.

## 4.2 The rules

The set of rules we are using in the correctness proof in section 5.6 can be found in appendix A, and the entire set of rules in the calculus and semantic justifications can be found in [Tas93]. Here, we will concentrate on the particular rules for contexts and substitutions, since they are not so well known as the others. The object formation rules (typing rules) will also be presented, since they form the basis for the type checking algorithm.

The rules are divided according to the different judgement forms, where the conclusion of the rule decides to which category the rule belongs.

**Context rules**

The two formation rules for contexts state that a context is either empty, or is a valid context extended with a new variable declaration. The restrictions on the new declaration is that the variable is fresh, i.e. it does not already occur in the context, and the type is valid in the context.

$$\frac{}{[\,] : \mathit{Context}}\ \text{ConNil}$$

$$\frac{\Gamma : \mathit{Context} \quad \Gamma \vdash \alpha : \mathit{Type}}{[\Gamma, x : \alpha] : \mathit{Context}}\ \begin{array}{l}\text{ConExt}\\ (x \notin Dom(\Gamma))\end{array}$$

**Sub-context rules**

As already explained, the interesting relation on contexts is the sub-context relation. From this relation we can define the equality on contexts by the two requirements

$\Gamma \preceq \Delta$ and $\Delta \preceq \Gamma$.

The first rule states that the empty context is a sub-context of any valid context.

$$\frac{\Delta : \mathit{Context}}{[\,] \preceq \Delta}\ \text{SubNil}$$

A context $[\Gamma, x : \alpha]$ is a sub-context of $\Delta$ if first $\Gamma$ is a sub-context of $\Delta$ and if secondly, $\Delta \vdash x : \alpha$. The intuition of a sub-context is that any variable declared in the sub-context, must also be declared in the larger context. However there is a choice of what we mean by a variable "also declared in". One interpretation is that the two types of the variable in the two contexts are syntactically the *same*, or the weaker requirement that they are judgementally *equal*. The first approach corresponds to requiring that for two assumptions to be the same, they must be of the same form, and the other corresponds more or less to that the form can be different but the meaning the same. The rule

$$\frac{\Gamma \preceq \Delta \quad \Delta \vdash x : \alpha}{[\Gamma, x : \alpha] \preceq \Delta}\ \text{SubExt}$$

is of the second kind, since the requirement that $x$ of type $\alpha$ is *derivable* in $\Delta$ is weaker than $x : \alpha$ in $\Delta$ since we may have used the type conversion rule after the assumption rule (both defined below) for the first case.

**Substitution rules**

The first two rules are the two ways of constructing a substitution, one for constructing the "empty" or identity substitution

$$\frac{\Gamma : \mathit{Context}}{\Gamma \vdash \{\} : \Gamma}\ \text{Id}$$

and the other which updates the substitution with a new assigned variable. The reason the empty substitution is called the identity is because if we expand it to a normal substitution, that is all variables in the context are assigned a term in the substitution, it will only consist of variables assigned to themselves. Hence an empty substitution has no effect when it is applied to a term.

The updating rule

$$\frac{\Gamma \vdash \gamma : \Delta \quad \Gamma \vdash a : \alpha\gamma}{\Gamma \vdash \{\gamma, x{:=}a\} : [\Delta, x : \alpha]}\ \text{Upd}$$

extends the substitution with a new assigned variable. The reason it is called update rather than extend as in the case of contexts, is because the new variable may already be assigned a term in the substitution and then this variable is really updated because the substitution is treated as a stack.

Substitutions can be composed, as we explained in section 4.1, by the following rule:

$$\frac{\Delta \vdash \gamma : \Gamma \quad \Psi \vdash \delta : \Delta}{\Psi \vdash \gamma\delta : \Gamma}\ \text{Comp}$$

There is a thinning rule

$$\frac{\Psi \vdash \delta : \Delta \quad \Psi \preceq \Gamma}{\Gamma \vdash \delta : \Delta}\ \text{Thin}$$

for every judgement form stating that if a judgement is derivable relative a context, then it is also derivable relative any extension of that context. The intuition is that if we know something under a certain set of assumptions, we still know the same thing if we have added more assumptions. Here we state the particular rule for substitution, below we give a general rule schema of which this one is an instance.

The target-thinning rule

$$\frac{\Gamma \vdash \delta : \Delta \quad \Psi \preceq \Delta}{\Gamma \vdash \delta : \Psi}\ \text{T-Thin}$$

is a specific thinning rule for substitutions, and what is says is that if we have a well-typed substitution fitting a context $\Delta$, then it also fits the smaller context $\Psi$. The consequence of this rule is that substitutions may contain superfluous assignments to variables relative to its context. This is a natural rule if we think of the implementation analogy, since it says that the environment may contain more assigned variables than we are interested in.

**Type rules**

The basic type is *Set* which is the type of all inductively defined sets:

$$\frac{\Gamma\ :\ \mathit{Context}}{\Gamma \vdash \mathit{Set}\ :\ \mathit{Type}}\ \text{SetForm}$$

The function formation rule takes a type and family of that type, and constructs a function type:

$$\frac{\Gamma \vdash \alpha\ :\ \mathit{Type} \quad \Gamma \vdash \beta : \alpha{\rightarrow}\mathit{Type}}{\Gamma \vdash \alpha{\rightarrow}\beta\ :\ \mathit{Type}}\ \text{FunForm}$$

Compared to the type formation rules in the theory without explicit substitution [NPS90], there is many more rules in this presentation and the main reason is that the notion of a family of types is formalised as well. The rule of function formation is in that presentation

$$\frac{\Gamma \vdash A\ :\ \mathit{Type} \quad [\Gamma, x : A] \vdash B\ :\ \mathit{Type}}{\Gamma \vdash (x : A)B\ :\ \mathit{Type}}$$

and a family of types is here a type which depend on variables in a context. In this rule we must require that all free occurrences of the variable $x$ in $B$ becomes bound in the formation of the function type. Here the binding of the variable will occur in the family rather than when the function type is constructed, so we will have an abstraction rule on the level of types as well as the level of terms. Thus, a family is a function from elements in a type $A$ to a type $B$ where $x$ (of type $A$) may occur free. The function is denoted by $[x]B$, and it is of type $A \to Type$. Hence, from the premisses in the rule above there are two steps before we can construct the function type $A \to [x]B$:

$$\dfrac{\Gamma \vdash A : Type \qquad \dfrac{\Gamma \vdash A : Type \quad [\Gamma, x : A] \vdash B : Type}{\Gamma \vdash [x]B : A \to Type}\,\text{Abs}}{\Gamma \vdash A \to [x]B : Type}\,\text{FunForm}$$

Since families are functions, they can be applied to elements in the proper domain, so we have an application rule on the type level corresponding to the application rule on the term level.

$$\dfrac{\Gamma \vdash \beta : \alpha \to Type \quad \Gamma \vdash a : \alpha}{\Gamma \vdash \beta a : Type}\,\text{App}$$

All judgement forms can be applied to a substitution, and we will give a general rule schema below, just as for the thinning rule. The rule for applying a type to a substitution

$$\dfrac{\Gamma \vdash \alpha : Type \quad \Delta \vdash \gamma : \Gamma}{\Delta \vdash \alpha\gamma : Type}\,\text{Subst}$$

says that if we have a type $\alpha$ in a context $\Gamma$, that is the variables of $\Gamma$ may occur free in $\alpha$, then we can apply a substitution which may assign terms to these free variables. However, these assignments may contain free variables from another context $\Delta$, so the resulting type if we perform the substitution will be a type which may contain variables from the context $\Delta$. The free variables of $\alpha$ become "bound" in the substitution $\gamma$, and the new free variables are these in the assigned terms of $\gamma$, which must all be in $\Delta$ (by the meaning of $\Delta \vdash \gamma : \Gamma$).

**Family rules**

Recall the informal type formation rules from section 2.1, which essentially is the *El*-formation rule in [NPS90]:

$$\dfrac{\Gamma \vdash A : Set}{\Gamma \vdash El(A) : Type}$$

It says that from any set $A$, we can form the type of the elements of $A$, that is *El* is a type-constructor (function) which from a set constructs a type. Since we now can define type-valued functions, i.e. families, we can define *El* to be a predefined family of types, which range over sets:

$$\frac{\Gamma : \mathit{Context}}{\Gamma \vdash \mathit{El} : \mathit{Set} \to \mathit{Type}} \text{ ElForm}$$

A type family is constructed from a type which depends on a variable $x$:

$$\frac{[\Gamma, x : \alpha'] \vdash \alpha : \mathit{Type}}{\Gamma \vdash [x]\alpha : \alpha' \to \mathit{Type}} \text{ Abs}$$

**Term rules**

The first five rules correspond to the different forms of a term, which means we have rules corresponding to variables (assumptions), constants, abstractions, applications and terms applied to substitutions. The additional rule is the rule of type conversion, which says that if a term is well-typed by a type which is equal to another type, then the term is also well-typed with the other type.

The assumption rule

$$\frac{}{\Gamma \vdash x : \alpha} \ {}^{\text{Ass}}_{(x\, :\, \alpha \,\in\, \Gamma)}$$

says that if we have assumed a variable of a type, then we can derive that the variable is of its declared type. We know for any well-formed context $\Gamma$, that if $x : \alpha$ is declared in $\Gamma$, then $\alpha$ is a valid type in the context prior to the declaration of $x$ and by the thinning rule we can extend the context.

Analogously for constants, if we have defined a constant with type $\alpha_c$ and context $\Gamma_c$ in a given theory $\Sigma$, then we can derive that the constant has this type in its declared context.

$$\frac{}{\Gamma_c \vdash c : \alpha_c} \ {}^{\text{Const}}_{(c\, :\, \alpha_c\ \Gamma_c \,\in\, \Sigma)}$$

Recall that explicit constants could be defined in a context, but for the primitive and implicit constants $\Gamma_c$ will be the empty context. For any valid theory, we know that $\alpha_c$ is a correct type in $\Gamma_c$.

This rule is not part of the general framework, but since the idea is to define constant definitions in a theory and then use these constant, we have added this rule.

A function is constructed by abstracting the last variable from the context:

$$\frac{[\Gamma, x : \alpha] \vdash b : \alpha'}{\Gamma \vdash [x]b : \alpha \to [x]\alpha'} \text{ Abs}$$

The abstracted variable is removed from the context, so it must be the last variable to ensure well-formedness of the new context. The type of the function is a function type from the type of the abstracted variable, to a family of types depending on this variable.

We can apply an object which is of function type to an argument of the proper type:

$$\frac{\Gamma \vdash f : \alpha \to \beta \quad \Gamma \vdash a : \alpha}{\Gamma \vdash fa : \beta a}\ \text{App}$$

Here we can see that the result type of the application may depend on the argument, since the family $\beta$ is also applied to the argument $a$ yielding the type $\beta a$.

Since types can contain terms, and terms can contain variables assigned by the substitution, we must apply a substitution both to the term and the type of a typing judgement. Hence we have the rule:

$$\frac{\Gamma \vdash a : \alpha \quad \Delta \vdash \gamma : \Gamma}{\Delta \vdash a\gamma : \alpha\gamma}\ \text{Subst}$$

The above rules are all structural, that is the outer form of the term in the conclusion reflects the rule which is just applied. That is not the case for the next two rules; the type conversion rule, which only changes the type, and the thinning rule which only changes the context. One cannot see in the structure of the term that any of these rules are applied. This creates some problems concerning type checking, since we cannot know by simply examine the term if any of these two rules need to be applied. This problem will be discussed in section 4.2.1.

The type conversion rule

$$\frac{\Gamma \vdash a : \alpha \quad \Gamma \vdash \alpha = \alpha' \ : \ Type}{\Gamma \vdash a : \alpha'}\ \text{TConv}$$

says that a term which is of type $\alpha$ also has any other type $\alpha'$ which is equal to $\alpha$. This rule is important for the application rule, for instance, since the type of an argument may not have exactly the same form as the domain type of the function, but the two types are still equal. For example, we can have a function

$f : (P(0))\beta$

which takes an argument of type $P(0)$, where $P$ is a predicate of over natural numbers, and an argument which has the type

$a : P(0 + 0)$

which has not the same form but which are equal. To be able to apply the application rule we can use the type conversion rule so that the argument type and domain type match exactly.

The last term rule is the thinning rule:

$$\frac{\Gamma \vdash a : \alpha \quad \Gamma \preceq \Delta}{\Delta \vdash a : \alpha}\ \text{Thin}$$

The remaining rules are all about equality judgements. The presentation here will not be complete, but we will try to give the general ideas. For instance, all structural rules will be left out. The structural equality rules states that if the respective parts of two compound expression are equal, then the two expressions are equal. For instance if two functions are equal and the two arguments are equal, then the respective applications are equal.

**Type Equality rules**

Since a type family is a function, it can be applied to objects and the result is a type. The $\beta$-rule says that a function applied to an argument is the same as the function body where the bound variable is assigned to the argument.

One of the points of the substitution calculus is that substitutions are never pushed inside a binder, but rather stays as a closure to the function, since that is when the problem of capturing variables appear. This means that we can never "get rid of" a substitution applied to an abstraction until the function is applied to an argument. Hence, in general we will have an abstraction applied to a substitution which then is applied to an argument. This is the $\beta Subst$-rule below, and the ordinary $\beta$-rule

$$\frac{[\Gamma, x : \alpha'] \vdash \alpha : \mathit{Type} \quad \Gamma \vdash a : \alpha'}{\Gamma \vdash ([x]\alpha)a = \alpha\{x{:=}a\} : \mathit{Type}}\ \beta$$

is just a special case of that rule, so it can be derived. We include it since it is such an important rule. The premisses to the rule ensures the type correctness of the application which also means that the function body applied to the substitution is type correct.

The following example illustrates how the $\beta Subst$-rule works, when we direct is to the right so it becomes a computation rule. Consider the function

$[x]([x]x)$

which takes two arguments and return the second argument. We use the same bound variable on purpose, to show that when we apply the function to two arguments, they appear in the proper order in the substitution. If we apply the function above to arguments $a$ and $b$, we get

$$(([x]([x]x))\ a)\ b.$$

Since the function-part of the outermost application is a $\beta$-redex, we must compute it first (by the $\beta$-rule), yielding

$$(([x]x)\{x{:=}a\})\ b.$$

Now we have an instance of the $\beta Subst$-computation rule, and according to the rule it should compute to

$$x\{x{:=}a, x{:=}b\}$$

which becomes $b$ when the variable $x$ is looked up in the substitution $\{x{:=}a, x{:=}b\}$. We can see that the updating of the substitution makes sure that the scope of the bound variables are preserved after the application of the rule.

Again, the premisses guarantees the well-formedness of the included parts of the conclusion.

$$\frac{[\Gamma, x : \alpha'] \vdash \alpha\ :\ Type \quad \Delta \vdash \gamma : \Gamma \quad \Delta \vdash a : \alpha'\gamma}{\Delta \vdash (([x]\alpha)\gamma)a = \alpha\{\gamma, x{:=}a\}\ :\ Type}\ \beta\text{Subst}$$

As we have seen, a substitution can be distributed inside the function-type constructor without problem, so we have the following distributivity rule:

$$\frac{\Gamma \vdash \alpha\ :\ Type \quad \Gamma \vdash \beta : \alpha \to Type \quad \Delta \vdash \gamma : \Gamma}{\Delta \vdash (\alpha \to \beta)\gamma = \alpha\gamma \to \beta\gamma\ :\ Type}\ \text{FunDistr}$$

There is also a corresponding distributivity rule for an application of a type to an object. The only construction in which the substitution can not be distributed, is an abstraction.

Recall the example of compositions of substitutions in section 4.1, where we show an example of composition of substitutions. There we saw that

$$(f(x, y)\{x{:=}y\})\{y{:=}b\}$$

gives the same result as

$$f(x, y)(\{x{:=}y\}\{y{:=}b\})$$

namely

$$f(x, y)\{y{:=}b, x{:=}b\}.$$

This was an instance of the associativity rule for terms applied to substitutions, which corresponds to the following rule for types:

$$\frac{\Delta \vdash \alpha\ :\ Type \quad \Gamma \vdash \delta : \Delta \quad \Psi \vdash \gamma : \Gamma}{\Psi \vdash (\alpha\delta)\gamma = \alpha(\delta\gamma)\ :\ Type}\ \text{Assoc}$$

Beside these rules, there are some simple rules which says that applying a substitution to *Set* has no effect, neither has the empty substitution any effect when it is applied to a type or a term.

**Family Equality rules**

The next rule is an extensionality rule, which says that if two type families are equal when they are applied to a variable $x$, that is equal as types, then they are equal also as families:

$$\frac{[\Gamma, x : \alpha] \vdash \beta x = \beta' x \ : \ Type}{\Gamma \vdash \beta = \beta' : \alpha \rightarrow Type} \ {\scriptstyle \text{Ext} \atop (x \notin Dom(\Gamma))}$$

Note that, despite the name of the rule, this is a very weak rule. Again, thinking of the family as a function, the only thing the rule says is that if the bodies of the two functions are equal *when we know nothing about its argument*, then the functions are equal. Usually functions cannot be computed without knowing the argument, and therefore the function bodies must be essentially the same. This is very different from functions which are extensionally equal in the usual set theoretic sense, that is functions which give the same result for any argument.

The $\eta$-rule

$$\Gamma \vdash \beta = [x](\beta x) : \alpha \rightarrow Type$$

can be derived from the extensionality rule and the abstraction rule.

We also have an associativity rule for type families analogue to the associativity rule for types. As mentioned, the distributivity rule does not hold for type families.

**Term Equality rules**

Most equality rules for the terms are very similar to the corresponding rules for types, or for families of types for the particular rules concerning functions. For instance we have the distributivity rule, the associativity rule, the extensionality rule and the two $\beta$-rules. These rules can be found in the appendix.

There are two new rules, which concern variables applied to substitutions. We think it is again easier to read the rules as computation rules. Then the first rule

$$\frac{\Gamma \vdash \delta : \Delta \quad \Gamma \vdash a : \alpha\delta}{\Gamma \vdash x\{\delta, x{:=}a\} = a : \alpha\delta} \ {\scriptstyle \beta_1}$$

is simply looking up the variable $x$ in the stack, in which the top element is the variable we are looking for, that is $x$ assigned to $a$.

The other rule

$$\frac{\Gamma \vdash \delta : \Delta \quad \Gamma \vdash a : \alpha\delta}{\Gamma \vdash y\{\delta, x{:=}a\} = y\delta : \alpha'\delta} \ {\scriptstyle \beta_2 \atop (y \,:\, \alpha' \,\in\, \Delta)}$$

concerns the case when the top element is not the variable we are looking for, so then we remove the top element and continue looking for $y$ in the rest of the stack.

Finally, we describe the equality rules for substitutions.

**Substitution Equality rules**
The first two rules says that the composition of a substitution and an empty substitution, has no effect.

$$\frac{\Gamma \vdash \delta : \Delta}{\Gamma \vdash \{\}\delta = \delta : \Delta}\ \mathrm{Empty}_L \qquad \frac{\Gamma \vdash \delta : \Delta}{\Gamma \vdash \delta\{\} = \delta : \Delta}\ \mathrm{Empty}_R$$

The distributivity rule for substitutions explains how composition is computed together with the left Empty-rule above.

$$\frac{\Gamma \vdash \delta : \Delta \quad \Gamma \vdash a : \alpha\delta \quad \Psi \vdash \gamma : \Gamma}{\Psi \vdash (\{\delta, x{:=}a\})\gamma = \{\delta\gamma, x{:=}a\gamma\} : [\Delta, x : \alpha]}\ \mathrm{Distr}$$

The substitution $\gamma$ is pushed through and applied to all the assignments in $\delta$ until the end where $\gamma$ is placed.

The next rule simply says that composition is associative.

$$\frac{\Delta \vdash \theta : \Theta \quad \Gamma \vdash \delta : \Delta \quad \Psi \vdash \gamma : \Gamma}{\Psi \vdash (\theta\delta)\gamma = \theta(\delta\gamma) : \Theta}\ \mathrm{Assoc}$$

The last substitution rule

$$\frac{\Gamma \vdash \delta : \Delta \quad \Gamma \vdash a : \alpha\delta}{\Gamma \vdash \{\delta, x{:=}a\} = \delta : \Delta}\ \begin{array}{l}\beta_0\\ (x \notin \Delta)\end{array}$$

may seem a bit strange, but what is does is removing superfluous assignments in the substitution which we may have introduced by the target thinning rule. The reason we can simply forget these assignments is that since the substitution is typed with context $\Delta$, it can only be applied to a term or a type which are valid in $\Delta$. Thus the assignments of additional variables can never be used.

**General rule schemas** The general thinning rule for any judgement form $\mathcal{J}$:

$$\frac{\Gamma \vdash \mathcal{J} \quad \Gamma \preceq \Delta}{\Delta \vdash \mathcal{J}}\ \mathrm{Thin}$$

and the general rule of applying a substitution to (the parts of) a judgement:

$$\frac{\Gamma \vdash \mathcal{J} \quad \Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathcal{J}\gamma}\ \mathrm{Subst}$$

Reflexivity, symmetry and transitivity hold for all equality relations.

### 4.2.1 Problematic rules for type checking

Usually inference rules are read in the direction from premisses to conclusion. However, when we are interested in type checking we read the rules the other way around - from conclusion to the premisses. The reason is that we have a term and an expected type, and we want to know if this term is of the expected type. What we can do is to analyse the term, and try to find out of which rule this could be the conclusion. For any term, we can always guess that the last rule is the structural rule corresponding to the form of the term. However, it may also come from the type conversion rule

$$\frac{\Gamma \vdash a : \alpha \quad \Gamma \vdash \alpha = \alpha' : \mathit{Type}}{\Gamma \vdash a : \alpha'}\ \text{TConv}$$

and if the context is non-empty, it could come from the thinning rule as well.

$$\frac{\Gamma \vdash a : \alpha \quad \Gamma \preceq \Delta}{\Delta \vdash a : \alpha}\ \text{Thin}$$

Neither of these rules leave any trace in the term, so we cannot from the term decide if we need to apply any of these rules.

If we would implement the typing rules exactly as they are presented in the previous section, we would get a non-deterministic, non-terminating algorithm. Non-deterministic since it we could not decide whether the structural rule, the type conversion rule or the thinning rule should be applied. Non-termination is possible since the type conversion rule is always applicable.

There are three kinds of problems; the problem of the type conversion rule, the problem of computing types and the problem connected with the thinning rule. The first problem can be solved rather easily, but the remaining to will give rise to restrictions in the type checking algorithm.

**Type conversion rule - solution**

The type conversion rule can be eliminated if we incorporate the convertibility check in the remaining rules. For instance, the assumption rule would instead be

$$\frac{\Gamma \vdash \alpha = \alpha' : \mathit{Type}}{\Gamma \vdash x : \alpha'}\ \begin{array}{l}\text{Ass}\\ (x : \alpha \in \Gamma)\end{array}$$

and the application rule

$$\frac{\Gamma \vdash f : \alpha \to \beta \quad \Gamma \vdash a : \alpha_1 \quad \Gamma \vdash \alpha = \alpha_1 : \mathit{Type} \quad \Gamma \vdash \beta a = \alpha_2 : \mathit{Type}}{\Gamma \vdash fa : \alpha_2}\ \text{App}$$

By modifying all rules in a similar way, we can get rid off the type conversion rule.

**Computing types - problem**

Another problem is that we do not know the type of $f$ in the above application rule. We mentioned earlier, in section 2.4, that it is not possible to compute the type of the head in an arbitrary application in the general case; it needs to be a variable or a constant. Hence, a $\beta$-redex, for instance, is not possible to type check given the type of the application only. The reason is that if $f$ is an abstraction $[x]b$, then we know

$$([x]b)a : \alpha_2$$

but we do not have the type $\alpha \rightarrow \beta$. We cannot always derive the type of the abstraction $[x]b$, since we only know that $[x]b$ must have a function type, which when applied to $a$ is equal to $\alpha_2$. In some cases the type of an abstraction can be inferred, but this problem has not always a unique solution, neither does a most general unifier always exist [Sal88].

If the abstraction was decorated with a type as in, for instance, the Calculus of Constructions, the type can always be inferred. Hence, in Coq and LEGO the type of a term can be inferred. However, the restriction is not a problem in practice.

A similar argument can be used for the (modified) substitution rule

$$\frac{\Gamma \vdash a : \alpha \quad \Delta \vdash \gamma : \Gamma \quad \Delta \vdash \alpha\gamma = \alpha' \,:\, \mathit{Type}}{\Delta \vdash a\gamma : \alpha'}\ \text{Subst}$$

Assume we can compute the type of $a$ and the context $\Gamma = [x_1 : \alpha_1, \ldots, x_n : \alpha_n]$ of $\gamma = \{x_1{:=}a_1, \ldots, x_n{:=}a_n\}$ from the information in the conclusion $\Delta \vdash a\gamma : \alpha'$. Then we know the types of $[x_1, \ldots, x_n]a$ and also the types of $a_1, \ldots, a_n$ which are the same as for the variables. Hence, we could type a $\beta$-redex just by performing the $\beta$-rule and then compute the type of the function this way. Since we know that this is not possible, we can not compute the type for the parts in a substitution either without knowing the types of the variables, i.e. the context of the substitution.

There is a special case when we know the context of the term $a$, and that is when $a$ is an explicitly defined constant since it must be defined in the theory. Hence we know both the type and the context of the constant. In this case it is possible to type check the constant applied to a substitution, provided the substitution is restricted to the variables in the local context of the constant.

**Thinning rule - problem**

If we would have a type checking rule corresponding to the thinning rule, then there would be two problems

1. *when* to apply the rule, and
2. *which* sub-context to choose.

There are only two rules which changes the context of the judgements in a derivation, the thinning rule and the abstraction rule. Going the direction of type checking (conclusion to premisses), the abstraction rule *extends* the context and the thinning rule *decreases* the context. The problem occurs when we are in the situation

$$\frac{???}{[\Gamma, x : \alpha', \Delta] \vdash [x]b : \alpha \to \beta}$$

that is we need to extend the context to check the function body $b$; but if we extend the context it is no longer a valid context since $x$ already occurs in it. On the other hand, to be able to type check the body of the function, we need to extend the context. We could try to rename the bound variable $x$, but this operation would not be justified by the rules of the substitution calculus. The derived $\alpha$-conversion rule in the this calculus is weaker than informulations of type theory without explicit substitution; it says:

$$\frac{[\Gamma, x : \alpha_1] \vdash b : \alpha_2}{\Gamma \vdash [x]b = [y](b\{x{:=}y\}) : \alpha_1 \to [x]\alpha_2} \; y \notin Dom(\Gamma)$$

The usual restriction is that $y$ does not occur free in $b$, which is not the same as $y \notin Dom(\Gamma)$, even if the context $\Gamma$ is the smallest possible context. The reason is that variables may occur free in the type of $b$ without occurring in $b$. Even worse, the term may depend on a variable in $\Gamma$ which does neither occur free in the term nor in its type. In section 4.4, we will give examples of these two situations.

In [MP93], the problem of $\alpha$-conversion is solved by using the idea presented in [Coq91], which is to separate free variables (called parameters) from bound variables. The disadvantage is that in the abstraction rule, the bound variable must be replaced by a fresh parameter, i.e. it must be renamed. Since one of the points with the substitution calculus is to avoid renaming, this solution is not really satisfactory for the substitution calculus.

A simple solution is to not use the thinning rule at all. This means that we will only extend the context during type checking and never decrease the context. Then the above example could not be type checked, since it requires that all bound variables within the same scope are distinct and different from the context. This is the solution we will use in our type checking algorithm.

## 4.3 Meta theory assumptions

The type checking algorithm presented in chapter 5 is proved correct assuming the meta properties of the substitution calculus stated below. The following basic meta properties are assumed:

1. To be able to decompose a term on constructor form in the conversion algorithm, we need the assumption that constructors are one-to-one. We know by the equality rules that

$$\frac{\begin{array}{c} a_1 = a_1' \\ \vdots \\ a_n = a_n' \end{array}}{c(a_1, \ldots, a_n) = c(a_1' \ldots, a_n')}$$

   but we also need the converse, that is

$$\frac{c(a_1, \ldots, a_n) = c(a_1' \ldots, a_n')}{\begin{array}{c} a_1 = a_1' \\ \vdots \\ a_n = a_n' \end{array}}$$

   to be able to simplify equations where both terms are on constructor form, in the conversion algorithm.

2. We need to know that equal, ground terms have the same outermost constructor form, that is

   $c(a_1, \ldots, a_n) \neq c'(b_1 \ldots, b_m)$ if $c = c'$.

3. We need to assume that if $\Gamma \vdash a = b : \alpha$, then $a$ and $b$ have the same $\beta$-normal, $\eta$-expanded form, and that the computation to $\beta$-normal, $\eta$-expanded form terminates.

The last property is about computation of terms, and contains in particular the normalisation of the normal-order reduction strategy used here. We know from a counter example in [Mel94] that simply typed $\lambda$-calculus with explicit substitution is not strongly normalisable, and that the counter example can be adopted to the substitution calculus of type theory [Rit]. However, it does not apply to the reduction algorithm presented here.

## 4.4 Problem with $\alpha$ conversion

The notion of $\alpha$-conversion in the calculus with explicit calculus is weaker than the usual rule of $\alpha$-conversion, but this is not the case for Martin-Löf's type theory in general. The rule which is derivable in the calculus is

$$\frac{[\Gamma, x : \alpha] \vdash b : \beta}{\Gamma \vdash [x]b = [y](b\{x{:=}y\}) : \alpha \to [x]\beta}\ {}^{\alpha}_{(y \notin \Gamma)}$$

Here, $y$ is required not to be in $\Gamma$, which means that $b$ cannot depend on $y$ in any way. We will see in the following two counter examples that "depends on" can mean that the variable $y$ occur only in the type $\beta$, or even that $y$ does not occur in either the term or the type, but that other variables which do occur depend on the variable $y$. The usual formulation is that $y$ does not occur free in the term $b$ which clearly is a less restrictive condition.

We will first give an example (similar to the one given in [Pol93]) of two $\alpha$-convertible terms, where the first is derivable, whereas the second is not. Assume we have a constant $N$ of type $Set$ and a constant $Seq$ of type $N \to Set$ in our environment. Then $[x]([y]y)$ has type $N \to [x](Seq(x) \to Seq(x))$ by the derivation:

$$\cfrac{\cfrac{\cfrac{}{[x : N, y : Seq(x)] \vdash y : Seq(x)}\ \text{Ass}}{[x : N] \vdash [y]y : Seq(x) \to Seq(x)}\ \text{Abs}}{\vdash [x]([y]y) : N \to [x](Seq(x) \to Seq(x))}\ \text{Abs}$$

and $[x : N, y : Seq(x)]$ is a valid context, since $x$ and $y$ are distinct, $N$ is a type and $Seq(x)$ is a type when $x : N$.

**Counterexample 1** *Assume we have a constant N of type Set and a constant Seq of type N→Set in our environment. Then we have*

$$\vdash [x]([y]y) : N \to [x](Seq(x) \to Seq(x))$$

*but*

$$\not\vdash [x]([x]x) : N \to [x](Seq(x) \to Seq(x))$$

*i.e. the last judgement is not derivable.*

***Proof:*** Assume $\vdash [x]([x]x) : N \to [x](Seq(x) \to Seq(x))$. Then by lemma 1.1 below, we have that

$$[x : N] \vdash [x]x : Seq(x) \to Seq(x).$$

If we try to apply the lemma again, we must choose a sub-context of $[x : N]$ not containing $x$, so the empty context is the only possibility. But then the extension of the new abstracted variable $x$ gives the context $[x : Seq(x)]$ which is not a valid context since $Seq(x)$ is not a valid type in the empty context. Hence,

$$[x : N] \nvdash [x]x : Seq(x) \rightarrow Seq(x)$$

so we have a contradiction. $\square$

**Lemma 1.1** *If* $\Gamma \vdash [x]e : \alpha_1 \rightarrow [x]\alpha_2$*, then* $[\Delta, x : \alpha_1] \vdash e : \alpha_2$ *for some* $\Delta \preceq \Gamma$.

***Proof:*** Induction on the length of the derivation of $\Gamma \vdash [x]e : \alpha_1 \rightarrow [x]\alpha_2$. There are only three possible rules for the last step; the abstraction rule, the thinning rule and the type conversion rule. The type conversion rule only effects the type, so the term and the context is not changed. Thus this case holds directly by the induction hypothesis. If the last step is the abstraction rule, then $x$ does not occur in $\Gamma$ so we have $[\Gamma, x : \alpha_1] \vdash e : \alpha_2$. If the last step is the thinning rule, we have $\Delta \vdash [x]e : \alpha_1 \rightarrow [x]\alpha_2$, for some $\Delta \preceq \Gamma$, and it follows by the induction hypothesis that $[\Delta, x : \alpha_1] \vdash e : \alpha_2$. $\square$

In this example the problem was that a variable occured free in the type which made it impossible to derive. The next example is the following

**Counterexample 2**
*We have*

$$[X : Set, Y : X \rightarrow Set, x : X] \vdash [Z]Y(x) : \alpha \rightarrow Set$$

*but if we change the name of the bound variable* $X$*, then*

$$[X : Set, Y : X \rightarrow Set, x : X] \nvdash [X]Y(x) : \alpha \rightarrow Set.$$

***Proof:*** Assume $[X : Set, Y : X \rightarrow Set, x : X] \vdash [X]Y(x) : \alpha \rightarrow Set$. By lemma 1.1, we have that $[X : \alpha] \vdash Y(x) : Set$, since $[\,]$ is the largest sub-context not containing the variable $X$. Hence we have a contradiction since $Y(x)$ is not a valid term in the empty context. $\square$

Note that in this example the problematic variable $X$ does not even occur in the term or the type. However, the variables $Y$ and $x$ both depend on $X$, so no variables can be removed from the context. Neither can the context be extended by applying the abstraction rule, so we are stuck. Hence, this term cannot be derivable either, even though any $\alpha$-convertible term is derivable.

As we have seen, both these counter examples rely on the dependent types, since the reason we cannot rename the bound variable is that the same variable occurs in the type or the variables occurring in the term depends on the bound variable. Hence, we cannot remove the variable from the context.

For the simply typed case we do not have this problem, since types cannot depend on variables and, hence, there is no order on the context. Thus, any variable which does not occur free in the term can be removed from the context, and we do not get this problem of $\alpha$-conversion.

# Chapter 5

# Judgement checking

Our main objective is to define a type checking algorithm, that is to to check a judgement of the form

$\Gamma \vdash a : \alpha$

which has as prerequisite that $\Gamma$ is a valid context and $\alpha$ is a valid type in the context $\Gamma$, thereby requiring algorithms for checking the judgements

$\Gamma : \mathit{Context}$, and
$\Gamma \vdash \alpha : \mathit{Type}$.

Moreover, the type conversion rule

$$\frac{\Gamma \vdash a : \alpha \quad \Gamma \vdash \alpha = \alpha' : \mathit{Type}}{\Gamma \vdash a : \alpha'}\ \text{TConv}$$

involves the judgement

$\Gamma \vdash \alpha = \alpha' : \mathit{Type}$

and type equality depends on term equality. This we can see in, for instance, the special case of the $AppEq$-rule where the family is $El$;

$$\frac{\Gamma \vdash A = B : \mathit{Set}}{\Gamma \vdash \mathit{El}(A) = \mathit{El}(B) : \mathit{Type}}\ \text{ElEq}$$

which is a rule we will use in the type conversion algorithm.

In general, an algorithm for checking one judgement form involves all the other judgement forms, since all judgements may be relative a context which must be valid. Thus, we have the following picture of dependency, where

$CF, TF, TC, TConv$ and $Conv$ are the algorithms for checking the corresponding judgement forms, and where $\Downarrow$ denotes that the algorithm "uses" or calls to algorithm it points to. The requirements to the right of each algorithm are the preconditions which must be satisfied:

$$
\begin{array}{ll}
CF: & \Gamma : \mathit{Context} \\
\Downarrow & \\
TF: & \Gamma \vdash \alpha : \mathit{Type} \text{ (requires } \Gamma : \mathit{Context}) \\
\Downarrow & \\
TC: & \Gamma \vdash a : \alpha \text{ (req. } \Gamma : \mathit{Context}, \Gamma \vdash \alpha : \mathit{Type}) \\
\Downarrow & \\
TConv: & \Gamma \vdash \alpha = \alpha' : \mathit{Type} \text{ (req. } \Gamma : \mathit{Context}, \Gamma \vdash \alpha : \mathit{Type}, \Gamma \vdash \alpha' : \mathit{Type}) \\
\Downarrow & \\
Conv: & \Gamma \vdash a = b : \alpha \text{ (req. } \Gamma : \mathit{Context}, \Gamma \vdash \alpha : \mathit{Type}, \Gamma \vdash a : \alpha, \Gamma \vdash b : \alpha)
\end{array}
$$

The type checking algorithm we will present here, is a *general* and *modular* algorithm, which can be used for type checking various calculi, and it can easily be extended to checking *incomplete* terms, which we will show in chapter 6. The type checking is divided in two parts, one which generates a list of equations ($GE$ for $G$enerate $E$quations) and one which checks if the equations holds by simplifying the equations ($Simple$). The equations are relating two terms in a given type and context. The idea that type checking can be reduced to checking a set of equations, was present already in Automath [dB87]. An overview of the algorithms is described by the following picture:

$$
e : \alpha\ \Gamma \quad \overset{GTE}{\Longrightarrow} \quad
\left|
\begin{array}{c}
\alpha_1 = \alpha'_1\ \Gamma_1 \\ \\ \vdots \\ \\ \alpha_n = \alpha'_n\ \Gamma_n
\end{array}
\right|
\begin{array}{c}
\overset{TConv}{\longrightarrow} \\ \\ \\ \\ \overset{TConv}{\longrightarrow}
\end{array}
\left.
\begin{array}{c}
\left[
\begin{array}{c}
a_1^1 = b_1^1 : \alpha_1^1\ \Gamma_1^1 \\ \vdots \\ a_1^k = b_1^k : \alpha_1^k\ \Gamma_1^k
\end{array}
\right]
\begin{array}{c}
\overset{Conv}{\longrightarrow}\ []/F \\ \vdots \\ \overset{Conv}{\longrightarrow}\ []/F
\end{array} \\
\vdots \\
\left[
\begin{array}{c}
a_n^1 = b_n^1 : \alpha_n^1\ \Gamma_n^1 \\ \vdots \\ a_n^p = b_n^p : \alpha_n^p\ \Gamma_n^p
\end{array}
\right]
\begin{array}{c}
\overset{Conv}{\longrightarrow}\ []/F \\ \vdots \\ \overset{Conv}{\longrightarrow}\ []/F
\end{array}
\end{array}
\right\}
[]/F
$$

```
                  ------------------>    ----------------->
                       TSimple                 Simple
---------------------------------->
       GE = TSimple o GTE
------------------------------------------------------->
                     TC = Simple o GE
```

As mentioned, we have as a prerequisite for the type checking algorithm that $\alpha$ is a proper type in $\Gamma$.

We will show (in proposition 9) that

If $GE(e, \alpha, \Gamma) \Rightarrow \mathcal{C}$, then $\Gamma \vdash e : \alpha \Leftrightarrow \mathcal{C}$ holds

where $\mathcal{C}$ is a list of term equations. This property holds whether or not the equations are decidable, but in our case they are decidable and the algorithm *Simple* checks if all equations in $\mathcal{C}$ hold, and hence we get a decision procedure where an empty list of equations is interpreted as true, and when the algorithm reports a failure (due to that GTE fails or some equation does not hold) we interpret this as false. Thus, the main results of this chapter is the soundness theorem 1 and the completeness theorem 2 of the type checking algorithm, which are presented in the end of this chapter.

The generation of term equations is done in two steps, first a list of type equations are generated (by $GTE$) and then this list is transformed into a list of term equations by the algorithm $TSimple$. $TSimple$ applies $TConv$ to each type equation yielding a list of term equations. In a similar manner, the *Simple* algorithm checks each term equation by calling $Conv$, which in turns returns an empty list if the equation holds, and a failure ($F$) otherwise. The point of letting the conversion algorithm return a list of equations as result rather than simply true or false, becomes apparent when we extend the algorithm to incomplete terms. Then all equations may not be possible to decide due to placeholders occuring in the equations, and we get a *unification problem*, i.e. a list of equations containing placeholders as unknown objects.

The first part of the type checking, the generation of equations, does not require the calculus to be normalising since all term reduction takes place in the second part of the algorithm. Even though types are reduced in type conversion, we can give a syntactic measure of the type reduction which guarantees termination. In the conversion however, terms are reduced, and to assure termination the calculus must be normalising for the reduction strategy. Moreover, we must impose an order in which *Simple* simplifies the equations, to guarantee that the equations are well-typed.

One of the reasons to define the type checking algorithm this way is that it can easily be extended to handle incomplete terms as well as complete ones. We will shortly present the extension here already, to motivate our choice. The detailed description of the modified type checking algorithms for incomplete terms is presented in the next chapter. When we extend the algorithms to incomplete terms, the algorithms are extended by rules for handling placeholders (denoted $?_1, \ldots$), and we get the following picture of type checking the incomplete term $e$ with (complete) type and context $\alpha$ and $\Gamma$, respectively:

$$
e : \alpha\ \Gamma \quad \overset{GTE_p}{\Longrightarrow} \quad
\left|\begin{array}{c} \alpha_1 = \alpha_1'\ \Gamma_1 \\ \\ \vdots \\ ?_m : \alpha_m\ \Gamma_m \\ \vdots \\ \\ \alpha_n = \alpha_n'\ \Gamma_n \end{array}\right|
\begin{array}{c} \xrightarrow{TConv_p} \\ \\ \\ \longrightarrow \\ \\ \\ \xrightarrow{TConv_p} \end{array}
\left.\begin{array}{c} \begin{bmatrix} a_1^1 = b_1^1 : \alpha_1^1\ \Gamma_1^1 \\ \vdots \\ a_1^k = b_1^k : \alpha_1^k\ \Gamma_1^k \end{bmatrix} \\ \vdots \\ ?_m : \alpha_m\ \Gamma_m \\ \vdots \\ \begin{bmatrix} a_n^1 = b_n^1 : \alpha_n^1\ \Gamma_n^1 \\ \vdots \\ a_n^p = b_n^p : \alpha_n^p\ \Gamma_n^p \end{bmatrix} \end{array}\right\}
\begin{array}{l} \text{typed} \\ \text{unification} \\ \text{problem} \end{array}
$$

$$
\overset{\phantom{x}}{\dashrightarrow}_{\;TSimple_p}
$$

$$
\dashrightarrow_{\;GE_p \,=\, TSimple_p \,\circ\, GTE_p}
$$

Here, the modified algorithm for generating equations ($GTE_p$) produces a list of type equation together with typing constraints

$$?_m : \alpha_m \quad \Gamma_m$$

on the placeholders occuring in the incomplete term $e$. Since placeholders stand in place of terms, the $TSimple_p$ algorithm will simplify the incomplete type equations as before and leave the typing constraints unchanged. Thus, $GE_p$ produces a list of incomplete term equations together with typing constraints, which we will call a *typed unification problem*. The equations in the typed unification problem may be possible to simplify, but in general there will be equations which can not be simplified any further due to the unknown placeholders. The remaining equations are *constraints* on future instantiations of the placeholders occurring in the unification problem. Thus, we have the corresponding type checking algorithm for incomplete terms

$$TC_p = Simple_p \circ GE_p.$$

We will show the corresponding property for incomplete terms $e$ (containing distinct placeholders)

If $TC_p\ ([\,], e, \alpha, \Gamma) \Rightarrow \mathcal{C}$, then $\Gamma \vdash e\sigma : \alpha \Longleftrightarrow \mathcal{C}\sigma$ holds

where $\sigma$ is a complete instantiation of the placeholders in $\mathcal{C}$. This will be explained in detail in section 6.3.

The algorithms will be computed in an *environment*, where constants are declared. Since the environment stays constant during the judgement checking, we will assume a valid environment throughout this presentation. The environment is denoted by $\Sigma$ below.

The algorithms which will be presented are functions even though we present them as inductive relations

$$\mathcal{F}(A) \Rightarrow R$$

where $A$ is the list of arguments and $R$ is the result of applying the function $\mathcal{F}$. We will sometimes use the functional notation $\mathcal{F}(A)$ to denote the result $R$.

## 5.1 Checking Contexts and Types

A context is checked to be valid by extending the context with the variable declarations one by one and checking that the variable name is fresh and that the type of the variable is valid in the previous context. Hence, it follows exactly the definition of a valid context:

**Definition 5.1** A *valid context* is inductively defined by the following rules corresponding to the context formation rules

$$\frac{}{Valid\ [\,]}\ \text{CF-empty} \qquad\qquad \frac{Valid\ \Gamma \quad \Gamma \vdash \alpha : Type}{Valid\ [\Gamma, x : \alpha]}\ \begin{array}{l}\text{CF-ext}\\ (x \notin Dom(\Gamma))\end{array}$$

where the domain of a context $\Gamma$ (written $Dom(\Gamma)$) is the set of variable names declared in $\Gamma$.

Before we describe the type formation algorithm, we need to define the restrictions we must impose on terms (and hence types) in the type checking and type formation algorithms. There are two restrictions:

1. Terms are required to be $\beta S$-normal, that is there are no $\beta$-redexes in the term and substitutions are only applied to constants.
2. The bound variables in the term or the type must be distinct and different from the variables of the context in which the term or type is check relative to.

As motivation of the restrictions, recall the section 4.2.1 which describes the rules of the calculus which are problematic for type checking.

A $\beta S$-normal term may contain substitutions which are required to be normal relative to a context:

**Definition 5.2** Let $\Gamma$ be a context. A $\Gamma$-*normal* substitution is defined inductively by

$$\frac{}{\{\,\}\text{ is }[\,]\text{-normal}} \qquad \frac{\gamma \text{ is } \Gamma\text{-normal}}{\{\gamma, x{:=}a\} \text{ is } [\Gamma, x : \alpha]\text{-normal}}$$

The reason we require substitutions to be normal, is that if the substitution contains "too many" assignments, we do not have the type information of these variables, and hence we can not check these assignments. This violates the property that a term is well-typed if and only if all its components are well-typed, and therefore we will not allow such substitutions. On the other hand, in practice the substitution need not assign every variable in the context, since we use the convention that a variable which is not explicitly given an assignment is assigned to itself.

**Definition 5.3** A $\beta S$-normal term is defined by the following grammar

$$\begin{array}{lcl} e_1^{\beta S} & ::= & [x]e_1^{\beta S} \mid e_2^{\beta S} \\ e_2^{\beta S} & ::= & x \mid c \mid (e_2^{\beta S}\ e_1^{\beta S}) \mid c\gamma^{\beta S} \end{array}$$

where $\gamma^{\beta S}$ is normal relative to the local context of $c$.

A $\beta S$-normal substitution is defined by

$$\gamma^{\beta S} ::= \{\,\} \mid \{\gamma^{\beta S}, x{:=}e_1^{\beta S}\}$$

Analogously, we have the following definition for types:

**Definition 5.4** A $\beta S$-normal type is of the following form

$$\begin{array}{lcl} \alpha^{\beta S} & ::= & \mathit{Set} \mid \alpha^{\beta S} \to \beta^{\beta S} \mid \mathit{El}(e^{\beta S}) \\ \beta^{\beta S} & ::= & \mathit{El} \mid [x]\alpha^{\beta S} \end{array}$$

We also need to define the restriction on bound variables in terms and types to avoid the problem of alpha conversion.

**Definition 5.5** Let $l$ be a list of identifiers. A term (type) is defined to be *l-distinct* by the following rules

- constants and variables are $l$-distinct
- $[x]e$ ($[x]\alpha$) is $l$-distinct if $x \notin l$ and $e$ ($\alpha$) is $(x, l)$-distinct
- $(fe)$ $(\beta e)$ is $l$-distinct if both $f$ $(\beta)$ and $e$ are $l$-distinct

- $e\gamma$ ($\alpha\gamma$) is $l$-distinct if both $e$ ($\alpha$) and $\gamma$ are $l$-distinct
- $\gamma$ is $l$-distinct if all its assigned terms are $l$-distinct
- $\alpha \rightarrow \beta$ is $l$-distinct if both $\alpha$ and $\beta$ are $l$-distinct

Thus, a $l$-distinct term is a term in which the bound variables are mutually distinct and different from the variables in $l$.

**Definition 5.6** A term (type) is said to be $\Gamma$-*distinct* if the term (type) is distinct relative to $Dom(\Gamma)$.

The Type Formation algorithm ($TF$), takes as input a type and a context and it returns a list of type equations or a failure. However, the list of type equations will always be an empty list, denoting that the input is a valid type in the given context. Thus, seen as a decision procedure, the empty list denotes "yes" and a failure denotes "no". We will not actually use the list for anything useful until we extend the algorithm to incomplete types. The algorithm is inductively defined on the structure of the type, and it calls the type checking algorithm $TC$ to check that the argument of the type constructor *El* is of type *Set*. We must require the type to be $\beta S$-normal since $TF$ is defined in terms of the type checking algorithm in which this requirement is imposed. The terms in a type are $\beta S$-normal precisely when the type itself is $\beta S$-normal.

**Type Formation** $TF(\alpha,\Gamma) \Rightarrow \xi$, where $\xi$ is a list of equations.
Preconditions:

1. $\Gamma$ is a valid context
2. $\alpha$ is $\beta S$-normal
3. $\alpha$ is $\Gamma$-distinct

$$\frac{}{TF(Set, \Gamma) \Rightarrow [\,]}\ \text{TF-Set} \qquad\qquad \frac{TC(A, Set, \Gamma) \Rightarrow \xi}{TF(El(A), \Gamma) \Rightarrow \xi}\ \text{TF-El}$$

$$\frac{}{TF(Set \rightarrow El, \Gamma) \Rightarrow [\,]}\ \text{TF-Fun'}$$

$$\frac{TF(\alpha, \Gamma) \Rightarrow \xi_1 \quad TF(\alpha', [\Gamma, x : \alpha]) \Rightarrow \xi_2}{TF(\alpha \rightarrow [x]\alpha', \Gamma) \Rightarrow \xi_1 \,@\, \xi_2}\ \text{TF-Fun}$$

The rules should be understood in the following way; for example the TF-El means: to check if *El(A)* is a valid type in $\Gamma$, call *TC* to check *TC(A,Set,*$\Gamma$*)*. If *TC* computes to $\xi$ then *TF(El(A),*$\Gamma$*)* computes to $\xi$. The @-sign denotes the operation of appending two lists.

## 5.2 Type Checking

As already mentioned, type checking is defined in terms of $GTE$, $TSimple$ and $Simple$ with the definition

$$TC = Simple \circ GE = Simple \circ TSimple \circ GTE$$

and we will start by defining $GTE$, the generation of type equations.

The $GTE$-algorithm takes as input a term, a type and a context, and it returns a list of type equations. Here the result is really as list of equations, since the equations are only generated and not yet simplified. It proceeds by examining the term structure, and there is exactly one rule for each term construction. It is mutually defined with *CT* and *FC*. *CT* (*C*ompute *T*ype) takes a term of the form $b(a_1, \ldots, a_n)$ where $b$ is a variable or constant (assured by $\beta S$-normality). It returns the type of the term together with a list of equations. The type can always be computed, since the type of the head can simply be looked up, and then be successively applied to the arguments. *FC* (*F*its *C*ontext) checks that each assignment of the substitution is of proper type.

**Generate Type Equations** $GTE$(a,$\alpha$,$\Gamma$) $\Rightarrow \xi$, where $\xi$ is a list of type equations.
Preconditions:

1. $\Gamma$ is a valid context
2. $\Gamma \vdash \alpha$ : *Type*
3. $a$ is $\beta S$-normal (this implies that $f$ (in *CT*) and $\gamma$ (in *FC*) are $\beta S$-normal as well)
4. $a$ is $\Gamma$-distinct

First, the structural rules for the $GTE$-algorithm:

$$\frac{}{GTE(x, \alpha, \Gamma) \Rightarrow [\langle \alpha, \alpha', \Gamma \rangle]}\ \text{GTE-Var}\ (x : \alpha' \in \Gamma)$$

$$\frac{}{GTE(c, \alpha, \Gamma) \Rightarrow [\langle \alpha, \alpha', \Gamma \rangle]}\ \text{GTE-Const}\ (c : \alpha' \in \Sigma)$$

$$\frac{FC(\gamma,\Gamma_c,\Gamma) \Rightarrow \xi}{GTE(c\gamma,\alpha,\Gamma) \Rightarrow \xi \mathbin{@} [\langle \alpha_c\gamma,\alpha,\Gamma\rangle]}\ \text{GTE-Subst}\ {\scriptstyle (c\,:\,\alpha_c \text{ in } \Gamma_c \in \Sigma)}$$

$$\frac{GTE(b,\beta x,[\Gamma,x:\alpha]) \Rightarrow \xi}{GTE([x]b,\alpha\!\rightarrow\!\beta,\Gamma) \Rightarrow \xi}\ \text{GTE-Abs}$$

When we type check an application, we must be able to compute the head of the application. Thus, the restriction to $\beta S$-normal application-terms, which means that we know that the head of the application is always a variable or a constant. The constant may be applied to a substitution. In these cases, we can simply look up the type of the head. Then, the application is type correct if all arguments are type correct after the substitution of previous arguments in the argument types. Furthermore, we must check that the result type is the expected type of the application. This idea is reflected in the rule

$$\frac{\begin{array}{l} GTE(a_1,\alpha_1,\Gamma) \Rightarrow \xi_1 \\ GTE(a_2,\alpha_2\{x_1{:=}a_1\},\Gamma) \Rightarrow \xi_2 \\ \quad\vdots \\ GTE(a_n,\alpha_n\{x_1{:=}a_1,\ldots,x_{n-1}{:=}a_{n-1}\},\Gamma) \Rightarrow \xi_n \end{array}}{GTE(f(a_1,\ldots,a_n),\alpha,\Gamma) \Rightarrow \xi_1 \mathbin{@} \cdots \mathbin{@} \xi_n \mathbin{@} [\langle \alpha,\alpha_f\{x_1{:=}a_1,\ldots,x_n{:=}a_n\},\Gamma\rangle]}$$

where $f$ is of type $(x_1{:}\alpha_1)\cdots(x_n{:}\alpha_n)\alpha_f$.

However, the following application rule together with the rules which compute the type of the partially applied term (the $FC$-rules), is a formalisation of the more intuitive rule above.

$$\frac{CT(f,\Gamma) \Rightarrow \langle \xi_1,\alpha'\!\rightarrow\!\beta\rangle \quad GTE(e,\alpha',\Gamma) \Rightarrow \xi_2}{GTE(fe,\alpha,\Gamma) \Rightarrow \xi_1 \mathbin{@} \xi_2 \mathbin{@} [\langle \beta e,\alpha,\Gamma\rangle]}\ \text{GTE-App}$$

$$\frac{}{CT(x,\Gamma) \Rightarrow \langle [\,],\alpha\rangle}\ \begin{array}{l}\text{CT-Var}\\ {\scriptstyle (x\,:\,\alpha \in \Gamma)}\end{array} \qquad \frac{}{CT(c,\Gamma) \Rightarrow \langle [\,],\alpha\rangle}\ \begin{array}{l}\text{CT-Const}\\ {\scriptstyle (c\,:\,\alpha \in \Sigma)}\end{array}$$

$$\frac{FC(\gamma,\Gamma_c,\Gamma) \Rightarrow \xi}{CT(c\gamma,\Gamma) \Rightarrow \langle \xi,\alpha_c\gamma\rangle}\ \begin{array}{l}\text{CT-Subst}\\ {\scriptstyle (\Gamma_c \vdash c\,:\,\alpha_c \in \Sigma)}\end{array}$$

$$\frac{CT(f,\Gamma) \Rightarrow \langle \xi_1, \alpha{\rightarrow}\beta\rangle \quad GTE(e,\alpha,\Gamma) \Rightarrow \xi_2}{CT(fe,\Gamma) \Rightarrow \langle \xi_1 \mathbin{@} \xi_2, \beta e\rangle}\ \text{CT-App}$$

Finally, the rules for checking that a substitution fits a context, are defined over the structure of the substitution:

$$\frac{}{FC(\{\},[\,],\Gamma) \Rightarrow [\,]}\ \text{FC-Empty}$$

$$\frac{FC(\gamma,\Delta,\Gamma) \Rightarrow \xi_1 \quad GTE(a,\alpha\gamma,\Gamma) \Rightarrow \xi_2}{FC(\{\gamma, x{:=}a\},[\Delta, x:\alpha],\Gamma) \Rightarrow \xi_1 \mathbin{@} \xi_2}\ \text{FC-Ext}$$

The $GTE$-algorithm will fail if the term is not $\beta S$-normal, if the term contains free variables which are not declared in the input context or if there is any arity mismatch between the term and the type.

This completes the algorithm which generates type equations from a type checking problem.

## 5.3 Type Conversion

The list of type equations produced by $GTE$ is the input to the *TSimple*-algorithm, which in turn uses *TConv*, i.e. the type conversion algorithm. *TConv* simplifies each type equation to a list of term equations, so the result of *TSimple* is a list of term equations.

**The TSimple algorithm**

The order of the input equations is important (and is therefore represented as lists rather than sets). The reason is found in the CT-App rule. The first premiss states that $f : \alpha{\rightarrow}\beta$ and the second checks if the argument is of type $\alpha$. Then the conclusion states that $fa$ is of type $\beta a$ assuming the equations in $\xi_1$ and $\xi_2$ hold. If this is not the case, for example if the argument is ill-typed ($\xi_2$ is inconsistent), then $\beta a$ is not even a type. Therefore we will define what we mean by a proper list of equations. We will say that a list of equations ***holds*** if all equations in the list hold. In the substitution calculus, all judgements are decidable, so "hold" will in this calculus mean that the equations are derivable.

**Definition 5.7** A *well-formed list of complete type equations* is defined inductively by

$$\frac{}{[\,]\ \textit{well-formed}}$$

$$\frac{\xi\ \textit{well-formed} \qquad \xi \text{ holds} \supset \Gamma \vdash \alpha : \mathit{Type} \wedge \Gamma \vdash \beta : \mathit{Type}}{[\xi, \langle \alpha, \beta, \Gamma \rangle]\ \textit{well-formed}}$$

The well-formedness condition says that the types are valid relative the previous equations in the list. It is easy to see that $GTE$ produces a well-formed list of type equations (proposition 4). However, if s list of equations do not hold, there is no guarantee that the next equation relate two types. For instance, if we have a function

$f : (X : Set; x : El(X))Set$

and we apply the $GTE$ algorithm to the type checking problem

$f(Id(N, 0, true), refl(N, 0)) : Set$

where $Id$ is the identity relation as defined in section 2.2. Then the generation of type equations yields the following well-formed list:

$$\begin{array}{rcl} Set &=& Set \\ El(N) &=& El(N) \\ El(N) &=& El(Bool) \\ Set &=& Set \\ El(N) &=& El(N) \\ El(Id(N, 0, 0)) &=& El(Id(N, 0, true)) \\ Set &=& Set \end{array}$$

Here we can see that $Id(N, 0, true)$ is not well-typed, so $El(Id(N, 0, true))$ is not a valid type, but since the equation $El(N) = El(Bool)$ does not hold, the list is well-formed anyhow.

The $TSimple$ algorithm takes a list of type equations $\xi$ as input and produces a list of term equations $\mathcal{C}$ as output.

**TSimple-algorithm**

Precondition: $\xi$ is *well-formed*

$$\frac{}{TSimple\,[\,] \Rightarrow [\,]} \qquad \frac{TSimple\,(\xi) \Rightarrow \mathcal{C}_1 \quad TConv\,(\alpha, \alpha', \Gamma) \Rightarrow \mathcal{C}_2}{TSimple\,[\xi, \langle \alpha, \alpha', \Gamma \rangle] \Rightarrow \mathcal{C}_1 \mathbin{@} \mathcal{C}_2}$$

The order of the output list $\mathcal{C}$ is also important, and we will define what we mean by a well-formed list of term equations

**Definition 5.8** A *well-formed list of term equations* is defined inductively by

$$\frac{}{[\,]\ \textit{well-formed}}$$

$$\frac{\mathcal{C}\ \textit{well-formed} \qquad \mathcal{C}\text{ holds} \supset \Gamma \vdash a : \alpha \wedge \Gamma \vdash b : \alpha}{[\mathcal{C}, \langle a, b, \alpha, \Gamma\rangle]\ \textit{well-formed}}$$

The *TSimple* algorithm has the property that it takes well-formed type equation lists to well-formed term equation lists.

### 5.3.1 The Type Conversion algorithm (TConv)

The type conversion algorithm reduces the problem of whether two types are convertible into a problem of checking the convertibility of a list of term equations. The types may have to be reduced, and a weak reduction strategy is used to reduce the types to outermost constructor form, which is defined as follows:

**Definition 5.9** A type on outermost *constructor form* is defined syntactically by

$$\alpha_{cf} ::= \ Set \mid \alpha \rightarrow \beta \mid El(A)$$

Type conversion is computed in the following way, first the forms of the two types are checked; if they have the same outermost constructor form, then the parts are checked. If they do not have the same form but are on constructor form, type conversion returns a failure, since equal types have the same constructor form (lemma 4.2). The remaining case is when at least one of the types is not on constructor form, then they are reduced to constructor form and the algorithm is called recursively with the reduced types.

The reduction to constructor form is done by applying the $\beta Subst$-reduction rules below until no rule is applicable. The idea is that all beta redexes are performed creating substitutions, which are pushed inside the type constructors. There are no type variables so the type structure can not be effected by a substitution. The result is a type on constructor form.

**One step type reduction rules**

preconditions: $\Gamma \vdash \alpha : \ Type$

| | | | |
|---|---|---|---|
| *SubstSet*: | $Set\gamma \xrightarrow{T\beta S} Set$ | *App*$\beta$: | $([x]\alpha)a \xrightarrow{T\beta S} \alpha\{x{:=}a\}$ |
| *SubstFun*: | $(\alpha\to\beta)\gamma \xrightarrow{T\beta S} (\alpha\gamma\to\beta\gamma)$ | *App*$\beta$*Subst*: | $(([x]\alpha)\gamma)a \xrightarrow{T\beta S} \alpha\{\gamma, x{:=}a\}$ |
| *SubstSubst*: | $(\alpha\delta)\gamma \xrightarrow{T\beta S} \alpha(\delta\gamma)$ | *AppElSubst*: | $(El\gamma)A \xrightarrow{T\beta S} El(A)$ |
| *SubstApp*: | $(\beta a)\gamma \xrightarrow{T\beta S} (\beta\gamma)(a\gamma)$ | *AppSubstSubst*: | $((\beta\delta)\gamma)a \xrightarrow{T\beta S} (\beta(\delta\gamma))a$ |

The type conversion algorithm takes two types and a context as input, and produces a list of term equations as output. The types are equal if and only if the list of equations holds.

**TConv-algorithm**

preconditions: $\Gamma \vdash \alpha : Type$ and $\Gamma \vdash \beta : Type$

$$\frac{}{TConv\,(\alpha, \alpha, \Gamma) \Rightarrow [\,]}\ \text{TConv-Id}$$

$$\frac{}{TConv\,(El(A), El(B), \Gamma) \Rightarrow [\langle A, B, Set, \Gamma\rangle]}\ \text{TConv-El}$$

$$\frac{TConv\,(\alpha, \alpha', \Gamma) \Rightarrow \mathcal{C}_1 \quad TConv\,(\beta z, \beta' z, [\Gamma, z : \alpha]) \Rightarrow \mathcal{C}_2}{TConv\,(\alpha\to\beta, \alpha'\to\beta', \Gamma) \Rightarrow \mathcal{C}_1 \,@\, \mathcal{C}_2}\ \begin{array}{l}\text{TConv-Fun}\\ (z \notin Dom(\Gamma))\end{array}$$

$$\frac{\alpha_1 \xrightarrow{T\beta S}_* \alpha_1' \quad \alpha_2 \xrightarrow{T\beta S}_* \alpha_2' \quad TConv\,(\alpha_1', \alpha_2', \Gamma) \Rightarrow \mathcal{C}}{TConv\,(\alpha_1, \alpha_2, \Gamma) \Rightarrow \mathcal{C}}\ \begin{array}{l}\text{TConv-}\beta\text{Subst}\\ (\alpha_1\ or\ \alpha_2\ not\ CF)\end{array}$$

where $\xrightarrow{T\beta S}_*$ is the transitive closure of $\xrightarrow{T\beta S}$. Note that we check the equality of two type families by applying them to a fresh variable (justified by the extensionality rule). That way we need not rename bound variables of the type families, since if the family is an abstraction, then the application of the variable creates a $\beta$-redex which is reduced by the *App*$\beta$-rule to a substitution. The substitution is pushed inside type constructors, and the "problem" is postponed to the conversion on terms.

## 5.4 Conversion

The *Simple* algorithm simplifies a list of term equations by checking each equation by the *Conv*-algorithm. If all equations hold, that is *Conv* returns the empty list for every equation, then *Simple* returns the empty list. If one of the equations does not hold, that is *Conv* returns a failure, then *Simple* returns a failure as well. Hence, an empty list as result means that all equations hold. Since *Conv* requires the terms to be well-typed, we must require the list to be well-formed and check the equations in order, to achieve termination, since the conversion algorithm may reduce the terms and they must therefore be well-typed.

**Simple-algorithm**

preconditions: $\mathcal{C}$ is *well-formed*

$$\frac{}{\mathit{Simple}\,[\,] \Rightarrow [\,]} \qquad \frac{\mathit{Simple}\,(\mathcal{C}) \Rightarrow \mathcal{C}_1 \quad \mathit{Conv}\,(a, b, \alpha, \Gamma) \Rightarrow \mathcal{C}_2}{\mathit{Simple}\,[\mathcal{C}, \langle a, b, \alpha, \Gamma\rangle] \Rightarrow \mathcal{C}_1 \mathbin{@} \mathcal{C}_2}$$

The conversion algorithm follows very much the same idea as the type conversion, but the reduction is of course more complicated. The algorithm proceeds as follows;

1. First it checks if the terms are syntactically equal. If that is the case we are done.

2. Otherwise, if the terms have a function type, then both terms are applied to a fresh variable, and conversion is called recursively.

3. Finally, it computes both terms to *head normal form* (corresponding to constructor form on types), and the simpler head conversion is invoked.

(2) guarantees that the terms are *ground* (i.e have a ground type), and (3) implies that the term is of the form $b(a_1, \ldots, a_n)$ where $b$ is either a variable or a constructor. (Note that this is a stronger restriction on $b$ than $\beta S$-normal, where $b$ may be any constant, and since constants may have reduction rules associated with them - see below - the term may be further reduced). The only possibility for two ground terms on head normal form to be equal, is that the heads are identical and the arguments are pairwise convertible. Therefore, this is exactly what the head conversion algorithm checks. There are two advantages of performing the rule (2), first $\eta$-conversion need not be checked separately ($[x](fx)$ will be equal to $f$ when applied to a variable) and second, we know that a function defined by pattern matching (see below) will be applied to all its arguments and this simplifies the matching.

### 5.4.1 The Conv-algorithm (Conv)

The conversion algorithm takes as input two terms $a$ and $b$, a type $\alpha$ and a context $\Gamma$. It returns an empty list if $a$ and $b$ are equal terms and a failure if they are not. We need to know that the terms are well-typed, since the conversion algorithm reduces terms which are not on constructor form.

**Conv-algorithm**

preconditions: $\Gamma \vdash a : \alpha$ and $\Gamma \vdash b : \alpha$

$$\frac{}{Conv\,(a, a, \alpha, \Gamma) \Rightarrow [\,]}\ \text{Conv-Id}$$

$$\frac{Conv\,(az, bz, \beta z, [\Gamma, z : \alpha]) \Rightarrow \mathcal{C}}{Conv\,(a, b, \alpha \to \beta, \Gamma) \Rightarrow \mathcal{C}}\ \begin{array}{l}\text{Conv-fun}\\ (z \notin Dom(\Gamma))\end{array}$$

$$\frac{a \xrightarrow{hnf} a' \quad b \xrightarrow{hnf} b' \quad HConv\,(a', b', \Gamma) \Rightarrow \langle \mathcal{C}, \alpha' \rangle}{Conv\,(a, b, \alpha, \Gamma) \Rightarrow \mathcal{C}}\ \text{Conv-hnf}$$

where we know by lemma 2.1 that $\Gamma \vdash \alpha = \alpha' \,:\, Type$.

**HConv-algorithm**

$$\frac{}{HConv\,(b, b, \Gamma) \Rightarrow \langle [\,], \alpha \rangle}\ \begin{array}{l}\text{HConv-head}\\ (b \in \{x, c\}, b : \alpha)\end{array}$$

$$\frac{HConv\,(f, g, \Gamma) \Rightarrow \langle \mathcal{C}_1, \alpha \to \beta \rangle \quad Conv\,(a, b, \alpha, \Gamma) \Rightarrow \mathcal{C}_2}{HConv\,(fa, gb, \Gamma) \Rightarrow \langle \mathcal{C}_1 \,@\, \mathcal{C}_2, \beta a \rangle}\ \text{HConv-app}$$

## 5.5 Term reduction

We will distinguish between three kinds of constants - constructors ($c_c$), explicitly defined constants ($c_e$) and implicitly defined constants ($c_i$). Constructors are primitive and are not subject to reduction, whereas the defined constants have reduction rules associated with them. Implicitly defined constants are defined by pattern matching ($\xrightarrow{M}$) which reduces a term matching a pattern to its corresponding definition. Explicitly defined constants are simple abbreviations and the unfolding reduction ($\xrightarrow{\delta}$) expands a constant to its definition. The third reduction is the $\beta$-Subst reduction $\xrightarrow{\beta S}$ which performs $\beta$-reduction and handles explicit substitutions. Finally, we have the head normal form reduction $\xrightarrow{hnf}$ which determines the reduction strategy by administrating calls of the other reductions. Here, we are interested in the normal order strategy, which always reduces the head redex first.

**Definition 5.10** The *head* of a term is defined by

$$\begin{cases} head(fa) = head(f) \\ head(a) = a \end{cases}$$

The reason for choosing the head reduction strategy is that we are mainly interested in checking conversion between terms, and if the terms are not syntactically equal or on normal form, with this strategy we may detect that the terms are not equal at an earlier stage than when they are fully reduced. This is because constructors are assumed to be one-to-one, so two terms starting with different constructors will never become equal during further reductions. Free variables are also unique and not subject to reduction, and will act as constructors in this respect.

**Definition 5.11** A ground term is on *head normal form (hnf)* if the head of the term is a variable or constructor.

### 5.5.1 $\delta$- and $\beta$Subst-reductions

The $\delta$-reduction simply replaces the head constant with its corresponding definition:

**$\delta$-reduction**

$$\left.\begin{array}{l} c_e \xrightarrow{\delta} e \\ c_e\gamma \xrightarrow{\delta} e\gamma \end{array}\right\} \quad c_e = e \in \Sigma \qquad\qquad \frac{f \xrightarrow{\delta} f'}{(fa) \xrightarrow{\delta} (f'a)}$$

The $\beta$-Subst reduction removes $\beta$-redexes and substitution from the head of the term. Similarly as for types, $\beta$-redexes are reduced to substitutions which are successively pushed inside compound terms, when needed. As mentioned in chapter 4, substitutions could be seen as local environments. They are manipulated by composition and updating, and substitutions are not performed on terms, rather values of variables are looked up in the appropriate substitution (see the *SubstVar*-rules). The scope of variables is taken care of in the manipulation of the substitutions. A substitution is never pushed inside an abstraction, instead the substitution becomes a closure of the abstraction. The substitution is updated when the abstraction is applied to an argument. The $\beta$-Subst reduction computes a term to $\beta$-Subst head normal form, which is defined by:

**Definition 5.12** A term is on *$\beta$-Subst head normal form ($\beta S$-hnf)* if it is on the form

$$e_{\beta Sh} ::= x \mid c_c \mid c_i \mid c_e \mid c_e\gamma \mid (e_{\beta Sh} e), \quad \text{for ground terms}$$
$$f_{\beta Sh} ::= e_{\beta Sh} \mid [x]e \mid ([x]e)\gamma, \quad \text{for functional terms}$$

### $\beta$-Subst reduction

preconditions: $\Gamma \vdash a : \alpha$

| | | | |
|---|---|---|---|
| $SubstVar_1$: | $x\{\} \xrightarrow{\beta S} x$ | $SubstConstr$: | $c_c\gamma \xrightarrow{\beta S} c_c$ |
| $SubstVar_2$: | $x\{\gamma, x{:=}a\} \xrightarrow{\beta S} a$ | $SubstImpl$: | $c_i\gamma \xrightarrow{\beta S} c_i$ |
| $SubstVar_3$: | $x\{\gamma, y{:=}a\} \xrightarrow{\beta S} x\gamma,\ x \neq y$ | $SubstApp$: | $(fa)\gamma \xrightarrow{\beta S} (f\gamma)(a\gamma)$ |
| $SubstVar_4$: | $x((\gamma\gamma')\delta) \xrightarrow{\beta S} x(\gamma(\gamma'\delta))$ | $SubstSubst$: | $(a\delta)\gamma \xrightarrow{\beta S} a(\delta\gamma)$ |
| $SubstVar_5$: | $x(\{\delta, x{:=}a\}\gamma) \xrightarrow{\beta S} a\gamma$ | $App\beta$: | $([x]b)a \xrightarrow{\beta S} b\{x{:=}a\}$ |
| $SubstVar_6$: | $x(\{\delta, y{:=}a\}\gamma) \xrightarrow{\beta S} x(\delta\gamma)$ | $App\beta Subst$: | $(([x]b)\gamma)a \xrightarrow{\beta S} b\{\gamma, x{:=}a\}$ |
| $SubstVar_7$: | $x(\{\}\gamma) \xrightarrow{\beta S} x\gamma$ | $AppApp$: | $\dfrac{f \xrightarrow{\beta S} f'}{(fa) \xrightarrow{\beta S} (f'a)}$ $f$ not $\beta S$-hnf |

There are no rule in the $\beta - Subst$ reduction if the head of the term is an explicit constant since the $\delta$-reduction, that is the expansion to the constant definition, must be performed before the substitution can be applied.

Note that bound variables need *never* be renamed. This can be illustrated by the following example; we want to reduce the term

$$(([y]([x](yx))x)y)$$

where usually the bound variable $x$ must be renamed before $x$ is substituted for $y$ in $[x](yx)$, to avoid the capture of $x$. The only applicable rule is *AppApp*, so we start by reducing

$([y]([x](yx))x)$

which reduces to

$([x](yx))\{y{:=}x\}$, by $App\beta$

and now this term is on functional $\beta$S-hnf, so we have the term

$(([x](yx))\{y{:=}x\})y$, which reduces to
$(yx)\{y{:=}x, x{:=}y\}$ by the $App\beta Subst$ rule.

Now the substitution is pushed inside by $SubstApp$, and finally the assignments of the variables are looked up in the substitution by the $SubstVar$ rules, resulting in

$(xy)$.

### 5.5.2 Head normal form reduction

The hnf-reduction and pattern match reduction are mutually dependent, since the matching is needed to reduce a term whose head is an implicit constant, and arguments of this term may have to be reduced to head normal-form (hnf) to determine which pattern matches. We will start by describing the hnf-reduction.

The hnf-reduction algorithm reduces a term of ground type to head normal form, or possibly to *irreducible form*. The reason that a term can be irreducible but not on head normal form is that we handle open terms, i.e. terms containing free variables, and allow functions to be defined by pattern matching. In ordinary functional languages, there are no open terms during reduction, so the matching will always succeed if the function is defined by pattern matching and the patterns are exhaustive. Here, we can have the situation that a variable (or another irreducible term) is blocking the matching, even thought the patterns are exhaustive. For example, if addition is defined by

$$\begin{cases} add(0, y) = y \\ add(S(x), y) = S(add(x, y)) \end{cases}$$

then the term $add(x, y)$ is irreducible since $x$ is a variable and $add$ is defined by pattern matching over the first argument. The definition is as follows

**Definition 5.13** A ground term is called *irreducible* if it is of the form $c_i(a_1, \ldots, a_n)$, where some $a_j$ is a variable or an irreducible term, and $c_i$ is defined by pattern matching over argument $j$.

A hnf-term and an irreducible term have the property that no head reduction can alter the outermost form of the term, so the term is in a sense stable.

**Definition 5.14** We will call a (ground) term *rigid* if it is either on head normal form, or irreducible.

It will become clear why we distinguish between irreducible and hnf-terms, and why we introduce this notion of rigid terms, when we extend our language to include incomplete terms. Because when we have incomplete terms, a term can also be *flexible* which means that the outermost form of the term will change depending on different instantiations.

However, the distinction between irreducible terms and terms on head normal form is also good for efficiency reasons. In the conversion, for instance, we need never compare an irreducible term with a term on head normal form, since the head of the former is always an implicit constant whereas the latter is a constructor or a variable. We will also see that the information is valuable in the pattern matching algorithm presented below.

The hnf reduction algorithm proceeds by examining the head of the term and applies the appropriate rule until the head cannot be reduced any further. The result will be a rigid term, together with a label ($L \in \{R_{hnf}, R_{irr}\}$) indicating which kind of term the result is.

**Head normal form reduction**

preconditions: $\Gamma \vdash a : \alpha$, where $\alpha$ is a ground type

$$head(a) \in \{x, c_c\} : \quad \frac{}{a \xrightarrow{hnf} R_{hnf}(a)}\ \text{Hnf}$$

$$head(a) \in \{c_e, c_e\gamma\} : \quad \frac{a \xrightarrow{\delta} a' \quad a' \xrightarrow{hnf} L(a'')}{a \xrightarrow{hnf} L(a'')}\ \text{Unfold}$$

$$head(a) \in \{[x]b, b'\gamma\}, b' \neq c_e : \quad \frac{a \xrightarrow{\beta S} a' \quad a' \xrightarrow{hnf} L(a'')}{a \xrightarrow{hnf} L(a'')}\ \beta\text{Subst}$$

$$head(a) \in \{c_i\} : \quad \frac{a \xrightarrow{M} Ok(a') \quad a' \xrightarrow{hnf} L(a'')}{a \xrightarrow{hnf} L(a'')}\ \text{Match}$$

$$head(a) \in \{c_i\} : \quad \frac{a \xrightarrow{M} R_{irr}(a)}{a \xrightarrow{hnf} R_{irr}(a)}\ \text{Irred}$$

### 5.5.3 Pattern matching reduction

The Match reduction takes as input a ground term, whose head is an implicit constant $c_i$, and a set of pattern rules associated with this constant. Since the term is ground, we know that the function is applied to all its arguments. We have the following definition of a pattern and a pattern rule:

**Definition 5.15** A pattern is of the form

$$p ::= x \mid c_c(p_1, \ldots, p_n),$$

where $n$ is the arity of the constructor $c_c$. The _ sign as we have seen in the examples of section 2.2, is used to denote that an argument is completely determined for any type correct term matching the the left-hand side of a pattern. It can be seen as a special pattern variable, and could be replaced by a fresh pattern variable wherever it occurs. Since it is used for efficiency reasons, we will consider only patterns with ordinary pattern variables.

A pattern rule consists of two parts, the left-hand side (which is often called a pattern as well) and the right-hand side which is the definition of the pattern.

$$P_{rule} ::= c_i(p_1, \ldots, p_n) = e$$

where $n$ is the arity of $c_i$, $FV(e) \subseteq FV(p_1) \cup \cdots \cup FV(p_n)$, and all variables in the patterns are distinct.

The last restriction is simply to prohibit that new pattern variables are introduced in the definition part of a pattern rule.

The pattern rule is matched against the input term, and if the pattern matches it will produce a substitution of the pattern variables, which is applied to the right-hand side of the rule. We do not want to impose any order of the pattern rules, which implies that the patterns in the set of rules must be *non overlapping*, i.e. at most one pattern can possibly match, to guarantee a deterministic behaviour of the matching algorithm. We will also require the patterns to be *exhaustive*, i.e. at least one pattern will always match, of semantical reasons, see [Coq92]. Non-exhaustive patterns would cause the set of irreducible terms to increase.

The simplest algorithm would check each pattern rule until a rule matches with the term, and if there is such rule (we know there is at most one), its instantiated right-hand side would be the result of the reduction. But it is easy to see that we can do better than that, since if the match of the first rule fails because there is a variable in a position where the pattern's head is a constructor, we know that all other patterns will fail because of the same reason (due to the non-overlapping condition). Therefore, the matching of a rule will result in a substitution when the match succeeded, a failure when the rule did not match due to a different constructor (denoted $Fail$), and otherwise an indication that the term is irreducible ($Irred$). In the first and the last case we need not check the rest of the pattern rules.

The matching of a term $e$ relative to a set of pattern rules either succeeds and the term is reduced (denoted by *Reduced*) or the term is irreducible:

**$\xrightarrow{M}$-reduction**

$$\frac{\langle prules, e\rangle \overset{M}{\Longrightarrow} Reduced(d\gamma)}{e \xrightarrow{M} Reduced(d\gamma)} \qquad \frac{\langle prules, e\rangle \overset{M}{\Longrightarrow} Irred}{e \xrightarrow{M} R_{irr}(e)}$$

In the above rules, $prules$ is the set of pattern rules associated with the implicit constant which is the head of $e$. The rule matching algorithm calls the pattern matching, and if the matching succeeds it returns the right-hand side applied to the substitution of the pattern variables, if it fails the rest of the pattern rules are checked and otherwise the term is irreducible. With our restrictions of an exhaustive set of pattern rules, the rule matching cannot fail, that is we never reach $\langle \emptyset, e\rangle$.

**$\overset{M}{\Longrightarrow}$-rule matching**

$$\frac{\langle p, e\rangle \Rightarrow_m \gamma}{\langle (p{=}d) \cup prules, e\rangle \overset{M}{\Longrightarrow} Reduced(d\gamma)} \qquad \frac{\langle p, e\rangle \Rightarrow_m Irred}{\langle (p{=}d) \cup prules, e\rangle \overset{M}{\Longrightarrow} Irred}$$

$$\frac{\langle p, e\rangle \Rightarrow_m Fail \quad \langle prules, e\rangle \overset{M}{\Longrightarrow} X}{\langle (p{=}d) \cup prules, e\rangle \overset{M}{\Longrightarrow} X}$$

where $X$ can be either $Irred$ or $Reduced(e')$. Pattern matching can either succeed with a substitution of the pattern variables, indicate that the term is irreducible, or fail when two distinct constructors are found.

**$\Rightarrow_m$-pattern matching**

In the rules below, $x$ will denote an arbitrary variable, so the side-condition $p' \neq x$ simply means that the pattern $p'$ is not a variable.

Successful rules:

$$\frac{}{\langle c, c\rangle \Rightarrow_m \{\}} \qquad \frac{\langle p, e\rangle \Rightarrow_m \gamma}{\langle (px), (ea)\rangle \Rightarrow_m \{\gamma, x{:=}a\}}$$

$$\frac{\langle p, e\rangle \Rightarrow_m \gamma \quad a \overset{hnf}{\longrightarrow} R_{hnf}(a') \quad \langle p', a'\rangle \Rightarrow_m \delta}{\langle (pp'), (ea)\rangle \Rightarrow_m \gamma \cup \delta}\ p' \neq x$$

Since the pattern variables are all distinct, we can simply take the union of the assigned variables in the substitutions.

The term is irreducible:

$$\frac{\langle p, e\rangle \Rightarrow_m Irred}{\langle (pp'), (ea)\rangle \Rightarrow_m Irred}\ p' \neq x \qquad \frac{\langle p, e\rangle \Rightarrow_m \gamma \quad a \overset{hnf}{\longrightarrow} R_{irr}(a')}{\langle (pp'), (ea)\rangle \Rightarrow_m Irred}\ p' \neq x$$

$$\frac{\langle p, e\rangle \Rightarrow_m \gamma \quad a \overset{hnf}{\longrightarrow} R_{hnf}(a')}{\langle (pp'), (ea)\rangle \Rightarrow_m Irred}\ p' \neq x,\, head(a) = x$$

Here we can see that in the second rule we did not have to check if the reduced term $a'$ would match the pattern $p'$ or not, since the label $R_{irr}$ already told us that the term is an irreducible term. This means, as we have seen, that the head of the term is an implicit constant and since patterns only contain constructors and variables, this term can not possibly match the pattern.

The matching fails when two constructors are distinct:

$$\frac{}{\langle c, c' \rangle \Rightarrow_m Fail}\ c \neq c' \qquad\qquad \frac{\langle p, e \rangle \Rightarrow_m Fail}{\langle pp', ea \rangle \Rightarrow_m Fail}$$

This completes the description of the type checking algorithm.

## 5.6 Correctness of judgement checking

The main result of this chapter is the correctness proof of the type checking algorithm for complete terms. It is divided in two parts, the soundness proof and the completeness proof. In this section we will give an overview of the main propositions and theorems, the complete proofs can be found in the appendix B and C.

### 5.6.1 Soundness

The proof follows the structure of the algorithms, and hence we get the main proposition that the type checking algorithm $TC$ is sound, by simply composing the results for the involved algorithms. Thus we get the following main theorem

**Theorem 1 (TC-soundness)**
*Let $\alpha$ be a type in context $\Gamma$, and let $e$ be a $\beta S$-normal term which is $\Gamma$-distinct. If $TC(e, \alpha, \Gamma) \Rightarrow [\,]$, then $\Gamma \vdash e : \alpha$.*

***Proof:*** Follows directly from GTE-, TSimple- and Simple-soundness. □

The soundness of type formation follows easily from the soundness of type checking:

**Proposition 3 (TF-soundness)**
*Let $\Gamma$ be a valid context and $\alpha$ $\Gamma$-distinct and $\beta S$-normal.*

*If $TF(\Gamma, \alpha) \Rightarrow \xi$ and $\xi$ holds, then $\Gamma \vdash \alpha : Type$.*

***Proof:*** Induction on the structure of $\alpha$. Since $\alpha$ is assumed to be $\beta S$-normal, then $\alpha$ is of the form *Set*, *El*(A) or $\alpha \rightarrow \beta$. □

The soundness of the generation of type equations says that if all the equations which are generated by $GTE$ hold, then the checked term is type correct. It is defined simultaneously as the corresponding results for the mutually defined algorithms $FC$ and $CT$. The proof is a straightforward but somewhat tedious induction on the structure of the term.

**Proposition 4 (GTE-soundness)**
*Let $\Gamma, \Delta$ be valid contexts and $\Gamma \vdash \alpha$ : Type. Let $q$ be a $\beta S$-normal term (and not an abstraction in the CT-case) and let $q$ be $\Gamma$-distinct. Then we have the following*

$$\begin{cases} GTE(q, \alpha, \Gamma) \Rightarrow \xi & \wedge \quad \xi \ holds \quad \supset \quad \Gamma \vdash q : \alpha \\ FC(q, \Delta, \Gamma) \Rightarrow \xi & \wedge \quad \xi \ holds \quad \supset \quad \Gamma \vdash q : \Delta \\ CT(q, \Gamma) \Rightarrow \langle \xi, \alpha \rangle & \wedge \quad \xi \ holds \quad \supset \quad \Gamma \vdash q : \alpha \end{cases}$$

*where $\xi$ is a well-formed list of type equations.*

***Proof:*** Induction on the structure of $q$. □

The next proposition shows that well-formedness is preserved by $TSimple$, and that if all equations hold in the resulting list of term equations, then the input list of type equations also hold. The soundness of type conversion is used to prove this proposition. We need the well-formedness requirement to fulfill the prerequisite of type conversion.

**Proposition 5 (TSimple-soundness)**
*Let $\xi$ be a well-formed list of type equations.*

*If TSimple$(\xi) \Rightarrow \mathcal{C}$, then $\mathcal{C}$ is well-formed*

*and*

*if $\mathcal{C}$ holds, then $\forall \langle \alpha, \alpha', \Gamma \rangle \in \xi \ . \ \Gamma \vdash \alpha = \alpha'$ : Type.*

***Proof:*** Induction on the length of the list. (Uses TConv-sound). □

**Proposition 6 (TConv-soundness)**
*Let $\alpha$ and $\alpha'$ be valid types in context $\Gamma$.*

*If TConv$(\alpha,\alpha',\Gamma) \Rightarrow \mathcal{C}$ and $\mathcal{C}$ holds, then $\Gamma \vdash \alpha = \alpha'$ : Type.*

***Proof:*** Induction on the length of the derivation $TConv(\alpha,\alpha',\Gamma) \Rightarrow \mathcal{C}$. □

Analogously, the soundness of the simplification of term equations says that if the simplification results in an empty list, then all the equations are derivable in the calculus. The proof relies on the next proposition which states that if the term conversion algorithm succeeds, then the equation is derivable.

**Proposition 7 (Simple-soundness)**
*Let $\mathcal{C}$ be a well-formed list of term equations.*

$$\textit{If } \mathit{Simple}(\mathcal{C}) \Rightarrow [\,],\ \textit{then } \forall\langle a,b,\alpha,\Gamma\rangle \in \mathcal{C}\ .\ \Gamma \vdash a = b : \alpha.$$

***Proof:*** Induction on the length of the list $\mathcal{C}$. (Uses Conv-sound). □

The conversion and the simpler head conversion algorithm return an empty list only when the term are convertible according to the rules of the calculus. The main lemma of this proposition is that the type of a term is preserved during reduction.

**Proposition 8 (Conv-soundness)**
*If $a$ and $b$ are terms of type $\alpha$ in context $\Gamma$, then*

$$\left\{\begin{array}{lll} \mathit{Conv}(a,b,\alpha,\Gamma) \Rightarrow [\,] & \textit{implies} & \Gamma \vdash a = b : \alpha \\ \mathit{HConv}(a,b,\Gamma) \Rightarrow \langle [\,], \alpha\rangle & \textit{implies} & \Gamma \vdash a = b : \alpha \end{array}\right.$$

***Proof:*** Induction on the length of the derivations of $\mathit{Conv}(a,b,\alpha,\Gamma) \Rightarrow [\,]$ and $\mathit{HConv}(a,b,\Gamma) \Rightarrow \langle [\,], \alpha\rangle$, respectively. Uses that $\xrightarrow{hnf}$ preserves typing.

□

**Lemma 8.1** *If $\Gamma \vdash a : \alpha$ and $a \xrightarrow{hnf} L(a')$, then $\Gamma \vdash a = a' : \alpha$.*

***Proof:*** Case analysis on $a \xrightarrow{hnf} L(a')$. □

### 5.6.2 Completeness

The completeness of the type checking algorithm relies on the meta theory assumption that the reduction to head normal form is normalising. The main theorem is of course the completeness of the type checking algorithm:

**Theorem 2 (TC-completeness)**
*Let $\alpha$ be a type in context $\Gamma$, and let $e$ be a $\beta S$-normal term which is $\Gamma$-distinct.*

*Assuming normalisation of the head-normal reduction, we have that*

*If* $\Gamma \vdash e : \alpha$*, then* $TC(e, \alpha, \Gamma) \Rightarrow [\,]$.

***Proof:*** Follows as a special case of GTE-, TSimple- and Simple-complete. □

We can also state the property mentioned in the introduction of chapter 5:

**Corollary 9**
*Let* $\alpha$ *be a type in context* $\Gamma$*, and let* $e$ *be a* $\beta S$*-normal term which is* $\Gamma$*-distinct. If* $GE(e, \alpha, \Gamma) \Rightarrow \mathcal{C}$*, then*

$\Gamma \vdash e : \alpha \Leftrightarrow \mathcal{C}$ *holds.*

***Proof:*** Follows from the soundness and completeness of GTE and TSimple. □

We had to generalise the completeness statement to involve all extensions of the context, to include derivations which use the thinning rule, since there is no correspondence to the thinning rule in the algorithm.

The completeness of type formation follows from completeness of type checking:

**Proposition 10 (TF-complete)**
*Let* $\Gamma \vdash \alpha : Type$ *be a derivation where* $\alpha$ *is* $\Gamma$*-distinct and* $\beta S$*-normal. Assume normalisation of the head-normal reduction. Then* $\forall \Gamma^* \geq \Gamma$ *such that* $\alpha$ *is* $\Gamma^*$*-distinct, we have*

$TF(\alpha, \Gamma^*) \Rightarrow [\,]$.

***Proof:*** A judgement of the form $\alpha \vdash \Gamma : Type$ can only be built up by structural rules from forming types and by the thinning rule. The thinning rule is taken care of by the induction hypothesis since we show the property $\forall \Gamma^* \geq \Gamma$. Since $\alpha$ is restricted to be $\beta S$-normal, we have only to consider three cases, i.e. when $\alpha$ is *Set*, *El*$(A)$ or a function $\alpha \rightarrow \beta$. □

Analogously, we have to consider all extensions of the context in the next proposition, which states the completeness of the $GTE$ algorithm. Moreover, we also need to consider all types convertible to the type $\alpha$ in the derivation

$\Gamma \vdash a : \alpha$,

since the type conversion rule is not used explicitly in the type checking algorithm. Hence, to be able to use the induction hypothesis in a derivation step which has applied the thinning rule, we had to strengthen the induction hypothesis. With this formulation, we need only consider the structural rules since the thinning rule and the type conversion rule become trivial, and for these rules there are corresponding rules in the $GTE$ algorithm.

**Proposition 11 (GTE-complete)**
*Let $\Gamma \vdash a : \alpha$, $\Gamma \vdash \gamma : \Psi$ and $\Gamma \vdash f : \alpha$ be derivations where*

> *$a, \gamma$ and $f$ are $\Gamma$-distinct and $\beta S$-normal,*
> *$f$ is not an abstraction, and*
> *$\gamma$ is $\Psi$-normal.*

*Let $\Gamma^*$ denote an extension of $\Gamma$, which respects the restriction of being distinct relative to $a$, $\gamma$ and $f$, respectively. Then we have the following properties of GTE, FC and CT:*

$$\begin{array}{lcllcl}
\Gamma \vdash a : \alpha,\ \forall \Gamma^* \geq \Gamma, & & & & & \\
\quad \forall \alpha^*. \Gamma^* \vdash \alpha = \alpha^* : \mathit{Type} & \supset & GTE(a, \alpha^*, \Gamma^*) \Rightarrow \xi & \wedge & \xi \text{ holds} \\
\Gamma \vdash \gamma : \Psi,\ \forall \Gamma^* \geq \Gamma & \supset & FC(\gamma, \Psi, \Gamma^*) \Rightarrow \xi & \wedge & \xi \text{ holds} \\
\Gamma \vdash f : \alpha,\ \forall \Gamma^* \geq \Gamma & \supset & CT(f, \Gamma^*) \Rightarrow \langle \xi, \alpha_0 \rangle & \wedge & \xi \text{ holds}
\end{array}$$

*where $\Gamma^* \vdash \alpha = \alpha_0 : \mathit{Type}$.*

***Proof:*** We have that if $\Gamma \vdash a : \alpha$ then the last step in the derivation is either a structural rule depending on $a$, the type conversion rule or the thinning rule. The induction hypothesis is strengthened in such a way that the last two become trivial. Therefore we need only consider the structural rules. Similarly, for $\Gamma \vdash \gamma : \Psi$ we need only consider the structural rules of $\gamma$, since the thinning rules is accounted for in the induction hypothesis and the target thinning rule is not applicable since $\gamma$ is required to be $\Psi$-normal.

We will prove the properties by induction on the structure of the terms $a, f$ and the substitution $\gamma$.

□

The completeness of $TSimple$ follows immediately from the next proposition, the completeness of type conversion, by a simple induction on the list:

**Proposition 12 (TSimple-complete)** *Let $\xi$ be a well-formed list of type equations. If*

$$\Gamma \vdash \alpha = \alpha' : \mathit{Type} \quad \forall \langle \alpha, \alpha', \Gamma \rangle \in \xi$$

*then*

$$\mathit{TSimple}(\xi) \Rightarrow \mathcal{C} \text{ and } \mathcal{C} \text{ holds.}$$

***Proof:*** Induction on the length of $\xi$. $\square$

As already mentioned, we do not need normalisation to show completeness of type conversion. There is a simple syntactic ordering on the types which ensures termination. Moreover, that the constructor form of the type is preserved during reduction, is showed in a lemma below.

**Proposition 13 (TConv-complete)**
*If $\Gamma \vdash \alpha = \alpha' : Type$, then $TConv(\alpha, \alpha', \Gamma) \Rightarrow \mathcal{C}$ and $\mathcal{C}$ holds.*

***Proof:*** The proof proceeds by case analysis of the types $\alpha$ and $\alpha'$. By lemma 4.1 we have a well-founded order on $\alpha$ and $\alpha'$ such that TConv terminates. There are two main cases, whether the types are on (outermost) constructor form or not. Lemma 4.2 gives that equal types have the same constructor form, so either of the structural rules will apply in this case. Otherwise, we have by lemma 4.3 we have that the types can be reduced to constructor form.

$\square$

**Lemma 13.1** *TConv terminates.*

***Proof:*** We will construct a well-founded order on a pair of types, which is always decreasing in each recursive call of TConv. Hence, the algorithm will terminate.

$\square$

**Lemma 13.2** *If $\Gamma \vdash \alpha = \alpha' : Type$, then the constructor form of $\alpha$ ($CF(\alpha)$) is the same as the $CF(\alpha')$, and if $\Gamma \vdash \beta = \beta' : \alpha \to Type$, then $CF(\beta) = CF(\beta')$.*

***Proof:*** Induction on the length of the derivations $\Gamma \vdash \alpha = \alpha' : Type$ and $\Gamma \vdash \beta = \beta' : \alpha \to Type$. $\square$

**Lemma 13.3** *$\forall \alpha : Type$, $\alpha$ is on constructor form or there exists a reduction $\alpha \xrightarrow{T\beta S}_* \alpha'$ where $\alpha'$ is on constructor form and $CF(\alpha) = CF(\alpha')$.*

***Proof:*** Follows from lemma 4.3.1, 4.1.1 and 4.3.2. $\square$

The simplification of term equations depends on the conversion algorithm, and here we clearly need normalisation since the terms are reduced in the conversion algorithm. However, assuming the meta properties of section 4.3, there is not not much left to prove to get completeness of conversion and thus also of *Simple*.

**Proposition 14 (Simple-complete)** *Let $\mathcal{C}$ be a well-formed list of term equations. Assuming $\xrightarrow{hnf}$ is normalising, we have if*

$$\Gamma \vdash a = b : \alpha \quad \forall \langle a, b, \alpha, \Gamma \rangle \in \mathcal{C}$$

*then*

$$Simple\,(\mathcal{C}) \Rightarrow [\,].$$

***Proof:*** Induction on the length of $\mathcal{C}$. □

**Proposition 15 (Conv-complete)** *Assuming $\xrightarrow{hnf}$ is normalising, we have that if $\Gamma \vdash a = b : \alpha$, then $Conv\,(a, b, \alpha, \Gamma) \Rightarrow [\,]$.*

***Proof:*** Two cases, one for $\alpha$ ground and one for $\alpha$ functional. □

# Chapter 6

# Type checking incomplete terms

In this chapter, we will describe how the type checking algorithm can be extended to handle incomplete terms, by some simple modifications. The point of these terms is that they represent incomplete proof objects, and during the process of proof development we need to represent incomplete proof objects. We will show that with our notion of incomplete terms, incremental refinement of a proof object, is nothing but a valid replacement of a placeholder by a term. This allows for greater flexibility, since no particular order of the refinements of placeholders is imposed. Moreover, we can apply the general delete (local undo) operation explained in section 9.2.

The algorithm has the following properties

- The algorithm is a conservative extension of the type checking algorithm for complete terms, so the same algorithm can be used for both purposes.
- The algorithm is modular, which means for instance that we can alter the unification strategy in the algorithm simply by replacing the unification module.
- The basic tactics "intro" and "refine" used in most proof assistants, can *both* be described in terms of replacing a placeholder by an incomplete term. The correctness of such tactic is established by type checking the incomplete term. This is explained in section 9.2.
- The type checking algorithm can be optimised by *localisation*, which

means that if we instantiate a placeholder in a term, it is enough to type check the instantiation relative to the placeholder's expected type in its local context. Hence, we need not type check the entire instantiated term again after each refinement.

- When an incomplete term is completed by successively refining the term, we know that the term is type correct (proposition 5). This means that we need not type check the completed term, and therefore the risk of constructing a large proof term which turns out not to be type correct, is eliminated.

Recall the figure from the introduction of chapter 5, which is the overview of the algorithms extended to incomplete terms.

$$
e : \alpha\ \Gamma \quad \stackrel{GTE_p}{\Longrightarrow} \quad
\left|\begin{array}{c}
\alpha_1 = \alpha_1'\ \Gamma_1 \\[2em]
\vdots \\
?_m : \alpha_m\ \Gamma_m \\
\vdots \\[2em]
\alpha_n = \alpha_n'\ \Gamma_n
\end{array}\right|
\begin{array}{c}
\stackrel{TConv_p}{\longrightarrow} \\[2em]
\\
\longrightarrow \\
\\[2em]
\stackrel{TConv_p}{\longrightarrow}
\end{array}
\left.\begin{array}{c}
\left[\begin{array}{c} a_1^1 = b_1^1 : \alpha_1^1\ \Gamma_1^1 \\ \vdots \\ a_1^k = b_1^k : \alpha_1^k\ \Gamma_1^k \end{array}\right] \\
\vdots \\
?_m : \alpha_m\ \Gamma_m \\
\vdots \\
\left[\begin{array}{c} a_n^1 = b_n^1 : \alpha_n^1\ \Gamma_n^1 \\ \vdots \\ a_n^p = b_n^p : \alpha_n^p\ \Gamma_n^p \end{array}\right]
\end{array}\right\}
\begin{array}{l} \text{typed} \\ \text{unification} \\ \text{problem} \end{array}
$$

$$
\xrightarrow[TSimple_p]{\quad\quad\quad\quad\quad}
$$

$$
\xrightarrow[GE_p\ =\ TSimple_p\ \circ\ GTE_p]{\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}
$$

The term equations in the typed unification problem we get from $GE_p$, can be simplified just as in the complete case. The difference is that for complete equations we have a decision procedure, but for the unification problem we may not be able to decide if the equations hold or not. Still, we can simplify them as far as possible which is done by the modification of *Simple*. Just as before, type checking will be the composition of the modified algorithms

$$TC_p = Simple_p \circ GE_p.$$

However, we can do better than that, we can try to solve the unification problem by applying a unification algorithm to the result of type checking. The unification problems we are dealing with are higher order, but the algorithm we will present in chapter 7 is a first-order algorithm so it may not solve all unification problems. Thus, the unification will take a unification problem as input and returns a partially solved unification problem. The algorithm implemented in ALF is a *type checking algorithm with unification*, i.e. we have

$$TC_pU = Unify \circ TC_p.$$

We can see that since the algorithms are compositional, we have the modularity we claimed, and we can for instance change the chosen unification algorithm to another one rather easily.

An incomplete term represents an incomplete proof object, where placeholders denote the "holes" yet to be filled in. An incomplete term is defined as follows:

**Definition 6.1** An incomplete term $e$ is defined by the following grammar

$$e ::= \; x \mid c \mid [x]e \mid (ee) \mid e\gamma \mid\, ?_n$$

An *incomplete type* is a type containing incomplete terms and an *incomplete context* is a context containing incomplete types.

We can also see an incomplete term together with a type and a context, as the representative of a partial derivation, due to the close correspondence of terms and derivations. For instance, the typed term

$$add(?_1, s(?_2)) : N$$

represents the following partial derivation:

$$\dfrac{\dfrac{\dfrac{}{add : (N;N)N}\,\text{Const} \qquad ?_1 : N}{add(?_1) : (N)N}\,\text{App} \qquad \dfrac{\dfrac{}{s : (N)N}\,\text{Const} \qquad ?_2 : N}{s(?_2) : N}\,\text{App}}{add(?_1, s(?_2)) : N}\,\text{App}$$

In the above derivation, we can replace the incomplete leaf $?_1 : N$ by any derivation of a natural number, and the same holds for $?_2 : N$. This is exactly reflected in the unification problem we get by type checking the term $add(?_1, s(?_2))$ (of type $N$), which after removal of trivial equations, results in the problem of finding terms of type $N$, to replace $?_1$ and $?_2$.

$$\begin{bmatrix} ?_1 : N \\ ?_2 : N \end{bmatrix}$$

However, this is not always the case, since we may have dependencies among the different sub-derivations, due to dependent function types. Assume we want to prove that

$$n + 0 = 0 + n,$$

for an arbitrary $n$, that is, we want to find a proof object in the type

$$(n : N)Id(N, add(n, 0), add(0, n)).$$

The definitions of $add$, $Id$ and $natrec$ can be found in section 2.3. Suppose we try with the incomplete term $[n]natrec(?_P, ?_d, ?_e, ?_n)$, which correspond to assuming an arbitrary $n$ (abstract with respect to $n$) and to apply the rule of induction over natural numbers (apply the constant $natrec$):

$$[n]natrec(?_P, ?_d, ?_e, ?_n) : (n : N)Id(N, add(n, 0), add(0, n)).$$

Then the last step in the derivation would be applying the abstraction-rule

$$\left.\frac{natrec(?_P, ?_d, ?_e, ?_n) : Id(N, add(n, 0), add(0, n))\ [n : N]}{[n]natrec(?_P, ?_d, ?_e, ?_n) : (n : N)Id(N, add(n, 0), add(0, n))}\ \right\}\mathcal{D}$$

Now, consider the structure of any partial derivation corresponding to the term $natrec(?_P, ?_d, ?_e, ?_n)$, which is a partial derivation with four incomplete leaves, one for each placeholder:

$$\left.\frac{\dfrac{\dfrac{\dfrac{\overline{natrec : \cdots} \quad ?_P : (N)Set}{natrec(?_P) : \cdots} \quad ?_d : ?_P(0)}{natrec(?_P, ?_d) : \cdots} \quad ?_e : (x : N; ?_P(x))?_P(s(x))}{natrec(?_P, ?_d, ?_e) : \cdots} \quad ?_n : N}{natrec(?_P, ?_d, ?_e, ?_n) : ?_P(?_n)}\ \right\}\mathcal{E}$$

Here we can see the dependency between sub-derivations, since once we have a derivation in place of the leaf of $?_P$, the sub-derivations which can possibly replace the leaves $?_d$ and $?_e$ *depend* on the actual proof object replacing $?_P$. The conclusion in the derivation above, depends also on the proof object in the derivation replaced by the leaf $?_n$.

Therefore, the partial derivation corresponding to the entire incomplete term $[n]natrec(?_P, ?_d, ?_e, ?_n)$ must be the derivation of shape as $\mathcal{E}$ above the derivation $\mathcal{D}$, that is

$$\cfrac{\left.\begin{array}{c}\vdots\\ natrec(?_P, ?_d, ?_e, ?_n) : Id(N, add(n,0), add(0,n))\ [n : N]\end{array}\right\}\mathcal{E}}{[n]natrec(?_P, ?_d, ?_e, ?_n) : (n : N)Id(N, add(n,0), add(0,n))}$$

Hence, the conclusion of derivation $\mathcal{E}$ must match the premiss of derivation $\mathcal{D}$. This means that all judgements in $\mathcal{E}$ must be relative the context $[n : N]$ and the additional restriction

$$?_P(?_n) = Id(N, add(n,0), add(0,n)).$$

must be satisfied for the derivations to match.

Again, these restriction are exactly reflected in the unification problem we get from type checking the incomplete term, which in this case yields

$$\left[\begin{array}{l} ?_P : (N)Set \quad [n : N] \\ ?_d : ?_P(0) \quad [n : N] \\ ?_e : (x : N; ?_P(x))?_P(s(x)) \quad [n : N] \\ ?_n : N \quad [n : N] \\ ?_P(?_n) = Id(N, add(n,0), add(0,n)) \quad [n : N] \end{array}\right]$$

If the third argument to *natrec* was an abstraction $[x, h]?_k$ instead of the placeholder $?_e$, then the placeholder $?_k$ would be of type $?_P(s(x))$ and in the context extended with the variables $x$ and $h$:

$$?_k : ?_P(s(x)) \quad [n : N, x : N, h : ?_P(x)]$$

Hence, both the type and the context of a placeholder declaration may contain other placeholders.

Let us continue to refine our proof object, for instance by choosing the induction variable to be $n$, which means we want to replace $?_n$ by the variable $n$. One way would of course be to perform the replacement in the partial proof object, and then type check again. That is, we should type check

$$[n]natrec(?_P, ?_d, ?_e, n) \overset{?}{:} (n : N)Id(N, add(n,0), add(0,n))$$

which will result in the unification problem

$$\left[\begin{array}{l} ?_P : (N)Set \quad [n : N] \\ ?_d : ?_P(0) \quad [n : N] \\ ?_e : (x : N; ?_P(x))?_P(s(x)) \quad [n : N] \\ N = N \quad [n : N] \\ ?_P(n) = Id(N, add(n,0), add(0,n)) \quad [n : N] \end{array}\right]$$

The difference from before is that the fourth argument of *natrec* is now the variable $n$ which is of type $N$, and the expected type of the fourth argument is $N$ as well, thus yielding the equation $N = N\ \ [n : N]$. Moreover, $?_n$ is replaced by $n$ throughout the rest of the list, since the same variable which was assigned to the value $?_n$ in the type checking of the uninstantiated term, is now assigned to the value $n$.

The point of the localisation of type checking is that this is *exactly* what we get if we locally type check the instantiation with the expected type of the placeholder. The instantiation $?_n = n$ is checked by replacing the *placeholder declaration*, that is the placeholder declaration

$$?_n : N\ \ [n : N]$$

is replaced by the result of type checking $n : N$ in the context $[n : N]$, which is the equation

$$N = N\ \ [n : N].$$

Again, $?_n$ is replaced by $n$ throughout the rest of the list.

The main property of localisation is that we get the same result if we type check the term $e$ instantiated by $\sigma$ or if we instantiate the placeholders directly in the list of placeholder declarations and equation $(\mathcal{C})$, as is shown in propositions 16 and 18:

If $TC_p\ ([\,], e, \alpha, \Gamma) \Rightarrow \mathcal{C}$, then $TC_p\ ([\,], e\sigma, \alpha, \Gamma) \Rightarrow \mathcal{C}\sigma$.

Correctness of localisation is an important property which states that any instantiation $\sigma$ which makes $e\sigma$ type correct, will make all equations in $\mathcal{C}\sigma$ hold, and vice versa:

If $TC_p\ ([\,], e, \alpha, \Gamma) \Rightarrow \mathcal{C}$, then $\Gamma \vdash e\sigma : \alpha \Leftrightarrow \mathcal{C}\sigma$ holds.

The result is a special case of theorem 3.

## 6.1 Type checking placeholders

Unfortunately, the above results only hold under the restriction that all placeholders in $e$ are distinct. The reason is that instantiations are terms assigned to placeholders, and terms do not have unique types. Therefore, the situation can occur where the term contains two occurrences of a placeholder with different types, and yet there exists a term which has both types. However, in the unification problem we have sharing and the two occurrences are not possible to share. This is best illustrated by a simple example. Suppose we have a function, which takes two functions as arguments and returns an object in some type, say $Set$

$f : (g : (N)N; h : (Bool)Bool)Set.$

Now, consider the incomplete term

$f(?_1, ?_1) : Set$

which has the solution

$?_1{:=}[x]x.$

However, after type checking this incomplete term, we get a unification problem which has no solution, which is what we will show in this following section.

### Placeholders and their declarations

A placeholder represents possibly an *open term*, and the variables it may depend on occur in its local context. The opened-ness is important, since a placeholder may occur under a $\lambda$-binder, and we may use the bound variable in completing the placeholder. When viewed as an unfinished proof, the variables in the local context correspond to the assumptions we may use in completing the proof.

The local context of a placeholder reflects its *scope*, and thus we can forget about the actual occurrence of the placeholder since the placeholder declaration contains all information of the placeholder we need. Therefore we can type check other occurrences of the same placeholder *relative* its expected type and local context. These type checking equations guarantee that an instantiation of the placeholder which satisfy the equations will be correct in the other occurrences of the placeholder. Hence, we achieve *sharing* among different occurrences of a placeholder which is constrained by its (unique) expected type and context.

A priori, we could choose another formulation of a unification problem, where we would allow several placeholder declarations, i.e. one declaration per occurrence. This is related to completeness of type checking incomplete term with unification, and is discussed in section 8.4.

What does it mean that a placeholder is type correct relative a previous declaration? Assume we have the following declaration

(1) $?_1 : \alpha_1 \quad \Gamma_1$

and we want to type check

(2) $?_1 : \beta \quad \Delta.$

We want to describe the problem of finding a solution of $?_1$ which is of type $\beta$ in context $\Delta$, in terms of the existing declaration of $?_1$. Formulated differently, we want to transform the partial derivation with the incomplete leaf (2) into a partial derivation with another occurrence of the leaf (1). If we manage to do this, any derivation fitting the first leaf will also fit the second leaf and hence these two sub-derivations can be shared. A sufficient requirement is that the expected types and contexts are the same, but we may weaken the requirement on the local contexts. It is enough that the declared local context is a *sub-context* of the other local context, since any term which is a solution in the smaller context is clearly a solution in the extended context, due to the thinning rule. Neither can the term contain any variables not in the smaller context, since then it would not be a solution. Therefore, type checking results in the equations

(1) $\Gamma_1 \preceq \Delta$
(2) $\alpha_1 = \beta \quad \Delta.$

which corresponds to the following transformation of a partial derivation

$$\dfrac{\dfrac{?_1 : \alpha_1 \;\; \Gamma_1}{\ddots} \qquad \dfrac{?_1 : \beta \;\; \Delta}{\cdot^{\cdot^{\cdot}}}}{\Gamma \vdash e : \alpha}$$

into the derivation with shared leaves

$$\dfrac{\dfrac{\boxed{?_1 : \alpha_1 \;\; \Gamma_1}}{\ddots} \qquad \dfrac{\dfrac{\dfrac{\boxed{?_1 : \alpha_1 \;\; \Gamma_1} \quad \dfrac{}{\Gamma_1 \preceq \Delta}\,(1)}{?_1 : \alpha_1 \;\; \Delta} \qquad \dfrac{}{\Delta \vdash \alpha_1 = \beta \,:\, \mathit{Type}}\,(2)}{?_1 : \beta \;\; \Delta}}{\cdot^{\cdot^{\cdot}}}}{\Gamma \vdash e : \alpha}$$

Finally, we can come back to the example above by simply stating that the unification problem we get by type checking the above example,

$$\begin{bmatrix} ?_1 : (N)N \\ (N)N = (\mathit{Bool})\mathit{Bool} \end{bmatrix}$$

clearly has no solutions.

So we have seen that there is a problem if several occurrences of a placeholder have different types, but there are also problems related to different local contexts of the same placeholder. Even if the placeholders have the same type, we must make sure that the "major" placeholder have a local context which is a sub-context of all other local contexts of the same placeholder. If this is not assured, we loose soundness as is shown in the following example. Assume we have a function $f$:

$$f : ((A)A; A)Set$$

and want to type check

$$f([x]?_1, ?_1) \overset{?}{:} Set.$$

If we did not check the scope of the occurrences of $?_1$, we simply get the unification problem

$$[?_n : A \quad [x : A]] \text{ which has a solution } \{?_1 = x\}$$

but of course we have

$$\nvdash f([x]x, x) : Set.$$

Therefore we must choose one occurrence to be the representative for the other occurrences, the problem is to decide which should be the "major" occurrence and give rise to the placeholder declaration. The simplest solution is to take the first occurrence, but then the type checking would immediately fail for the example above, since we get the unification problem

$$\begin{bmatrix} ?_1 : A \quad [x : A] \\ A = A \quad [x : A] \\ [x : A] \leq [\,] \end{bmatrix}$$

which fails due to the third constraint. Thus, we can not have completeness, since for any nonempty set $A$, there is a solution to the type checking problem.

An alternative is to choose the local context of the placeholder to be the smallest context which is a sub-context of all the occurrences. However, this may become rather unclean, since the scope of a placeholder could change during type checking. Hence, we have chosen to restrict the user from using the same placeholder several times, by letting ALF assign distinct names to all placeholders. In the unification algorithm, on the other hand, we need to type check terms containing placeholders which are already declared. However, here we know that the new occurrences refer to the declared placeholders, and therefore ought to have the same type and the same (or possibly an extension of) the local context. Since we will often have to distinguish between the two, we will give a name to such terms and instantiations:

**Definition 6.2** Let U be a unification problem. We will call a term $e$ a *refinement term* (relative U), if all placeholders in $e$ are distinct and different from these declared in U, and a *unification term* if all placeholders are already declared in U.

Accordingly, we will say *refinement instantiations* for instantiations containing only refinement terms and likewise *unification instantiations* if the assigned terms are unification terms.

## 6.2 The modified algorithms

Now when we have introduced the idea of incomplete proof terms representing incomplete proofs, we must also extend the type checking algorithms in chapter 5 to handle incomplete terms. We will add rules for the placeholders in the algorithms, when needed, but the point is that we will use *the same* type checking algorithms for incomplete terms as well as complete terms. For complete terms, the algorithms will behave just as before.

As mentioned, type checking complete terms is a decidable property, but when we extend it to incomplete terms we may receive an answer "maybe", i.e. we get as the result a set of constraints which must be satisfied. The constraints are of two kinds; typing restrictions on placeholders that must be fulfilled or equations that must hold. However, we also have the existence problem, since when we have a term with placeholders in it, there is always the question whether there exists instantiations or not. Therefore, when we talk about correctness of an incomplete proof term, we will have to relate correctness to *if there exists type correct instantiations of the remaining placeholders such that the equations hold.*

We will give an example which shows that the existence of instantiations and fulfilment of the equations may be contradictory.

**Ex.** Assume we define a set $Seq(A, n)$ (denoting a sequence of type $A$ and length $n$) with the two constructors *atom* and *cons*

$$\begin{array}{l} Seq : (Set; N)Set \\ \quad atom : (A{:}Set; a{:}A)Seq(A, 1) \\ \quad cons : (A{:}Set; a{:}A; n{:}N; l{:}Seq(A, n))Seq(A, s(n)) \end{array}$$

and an *append* function with the type

$$append : (A{:}Set; n, m{:}N; l_1{:}Seq(A, n); l_2{:}Seq(A, m))Seq(A, n + m).$$

Assume we want to find an element in $Seq(N, 2)$ by using the *append* function:

$$append(N, ?n, ?m, ?s_1, ?s_2) : Seq(N, 2)$$

where $?n, ?m, ?s_1$ and $?s_2$ are placeholders, with the type restrictions

$$\begin{array}{l} ?n, ?m : N, \\ ?s_1 : Seq(N, ?n) \text{ and} \end{array}$$

$?s_2 : Seq(N, ?m)$.

Now, we will do an instantiation of the placeholder $?s_1$ which leads to an incomplete term which is impossible to complete, but which cannot be detected by simplification of the equations. If $?s_1$ is refined by *cons*, we get the object

$$append(N, ?n, ?m, cons(N, ?a, ?n_1, ?s), ?s_2) : Seq(N, 2)$$

and the type checking will produce the constraints

$$?n = s(?n_1) \text{ and } ?n{+}?m = 2.$$

The first equation will instantiate $?n$, leaving us with the constraint

$$s(?n_1){+}?m = 2,$$

and we have to find two sequences

$$?s : Seq(N, ?n_1)$$
$$?s_2 : Seq(N, ?m).$$

Now, we can see that since a sequence always has a length $\geq 1$, it is impossible to both find instantiations of the remaining placeholders and satisfy the constraint.

Before we start explaining the generalisation of type checking, we have to introduce some notions. Placeholders denote unknown objects, and their types and contexts are determined by their respective occurrence in the incomplete term. Therefore, placeholders will be given their expected types and contexts during the type checking. This means that type checking will produce not only equations, but also *placeholder declarations* which are defined as follows:

**Definition 6.3** A *placeholder declaration* is a placeholder $?_j$ together with an expected (incomplete) type $\alpha_j$ and an expected (incomplete) context $\Gamma_j$. We will write

$$?_j : \alpha_j \; \Gamma_j$$

to denote a placeholder declaration.

Since we have dependent types and the term may contain placeholders, the equations produced by type checking may also contain placeholders. Hence we have a collection of equations containing unknown objects, and we are interested in finding assignments to the unknown such that the equations are satisfied. This is a unification problem, but since we in addition have typing restrictions on the unknowns, i.e. the placeholder declarations, we will call this a typed unification problem.

**Definition 6.4** A *typed unification problem* (TUP) is a collection of incomplete equations together with a placeholder declaration for every placeholder occuring in the equations or in the expected types and contexts of the placeholder declarations. We will call a TUP with type (term) equations a type-TUP (term-TUP), respectively.

We will denote the assignments of terms to placeholders for *instantiations*, despite the common terminology (substitutions), to not confuse instantiations with the explicit substitutions we have in our language, i.e. the assignments of terms to variables.

**Definition 6.5** An *instantiation* is a collection of assignments

$$\{?_1 = a_1, \ldots, ?_n = a_n\}$$

where all assigned placeholders $?_1, \ldots, ?_n$ are distinct and $a_1, \ldots, a_n$ are incomplete terms not containing the placeholders $?_1, \ldots, ?_n$. An instantiation is said to be *complete* if all $a_1, \ldots, a_n$ are complete terms. Instantiations will be denoted by $\sigma, \theta$ and $\rho$.

The *domain of an instantiation* $\sigma = \{?_1 = a_1, \ldots, ?_n = a_n\}$, is defined by $Dom(\sigma) = \{?_1, \ldots, ?_n\}$

We have chosen to require the assignments in instantiations to be independent of the other placeholders in the instantiations, to simplify the application of the instantiation. Because with this restriction, there is no order on the assignments, and any placeholder can be replaced by its assigned term in the instantiation independently of other assignments. Any collection of assignments which is not circular, can easily be put in such a form, by replacing the placeholders in the other assignments in some appropriate order.

**Definition 6.6** We will define what it means to apply an instantiation to incomplete terms, types and contexts, which will be denoted by $e\sigma, \alpha\sigma$ and $\Gamma\sigma$, respectively. A term or a substitution applied to an instantiation is mutually defined by:

$$\begin{array}{lcllcl}
x\sigma & = & x & (fa)\sigma & = & f\sigma(a\sigma) \\
c\sigma & = & c & ([x]b)\sigma & = & [x](b\sigma)^{(*)} \\
?_n\sigma & = & b,\ \text{if } ?_n = b \in \sigma & (e\gamma)\sigma & = & e\sigma(\gamma\sigma) \\
?_n\sigma & = & ?_n,\ \text{if } ?_n \notin Dom(\sigma) & \{\gamma, x{:=}e\}\sigma & = & \{\gamma\sigma, x{:=}e\sigma\} \\
 & & & \{\}\sigma & = & \{\}
\end{array}$$

Since instantiations only effect terms, they are simply distributed inside any type- or context structure.

(*) Note that $\sigma$ is simply moved inside the abstraction. The reason we can do this is that placeholders are treated as real open terms, which means that if $[x]b(?_n)$ is a correct term (in some context), then the scope of $?_n$ is already checked. Hence, any instantiation fitting the scope of $?_n$ will be a proper term inside the binder $x$.

Now we will explain what it means to apply an instantiation to a typed unification problem. The idea is, as we already mentioned, that we want the application of the instantiation to the TUP to mimic the result of type checking the instantiated term. Therefore, when there is a placeholder declaration in the TUP which is given an assignment in the instantiation, it will be replaced by the result of type checking the assignment. Otherwise, the type and context of the declaration are instantiated. For equations, the instantiation is applied to the respective parts.

**Definition 6.7** An instantiation $\sigma$ applied to a type-TUP $\xi$ is defined by

$$\begin{array}{lcl}
[\,]\sigma & = & [\,] \\
[\xi, \alpha = \alpha' \;\; \Gamma]\sigma & = & [\xi\sigma, \alpha\sigma = \alpha'\sigma \;\; \Gamma\sigma] \\
[\xi, ?_n : \alpha_n \;\; \Gamma_n]\sigma & = & \begin{cases} \xi\sigma \;\; @ \;\; GTE_p\,(\xi\sigma, b, \alpha\sigma, \Gamma\sigma), \text{ if } ?_n = b \in \sigma \\ [\xi\sigma, ?_n : \alpha_n\sigma \;\; \Gamma_n\sigma], \text{ otherwise} \end{cases}
\end{array}$$

and analogously, an instantiation $\sigma$ applied to a term-TUP $\mathcal{C}$ is defined by

$$\begin{array}{lcl}
[\,]\sigma & = & [\,] \\
[\mathcal{C}, a = b : \alpha \;\; \Gamma]\sigma & = & [\mathcal{C}\sigma, a\sigma = b\sigma : \alpha\sigma \;\; \Gamma\sigma] \\
[\mathcal{C}, ?_n : \alpha_n \;\; \Gamma_n]\sigma & = & \begin{cases} \mathcal{C}\sigma \;\; @ \;\; GE_p\,(\mathcal{C}\sigma, b, \alpha\sigma, \Gamma\sigma), \text{ if } ?_n = b \in \sigma \\ [\mathcal{C}\sigma, ?_n : \alpha_n\sigma \;\; \Gamma_n\sigma], \text{ otherwise} \end{cases}
\end{array}$$

### 6.2.1 Generating type equation

We will have to generalise the generation of type equations in two respects; first the arguments may be incomplete terms, types and contexts and secondly, we need another argument which is a type-TUP. The reason we need the extra argument is that we want to type check terms containing already known placeholders. These already known placeholders should be declared in the additional argument. Moreover, the type and context may contain placeholders, and just as for the complete case we need to know that the type is a correct type in the context. However, with this generalisation, we will instead require that the type (and context) are correct *relative* the type-TUP.

There is one additional restriction on the term $e$, which is that new placeholders cannot occur as the head of an application. The reason is the same as for the $\beta S$-normal restriction; we cannot compute the type of the function in an application. Since placeholders are given types during type checking, we can only allow new placeholders to occur where we can compute their types.

**Definition 6.8** We will say that a $\beta S$-normal term $e$ is *head-known* to $\xi$ if every sub-term of $e$ which is an application has a head which is either a constant, a variable or a placeholder declared in $\xi$.

**Generate Type Equations**

$GTE_p\,(\xi,\text{e},\alpha,\Gamma) \Rightarrow \xi$, where

$\xi$ is a type-TUP
$e$ is a $\beta S$-normal, $\Gamma$-distinct incomplete term which is head-known to $\xi$
$\alpha$ is an incomplete type
$\Gamma$ is an incomplete context

We will add the following new rules which concern placeholders:

$$\frac{}{GTE_p\,(\xi, ?_n, \alpha, \Gamma) \Rightarrow [?_n : \alpha_n \;\; \Gamma_n]}\;\text{GTE-PH1}\;(?_n \not\in \xi)$$

Here we have found a new placeholder, and we simply add it as a placeholder declaration to the TUP.

$$\frac{SC_p\,(\xi, \Gamma_n, \Gamma) \Rightarrow \xi'}{GTE_p\,(\xi, ?_n, \alpha, \Gamma) \Rightarrow \xi' \mathbin{@} [\langle \alpha, \alpha_n, \Gamma\rangle]}\;\text{GTE-PH2}\;(?_n : \alpha_n \;\; \Gamma_n \in \xi)$$

In this rule, we want to check the type of a placeholder which is already declared in $\xi$. It is done by verifying that the placeholder is used in a proper scope, that is the placeholder's context is a sub-context of the context of this occurrence. This is what $SC_p$ does. Then we must also check that the two occurrences have the same type, which corresponds to the added equation.

$$\frac{SC_p\,(\xi, \Gamma_n, \Gamma) \Rightarrow \xi'}{CT_p\,(\xi, ?_n, \Gamma) \Rightarrow \langle \xi', \alpha_n\rangle}\;\text{CT-PH}\;(?_n : \alpha_n \;\; \Gamma_n \in \xi)$$

Now we can see why we needed to restrict the term to be head-known relative $\xi$, since if $?_n$ was not in $\xi$, we would not know the type of $?_n$.

The two following rules are used to check that a known placeholder is used in a proper scope, where the first context is the local context of the placeholder in question and the other is the context in which it occurred.

$$\frac{}{SC_p\,(\xi, [\,], \Gamma) \Rightarrow [\,]}\;\text{SC-Empty}$$

If the placeholder is declared in the empty context, it cannot depend on any variable at all. Clearly such a placeholder can be used in any context $\Gamma$.

$$\frac{SC_p\ (\xi, \Delta, \Gamma) \Rightarrow \xi'}{SC_p\ (\xi, [\Delta, x : \alpha], \Gamma) \Rightarrow \xi' \mathbin{@} [\langle \alpha, \alpha', \Gamma \rangle]}\ \text{SC-Ext}\ {\scriptstyle (x\,:\,\alpha' \in \Gamma)}$$

To check that $[\Delta, x : \alpha]$ is a sub-context of $\Gamma$, we need to know that $\Delta$ is a sub-context of $\Gamma$ and that $x$ is a variable in $\Gamma$ with the same type as $x$ has in $\Delta$. The same type condition corresponds to the added equation.

Rules with two premisses are modified such that the second premiss depends also on the result of the first premiss. The reason is that the first premiss may generate a TUP which contains new placeholders, and the term in the second premiss should be type checked relative the extended TUP.

$$\frac{CT_p\ (\xi, f, \Gamma) \Rightarrow \langle \xi_1, \alpha' \to \beta \rangle \quad GTE_p\ (\xi \mathbin{@} \xi_1, e, \alpha', \Gamma) \Rightarrow \xi_2}{GTE_p\ (\xi, fe, \alpha, \Gamma) \Rightarrow \xi_1 \mathbin{@} \xi_2 \mathbin{@} [\langle \beta e, \alpha, \Gamma \rangle]}\ \text{GTE-App}$$

$$\frac{CT_p\ (\xi, f, \Gamma) \Rightarrow \langle \xi_1, \alpha \to \beta \rangle \quad GTE_p\ (\xi \mathbin{@} \xi_1, e, \alpha, \Gamma) \Rightarrow \xi_2}{CT_p\ (\xi, fe, \Gamma) \Rightarrow \langle \xi_1 \mathbin{@} \xi_2, \beta e \rangle}\ \text{CT-App}$$

$$\frac{FC_p\ (\xi, \gamma, \Delta, \Gamma) \Rightarrow \xi_1 \quad GTE_p\ (\xi \mathbin{@} \xi_1, a, \alpha\gamma, \Gamma) \Rightarrow \xi_2}{FC_p\ (\xi, \{\gamma, x{:=}a\}, [\Delta, x : \alpha], \Gamma) \Rightarrow \xi_1 \mathbin{@} \xi_2}\ \text{FC-Ext}$$

All other rules are simply modified to take $\xi$ as an additional argument.

It should be obvious that if we use this modified algorithm with an empty TUP and a complete term, type and context, it is exactly the algorithm presented in section 5.2, since the new rules only concern placeholders and the additional argument is only used to look up placeholder declarations.

### 6.2.2 Type conversion and conversion of incomplete terms

The next step is to modify the algorithms which simplify type- and term-TUPs. Just as for the complete case, we will have notions of well-formed TUPs. Recall the definition of a well-formed list a complete equations; there the type correctness of the equations depended on if the previous equations hold or not. For the TUPs, we will have to generalise this notion to consider all instantiations such that the equations applied to the instantiation hold.

In the remaining part of this chapter, we will often relate correctness statements to "all instantiations such that some conditions hold", so we will define a notion of this kind of statements:

**Definition 6.9** We will say that $\mathcal{J}$ *ensures* $\mathcal{K}$ if

$$\forall\sigma.\ \mathcal{J}\sigma \text{ holds} \Rightarrow \mathcal{K}\sigma \text{ holds}$$

where $\mathcal{J}$ and $\mathcal{K}$ are collections of incomplete judgements.

We have the following generalisation of the notion of well-formedness:

**Definition 6.10** A *well-formed TUP of type equations* are defined by

$$\frac{}{[\,]\ \textit{well-formed}}$$

$$\frac{\xi\ \textit{well-formed} \qquad \xi \text{ ensures } \Gamma \vdash \alpha : \mathit{Type} \text{ and } \Gamma \vdash \beta : \mathit{Type}}{[\xi, \alpha = \beta \quad \Gamma]\ \textit{well-formed}}$$

$$\frac{\xi\ \textit{well-formed} \qquad \xi \text{ ensures } \Gamma_n \vdash \alpha_n : \mathit{Type}}{[\xi, ?_n : \alpha_n \quad \Gamma_n]\ \textit{well-formed}}$$

Accordingly, we have the following definition of a well-formed term-TUP:

**Definition 6.11** A *well-formed TUP of term equations* are defined by

$$\frac{}{[\,]\ \textit{well-formed}}$$

$$\frac{\mathcal{C}\ \textit{well-formed} \qquad \mathcal{C} \text{ ensures } \Gamma \vdash a : \alpha \text{ and } \Gamma \vdash b : \alpha}{[\mathcal{C}, a = b : \alpha \quad \Gamma]\ \textit{well-formed}}$$

$$\frac{\mathcal{C}\ \textit{well-formed} \qquad \mathcal{C} \text{ ensures } \Gamma_n \vdash \alpha_n : \mathit{Type}}{[\mathcal{C}, ?_n : \alpha_n \quad \Gamma_n]\ \textit{well-formed}}$$

The simplification of a type-TUP, proceeds as before by replacing each equation by the result of applying the type conversion algorithm to the equation. The placeholder declarations are simply left unchanged, so we have to add this rule to the $TSimple_p$-algorithm:

$$\frac{\mathit{TSimple}_p\,(\xi) \Rightarrow \mathcal{C}}{\mathit{TSimple}_p\,[\xi, ?_n : \alpha_n \quad \Gamma_n] \Rightarrow \mathcal{C}\ @\ [?_n : \alpha_n \quad \Gamma_n]}$$

Since the type conversion is only concerned about the structure of the types and never "looks" inside an *El*-type (where there may occur placeholders), we do not have to modify the type conversion algorithm at all.

Similarly, we will have to add a rule to the simplification of term-TUPs, which does nothing with the placeholder declaration:

$$\frac{Simple_p\ (\mathcal{C}) \Rightarrow \mathcal{C}'}{Simple_p\ [\mathcal{C}, ?_n : \alpha_n \quad \Gamma_n] \Rightarrow \mathcal{C}' \ @\ [?_n : \alpha_n \quad \Gamma_n]}$$

We will sometimes denote $Simple_p\ (\mathcal{C}) \Rightarrow \mathcal{C}'$ by $\mathcal{C} \xrightarrow{S} \mathcal{C}'$, as an abbreviation.

**The Conv$_p$-algorithm**

The conversion rule which computes the arguments to head-normal form, is split into two rules depending on whether the term was reduced to a rigid term ($Rigid(e)$) or a flexible term ($Flex(e)$). For conversion we need not distinguish between a rigid term which is on head-normal term ($R_{hnf}(e)$) or a term which is irreducible ($R_{irr}(e)$)), since both terms are decomposed by the head-conversion rules. Hence, we have two new rules replacing the Conv-hnf rule:

$$\frac{a \xrightarrow{hnf} Rigid(a') \quad b \xrightarrow{hnf} Rigid(b') \quad HConv_p\ (a', b', \Gamma) \Rightarrow \langle \mathcal{C}, \alpha' \rangle}{Conv_p\ (a, b, \alpha, \Gamma) \Rightarrow \mathcal{C}}\ \text{Conv-rigid}$$

$$\frac{a \xrightarrow{hnf} Flex(a') \quad b \xrightarrow{hnf} L(b')}{Conv_p\ (a, b, \alpha, \Gamma) \Rightarrow [\langle a', b', \alpha, \Gamma \rangle]}\ \text{Conv-flex}$$

where $L \in \{Rigid, Flex\}$. Naturally, we also have the symmetric rule.

The other two rules, i.e. the rule which removes an equation between syntactically equal terms and the rule which applies new variables to both terms in the equation as long as they are of function types, are left unchanged. The head-conversion rules need not be altered either, since these are only used when we know that the head is rigid, which means it is a variable or a constant.

### 6.2.3 Reduction of incomplete terms

The general idea of reducing incomplete terms, is simply that the reduction rules are applied as before until there is a placeholder blocking any further reductions. An incomplete term which is reduced as far as possible will be of the form

$$e ::= ?_n \mid ?_n\gamma \mid ?_n(a_1, \ldots, a_n) \mid ?_n\gamma(a_1, \ldots, a_n) \mid c_i(b_1, \ldots, b_n)$$

where some $b_i$ is an incomplete term which prevents the pattern matching from being performed.

**Head normal form reduction**

The result of reducing an incomplete term will produce a *labelled* term, where the label denotes whether the term is rigid or flexible. Also, we need to distinguish between rigid terms which are on head-normal form and rigid terms which are irreducible, in the same way as before. Hence, we have

$$a \xrightarrow{hnf} L(a), \text{ where } L \in \{R_{hnf}, R_{irr}, Flex\}.$$

The $\delta$- and $\beta S$-reduction are unchanged.

**$\xrightarrow{hnf}$-reduction**

New rules:

$$head(a) \in \{?_n, ?_n\gamma\} : \quad \frac{}{a \xrightarrow{hnf} Flex(a)} \text{ Flex}$$

$$head(a) \in \{c_i\} : \quad \frac{a \xrightarrow{M} Flex(a)}{a \xrightarrow{hnf} Flex(a)} \text{ M-Flex}$$

The other rules are unchanged.

**$\xrightarrow{M}$-reduction**

New rule:

$$\frac{\langle prules, e\rangle \stackrel{M}{\Longrightarrow} Flexible}{e \xrightarrow{M} Flex(e)}$$

The old rules unchanged:

$$\frac{\langle prules, e\rangle \stackrel{M}{\Longrightarrow} Reduced(d\gamma)}{e \xrightarrow{M} Reduced(d\gamma)} \qquad \frac{\langle prules, e\rangle \stackrel{M}{\Longrightarrow} Irred}{e \xrightarrow{M} R_{irr}(e)}$$

**$\stackrel{M}{\Longrightarrow}$-rule matching**

New rule:

$$\frac{\langle p, e\rangle \Rightarrow_m Flexible}{\langle (p{=}d) \cup prules, e\rangle \stackrel{M}{\Longrightarrow} Flexible}$$

This rule says that it is enough to find that the term is flexible relative *one pattern*, since the term will be flexible relative all other patterns. It is guaranteed by the fact that patterns are non-overlapping and that the matching is only postponed if the pattern requires a constructor term whereas the term is flexible. Therefore, we must postpone the pattern matching, since we would not know which pattern matches. By the non-overlapping condition we know all other patterns in the same argument position will have a constructor patterns as well. We can conclude that if $prules$ is a set of non-overlapping patterns and for any $p \in prules$, we have

$$\langle p, e\rangle \Rightarrow_m Flexible$$

for some term $e$, then

$$\forall p' \in prules.\langle p', e\rangle \Rightarrow_m Flexible.$$

The other rules remain the same.

### $\Rightarrow_m$-pattern matching

For complete terms, pattern matching can either succeed with a substitution of the pattern variables, indicate that the term is irreducible, or fail when two distinct constructors are found. However, when the term may contain incomplete arguments, the matching may not be possible to decide until some placeholders are instantiated. The matching must respect the following rules:

- the matching succeeds if *all* arguments match,
- the matching fails if *at least one* argument fails to match,
- the matching is irreducible if *at least one* argument is irreducible,
- the matching is flexible if *at least one* argument is flexible and *no other* argument is irreducible.

These rules have the consequences that we must check all arguments to get a match, we may stop the matching when we find a failure (and try other patterns) and we may stop the matching completely when we encounter an irreducible term. When we reach a flexible argument, we cannot yet decide what to do, since we do not know if the pattern match. However, if some later argument turns out to be irreducible, we will not be able to reduce the term even when the incomplete argument is instantiated. Therefore, we know that a term is always irreducible, if it has an irreducible argument. Hence, we have to add the following rules:

$$\frac{\langle p, e\rangle \Rightarrow_m \gamma \quad a \xrightarrow{hnf} Flex(a')}{\langle (pp'), (ea)\rangle \Rightarrow_m Flexible}\; p' \neq x$$

Here we have a match so far, but the next argument is flexible, so the result of the match is flexible.

$$\frac{\langle p, e\rangle \Rightarrow_m Flexible \quad a \xrightarrow{hnf} R_{irr}(a')}{\langle (pp'), (ea)\rangle \Rightarrow_m Irred}\; p' \neq x$$

$$\frac{\langle p, e\rangle \Rightarrow_m Flexible \quad a \xrightarrow{hnf} R_{hnf}(a')}{\langle (pp'), (ea)\rangle \Rightarrow_m Irred}\; p' \neq x, head(a) = x$$

In the two last rules we can see that an irreducible argument "overrides" the previous flexible ones. The point is that even when the placeholders will become instantiated, this term will still be an irreducible term. The reason we are eager to label a term irreducible, is that in the conversion irreducible terms are decomposed whereas flexible terms are just left unchanged. This means that we can find more instantiations to placeholders automatically if we label this kind of terms as irreducible. Consider for instance the equation

$$add(x, y) = add(?_n, y) : N \quad [x, y : N]$$

where $x$ and $y$ are variables, so clearly $add(x, y)$ is an irreducible term. The question is whether $add(?_n, y)$ is irreducible or flexible. The above rules will label $add(?_n, y)$ irreducible, since even though the matching of the first argument will result in "Flexible", the matching of the second argument will change the label to irreducible, by the last rule. Hence, we have two irreducible terms and they will be decomposed and the arguments checked pairwise with the head-conversion algorithm, yielding an instantiation of $?_n$ to $x$.

## 6.3 Correctness proof

The correctness proof of type checking incomplete terms states that when we transform the type checking problem to a unification problem, the set of solutions is preserved, where a solution is an instantiation of the placeholders. We will call a solution to a type checking problem a unifier as well as solutions to unification problems, since in both cases the solution is an instantiation which satisfies the given condition. Hence, we have the following definition:

**Definition 6.12** The *set of unifiers* for a type checking problem, a type-TUP and a term-TUP are defined as follows:

$$
\begin{array}{lcl}
\mathcal{U}(\langle e, \alpha, \Gamma \rangle) & = & \{\sigma \mid \Gamma\sigma \vdash e\sigma : \alpha\sigma\} \\
\mathcal{U}(\xi) & = & \{\sigma \mid \xi\sigma \text{ holds}\} \\
\mathcal{U}(\mathcal{C}) & = & \{\sigma \mid \mathcal{C}\sigma \text{ holds}\}
\end{array}
$$

Then we have the main result (theorem 3) which says that if we get the unification problem $\mathcal{C}$ by type checking $e : \alpha \quad \Gamma$, then for any complete instantiation $\sigma$, we have

$\mathcal{C}\sigma$ holds if and only if $\Gamma\sigma \vdash e\sigma : \alpha\sigma$.

The type checking algorithm transforms the original type checking problem in three steps; first it is transformed to a unification problem of type equations ($GTE_p$), then to a unification problem of term equations ($TSimple_p$) and finally this term-TUP is simplified as far as possible by $Simple_p$. There are three properties we are interested in for the transformations

($U$) the set of unifiers is preserved,

($W$) the well-formedness is preserved, and

($C$) the transformation commutes with refinement instantiations.

Clearly we want the set of unifiers to be preserved, since the transformations should result in an equivalent problem. The well-formedness we need to be able to know that correctness of placeholders declarations are ensured by the previous equations and declarations. This is needed since instantiations are type checked relative the placeholder declarations and this is a requirement for the type checking to be correct.

The reason we want the transformations to commute with refinement instantiations is to show that the localisation of type checking a given instantiation relative its expected type is a correct optimisation of type checking the instantiated term. We will show that the following diagram commutes:

$$
\begin{array}{ccccccc}
e : \alpha \quad \Gamma & \overset{GTE_p}{\Longrightarrow} & \xi & \overset{TSimple_p}{\longrightarrow} & \mathcal{C} & \overset{Simple_p}{\longrightarrow} & \mathcal{C}' \\
 & & & & & & \downarrow \sigma \\
\downarrow \sigma & \boxed{\text{Prop } 16} & \downarrow \sigma & \boxed{\text{Prop } 18} & \downarrow \sigma & \boxed{\text{Prop } 24} & \mathcal{C}'\sigma \\
 & & & & & & \downarrow {\scriptstyle Simple_p} \\
e\sigma : \alpha \quad \Gamma & \overset{GTE_p}{\Longrightarrow} & \xi\sigma & \overset{TSimple_p}{\longrightarrow} & \mathcal{C}\sigma & \overset{Simple_p}{\longrightarrow} & \mathcal{C}''
\end{array}
$$

Note that the last transformation, the simplification of term equations, does not commute directly with the application of instantiations, we have to simplify again. The reason is that the equations in $\mathcal{C}'$ is not simplified further because one of the terms is flexible, but the term applied to the instantiation could possibly be reduced further. For example, we get the following picture with the instantiation $\sigma = \{?_1 = s(?_2)\}$:

$$\begin{tikzcd}
add(n, s(?_1)) = add(m, s(s(0))) \arrow[r, "Simple_p"] \arrow[dd, "\sigma"] & add(n, ?_1) = s(m) \arrow[d, "\sigma"] \\
 & add(n, s(?_2)) = s(m) \arrow[d, "Simple_p"] \\
add(n, s(s(?_2))) = add(m, s(s(0))) \arrow[r, "Simple_p"] & add(n, ?_2) = m
\end{tikzcd}$$

Therefore we need to simplify the equations again after the instantiation, to get the same equations as if the instantiation was applied before the simplification. However, we do get exactly the same equations either way, since the reduction of terms is completely deterministic, and a flexible term is simply postponing the rest of the reductions to head-normal form until it got an instantiation of the placeholders.

We will show the three correctness properties for the transformations in turns.

### 6.3.1 Correctness of $GTE_p$

First we have the proposition which shows that the transformation from a type checking problem to a unification problem of type equations preserves the set of unifiers and commutes with a refinement instantiation. The proposition has two parts depending on if the term contains only new placeholders $(i)$ or if it contains placeholders which are previously declared $(ii)$. For this part we have a weaker result about the commutation with instantiations, we do not get exactly the same unification problem and this is because we have sharing in the unification problem but not in the type checking problem. Part $(ii)$ will be used in the soundness proof of the unification algorithm in the next chapter.

The proof relies on the next proposition, that is the well-formedness property.

**Proposition 16**
*Let $\xi$ be a well-formed type-TUP such that $\xi$ ensures $\Gamma \vdash \alpha$ : Type. If*

$$GTE_p\ (\xi, e, \alpha, \Gamma) \Rightarrow \xi',$$

*then*

(*i*) *If e is a refinement term relative* $\xi$, *then*

> $(C)$ $GTE_p\ (\xi\sigma, e\sigma, \alpha\sigma, \Gamma\sigma) \Rightarrow \xi'\sigma$
>
> $(U)$ $\mathcal{U}(\langle e, \alpha, \Gamma\rangle) = \mathcal{U}(\xi \mathbin{@} \xi')$

(*ii*) *If e is a unification term relative* $\xi$ *then*

> $(C)$ $GTE_p\ (\xi\sigma, e\sigma, \alpha\sigma, \Gamma\sigma) \Rightarrow \xi''$, *and* $\mathcal{U}(\xi\sigma \mathbin{@} \xi'') \supseteq \mathcal{U}(\xi\sigma \mathbin{@} \xi'\sigma)$
>
> $(U)$ $\mathcal{U}(\langle e, \alpha, \Gamma\rangle) \supseteq \mathcal{U}(\xi \mathbin{@} \xi')$

***Proof:*** First, we can note that $(i)(U)$ follows from $(i)(C)$ and the corresponding case for complete terms, i.e. the soundness proof for $GTE$, since we have the precondition that $\alpha$ and $\Gamma$ are proper type and context, respectively, relative to $\xi$. Analogously, $(ii)(U)$ follows from $(ii)(C)$ and $GTE$-soundness.

For the cases $(i)(C)$ and $(ii)(C)$ we will show that if

$$GTE_p\ (\xi\sigma, e\sigma, \alpha\sigma, \Gamma\sigma) \Rightarrow \xi''$$

then

$$\xi\sigma \mathbin{@} \xi'\sigma = \xi\sigma \mathbin{@} \xi'', \qquad \text{for the case } (i), \text{ and}$$
$$\mathcal{U}(\xi\sigma \mathbin{@} \xi'\sigma) \supseteq \mathcal{U}(\xi\sigma \mathbin{@} \xi''), \qquad \text{for case } (ii).$$

However, since the case $(i)$ clearly implies case $(ii)$, we will only show this case when it holds. The proof is by induction on the structure of $e$.

*Var:*

$$\frac{}{GTE_p\ (\xi, x, \alpha, \Gamma) \Rightarrow [\langle \alpha, \alpha', \Gamma\rangle]}\ {\scriptstyle (x\,:\,\alpha' \,\in\, \Gamma)}$$

Consider $GTE_p\ (\xi\sigma, x\sigma, \alpha\sigma, \Gamma\sigma) \Rightarrow \xi''$. Since $x\sigma = x$, we can apply the Var-rule and since $x : \alpha' \in \Gamma$ we have $x : \alpha'\sigma \in \Gamma\sigma$. Hence, $\xi'' = [\langle \alpha\sigma, \alpha'\sigma, \Gamma\sigma\rangle] = [\langle \alpha, \alpha', \Gamma\rangle]\sigma$.

*Const:* Analogously.

*Placeholder 1:*

$$\frac{}{GTE_p\ (\xi, ?_n, \alpha, \Gamma) \Rightarrow [?_n : \alpha_n \quad \Gamma_n]}\ {\scriptstyle \begin{array}{l}\text{GTE-PH1}\\ (?_n\ \notin\ \xi\ )\end{array}}$$

We have that

$$GTE_p\ (\xi\sigma, ?_n\sigma, \alpha\sigma, \Gamma\sigma) \Rightarrow \begin{cases} GTE_p(\xi\sigma, b, \alpha\sigma, \Gamma\sigma) & \text{if } ?_n = b \in \sigma \\ [?_n : \alpha\sigma \quad \Gamma\sigma] & \text{otherwise} \end{cases}$$

which is exactly the definition of $\xi\sigma \mathbin{@} [?_n : \alpha \quad \Gamma]\sigma$.

*Placeholder 2:*

$$\frac{SC_p\,(\xi, \Gamma_n, \Gamma) \Rightarrow \xi'}{GTE_p\,(\xi, ?_n, \alpha, \Gamma) \Rightarrow [\langle \alpha, \alpha_n, \Gamma \rangle]}\ \text{GTE-PH2}\ (?_n : \alpha_n \ \ \Gamma_n \in \xi)$$

By lemma 16.1, we have that $SC_p\,(\xi\sigma, \Gamma_n\sigma, \Gamma\sigma) \Rightarrow \xi'\sigma$. We have two cases

1. $?_n \notin Dom(\sigma)$. Then $(i)$ holds, since
$$GTE_p\,(\xi\sigma, ?_n\sigma, \alpha\sigma, \Gamma\sigma) \Rightarrow \xi'\sigma \mathbin{@} [\langle \alpha\sigma, \alpha_n\sigma, \Gamma\sigma \rangle].$$

2. $?_n = b \in \sigma$. Then $(i)$ does not hold, but we will show $(ii)$, that is if
$$GTE_p\,(\xi\sigma, ?_n\sigma, \alpha\sigma, \Gamma\sigma) \Rightarrow \xi''$$
then
$$\mathcal{U}(\xi\sigma \mathbin{@} \xi'\sigma \mathbin{@} [\langle \alpha\sigma, \alpha_n\sigma, \Gamma\sigma \rangle]) \subseteq \mathcal{U}(\xi\sigma \mathbin{@} \xi'')$$
We know that $?_n : \alpha_n \ \ \Gamma_n \in \xi$, so we have the following situation for the first case
$$\xi_1 \left\{ \begin{array}{c} \left.\left[ \begin{array}{c} \vdots \\ GTE_p(b, \alpha_n\sigma, \Gamma_n\sigma) \\ \vdots \end{array} \right]\right\} \xi\sigma \\ + \\ \left.[\ SC_p(\xi\sigma, \Gamma_n\sigma, \Gamma\sigma)\ ]\right\} \xi'\sigma \\ + \\ [\ \alpha\sigma = \alpha_n\sigma \ \ \Gamma\sigma\ ] \end{array} \right.$$
and we need to show that any unifier of $\xi_1$ is also a unifier of $\xi_2$, that is satisfying the second unification problem:
$$\xi_2 \left\{ \begin{array}{c} \left.\left[ \begin{array}{c} \vdots \\ GTE_p(b, \alpha_n\sigma, \Gamma_n\sigma) \\ \vdots \end{array} \right]\right\} \xi\sigma \\ + \\ \left.[\ GTE_p(b, \alpha\sigma, \Gamma\sigma)\ ]\right\} \xi'' \end{array} \right.$$
Now, for any $\sigma^*$ such that $\xi\sigma^*$ holds we have
$$\Gamma_n\sigma^* \vdash b\sigma^* : \alpha_n\sigma^*$$
$$\Gamma_n\sigma^* \preceq \Gamma\sigma^*$$

$$\Gamma\sigma^* \vdash \alpha_n\sigma^* = \alpha\sigma^* : \mathit{Type}$$

which implies

$$\Gamma\sigma^* \vdash b\sigma^* : \alpha\sigma^*.$$

The converse, on the other hand, does not hold since we do not have unicity of types.

The remaining cases are straightforward by the induction hypothesis, due to proposition 17.

□

**Lemma 16.1**
*(C) If $SC_p\,(\xi, \Gamma_n, \Gamma) \Rightarrow \xi'$, then $SC_p\,(\xi\sigma, \Gamma_n\sigma, \Gamma\sigma) \Rightarrow \xi'\sigma$*

***Proof:*** Induction on the structure of $\Gamma_n$ □

Next, we have the proposition which says that $GTE_p$ produces a well-formed type-TUP, and if the arguments already depend on a unification problem, the appended TUPs are well-formed.

**Proposition 17**
*Let $\xi$ be a well-formed type-TUP such that $\xi$ ensures $\Gamma \vdash \alpha : \mathit{Type}$, $\Gamma : \mathit{Context}$ and $\Delta : \mathit{Context}$ for (i), (ii) and (iii), respectively. Then we have the following (W)-properties*

*(i) if $GTE_p\,(\xi, q, \alpha, \Gamma) \Rightarrow \xi'$, then $\xi \,@\, \xi'$ is well-formed,*
*(ii) if $CT_p\,(\xi, q, \Gamma) \Rightarrow \langle \xi', \alpha\rangle$, then $\xi \,@\, \xi'$ is well-formed and $(\xi \,@\, \xi')$ ensures $\Gamma \vdash \alpha : \mathit{Type}$,*
*(iii) if $FC_p\,(\xi, q, \Delta, \Gamma) \Rightarrow \xi'$, then $\xi \,@\, \xi'$ is well-formed.*

***Proof:*** The proof is by induction on the structure of $q$.

*(i)*

*GTE-Var:* By precondition we have $\Gamma\sigma \vdash \alpha\sigma : \mathit{Type}$ and $\Gamma\sigma : \mathit{Context}$, for any $\sigma$ such that $\xi\sigma$ holds. Hence, $\Gamma\sigma \vdash \alpha'\sigma : \mathit{Type}$ since $x : \alpha' \in \Gamma$, so we have that $\xi \,@\, [\langle \alpha, \alpha', \Gamma\rangle]$ is well-formed.

*GTE-Const:* Analogously.

*GTE-PH1:* By the precondition.

*GTE-PH2:* Consider the rule

$$\frac{SC_p\ (\xi, \Gamma_n, \Gamma) \Rightarrow \xi'}{GTE_p\ (\xi, ?_n, \alpha, \Gamma) \Rightarrow [\langle \alpha, \alpha_n, \Gamma \rangle]}\ \substack{\text{GTE-PH2} \\ (?_n \,:\, \alpha_n \quad \Gamma_n \in \xi)}$$

By precondition we have

(1) $\forall \sigma . \xi\sigma$ holds $\Rightarrow \Gamma\sigma \vdash \alpha\sigma : \mathit{Type}.$

Since $?_n$ is declared in $\xi$, we also have

(2) $\forall \sigma . \xi\sigma$ holds $\Rightarrow \Gamma_n\sigma \vdash \alpha_n\sigma : \mathit{Type}.$

Thus $\Gamma$ and $\Gamma_n$ are proper contexts relative to $\xi$, so we can apply lemma 17.1, and by $(i)$ we have that $\xi @ \xi'$ is well-formed. We need to show

$$\forall \sigma' . (\xi @ \xi')\sigma' \text{ holds} \Rightarrow \Gamma\sigma' \vdash \alpha\sigma' : \mathit{Type} \wedge \Gamma\sigma' \vdash \alpha_n\sigma' : \mathit{Type}.$$

The first holds by (1), since $\sigma'$ is more restricted than $\sigma$. The other follows by (2) and lemma 17.1$(ii)$, since then we know $\Gamma_n\sigma' \preceq \Gamma\sigma'$. Hence, $\xi @ \xi' @ [\langle \alpha, \alpha_n, \Gamma \rangle]$ is well-formed.

*GTE-Abs:* By induction hypothesis.

*GTE-App:* By $(ii)$ we have that the preconditions of the second premiss is satisfied, and hence the desired result follows by the induction hypothesis and proposition GTEp-sigma$(ib)$.

*GTE-Subst:* The premiss is well-formed by the induction hypothesis, and hence $\gamma$ fits context $\Gamma_c$ (relative $\xi @ \xi'$) which means that $\alpha_c\gamma$ is well-typed. Hence, $\xi @ \xi' @ [\langle \alpha_c\gamma, \alpha, \Gamma \rangle]$ is well-formed.

$(ii)$

*CT-Var:* We know (by the precondition) that $\Gamma\sigma : \mathit{Context}$, so $\Gamma\sigma \vdash \alpha\sigma : \mathit{Type}$ since $x : \alpha \in \Gamma$ and $\xi$ is well-formed by assumption.

*CT-Const:* Analogously.

*CT-App:* By induction hypothesis, $(i)$ and proposition GTEp-sigma$(ib)$.

*CT-Placeholder:* By lemma 17.1.

$(iii)$

*FC-empty:* Immediate.

*FC-ext:* Analogous to the *GTE-Subst* case.

□

**Lemma 17.1**
*Assume $\xi$ is well-formed and that $\xi$ ensures $\Delta$ : Context and $\Gamma$ : Context. If $SC_p\,(\xi, \Delta, \Gamma) \Rightarrow \xi'$, then*

$(W)\quad \xi \,@\, \xi'$ *is well-formed, and*
$(U)\quad \mathcal{U}(\xi \,@\, \xi') = \mathcal{U}(\Delta \preceq \Gamma)$

***Proof:*** By induction on the structure of $\Delta$. □

### 6.3.2 Correctness of *TSimple*$_p$

The simplification of type equations is done by replacing each type equation with the result of the type conversion algorithm which yield a list of term equations. Hence, we have to show the desired properties for the type conversion as well, and they will appear after the propositions about the type simplification.

Next is the proposition and proof of the middle part of the diagram, which says that *TSimple*$_p$ commutes with an instantiation. Here we do not need the requirement that the instantiation is a refinement instantiation since all placeholder are already shared in the unification problem.

**Proposition 18**
$(C)$ *If $TSimple_p\,(\xi) \Rightarrow \mathcal{C}$, then $TSimple_p\,(\xi\sigma) \Rightarrow \mathcal{C}\sigma$,*

***Proof:*** Induction on the length of $\xi\sigma$.

$\xi = [\,]$ Immediate.

$\xi = [\xi, \alpha = \alpha' \quad \Gamma]$ By induction hypothesis and proposition 21.

$\xi = [\xi, ?_n : \alpha_n \quad \Gamma_n]$ We have that

$$[\xi, ?_n : \alpha_n \quad \Gamma_n]\sigma = \begin{cases} (1) \quad \xi\sigma \,@\, GTE_p(\xi\sigma, b, \alpha\sigma, \Gamma\sigma) & \text{if } ?_n = b \in \sigma \\ (2) \quad [\xi\sigma, ?_n : \alpha_n\sigma \quad \Gamma_n\sigma] & \text{otherwise} \end{cases}$$

(1) By induction hypothesis we have $TSimple_p\,(\xi\sigma) \Rightarrow \mathcal{C}\sigma$. We also have that $TSimple_p\,(GTE_p(\xi\sigma, b, \alpha\sigma, \Gamma\sigma)) \Rightarrow GE_p(\mathcal{C}\sigma, b, \alpha\sigma, \Gamma\sigma)$ by induction, since the list produced by $GTE_p$ is a sublist of $\xi\sigma$.

(2) holds by the induction hypothesis since the placeholders declaration is left unchanged.

□

**Proposition 19**
*If $\xi$ is well-formed and $TSimple_p\,(\xi) \Rightarrow \mathcal{C}$, then*

$(W)$ $\quad \mathcal{C}$ *is well-formed*

***Proof:*** By induction on the length of $\xi$.

$\xi = [\,]$ Immediate.

$\xi = [\xi', \alpha = \alpha' \quad \Gamma]$ Assume $TSimple_p\,(\xi') \Rightarrow \mathcal{C}$. We have by induction hypothesis

$(IH)$ $\mathcal{C}$ is well-formed

Since $\xi$ is assumed to be well-formed, we know

$(*)$ $\xi'$ ensures $\Gamma \vdash \alpha : Type \wedge \Gamma \vdash \alpha' : Type$,

but since $\xi'$ and $\mathcal{C}$ have the same set of unifiers (20), we know that also $\mathcal{C}$ ensures $(*)$. Hence, we get well-formedness by $(IH)$ and proposition 22.

$\xi = [\xi', ?_n : \alpha_n \quad \Gamma_n]$ Immediate by the induction hypothesis and proposition 20.

□

**Proposition 20**
*If $TSimple_p\,(\xi) \Rightarrow \mathcal{C}$, then*

$(U)$ $\quad \mathcal{U}(\xi) = \mathcal{U}(\mathcal{C})$

***Proof:*** By induction on the length of $\xi$, using that $TConv_p$ preserves unifiers (23). □

We also have to verify the well-formedness, the preservation of unifiers and the commutation with instantiations for the type conversion algorithm, which are the following two propositions:

**Proposition 21**
$(C)$ $\quad$ *If $TConv_p\,(\alpha, \alpha', \Gamma) \Rightarrow \mathcal{C}$, then $TConv_p\,(\alpha\sigma, \alpha'\sigma, \Gamma\sigma) \Rightarrow \mathcal{C}\sigma$*

***Proof:*** By induction on the derivation of $TConv_p\,(\alpha, \alpha', \Gamma) \Rightarrow \mathcal{C}$, where we use lemma 21.1 in the $TConv - \beta Subst$-rule. □

**Lemma 21.1** *If $\alpha \xrightarrow{T\beta S} \alpha'$, then $\alpha\sigma \xrightarrow{T\beta S} \alpha'\sigma$.*

***Proof:*** Case analysis on $\alpha \xrightarrow{T\beta S} \alpha'$. □

**Proposition 22**
*Let $\mathcal{C}$ be a well-formed unification problem which ensures $\Gamma \vdash \alpha_1 : Type$ and $\Gamma \vdash \alpha_2 : Type$. If $TConv_p\ (\alpha_1, \alpha_2, \Gamma) \Rightarrow \mathcal{C}'$, then we have the property*

$(W)$ $\quad \mathcal{C} \mathbin{@} \mathcal{C}'$ *is well-formed*

***Proof:*** Induction on the length of the derivation $TConv_p\ (\alpha_1, \alpha_2, \Gamma) \Rightarrow \mathcal{C}'$.

*TConv-id:* Immediate.

*TConv-El:* We have by assumption that $\Gamma\sigma \vdash El(A)\sigma : Type$, for any $\sigma$ such that $\mathcal{C}$ holds. Since *El* is a type constructor this implies $\Gamma\sigma \vdash A\sigma : Set$, so clearly $C \mathbin{@} [A = B : Set \;\; \Gamma]$ is well-formed.

*TConv-fun:* Follows by the induction hypothesis.

*TConv-βSubst:* We have by assumption that $\Gamma\sigma \vdash \alpha_1\sigma : Type$. By lemma 21.1 we have $\alpha_1\sigma \xrightarrow{T\beta S} \alpha_1'\sigma$ and by lemma 4.1 that $\xrightarrow{T\beta S}$ preserves types, so we can apply the induction hypothesis to the premiss. Thus, $(W)$ holds.

□

**Proposition 23** *If $TConv_p\ (\alpha_1, \alpha_2, \Gamma) \Rightarrow \mathcal{C}$, then*

$(U)$ $\quad \mathcal{U}(\alpha_1 = \alpha_2 \;\; \Gamma) = \mathcal{U}(\mathcal{C}')$.

***Proof:*** Induction on the length of the derivation $TConv_p\ (\alpha_1, \alpha_2, \Gamma) \Rightarrow \mathcal{C}$.

*TConv-id:* Immediate.

*TConv-El:* Constructors are assumed to be one-to-one, thus

$$\forall\sigma.\Gamma\sigma \vdash El(A)\sigma = El(B)\sigma : Type \Leftrightarrow \Gamma\sigma \vdash A\sigma = B\sigma : Set$$

which gives us $(U)$.

*TConv-fun:* Follows by the induction hypothesis.

*TConv-βSubst:* Assume $\Gamma\sigma \vdash \alpha_1\sigma = \alpha_2\sigma : Type$. By lemma 21.1 we have that $\alpha_1\sigma \xrightarrow{T\beta S} \alpha_1'\sigma$ and by lemma 4.1 $\Gamma\sigma \vdash \alpha_1\sigma = \alpha_1'\sigma : Type$. Analogously for $\alpha_2$. Thus, we have by transitivity that $\Gamma\sigma \vdash \alpha_1'\sigma = \alpha_2'\sigma : Type$, so $(U)$ follows by the induction hypothesis.

□

### 6.3.3 Correctness of $Simple_p$

We have to verify the properties (C), (W) and (U) also for the transformation $Simple_p$, that is the simplification of term equations. These depends on the corresponding proofs for the conversion algorithm which will appear later in this section.

**Proposition 24**
*(C)* *If* $\mathcal{C} \xrightarrow{S} \mathcal{C}'$ *and* $\mathcal{C}\sigma \xrightarrow{S} \mathcal{C}''$, *then* $\mathcal{C}'\sigma \xrightarrow{S} \mathcal{C}''$.

***Proof:*** Induction on the length of $\mathcal{C}$. The only non-trivial case is when $\mathcal{C} = [\mathcal{C}, a = b : \alpha \ \ \Gamma]$. We must verify that the following picture commutes:

$$\begin{array}{ccc}
\mathcal{C} & \xrightarrow{S} & \mathcal{C}' \\
 & & \downarrow \sigma \\
\Big\downarrow \sigma & & \mathcal{C}'\sigma \\
 & & \downarrow S \\
\mathcal{C}\sigma & \xrightarrow{S} & \mathcal{C}''
\end{array}
\qquad \text{which follows by} \qquad
\begin{array}{ccc}
Conv_p(a, b, \alpha, \Gamma) & \Rightarrow & \mathcal{C} \\
 & & \downarrow \sigma \\
\Big\downarrow \sigma & & \mathcal{C}\sigma \\
 & & \downarrow S \\
Conv_p(a\sigma, b\sigma, \alpha\sigma, \Gamma\sigma) & \Rightarrow & \mathcal{C}'
\end{array}$$

that is, by proposition 27.

$\Box$

**Proposition 25**
*If* $\mathcal{C}$ *is well-formed and* $Simple_p\ (\mathcal{C}) \Rightarrow \mathcal{C}'$, *then*

*(W)* $\mathcal{C}'$ *is well-formed*

***Proof:*** Analogous to the *TSimple*-case, by using propositions 28 and 29 instead. $\Box$

**Proposition 26**
*If* $Simple_p\ (\mathcal{C}) \Rightarrow \mathcal{C}'$, *then*

*(U)* $\mathcal{U}(C) = \mathcal{U}(C')$

***Proof:*** By induction on $\mathcal{C}$, using proposition 29. $\Box$

Finally, we have the same three properties for the conversion algorithm:

**Proposition 27**
*We have the following two properties*
$(C_1)$ *if* $Conv_p\ (a, b, \alpha, \Gamma) \Rightarrow \mathcal{C}$ *and* $Conv_p\ (a\sigma, b\sigma, \alpha\sigma, \Gamma\sigma) \Rightarrow \mathcal{C}'$*, then* $\mathcal{C}\sigma \xrightarrow{S} \mathcal{C}'$*.*
$(C_2)$ *if* $HConv_p\ (a, b, \Gamma) \Rightarrow \langle \mathcal{C}, \alpha \rangle$ *and* $HConv_p\ (a\sigma, b\sigma, \Gamma\sigma) \Rightarrow \langle \mathcal{C}', \alpha' \rangle$*, then we have* $\mathcal{C}\sigma \xrightarrow{S} \mathcal{C}'$ *and* $\alpha\sigma = \alpha'$*.*

***Proof:*** Induction on the length of the derivations $Conv_p\ (a, b, \alpha, \Gamma) \Rightarrow \mathcal{C}$ and $HConv_p\ (a, b, \Gamma) \Rightarrow \langle \mathcal{C}, \alpha \rangle$, respectively.

*Conv-id* Immediate.

*Conv-fun* Follows by the induction hypothesis and the fact that an instantiation is distributed inside term- and type constructions.

*Conv-rigid* Follows by lemma 27.1 and $(C_2)$.

*Conv-flex* We need to show

$$(*) \text{ if } Conv_p\ (a\sigma, b\sigma, \alpha\sigma, \Gamma\sigma) \Rightarrow \mathcal{C}, \text{ then } [a'\sigma = b'\sigma : \alpha\sigma \quad \Gamma\sigma] \xrightarrow{S} \mathcal{C},$$

where $a \xrightarrow{hnf} Flex(a')$ and $b \xrightarrow{hnf} L(b')$ $(L \in \{Flex, R_irr, R_hnf\})$. By lemma 27.1 we have that

$$\text{if } a\sigma \xrightarrow{hnf} L(a''), \text{ then } a\sigma \xrightarrow{hnf} L(a'')$$

and also

$$\text{if } e \xrightarrow{hnf} L(e') \text{ and } e\sigma \xrightarrow{hnf} L'(e''), \text{ then } e'\sigma \xrightarrow{hnf} L'(e'').$$

Thus, we have the following pictures

$$\begin{array}{ccc}
a & \xrightarrow{hnf} & Flex(a') \\
 & & \downarrow \sigma \\
\Big\downarrow \sigma & & a'\sigma \\
 & & \downarrow hnf \\
a\sigma & \xrightarrow{hnf} & L_a(a'')
\end{array}
\qquad \text{and} \qquad
\begin{array}{ccc}
b & \xrightarrow{hnf} & L(b') \\
 & & \downarrow \sigma \\
\Big\downarrow \sigma & & a'\sigma \\
 & & \downarrow hnf \\
b\sigma & \xrightarrow{hnf} & L_b(b'')
\end{array}$$

Now, we can see that $Conv_p(a\sigma, b\sigma, \alpha\sigma, \Gamma\sigma)$ must be reduced to checking $Conv_p(a'', b'', \alpha\sigma, \Gamma\sigma)$. Moreover, $[a'\sigma = b'\sigma : \alpha\sigma \quad \Gamma\sigma]$ is simplified by $Conv_p(a'\sigma, b'\sigma, \alpha\sigma, \Gamma\sigma)$, which is also reduced to $Conv_p(a'', b'', \alpha\sigma, \Gamma\sigma)$ as shown in the picture above. Hence, $(*)$ holds.

*HConv-head* Immediate.

*HConv-app* Follows by the induction hypothesis.

□

**Lemma 27.1**

*(i) If* $a \xrightarrow{hnf} Rigid(a')$*, then* $a\sigma \xrightarrow{hnf} Rigid(a'\sigma)$*,*
*(ii) if* $a \xrightarrow{M} Reduced(a')$*, then* $a\sigma \xrightarrow{M} Reduced(a'\sigma)$*,*
*(iii) if* $a \xrightarrow{M} Irr(a)$*, then* $a\sigma \xrightarrow{M} Irr(a\sigma)$*, and*
*(iv) if* $a \xrightarrow{hnf} Flex(a')$ *and* $a\sigma \xrightarrow{hnf} L(a'')$*, then* $a'\sigma \xrightarrow{hnf} L(a'')$*.*

***Proof:*** All three cases are proved by induction on the length of the corresponding derivations.

$(i)$ We will have to consider all rules except the Flex- and M-Flex rules.

*Hnf:* The term $a = b(a_1, \ldots, a_n)$, where $b$ is a variable or constant, so $a\sigma = b(a_1\sigma, \ldots, a_n\sigma)$.

*Unfold:* By lemma 27.1.1, the induction hypothesis and the Unfold-rule.

*$\beta$Subst:* By lemma 27.1.2, the induction hypothesis and the $\beta$Subst-rule.

*Match:* By $(ii)$, the induction hypothesis and the Match-rule.

*Irred:* By $(iii)$ and the Irred-rule.

$(ii)$ We need to show that

if $\langle plist, e\rangle \stackrel{M}{\Longrightarrow} Reduced(d\gamma)$, then $\langle plist, e\sigma\rangle \stackrel{M}{\Longrightarrow} Reduced((d\gamma)\sigma)$.

We have $(d\gamma)\sigma = d\sigma(\gamma\sigma) = d(\gamma\sigma)$ since $d$ is the right-hand side of some pattern rule in the theory, so $d$ can not contain any placeholders. Thus, it suffices to show

if $\langle p, e\rangle \Rightarrow_m \gamma$, then $\langle p, e\sigma\rangle \Rightarrow_m \gamma\sigma$,

which holds by induction of the length of $\langle p, e\rangle \Rightarrow_m \gamma$, using $(i)$.

$(iii)$ We know that

if $a \xrightarrow{M} Irr(a)$, then $\exists p \in plist.\langle p, e\rangle \Rightarrow_m Irred$.

Hence, we need to show

if $\langle p, e\rangle \Rightarrow_m Irred$, then $\langle p, e\sigma\rangle \Rightarrow_m Irred$.

It is proved by a straightforward induction on $\langle p, e\rangle \Rightarrow_m Irred$, using $(i)$.

$(iv)$ The intuitive picture is that placeholders only stops the (deterministic) reduction at the point where a placeholder is blocking further reductions, and when that placeholder is instantiated, the reduction may proceed:

$$
\begin{array}{ccccc}
 & a & & & a\sigma \\
 & \downarrow & \leftrightarrow & & \downarrow \\
 & \vdots & & & \vdots \\
 & \downarrow & \leftrightarrow & & \downarrow \\
\sigma \Rightarrow & \dfrac{Flex(a')}{a'\sigma} & & & a'\sigma \\
 & & \searrow & \swarrow & \\
 & & \vdots & & \\
 & & L(a'') & &
\end{array}
$$

We will show this case by induction on the derivation of $a \xrightarrow{hnf} Flex(a')$.

*Flex,P-Flex:* Immediate.

*Unfold:* We have that

$$a \xrightarrow{\delta} a_0 \xrightarrow{hnf} Flex(a')$$

and by lemma 27.1.1 we have $a\sigma \xrightarrow{\delta} a_0\sigma$. Further, the induction hypothesis gives

if $a_0\sigma \xrightarrow{hnf} L(a^*)$, then $a'\sigma \xrightarrow{hnf} L(a^*)$

so we get the following sequence of reduction

$$a\sigma \xrightarrow{\delta} a_0\sigma \xrightarrow{hnf} L(a^*)$$

which is what we wanted to show.

*$\beta$Subst:* Analogously, by using lemma 27.1.2.

*Match:* Analogously, by using $(ii)$.

The other rules are not applicable.

□

**Lemma 27.1.1** *If $a \xrightarrow{\delta} a'$, then $a\sigma \xrightarrow{\delta} a'\sigma$.*

***Proof:*** Case analysis on $a \xrightarrow{\delta} a'$. □

**Lemma 27.1.2** *If $a \xrightarrow{\beta S} a'$, then $a\sigma \xrightarrow{\beta S} a'\sigma$.*

***Proof:*** Case analysis on $a \xrightarrow{\beta S} a'$. □

**Proposition 28**
*Let $\mathcal{C}$ be a well-formed unification problem which ensures $a \vdash \alpha : \Gamma$ and $b \vdash \alpha : \Gamma$. If $Conv_p\ (a, b, \alpha, \Gamma) \Rightarrow \mathcal{C}'$, then*

$(W)$ $\mathcal{C} @ \mathcal{C}'$ *is well-formed*

***Proof:*** We must simultaneously show the property for *HConv*, and the proof is by induction on the length of the respective derivations of $Conv_p\ (a, b, \alpha, \Gamma) \Rightarrow \mathcal{C}'$ and $HConv_p\ (a, b, \Gamma) \Rightarrow \langle \mathcal{C}', \alpha \rangle$.

*Conv-id:* Immediate.

*Conv-fun:* By induction hypothesis.

*Conv-rigid:* Consider the rule

$$\frac{a \xrightarrow{hnf} Rigid(a') \quad b \xrightarrow{hnf} Rigid(b') \quad HConv_p\ (a', b', \Gamma) \Rightarrow \langle \mathcal{C}, \alpha' \rangle}{Conv_p\ (a, b, \alpha, \Gamma) \Rightarrow \mathcal{C}}\ \text{Conv-rigid}$$

We have $\Gamma\sigma \vdash a\sigma : \alpha\sigma$ and $\Gamma\sigma \vdash b\sigma : \alpha\sigma$ for an arbitrary $\sigma$ satisfying $\mathcal{C}$, by assumption. By lemma 27.1$(i)$ we have $a\sigma \xrightarrow{hnf} a'\sigma$ and by lemma 6.1 that $\Gamma\sigma \vdash a'\sigma : \alpha\sigma$ (analogously for $b$). Hence, we can apply the induction hypothesis to the last premiss getting

$\mathcal{C} @ \mathcal{C}'$ is well-formed, so $(W)$ holds.

*Conv-flex:* We have the rule

$$\frac{a \xrightarrow{hnf} Flex(a') \quad b \xrightarrow{hnf} L(b')}{Conv_p\ (a, b, \alpha, \Gamma) \Rightarrow [\langle a', b', \alpha, \Gamma \rangle]}\ \text{Conv-flex}$$

We have $\Gamma\sigma \vdash a\sigma : \alpha\sigma$ and $\Gamma\sigma \vdash b\sigma : \alpha\sigma$ and we need to show

$(W)$ $\Gamma\sigma \vdash a'\sigma : \alpha\sigma$ and $\Gamma\sigma \vdash b'\sigma : \alpha\sigma$

which holds by lemma 6.1 and since if $a$ is reduced as $a \to \cdots \to a'$, then $a\sigma$ is reduced as $a\sigma \to \cdots \to a'\sigma \to \cdots$ as shown in lemma 27.1$(iv)$.

*HConv-head:* Immediate.

*HConv-app:* Follows by induction hypothesis.

$\square$

**Proposition 29**
*If* $Conv_p\ (a, b, \alpha, \Gamma) \Rightarrow \mathcal{C}$, *then*

$(U)\quad \mathcal{U}(a = b : \alpha \quad \Gamma) = \mathcal{U}(\mathcal{C})$.

***Proof:*** By induction on the length of the derivations of $Conv_p\ (a, b, \alpha, \Gamma) \Rightarrow \mathcal{C}'$ and $HConv_p\ (a, b, \Gamma) \Rightarrow \langle \mathcal{C}', \alpha \rangle$.

*Conv-id:* Immediate.

*Conv-fun:* By induction hypothesis.

*Conv-rigid:* Consider the rule

$$\frac{a \xrightarrow{hnf} Rigid(a') \quad b \xrightarrow{hnf} Rigid(b') \quad HConv_p\ (a', b', \Gamma) \Rightarrow \langle \mathcal{C}, \alpha' \rangle}{Conv_p\ (a, b, \alpha, \Gamma) \Rightarrow \mathcal{C}}\ \text{Conv-rigid}$$

Assume $\Gamma\sigma \vdash a\sigma = b\sigma : \alpha\sigma$. Then by lemma 27.1($i$) and 6.1 we have that $\Gamma\sigma \vdash a\sigma = a'\sigma : \alpha\sigma$, and similarly for $b$. Hence by transitivity we have $\Gamma\sigma \vdash a'\sigma = b'\sigma : \alpha\sigma$, and by induction hypothesis we have $\mathcal{U}(a' = b' : \alpha \quad \Gamma = \mathcal{U}(\mathcal{C})$.

*Conv-flex:* Analogous to the rigid case, by using case ($iv$) instead of case ($i$) of lemma 27.1.

*HConv-head:* Immediate.

*HConv-app:* Follows by induction hypothesis.

□

### 6.3.4 The main result

Due to the compositional nature of the type checking algorithm, we get the main results simply by composing the corresponding result of the included algorithms.

**Theorem 3**
*Let* $\mathcal{C}$ *be a well-formed term-TUP such that* $\mathcal{C}$ *ensures* $\Gamma \vdash \alpha : Type$. *Assume* $TC_p\ (\mathcal{C}, e, \alpha, \Gamma) \Rightarrow \mathcal{C}'$. *If* $e$ *is a refinement term relative* $\mathcal{C}$, *then we have*

$(W)\quad \mathcal{C} \mathbin{@} \mathcal{C}'$ *is well-formed,*

(*U*) $\mathcal{U}(\mathcal{C} \mathbin{@} \mathcal{C}') = \mathcal{U}(\langle e, \alpha, \Gamma \rangle)$, *and*
(*C*) $TC_p$ *commutes with the application of a refinement instantiation.*

*In general, if $e$ is not necessarily a refinement term, we only have the preservation of well-formedness and the weaker properties of (U) and (C) corresponding to the soundness direction*

(*W*) $\mathcal{C} \mathbin{@} \mathcal{C}'$ *is well-formed,*
(*U*) $\mathcal{U}(\mathcal{C} \mathbin{@} \mathcal{C}') \subseteq \mathcal{U}(\langle e, \alpha, \Gamma \rangle)$
(*C*) *if* $TC_p\ (\mathcal{C}\sigma, e\sigma, \alpha\sigma, \Gamma\sigma) \Rightarrow \mathcal{C}''$, *then* $\mathcal{U}(\mathcal{C}\sigma \mathbin{@} \mathcal{C}'') \subseteq \mathcal{U}((\mathcal{C} \mathbin{@} \mathcal{C}')\sigma)$

***Proof:*** We have that $TC_p = Simple_p \circ TSimple_p \circ GTE_p$, and that each algorithms preserves the three properties $(W)$, $(U)$ and $(C)$ by propositions 16, 17, 18, 19, 20, 24, 25 and 26. □

# Chapter 7

# Unification

The unification we are interested in is one which finds instantiations to the placeholders in our typed unification problem. It is a higher order unification problem, since the placeholders can be of function type. We also have dependent types, so the types of the placeholders may be effected by instantiations. There are complete unification algorithms for the $\lambda\Pi$-calculus, (see [Ell89], [Pym92]), which are generalisations of the unification algorithm for simply typed $\lambda$-calculus in [Hue75]. However, these algorithms do not apply in our case for the following reasons:

1. These algorithms rely on that flexible-flexible pairs are always solvable. However, in our formulation, we have to handle functions defined by pattern matching, which implies that flexible-flexible pairs can also be of the form

   $$g(a_1, \ldots, a_n) = f(b_1, \ldots, b_k)$$

   where $f$ and $g$ are any functions defined by pattern matching. The pattern matching can not take place before the main argument is instantiated to a term on constructor form, since we would not know which pattern to choose. Therefore, this is also a flexible term. The solvability of such equations in general is undecidable, so we can only hope for an algorithm which leaves difficult equations as constraints and gives a partial solution and a new unification problem.

2. The algorithms require the unknowns in the unification problem to be well-typed in a context. Thus, the equations depend on the types of the unknowns, but not the converse. As we have seen, the type of a placeholder may depend on equations.

In [Dow93], a semi-decision unification algorithm for the type systems of Barendregt's cube is suggested. For these type systems, flexible-flexible pairs are not always solvable either since there exists empty types. It is questioned whether open unification, that is when the unification instantiations may contain unknowns, is of any use since it may be impossible to find (ground) instantiations to these remaining unknowns.

Here we will present an open unification algorithm. However, we will not even attempt to solve these difficult flexible-flexible pairs. The purpose of our unification algorithm will be to find partial solutions to the unification problem. It will transform a tuple of the unification problem $\mathcal{C}$ and a set of solved equations $\mathcal{S}$ found so far, into a new tuple where possibly the set of solved equations has increased. Solved equations are always of the form

$$?_n = e : \alpha_n \quad \Gamma_n$$

where $\alpha_n$ and $\Gamma_n$ is the expected type and the local context of $?_n$. Furthermore, $?_n$ does not occur in either $e$, $\alpha_n$ or $\Gamma_n$, or elsewhere in the unification problem. The instantiation corresponding to the solved equations is then

$$\theta_{\mathcal{S}} = \{?_n = e \mid (?_n = e : \alpha_n \quad \Gamma_n) \in \mathcal{S}\}.$$

First, we have no solved equations so the starting point will be

$$\langle \mathcal{C}, \{\,\} \rangle.$$

A completely solved unification problem is then of the form

$$\langle [\,], \mathcal{S} \rangle$$

where all terms, types and contexts in the solved equations are complete (contains no placeholders). What we will show in section 7.4.1 is that when the unification problem is transformed by the unification rule, that is

$$\langle \mathcal{C}_1, \{\,\} \rangle \stackrel{Unify}{\longrightarrow} \langle \mathcal{C}_2, \mathcal{S}_2 \rangle \stackrel{Unify}{\longrightarrow} \cdots \stackrel{Unify}{\longrightarrow} \langle \mathcal{C}_n, \mathcal{S}_n \rangle$$

then for any solution $\sigma$ of $\mathcal{C}_n$, the instantiation of $\mathcal{S}_n$ composed with $\sigma$ is a solution to the original unification problem $\mathcal{C}_1$. Hence, if $\mathcal{C}_n$ is empty then $\theta_{\mathcal{S}n}$ is a complete solution to $\mathcal{C}_1$.

After defining the unification algorithm, we will see how it is used to improve the type checking algorithm for incomplete terms. The algorithm is applied to the unification problem produced by the type checking algorithm in previous chapter. We will show a soundness result for type checking with unification in section 8.3 and conclude the chapter with some discussions about completeness of this algorithm.

## 7.1 Problems

The unification algorithm will take as input a term-TUP, which is simplified as far as possible, and should output a new term-TUP together with a partial solution. A partial solution will be a set of *solved equations* of the form

$$?_n = e : \alpha_n \quad \Gamma_n$$

where $\alpha_n$ is the expected type and $\Gamma_n$ the expected context of $?_n$. The partial solution will contain the placeholders which were given assignments during unification. An instantiation can easily be extracted from these solved equations.

The solved equations are successively built from "simple" equations, i.e. equations on the form

$$?_n = e : \beta \quad \Delta$$

where the only difference from the solved equations above is that we do not know if $\beta$ is the same as the expected type of $?_n$. This is a problem since then we can not be sure that $e$ is a partial solution to $?_n$.

We will require the unification problem to be well-formed. If there is a solved equation, we know the TUP must be of the following form, since placeholders must be declared before they occur:

$$\begin{bmatrix} & \vdots \\ (1) & ?_n : \alpha_n \quad \Gamma_n \\ & \vdots \\ (2) & ?_n = e : \beta \quad \Delta \\ & \vdots \end{bmatrix}$$

However, we need some requirements on the simple equation (2):

$$?_n = e : \beta \quad \Delta.$$

Usually in unification there is an occur check, which in this case would correspond to that $?_n$ does not occur in $e$. However, we will see that we must generalise the occur check because we also have to check that $e$ is of the expected type and within the expected scope of $?_n$.

The problem is that we only know from well-formedness of the unification problem that

$$\Delta\sigma \vdash ?_n\sigma : \beta\sigma$$

for some complete instantiation $\sigma$ such that the equations and placeholder declarations prior to (2) hold. Since the instantiation $\sigma$ must be type correct, we also know from (1)

$$\Gamma_n\sigma \vdash ?_n\sigma : \alpha_n\sigma.$$

These two judgements alone, do not imply that $\beta\sigma$ and $\alpha_n\sigma$ are the same type, since we do not have uniqueness of types. At this point, it is not clear whether we always have

$$(*)\ \Gamma_n\sigma \vdash \beta\sigma = \alpha_n\sigma\ :\ \mathit{Type}$$

and we can not be sure that

$$\Gamma_n \preceq \Delta$$

and hence we need to type check $e$ relative the expected type $\alpha_n$ and context $\Gamma_n$.

We believe that it is possible to show that $(*)$ holds when the unification problem comes from a type checking problem with new and distinct placeholders. This matter will be discussed further in relation with the completeness conjecture. Even if the conjecture is true, we will have to check the scope of the unified term since the simple equation may have a larger context then the placeholder itself and therefore the unified term may contain variables "out of scope". Consider the following example, where we assume we have a relation $R$ on some set $A$

$R : (A; A)\mathit{Set}$, which is reflexive:
$\mathit{refl} : (x{:}A)R(x, x)$

and we will try to show from the above assumptions:

$$\exists x.\forall y.R(x, y)$$

(This should not be possible since the statement is false if $A$ contains more than one element.) The statement can be represented by the unification problem

$$\left[\begin{array}{c} ?x : A\ \ [\,] \\ ?_1 : R(?x, y)\ \ [y : A] \end{array}\right]$$

Now, if we try to solve the problem by using the reflexivity rule, we need to type check

$$\mathit{refl}(?_2) \overset{?}{:} R(?x, y)\ \ [y : A]$$

which yields the new problem

$$\left[\begin{array}{c} ?x : A\ \ [\,] \\ ?_2 : A\ \ [y : A] \\ R(?x, y) = R(?_2, ?_2)\ \ [y : A] \end{array}\right] \overset{\mathit{Unify}}{\longrightarrow} \left\{\begin{array}{c} ?x = y \\ ?_2 = y \end{array}\right\} \leftarrow \text{out of scope}$$

Here we can see that all equations are well-typed, but we need to check that the unified term is within its scope, which is not the case for $?x = y$, since $y$ is not a variable in $?x$'s local context.

Since we need to type check the instantiation suggested by unification, we must impose a stronger non-circularity requirement. This is due to the precondition of the type checking algorithm in section 6, which requires that the input type and context are correct relative some smaller unification problem. If we remove the non-circularity requirement, we may get a placeholder declaration

$$?_n : \alpha_n \quad \Gamma_n$$

where $?_n$ occurs in $\alpha_n$ or $\Gamma_n$, which is clearly circular.

To summarise, there are two conditions which must be checked for the unified term

1. the term must be of the expected type, and
2. the term must be within its expected scope.

Therefore, we must type check the unified term, and the type checking requires a notion of well-formedness on the unification problem in order to justify the precondition.

## 7.2 Towards a unification algorithm

The first attempt towards a unification algorithm, is an algorithm which is either too restrictive, or depends on a condition which is undecidable. It depends on a definition of simple constraints and this condition is not feasible to check. The algorithm presented after this, corresponds to the representation of unification problems and the algorithm which was previously used in ALF (described in [Mag93]). We include it as a motivation for the optimisation to the current representation and algorithm presented in the next section.

### 7.2.1 Unification algorithm - first attempt

A partially solved unification problem will be a tuple

$$\langle \mathcal{C}, \mathcal{S} \rangle$$

where $\mathcal{C}$ is a unification problem and $\mathcal{S}$ a set of solved equations.

**Definition 7.1** The set of unifiers for a partially solved unification problem $\langle \mathcal{C}, \mathcal{S} \rangle$ is defined by

$$\mathcal{U}\langle \mathcal{C}, \mathcal{S} \rangle = \{\theta_{\mathcal{S}} \circ \sigma \mid \mathcal{C}\sigma \text{ holds}\}.$$

We could define the unification rule as follows:

$$\langle \mathcal{C}, \mathcal{S}\rangle \overset{Unify}{\longrightarrow} \langle Simple_p(\mathcal{C}\{?_n = e\}), \mathcal{S}\{?_n = e\} \cup \{?_n = e : \alpha_n \quad \Gamma_n\}\rangle$$

where $?_n = e$ is a simple constraint in $\mathcal{C}$
and $?_n : \alpha_n \quad \Gamma_n \in \mathcal{C}$.

Recall that when an instantiation is applied to a term TUP, the assigned placeholder declaration will be replaced by the corresponding type checking equations, which means that $e$ is checked to be of the expected type. Then the unification algorithm is simply to apply the rule until there are no more simple constraints. We will denote the many step relation by $\overset{Unify}{\longrightarrow}_*$.

The definition of a simple constraint corresponding to the above unification is defined as:

$?_n = e : \alpha \quad \Gamma$ is a simple constraint in $\mathcal{C}$ if there is a $\mathcal{C}^*$ such that

$(i)$ $\mathcal{C}^*$ is a well-formed permutation of $\mathcal{C}$
$(ii)$ $\mathcal{C}^* = \mathcal{C}_1 \mathbin{@} [?_n : \alpha_n \quad \Gamma_n] \mathbin{@} \mathcal{C}_2 \mathbin{@} [?_n = e : \alpha \quad \Gamma] \mathbin{@} \mathcal{C}_3$
and $PH(e) \subseteq PH(\mathcal{C}_1)$

The last restriction is the non-circularity restriction, since the placeholders declared before $?_n$ in $\mathcal{C}^*$ can not depend on $?_n$ in any way (neither in the type nor in the context).

The reason we need to talk about any well-formed permutation of $\mathcal{C}$ in the definition of simple constraints is that the well-formed condition corresponds to a total order of the placeholders declaration and the equations, which may be too restrictive. The actual dependency order is normally only a partial order. Hence, we do not want to restrict unification instantiations satisfying the non-circularity condition on the given total order, but on any well-formed order $\mathcal{C}^*$. For instance, consider the example where we have sequences of a certain length ($Seq(n)$) with the constructors

$atom : (a{:}N)Seq(1)$, and
$cons : (a{:}N; n{:}N; Seq(n))Seq(s(n))$

and some property $P$ over sequences of a given length

$P : (n{:}N; Seq(n))Set$

and we want to type check

$$P(?_m, cons(0, ?_n, ?_s)) \overset{?}{:} Set.$$

The unification problem we get is

$$\begin{bmatrix} ?_m : N \\ ?_n : N \\ ?_s : Seq(?_n) \\ ?_m = s(?_n) : N \end{bmatrix}$$

Here the non-circularity condition is not satisfied, but clearly we can swap the declarations of $?_m$ and $?_n$ without problem, since they do not depend on each other.

However, the definition above is not practical for an implementation, due to the "any well-formed permutation" part. Since in the general case, we could have a unification problem

$$\begin{bmatrix} \vdots \\ ?_1 : \alpha_1 \quad \Gamma_1 \\ \mathcal{C} \\ ?_n : \alpha_n \quad \Gamma_n \\ \vdots \\ ?_1 = f(?_n) : \alpha \quad \Gamma \\ \vdots \end{bmatrix}$$

which means that we would have to move the declaration of $?_n$ before the declaration of $?_1$. Then $\alpha_n$ and $\Gamma_n$ may not depend on $?_1$ which can be checked, but it may not depend on any equation or placeholder in $\mathcal{C}$ either, which is not allowed to be moved before the declaration of $?_1$. This requirement is much more difficult to check.

Therefore, we will give an alternative representation of the term-TUP, where we separate the placeholders declarations and the equations. Then we will define a strict partial order on the placeholders, and make sure that this order is preserved during unification. We will also present a different definition of the well-formed condition, a new definition of simple constraints and a revised unification algorithm.

### 7.2.2 A possible algorithm

The idea is that we only record the dependencies between the placeholder declarations (i.e. placeholders) and ignore the intervened order of the equations. Then we must strengthen the well-formed requirement to be relative the "smaller" placeholders and *all* equations, to account for the lack of order of the equations.

The order on the placeholders corresponds to the dependencies between the placeholder declarations. The order is more precisely between placeholder declarations, but since every placeholder has a unique declaration, we will define the order as follows:

**Definition 7.2** The placeholder $?_k$ depends on (is smaller than) placeholder $?_n$ (denoted by $<$) is inductively defined by

$$\frac{?_k \in PH(\alpha_n) \cup PH(\Gamma_n)}{?_k <?_n} \qquad \frac{?_k <?_m \quad ?_m \in PH(\alpha_n) \cup PH(\Gamma_n)}{?_k <?_n}$$

where $PH(e)$ is the set of placeholders occuring in the term $e$ and $PH(\alpha)$ and $PH(\Gamma)$ are the obvious extensions to types and contexts, respectively.

However, this order is not sufficient for the soundness proof of unification. The problem is that if a set of placeholder declarations is ordered by $<$, and we perform an instantiation to these declarations, we could have that $?_k <?_p$ in $\mathcal{P}$, but $?_k \not<?_p$ in $\mathcal{P}\{?_n = e\}$, even if $?_k, ?_p$ and $?_n$ are different placeholders. The reason is that we can have

$$\left[\begin{array}{c} ?_k : \alpha_k \quad \Gamma_k \\ ?_n : \alpha_n(?_k) \quad \Gamma_n \\ ?_p : \alpha_p(?_n) \quad \Gamma_p \end{array}\right] \{?_n = e\} \Rightarrow \left[\begin{array}{c} ?_k : \alpha_k \quad \Gamma_k \\ \mathcal{E} \\ ?_p : \alpha_p(e)) \quad \Gamma_p \end{array}\right]$$

where $\mathcal{E}$ is the set of equations we get from type checking $e$ to be of type $\alpha_n(?_k)$ in $\Gamma_n$. Since the order ignores the equations, we have lost some dependency information. In the first set we have that $?_k <?_n <?_p$, but in the new set $?_p$ only depends on the placeholders occuring in $e$. If $?_k$ is not among these, we have lost the information that $?_p$ depends on $?_k$. Therefore, we will also keep a *dependency graph* which keeps track of all the dependencies between the placeholders.

Another solution to the problem is to not perform the instantiation of $?_n$ to $\mathcal{P}$, but rather "update" $\mathcal{P}$ as:

$$\left[\begin{array}{c} ?_k : \alpha_k \quad \Gamma_k \\ ?_n = e : \alpha_n(?_k) \quad \Gamma_n \\ ?_p : \alpha_p(?_n) \quad \Gamma_p \end{array}\right]$$

and to change the order to also consider the assignment, if the placeholder is updated. Here it becomes rather clear that the dependency order is conservative, and the updating of a placeholder simply makes it dependent on possibly more placeholders. Seen as a dependency graph, the updating means simply adding some arcs from the updated placeholder. The drawback of this solution is that we keep all placeholders, even if they are assigned an instantiation, and to get the "real" set of placeholder declarations, we need to unfold all updated placeholders. This was how unification problems in ALF used to be represented, until we realized it could be optimised.

## 7.3 The unification algorithm

The representation of a unification problem is a triple

$$\langle \mathcal{P}, \mathcal{E}, \mathcal{G} \rangle$$

where $\mathcal{P}$ is a set of placeholder declarations, $\mathcal{E}$ a set of equations and $\mathcal{G}$ a dependency graph ordering the placeholders in $\mathcal{P}$. The set of unifiers is then defined analogously as for the term-TUP:

**Definition 7.3** The set of unifiers for a partially solved unification problem $\mathcal{U}\langle\langle \mathcal{P}, \mathcal{E}, \mathcal{G} \rangle, \mathcal{S}\rangle$ is defined by

$$\mathcal{U}\langle\langle \mathcal{P}, \mathcal{E}, \mathcal{G} \rangle, \mathcal{S}\rangle = \{\theta_{\mathcal{S}} \circ \sigma \mid \langle \mathcal{P}, \mathcal{E} \rangle\sigma \text{ holds}\}$$

where $\theta_{\mathcal{S}}$ is the instantiation corresponding to the solved equations $\mathcal{S}$, and

$$\langle \mathcal{P}, \mathcal{E} \rangle\sigma \text{ holds if } \quad \begin{cases} \Gamma_n\sigma \vdash ?_n\sigma : \alpha_n\sigma & \forall(?_n : \alpha_n \;\; \Gamma_n) \in \mathcal{P} \\ \Gamma\sigma \vdash a\sigma = b\sigma : \alpha\sigma & \forall(a = b : \alpha \;\; \Gamma) \in \mathcal{E}. \end{cases}$$

We will represent the dependency graph as a set of pairs $\langle ?_m, M\rangle$, where $?_m$ is a placeholder and $M$ is a set of placeholders. We let $M$ be the set of *all* placeholders that $?_m$ depends on which means that $M$ will be the transitive closure of the dependency order.

**Definition 7.4** We will call a graph *transitive* if whenever there is an arc from node $n$ to node $m$ and also an arc from $m$ to node $p$, then there is an arc from $n$ to $p$.

We will introduce some notation concerning dependencies in the graph, where $DO_{\mathcal{G}}$ is an (overloaded) function which gives the set of placeholders the argument depends on. In the latter, the argument is itself a set of placeholders.

$$DO_{\mathcal{G}}(?_k) = K \text{ where } \langle ?_k, K\rangle \in \mathcal{G}$$
$$DO_{\mathcal{G}}(K) = \bigcup_{?_k \in K} DO(?_k)$$

We often need to check the non-circularity condition, that is we need to check that the graph is acyclic. The following remark motivates the choice of representation:

**Remark.** The transitive graph $\mathcal{G}$ is acyclic if and only if $\forall ?_m \in \mathcal{G}.\ ?_m \notin DO_{\mathcal{G}}(?_m)$.

We will also define some operations on a graph:

**Definition 7.5** A transitive graph $\mathcal{G}$ can be updated with a set of nodes $N$ replacing the node $?_n$ by removing the node $?_n$ from the graph and replacing any occurrence of $?_n$ in the remaining graph by $N$:

$$Update(\mathcal{G}, ?_n, N) = \{\langle ?_k, K[?_n/N]\rangle \mid \langle ?_k, K\rangle \in \mathcal{G}, ?_k \neq ?_n\}$$

We also define how two transitive graphs can be merged, where the graph $\mathcal{G}'$ may depend on nodes in the graph $\mathcal{G}$ but not vice versa:

$$Merge(\mathcal{G}, \mathcal{G}') = \mathcal{G} \cup \{\langle ?_k, K[?_n/DO_{\mathcal{G}}(?_n) \cup \{?_n\} \mid \langle ?_k, K\rangle \in \mathcal{G}'\}$$

The merging operation thus leaves the $\mathcal{G}$ intact and adds arcs from the graph $\mathcal{G}'$ into the nodes in $\mathcal{G}$.

The updating operation is used when a placeholder is instantiated by unification, since then no new nodes will be added to the graph (no new placeholders is introduced by unification). We will see that the unification algorithm will exactly allow the unification instantiations which guarantee that the updated graph remains acyclic if the original graph is. The merging operation is used when a placeholder is instantiated by a refinement term containing new placeholders. If both graphs are acyclic to start with, the merging operation will preserve acyclicity, since arcs can only be added from one graph into the other. We have the following properties of the graph operations:

**Proposition 30** *Let $\mathcal{G}$ be a transitive, acyclic graph. Let $M$ be the set of nodes of a transitive sub-graph of $\mathcal{G}$. Then we have that*

*$(i)$ if $m \notin M$ then $Update(\mathcal{G}, m, M)$ is transitive and acyclic.*

*Furthermore, let $\mathcal{G}'$ be another transitive, acyclic graph which may depend on the nodes in $\mathcal{G}$ but where the nodes of $\mathcal{G}$ do not depend on $\mathcal{G}'$, then we have that*

*$(ii)$ $Merge(\mathcal{G}, \mathcal{G}')$ is a transitive, acyclic graph.*

***Proof:*** $(i)$ : Since $M$ contains all nodes in a transitive sub-graph, when we replace the node $m$ by $M$ everywhere this operation corresponds to replcing the node $m$ by a transitive graph, so the result is clearly transitive. Moreover, when we replace $m$ by $M$ everywhere we could only create a cycle if any node in $M$ depends on $m$, that is for some $k \in M$, we have $\langle k, K \cup \{m\}\rangle$. But since

$M$ is a transitive sub-graph of $\mathcal{G}$, we have that if $k \in M$ and $k$ depends on $m$, then $m$ is also in $M$. Hence, since $m \notin M$ we know that $Update(\mathcal{G}, m, M)$ is acyclic.

$(ii)$ : We have that $\mathcal{G}$ does not depend on $\mathcal{G}'$ and in $\mathcal{G}'$ we add the transitive closure of any node in $\mathcal{G}$, so $Merge(\mathcal{G}, \mathcal{G}')$ remains transitive. Moreover, since $\mathcal{G}$ and $\mathcal{G}'$ are both acyclic, and we only add new arcs from $\mathcal{G}'$ into $\mathcal{G}$, the resulting graph is also acyclic.

$\Box$

Before we present the algorithm, we need to define what it means to apply an instantiation to the new representation of a term TUP. The idea is the following: remove the placeholder declaration of the instantiated placeholder from the others and apply the instantiation to all these. The only thing that will happen is that the types and context of the remaining placeholders are instantiated. Instantiate also the equations and update the graph. Finally, we will type check the instantiation. The type checking will only produce new equations since the unified term only contains known placeholders. The equations can simply be added.

**Definition 7.6** A unification problem $\langle \mathcal{P}, \mathcal{E}, \mathcal{G} \rangle$ applied to a unification instantiation $\{?_k = b\}$ is defined by

$$\langle \mathcal{P}, \mathcal{E}, \mathcal{G} \rangle \{?_k = b\} = \langle \mathcal{P}', \mathcal{E}', \mathcal{G}' \rangle$$

where

$$\begin{array}{ll} \mathcal{P}' & = (\mathcal{P} - \{?_k : \alpha_k \quad \Gamma_k\})\{?_k = b\} \\ \mathcal{E}' & = \mathcal{E}\{?_k = b\} \cup TC_p(\langle \mathcal{P}, \mathcal{E}, \mathcal{G} \rangle, b, \alpha_k, \Gamma_k) \\ \mathcal{G}' & = Update(\mathcal{G}, ?_k, PH(b) \cup DO_{\mathcal{G}}(PH(b))) \end{array}$$

The definition can be extended to a general instantiation $\theta$:

$$\langle \mathcal{P}, \mathcal{E}, \mathcal{G} \rangle (\{?_k = b\} \cup \theta) = (\langle \mathcal{P}, \mathcal{E}, \mathcal{G} \rangle \{?_k = b\})\theta.$$

Recall what happens when we apply an instantiation to our previous representation. For placeholder declarations we have two cases, if the placeholder is assigned in the instantiation, we replace it by the result of type checking the assignement and otherwise we instantiate the type and the context. For equations, the instantiation is simply applied. It should be rather clear that this operation has the same effect on placeholder declarations and equations as the corresponding definition 6.7.

Finally we have the modified unification algorithm:

**Unification algorithm**

$$\langle\langle \mathcal{P}, \mathcal{E}, \mathcal{G}\rangle, \mathcal{S}\rangle \stackrel{Unify}{\longrightarrow} \langle\langle Simple_p(\langle \mathcal{P}, \mathcal{E}, \mathcal{G}\rangle\{?_n = e\})\rangle, \mathcal{S}'\rangle$$

where $\mathcal{S}' = \mathcal{S}\{?_n = e\} \cup \{?_n = e : \alpha_n \quad \Gamma_n\}$
and $?_n = e$ is a simple constraint in $\langle \mathcal{P}, \mathcal{E}, \mathcal{G}\rangle$.

Since simplification only effects equations, we have that

$$Simple_p\langle \mathcal{P}, \mathcal{E}, \mathcal{G}\rangle = \langle \mathcal{P}, Simple_p(\mathcal{E}), \mathcal{G}\rangle.$$

Even though the solved equations are technically equations, it is really their corresponding instantiations we are interested in. We will see that the operation performed on $\mathcal{S}$ in the unification is exactly composition of the corresponding instantiations.

We will define some simple operations and properties about instantiations, which mainly are the usual operation on substituions, but since we have separated the notion of variable and placeholder, we will state them for clarity. Instantiations can be combined in two ways, either the simple application of an instantiation to the assigned terms in another instantiation (denoted by $\theta\sigma$) or the composition of two instantiations ($\theta\circ\sigma$), where the resulting instantiation is extended with the assigned terms in $\sigma$ if they were not already assigned in $\theta$. In both cases, we must know that the terms in $\sigma$ contain no placeholder in the domain of $\theta$, since this would violate the condition that instantiations are independent assignments of the placeholders.

**Definition 7.7** Let $\sigma$ and $\theta = \{?_1 = a_1, \ldots, ?_n = a_n\}$ be instantiations, where the terms in $\sigma$ do not contain the placeholders $?_1, \ldots, ?_n$. Then we define the application of $\sigma$ to $\theta$ by

$$\{?_1 = a_1, \ldots, ?_n = a_n\}\sigma = \{?_1 = a_1\sigma, \ldots, ?_n = a_n\sigma\}$$

and the *composition of instantiations* by

$$\theta\circ\sigma = \theta\sigma \cup (\sigma - Dom(\theta))$$

where $\sigma - Dom(\theta)$ denotes $\sigma$ restricted to the placeholders distinct from $Dom(\theta)$.

With this restriction on the terms in $\sigma$ it is clear that the result is an instantiation, because the $a_i\sigma$'s can only contain placeholders already in $a_i$ or placeholders from the terms in $\sigma$.

**Proposition 31** *Let $e, \alpha$ and $\Gamma$ be an incomplete term, type and context, respectively. If $\sigma$, $\theta$ and $\rho$ are instantiations, then*

$(i)$ $(e\sigma)\theta = e(\sigma\circ\theta)$
$(ii)$ $(\sigma\circ\theta)\circ\rho = \sigma\circ(\theta\circ\gamma)$

*Clearly $(i)$ implies that $(\alpha\sigma)\theta = \alpha(\sigma\circ\theta)$ and $(\Gamma\sigma)\theta = \Gamma(\sigma\circ\theta)$*

So if we denote the instantiation of $\mathcal{S}$ by $\theta_S$, then the instantiation of $\mathcal{S}\{?_n = e\}\cup\{?_n = e : \alpha_n \quad \Gamma_n\}$ is exactly $\theta_S\{?_n = e\}\cup\{?_n = e\}$, or in other words, $\theta_S\circ\{?_n = e\}$. Moreover, since the solved equations are unfolded in the equations, and the new simple equation comes from there, we know that the placeholders in $e$ can not be among the solved ones, nor $?_n$ (due to the occur check). Hence we know that the solved equations always correspond to valid instantiations.

The reason we wanted to modify the representation of a unification problem was to get an algorithmic definition of a simple constraint. The new notion of a simple constraint becomes easy to check, since we can simply look up in the dependency graph which placeholders a placeholder depends on.

**Definition 7.8** A constraint $?_n = e : \alpha \quad \Gamma \in \mathcal{E}$ is *simple* relative $\langle\mathcal{P}, \mathcal{E}, \mathcal{G}\rangle$ if

$$?_n \notin PH(e) \cup DO_{\mathcal{G}}(PH(e))$$

The intuition is that $?_n$ can not occur in $e$ corresponds to the occur check. That the placeholders in $e$ can not depend on $?_n$ (in any way) is due to the non-circularity requirement of the placeholder declarations since such a dependency would create a cyclic graph.

Finally, we can give the revised definition of a well-formed, (partially solved) unification problem:

**Definition 7.9** The unification problem $\langle\langle\mathcal{P}, \mathcal{E}, \mathcal{G}\rangle, \mathcal{S}\rangle$ is *well-formed* if

$(i)$ $\mathcal{G}$ is acyclic
$(ii)$ For all $?_n$ in $\mathcal{P}$, $\mathcal{E}$ and $\mathcal{P}|_{DO(?_n)}$ ensures $\Gamma_n \vdash \alpha_n : Type$
$(iii)$ $\langle\mathcal{P}, \mathcal{E}\rangle$ ensures $\mathcal{S}$

where $\mathcal{P}|_{DO(?_n)}$ denotes $\mathcal{P}$ restricted to the placeholders that $?_n$ depends on according to the graph $\mathcal{G}$.

The idea behind this reformulation of well-formedness is that since we no longer have an order on the equations, we must consider all equations, but we only require that the instantiation is type correct for the "smaller" placeholders. So if a unification problem is well-formed, we know that the set of placeholder declarations is non-circular so we can always take the subset of $\mathcal{P}$ which a placeholder depends on. The second condition says that if we consider an instantiation which satisfies the equations and which is type correct for the placeholders that $?_n$ depends on, then we know that the expected type and context of $?_n$ are also correct. This property is exactly what we need in order to satisfy the precondition of type checking, which we use to check the type correctness of a unified term. The last condition states that for any solution of the remaining placeholders, the partially solved equations are type correct.

Before we define the type checking algorithm with unification, we will describe how the result of type checking is converted into our new representation. The placeholder declarations and the equations are simply separated, and the dependency graph is computed from the placeholders declarations. Here we will use the order $<$ defined on page 142, to initialise the graph:

**Definition 7.10** A term-TUP is converted to a triple $\langle \mathcal{P}, \mathcal{E}, \mathcal{G} \rangle$ and an empty set of solved equations by

$$Convert(\mathcal{C}) = \langle \langle \mathcal{P}, \mathcal{E}, \mathcal{G} \rangle, \{\} \rangle$$

where

$$\begin{array}{ll} \mathcal{P} & = \{?_n : \alpha_n \quad \Gamma_n \mid (?_n : \alpha_n \quad \Gamma_n) \in \mathcal{C}\} \\ \mathcal{E} & = \{a = b : \alpha \quad \Gamma \mid (a = b : \alpha \quad \Gamma) \in \mathcal{C}\} \\ \mathcal{G} & = \{\langle ?_n, \{?_k \mid ?_k < ?_n\} \rangle \mid ?_n \in \mathcal{P}\} \end{array}$$

Recall the definitions of unifers for the two representations

$\mathcal{U}\langle \mathcal{C}, \mathcal{S} \rangle = \{\theta_\mathcal{S} \circ \sigma \mid \mathcal{C}\sigma \text{ holds}\}$, and correspondingly
$\mathcal{U}\langle \langle \mathcal{P}, \mathcal{E}, \mathcal{G} \rangle, \mathcal{S} \rangle = \{\theta_\mathcal{S} \circ \sigma \mid \langle \mathcal{P}, \mathcal{E} \rangle \sigma \text{ holds}\}$.

Now it should be clear that the two representation have the same set of unifers, i.e. we have the following

**Remark.** The set of unifers is the same in both representations, that is

$$\mathcal{U}(\mathcal{C}) = \mathcal{U}(Convert(\mathcal{C}))$$

since $\mathcal{C}$ and $Convert(\mathcal{C})$ contain exactly the same placeholder declarations and equations.

Moreover, it does not matter which representation is used as the first argument to the type checking algorithm, that is the unification problem, since only the placeholder declarations are used by the algorithm. Hence, we have the following corollary of theorem 3, which restates the theorem with the graph representation:

**Corollary 32**
*Let $\langle \mathcal{P}, \mathcal{E}, \mathcal{G} \rangle$ be a well-formed unification problem which ensures $\Gamma \vdash \alpha : Type$. Assume that $TC_p\,(\mathcal{P}, e, \alpha, \Gamma) \Rightarrow \mathcal{C}$, and that $Convert(\mathcal{C}) = \langle \mathcal{P}', \mathcal{E}', \mathcal{G}' \rangle$. If $e$ is a refinement term relative $\mathcal{P}$, then we have*

- $(W)$ *$\langle \mathcal{P}'', \mathcal{E}'', \mathcal{G}'' \rangle$ is well-formed,*
- $(U)$ *$\mathcal{U}\langle \mathcal{P}'', \mathcal{E}'', \mathcal{G}'' \rangle = \mathcal{U}\langle e, \alpha, \Gamma \rangle$, and*
- $(C)$ *$TC_p$ commutes with the application of a refinement instantiation,*

*where $\langle \mathcal{P}'', \mathcal{E}'', \mathcal{G}'' \rangle = \langle \mathcal{P} \cup \mathcal{P}', \mathcal{E} \cup \mathcal{E}', Merge(\mathcal{G}, \mathcal{G}') \rangle$.*

*In general, if $e$ is not necessarily a refinement term, we only have the preservation of well-formedness and the weaker properties of $(U)$ and $(C)$ corresponding to the soundness direction*

- $(W)$ *$\langle \mathcal{P}'', \mathcal{E}'', \mathcal{G}'' \rangle$ is well-formed,*
- $(U)$ *$\mathcal{U}\langle \mathcal{P}'', \mathcal{E}'', \mathcal{G}'' \rangle \subseteq \mathcal{U}\langle e, \alpha, \Gamma \rangle$*
- $(C)$ *any unifier of $\langle \mathcal{P}'', \mathcal{E}'', \mathcal{G}'' \rangle\sigma$ is also a unifier of $\langle e\sigma, \alpha\sigma, \Gamma\sigma \rangle$.*

***Proof:*** We get the $(U)$ and $(C)$ properties directly from the remark above and theorem 3, since the two representations have the same set of solutions. The well-formedness property we get since $\mathcal{C}$ is well-formed relative $\langle \mathcal{P}, \mathcal{E}, \mathcal{G} \rangle$, by theorem 3, and hence $Convert(\mathcal{C})$ is also well-formed relative $\langle \mathcal{P}, \mathcal{E}, \mathcal{G} \rangle$ so $\mathcal{G}'$ is an acyclic graph. Moreover, since $\mathcal{G}'$ only contains nodes of the new placeholders the result of merging $\mathcal{G}$ and $\mathcal{G}'$ is also acyclic. $\Box$

## 7.4 Soundness of unification

In this section we show the soundness of our unification algorithm, that is we will show that unification preserves well-formedness and that any solution to the unified problem is also a solution of the original problem. Hence, these are the corresponding $(W)$ and $(U)$ properties of the unification algorithm which we showed for the type checking in section 6.3.

The main proposition states that any solution to the unified problem, is also a solution to the original problem. It relies on the property that unification preserves well-formedness, which is the the following proposition:

**Proposition 33** *If $\langle \langle \mathcal{P}, \mathcal{E}, \mathcal{G} \rangle, \mathcal{S} \rangle$ is a well-formed unification problem and if*

$$\langle\langle\mathcal{P},\mathcal{E},\mathcal{G}\rangle,\mathcal{S}\rangle \stackrel{Unify}{\longrightarrow} \langle\langle\mathcal{P}',\mathcal{E}',\mathcal{G}'\rangle,\mathcal{S}'\rangle,$$

*then* $\langle\langle\mathcal{P}',\mathcal{E}',\mathcal{G}'\rangle,\mathcal{S}'\rangle$ *is well-formed.*

***Proof:*** Assume $\langle\langle\mathcal{P},\mathcal{E},\mathcal{G}\rangle,\mathcal{S}\rangle$ is a well-formed, that is

$A(i)$ $\mathcal{G}$ is acyclic
$A(ii)$ if $\mathcal{E}\sigma$ holds and $\mathcal{P}|_{DO_{\mathcal{G}}(?_m)}\ \sigma$ holds, then $\Gamma_m\sigma \vdash \alpha_m\sigma : Type$
$A(iii)$ if $\langle\mathcal{P},\mathcal{E}\rangle\sigma$ holds, then $\mathcal{S}\sigma$ is well-typed.

We need to show that $\langle\mathcal{P}',\mathcal{E}',\mathcal{G}'\rangle,\mathcal{S}'\rangle$ is well-formed, where

$$\begin{cases} \mathcal{P}' &= (\mathcal{P} - \{?_k : \alpha_k \quad \Gamma_k\})\{?_k = e\} \\ \mathcal{E}' &= Simple_p(\mathcal{E}\{?_k = e\} \cup TC_p(\langle\mathcal{P},\mathcal{E}\rangle, e, \alpha_k, \Gamma_k)) \\ \mathcal{G}' &= Update(\mathcal{G}, ?_k, DO_{\mathcal{G}}(PH(e)) \cup PH(e)) \\ \mathcal{S}' &= \mathcal{S}\{?_k = e\} \cup ?_k = e : \alpha_k \quad \Gamma_k \end{cases}$$

and where $\{?_k = e\}$ is a simple constraint in $\langle\mathcal{P},\mathcal{E},\mathcal{G}\rangle$. What we must do is to verify the three conditions of well-formedness, i.e. show the following

$(i)$ $Update(\mathcal{G}, ?_k, DO_{\mathcal{G}}(PH(e)) \cup PH(e))$ is acyclic
$(ii)$ if $\mathcal{E}'\sigma'$ holds and $\mathcal{P}'|_{DO_{\mathcal{G}'}(?_m)}\ \sigma'$ holds, then $\Gamma_m\sigma' \vdash \alpha_m\sigma' : Type$
$(iii)$ if $\langle\mathcal{P}',\mathcal{E}'\rangle\sigma$ holds then $\mathcal{S}'\sigma$ is well-typed

for an arbitrary $?_m \in \mathcal{P}'$.

$(i)$ : We know that $\{?_k = e\}$ is simple in $\mathcal{G}$, which means that

$$?_k \notin PH(e) \cup DO_{\mathcal{G}}(PH(e)).$$

Clearly $PH(e) \cup DO_{\mathcal{G}}(PH(e))$ is a transitive sub-graph of $\mathcal{G}$ since $\mathcal{G}$ is transitive. Hence, $\mathcal{G}'$ is acyclic by proposition 30.

$(ii)$ : Let $?_m$ be a placeholder in $\mathcal{P}'$ such that

$(Ass1)$ $\mathcal{E}'\sigma'$ holds, and
$(Ass2)$ $\mathcal{P}'|_{DO_{\mathcal{G}'}(?_m)}\ \sigma'$ holds.

We need to show that

$$\Gamma_m\sigma' \vdash \alpha_m\sigma' : Type,$$

which we will do by using $A(ii)$ with $\sigma = \{?_k = e\}\circ\sigma'$.

We have by $(Ass1)$ that $\mathcal{E}'\sigma'$ holds, which means

$(a)$ $(\mathcal{E}\{?_k = e\})\sigma'$ holds, and
$(b)$ $TC_p(\langle\mathcal{P},\mathcal{E},\mathcal{G}\rangle, e, \alpha_k, \Gamma_k)\sigma'$ holds,

since $Simple_p$ preserves the set of unifiers (proposition 26). We can note that

$$(a) \Rightarrow \mathcal{E}(\{?_k = e\}\circ\sigma') \text{ holds},$$

by proposition 31, so the first condition of $A(ii)$ is satisfied.

For the second condition, we will consider the placeholder declarations $\mathcal{P}$, i.e. we have

$$?_p : \alpha_p \quad \Gamma_p$$

for every placeholder in $\mathcal{P}$. Then we know that in $\mathcal{P}'$ we have

$$?_p : \alpha_p\{?_k = e\} \quad \Gamma_p\{?_k = e\}, \text{ for every } ?_p \neq ?_k.$$

Hence by $(Ass2)$ we have

$$(\Gamma_p\{?_k = e\})\sigma' \vdash ?_p\sigma' : (\alpha_p\{?_k = e\})\sigma'$$

that is

$$\Gamma\sigma \vdash ?_p\sigma : \alpha_p\sigma$$

for all placeholders smaller than $?_m$, except for $?_k$.

We have two cases, either $?_m$ is less than $?_k$, that is $?_k \notin DO_{\mathcal{G}}(?_m)$ and then $DO_{\mathcal{G}}(?_m) = DO_{\mathcal{G}'}(?_m)$. For this case, we get $(ii)$ directly from $A(ii)$.

For the other case, when $?_m$ is greater than (depends on) $?_k$, then $?_m$ is also greater than all placeholders less than $?_k$ since the graph represents the transitive closure of this dependency. Hence, we have

$$DO_{\mathcal{G}'}(?_m) = DO_{\mathcal{G}}(?_m) \cup E - \{?_k\}$$

where $E$ is the set of all placeholders which the term $e$ depends on. So we must verify the type correctness of the instantiation of $?_k$, then we can apply $A(ii)$.

From $(b)$ we get the type correctness of the unified placeholder $?_k$, since the placeholders that $?_k$ depends on have not changed in $\mathcal{P}'$, so we have by $A(ii)$ that the precondition of type checking is fulfilled and we get

$$\Gamma_k\sigma' \vdash e\sigma' : \alpha_k\sigma'$$

by corollary 32. Since $?_k$ can not occur in neither $\Gamma_k$, $e$ nor $\alpha_k$, we can extend the instantiation to include $?_k$, that is

$$\Gamma_k\sigma \vdash e\sigma : \alpha_k\sigma.$$

So finally, we can conclude that for any $?_m \in \mathcal{P}'$, we have

$$\Gamma_m\sigma \vdash \alpha_m\sigma : Type$$

by assumption $A(ii)$ and with $\sigma = \{?_k = e\}\circ\sigma'$.

$(iii)$ : Assume $\langle\mathcal{P}', \mathcal{E}'\rangle\sigma$ hold. Then all placeholders in $\mathcal{P}$ but $?_k$ are well-typed by the instantiation $\{?_k = e\}\circ\sigma$ by assumption. We get the well-typedness of $?_k$ from the type checking equations in $\mathcal{E}'$, just as in $(ii)$. Hence, the added solved equation is well-typed, and the other equations we know are well-typed from $A(iii)$ since it says that $(\mathcal{S}'\{?_k = e\})\sigma$ is well-typed, which is all we needed to show.

□

The remaining propositions are about relating unification problems to each other. For these propositions, we will simply use the notation

$\langle \mathcal{C}, \theta \rangle$

where $\mathcal{C}$ can mean either representation of the unification problem since the proofs only concerns unifiers which coincide for the two representations. We only have to be careful that when we apply an instantiation to $\mathcal{C}$, the well-formedness is preserved. However, if the instantiation comes from unification, then we have just proved that well-formedness is preserved. The other case is when we apply a refinement instantiation, and this is proved in the next section (proposition 35).

First we will define an order between unification problems, which we will denote by $\geq_{\mathcal{U}}$ . Then we will show that if two unification problems are related by $\geq_{\mathcal{U}}$ , their corresponding sets of unifiers are related by $\supseteq$. We introduce this notion now, even though it is not needed until we will show in the next section that the order is preserved under application of a refinement instantiation. The fact that one unification problem has the same unifiers as another unification problem is not sufficient to show this, and the reason is that if we only have

$\mathcal{U}\langle \mathcal{C}, \theta \rangle \supseteq \mathcal{U}\langle \mathcal{C}', \theta' \rangle$

then we only know for some unifier $\rho$ of $\langle \mathcal{C}', \theta' \rangle$ that

$\rho = \theta' \circ \sigma'$ where $\mathcal{C}'\sigma'$ holds implies
$\rho = \theta \circ \sigma$ where $\mathcal{C}\sigma$ holds,

for some $\sigma$ and $\sigma'$. Thus, we have not enough information of how the remaining unification problems and the instantiations relate.

Therefore, we will define the order $\geq_{\mathcal{U}}$ as follows:

**Definition 7.11** We will say that $\langle \mathcal{C}, \theta \rangle \geq_{\mathcal{U}} \langle \mathcal{C}', \theta' \rangle$ if there is a $\theta''$ such that

$(i)$ $\theta' = \theta \circ \theta''$
$(ii)$ $\mathcal{U}(\mathcal{C}\theta'') \supseteq \mathcal{U}(\mathcal{C}')$

The intuition is that when $X \geq_{\mathcal{U}} Y$ then $Y$ is essentially the same unification problem, but it may be more solved than $X$. That $Y$ is more solved is reflected in $(i)$ since the instantiation has (possibly) more assigned placeholders. That

$X$ and $Y$ are essentially the same unification problem is expressed by $(ii)$, since it says that any solution to the problem in $Y$, is a solution to the problem in $X$ if we first assign the placeholders which were assigned in $Y$ but not in $X$.

What we are interested in finally, is that the unified problem gives correct solutions, so we have the following proposition:

**Proposition 34** *If* $\langle \mathcal{C}, \theta\rangle \geq_{\mathcal{U}} \langle \mathcal{C}', \theta'\rangle$, *then* $\mathcal{U}\langle \mathcal{C}, \theta\rangle \supseteq \mathcal{U}\langle \mathcal{C}', \theta'\rangle$.

***Proof:*** By the definition of $\geq_{\mathcal{U}}$ we know that $\theta' = \theta\circ\theta''$, so assume $\sigma$ is a unifier of $\mathcal{U}\langle \mathcal{C}', \theta\circ\theta''\rangle$, that is

$\exists\sigma'.\ \sigma = \theta\circ\theta''\circ\sigma'$ and $\mathcal{C}'\sigma'$ hold.

Then $\sigma'$ is a unifier of $\mathcal{C}'$ and by definition of $\geq_{\mathcal{U}}$ it is then also a unifier of $\mathcal{C}\theta''$. Hence, $\mathcal{C}(\theta''\circ\sigma')$ holds so $\sigma$ is also a unifier of $\mathcal{U}\langle \mathcal{C}, \theta\rangle$. □

### 7.4.1 Main result

Now we can prove the main result of this chapter; the theorem which guarantees that the unification really computes a partial unifier:

**Theorem 4** *If* $\langle \mathcal{C}, \mathcal{S}\rangle$ *is a well-formed unification problem and if*

$$\langle \mathcal{C}, \theta\rangle \xrightarrow{Unify}_{*} \langle \mathcal{C}', \theta'\rangle,$$

*then*

$$\langle \mathcal{C}, \theta\rangle \geq_{\mathcal{U}} \langle \mathcal{C}', \theta'\rangle.$$

***Proof:*** By induction on the number of steps of $\xrightarrow{Unify}$. We have by the definition of $\xrightarrow{Unify}$ that

$$\mathcal{C}_{n+1} = Simple_p(\mathcal{C}_n\theta_n), \text{ and}$$
$$\theta_{n+1} = \theta\circ\theta_n$$

We have to show that $\langle \mathcal{C}_n, \theta\rangle \geq_{\mathcal{U}} \langle \mathcal{C}_{n+1}, \theta\circ\theta_n\rangle$, that is

$$\mathcal{U}(\mathcal{C}_n\theta_n) \supseteq \mathcal{U}(\mathcal{C}_{n+1})$$

which is obvious since $Simple_p$ preserves the set of unifiers. Hence, since unification preserves well-formedness (proposition 33) and $\geq_{\mathcal{U}}$ is transitive by lemma 4.1, we have the result by the induction hypothesis. □

**Lemma 4.1** *The order* $\geq_{\mathcal{U}}$ *is transitive.*

***Proof:*** By associativity of composition and lemma 4.2. □

**Lemma 4.2** *Let $\mathcal{C}$ be a well-formed unification problem, $\theta$ a unification instantiation and $\sigma$ any instantiation. Then*

$$(\mathcal{C}\theta)\sigma \text{ ensures } \mathcal{C}(\theta \circ \sigma)$$

***Proof:*** The reason we only have "ensures" and not equivalent is that since $\theta$ is a unification instantiation, so the type checking performed in $(\mathcal{C}\theta)$ only give us this direction. Follows by theorem 3 and proposition 31. $\square$

# Chapter 8

# Applying unification to type checking

In this chapter we will describe how the unification algorithm in the previous chapter is used in combination with the type checking algorithm for incomplete terms. The purpose of the unification is to try to instantiate as many of the placeholders in the type checked term as possible. There is no search involved, the unification instantiates the placeholders which have only one choice, and all real choices are left to the user.

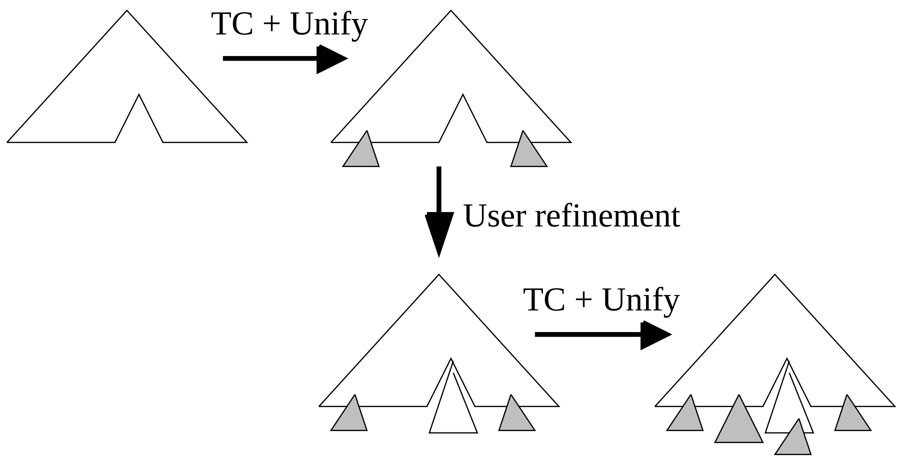



Figure 8.1: User guided search

The application of this algorithm we will describe here, is in interactive proof development. However, it could also be used as the starting point of a more automated search, by applying search strategies to the unification problem.

Now we can define the type checking algorithm (with unification) for incomplete terms, which simply applies the unification algorithm to the result of the type checking algorithm:

$$TC_pU = \overset{Unify}{\longrightarrow}_* \circ\ TC_p.$$

The corresponding soundness property that we will prove later in section 8.3, is that the unification computes correct instantiations to the type checking problem:

$$\text{if } TC_pU\ ([\,], e, \alpha, \Gamma) \Rightarrow \langle\langle \mathcal{P}, \mathcal{E}, \mathcal{G}\rangle, \mathcal{S}\rangle \text{ then } \langle \mathcal{P}, \mathcal{E}\rangle \text{ ensures } \Gamma\theta_{\mathcal{S}} \vdash e\theta_{\mathcal{S}} : \alpha\theta_{\mathcal{S}}$$

where $\theta_{\mathcal{S}}$ is the instantiation of $\mathcal{S}$. In particular, if $\mathcal{P}$ and $\mathcal{E}$ are empty, then we know that the unification found type correct instantiations to all the placeholders in the term.

Just as for the type checking without unification, we want to apply our optimisation of localising the type checking. That is, we want to apply the user instantiation directly to the unified problem, and not to the entire term, and then try to unify again. For the unification, we do not have that a refinement instantiation commutes with the unification, that is

$$\begin{array}{ccc}
\langle\langle \mathcal{P}, \mathcal{E}, \mathcal{G}\rangle, \mathcal{S}\rangle & \overset{Unify}{\longrightarrow}_* & \langle\langle \mathcal{P}', \mathcal{E}', \mathcal{G}'\rangle, \mathcal{S}'\rangle \\
\downarrow \sigma & & \downarrow \sigma \\
\langle\langle \mathcal{P}, \mathcal{E}, \mathcal{G}\rangle\sigma, \mathcal{S}\sigma\rangle & \overset{Unify}{\not\longrightarrow}_* & \langle\langle \mathcal{P}', \mathcal{E}', \mathcal{G}'\rangle\sigma, \mathcal{S}'\sigma\rangle
\end{array}$$

The reason is that we may have several simple equations instantiating the same placeholder, and the different choices of equations gives different solved equations. For instance, in the following example we start with two equations and only the second is simple equation and is therefore being solved. After instantiating $?_1$ the first equation becomes simple as well, and we have a choice. If the first simple equation is chosen as simple, we get a different unification problem than if we first unify and then instantiate $?_1$. In the example we will only give the equations $\mathcal{E}$ and the solved equations $\mathcal{S}$, and we omit the types and contexts for simplicity. The addition function is assumed to be defined by recursion over the second argument, which means that $add(m, ?_1)$ cannot be computed any further. All placeholders are of type $N$, and so are the terms $n$

and $m$. The chosen simple equations are framed in the picture.

$$
\begin{tikzcd}
\begin{bmatrix} add(m,?_1) = s(?_2) \\ ?_2 = n \end{bmatrix} \arrow[r, "Unify"', "*"] \arrow[dd, "\{?_1 = s(?_3)\}"] & \begin{bmatrix} add(m,?_1) = s(n) \\ \boxed{?_2 = n} \end{bmatrix} \arrow[dr, "\{?_1 = s(?_3)\}"] & \\
 & & \begin{bmatrix} add(m,?_3) = n \\ \boxed{?_2 = n} \end{bmatrix} \\
\begin{bmatrix} add(m,?_3) = ?_2 \\ ?_2 = n \end{bmatrix} \arrow[r, "Unify"', "*"] & \begin{bmatrix} \boxed{?_2 = add(m,?_3)} \\ add(m,?_3) = n \end{bmatrix} \arrow[ur, leftrightarrow] &
\end{tikzcd}
$$

Here we can see that the solved equations differ, but clearly the unification problems have the same solutions.

We will define what it means to apply a refinement instantiation to a unification problem, and it is a bit more complicated than the case for unification instantiations in section 7.3. The reason is that here the type checking will produce a term-TUP which contains the new placeholder declarations as well as new equations. Therefore, we need to convert the result of type checking into a new triple $\langle \mathcal{P}'', \mathcal{E}'', \mathcal{G}'' \rangle$ where $\mathcal{P}''$ consists of the new placeholder declarations, and $\mathcal{E}''$ the new equations. The new dependency graph, $\mathcal{G}''$, is the graph relating the new placeholders to each other. However, $\mathcal{G}''$ may also contain placeholder known before, since the type and context arguments of the type checking may contain such placeholders. To merge the two graphs in a correct manner, the transitive closure of the dependency order must be reestablished in the new graph.

**Definition 8.1** A unification problem $\langle \mathcal{P}, \mathcal{E}, \mathcal{G} \rangle$ applied to a refinement instantiation $\{?_n = b\}$ is defined by

$$\langle \mathcal{P}, \mathcal{E}, \mathcal{G} \rangle \{?_n = b\} = \langle \mathcal{P}', \mathcal{E}', \mathcal{G}' \rangle$$

where

$$
\begin{array}{ll}
\mathcal{P}' & = (\mathcal{P} - \{?_n : \alpha_n \;\; \Gamma_n\})\{?_n = b\} \cup \mathcal{P}'' \\
\mathcal{E}' & = \mathcal{E}\{?_n = b\} \cup \mathcal{E}'' \\
\mathcal{G}' & = Update(?_n, PH(b) \cup DO_{Merge(\mathcal{G},\mathcal{G}'')}(PH(b)), Merge(\mathcal{G}, \mathcal{G}''))
\end{array}
$$

and

$$\langle \mathcal{P}'', \mathcal{E}'', \mathcal{G}'' \rangle = Convert(TC_p(\mathcal{P}, b, \alpha_n, \Gamma_n))$$

The definition can be extended to a general instantiation $\sigma$:

$$\langle \mathcal{P}, \mathcal{E}, \mathcal{G} \rangle (\{?_n = b\} \cup \sigma) = (\langle \mathcal{P}, \mathcal{E}, \mathcal{G} \rangle \{?_n = b\})\sigma.$$

## 8.1 Application to proof refinement

In proof refinement, the incomplete term representing the partial proof is successively refined by giving instantiations to the placeholders in the term. The correctness of the incomplete term is ensured by the unification problem we get by type checking the term. The refinement is established by type checking the instantiation. Hence we manipulate an incomplete term (with a type and context) together with a unification problem. We will call this a type checking problem.

The type checking algorithm with unification is used to try to solve a type checking problem, which as we have seen produces a partially solved unification problem. Naturally we want to benefit from the localisation optimisation of the type checking algorithm, so we will represent a partially solved type checking problem as a type checking problem together with a partially solved unification problem $\langle \mathcal{C}, \mathcal{S} \rangle$:

$$\langle e : \alpha \quad \Gamma, \langle \mathcal{C}, \mathcal{S} \rangle \rangle.$$

The validity of such a representation is defined as expected:

**Definition 8.2** We will say that the *partially solved type checking problem* $\langle e : \alpha \quad \Gamma, \langle \mathcal{C}, \mathcal{S} \rangle \rangle$ *is valid* if $\langle \mathcal{C}, \mathcal{S} \rangle$ ensures $e \vdash \alpha : \Gamma$.

The picture below illustrates the process of a user successively instantiating an incomplete term. The vertical arrows labelled by $\sigma_i$ are the user instantiations. On the left side of the line, we have the "user's view", which is the incomplete proof term applied to the instantiations ($\theta_i$) corresponding to the solved equations in the unification problem.

The framed parts are the *internal representations* which are partially solved type checking problems. The placeholder declarations in $\mathcal{C}$ are the subgoals left to prove, and the equations in $\mathcal{C}$ are the constraints on the remaining placeholders. These are also presented to the user.

The localisation corresponds to that the algorithm follows the rightmost path of unification- and instantiation arrows. What this means is that when the user supplies an instantiation, it is type checked with respect to the placeholders expected type and context. If the type checking succeeds, the unification is applied again. If the unification also succeeds, the term is updated with the (user) instantiation.

**The type checking algorithm with unification**

$$
\begin{array}{cc|cccccccccc}
\multicolumn{2}{c|}{\textbf{User's view}} & \multicolumn{10}{c}{\textbf{Internal representation}} \\
\hline
e\theta_0 : \alpha \;\; \Gamma & \xleftarrow{\theta_0} & \boxed{e : \alpha \;\; \Gamma} & \stackrel{TC_p}{\Longrightarrow} & \mathcal{C} & \stackrel{Unify}{\longrightarrow}_* & \boxed{\langle \mathcal{C}_0, \theta_0 \rangle} & & & & & \\
\downarrow \sigma_1 & & \downarrow \sigma_1 & & \downarrow \sigma_1 & & \downarrow \sigma_1 & & & & & \\
e\theta_0\sigma_1\theta_1 : \alpha \;\; \Gamma & \xleftarrow{\theta_1} & \boxed{e\sigma_1 : \alpha \;\; \Gamma} & \stackrel{TC_p}{\Longrightarrow} & \mathcal{C}\sigma_1 & \supseteq & \langle \; , \; \rangle & \stackrel{Unify}{\longrightarrow}_* & \boxed{\langle \mathcal{C}_1, \theta_0\sigma_1 \circ \theta_1 \rangle} & & & \\
\downarrow \sigma_2 & & \downarrow \sigma_2 & & \downarrow \sigma_2 & & & & \downarrow \sigma_2 & & & \\
\vdots & & \vdots & & \vdots & & & & \ddots & & & \\
\downarrow \sigma_n & & \downarrow \sigma_n & & \downarrow \sigma_n & & & & & \downarrow \sigma_n & & \\
e\theta_0(\sigma_i\theta_i)^* : \alpha \;\; \Gamma & \xleftarrow{\theta_n} & \boxed{e\sigma_n^* : \alpha \;\; \Gamma} & \stackrel{TC_p}{\Longrightarrow} & \mathcal{C}\sigma_n^* & \supseteq & \cdots & \supseteq & \langle \; , \; \rangle & \stackrel{Unify}{\longrightarrow}_* & \boxed{\langle [\,], \theta_n^* \rangle} &
\end{array}
$$

| | |
|---|---|
| where $(\sigma_i\theta_i)^* = \sigma_1\theta_1 \cdots \sigma_n\theta_n$ and we know that $\theta_0(\sigma_i\theta_i)^* = \sigma_n^*\theta_n^*$ | where $\sigma_n^* = \sigma_1\sigma_2 \cdots \sigma_n$, $\theta_n^* = \theta_0\sigma_1 \circ \cdots \circ \theta_{n-1}\sigma_n \circ \theta_n \rangle$ and $\supseteq$ refer to the relationship of their corresponding sets of unifiers |

Figure 8.2: The type checking algorithm with unification

The picture also illustrates the soundness proof of the algorithm, since it can be read as a commuting diagram. We have already showed in the previous section that the set of unifiers is preserved by type checking. In the next section we will show that if we apply unification, any solution to the result is also a solution to the input, and that this relation is preserved after a user instantiation. The $\supseteq$ illustrates this relationship of the corresponding sets of solutions. So in the last line we can see that, when the user has instantiated the term enough so that the unification can solve the remaining placeholders, then we know by the soundness theorem 5 that this solution instantiates the proof term to a complete, type correct term. Hence, we need not type check the term when it is completed.

## 8.2 Proof refinement operations

There are only two basic operations on a type checking problem, and from these other can be defined:

- insert - replacing a placeholder by an incomplete refinement term, and
- delete - replacing a sub-term by a new placeholder.

The insert operation we have already defined, since it is exactly applying the instantiation of the placeholders in question with the refinement term.

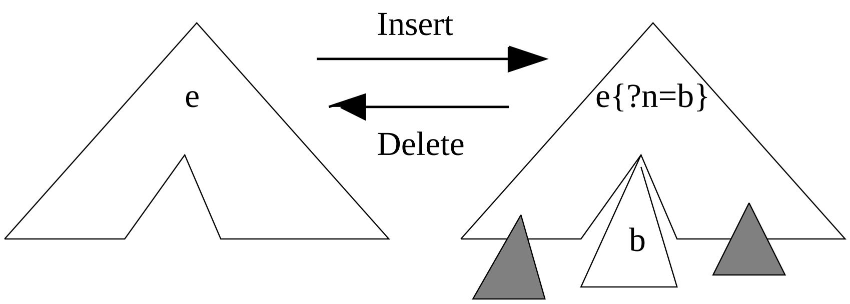



Figure 8.3: The two main operations

The idea is that these two operations should be dual to each other as is shown in figure 8.3, where the placeholder $?n$ in the incomplete term $e$ is replaced by the term $b$, and then the sub-term $b$ is replaced by a placeholder again. The filled triangles illustrate the placeholders which were instantiated by unification.

We claimed in the introduction of chapter 6 that the basic tactics "intro" and "refine" used in most proof assistants could both be defined in terms of the insert operation. This is the case, since we can define

$$intro = insert\ [x]?_1, \text{ and}$$
$$refine\ b = insert\ b(?_1, \ldots, ?_n)$$

where $n$ is the difference in arity between the type of the goal and the type of $b$. We can always compute the arity of the goal-type and the arity of the constant or variable $b$, so we only need to generate $n$ new placeholders for the refine tactic. A refine tactic succeeds if the goal-type matches the type of $b$ applied to the new subgoals, which exactly corresponds to generating the placeholder declarations for $?_1, \ldots, ?_n$ and type checking the inserted term with the expected goal-type. For the intro tactic, the type of the goal must be a function $(x : \alpha)\beta$ and the new subgoal should be of type $\beta$ in the context of the goal extended by the variable $x$ of type $\alpha$. Recalling the type checking, we can see that this is exactly what is checked when the term $[x]?_1$ is inserted for the placeholder of the goal.

The delete operation will "uninstantiate" a sub-term of the incomplete term. Any sub-term can be deleted, so it gives us a general way of locally undoing any instantiation including its consequences.

### 8.2.1 Motivation of local undo

Even though constructing a proof is the purpose of proof refinement, we believe that the delete operation is an important operation for a proof assistant. Since proofs can be rather large and complicated it is easy to make mistakes, and then there must be a way to go back. Most proof assistants support some undo mechanism which is a global state undo, which either reconstructs the proof up to an earlier point, or remembers the state changes and can that way rewind to an earlier state. However the problem with these undo is that they are chronological, which means that often parts of the proof totally unrelated to the part we want to undo have to be removed as well.

The desire of a local undo mechanism has been expressed in [TBK92]. The delete operation is a local undo mechanism (earlier presented in [Mag93]). Recently, a local undo mechanism was also presented in [FH94], but its application is to a tactic-based proof assistant which manipulates refinement trees.

We believe that the advantage of local undo is well illustrated in the little story below (see figure 8.4). The story is about Calvin who is getting dressed for school one winter morning.


Figure 8.4: Local undo beats global undo...

The story can be formalised in ALF. We will define the set of clothes by a datatype with 6 constructors:

$Clothes : Set$
$\quad hat : Clothes$
$\quad scarf : Clothes$
$\quad jacket : Clothes$
$\quad socks : Clothes$
$\quad shoes : Clothes$
$\quad mittens : Clothes$

and to be dressed as a proposition which says that there exists a list of clothes such that there is one of each of the clothes items and they are in the proper order:

$$Dressed \equiv \exists l \in List(Clothes)\ .\ OneOfEach(l)\ \&\ ProperOrder(l)$$

where *OneOfEach* can be defined as

$$\begin{aligned} OneOfEach(l) \equiv\ & (lenght(l) =_N 6)\ \&\ Mem(hat, l)\ \& \\ & Mem(scarf, l)\ \& \ldots\ \& Mem(mittens, l) \end{aligned}$$

where $a =_A b$ is an abbreviation of $Id(A, a, b)$. *Mem* is an inductive relation denoting membership of an element in a list. Here we use the polymorphic notation, i.e. the type of the list elements is a hidden argument.

A proper order to get dressed is defined as

$$\begin{aligned} ProperOrder(l) \equiv\ & last(l) =_{Clothes} mittens\ \& \\ & Before(socks, shoes, l)\ \& \\ & Before(jacket, scarf, l) \end{aligned}$$

since we all know that it is impossible to get dressed with big mittens on, that socks must be put on before shoes and that jackets must be put on before scarfs. *Before* is inductively defined with two constructors,

$Before : (c_1, c_2 : Clothes; l : List(Clothes))Set$
$\quad before1 : (c_1, c_2 : Clothes; l : List(Clothes); h : Mem(c_2, l))$
$\qquad\qquad\qquad\qquad Before(c_1, c_2, c_1 :: l)$
$\quad before2 : (c_1, c_2, c_3 : Clothes; l : List(Clothes); h : Before(c_2, c_3, l))$
$\qquad\qquad\qquad\qquad Before(c_2, c_3, c_1 :: l)$

where the first constructor states that if $c_1$ is the first element in a list and $c_2$ is a member of that list, then $c_1$ is before $c_2$ in that list. The second constructors corresponds to the inductive case, when we know that $c_2$ is before $c_3$ in a list it still holds if we put one element in front of the list.

Now Calvin has to prove that he is properly dressed. He must find a list $l$ such that $OneOfEach(l)$ and $ProperOrder(l)$ is satisfied, that is he must find instantiations of the placeholders $?_l, ?_P$ and $?_Q$ where

$$
\begin{aligned}
&?_l : \mathit{List}(\mathit{Clothes})\\
&?_p : (\mathit{lenght}(?_l) =_N 6)\&\mathit{Mem}(\mathit{hat}, ?_l)\& \ldots \&\mathit{Mem}(\mathit{mittens}, ?_l)\\
&?_q : \mathit{last}(?_l) = \mathit{mittens}\&\mathit{Before}(\mathit{socks}, \mathit{shoes}, ?_l)\&\mathit{Before}(\mathit{jacket}, \mathit{scarf}, ?_l)
\end{aligned}
$$

Hence, his proof will have the form

$$\langle ?_l, \langle\langle ?_{p0}, \ldots, ?_{p6}\rangle, \langle ?_{q1}, ?_{q2}, ?_{q3}\rangle\rangle\rangle$$

since a proof of a existential statement is a dependent pair, where the first component is the witness and the second component is the proof that the witness satisfies the property. Moreover, a proof of a conjunction is also a pair, but here the two components do not depend on each other. The placeholders $?_{p0}, \ldots, ?_{p6}$ therefore correspond to the conjuncts in $\mathit{OneOfEach}(?_l)$ and accordingly, the placeholders $?_{q1}, ?_{q2}, ?_{q3}$ correspond to the conjuncts in $\mathit{ProperOrder}(?_l)$.

He proceeds by finding the six items of clothes which corresponds to refining the unknown goal with a list of length six

$$?_l = [?_1, ?_2, ?_3, ?_4, ?_5, ?_6]$$

and he can prove $?_{p1}$, that is $\mathit{length}([?_1, ?_2, ?_3, ?_4, ?_5, ?_6]) =_N 6$. He puts on his shoes, by refining $?_1 = \mathit{shoes}$ so the list is refined to

$$?_l = [\mathit{shoes}, ?_2, ?_3, ?_4, ?_5, ?_6].$$

He continues with the jacket, scarf, hat and mittens, resulting in the list

$$?_l = [\mathit{shoes}, \mathit{jacket}, \mathit{scarf}, \mathit{hat}, \mathit{mittens}, ?_6].$$

Simultaneously, he can prove $\mathit{Mem}(\mathit{item}, ?_l)$ for all the items so far in the list as well as the property $\mathit{Before}(\mathit{jacket}, \mathit{scarf}, ?_l)$. This means he has an almost complete proof

$$\langle[\mathit{shoes}, \mathit{jacket}, \mathit{scarf}, \mathit{hat}, \mathit{mittens}, ?_6], \langle\langle p0, \ldots, p5, ?_{p6}\rangle, \langle ?_{q1}, ?_{q2}, q3\rangle\rangle\rangle$$

where $p0, \ldots, p5$ and $q3$ denote the proofs of $?_{p0}, \ldots, ?_{p5}$ and $?_{q3}$, respectively.

But at this point he realizes that his socks are left over... He tries to sneak out the door, but Calvin's mom (read ALF) forbids it. To fill in the proof of $?_{p6}$, that is to show that the socks are members of the list, he needs $?_6 = \mathit{socks}$ but this violates that $\mathit{last}(l) =_{\mathit{Clothes}} \mathit{mittens}$.

This is a dead end. After some time of deep contemplation, Calvin realizes that he must not get totally undressed, but it is enough to remove his mittens and shoes and their corresponding proofs of memberships in the list. Thus he can locally undo the proof object to

$$\langle[?_1, \mathit{jacket}, \mathit{scarf}, \mathit{hat}, ?_5, ?_6], \langle\langle p0, ?_{p1}, p2, \ldots, p4, ?_{p5}, ?_{p6}\rangle, \langle ?_{q1}, ?_{q2}, q3\rangle\rangle\rangle$$

and he can still keep the proofs of

$\mathit{Mem}(\mathit{jacket}, ?_l)$, $\mathit{Mem}(\mathit{scarf}, ?_l)$, $\mathit{Mem}(\mathit{hat}, ?_l)$, and

$Before(jacket, scarf, ?_l)$

since they do not depend on the choices of $?_1, ?_5$ and $?_6$. The only thing left to do is to complete the list to

$[socks, jacket, scarf, hat, shoes, mittens]$

and to prove the remaining few properties.

Note that after the deletion, the state of the list resulted in a state it had not been in before, since only the second, third and fourth item was filled in. Hence, this operation is clearly more powerful than a chronological undo which could not achieve such a thing.

### 8.2.2 Insert and delete

In this section we will define the operations insert and delete which are operations on a partially solved type checking problem. The operations are showed to be correct (theorem 6), which means that the validity of the type checking problem is preserved by the operations.

The insert operation takes an instantiation of a placeholder and applies it to the unification problem. Hence, the instantiation is type checked relative its expected type and context and then unification is applied to the result. The delete operation replaces a given sub-term position by a placeholder, and then type checks the entire term again. We will see in the examples below why the delete operation is difficult to localise.

**Definition 8.3** We define the operations insert and delete on a partially solved type checking problem as follows:

**insert:**

$$\langle e : \alpha \quad \Gamma, \langle \mathcal{C}, \mathcal{S} \rangle \rangle \quad \stackrel{?_n = b}{\longrightarrow} \quad \langle e\{?_n = b\} : \alpha \quad \Gamma, \langle \mathcal{C}', \mathcal{S}' \rangle \rangle$$

where $\langle \mathcal{C}', \mathcal{S}' \rangle = Unify(\langle \mathcal{C}\{?_n = b\}, \mathcal{S}\{?_n = b\} \rangle)$

**delete:**

$$\langle e : \alpha \quad \Gamma, \langle \mathcal{C}, \mathcal{S} \rangle \rangle \quad \stackrel{pos(e) = ?_n}{\longrightarrow} \quad \langle e\{pos(e) = ?_n\} : \alpha \quad \Gamma, \langle \mathcal{C}', \mathcal{S}' \rangle \rangle$$

where $pos(e)$ denotes a sub-term position in e,
and $\langle \mathcal{C}', \mathcal{S}' \rangle = TC_pU(e\{pos(e) = ?_n\}, \alpha, \Gamma)$.

The motivation of keeping user instantiations separate from unification instantiations is because we want duality of the insert and delete operations. If unification instantiations were updated in the term $e$, this would not be possible, as shown in the first example below. The third example illustrates why we need to type check the incomplete term after a delete operation.

**Example 1 - the need to separate user- and unification instantiations**

Assume we have a constant *Member* representing the membership of a set $A$, with the type

$Member : (A{:}Set; a{:}A)Set.$

If we apply the type checking algorithm to the type checking problem

$Member(?_A, ?_a) : Set$

it will produce the placeholder declarations

$?_A : Set$, and
$?_a : ?_A$.

Now if we instantiate $?_a$ by $0$, the unification will assign the value $N$ to the placeholder $?_A$, so the unification problem is solved. Our representation of this is

$Member(?_A, 0) : Set$ and the constraint $?_A = N : Set$

rather than

$Member(N, 0) : Set$

which is what is shown to the user. The reason is that if we delete the instantiation of $0$, and type check the term, we come back to the original problem

$Member(?_A, ?_a) : Set.$

If the unification instantiation would have been updated in the term, deleting the second argument would result in

$Member(N, ?_a) : Set.$

**Example 2 - the need to separate user- and unification instantiations**

This example shows that if we update the term with the unification instantiation, then it will be impossible to delete a sub-term, without first having to delete another sub-term. Assume we have a constant

$f : (x{:}N; h{:}Id(N, x, x))Set,$

and the we refine a goal which is refined with the constant $f$, yielding the type checking problem

$$f(?_x, ?_h) : Set$$

where

$?_x : N$, and
$?_h : Id(N, ?_x, ?_x)$.

If we now instantiate $?_x$ with 0, and then refine $?_h$ with the constructor *refl*, the term is completed to

$$f(0, refl(0)) : Set$$

since the argument to *refl* is instantiated by unification. If the term was really updated with $refl(0)$, then it is not possible to delete the first argument, since when the term

$$f(?_x, refl(0)) : Set$$

is type checked, $?_x$ is again instantiated to 0 by unification. The reason is that we get the unification problem

$$\left[\begin{array}{c} ?_x : N \\ Id(N, ?_x, ?_x) = Id(N, 0, 0) : Set \end{array}\right] \text{ which simplifies to } \left[\begin{array}{c} ?_x : N \\ ?_x = 0 : N \\ ?_x = 0 : N \end{array}\right]$$

where $?_x = 0$ clearly is a simple constraint so $?_x$ will be instantiated to 0 again. However, since we do not update the term with the unification instantiation, we actually get the result

$$f(?_x, refl(?_x)) : Set$$

after deleting the first argument, which is the same as if we refine the second argument with *refl* from the beginning.

So the point of the delete operation is to delete a sub-term *including the consequences in the unification caused by this sub-term.*

Due to dependent function types, the type of an argument to a function may depend on previous arguments. Therefore the type of a placeholder may depend on sub-terms of the incomplete term, and if such a sub-term is deleted the placeholder's type may change. This means we have to recompute the types of the placeholders after a delete operation. Moreover, there may be new constraints restricting the new placeholder which is replacing the deleted sub-term. Hence, we need to type check the entire term after deletion, to ensure type correctness of the new term as shown by the following example:

**Example 3 - the need to type check after deletion**

Consider the following example: Suppose we want to prove that 2 divides

6 by solving the goal

$$?x : DIV(2, 6)$$

where

$$DIV(m, n) \equiv \exists k.m * k =_N n$$

A canonical proof of an existential statement is a pair, where the first component is a witness and the second a proof that the witness satisfies the statement. A proof of $DIV(2, 6)$ is therefore a pair

$$<?_k, ?_b>$$

where $?_k$ is a natural number and $?_b$ a proof of $2{*}?_k =_N 6$.

The solution is obvious by choosing the witness $?_k$ to 3, however assume we make a mistake and instantiate $?_k$ to 2. Then we have the incomplete term

$$< 2, ?_b >$$

and one placeholder left to fill in

$$?_b : 2 * 2 =_N 6$$

which clearly is impossible. When we delete the sub-term 2, we must recover the proper type of the placeholder $?_b$, that is come back to

$$<?_k, ?_b>$$

where the type of $?_b$ is again

$$?_b : 2{*}?_k =_N 6.$$

If the type $?_b$ is not altered, it is impossible to recover from our mistake, so we must undo the instantiation of $?_k = 2$ in the type of $?_b$ as well as in the proof term. Undoing an instantiation in the types of placeholders may be difficult, because there can be several occurrences of the instantiated term (2 in this case) and we do not know which ones are due to the instantiation. Furthermore, in the general case we may have done several instantiations in between which effected the type, and the type may have been altered by reduction. Therefore, we need to recompute the type of the placeholders.

If we now instantiate the term properly with the witness 3, we can solve the second component by applying the reflexivity constructor, yielding the completed proof

$$< 3, \mathit{refl}(6) >$$

where the argument 6 to *refl* is instantiated by unification.

## 8.3 Soundness of type checking with unification

We want to show that the localisation of type checking with unification is sound. This means that we operate on the successively more solved unification problem and apply user instantiation to it, rather than to the type checking problem or the result of $TC_p$. Hence, we have the following picture:

$$\begin{tikzcd}
e : \alpha \ \ \Gamma \arrow[r, Rightarrow, "TC_p"] \arrow[d, "\sigma"] & \langle \mathcal{C}, \{\} \rangle \arrow[rr, "Unify", "*"'] \arrow[d, "\sigma"] & & \langle \mathcal{C}_0, \theta \rangle \arrow[d, "\sigma"] & \\
e\sigma : \alpha \ \ \Gamma \arrow[r, Rightarrow, "TC_p"] \arrow[d, "\rho"] & \langle \mathcal{C}\sigma, \{\} \rangle \arrow[d, "\rho"] & ??? & \langle \mathcal{C}_0\sigma, \theta\sigma \rangle \arrow[r, "Unify", "*"'] & \langle \mathcal{C}_1, \theta_1 \rangle \arrow[d, "\rho"] \\
e\sigma\rho : \alpha \ \ \Gamma \arrow[r, Rightarrow, "TC_p"] & \langle \mathcal{C}\sigma\rho, \{\} \rangle & & ??? & \langle \mathcal{C}_1\rho, \theta_1\rho \rangle
\end{tikzcd}$$

In this section we will show that the diagram can be completed, since we have that if $\langle \mathcal{C}, \theta \rangle$ is unified to $\langle \mathcal{C}', \theta' \rangle$, then $\langle \mathcal{C}, \theta \rangle \geq_{\mathcal{U}} \langle \mathcal{C}', \theta' \rangle$, by theorem 4. The completed picture of soundness is as follows, where $=_{\mathcal{U}}$ denotes the same set of unifers:

$$\begin{array}{ccccccc}
e : \alpha \ \ \Gamma & =_{\mathcal{U}} & \langle \mathcal{C}, \{\} \rangle & \geq_{\mathcal{U}} & \langle \mathcal{C}_0, \theta \rangle & & \\
\downarrow \sigma & \boxed{\text{Th. 3}} & \downarrow \sigma & \boxed{\text{Prop 36}} & \downarrow \sigma & & \\
e\sigma : \alpha \ \ \Gamma & =_{\mathcal{U}} & \langle \mathcal{C}\sigma, \{\} \rangle & \geq_{\mathcal{U}} & \langle \mathcal{C}_0\sigma, \theta\sigma \rangle & \geq_{\mathcal{U}} & \langle \mathcal{C}_1, \theta_1 \rangle \\
\downarrow \rho & \boxed{\text{Th. 3}} & \downarrow \rho & \boxed{\text{Prop 36}} & \downarrow \rho & \boxed{\text{Prop 36}} & \downarrow \rho \\
e\sigma\rho : \alpha \ \ \Gamma & =_{\mathcal{U}} & \langle \mathcal{C}\sigma\rho, \{\} \rangle & \geq_{\mathcal{U}} & \langle \mathcal{C}_0\sigma\rho, \theta\sigma\rho \rangle & \geq_{\mathcal{U}} & \langle \mathcal{C}_1\rho, \theta_1\rho \rangle
\end{array}$$

The first we will show is that well-formedness of the unification problem is preserved under application of a refinement instantiation. As for the well-fomedness proof for unification, we have to be precise about the representation. The idea is the following: we know that such instantiation only concerns new placeholders, and when these new placeholders declarations are added to the dependency graph, then they cannot create a cyclic graph unless they are circular themselves. This we know cannot be the case, since the additional unification problem we get by type checking the instantiation is a well-formed extension by theorem 3.

**Proposition 35** *If $\langle\langle \mathcal{P}, \mathcal{E}, \mathcal{G} \rangle, \mathcal{S} \rangle$ is a well-formed unification problem and $\sigma$ is a refinement instantiation, then*

$\langle\langle \mathcal{P}, \mathcal{E}, \mathcal{G} \rangle\sigma, \mathcal{S}\sigma \rangle$ *is well-formed.*

***Proof:*** Assume $\langle\langle\mathcal{P},\mathcal{E},\mathcal{G}\rangle,\mathcal{S}\rangle$ is well-formed, that is

| | |
|---|---|
| $A(i)$ | $\mathcal{G}$ is acyclic |
| $A(ii)$ | For all $?_m$ in $\mathcal{P}$, $\mathcal{E}$ and $\mathcal{P}\|_{DO_{\mathcal{G}}(?_m)}$ ensures $\Gamma_m \vdash \alpha_m : \mathit{Type}$ |
| $A(iii)$ | $\langle\mathcal{P},\mathcal{E}\rangle$ ensures $\mathcal{S}$ |

We will show that $\langle\langle\mathcal{P}',\mathcal{E}',\mathcal{G}'\rangle,\mathcal{S}'\rangle$ is well-formed, which denotes the result of applying $\{?_n = b\}$, where $b$ contains only new and distinct placeholders. We have by definition 7.6

$$\begin{aligned}
\mathcal{P}' &= (\mathcal{P} - \{?_n : \alpha_n \quad \Gamma_n\})\{?_n = b\} \cup \mathcal{P}'' \\
\mathcal{E}' &= \mathcal{E}\{?_n = b\} \cup \mathcal{E}'' \\
\mathcal{G}' &= \mathit{Update}(?_n, PH(b) \cup DO_{Merge(\mathcal{G},\mathcal{G}'')}(PH(b)), \mathit{Merge}(\mathcal{G},\mathcal{G}'')) \\
\mathcal{S}' &= \mathcal{S}\{?_n = b\}
\end{aligned}$$

and

$$\langle\mathcal{P}'',\mathcal{E}'',\mathcal{G}''\rangle = \mathit{Convert}(TC_p(\mathcal{P}, b, \alpha_n, \Gamma_n))$$

We need to show that

| | |
|---|---|
| $(i)$ | $\mathcal{G}'$ is acyclic |
| $(ii)$ | For all $?_m$ in $\mathcal{P}'$, $\mathcal{E}'$ and $\mathcal{P}'\|_{DO_{\mathcal{G}'}(?_m)}$ ensures $\Gamma_m \vdash \alpha_m : \mathit{Type}$ |
| $(iii)$ | $\langle\mathcal{P}',\mathcal{E}'\rangle$ ensures $\mathcal{S}\{?_n = b\}$ |

By the well-formedness assumption, we know that $\Gamma_n$ and $\alpha_n$ can only contain placeholders which are smaller than $?_n$ in $\mathcal{G}$. Then $\langle\mathcal{P}'',\mathcal{E}'',\mathcal{G}''\rangle$ can only contain placeholders smaller than $?_n$ or among the new placeholders. Moreover, we know that the precondition of type checking is satisfied, so by corollary 32 we have that $\langle\mathcal{P}'',\mathcal{E}'',\mathcal{G}''\rangle$ is a well-formed unification problem relative to $\langle\mathcal{P},\mathcal{E},\mathcal{G}\rangle$. Now we must show that the merging of the two preserves well-formedness.

$(i)$ The intuition is that we only replace node $?_n$ by an acyclic graph, where the new graph only depends on nodes "below" node $?_n$, and the nodes above $?_n$ now point to the acyclic graph instead.

We have that $\mathcal{G}$ is acyclic by assumption $A(i)$ and the new graph $\mathcal{G}''$ is acyclic by corollary 32. Moreover, since the nodes in $\mathcal{G}''$ are all new placeholders, we know that $\mathcal{G}$ can not depend on these placeholders, so we have that $\mathit{Merge}(\mathcal{G},\mathcal{G}'')$ is acyclic by proposition 30$(ii)$. The updating of the graph also preserves acyclicity, since $b$ is type checked relative the type and context of $?_n$, so the placeholders in $b$ can only depend on placeholders from $DO_{\mathcal{G}}(?_n)$ or the new placeholders, that is nodes "below" the node $?_n$. Hence, $?_n \notin PH(b) \cup DO_{Merge(\mathcal{G},\mathcal{G}'')}(PH(b))$, so we have that $Update(?_n, PH(b) \cup DO_{Merge(\mathcal{G},\mathcal{G}'')}(PH(b)), Merge(\mathcal{G},\mathcal{G}''))$ is acyclic by proposition $30(i)$.

$(ii)$ Analogous to case $(ii)$ in proposition 33. The only difference is that for the new placeholders, we get the well-formedness by corollary 32.

$(iii)$ Assume $\langle \mathcal{P}', \mathcal{E}' \rangle\sigma$ hold, that is

$((\mathcal{P} - \{?_n : \alpha_n \;\; \Gamma_n\})\{?_n = b\})\sigma \cup \mathcal{P}''\sigma$ hold, and
$(\mathcal{E}\{?_n = b\})\sigma \cup \mathcal{E}''\sigma$ hold.

We need to show that $\mathcal{S}\{?_n = b\}\circ\sigma$ is well-typed. We have by assumption $A(iii)$ that

$(*)$ if $\mathcal{P}\sigma'$ and $\mathcal{E}\sigma'$ hold, then $\mathcal{S}\sigma'$ is well-typed.

We have by corollary 32 that the type correctness of the instantiation $\{?_n = b\}$ is ensured by the new placeholder declarations $\mathcal{P}''$ and the new equations $\mathcal{E}''$. Hence, we know that $\mathcal{P}(\{?_n = b\}\circ\sigma)$ and $\mathcal{E}(\{?_n = b\}\circ\sigma)$ holds, so we get that $\mathcal{S}(\{?_n = b\}\circ\sigma)$ is well-typed by (*) with $\sigma' = \{?_n = b\}\circ\sigma$.

□

Now we know that the application of a refinement instantiation also preserves the well-formedness of a unification problem, so we can come back to our simpler notation. What remains to justify is that our optimisation of type checking is sound. This we will do by showing that the relation $\geq_{\mathcal{U}}$ is preserved under application of refinement instantiations, because then we know that we can apply the refinement instantiation to the unified problem and be sure that the transformed problem really solves the problem we started with.

**Proposition 36** *Let $\langle \mathcal{C}, \theta \rangle$ and $\langle \mathcal{C}', \theta' \rangle$ be a well-formed unification problems, and let $\sigma$ be a refinement instantiation such that $Dom(\sigma) \neq Dom(\theta)$.*

*If $\langle \mathcal{C}, \theta \rangle \geq_{\mathcal{U}} \langle \mathcal{C}', \theta' \rangle$, then $\langle \mathcal{C}\sigma, \theta\sigma \rangle \geq_{\mathcal{U}} \langle \mathcal{C}'\sigma, \theta'\sigma \rangle$*

***Proof:*** We have by assumption that for some $\theta''$:

$(i)$ $\theta' = \theta\circ\theta''$
$(ii)$ $\mathcal{U}(\mathcal{C}\theta'') \supseteq \mathcal{U}(\mathcal{C}')$

We need to show that the conditions are also satisfied for the instantiated unification problems:

$(i)$ We have that $\theta\sigma = (\theta\circ\theta'')\sigma = \theta\sigma\circ\theta''\sigma$ since $Dom(\sigma) \neq Dom(\theta)$.

$(ii)$ Here we need to show that $\mathcal{U}(\mathcal{C}\sigma(\theta''\sigma)) \supseteq \mathcal{U}(\mathcal{C}'\sigma)$. By the assumption we have

if $\mathcal{C}'\gamma$ holds then $(\mathcal{C}\theta'')\gamma$ holds.

Thus, for any $\gamma'$ such that $\mathcal{C}'(\sigma\circ\gamma')$ holds, we have that $(\mathcal{C}\theta'')(\sigma\circ\gamma')$ holds, which implies $(\mathcal{C}(\theta''\circ\sigma\circ\gamma')$ holds by lemma 4.2. Then we have by lemma 36.1 that $\mathcal{C}(\sigma\circ\theta''\sigma\circ\gamma')$ holds and by lemma 36.2 that $\mathcal{C}\sigma(\theta''\sigma\circ\gamma')$ holds. Hence, $\gamma'$ is also a unifier of $\mathcal{C}\sigma(\theta''\sigma)$.

□

**Lemma 36.1** *Let $\sigma$ be a refinement instantiation and $\theta$ any instantiation such that $Dom(\sigma) \neq Dom(\theta)$. Then $\theta\circ\sigma = \sigma\circ\theta\sigma$.*

***Proof:*** We have that $\theta\circ\sigma = \sigma \cup \theta\sigma$ since the domains are distinct. Moreover, $\sigma\circ\theta\sigma = \theta\sigma \cup \sigma(\theta\sigma) = \theta\sigma \cup \sigma$ since $\sigma$ is a refinement instantiation so the terms contains only new placeholders. □

**Lemma 36.2** *Let $\mathcal{C}$ be a well-formed unification problem, $\sigma$ a refinement instantiation and $\rho$ any instantiations. Then*

$$\mathcal{U}((\mathcal{C}\sigma)\rho) = \mathcal{U}(\mathcal{C}(\sigma\circ\rho))$$

***Proof:*** Follows by theorem 3, proposition 35 and proposition 31. □

### 8.3.1 The main results

The main result is that type checking with unification is sound:

**Theorem 5** *Let $\langle\langle\mathcal{P},\mathcal{E},\mathcal{G}\rangle,\mathcal{S}\rangle$ be a well-formed unification problem, $\alpha$ and $\Gamma$ a type and a context, respectively, such that $\langle\mathcal{P},\mathcal{E}\rangle$ ensures $\Gamma \vdash \alpha$ : Type, and let $e$ be a term. Then we have the following soundness result:*
*if*

$$TC_pU\ (\langle\langle\mathcal{P},\mathcal{E},\mathcal{G}\rangle,\mathcal{S}\rangle, e, \alpha, \Gamma) \Rightarrow \langle\langle\mathcal{P}',\mathcal{E}',\mathcal{G}'\rangle,\mathcal{S}'\rangle$$

*then*

$$\langle\mathcal{P}\cup\mathcal{P}',\mathcal{E}\cup\mathcal{E}'\rangle \textit{ ensures } \Gamma\theta_{\mathcal{S}'} \vdash e\theta_{\mathcal{S}'} : \alpha\theta_{\mathcal{S}'}$$

*where $\theta_{\mathcal{S}'}$ is the instantiation of $\mathcal{S}'$.*

***Proof:*** Follows by theorem 4 and corollary 32, since $TC_pU = \xrightarrow{Unify}_* \circ\ TC_p$. □

Finally, we can also show that the operations insert and delete performs valid transformations of a type checking problem:

**Theorem 6** *The operations insert and delete preserves the validity of a partially solved type checking problem.*

***Proof:*** Both operations respect the $\geq_{\mathcal{U}}$ relation by proposition 36 (for insert) and by theorems 4 and 3 (for delete). Hence, we have that the unification problem ensures the type checking problem (by proposition 34), which means that the partially solved type checking problem is valid. $\Box$

## 8.4 Completeness conjecture

There are mainly two problems with proving completeness for the unification algorithm:

1. We may simplify equations which are not well-typed, thereby loosing termination even if the reduction algorithm is normalising.
2. We have only proved that the type checking algorithm is complete for terms containing new and distinct placeholders (since terms do not have unique types), which is not the case for the terms we type check in the unification algorithm.

However, we will suggest some possible attempts of solving these problems.

### 8.4.1 Termination

The problem starts already in the $Simple_p$-algorithm, since it takes as input a term-TUP

$$\begin{bmatrix} a_1 = b_1 : \alpha_1 \quad \Gamma_1 \\ ?_2 : \alpha_2 \quad \Gamma_2 \\ \vdots \\ a_k = b_k : \alpha_k \quad \Gamma_k \\ \vdots \\ ?_n : \alpha_n \quad \Gamma_n \\ \vdots \\ a_p = b_p : \alpha_p \quad \Gamma_p \end{bmatrix}$$

and it tries to simplify all equations as far as possible. In the simplification terms are usually reduced, and we only know that these terms are well-typed relative the previous placeholder declarations and equations. Hence, we may reduce an ill-typed term. One solution would be to simplify the equations in the order they occur, and stop as soon as we encountered an unsolved equation. This is the method used in [Dow93]. However, in contrary to [Dow93], there is no search involved in our algorithm, so such a restriction would not be very satisfactory in practice, since very few placeholders would be instantiated by such unification algorithm.

A better way would be to show that even though the terms may not be well-typed in type theory, they will always be well-typed in a simpler type-system, i.e. non-dependent, simply typed $\lambda$-calculus with two base types. This is the idea used in [Ell89] and [Pym92]. The idea is to map every incomplete type into a type system with two constants `Set` and `El` and the non-dependent function type, which we will denote by $\mathcal{T}$. The types in $\mathcal{T}$ is defined by

$$t ::= \texttt{Set} \mid \texttt{El} \mid t \rightarrow t.$$

We would have to show that if the term-TUP is well-formed, then all equations are well-typed in $\mathcal{T}$. At least if we restrict ourselves to a system without computation rules, i.e. a system like LF, we know for instance from a formal proof by Catarina Coquand ([Coqb]) that such $\lambda$-calculus with explicit substitution is normalising.

The motivation is, if we transform the possibly incomplete types into simple types, they will become complete, since simple types contain no terms and the placeholders denotes terms only. Therefore, all types and context will be complete in the unification problem, which means that the type correctness of placeholders and equations are independent of instantiations of placeholders. If we can show that

1. the transformed types and context are valid types in $\mathcal{T}$
2. any well-typed term in $\vdash$ is also well-typed in the corresponding system $\vdash_{\mathcal{T}}$
3. our reduction rules preserve well-typedness in $\mathcal{T}$

then we can show that all equations and placeholders in a well-formed TUP is well-typed in $\mathcal{T}$, and hence, by normalisation of $\mathcal{T}$, the reduction of the terms in the equations terminate.

The transformation $\mathcal{F}$ is very easy, we simply "forget" all dependencies of terms in a type $\alpha$ and a family of types $\beta$:

$$
\begin{array}{lcl}
\mathcal{F}(Set) & = & \texttt{Set} \\
\mathcal{F}(\alpha \to \beta) & = & \mathcal{F}(\alpha) \to \mathcal{F}(\beta) \\
\mathcal{F}(\beta a) & = & \mathcal{F}(\beta) \\
\mathcal{F}(\alpha\gamma) & = & \mathcal{F}(\alpha) \\
\mathcal{F}(El) & = & \texttt{El} \\
\mathcal{F}([x]\alpha) & = & \mathcal{F}(\alpha) \\
\mathcal{F}(\beta\gamma) & = & \mathcal{F}(\beta)
\end{array}
$$

Note that the transformation corresponds to the computation of constructor form of a type, where we also compute the constructor form of the argument- and result type in a function. We expect the following to hold:

Let $e$ be a term, $\alpha$ a type and $\Gamma$ a context. Let $\mathcal{F}$ denote the transformation above. Then

$$
\begin{array}{lcl}
\vdash \Gamma : Context & \Rightarrow & \models_{\mathcal{T}} \mathcal{F}(\Gamma) : Context \\
\Gamma \vdash \alpha : Type & \Rightarrow & \mathcal{F}(\Gamma) \models_{\mathcal{T}} \mathcal{F}(\alpha) : Type \\
\Gamma \vdash e : \alpha & \Rightarrow & \mathcal{F}(\Gamma) \models_{\mathcal{T}} e : \mathcal{F}(\alpha)
\end{array}
$$

where $\mathcal{F}(\Gamma)$ means the obvious extension of the translation to contexts.

Assuming the properties above, we will have that the $Simple_p$-algorithm terminates for any well-formed TUP, since

- For $[\mathcal{C}, ?_n : \alpha_n \;\; \Gamma_n]$ we know $\forall \sigma.\mathcal{C}\sigma$ holds $\Rightarrow \Gamma_n\sigma \vdash \alpha_n\sigma : Type$ which means that $\mathcal{F}(\Gamma_n) \models_{\mathcal{T}} \mathcal{F}(\alpha_n)$: $Type$, without the condition that $\mathcal{C}\sigma$ holds, since the instantiation $\sigma$ does not effect the type or context at all.

- Similarly, if we know $\Gamma\sigma \vdash a\sigma : \alpha\sigma$ for any $\sigma$ satisfying $\mathcal{C}$, then we have that $\mathcal{F}(\Gamma) \models_{\mathcal{T}} e\sigma : \mathcal{F}(\alpha)$

Hence we know that the terms occuring in the equations have a type in $\mathcal{T}$, so they can be reduced without jeopardising termination.

### 8.4.2 Completeness of type checking

As already mentioned in the introduction of chapter 6, we have a counterexample showing that type checking is not complete when we allow several occurrences of the same placeholder in the incomplete term. The example is

$$f(?_1, ?_1) : N$$

where f is of type

$$f : ((N)N; (Bool)Bool)N.$$

The reason is that we expect different occurrences of a placeholder to have the same type, since in the transformation to equations we will have only one declaration of the placeholder, that is only one expected type and one expected context. Other occurrences are then type checked relative this declaration, that is their types and contexts are required to become equal in any instantiation. In the example above, $?_1$ can be instantiated by the identity function, which may have any type of the form $(A)A$, and which are clearly different types. Another example is when a placeholder have types $(A)B$ and $(C)B$, then such placeholder may be instantiated to any function which throws away its argument and has result type $B$. When we type check incomplete terms containing several occurrences of such placeholders, the type checking will fail.

One solution would be to allow several placeholder declarations of the *same* placeholder, i.e. drop the sharing of placeholder declarations. Then any instantiation would have to be type correct relative *all* expected types and contexts of that placeholder. Then we would, for the example above, get a unification problem

$$\left[\begin{array}{c} ?_1 : (N)N \ \ [\,] \\ ?_1 : (Bool)Bool \ \ [\,] \end{array}\right]$$

and it will have the same solutions as the original type checking problem. This means that every occurrence of a placeholder is considered as a "new placeholder" and we must be able to compute its expected type and context. This requirement will be a great restriction in the unification algorithm, since when we type check instantiations suggested by unification, we have no guarantee that placeholders occur in positions for which we can compute the type. For instance, we may have a term

$$?_1\{x{:=}?_2\}$$

where we could not compute the type of $?_2$ and not the context of $?_1$. Hence, we could either only unify terms in which the placeholders have computable types and contexts, or we could treat the type checking of unified terms in a different manner as other incomplete terms. Neither is very tempting, and since the counterexamples of completeness are rather "constructed" examples, we believe the best choice is to keep the sharing of placeholder declarations.

A more drastic solution would be to change the formulation of the type theory such that abstractions are labelled with the types of the bound variables, like in Generalised Type Systems [Bar91]. Then we would have uniqueness of types, and this problem vanishes.

Anyhow, we would like to show that unification preserves the set of solutions, i.e.

if $\mathcal{C} \overset{Unify}{\longrightarrow} \langle \mathcal{C}', \mathcal{S}\rangle$, then $\mathcal{U}(\mathcal{C}) = \mathcal{U}\langle \mathcal{C}', \mathcal{S}\rangle$

not only $\mathcal{U}(\mathcal{C}) \supseteq \mathcal{U}\langle \mathcal{C}', \mathcal{S}\rangle$. As mentioned, the reason we do not get equality, is that the type checking only gives us

if $TC_p\,(\mathcal{C}, e, \alpha, \Gamma) \Rightarrow \mathcal{C}'$, then $(\mathcal{C} \mathbin{@} \mathcal{C}')$ ensures $\Gamma \vdash e : \alpha$

but the other direction does not hold, i.e.

$\Gamma \vdash e : \alpha$ does not ensure $(\mathcal{C} \mathbin{@} \mathcal{C}')$,

since if any placeholder occurring in $e$ have a different type than the declaration in $\mathcal{C}$, then $\mathcal{C}'$ will be impossible to satisfy. However, if we *knew* that all occurrences of placeholders in $e$ were *properly shared*, that is, for any valid instantiation of $\mathcal{C} \mathbin{@} \mathcal{C}'$, all occurrences of the placeholders would have the same expected type, then we would get the converse as well.

**Definition 8.4** Let $?_n : \alpha_n \;\; \Gamma_n$ be a placeholder declaration in a unification problem $\mathcal{C}$. Let $e$ be a term, $\beta$ a type and $\Delta$ a context such that

$\mathcal{C}$ ensures $\Delta \vdash e : \beta$

We will say that $?_n$ *is properly shared* in $\langle e, \beta, \Delta\rangle$ (relative $\mathcal{C}$) if for all positions $p_1, \ldots, p_k$ in $e$ where $?_n$ occurs, we have that

$\mathcal{C}$ ensures $\Gamma_n \vdash \alpha_n = \tau(p_i, \langle e, \beta, \Delta\rangle) : \mathit{Type}$ for $i = 1, \ldots, k$

where $\tau(p, \langle e, \beta, \Delta\rangle)$ is the expected type of position $p$ in $e$.

We will say that $\langle e, \beta, \Delta\rangle$ is properly shared in $\mathcal{C}$ if all placeholders in $e$ are properly shared.

The notion can be extended to unification problems, as follows

**Definition 8.5** We will say that a well-formed unification problem is properly shared if for all equations $a = b : \alpha \;\; \Gamma$ in $\mathcal{C}$, we have that

$\langle a, \alpha, \Gamma\rangle$ is properly shared in $\mathcal{C}$, and
$\langle b, \alpha, \Gamma\rangle$ is properly shared in $\mathcal{C}$.

Now we can state the conjecture.

**Conjecture 1** *Let $\mathcal{C}$ be a well-formed, properly shared unification problem. Let $e$ be a refinement term, $\alpha$ a type and $\Gamma$ a context such that $\mathcal{C}$ ensures*

$\Gamma \vdash \alpha$ *: Type. If*

$$TC_p\,(\mathcal{C}, e, \alpha, \Gamma) \Rightarrow \mathcal{C}'$$

*then* $\mathcal{C} \,@\, \mathcal{C}'$ *is properly shared.*

The intuition is the following; since we are considering refinement terms, we know that each placeholder only occurs once. The reason that placeholders can occur in the equations is that we have a dependent function applied to arguments containing placeholders. This means that an argument is substituted in place of a variable in the dependent function, and if the argument contains placeholders, we get several occurrences of these placeholders in the unification problem. However, before the argument is substituted in the type of the function we have generated equations which ensures that the argument is of the expected type, that is the type of the variable we will substitute for. Finally, since type correctness of a term implies type correctness of all its parts, the type correctness of the placeholders in the argument is assured by the generated equations.

If the conjecture is true and a unification problem $\mathcal{C}$ is well-formed and properly shared, we know that for any simple constraint

$$?_n = e : \beta \quad \Delta$$

we have that

$\mathcal{C}$ ensures $?_n \vdash \beta : \Delta$ and
$\mathcal{C}$ ensures $e \vdash \beta : \Delta$ (by well-formedness), and
$\mathcal{C}$ ensures $\Gamma_n \vdash \alpha_n = \beta : \mathit{Type}$

due to the properly shared condition. The first condition means that all variables that $?_n$ depends on (that is all variables in $\Gamma_n$) is included in $\Delta$, so $\Gamma_n$ is a sub-context of $\Delta$. This, together with the two following conditions imply that

$\mathcal{C}$ ensures $e \vdash \alpha_n : \Delta$

but it does *not* imply that all variables in $e$ are declared in $\Gamma_n$.

Thus, if the conjecture is true, we can avoid type checking the unified term, but we must check its scope anyhow, that is check that all free variables in the term is defined in a given context. Then the unification can manipulate the equations, and instantiate simple constraints, and these operations should preserve the set of unifiers. Even if we did type check the unified term, type checking will preserve the set of unifiers for unification terms, if the known placeholders in the term are ensured to have their expected types.

# Chapter 9

# The ALF proof engine

The proof engine has mainly three parts, the administration of commands, the type checker and the environment. The environment has two parts, the *theory definitions* and the *scratch area*. In the theory, there are only complete definitions and in the scratch area the incomplete definitions are represented. All proof editing take place in the scratch area, and proofs are built up interactively by the scratch area operations which we will describe in section 9.2.

## 9.1 ALF theories

An ALF theory consists of a list of constant definitions, of the kind described in section 2.2. All constants are global definitions, therefore the constant names must all be distinct. This is not satisfactory when the theory is large, and there is ongoing work to introduce a notion of modules for ALF theories.

The theory definitions are only checked to be type correct, by the rules below. The justifications of primitive definitions must be done outside the system. For the implicit constants, the generation of pattern matching equations guarantees exhaustive, non-overlapping sets of patterns ([Coq92]), but termination of these functions is not guaranteed since they may be general recursive. One could restrict the recursive calls to only allow calls on structurally smaller arguments as is defined in [Coq92], [Dah92] and [Nor88]. However, to achieve flexibility, we may not want to impose on too many syntactical restrictions. Termination can be proved (by the user) as suggested in [Nor88], but some kind of termination check ought to be implemented. On the other hand, there are experiments being done in ALF to represent infinite objects in type theory (see [Coq94],

[Coqa], [Pra]), which would not be possible with such restrictions. Thus, some additional justifications must be done of an ALF theory. See [Alt94] for a discussion of consistent theories in ALF. A safe approach is to work within the "standard" part of type theory by only using the sets and elimination rules presented in [NPS90]. This monomorhpic set theory is provided as a standard library in ALF.

A theory definition is a constant definition, which can either be a primitive constant, an explicitly defined constant or an implicitly defined constant. Before we define what we mean be type correct definitions, we will introduce some notions. The *result type* $R(\alpha)$ of a $\beta S$-normal type $\alpha$, is defined by:

$$\begin{array}{lcll} R(\alpha_1 \rightarrow [x]\alpha_2) & = & R(\alpha_2) & \\ R(\alpha) & = & \alpha & \text{when } \alpha \text{ is a ground type} \end{array}$$

and we will say that $\alpha$ is

| | |
|---|---|
| *Set-valued* | if $R(\alpha) = Set$, and |
| *A-valued* | if $R(\alpha) = El(e)$ and $A$ is the head of $e$. |

### Primitive constants

Defining a primitive constant is to define a new inductive set or family of sets. The definition consists of a a set-constructor, that is a constant with a *Set*-valued type, and a collection of constructors of the set. A constructor of the set $A$ is a constant with a $A$-valued type. We have that a primitive constant definition

$$\begin{array}{l} A : \beta \\ \quad \left.\begin{array}{l} c_1 : \alpha_1 \\ \vdots \\ c_n : \alpha_n \end{array}\right\} constr(A) \end{array}$$

is valid (denoted by $ValidPrim\ (A : \beta, constr(A)))$ with respect to a theory $\mathcal{T}$ if the following hold

- $\beta$ is a *Set*-valued type,
- $\alpha_1, \ldots, \alpha_n$ are $A$-valued types,
- $A, c_1, \ldots, c_n$ are not previously defined in $\mathcal{T}$,
- $\mathcal{T} \vdash \beta : Type$, and
- $[\mathcal{T}, A : \beta] \vdash \alpha_i : Type$ for $1 \leq i \leq n$.

We may want to define inductive sets which depends mutually on one and another. For a more extensive presentation, see [Dyb94]. For instance, if we want to define a set of terms in a language with explicit substitution, the terms and substititions depend on each other. $\lambda$-terms extended with explicit substituion, could be defined as follows:

$$\begin{array}{ll}
\boldsymbol{Exp} : Set & \boldsymbol{Subst} : Set \\
\quad var \;\; : (n{:}N) Exp & \quad empty : Subst \\
\quad lam \;\; : (n{:}N; e{:}Exp) Exp & \quad ext \;\; : (s{:}Subst; n{:}N; e{:}\boldsymbol{Exp}) Subst \\
\quad app \;\; : (e_1, e_2{:}Exp) Exp & \\
\quad subst : (e{:}Exp; s{:}\boldsymbol{Subst}) Exp &
\end{array}$$

A $\lambda$-term is either a variable, an abstraction, an application or a term applied to a substitution. Hence, we have four constructors in the set, one for each case. Here we have represented variables as natural numbers, and one can think of these as the infinite set $x_0, x_1, \ldots$. Substitutions are either empty or a substitution extended by one new assignment, which is reflected in the types of the two constructors of $Subst$.

A valid primitive constant definition with respect to theory $\mathcal{T}$, is then a generalisation of the above requirements. We can define a collection of mutually inductive (families of) sets, by first giving all set-constructors, and afterwards the respective constructors of the corresponding sets. A primitive constant definition is valid if

$$\frac{\begin{array}{cccc}
\mathcal{T} \vdash \beta_1 : \mathit{Type} & [\mathcal{T}, \overrightarrow{A : \beta}] \vdash \alpha_1^1 : \mathit{Type} & & [\mathcal{T}, \overrightarrow{A : \beta}] \vdash \alpha_1^k : \mathit{Type} \\
\vdots & \vdots & \cdots & \vdots \\
\mathcal{T} \vdash \beta_n : \mathit{Type} & [\mathcal{T}, \overrightarrow{A : \beta}] \vdash \alpha_{n_1}^1 : \mathit{Type} & & [\mathcal{T}, \overrightarrow{A : \beta}] \vdash \alpha_{n_k}^k : \mathit{Type}
\end{array}}{\mathit{ValidPrim}\ (\{A_1 : \beta_1, \ldots, A_k : \beta_k\}, \{constr(A_1), \ldots, constr(A_k)\})}$$

where $\overrightarrow{A : \beta}$ denotes $A_1 : \beta_1, \ldots, A_k : \beta_k$. Further, $A_1, \ldots, A_k$ are required to be *Set*-valued and the types of $constr(A_i)$ $A_i$-valued, for $1 \leq i \leq k$ respectively.

### Implicit constants

An implicit constant definition consists of a new constant with a functional type, and a collection of pattern equations defining the function. Recall the definition of a pattern on page 88 that a pattern is of the form

$$p ::= \; x \mid c_c \mid c_c(p_1, \ldots, p_n),$$

where $n$ is the arity of the constructor $c_c$. A pattern rule is of the form

$$P_{rule} ::= \; c_i(p_1, \ldots, p_n) = e$$

where $n$ is the arity of $c_i$. We also had the restriction that all pattern variables in a pattern are distinct.

The type and the pattern context can be computed from a given left-hand side of a pattern rule by the algorithm $\Rightarrow_P$

$$c_i(p_1, \ldots, p_n) \Rightarrow_P \langle \Gamma_p, \alpha_p \rangle$$

where $\Gamma_p$ is the context of the pattern variables occuring in $p_1, \ldots, p_n$ and $\alpha_p$ is the type of $c_i(p_1, \ldots, p_n)$ in that context.

$$\frac{}{c_i \Rightarrow_P \langle [\,], \alpha_c \rangle}\ c_i : \alpha_c \in \Sigma \qquad \frac{f \Rightarrow_P \langle \Gamma, \alpha \rightarrow \beta \rangle}{(fx) \Rightarrow_P \langle [\Gamma, x : \alpha], \beta x \rangle}$$

$$\frac{f \Rightarrow_P \langle \Gamma, \alpha \rightarrow \beta \rangle \quad p \Rightarrow_P \langle \Delta, \alpha \rangle}{(fp) \Rightarrow_P \langle \Gamma + \Delta, \beta p \rangle}$$

where $\Gamma + \Delta$ is simply the concatenation of the two contexts. This is a valid context, since all pattern variables are required to be distinct, and since

- constructors have closed types, and
- all applications of constructors are saturated, i.e if $p_c$ is a constructor pattern, then the type of $p_c$ is ground.

Hence, we know that $\Gamma$ and $\Delta$ in the last rule is not related in any way, so $\Gamma + \Delta$ is a valid extension of $\Gamma$, and $\Gamma + \Delta = \Delta + \Gamma$.

If the type $\beta$ is a dependent type, then we will have different types in the different pattern-rules. The contexts will in general always be different, since constructors have different arities and types. For instance, in the example of commutativity of addition

$$\begin{array}{l} add\_comm : (n, m{:}N) Id(N, add(n, m), add(m, n)) \\ \quad add\_comm(0, m) = \ldots \\ \quad add\_comm(s(n), m) = \ldots \end{array}$$

we will have that

$add\_comm(0, m) \Rightarrow_P \langle [m : N], Id(N, add(0, m), add(m, 0)) \rangle$, whereas
$add\_comm(s(n), m) \Rightarrow_P \langle [n, m : N], Id(N, add(s(n), m), add(m, s(n))) \rangle$.

The $\Rightarrow_P$ computes correct types and contexts, which is expressed in the following result:

**Proposition 37** *If* $P_{lhs} \Rightarrow_P \langle \Delta_p, \alpha_p \rangle$ *then* $\Delta_p \vdash P_{lhs} : \alpha_p$.
***Proof:*** By induction on the structure of $P_{lhs}$. □

To check that an implicit constant definition

$$c : \beta$$

$$\left.\begin{array}{l} c(p_1^1,\ldots,p_1^n) = e_1 \\ \vdots \\ c(p_k^1,\ldots,p_k^n) = e_k \end{array}\right\} patterns(c)$$

is valid with respect to a theory $\mathcal{T}$, we must compute the type and pattern context for each pattern rule, and then check that the right-hand side is of the same type as the corresponding pattern. Thus, the definition is valid if

- $c \notin \mathcal{T}$,
- $\mathcal{T} \vdash \beta : \mathit{Type}$ and $\beta$ has arity $n$,
- all patterns are of the proper form, and
- $[\mathcal{T}, c : \beta], \Delta_i \vdash e_i : \alpha_i$ for $1 \leq i \leq k$, where $c(p_i^1,\ldots,p_i^n) \Rightarrow_P \langle \Delta_i, \alpha_i \rangle$. We will denote this property $ValidPatterns(c)$.

As already mentioned, this validity defined here is simply type correctness, and nothing more. The non-overlapping and exhaustiveness of patterns are assured by the creation of the pattern-rules, but is not checked here again. Also, implicit constants are allowed to be recursive, which means that the right-hand side of the pattern rule may refer to the constant itself.

Here as well as for primitive constants, we want to allow mutually recursive functions, so we will allow a collection of implicit constant definitions. Clearly, if we want to write functions defined over mutually defined primitives, we also need mutually defined functions. For example, we may want to define a function computing the set (or list) of free variables occurring in the terms defined by $Exp$ in the example above.

$$\begin{array}{l} \boldsymbol{FV} : (e : Exp)List(N) \\ \quad FV(var(n)) = [n] \\ \quad FV(lam(n,e)) = FV(e) - [n] \\ \quad FV(app(e_1,e_2)) = FV(e_1) \mathbin{@} FV(e_2) \\ \quad FV(subst(e,s)) = (FV(e) - Dom(s)) \mathbin{@} \boldsymbol{FVsubst}(s) \end{array}$$

and we must simultaneously define

$$\begin{array}{l} \boldsymbol{FVsubst} : (s : Subst)List(N) \\ \quad FVsubst(empty) = [] \\ \quad FVsubst(ext(s,n,e)) = FVsubst(s) \mathbin{@} \boldsymbol{FV}(e) \end{array}$$

Just as before, we check the types of the functions first, and then extend the theory with these new constants and check all their patterns afterwards: An implicit constant definition is valid if

$$\frac{\begin{array}{cc} \mathcal{T} \vdash \beta_1 : Type & [\mathcal{T}, \overrightarrow{c : \beta}] \vdash ValidPatterns(c_1) \\ \vdots & \vdots \\ \mathcal{T} \vdash \beta_m : Type & [\mathcal{T}, \overrightarrow{c : \beta}] \vdash ValidPatterns(c_m) \end{array}}{ValidImpl\ (\{c_1 : \beta_1, \ldots, c_m : \beta_k\}, \{patterns(c_1), \ldots, patterns(c_m)\})}$$

where $\overrightarrow{c : \beta}$ denotes $\{c_1 : \beta_1, \ldots, c_m : \beta_m\}$ and $c_1, \ldots, c_m$ are required to be new and distinct.

**Explicit constants**

An explicit constant definition is the simplest one, since it just gives a name to a type correct term, as we can see in the definition of a valid theory:

A *Valid* theory is defined inductively by

$$\frac{}{Valid\ [\,]} \qquad \frac{Valid\ \mathcal{T} \quad \Gamma \vdash e : \alpha}{Valid\ [\mathcal{T}, c = e : \alpha\ \ \Gamma]}\ c \notin \mathcal{T}$$

$$\frac{Valid\ \mathcal{T} \qquad ValidPrim\ (\mathcal{T}, d)}{Valid\ [\mathcal{T}, d]} \qquad \frac{Valid\ \mathcal{T} \qquad ValidImpl\ (\mathcal{T}, d)}{Valid\ [\mathcal{T}, d]}$$

This concludes the description of a valid theory. The next section describes the second part of the environment; the scratch area.

## 9.2 The Scratch Area

The scratch area is where theory definitions are built up interactively. It consists of a collection of *incomplete theory definitions* which are theory definitions containing *type checking problems*. A type checking problem is an incomplete term $e$, an expected type $\alpha$ and a local context $\Gamma$

$$e : \alpha \quad \Gamma$$

and the problem is to find instantiations to the placeholders in $e$ such that the instantiated term is type correct

$$\Gamma \vdash e\sigma : \alpha$$

where $e\sigma$ denotes the instantiated term.

An incomplete explicit constant definition contains one type checking problem, whereas an implicit constant definition contains several type checking problems, that is one for each right-hand side of the pattern equations. Primitive constants contains no type checking problems, since it only consists of constructors and types. The incomplete theory definitions is then of the form

$$c = \boxed{e : \alpha \quad \Gamma}$$

or

$$\begin{array}{l} c : \beta \\ \quad c(p_1^1, \ldots, p_1^n) = \boxed{e_1 : \alpha_1 \quad \Gamma_1} \\ \qquad \vdots \\ \quad c(p_k^1, \ldots, p_k^n) = \boxed{e_k : \alpha_k \quad \Gamma_k} \end{array}$$

One of the purposes of having several incomplete definitions available simultaneously, is to be able to define new lemmas and definitions at any time during the construction of a proof. This is useful, since if the proof is not fully carried out on paper beforehand, we may not know exactly which lemmas and definitions are needed. Moreover, we want to be able to use the lemmas before they are actually proved, in order to really construct proofs as well as program in a top-down fashion. Therefore we will allow the constants in the incomplete definitions to be used, even though they do not have a definition as yet. Since any constant has a given type, using the type information of the constant corresponds to assuming we have a proof of that proposition or a program of that type.

As already mentioned, incomplete theory definitions contain type checking problems which we want to solve. Since the type checking problems can be partially solved by unification, incomplete theory definitions can as well. Thus, the scratch area also contains a partially solved unification problem, which contains all the placeholders in the incomplete theory definitions. We require all placeholders to be distinct, since we associate one unification problem to the entire scratch area rather than to each type checking problem. We need to do this, since the definitions (i.e. constants) may depend on each other, and to get a valid theory in the end, the definitions must be ordered. Hence, we must consider the dependency between the constants in a scratch area and not only within the definitions. A convenient way is to use the dependency graph we already have for the placeholders, and include the scratch area constants in the graph.

### 9.2.1 Operations on the scratch area

The important operations on incomplete theory definitions are the operations on type checking problems, i.e. insert and delete. Besides these, there are operations constructing the body of a new definition, that is adding a new constant with a given type, creating a pattern of an implicit constant and splitting a pattern in cases. Finally we have operations to move a completed definition to the theory, move a theory definition to the scratch area (for modification) and to remove entire definitions.

New placeholders are always generated by ALF, and by that we know they will be different from other placeholders. This "filling-up" of placeholder-arguments which is used in the refine tactic, is applied everywhere in an incomplete term which is inserted for a placeholder. When a term $c(a_k, \ldots, a_n)$ has not enough arguments to match the arity of the expected type, new placeholders are filled as additional arguments. At present, they are always added in front of the given arguments, that is

$$c(?_1, \ldots, ?_j, a_k, \ldots, a_n),$$

since it is a simple and rather useful strategy. The reason is that all the monomorphic type information usually appear as the first arguments to functions, and these are the least interesting arguments. Moreover, since these arguments can almost always be instantiated automatically by unification, we can ignore them and use functions as if they were implicitly polymorphic in the sense of most functional programming languages. This together with the mechanism in the window interface of *hiding arguments*, gives the possibility of increasing the readability of proof terms.

However, there is no reason why not any argument could be hidden. This can be accomplished if either the interface fills in placeholders for the hidden arguments, or the proof engine has access to the hiding information. Our way of handling hidden arguments is simply a matter of not printing the argument, which is different from the one used in LEGO ([LP92]). In LEGO there are two different function formers, one with hidden arguments and one with visible arguments. Hence, two different kind of applications are needed as well.

We have the following operations on a scratch area:

**Insert** Takes a placeholder and an incomplete term, and applies the insert operation after filling the term up with the proper number of placeholder arguments. Placeholders are unique in the incomplete definitions, so the type checking problem is determined by the placeholder.

**Delete** Takes a search path to a (sub)-term to one of the type checking problems in the definitions, applies the delete operation, and type checks all the definitions in the scratch area in order. We must must type check all definitions, since they may depend on each other in an intrinsic way.

**New definition** We can add a new constant, by giving a name, a type and in the case of an explicit definition also a context. The type (and context) is checked to be correct relative the theory and the scratch area definitions. If the constant is a constructor of a primitive constant definition, the set-constructor must be given as well. Specific requirements such as the result type of constructors are also checked.

**Construct patterns** The pattern rules of an implicit constant is constructed in the following way: first a general pattern with only pattern variables are generated, then the pattern can be split by selecting a pattern variable to analyse. The algorithm described in [Coq92] will then generate a set of non-overlapping, exhaustive patterns with respect to case analysis on the chosen variable, which replaces the split pattern. The splitting of a pattern can be done if the type of the variable is an inductively defined set, and the right-hand side of the pattern rule is undefined.

**Delete patterns** The pattern rules of an implicit constant can be deleted, but all of them have to be deleted simultaneously since otherwise the collection of patterns would not remain exhaustive. The right-hand sides on the other hand, can of course be edited by the ordinary delete operation.

**Delete definition** This operation deletes an entire constant definition, and it is allowed if the constant is not used elsewhere in the scratch area.

**Move to theory** A definition can be moved to the theory, if it is completed and does not depend on any other definition in the scratch area.

**Move to scratch** A (complete) definition can be moved back to the scratch area, where it can be modified again. The requirement is that no other definitions in the theory depend on this definition.

**Save, open and import** Theories and scratch areas can be saved to files and loaded back into ALF.

There are some other features implemented in ALF, which are all of a rather experimental character.

- Analogously to explicit constant definitions, which are abbreviations of terms, we have also abbreviations of types. Moreover, there are type placeholders, which means that types can be built up incrementally in the same way as terms. Accordingly, we have type formation problems corresponding to type checking problems and the basic operations insert and delete for incomplete types. Hence, we have in practice type inference for all terms whose type can be computed.

- Depending on a placeholder's type, there is a subset of all constants which can possibly be used to refine that placeholder. To compute this set of matching constants and variables from its local context, each constants type would have to be unified with the expected type of the placeholder and this is not realistic if the theory is large. However, one can compute an incomplete set of matching constants by doing a simple matching of the types which is feasible and this set contains very often the desired constant.

- Another feature which is important in practice, is to be able to massage the form of terms and types to equivalent forms. That is, sub-terms and sub-types can be replaced by their corresponding head normal or normal form, or an explicit constant can be unfolded (replaced by its definition). For instance, recall the example from section 2.2, where we proved associativity of addition. We have to solve the two cases

  (1) $Id(N, add(add(m,n), 0), add(m, add(n, 0)))$
  (2) $Id(N, add(add(m,n), s(k')), add(m, add(n, s(k'))))$

  and it is much easier to see how to solve these, if the arguments of $Id$ can be reduced. We get instead the two cases

  (1) $Id(N, add(m,n), add(m,n))$
  (2) $Id(N, s(add(add(m,n), k')), s(add(m, add(n, k'))))$

  where it is obvious that the first case is solved by reflexivity and the second by a congruence rule and the induction hypothesis.

For a more detailed description of the operations, see the manual ([AGNv94]).

# Chapter 10

# Summary and related works

We have described the implementation of the current version of ALF, earlier partly presented in [Mag92], [Mag93] and [MN94]. The overall ideas of the system (see [Nor93], [CNSvS94], [MN94]) is similar to the previous version of ALF [ACN90]. One difference is that the previous version was based on a combination of generalised type systems [Bar91] and Martin-Löf's monomorphic type theory [NPS90], whereas the current ALF is solely based on Martin-Löf's type theory extended with explicit substitution [Tas93].

The main contribution of the author is the type checking algorithm for incomplete (and complete) terms, and the design of the local undo operation. We have seen that the operations used to edit proofs are defined in terms of the two basic operations (insert and delete) on incomplete terms. Hence, proof editing is reduced to type checking incomplete proof objects. The type checking algorithm is presented for Martin-Löf's framework, but the same ideas can be adopted for other formal systems of a similar kind.

The type checking algorithm for complete terms is proved sound and complete with respect to the substitution calculus of type theory, and the extension to incomplete terms is proved sound. We have also indicated some ways of possibly showing completeness for the extension.

There are several other proof assistants based on some variant of type theory, for instance Coq [DFH$^+$91], LEGO [LP92], Constructor [HA90] and NuPRL [Con86]. Coq is based on the Calculus of Inductive Constructions, which is the Calculus of Constructions [CH88] extended with a schema of inductive definitions [PM93]. LEGO is a proof assistant for the Extended Calculus of Constructions which allows inductive types to be defined as extensions to the theory. The Constructor system is a partly automated proof assistant for generalised type systems, which includes the Calculus of Constructions and LF [HHP87] as sub-systems. In [vBJMP94], type checking algorithms for Pure Type Systems are proved correct which is a justification of the type checking algorithms for complete terms implemented in LEGO, Constructor and LF. The proof synthesis method used in Constructor is an incomplete method, but can be generalised to a complete method as shown in [Dow93]. NuPRL is a proof development system based on a variant of Martin-Löf's polymorphic type theory. In NuPRL, proof search strategies can be programmed by the user.

All of these systems are more or less inspired by the pioneer systems in the area of machine checked formal mathematics, that is AUTOMATH [dB68] and later LCF [GMW79] which was the first tactic-based proof construction system, and [Pet84], which was the first implementation of Martin-Löf's type theory.

The main difference between ALF compared to these proof assistants is that in ALF proofs objects are manipulated directly. Coq, LEGO, Constructor and NuPRL are all tactic-based. As we have seen, the basic tactics intro and refine can both be defined in terms of the insert operation on an incomplete proof object in ALF. The operations insert and delete on incomplete proof objects give rise to a flexible way of editing proofs, since any part of the proof object can be worked on, and any part can be deleted if necessary. The scratch area allows the user to have several incomplete proofs simultaneously, add new definitions at any time and edit them in the order of his/her choice, which adds to the flexibility.

One advantage of tactic-based systems is that tactics give the possibility to systematise similar kinds of reasoning. We recognise this as a deficiency in ALF, but the idea of combining tactics fits poorly with the idea of direct manipulation of proof objects. However, systematisation of reasoning is not a priori tied to the idea of tactics. Instead, we need to find new approaches of interaction with proofs which provide such a possibility of systematisation.

# Appendix A

# Substitution calculus rules

We will here briefly state the rules of the calculus, which are being referred to in the correctness and completeness proofs. The entire set of rules in the calculus and semantic justifications can be found in [Tas93].

**Context rules**

The two formation rules for contexts are

$$\frac{}{[\,] : \mathit{Context}} \ \text{ConNil}$$

$$\frac{\Gamma : \mathit{Context} \quad \Gamma \vdash \alpha : \mathit{Type}}{[\Gamma, x : \alpha] : \mathit{Context}} \ \text{ConExt} \ (x \notin \mathit{Dom}(\Gamma))$$

And the corresponding rules expressing the sub context relation

$$\frac{\Delta : \mathit{Context}}{[\,] \preceq \Delta} \ \text{SubNil}$$

$$\frac{\Gamma \preceq \Delta \quad \Delta \vdash x : \alpha}{[\Gamma, x : \alpha] \preceq \Delta} \ \text{SubExt}$$

The above rule of sub context extension was presented by Per Martin-Löf [1] and it serves better our purposes since it allows the types of variables

[1] The rule was presented on a workshop in Helsinki, September 1993

to be convertible to each other, rather than identical as in Tasistro's formulation:

$$\frac{\Gamma \preceq \Delta}{[\Gamma, x : \alpha] \preceq \Delta}\ \text{SubExt'}\ (x : \alpha \in \Delta)$$

**Substitution rules**

$$\frac{\Gamma : \mathit{Context}}{\Gamma \vdash \{\} : \Gamma}\ \text{Id}$$

$$\frac{\Gamma \vdash \gamma : \Delta \quad \Gamma \vdash a : \alpha\gamma}{\Gamma \vdash \{\gamma, x{:=}a\} : [\Delta, x : \alpha]}\ \text{Upd}$$

$$\frac{\Delta \vdash \gamma : \Gamma \quad \Psi \vdash \delta : \Delta}{\Psi \vdash \gamma\delta : \Gamma}\ \text{Comp}$$

$$\frac{\Psi \vdash \delta : \Delta \quad \Psi \preceq \Gamma}{\Gamma \vdash \delta : \Delta}\ \text{Thin}$$

$$\frac{\Gamma \vdash \delta : \Delta \quad \Psi \preceq \Delta}{\Gamma \vdash \delta : \Psi}\ \text{T-Thin}$$

**Type rules**

$$\frac{\Gamma : \mathit{Context}}{\Gamma \vdash \mathit{Set} : \mathit{Type}}\ \text{SetForm}$$

$$\frac{\Gamma \vdash \alpha : \mathit{Type} \quad \Gamma \vdash \beta : \alpha \to \mathit{Type}}{\Gamma \vdash \alpha \to \beta : \mathit{Type}}\ \text{FunForm}$$

$$\frac{\Gamma \vdash \beta : \alpha \to \mathit{Type} \quad \Gamma \vdash a : \alpha}{\Gamma \vdash \beta a : \mathit{Type}}\ \text{App}$$

$$\frac{\Gamma \vdash \alpha : \mathit{Type} \quad \Delta \vdash \gamma : \Gamma}{\Delta \vdash \alpha\gamma : \mathit{Type}}\ \text{Subst}$$

$$\frac{\Gamma \vdash \alpha : \mathit{Type} \quad \Gamma \preceq \Delta}{\Delta \vdash \alpha : \mathit{Type}}\ \text{Thin}$$

**Family rules**

$$\frac{\Gamma : \mathit{Context}}{\Gamma \vdash \mathit{El} : \mathit{Set} \rightarrow \mathit{Type}}\ \text{ElForm}$$

$$\frac{[\Gamma, x : \alpha'] \vdash \alpha : \mathit{Type}}{\Gamma \vdash [x]\alpha : \alpha' \rightarrow \mathit{Type}}\ \text{Abs}$$

**Term rules**

$$\frac{}{\Gamma \vdash x : \alpha}\ \begin{array}{l}\text{Ass}\\ (x : \alpha \in \Gamma)\end{array}$$

$$\frac{}{\Gamma_c \vdash c : \alpha_c}\ \begin{array}{l}\text{Const}\\ (c : \alpha_c\ \Gamma_c \in \Sigma)\end{array}$$

$$\frac{[\Gamma, x : \alpha] \vdash b : \alpha'}{\Gamma \vdash [x]b : \alpha \rightarrow [x]\alpha'}\ \text{Abs}$$

$$\frac{\Gamma \vdash f : \alpha \rightarrow \beta \quad \Gamma \vdash a : \alpha}{\Gamma \vdash fa : \beta a}\ \text{App}$$

$$\frac{\Gamma \vdash a : \alpha \quad \Delta \vdash \gamma : \Gamma}{\Delta \vdash a\gamma : \alpha\gamma}\ \text{Subst}$$

$$\frac{\Gamma \vdash a : \alpha \quad \Gamma \vdash \alpha = \alpha' : \mathit{Type}}{\Gamma \vdash a : \alpha'}\ \text{TConv}$$

$$\frac{\Gamma \vdash a : \alpha \quad \Gamma \preceq \Delta}{\Delta \vdash a : \alpha}\ \text{Thin}$$

**Type Equality rules**

$$\frac{\Gamma \vdash \alpha = \alpha' : \mathit{Type} \quad \Gamma \vdash \beta = \beta' : \alpha \to \mathit{Type}}{\Gamma \vdash \alpha \to \beta = \alpha' \to \beta' : \mathit{Type}}\ \text{FunEq}$$

$$\frac{\Gamma \vdash \beta_1 = \beta_2 : \alpha \to \mathit{Type} \quad \Gamma \vdash a_1 = a_2 : \alpha}{\Gamma \vdash \beta_1 a_1 = \beta_2 a_2 : \mathit{Type}}\ \text{AppEq}$$

$$\frac{\Gamma \vdash \alpha : \mathit{Type} \quad \Delta \vdash \gamma = \delta : \Gamma}{\Delta \vdash \alpha\gamma = \alpha\delta : \mathit{Type}}\ \text{SubstEq}$$

$$\frac{[\Gamma, x : \alpha'] \vdash \alpha : \mathit{Type} \quad \Delta \vdash \gamma : \Gamma \quad \Delta \vdash a : \alpha'\gamma}{\Delta \vdash (([x]\alpha)\gamma)a = \alpha\{\gamma, x{:=}a\} : \mathit{Type}}\ \beta\text{Subst}$$

$\beta$-rule is derived:

$$\frac{[\Gamma, x : \alpha'] \vdash \alpha : \mathit{Type} \quad \Gamma \vdash a : \alpha'}{\Gamma \vdash ([x]\alpha)a = \alpha\{x{:=}a\} : \mathit{Type}}\ \beta$$

$$\frac{\Psi \vdash \gamma : \Gamma}{\Psi \vdash (\mathit{Set})\gamma = \mathit{Set} : \mathit{Type}}\ \text{SetSubst}$$

$$\frac{\Gamma \vdash \alpha : \mathit{Type}}{\Gamma \vdash \alpha\{\} = \alpha : \mathit{Type}}\ \text{Empty}$$

$$\frac{\Gamma \vdash \alpha : \mathit{Type} \quad \Gamma \vdash \beta : \alpha \to \mathit{Type} \quad \Delta \vdash \gamma : \Gamma}{\Delta \vdash (\alpha \to \beta)\gamma = \alpha\gamma \to \beta\gamma : \mathit{Type}}\ \text{FunDistr}$$

$$\frac{\Gamma \vdash \beta : \alpha \to \mathit{Type} \quad \Gamma \vdash \alpha : \mathit{Type} \quad \Delta \vdash \gamma : \Gamma}{\Delta \vdash (\beta a)\gamma = (\beta\gamma)(a\gamma) : \mathit{Type}}\ \text{AppDistr}$$

$$\frac{\Delta \vdash \alpha : \mathit{Type} \quad \Gamma \vdash \delta : \Delta \quad \Psi \vdash \gamma : \Gamma}{\Psi \vdash (\alpha\delta)\gamma = \alpha(\delta\gamma) : \mathit{Type}}\ \text{Assoc}$$

**Family Equality rules**

$$\frac{\Psi \vdash \gamma : \Gamma}{\Psi \vdash (El)\gamma = El : Set \rightarrow Type}\ \text{ElSubst}$$

$$\frac{[\Gamma, x : \alpha] \vdash \beta x = \beta' x \ : \ Type}{\Gamma \vdash \beta = \beta' : \alpha \rightarrow Type}\ \begin{array}{l}\text{Ext}\\ (x \notin Dom(\Gamma))\end{array}$$

The $\eta$-rule is derived:

$$\frac{\Gamma \vdash \beta : \alpha \rightarrow Type}{\Gamma \vdash \beta = [x](\beta x) : \alpha \rightarrow Type}\ \begin{array}{l}\eta\\ (x \notin Dom(\Gamma))\end{array}$$

$$\frac{\Delta \vdash \beta : \alpha \rightarrow Type \quad \Gamma \vdash \delta : \Delta \quad \Psi \vdash \gamma : \Gamma}{\Psi \vdash (\beta\delta)\gamma = \beta(\delta\gamma) : \alpha(\delta\gamma) \rightarrow Type}\ \text{Assoc}$$

**Term Equality rules**

$$\frac{\Gamma \vdash f = g : \alpha \rightarrow \beta \quad \Gamma \vdash a = b : \alpha}{\Gamma \vdash fa = gb : \beta a}\ \text{AppEq}$$

$$\frac{[\Gamma, x : \alpha] \vdash fx = gx : \beta x}{\Gamma \vdash f = g : \alpha \rightarrow \beta}\ \begin{array}{l}\text{Ext}\\ (x \notin Dom(\Gamma))\end{array}$$

$$\frac{[\Gamma, x : \alpha'] \vdash b : \alpha \quad \Delta \vdash \gamma : \Gamma \quad \Delta \vdash a : \alpha'\gamma}{\Delta \vdash (([x]b)\gamma)a = b\{\gamma, x{:=}a\} : \alpha\{\gamma, x{:=}a\}}\ \beta\text{Subst}$$

$\beta$-rule is derived:

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

$$\frac{\Gamma \vdash \delta : \Delta \quad \Gamma \vdash a : \alpha\delta}{\Gamma \vdash x\{\delta, x{:=}a\} = a : \alpha\delta}\ \beta_1$$

$$\frac{\Gamma \vdash \delta : \Delta \quad \Gamma \vdash a : \alpha\delta}{\Gamma \vdash y\{\delta, x{:=}a\} = y\delta : \alpha'\delta} \; {}^{\beta_2}_{(y \,:\, \alpha' \in \Delta)}$$

$$\frac{\Gamma \vdash a : \alpha}{\Gamma \vdash a\{\} = a : \alpha\{\}} \; \text{Empty}$$

$$\frac{\Gamma \vdash f : \alpha{\to}\beta \quad \Gamma \vdash a : \alpha \quad \Delta \vdash \gamma : \Gamma}{\Delta \vdash (fa)\gamma = (f\gamma)(a\gamma) : (\beta a)\gamma} \; \text{AppDistr}$$

$$\frac{\Delta \vdash a : \alpha \quad \Gamma \vdash \delta : \Delta \quad \Psi \vdash \gamma : \Gamma}{\Psi \vdash (a\delta)\gamma = a(\delta\gamma) : \alpha(\delta\gamma)} \; \text{Assoc}$$

**Substitution Equality rules**

$$\frac{\Gamma \vdash \delta : \Delta \quad \Gamma \vdash a : \alpha\delta}{\Gamma \vdash \{\delta, x{:=}a\} = \delta : \Delta} \; {}^{\beta_0}_{(x \notin \Delta)}$$

$$\frac{\Gamma \vdash \delta : \Delta}{\Gamma \vdash \{\}\delta = \delta : \Delta} \; \text{Empty}_L \qquad \frac{\Gamma \vdash \delta : \Delta}{\Gamma \vdash \delta\{\} = \delta : \Delta} \; \text{Empty}_R$$

$$\frac{\Gamma \vdash \delta : \Delta \quad \Gamma \vdash a : \alpha\delta \quad \Psi \vdash \gamma : \Gamma}{\Psi \vdash (\{\delta, x{:=}a\})\gamma = \{\delta\gamma, x{:=}a\gamma\} : [\Delta, x : \alpha]} \; \text{Distr}$$

$$\frac{\Delta \vdash \theta : \Theta \quad \Gamma \vdash \delta : \Delta \quad \Psi \vdash \gamma : \Gamma}{\Psi \vdash (\theta\delta)\gamma = \theta(\delta\gamma) : \Theta} \; \text{Assoc}$$

Reflexivity, symmetry and transitivity hold for all equality relations.

# Appendix B

# Soundness proofs

## B.1 Soundness of type formation

**Proposition 2 (TF-soundness)** *Let $\Gamma$ be a valid context and $\alpha$ $\Gamma$-distinct and $\beta S$-normal.*

*If $TF(\Gamma, \alpha) \Rightarrow \xi$ and $\xi$ holds, then $\Gamma \vdash \alpha : Type$.*

***Proof:*** Induction on the structure of $\alpha$. Since $\alpha$ is assumed to be $\beta S$-normal, then $\alpha$ is of the form *Set*, *El*(A) or $\alpha \rightarrow \beta$. The preconditions are preserved in recursive calls as shown in lemma 1.1. Hence, we can do a simple induction:

*Set*:

$\Gamma \vdash Set : Type$ follows directly from *Set*-formation.

*El(A)*:

Assume $TF(\Gamma, El(A)) \Rightarrow \xi$ and $\xi$ holds. By the premiss of the the TF-El rule, we must have $GTE(\Gamma, A, Set) \Rightarrow \xi$. Then, by GTE-sound, we have

(1) $\Gamma \vdash A : Set$,

which gives us

$\Gamma \vdash El(A) : Type$

by (1) and application.

$\alpha \rightarrow \beta$ :

Assume $TF(\Gamma, \alpha \rightarrow \beta) \Rightarrow \xi$ and $\xi$ holds. Since $\alpha \rightarrow \beta$ is assumed to be $\beta S$-normal, we know that $\beta$ must be either the constant *El* or a family $[x]\alpha'$. If $\beta$ is *El*, then $\alpha$ must be *Set* and $\Gamma \vdash Set \rightarrow El : Type$ follows directly from *Set*- and *El*-formation. Otherwise we have by induction hypothesis that

(1) $\Gamma \vdash \alpha : Type$, and
(2) $[\Gamma, x : \alpha] \vdash \alpha' : Type$

and we can derive

$$\dfrac{\dfrac{(1)}{\Gamma \vdash \alpha : Type} \qquad \dfrac{\dfrac{(2)}{[\Gamma, x : \alpha] \vdash \alpha' : Type}}{\Gamma \vdash [x]\alpha' : \alpha \rightarrow Type}\,\text{Abs}}{\Gamma \vdash \alpha \rightarrow [x]\alpha' : Type}\,\text{FunForm}$$

□

**Lemma 2.1** *The preconditions of* TF *are preserved in recursive calls.*

***Proof:*** Assuming $\Gamma$ to be a valid context, we know that $[\Gamma, x : \alpha]$ is a valid extension of $\Gamma$ since $x$ is $\Gamma$-fresh (by precondition 3), and $\alpha$ is a type in $\Gamma$ by the first premiss of TF-Fun. It is easy to see that if $\alpha \rightarrow [x]\alpha'$ is $\beta S$-normal, then so is $\alpha$ and $\alpha'$, so precondition 2 is preserved. Finally, if $\alpha \rightarrow [x]\alpha'$ is $\Gamma$-distinct, then so is $\alpha$ and $\alpha'$ is distinct relative to the extended context $[\Gamma, x : \alpha]$ by the definition of $\Gamma$-distinct.

□

## B.2 Soundness of type checking

**Proposition 3 (GTE-soundness)**
*Let $\Gamma, \Delta$ be valid contexts and $\Gamma \vdash \alpha : Type$. Let $q$ be a $\beta S$-normal term (and not an abstraction in the CT-case) and let $q$ be $\Gamma$-distinct. Then we have the following*

$$\begin{cases} GTE(q, \alpha, \Gamma) \Rightarrow \xi & \wedge \quad \xi \textit{ holds} \quad \supset \quad \Gamma \vdash q : \alpha \\ FC(q, \Delta, \Gamma) \Rightarrow \xi & \wedge \quad \xi \textit{ holds} \quad \supset \quad \Gamma \vdash q : \Delta \\ CT(q, \Gamma) \Rightarrow \langle \xi, \alpha \rangle & \wedge \quad \xi \textit{ holds} \quad \supset \quad \Gamma \vdash q : \alpha \end{cases}$$

*where $\xi$ is a well-formed list of type equations.*

***Proof:*** Induction on the structure of $q$. The preconditions hold in the recursive calls by lemma 2.2.

*GTE-Var:*

Assume $GTE(x, \alpha, \Gamma) \Rightarrow \xi$ and that $\xi$ holds. We have that

(1) $\Gamma \vdash x : \alpha'$

since $x : \alpha' \in \Gamma$ (by the side-condition of GTE-Var) and $\Gamma$ is a valid context (by precondition). Also, $\alpha$ is a type by precondition of GTE so $\xi$ is well-formed. By assumption we have that $\xi$ holds which means

(2) $\Gamma \vdash \alpha = \alpha' : \mathit{Type}$.

Thus, we can derive

$\Gamma \vdash x : \alpha$

by (1), (2) and type conversion.

*GTE-Const:*

Analogous to GTE-Var, but we have $\Sigma$ valid, $c : \alpha' \in \Sigma$ and by thinning ($\Gamma$ valid) we get $\Gamma \vdash c : \alpha'$.

*GTE-Subst:*

Assume $GTE(c\gamma, \alpha, \Gamma) \Rightarrow \xi$ and that $\xi$ holds. Then we know by GTE-Subst that $\xi = \xi' \mathbin{@} [\langle \alpha_c\gamma, \alpha, \Gamma\rangle]$, for some $\xi'$. By ind. hyp. we have that $\xi'$ is well-formed and that

(1) $\Gamma \vdash \gamma : \Gamma_c$.

We know that $\Gamma \vdash \alpha : \mathit{Type}$ (by precondition 3) and ($\star$) $\Gamma_c \vdash \alpha_c : \mathit{Type}$ since $\Sigma$ is valid, and $c$ has type $\alpha_c$. Thus, we get $\Gamma \vdash \alpha_c\gamma : \mathit{Type}$ by (1),($\star$) and the substitution rule, so $\xi$ is well-formed. Moreover, since $\xi$ holds, we know

(2) $\Gamma \vdash \alpha_c\gamma = \alpha : \mathit{Type}$.

which gives the derivation

$$\dfrac{\dfrac{\dfrac{\Sigma \text{ valid}}{\Gamma_c \vdash c : \alpha_c} \quad \dfrac{(1)}{\Gamma \vdash \gamma : \Gamma_c}}{\Gamma \vdash c\gamma : \alpha_c\gamma}\,\text{Subst} \qquad \dfrac{(2)}{\Gamma \vdash \alpha_c\gamma = \alpha : \mathit{Type}}}{\Gamma \vdash c\gamma : \alpha}\,\text{TConv}$$

*GTE-Abs:*

Assume $GTE([x]b, \alpha \to \beta, \Gamma) \Rightarrow \xi$ and that $\xi$ holds. Now, due to the restriction of $[x]b$ to be $\Gamma$-distinct, we know that $x$ is $\Gamma$-fresh so $\Gamma$ can be extended without name clash. The GTE-Abs rule gives that

$GTE(b, \beta x, [\Gamma, x : \alpha]) \Rightarrow \xi$,

so by ind. hyp. we have

(1) $[\Gamma, x : \alpha] \vdash b : \beta x$

and that $\xi$ is well-formed. By precondition 3 we know that

$\Gamma \vdash \alpha \to \beta : \mathit{Type}$,

which implies

(2) $\Gamma \vdash \alpha : \mathit{Type}$, and
(3) $\Gamma \vdash \beta : \alpha \to \mathit{Type}$.

Now we can build the following derivation

$$\dfrac{\dfrac{\dfrac{(1)}{[\Gamma, x : \alpha] \vdash b : \beta x}}{\Gamma \vdash [x]b : \alpha \to [x](\beta x)} \qquad \dfrac{\dfrac{(2)}{\Gamma \vdash \alpha : \mathit{Type}} \quad \dfrac{\dfrac{(3)}{\Gamma \vdash \beta : \alpha \to \mathit{Type}}}{\Gamma \vdash \beta = [x](\beta x) : \alpha \to \mathit{Type}}}{\Gamma \vdash \alpha \to \beta = \alpha \to [x](\beta x) : \mathit{Type}}}{\Gamma \vdash [x]b : \alpha \to \beta}$$

where the $\eta$-rule is properly applied, since $x$ is $\Gamma$-fresh.

*GTE-App:*

Assume $GTE(fa, \alpha, \Gamma) \Rightarrow \xi$ and $\xi$ holds, where $\xi$ must be of the form $\xi_1 @ \xi_2 @ [\langle \beta e, \alpha, \Gamma \rangle]$. Since $\xi$ holds, so does $\xi_1$, so by CT-lemma we know $(\star)$ $\Gamma \vdash \alpha' \to \beta : \mathit{Type}$, which means that the induction hypothesis can be applied to $CT(f, \Gamma) \Rightarrow \langle [\,], \alpha' \to \beta \rangle$ (since precondition 2 holds). Thus, we have by ind. hyp.

(1) $\Gamma \vdash f : \alpha' \to \beta$, and
(2) $\Gamma \vdash a : \alpha'$.

We know that $\langle \beta a, \alpha, \Gamma \rangle$ is well-formed, since we can derive $\Gamma \vdash \alpha : \mathit{Type}$ (by precondition 2) and by $(\star)$ and (2) we can derive $\Gamma \vdash \beta a : \mathit{Type}$. Further, since $\xi$ holds we have that

(3) $\Gamma \vdash \beta a = \alpha' : \mathit{Type}$

and we get the following derivation

$$\dfrac{\dfrac{\dfrac{(1)}{\Gamma \vdash f : \alpha' \to \beta} \quad \dfrac{(2)}{\Gamma \vdash a : \alpha'}}{\Gamma \vdash fa : \beta a}\,\text{App} \qquad \dfrac{(3)}{\Gamma \vdash \beta a = \alpha' : \mathit{Type}}}{\Gamma \vdash fa : \alpha'}\,\text{TConv}$$

*CT-Var:*

$\Gamma \vdash x : \alpha$ is immediate by the side condition $x : \alpha \in \Gamma$ since $\Gamma$ is a valid context.

*CT-Const:*

By the side condition $c : \alpha \in \Sigma$ and the thinning rule $([\,] \preceq \Gamma)$, we have $\Gamma \vdash c : \alpha$.

*CT-Subst:*

By assumption we have $CT(c\gamma, \Gamma) \Rightarrow \langle \xi, \alpha_c\gamma \rangle$, where $\xi$ holds, so by ind. hyp. we have

(1) $\Gamma \vdash \gamma : \Gamma_c$, and by the side condition we get (since $\Sigma$ is valid)
(2) $\Gamma_c \vdash c : \alpha_c$.

Then applying the substitution rule to (1) and (2), we yield the desired relation

$$\Gamma \vdash c\gamma : \alpha_c\gamma.$$

*CT-App:*

Assume $CT(fa, \Gamma) \Rightarrow \langle \xi, \beta a \rangle$ and assume $\xi$ holds. Then we will have $CT(f, \Gamma) \Rightarrow \langle \xi, \alpha \to \beta \rangle$ by the CT-App rule. Since $\xi$ is holds, CT-lemma gives us $\Gamma \vdash \alpha \to \beta : \mathit{Type}$ which implies $\Gamma \vdash \alpha : \mathit{Type}$. Thus, precondition 2 holds and the induction hypothesis applies also for the second premiss, yielding

(1) $\Gamma \vdash f : \alpha \to \beta$, and
(2) $\Gamma \vdash a : \alpha$,

which gives directly gives

$$\Gamma \vdash fa : \beta a$$

by application.

*FC-Empty:*

We want to show $\Gamma \vdash \{\} : [\,]$. Since $\Gamma$ is a valid context, we have directly the derivation

$$\dfrac{\dfrac{}{[\,] \vdash \{\} : [\,]}\,\text{Id} \qquad \dfrac{\Gamma : \mathit{Context}}{[\,] \preceq \Gamma}\,\text{SubNil}}{\Gamma \vdash \{\} : [\,]}\,\text{Thin}$$

*FC-Ext:*

Assume $FC(\{\gamma, x{:=}a\}, [\Delta, x : \alpha], \Gamma) \Rightarrow \xi$ and assume $\xi$ holds, where $\xi =$

$\xi_1 @ \xi_2$. By ind. hyp we have

(1) $\Gamma \vdash \gamma : \Delta$, and
(2) $\Gamma \vdash a : \alpha\gamma$,

so by using the updating rule, we get the derivation

$$\frac{\dfrac{(1)}{\Gamma \vdash \gamma : \Delta} \qquad \dfrac{(2)}{\Gamma \vdash a : \alpha\gamma}}{\Gamma \vdash \{\gamma, x{:=}a\} : [\Delta, x : \alpha]}\ \text{Upd}$$

□

**Lemma 3.1 (CT-lemma)**
*Let $f$ be a $\beta S$-normal term which is not an abstraction, and let $\Gamma$ be a valid context.*

*If $CT(f,\Gamma) \Rightarrow \langle \xi, \alpha\rangle$ and $\xi$ holds, then $\Gamma \vdash \alpha$ : Type.*

***Proof:*** Induction on the structure of $f$.

- $$\frac{}{CT(x,\Gamma) \Rightarrow \langle [\,], \alpha\rangle}\ \begin{array}{l}\text{CT-Var}\\ (x : \alpha \in \Gamma)\end{array}$$

  Since $\Gamma$ is valid by precondition 1, we have $\Gamma \vdash \alpha$ : *Type*.

- $$\frac{}{CT(c,\Gamma) \Rightarrow \langle [\,], \alpha\rangle}\ \begin{array}{l}\text{CT-Const}\\ (\ c : \alpha \in \Sigma\ )\end{array}$$

  Since $\Sigma$ is valid by assumption, we have $\vdash \alpha$ : *Type*, and thinning gives us $\Gamma \vdash \alpha$ : *Type*.

- $$\frac{FC(\gamma, \Gamma_c, \Gamma) \Rightarrow \xi}{CT(c\gamma,\Gamma) \Rightarrow \langle \xi, \alpha_c\gamma\rangle}\ \begin{array}{l}\text{CT-Subst}\\ (\Gamma_c \vdash c : \alpha_c \in \Sigma)\end{array}$$

  Since $\Sigma$ is valid, we have that $\Gamma \vdash \alpha_c$ : *Type*, and if $\xi$ holds, we have $\Gamma \vdash \gamma : \Gamma_c$, so by substitution we get $\Gamma \vdash \alpha_c\gamma$ : *Type*.

- $$\frac{CT(f,\Gamma) \Rightarrow \langle \xi_1, \alpha \rightarrow \beta\rangle \quad GTE(a, \alpha, \Gamma) \Rightarrow \xi_2}{CT(fa,\Gamma) \Rightarrow \langle \xi_1 @ \xi_2, \beta a\rangle}\ \text{CT-App}$$

  If $\xi_1$ holds, we have by ind. hyp. that (1) $\Gamma \vdash \alpha \rightarrow \beta$ : *Type*, which implies $\Gamma \vdash \alpha$ : *Type*. Since now the precondition of $GTE(a, \alpha, \Gamma) \Rightarrow \xi_2$ is fulfilled, we know that if $\xi_2$ also holds, we have that (2) $\Gamma \vdash a : \alpha$. The application rule applied on (1) and (2) gives the desired property, namely $\Gamma \vdash \beta a : Type$.

$\Box$

**Lemma 3.2** *If $\xi$ holds, then the preconditions of* GTE,FC *and* CT *are preserved in recursive calls.*

***Proof:*** Since a term is $\beta S$-normal when all its parts are and the recursive calls are on structurally smaller terms, precondition 3 is obvious. We have the same situation for the $\Gamma$-distinct property, so precondition 4 is preserved.

The context $\Gamma$ remains constant except in the GTE-Abs rule, so we need to check that $[\Gamma, x : \alpha]$ is a valid extension of $\Gamma$. This is the case, since $x$ is $\Gamma$-fresh by precondition 4, and $\Gamma \vdash \alpha \rightarrow \beta : Type$ holds due to precondition 2, which implies that $\Gamma \vdash \alpha : Type$. Thus, the extension is valid.

Finally, we need to show that $\Gamma \vdash \alpha : Type$ holds for all recursive calls. In the GTE-Abs rule, we know

(1) $\Gamma \vdash \alpha \rightarrow \beta : Type$

and since $[\Gamma, x : \alpha]$ is a valid context, we get

(2) $[\Gamma, x : \alpha] \vdash x : \alpha$

by the assumption rule, which gives

$[\Gamma, x : \alpha] \vdash \beta x : Type$ by (1),(2) and the application rule.

In GTE-App, CT-Subst and CT-App the condition holds by the CT-lemma. Left to justify is the FC-Ext rule, where $\Delta$ acts as the type of the substitution. If $[\Delta, x : \alpha]$ is a valid context, then so is $\Delta$, and if $\xi_1$ is consistent we know

(3) $\Gamma \vdash \gamma : \Delta$.

Also, we have

(4) $\Delta \vdash \alpha : Type$

(since $[\Delta, x : \alpha]$ is a valid), so the substitution rule applied to (3) and (4) gives us $\Gamma \vdash \alpha\gamma : Type$.

$\Box$

## B.3 Soundness of type conversion

**Proposition 4 (TSimple-soundness)**
*Let $\xi$ be a well-formed list of type equations.*

*If $TSimple(\xi) \Rightarrow \mathcal{C}$, then $\mathcal{C}$ is well-formed*

*and*

*if $\mathcal{C}$ holds, then $\forall\langle\alpha, \alpha', \Gamma\rangle \in \xi \;.\; \Gamma \vdash \alpha = \alpha' : Type$.*

***Proof:*** Induction on the length of the list $\xi$. (Uses TConv-sound). □

**Proposition 5 (TConv-soundness)**
*Let $\alpha$ and $\alpha'$ be valid types in context $\Gamma$.*

*If $TConv(\alpha,\alpha',\Gamma) \Rightarrow \mathcal{C}$ and $\mathcal{C}$ holds, then $\Gamma \vdash \alpha = \alpha' : Type$.*

***Proof:*** Induction on the length of the derivation $TConv(\alpha,\alpha',\Gamma) \Rightarrow \mathcal{C}$.

*TConv-Id:*

Since $\Gamma \vdash \alpha : Type$ holds, we get $\Gamma \vdash \alpha = \alpha : Type$ directly by reflexivity.

*TConv-El:*

Assume $TConv(El(A), El(B), \Gamma) \Rightarrow [\langle A, B, Set, \Gamma\rangle]$ and that the equation in $[\langle A, B, Set, \Gamma\rangle]$ holds. This implies directly

(1) $\Gamma \vdash A = B : Set$

and we can derive $\Gamma \vdash El(A) = El(B) : Type$ from (1) and the AppEq rule.

*TConv-Fun:*

Assume $TConv(\alpha \to \beta, \alpha' \to \beta', \Gamma) \Rightarrow \mathcal{C}_1 @ \mathcal{C}_2$ and that $\mathcal{C}_1 @ \mathcal{C}_2$ holds. By induction hypothesis we get

(1) $\Gamma \vdash \alpha = \alpha' : Type$, and
(2) $[\Gamma, z : \alpha] \vdash \beta z = \beta' z : Type$

so we get the derivation

$$\dfrac{\dfrac{(1)}{\Gamma \vdash \alpha = \alpha' : Type} \qquad \dfrac{\dfrac{(2)}{[\Gamma, z : \alpha] \vdash \beta z = \beta' z : Type}}{\Gamma \vdash \beta = \beta' : \alpha \to Type}\,\text{Ext}}{\Gamma \vdash \alpha \to \beta = \alpha' \to \beta' : Type}\,\text{FunForm}$$

*TConv-βSubst:*

Assume $TConv(\alpha_1, \alpha_2, \Gamma) \Rightarrow \mathcal{C}$ and that $\mathcal{C}$ holds. By induction hypothesis we get

(1) $\Gamma \vdash \alpha'_1 = \alpha'_2 : Type$

and by the $\xrightarrow{T\beta S}$-lemma we know

(2) $\Gamma \vdash \alpha_1 = \alpha'_1 : Type$, and
(3) $\Gamma \vdash \alpha_2 = \alpha'_2 : Type$

so we can easily derive

$\Gamma \vdash \alpha_1 = \alpha_2 : Type$

by transistivity from (1),(2) and (3).

$\square$

**Lemma 5.1 ($\xrightarrow{T\beta S}$-lemma)**
*If $\Gamma \vdash \alpha : Type$ and $\alpha \xrightarrow{T\beta S}_* \alpha'$, then $\Gamma \vdash \alpha = \alpha' : Type$.*

***Proof:*** We will show by case analysis on the $\xrightarrow{T\beta S}$-reduction that if $\alpha \xrightarrow{T\beta S} \alpha_1$ (one step reduction) then $\Gamma \vdash \alpha = \alpha_1 : Type$, and then $\Gamma \vdash \alpha = \alpha' : Type$ follows by transitivity for any reduction sequence $\alpha \xrightarrow{T\beta S} \cdots \xrightarrow{T\beta S} \alpha'$. First, we must note that if we have a derivation

$\Gamma \vdash \alpha\gamma : Type$

in the substitution calculus, then there exist a context $\Delta$ and derivations

$\Delta \vdash \alpha : Type$ $(MT1)$
$\Gamma \vdash \gamma : \Delta$ $(MT2)$

Moreover, if we have a derivation of

$\Gamma \vdash \beta a : Type$,

then there exists a type $\alpha$ such that the following derivations are possible

$\Gamma \vdash \beta : \alpha \to Type$ $(MT3)$
$\Gamma \vdash a : \alpha$ $(MT4)$

These $M$eta-$T$heory properties will be frequently used below.

*SubstSet:*

$Set\gamma \xrightarrow{T\beta S} Set$. By assumption we have $\Gamma \vdash Set\gamma : Type$, so by $(MT2)$ we have $\Gamma \vdash \gamma : \Delta$, which implies $\Gamma \vdash Set\gamma = Set : Type$ by the set-formation rule.

*SubstFun:*

$(\alpha \to \beta)\gamma \xrightarrow{T\beta S} (\alpha\gamma \to \beta\gamma)$. By assumption we have $\Gamma \vdash \alpha \to \beta\gamma : Type$, so by $MT2$ we have

(1) $\Gamma \vdash \gamma : \Delta$

By $MT3$ we get $\Delta \vdash \alpha \to \beta : Type$, which implies

(2) $\Gamma \vdash \alpha : Type$

and

(3) $\Gamma \vdash \beta : \alpha \to Type$.

Hence, we get

$\Gamma \vdash (\alpha \to \beta)\gamma = \alpha\gamma \to \beta\gamma : Type$

by (1),(2),(3) and the distributivity of a substitution inside a function.

*SubstSubst:*

$(\alpha\delta)\gamma \xrightarrow{T\beta S} \alpha(\delta\gamma)$. We have $\Gamma \vdash (\alpha\delta)\gamma : Type$ by assumption, so there are contexts $\Delta$ and $\Psi$ such that

(1) $\Gamma \vdash \gamma : \Delta$ (by $MT2$)
(2) $\Delta \vdash \delta : \Psi$ (by $MT2$ since $MT1$ gives us $(\star)$ $\Delta \vdash \alpha\delta : Type$), and
(3) $\Psi \vdash \alpha : Type$ (by $MT1$ from $(\star)$)

and we can derive

$\Gamma \vdash (\alpha\delta)\gamma = \alpha(\delta\gamma) : Type$

by (1),(2),(3) and associativity of substitutions.

*SubstApp:*

$(\beta a)\gamma \xrightarrow{T\beta S} (\beta\gamma)(\alpha\gamma)$. By assumption we have $\Gamma \vdash (\beta a)\gamma : Type$, so we get

(1) $\Gamma \vdash \gamma : \Delta$ (by $MT2$)

and we know by $MT1$ that $\Delta \vdash \beta a : Type$, which implies

(2) $\Delta \vdash \beta : \alpha \to Type$ (by $MT3$), and

(3) $\Delta \vdash a : \alpha$ by $MT4$

so we can derive

$$\Gamma \vdash (\beta a)\gamma = (\beta\gamma)(a\gamma) : Type$$

by (1),(2),(3) and the distributivity of $\gamma$.

*App$\beta$:*

$([x]\alpha)a \xrightarrow{T\beta S} \alpha\{x{:=}a\}$. We know by assumption that $\Gamma \vdash ([x]\alpha)a : Type$, so we have $\Gamma \vdash [x]\alpha : \alpha' \rightarrow Type$ for some $\alpha'$ (by $MT3$). This implies

(1) $[\Gamma, x : \alpha'] \vdash \alpha : Type$, and
(2) $\Gamma \vdash a : \alpha'$ by $MT4$

Applying the $\beta$-rule gives us the desired result.

*App$\beta$Subst:*

$(([x]\alpha)\gamma)a \xrightarrow{T\beta S} \alpha\{\gamma, x{:=}a\}$. Analogous to App$\beta$, using the $\beta$Subst-rule instead.

*AppElSubst:*

$(El\gamma)A \xrightarrow{T\beta S} El(A)$. Here, we assume $\Gamma \vdash (El\gamma)A : Type$, which by $MT3$ implies $\Gamma \vdash (El\gamma) : \alpha \rightarrow Type$ for some type $\alpha$. But since we know (by El-formation) that $\Delta \vdash El : Set \rightarrow Type$ for some context $\Delta$, we can see that $\alpha$ must be $Set\gamma$, where $\gamma$ satisfies

(1) $\Gamma \vdash \gamma : \Delta$.

Thus, we also have $\Gamma \vdash (El\gamma) : Set\gamma \rightarrow Type$, and

(2) $\Gamma \vdash A : Set\gamma$.

Now we can derive the following

$$\dfrac{\dfrac{\dfrac{(1)}{\Gamma \vdash \gamma : \Delta}}{\Gamma \vdash El\gamma = El : Set \rightarrow Type} \qquad \dfrac{\dfrac{(2)}{\Gamma \vdash A : Set\gamma} \qquad \dfrac{\dfrac{(1)}{\Gamma \vdash \gamma : \Delta}}{\Gamma \vdash Set\gamma = Set : Type}}{\Gamma \vdash A : Set}}{\Gamma \vdash (El\gamma)A = El(A) : Type}$$

*AppSubstSubst:*

$((\beta\delta)\gamma)a \xrightarrow{T\beta S} (\beta(\delta\gamma))a$. We have $\Gamma \vdash ((\beta\delta)\gamma)a : Type$, so we may assume the following derivations; $(\star)\Gamma \vdash (\beta\delta)\gamma : (\alpha\delta)\gamma \rightarrow Type$ and

(1) $\Gamma \vdash a : (\alpha\delta)\gamma$.

Further, we have that $(\star)$ implies

(2) $\Gamma \vdash \gamma : \Delta$, and also $\Delta \vdash \beta\delta : \alpha\delta \rightarrow Type$, which implies

(3) $\Delta \vdash \delta : \Psi$, and
(4) $\Psi \vdash \beta : \alpha \rightarrow Type$

so we get the derivation

$$\dfrac{\dfrac{\dfrac{(4)}{\Psi \vdash \beta : \alpha \rightarrow Type} \quad \dfrac{(3)}{\Delta \vdash \delta : \Psi} \quad \dfrac{(2)}{\Gamma \vdash \gamma : \Delta}}{\Gamma \vdash (\beta\delta)\gamma = \beta(\delta\gamma) : \alpha(\delta\gamma) \rightarrow Type} \quad \dfrac{(1)}{\Gamma \vdash a : (\alpha\delta)\gamma}}{\Gamma \vdash ((\beta\delta)\gamma)a = (\beta(\delta\gamma))a \ : \ Type}$$

□

**Lemma 5.2**
*The preconditions of* TConv *are preserved in recursive calls.*

***Proof:*** In TConv-Fun we know $\Gamma \vdash \alpha \rightarrow \beta : Type$, so $\alpha$ is a type in $\Gamma$ and $z$ is a fresh variable which makes this a valid extension of $\Gamma$, and hence, $\beta z$ is a type in the extended context. Type correctness of the reduced types in the TConv-$\beta$Subst rule is guaranteed by the $\xrightarrow{T\beta S}$-lemma.

□

## B.4 Soundness of term conversion

**Proposition 6 (Simple-soundness)**
*Let $\mathcal{C}$ be a well-formed list of term equations.*

$$\text{If } Simple\,(\mathcal{C}) \Rightarrow [\,],\ \text{then } \forall \langle a, b, \alpha, \Gamma\rangle \in \mathcal{C}\ .\ \Gamma \vdash a = b : \alpha.$$

***Proof:*** Induction on the length of the list $\mathcal{C}$. (Uses Conv-sound). □

**Proposition 7 (Conv-soundness)**
*If $a$ and $b$ are terms of type $\alpha$ in context $\Gamma$, then*

$$\begin{cases} Conv\,(a, b, \alpha, \Gamma) \Rightarrow [\,] & implies \quad \Gamma \vdash a = b : \alpha \\ HConv\,(a, b, \Gamma) \Rightarrow \langle [\,], \alpha\rangle & implies \quad \Gamma \vdash a = b : \alpha \end{cases}$$

***Proof:*** Induction on the length of the derivations of $Conv\,(a, b, \alpha, \Gamma) \Rightarrow [\,]$ and $HConv\,(a, b, \Gamma) \Rightarrow \langle [\,], \alpha\rangle$. By lemma 6.2 we know that the preconditions are preserved in recursive calls.

*Conv-Id:* By precondition we have $\Gamma \vdash a : \alpha$, so reflexivity gives $\Gamma \vdash a = a : \alpha$.

*Conv-fun:* By induction hypothesis we have $[\Gamma, z : \alpha] \vdash az = bz : \beta z$, so by extensionality we get $\Gamma \vdash a = b : \alpha \rightarrow \beta$.

*Conv-hnf:* By ind. hyp we have

(1) $\Gamma \vdash a' = b' : \alpha'$, and by lemma 6.1 we get
(2) $\Gamma \vdash a = a' : \alpha$ and
(3) $\Gamma \vdash b = b' : \alpha$.

since we know that $a$ and $b$ are of type $\alpha$ by the preconditions. Moreover, by meta theory assumption 1, we have

(4) $\Gamma \vdash \alpha = \alpha' : \mathit{Type}$

since $a'$ and $b'$ are not abstractions and $\beta S$-normal. We get

$\Gamma \vdash a = b : \alpha$

by (1),(2),(3) and transitivity.

*HConv-head:* We have that $\Gamma \vdash b : \alpha$ either by the assumption rule or the constant rule, so reflexivity gives $\Gamma \vdash b = b : \alpha$.

*HConv-app:* By induction hypothesis we have

(1) $\Gamma \vdash f = g : \alpha \rightarrow \beta$, and
(2) $\Gamma \vdash a = b : \alpha$

so by the AppEq rule we get $\Gamma \vdash fa = gb : \beta a$.

□

**Lemma 7.1** *If* $\Gamma \vdash a : \alpha$ *and* $a \xrightarrow{hnf} L(a')$, *then* $\Gamma \vdash a = a' : \alpha$.

***Proof:*** Case analysis on $a \xrightarrow{hnf} L(a')$.

*Hnf:* By reflexivity.

*Unfold:* We get $\Gamma \vdash a = a' : \alpha$ by lemma 6.1.1, $\Gamma \vdash a' = a'' : \alpha$ by the induction hypothesis, so by transitivity we have $\Gamma \vdash a = a'' : \alpha$.

*$\beta$Subst:* By lemma 6.1.2, ind. hyp. and transistivity.

*Match:* By lemma 6.1.3, ind. hyp. and transistivity.

*Irred:* By reflexivity.

□

**Lemma 7.1.1** *If* $\Gamma \vdash a : \alpha$ *and* $a \xrightarrow{\delta} a'$, *then* $\Gamma \vdash a = a' : \alpha$.

***Proof:*** Case analysis on $a \xrightarrow{\delta} a'$.

$c_e \xrightarrow{\delta} e$ :

Since $c_e = e \in \Sigma$ and $\Sigma$ is valid.

$c_e\gamma \xrightarrow{\delta} e\gamma$ :

By assumption we have that $\Gamma \vdash c_e\gamma : \alpha$, and since $\Sigma$ is valid, we know $\Delta_c \vdash c_e = e : \alpha_e$, where $\Delta_c$ is the local context and $\alpha_c$ the type of $c_e$. Since $c_e\gamma$ is well-typed in $\Gamma$, we have $\Gamma \vdash \gamma : \Delta_c$, so by the Subst-rule we get $\Gamma \vdash c_e\gamma = e\gamma : \alpha_c\gamma$. Finally, $\alpha$ and $\alpha_c\gamma$ are equal types by meta theory assumption 1.

$fe \xrightarrow{\delta} f'e$ :

By induction hypothesis of $f \xrightarrow{\delta} f'$ and the AppEq rule.

□

**Lemma 7.1.2** *If* $\Gamma \vdash a : \alpha$ *and* $a \xrightarrow{\beta S} a'$, *then* $\Gamma \vdash a = a' : \alpha$.

***Proof:*** Case analysis on $a \xrightarrow{\beta S} a'$. Analogous to the meta properties in the proof of 4.1, we can note the following properties; If we have a derivation

$\Gamma \vdash a\gamma : \alpha$

in the substitution calculus, then there exist a context $\Delta$, a type $\alpha'$ and derivations

$$
\begin{array}{ll}
\Delta \vdash a : \alpha' & (MT5) \\
\Gamma \vdash \gamma : \Delta & (MT6) \\
\Gamma \vdash \alpha'\gamma = \alpha : \ Type & (MT7)
\end{array}
$$

Moreover, if we have a derivation of

$\Gamma \vdash ba : \alpha,$

then there exists a type $\alpha$ and a family of types $\beta'$ such that the following derivations are possible

$$
\begin{array}{ll}
\Gamma \vdash b : \alpha' \rightarrow \beta' & (MT8) \\
\Gamma \vdash a : \alpha' & (MT9) \\
\Gamma \vdash \beta' a = \alpha : \ Type & (MT10)
\end{array}
$$

$x\{\} \xrightarrow{\beta S} x$ :

By assumption we have $\Gamma \vdash x\{\} : \alpha$, which implies

(1) $\Gamma \vdash \alpha : \mathit{Type}$,

and there is a context $\Delta$ and type $\alpha'$ (by $MT5 - MT7$) such that

(2) $\Delta \vdash x : \alpha'$,
(*) $\Gamma \vdash \{\} : \Delta$,and
(**) $\Gamma \vdash \alpha'\{\} = \alpha : \mathit{Type}$.

Since $\alpha'\{\} = \alpha'$, we get by $(**)$ that

(3) $\Gamma \vdash \alpha' = \alpha : \mathit{Type}$.

Moreover, by (2) we know that

(4) $\Delta \preceq \Gamma$

since the meaning of $\Gamma \vdash \{\} : \Delta$ is that all clauses $x : \alpha$ in $\Delta$ is well-typed in $\Gamma$ which means that all clauses are also in $\Gamma$.

$$\dfrac{\dfrac{\dfrac{\dfrac{(2)}{\Delta \vdash x : \alpha'} \quad \dfrac{(4)}{\Delta \preceq \Gamma}}{\Gamma \vdash x : \alpha'} \quad \dfrac{(3)}{\Gamma \vdash \alpha' = \alpha : \mathit{Type}}}{\dfrac{\Gamma \vdash x : \alpha}{\Gamma \vdash x\{\} = x : \alpha\{\}}} \quad \dfrac{\dfrac{(1)}{\Gamma \vdash \alpha : \mathit{Type}}}{\Gamma \vdash \alpha\{\} = \alpha : \mathit{Type}}}{\Gamma \vdash x\{\} = x : \alpha}$$

$x\{\gamma, x{:=}a\} \xrightarrow{\beta S} a$ :

By assumption we have $\Gamma \vdash x\{\gamma, x{:=}a\} : \alpha$ which by $MT5 - MT7$ gives

$(*)$ $\Delta \vdash x : \alpha'$
$(**)$ $\Gamma \vdash \{\gamma, x{:=}a\} : \Delta$
(1) $\Gamma \vdash \alpha = \alpha'\{\gamma, x{:=}a\} : \mathit{Type}$

for some context $\Delta$ and type $\alpha'$. By $(**)$ and the updating rule, we have that

(2) $\Gamma \vdash a : \alpha'\gamma$.

Let $\Delta'$ be the largest subcontext of $\Delta$ which does not contain $x$. Then

$\Gamma \vdash \{\gamma, x{:=}a\} : \Delta'$

hold by the (target) thinning rule. Now, since $x$ does not occur in $\Delta'$, we know that the smaller substitution $\gamma$ fits $\Delta'$ also, i.e.

(3) $\Gamma \vdash \gamma : \Delta'$.

Finally, since $x : \alpha'$ is a clause in $\Delta$, and $\Delta'$ is the largest subcontext not containing $x$, we know that $\alpha'$ is a type in $\Delta'$ so we get

(4) $\Delta' \vdash \alpha' : Type$.

Let us denote the following derivation by $\mathcal{D}$:

$$\left.\dfrac{\dfrac{(4)}{\Delta' \vdash \alpha' : Type} \qquad \dfrac{\dfrac{(3)}{\Gamma \vdash \gamma : \Delta'} \quad \dfrac{(2)}{\Gamma \vdash a : \alpha'\gamma}}{\Gamma \vdash \{\gamma, x{:=}a\} = \gamma : \Delta'}\beta_0}{\Gamma \vdash \alpha'\gamma = \alpha'\{\gamma, x{:=}a\} : Type}\right\}\mathcal{D}$$

where the $\beta_0$ rule is applicable since $x \notin \Delta'$.

Now, we can derive

$$\dfrac{\dfrac{\dfrac{(3)}{\Gamma \vdash \gamma : \Delta'} \quad \dfrac{(2)}{\Gamma \vdash a : \alpha'\gamma}}{\Gamma \vdash x\{\gamma, x{:=}a\} = a : \alpha'\gamma}\beta_1 \qquad \dfrac{\begin{matrix}\mathcal{D}\\ \vdots\end{matrix} \quad \dfrac{(1)}{\Gamma \vdash \alpha'\{\gamma, x{:=}a\} = \alpha : Type}}{\Gamma \vdash \alpha'\gamma = \alpha : Type}}{\Gamma \vdash x\{\gamma, x{:=}a\} = a : \alpha}$$

$x\{\gamma, y{:=}a\} \xrightarrow{\beta S} x\gamma$ :

Analogous to the previous case, but the $\beta_2$ rule is used instead of the $\beta_1$-rule.

$x((\gamma\gamma')\delta) \xrightarrow{\beta S} x(\gamma(\gamma'\delta))$ :

Since $(\gamma\gamma')\delta$ and $\gamma(\gamma'\delta)$ are equal substitutions by associativity.

$x(\{\delta, x{:=}a\}\gamma) \xrightarrow{\beta S} a\gamma$ :

Since $\{\delta, x{:=}a\}\gamma = \{\delta\gamma, x{:=}a\gamma\}$ and $x\{\delta\gamma, x{:=}a\gamma\} = a\gamma$.

$x(\{\delta, x{:=}a\}\gamma) \xrightarrow{\beta S} x(\delta\gamma)$ :

Since $\{\delta, x{:=}a\}\gamma = \{\delta\gamma, x{:=}a\gamma\}$ and $x\{\delta\gamma, x{:=}a\gamma\} = x(\delta\gamma)$ if $x \neq y$.

$x(\{\}\gamma) \xrightarrow{\beta S} x\gamma$ :

Since $\{\}\gamma = \gamma$.

The other substitution rules follows exactly the equalities defined in the substitution calculus (distributivity and associativity) and the application rules corresponds to the $\beta$- and $\beta^*$-rule. Finally, the *AppApp* rule holds by the induction hypothesis and the AppEq rule.

$\square$

**Lemma 7.1.3** *If* $\Gamma \vdash a : \alpha$ *and* $a \xrightarrow{M} L(a')$, *then* $\Gamma \vdash a = a' : \alpha$.

***Proof:*** If $L = \mathit{Irr}$ then $a$ and $a'$ are identical, so let us assume $a \xrightarrow{M} \mathit{Reduced}(d\gamma)$. Then we have $\langle p, a\rangle \stackrel{M}{\Longrightarrow} \gamma$ for some pattern rule $p = d \in \Sigma$. By the validity of $\Sigma$ we know

(1) $\Delta_p \vdash p = d : \alpha_p$

where $\Delta_p$ is the pattern context and $\alpha_p$ the type of $p$. By lemma 6.1.3.1 we have

(2) $\Gamma \vdash a = p\gamma : \alpha$

which by $MT6$ and $MT7$ gives the derivations

(3) $\Gamma \vdash \gamma : \Delta_p$, and
(4) $\Gamma \vdash \alpha = \alpha_p\gamma : \mathit{Type}$.

Now, we can derive

$$\dfrac{\dfrac{(2)}{\Gamma \vdash a = p\gamma : \alpha} \qquad \dfrac{\dfrac{\dfrac{(1)}{\Delta_p \vdash p = d : \alpha_p} \quad \dfrac{(3)}{\Gamma \vdash \gamma : \Delta_p}}{\Gamma \vdash p\gamma = d\gamma : \alpha_p\gamma} \quad \dfrac{(4)}{\Gamma \vdash \alpha = \alpha_p\gamma : \mathit{Type}}}{\Gamma \vdash p\gamma = d\gamma : \alpha}}{\Gamma \vdash a = d\gamma : \alpha}$$

□

**Lemma 7.1.3.1** *If* $\Gamma \vdash a : \alpha$ *and* $\langle p, a\rangle \stackrel{M}{\Longrightarrow} \gamma$, *then* $\Gamma \vdash a = p\gamma : \alpha$.

**Lemma 7.2** *If* $\mathcal{C}$ *holds, then the preconditions of* Conv *and* HConv *are preserved in recursive calls.*

***Proof:*** In Conv-fun we know $\Gamma \vdash a : \alpha \to \beta$, so we have $\Gamma \vdash \alpha : \mathit{Type}$ and since $z \notin \mathit{Dom}(\Gamma)$, $[\Gamma, z : \alpha]$ is a valid extension of $\Gamma$. Hence, $[\Gamma, z : \alpha] \vdash az : \beta z$ is derivable. Analogous for $\Gamma \vdash b : \alpha \to \beta$. In Conv-hnf we have $\Gamma \vdash a : \alpha$ by the precondition, so since $a \xrightarrow{hnf} L(a')$ we get (by lemma 6.1) $\Gamma \vdash a = a' : \alpha$, which implies $\Gamma \vdash a' : \alpha$. Analogous for $\Gamma \vdash b' : \alpha$. In HConv-app, we know that if $\mathcal{C}_1$ holds, then $\Gamma \vdash f = g : \alpha \to \beta$ and if $\mathcal{C}_2$ also holds, we know $\Gamma \vdash a = b : \alpha$. Hence, by the AppEq rule we get $\Gamma \vdash \beta a = \beta b : \mathit{Type}$, so both $fa$ and $gb$ are well-typed.

□

# Appendix C

# Completeness proofs

## C.1 Completeness of type formation

**Proposition 10 (TF-complete)**
*Let $\Gamma \vdash \alpha : Type$ be a derivation where $\alpha$ is $\Gamma$-distinct and $\beta S$-normal. Assume normalisation of the head-normal reduction. Then $\forall \Gamma^* \geq \Gamma$ such that $\alpha$ is $\Gamma^*$-distinct, we have*

$$TF(\alpha, \Gamma^*) \Rightarrow [\,].$$

***Proof:*** A judgement of the form $\alpha \vdash \Gamma : Type$ can only be built up by structural rules from forming types and by the thinning rule. The thinning rule is taken care of by the induction hypothesis since we show the property $\forall \Gamma^* \geq \Gamma$. Since $\alpha$ is restricted to be $\beta S$-normal, we have only to consider three cases, i.e. when $\alpha$ is *Set*, *El*$(A)$ or a function $\alpha \rightarrow \beta$.

The proof is by induction on the structure of $\alpha$.

**$\alpha = Set$:**

$TF(Set, \Gamma^*) \Rightarrow [\,]$ is immediate by the TF-Set rule.

**$\alpha = El(A)$:**

Assume $\Gamma \vdash El(A) : Type$, and let $\Gamma^*$ be an extension of $\Gamma$ which is distinct relative to $El(A)$. Since $\Gamma \vdash El : Set \rightarrow Type$, we must have $\Gamma \vdash A : Set$.

By $TC$-complete we get

$$TC(A, Set, \Gamma) \Rightarrow [\,], \quad \forall\Gamma^* \geq \Gamma.$$

and according to the TF-El rule, this is all we need to show.

**$\alpha = \alpha{\rightarrow}\beta$:**

Assume $\Gamma \vdash \alpha{\rightarrow}\beta : Type$. We must show that

$$TF(\alpha{\rightarrow}\beta, \Gamma^*) \Rightarrow [\,], \quad \forall\Gamma^* \geq \Gamma.$$

There are two possibilities for $\beta$, which is either *El* or $[x]\alpha'$. If $\beta$ is *El*, then $\alpha$ must be *Set* and it follows directly from the TF-Fun' rule. Otherwise we have by induction hypothesis

$$\begin{array}{l}(1)\ TF(\alpha, \Gamma^*) \Rightarrow [\,], \quad \forall\Gamma^* \geq \Gamma \\ (2)\ TF(\alpha', \Gamma_1^*) \Rightarrow [\,], \quad \forall\Gamma_1^* \geq [\Gamma, x : \alpha].\end{array}$$

Since we are only interested in extensions of $\Gamma$which are distinct from $\alpha{\rightarrow}[x]\alpha'$, this means that $x$ may not occur in these extensions. Therefore, we may only consider the extensions $\Gamma_1^* \geq [\Gamma, x : \alpha]$ where no clause in $\Gamma_1^*$ depends on $x$. Thus, we may set $\Gamma_1^* = [\Gamma^*, x : \alpha]$. Then , by the TF-Fun rule, we get

$$TF(\alpha{\rightarrow}[x]\alpha', \Gamma^*) \Rightarrow [\,], \quad \forall\Gamma^* \geq \Gamma.$$

from (1) and (2).

□

## C.2 Completeness of type checking

**Proposition 11 (GTE-complete)**
*Let $\Gamma \vdash a : \alpha$, $\Gamma \vdash \gamma : \Psi$ and $\Gamma \vdash f : \alpha$ be derivations where*

*$a, \gamma$ and $f$ are $\Gamma$-distinct and $\beta S$-normal,*
*$f$ is not an abstraction, and*
*$\gamma$ is $\Psi$-normal.*

*Let $\Gamma^*$ denote an extension of $\Gamma$, which respects the restriction of being distinct relative to $a$, $\gamma$ and $f$, respectively. Then we have the following properties of GTE, FC and CT:*

$$\begin{array}{llll}\Gamma \vdash a : \alpha,\ \forall\Gamma^* \geq \Gamma, & & & \\ \quad \forall\alpha^*.\Gamma^* \vdash \alpha = \alpha^* : Type & \supset\ GTE(a, \alpha^*, \Gamma^*) \Rightarrow \xi & \wedge & \xi \text{ holds} \\ \Gamma \vdash \gamma : \Psi,\ \forall\Gamma^* \geq \Gamma & \supset\ FC(\gamma, \Psi, \Gamma^*) \Rightarrow \xi & \wedge & \xi \text{ holds} \\ \Gamma \vdash f : \alpha,\ \forall\Gamma^* \geq \Gamma & \supset\ CT(f, \Gamma^*) \Rightarrow \langle\xi, \alpha_0\rangle & \wedge & \xi \text{ holds}\end{array}$$

*where $\Gamma^* \vdash \alpha = \alpha_0 : Type$ (by lemma 2.1).*

***Proof:*** We have that if $\Gamma \vdash a : \alpha$ then the last step in the derivation is either a structural rule depending on $a$, the type conversion rule or the thinning rule. The induction hypothesis is strengthened in such a way that the last two becomes trivial. Therefore we need only consider the structural rules. Similarly, for $\Gamma \vdash \gamma : \Psi$ we need only consider the structural rules of $\gamma$, since the thinning rules is accounted for in the induction hypothesis and the target thinning rule is not applicable since $\gamma$ is required to be $\Psi$-normal.

We will prove the properties by induction on the structure of the terms $a, f$ and the substitution $\gamma$.

- $$\frac{}{\Gamma \vdash x : \alpha}\ {\scriptstyle \text{Ass}}\ {\scriptstyle (x\,:\,\alpha\,\in\,\Gamma)}$$

  Assume

  (1) $\Gamma^* \geq \Gamma$, and
  (2) $\Gamma^* \vdash \alpha = \alpha^* : \mathit{Type}$.

  We need to show

  $GTE(x, \alpha^*, \Gamma^*) \Rightarrow \xi\ \wedge\ \xi$ holds, and
  $CT(x, \Gamma^*) \Rightarrow \langle \xi, \alpha_0 \rangle\ \wedge\ \xi$ holds.

  Since $\Gamma^* \geq \Gamma$ we must have $x : \alpha \in \Gamma^*$. Applying GTE-Var we get

  $GTE(x, \alpha^*, \Gamma^*) \Rightarrow [\langle \alpha, \alpha^*, \Gamma \rangle]$.

  Thus, we can conclude $[\langle \alpha, \alpha^*, \Gamma^* \rangle]$ holds by (2).

  The $CT$ case is immediate since

  $CT(x, \Gamma) \Rightarrow \langle [\,], \alpha \rangle$.

- $$\frac{}{\Gamma_c \vdash c : \alpha_c}\ {\scriptstyle \text{Const}}\ {\scriptstyle (c\,:\,\alpha_c\ \Gamma_c\,\in\,\Sigma)}$$

  Analogous to the case above, but we have the typing of the constant in the environment rather than in the context.

- $$\frac{\Psi \vdash c : \alpha \quad \Gamma \vdash \gamma : \Psi}{\Gamma \vdash c\gamma : \alpha\gamma}\ {\scriptstyle \text{Subst}}$$

  Assume

  (1) $\Gamma^* \geq \Gamma$, and
  (2) $\Gamma^* \vdash \alpha\gamma = \alpha^* : \mathit{Type}$.

  We need to show

  $GTE(c\gamma, \alpha^*, \Gamma^*) \Rightarrow \xi\ \wedge\ \xi$ holds, and

$CT(c\gamma, \Gamma^*) \Rightarrow \langle \xi, \alpha_0 \rangle \wedge \xi$ holds.

Note that $c$ is not in $\beta S$-normal form, so the induction hypothesis does not apply to the first premiss. However, we must have $c$ declared in $\Sigma$ with some type $\alpha_c$ and context $\Gamma_c$, and by meta theory assumption 1 we know that $\alpha_c$ and $\alpha$ are convertible types.

For the second premiss, we must make sure that $\gamma$ is $\Psi$-normal. We know that $\gamma$ is $\Gamma_c$-normal, since $c\gamma$ is $\beta S$-normal, so we will show that $\Psi = \Gamma_c$. Since $\gamma$ is $\Gamma_c$-normal and the target context may only be made smaller (by target thinning), we must have

$\Psi \leq \Gamma_c$.

Also, since $c$ is declared in $\Gamma_c$ and the thinning rule extends the context, the first premiss is only possible if

$Ps \geq \Gamma_c$.

Thus, $\Psi = \Gamma_c$ and we can assume the induction hypothesis

(3) $FC(\gamma, \Psi, \Gamma^*) \Rightarrow \xi \wedge \xi$ holds.

Now, the GTE-Subst rule can be applied to (3), yielding

$GTE(c\gamma, \alpha^*, \Gamma^*) \Rightarrow [\langle \alpha_c\gamma, \alpha^*, \Gamma^* \rangle]$.

Since $\alpha_c = \alpha$ we have $\alpha_c\gamma = \alpha\gamma = \alpha^*$ (by (2)) we have the desired property.

Finally, the CT-case follows by the CT-Subst rule applied to (3).

- $$\frac{[\Gamma, x : \alpha_1] \vdash b : \alpha_2}{\Gamma \vdash [x]b : \alpha_1 \rightarrow [x]\alpha_2}\ \text{Abs}$$

Assume

(1) $\Gamma^* \geq \Gamma$, and
(2) $\Gamma^* \vdash \alpha_1 \rightarrow [x]\alpha_2 = \alpha_1^* \rightarrow [y]\alpha_2^* : Type$,

where $\alpha_1^* \rightarrow [y]\alpha_2^*$ is an arbitrary type convertible to $\alpha_1 \rightarrow [x]\alpha_2$ (lemma 4.2). We need to show

$GTE([x]b, \alpha_1^* \rightarrow [y]\alpha_2^*, \Gamma^*) \Rightarrow \xi \wedge \xi$ holds.

We have by induction hypothesis that

$\forall(\Gamma_{ih} \geq [\Gamma, x : \alpha_1])$ and $\forall \alpha_{ih}$ such that $\Gamma_{ih} \vdash \alpha_{ih} = \alpha_2 : Type$ we know that

(IH) $GTE(b, \alpha_{ih}, \Gamma_{ih}) \Rightarrow \xi_{ih}$ and $\xi_{ih}$ holds.

Since all context extensions must respect the restriction to be distinct from bound variables, we know that $\Gamma^*$ may not contain $x$. Thus, we may consider only the contexts $\Gamma_{ih}$ which extends $[\Gamma, x : \alpha_1]$ but in which no types depend on $x$, .i.e all contexts which satisfies $\Gamma_{ih} = [\Gamma^*, x : \alpha_1] =$ (by SubExt) [1] $[\Gamma^*, x : \alpha_1^*]$.

Finally, we need to show that $\alpha_{ih} = ([y]\alpha_2^*)x$ since then $\xi = \xi_{ih}$ and we know that $\xi_{ih}$ holds by (IH). We have

$\alpha_{ih} = \alpha_2$ (by IH)

and by (2) we get $[x]\alpha_2 = [y]\alpha_2^*$ which implies

$\alpha_2 = ([y]\alpha_2^*)x$

so we are done.

- $$\frac{\Gamma \vdash f : \alpha_1 \to \beta \quad \Gamma \vdash a : \alpha}{\Gamma \vdash fa : \beta a}\ \text{App}$$

Assume

(1) $\Gamma^* \geq \Gamma$, and
(2) $\Gamma^* \vdash \alpha^* = \beta a : \mathit{Type}$.

We have by induction hypothesis that

(IH1) $CT(f, \Gamma^*) \Rightarrow \langle \xi_1, \alpha_1 \to \beta_1 \rangle$, and
(IH2) $GTE(a, \alpha_2^*, \Gamma^*) \Rightarrow \xi_2$, $\forall \alpha_2^*$ such that $\Gamma^* \vdash \alpha_2^* = \alpha_1 : \mathit{Type}$ where $\xi_1$ and $\xi_2$ hold.

By lemma 2.1, we have that if $f$ has type $\alpha_1^* \to \beta_1^*$, then

$\Gamma^* \vdash \alpha_1^* \to \beta_1^* = \alpha_1 \to \beta_1 : \mathit{Type}$

and in particular we have

(3) $\Gamma^* \vdash \alpha_1^* \to \beta_1^* = \alpha \to \beta : \mathit{Type}$.

Hence, if we apply GTE-App to the induction hypothesis we get

$GTE(fa, \beta_1^* a, \Gamma^*) \Rightarrow \xi_1 \mathbin{@} \xi_2 \mathbin{@} [\langle \beta_1^* a, \alpha^*, \Gamma^* \rangle]$.

Now, we can derive

$$\frac{\dfrac{(2)}{\Gamma^* \vdash \alpha^* = \beta a : \mathit{Type}} \qquad \dfrac{\dfrac{(3)}{\Gamma^* \vdash \alpha_1^* \to \beta_1^* = \alpha \to \beta : \mathit{Type}} \quad \dfrac{(IH2)}{\Gamma^* \vdash a : \alpha_1^*}}{\Gamma^* \vdash \beta a = \beta_1^* a : \mathit{Type}}}{\Gamma^* \vdash \beta_1^* a = \alpha^* : \mathit{Type}}$$

and hence $[\langle \beta_1^* a, \alpha^*, \Gamma^* \rangle]$ holds.

[1] The alternative SubExt rule in [Tas93] requires more work at this point since it requires equal contexts to have identical types of the variables.

Finally, we get by applying the induction hypothesis to the CT-App rule that

$$CT(fa, \Gamma^*) \Rightarrow \langle \xi_1 \mathbin{@} \xi_2, \beta_1^* a\rangle$$

which by the same reasoning as above proves this case.

- $$\frac{\Gamma : \mathit{Context}}{\Gamma \vdash \{\} : \Gamma}\ \text{Id}$$

  Immediate by FC-Empty.

- $$\frac{\Gamma \vdash \gamma : \Delta \quad \Gamma \vdash a : \alpha\gamma}{\Gamma \vdash \{\gamma, x{:=}a\} : [\Delta, x : \alpha]}\ \text{Upd}$$

  Follows directly from the induction hypothesis and the FC-ext rule.

□

**Lemma 11.1** *$\beta S$-normal terms which are not abstraction, have unique types.*
***Proof:*** Induction on the structure of the term. □

## C.3 Completeness of type conversion

**Proposition 12 (TSimple-complete)**
*Let $\xi$ be a well-formed list of type equations. If*

$$\Gamma \vdash \alpha = \alpha' : \mathit{Type} \quad \forall \langle \alpha, \alpha', \Gamma\rangle \in \xi$$

*then*

$$\mathit{TSimple}(\xi) \Rightarrow \mathcal{C} \text{ and } \mathcal{C} \text{ holds.}$$

***Proof:*** Induction on the length of $\xi$ (using TConv-complete). □

**Proposition 13 (TConv-complete)**
*If $\Gamma \vdash \alpha = \alpha' : \mathit{Type}$, then $\mathit{TConv}(\alpha, \alpha', \Gamma) \Rightarrow \mathcal{C}$ and $\mathcal{C}$ holds.*

***Proof:*** The proof proceeds by case analysis of the types $\alpha$ and $\alpha'$. By lemma 4.1 we have a well-founded order on $\alpha$ and $\alpha'$ such that TConv terminates. There are two main cases, whether the types are on (outermost) constructor form or not.

*Case 1.* If $\alpha$ and $\alpha'$ are both on constructor form, then we have by lemma 4.2 that

$$CF(\alpha) = CF(\alpha').$$

Thus we have three cases corresponding to the possible forms of a type on constructor form:

*Set* : Immediate by TConv-Id.

$El(A)$ : Assuming $\Gamma \vdash El(A) = El(B) : Type$, we have $\Gamma \vdash A = B : Set$, since constructors are one-to-one. Hence, the equation $[\langle A, B, Set, \Gamma\rangle]$ holds and the TConv-El rule gives us the desired property.

$\alpha \rightarrow \beta$ : Assuming $\Gamma \vdash \alpha \rightarrow \beta = \alpha' \rightarrow \beta' : Type$, we must have

$$(1)\ \Gamma \vdash \alpha = \alpha' : Type$$

and $\Gamma \vdash \beta = \beta' : \alpha \rightarrow Type$ by lemma 4.4. Now, we can extend the context with a fresh variable and apply that variable to the type families, yielding

$$(2)\ [\Gamma, z : \alpha] \vdash \beta z = \beta' z : Type.$$

Since (1) and (2) are smaller with respect to our order, we can apply the induction hypothesis getting

(IH1) $TConv(\alpha, \alpha', \Gamma) \Rightarrow \mathcal{C}_1$, and $\mathcal{C}_1$ holds
(IH2) $TConv(\beta z, \beta' z, [\Gamma, z : \alpha]) \Rightarrow \mathcal{C}_2$ and $\mathcal{C}_2$ holds

which by the TConv-Fun rule proves this case.

*Case 2.* Otherwise at least one of $\alpha$ or $\alpha'$ is not on constructor form. Let us assume $\alpha$ is not on constructor form. Then there exist a reduction $\alpha \xrightarrow{T\beta S} \alpha^*$, where $\alpha^*$ is on constructor form and $CF(\alpha) = CF(\alpha^*)$ (by lemma 4.3). We have by induction hypothesis that

$$TConv(\alpha^*, \alpha', \Gamma) \Rightarrow \mathcal{C} \text{ and } \mathcal{C} \text{ holds}$$

and since $\Gamma \vdash \alpha = \alpha^* : Type$ (by lemma 4.1) we can conclude

$$TConv(\alpha, \alpha', \Gamma) \Rightarrow \mathcal{C}. \text{ and that } \mathcal{C} \text{ holds.}$$

□

**Lemma 13.1** *TConv terminates.*

***Proof:*** We will construct a well-founded ordering on a pair of types. Hence, the algorithm will terminate. The order on $\langle \alpha, \alpha' \rangle$ is a lexicographical order on the pair

$$\langle \#Arr(\alpha), O(\alpha) + O(\alpha')\rangle$$

where $\#Arr(\alpha)$ is the number of function arrows in the type $\alpha$ and $O(\alpha)$ is a measure of $\alpha$ which will decrease in each step of the reduction to constructor form. We can see that given a function type, the number of arrows is always strictly less in the parts of the function, which is needed for the TConv-Fun rule. We have defined this order since even though $\alpha$ is structurally smaller than $\alpha \to \beta$, $\beta z$ is not. Note that the number of arrows in a type is not effected by reduction. The second component in the pair guarantees that the recursive call in the TConv-$\beta Subst$ rule is smaller since we know that at least one of the types are not on constructor form, and since reduction of a type decreases $O$ (by lemma 4.1.1), the sum will always be smaller in the recursive call.

□

**Lemma 13.1.1** *There exist a well-founded order $O$ such that if $\alpha \stackrel{T\beta S}{\longrightarrow} \alpha$', then $O(\alpha) > O(\alpha')$.*

***Proof:*** The order of a type is an ordered pair, where the first component is the number of type family applications and the second component is a measure of "how deep inside the type" substitutions are pushed.

**Definition C.1** The *order* $O$ of a type $\alpha$ is defined by

$$O(\alpha) = \langle \#App(\alpha), |\alpha|\rangle$$

where the $\#App(\alpha)$ is the number of type applications in $\alpha$.

The measure of the *level of substitutions* is defined by

$$\begin{array}{ll}
|Set| = 1 & |El| = 1 \\
|\alpha \to \beta| = |\alpha| + |\beta| & |[x]\alpha| = |\alpha| \\
|\beta a| = |\beta| & |\beta\gamma| = |\beta| + D(\beta) * |\gamma| \\
|\alpha\gamma| = |\alpha| + D(\alpha) * |\gamma| &
\end{array}$$

where the *depth* $D$ of a type is defined by

$$\begin{array}{ll}
D(Set) = 1 & D(El) = 1 \\
D(\alpha \to \beta) = D(\alpha) + D(\beta) + 1 & D([x]\alpha) = D(\alpha) + 1 \\
D(\beta a) = D(\beta) + 1 & D(\beta\gamma) = D(\beta) + 1 \\
D(\alpha\gamma) = D(\alpha) + 1 &
\end{array}$$

Now, it is easy to check that the order of a type is strictly decreasing in each step of the reduction to constructor form, by case analysis on $\alpha \stackrel{T\beta S}{\longrightarrow} \alpha'$.

□

**Lemma 13.2** *If* $\Gamma \vdash \alpha = \alpha' : Type$*, then the constructor form of* $\alpha$ *(CF(*$\alpha$*)) is the same as the CF(*$\alpha$*'), and if* $\Gamma \vdash \beta = \beta' : \alpha \rightarrow Type$*, then CF(*$\beta$*) = CF(*$\beta$*').*

***Proof:*** Induction on the length of the derivations of $\Gamma \vdash \alpha = \alpha' : Type$ and $\Gamma \vdash \beta = \beta' : \alpha \rightarrow Type$, respectively. □

**Lemma 13.3** $\forall \alpha : Type$*,* $\alpha$ *is on constructor form or there exists a reduction* $\alpha \xrightarrow{T\beta S}_* \alpha'$ *where* $\alpha'$ *is on constructor form and* $CF(\alpha) = CF(\alpha')$*.*

***Proof:*** Follows from lemma 4.3.1, 4.1.1 and 4.3.2. □

**Lemma 13.3.1** $\forall \alpha : Type$*, there exists a reduction* $\alpha \xrightarrow{T\beta S} \alpha'$ *or* $\alpha$ *is on constructor form.*

***Proof:*** Case analysis on the structure of $\alpha$. □

**Lemma 13.3.2** *If* $\alpha \xrightarrow{T\beta S} \alpha'$*, then CF(*$\alpha$*) = CF(*$\alpha$*').*

***Proof:*** Follows from lemma 4.1 and 4.2. □

**Lemma 13.4** *If* $\Gamma \vdash \alpha_1 \rightarrow \beta_1 = \alpha_2 \rightarrow \beta_2 : Type$ *then* $\Gamma \vdash \alpha_1 = \alpha_2 : Type$ *and* $\Gamma \vdash \beta_1 = \beta_2 : \alpha \rightarrow Type$*.*

## C.4 Completeness of term conversion

**Proposition 14 (Simple-complete)**
*Let* $\mathcal{C}$ *be a well-formed list of term equations. Assuming* $\xrightarrow{hnf}$ *is normalising, we have if*

$$\Gamma \vdash a = b : \alpha \quad \forall \langle a, b, \alpha, \Gamma \rangle \in \mathcal{C}$$

*then*

$$Simple(\mathcal{C}) \Rightarrow [\,].$$

***Proof:*** Induction on the length of $\mathcal{C}$, using Conv-complete. □

**Proposition 15 (Conv-complete)**
*Assuming* $\xrightarrow{hnf}$ *is normalising and if* $\Gamma \vdash a = b : \alpha$*, then* $Conv(a, b, \alpha, \Gamma) \Rightarrow [\,]$.

***Proof:*** Two cases, one for $\alpha$ ground and one for $\alpha$ functional. $\Box$

**Lemma 15.1** *If* $\xrightarrow{hnf}$ *is normalising, then* $Conv$ *terminates..*

**Lemma 15.2** $\forall a : \alpha$*,* $\alpha$ *ground, there exists a reduction* $a \xrightarrow{hnf} a'$*, where* $a'$ *is on head normal form.*

***Proof:*** Case analysis on $a$. $\Box$

# Bibliography


[ACN90] L. Augustsson, T. Coquand, and B. Nordström. A short description of Another Logical Framework. In *Proceedings of the First Workshop on Logical Frameworks, Antibes*, pages 39–42, 1990.

[AGNv94] T. Altenkirsch, V. Gaspes, B. Nordström, and B. von Sydow. *A user's guide to ALF*, 1994. Draft.

[Alt94] Thorsten Altenkirch. Consistency in ALF. Proceedings of La Wintermöte, Programming Methodology Group, Chalmers University, Göteborg, Sweden, January 1994. Draft.

[Bar91] H. P. Barendregt. Introduction to Generalised Type Systems. *J. Functional Programming*, 1(2):125–154, April 1991.

[BBKM93] D. Basin, A. Bundy, I. Kraan, and S. Matthews. A framework for program development based on schematic proof. In *7th International Workshop on Software Specification and Design*, Los Angeles, December 1993. IEEE Computer Society Press.

[CH88] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, 1988.

[CHL] P.-L. Curien, T. Hardin, and J.-J. Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM, to appear*. Also in 1992 INRIA report 1617.

[CKT94] Y. Coscoy, G. Kahn, and L. Théry. Extracting text from proof. In *Draft paper*, September 1994.

[CNSvS94] Thierry Coquand, Bengt Nordström, Jan M. Smith, and Björn von Sydow. Type theory and programming. *EATCS*, (52), February 1994.

[Con86] R. L. Constable et al. *Implementing Mathematics with the NuPRL Proof Development System*. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[Coqa] Catarina Coquand. Combinator Graph Reduction and Infinite Terms in Type Theory. In preparation.

[Coqb] Catarina Coquand. A machine assisted normalisation proof of simply typed $\lambda$-calculus with explicit substitution. To appear in a forthcoming Ph.D thesis.

[Coq91] Thierry Coquand. An algorithm for testing conversion in type theory. In *Logical Frameworks*. Cambridge University Press, 1991.

[Coq92] Thierry Coquand. Pattern matching with dependent types. In *Proceeding from the logical framework workshop at Båstad*, June 1992.

[Coq94] Thierry Coquand. Infinite Objects in Type Theory. In *Types for Proofs and Programs*, LNCS, pages 62–78, Nijmegen, 1994. Springer-Verlag.

[CP90] Thierry Coquand and Christine Paulin. Inductively defined types. In *Proceedings of COLOG-88*, number 417 in Lecture Notes in Computer Science. Springer-Verlag, 1990.

[D+91] G. Dowek et al. The coq proof assistant user's guide version 5.6. Technical report, Rapport Technique 134, INRIA, December 1991.

[Dah92] O-J Dahl. *Verifiable Programming*. Prentice Hall International, 1992.

[dB68] N. G. de Bruijn. The Mathematical Language AUTOMATH, its usage and some of its extensions. In *Symposium on Automatic Demonstration*, volume 125 of *Lecture Notes in Mathematics*, pages 29–61, Versailles, France, 1968. IRIA, Springer-Verlag.

[dB87] N.G. de Bruijn. Generalizing automath by means of a lambda-typed lambda calculus. In *Mathematical Logic and Theoretical Computer Science, Lecture Notes in pure and applied mathematics*, pages 71–92. 1987.

[DFH+91] G. Dowek, A. Felty, H. Herbelin, H. Huet, G. P. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The coq proof assistant user's guide version 5.6. Technical report, Rapport Technique 134, INRIA, December 1991.

[DGKLM84] V. Donzeau-Gouge, G. Kahn, B. Lang, and B Mélèse. Document Structure and Modularity in Mentor. In *Proceedings of the ACM SIGSOFT/SIGPLAN - Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, 1984. Software Engineering Notes Vol. 9, No 3.

[Dow93] Gilles Dowek. A Complete Proof Synthesis Method for the Cube of Type Systems. *Journal of Logic and Computation*, 3(3):287–315, 1993.

[Dyb91] Peter Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In *Logical Frameworks*, pages 280–306. Cambridge University Press, 1991.

[Dyb94] Peter Dybjer. Inductive families. *Formal Aspects of Computing*, pages 440–465, 1994.

[Ell89] Conal M. Elliot. Higher-order unification with dependent function types. In N. Derikowitz, editor, *Proceedings of the 3rd International Conference on Rewriting Techniques and Applications*, pages 121–136, April 1989.

[FH94] A. Felty and D. Howe. Tactic Theorem Proving with Refinement-Tree Proofs and Metavariables. In *Proceedings of Automated Deduction - CADE-12*. Lecture Notes in Artificial Intelligence 814, Springer Verlag, 1994.

[GM92] M.J.C. Gordon and T.F Melham. *HOL: a proof generating system for higher-order logic*. Cambridge University Press, 1992.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[HA90] L. Helmink and R. Ahn. Goal Directed Proof Construction in Type Theory. In G. Huet and G. Plotkin, editors, *Proceedings of First Workshop on Logical Frameworks*, pages 259–297. Esprit Basic Research Action 3245, May 1990.

[Hag91] M. Hagiya. Higher-Order Unification as a Theorem Proving Procedure. In *Proceedings of the Eigth International Conference on Logic Programming*, pages 270–284, Cambridge, Massachussets, 1991. Koichi Furukawa (Ed.) MIT Press.

[HHP87] Robert Harper, Furio Honsell, and Gordon Plotkin. A Framework for Defining Logics. In *Proceedings of the Symposium on Logic in Computer Science*, pages 194–204, Ithaca, New York, June 1987.

[Hof93] Martin Hofmann. A model of intensional Martin-Löf type theory in which unicity of identity proofs does not hold. Technical report, Dept. of Computer Science, University of Edinburgh, June 1993.

[How80] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, London, 1980.

[Hue75] Gérard Huet. A unification algorithm for typed $\lambda$-calculus. *Theoretical Computer Science*, 1(1):27–57, 1975.

[Kol32] A. N. Kolmogorov. Zur Deutung der intuitionistischen Logik. *Matematische Zeitschrift*, 35:58–65, 1932.

[LP92] Z. Luo and R. Pollack. LEGO Proof Development System: User's Manual. Technical report, LFCS Technical Report ECS-LFCS-92-211, 1992.

[Mäe93] Petri Mäenpää. *The Art of Analysis. Logic and History of Problem Solving*. PhD thesis, University of Helsinki, September 1993.

[Mag91] Lena Magnusson. An Implementation of Martin-Löf's Logical Framework. Licentiate Thesis, Chalmers University of Technology and University of Göteborg, Sweden, June 1991.

[Mag92] Lena Magnusson. The new Implementation of ALF. In *The informal proceeding from the logical framework workshop at Båstad, June 1992*, 1992.

[Mag93] Lena Magnusson. Refinement and local undo in the interactive proof editor ALF. In *The Informal Proceeding of the 1993 Workshop on Types for Proofs and Programs*, May 1993.

[Mel94] P.A. Mellies. Typed $\lambda$-calculi with explicit substitutions may not terminate. In *Proceedings of the CONFER Workshop*, April 1994.

[Mil89] D. Miller. A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification. In *Extensions of Logic Programming*. Lecture Notes in Artificial Intelligence 449, Springer Verlag, 1989.

[MN94] Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In *Types for Proofs and Programs*, LNCS, pages 213–237, Nijmegen, 1994. Springer-Verlag.

[MP93] James McKinna and Randy Pollack. Pure type system formalized. In M. Bezem and J.F. Groote, editors, *Proceeding of the International Conference on Typed Lambda Calculi and Applications, TLCA'93*, pages 289–305. Springer-Verlag, LNCS 664, March 1993.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Nor88] Bengt Nordström. Terminating General Recursion. *BIT*, 28(3):605–619, October 1988.

[Nor93] Bengt Nordström. The ALF proof editor. In *Proceedings 1993 Informal Proceedings of the Nijmegen workhop on Types for Proofs and Programs*, 1993.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press, 1990.

[Pet84] Kent Petersson. A Programming System for Type Theory. PMG report 9, Chalmers University of Technology, S–412 96 Göteborg, 1982, 1984.

[Pfe89] Frank Pfenning. Elf: A language for logic definition and verified metaprogramming. In *LICS'89*, pages 313–322. IEEE, June 1989.

[PM89] Christine Paulin-Mohring. *Extraction de Programmes dans le Calcul des Constructions*. PhD thesis, Universite Paris VII, 1989.

[PM93] Christine Paulin-Mohring. Inductive Definitions in the system Coq; rules and properties. In M. Bezem and J.F. Groote, editors, *Proceeding of the International Conference on Typed Lambda Calculi and Applications, TLCA'93*, pages 328–345. Springer-Verlag, LNCS 664, March 1993.

[PN90] Lawrence C. Paulson and Tobias Nipkow. Isabelle tutorial and user's manual. Technical report 189, Universtiy of Cambridge Computer Laboratory, Cambridge, January 1990.

[Pol93] Randy Pollack. Closure under Alpha conversion. In *The Informal Proceeding of the 1993 Workshop on Types for Proofs and Programs*, May 1993.

[Pra] K. V. S. Prasad. Computer aided reasoning about broadcasting systems. In preparation.

[PW90] D.J. Pym and L.A. Wallen. Investigations into proof-search in a system of first-order dependent function types. In *Proceedings of 10th International Conference on Automated Deduction*, July 1990.

[Pym92] David Pym. A unification algorithm for the logical framework. Technical Report ECS-LFCS-92-229, University of Edinburgh, August 1992.

[Rit] E. Ritter. Adapting a counter example to strong normalisation of simply typed $\lambda$-calculus with explicit substitution to the substitution calculus of Martin-Löf. Personal communication.

[Sal88] Anne Salvesen. Polymorphism and Monomorphism in Martin-Löf's Type Theory. Technical report, Norwegian Computing Center, P.b. 114, Blindern, 0316 Oslo 3, Norway, December 1988.

[Tas93] Alvaro Tasistro. Formulation of Martin-Löf's Theory of Types with Explicit Substitution. Licentiate Thesis, Chalmers University of Technology and University of Göteborg, Sweden, May 1993.

[TBK92] L. Théry, Y. Bertot, and G. Kahn. Real Theorem Provers Deserve Real User-Interfaces. Technical Report 1684, INRIA Sophia-Antipolis, May 1992.

[TR81] Teitelbaum and Reps. The Cornell program synthesizer: a syntax-directed programming environment. *Commun. ACM*, 24(9):563–573, 1981.

[TS94] Tanel Tammet and Jan M. Smith. Optimized Encodings of Fragments of Type Theory in First Order Logic. Presented at the Workshop on Proof Search in Type Theoretic Languages, CADE-12, Nancy, June 1994.

[vBJMP94] L.S. van Benthem Jutting, J. McKinna, and R. Pollack. Checking Algorithms for Pure Type Systems. In *Types for Proofs and Programs*, LNCS, pages 19–61, Nijmegen, 1994. Springer-Verlag.

# Index of definitions