Universität Ulm
Abteilung Künstliche Intelligenz
Leiter: Prof. Dr. Friedrich W. von Henke

Construction and Deduction in Type Theories

Dissertation zur Erlangung des Doktorgrades Dr. rer. nat.
der Fakultät für Informatik der Universität Ulm

Martin Strecker
aus Darmstadt

1999

Amtierender Dekan:	Prof. Dr. Uwe Schöning
1. Gutachter:	Prof. Dr. Friedrich W. von Henke
2. Gutachter:	Prof. Dr. Susanne Biundo-Stephan
Externer Gutachter:	Prof. Tobias Nipkow, PhD
Tag der Promotion:	28. April 1999

Abstract

This dissertation is concerned with interactive proof construction and automated proof search in type theories, in particular the Calculus of Constructions and its subsystems.

Type theories can be conceived as expressive logics which combine a functional programming language, strong typing and a higher-order logic. They are therefore a suitable formalism for specification and verification systems. However, due to their expressiveness, it is difficult to provide appropriate deductive support for type theories. This dissertation first examines general methods for proof construction in type theories and then explores how these methods can be refined to yield proof search procedures for specialized fragments of the language.

Proof development in type theories usually requires the construction of a term having a given type in a given context. For the term to be constructed, a *metavariable* is introduced which is successively instantiated in the course of the proof. A naive use of metavariables leads to problems, such as non-commutativity of reduction and instantiation and the generation of ill-typed terms during reduction. For solving these problems, a calculus with *explicit substitutions* is introduced, and it is shown that this calculus preserves properties such as strong normalization and decidability of typing.

In order to obtain a calculus appropriate for proof search, the usual natural deduction presentation of type theories is replaced by a *sequent style presentation*. It is shown that the calculus thus obtained is correct with respect to the original calculus. Completeness (proved with a cut-elimination argument) is shown for all predicative fragments of the lambda cube.

The dissertation concludes with a discussion of some techniques that make proof search practically applicable, such as unification and pruning of the proof search space by exploiting impermutabilities of the sequent calculus.

Acknowledgements

I want to express my gratitude to Prof. von Henke for supporting throughout the years the work described in this thesis. I want to thank Prof. Biundo for accepting to act as second corrector. I am particularly grateful to Prof. Nipkow for agreeing spontaneously to join in as external referee.

Contents

List of Figures

1. Introduction

1.1. Background and Applications

Specification and verification is becoming an increasingly important activity accompanying the design and development of complex systems. *Formal* specification and verification, in particular, permit to attain a degree of precision that cannot be achieved with informal methods. A more widespread use of formal techniques is however often hampered by a lack of expressiveness of specification languages or by insufficient deductive capabilities of verification environments.

Some advanced type theories are promising candidates for specification formalisms, since they combine computational aspects, strong typing and a higher-order logic within one homogeneous framework. Moreover, essential concepts such as module, parameterization, refinement and realization of a specification can be internalized in type theory and thus do not require an extra-logical treatment.

This thesis contributes to the study of deductive support for type theories. In order to take advantage of the full expressiveness of the type theoretic languages under consideration, the investigation starts from a rather broad perspective. Since fully automated proof procedures can realistically only be formulated for limited language fragments, appropriate restrictions have to be imposed in the sequel. Thus, as suggested by the title, this thesis is concerned with

- *construction* of entities of the full language in a way which is sound (in that it respects typing) and consistent (in that it is in accordance with evaluation of programs).
- *deduction* in a more traditional sense, which however is smoothly integrated with the "constructive" aspect and which makes essential use of techniques developed in the more general setting.

In the following, we will informally present some key concepts of type theories (Sections 1.1.1 to 1.1.4) and proof assistants based on type theory (Sec-

tion 1.1.5), with the purpose of exposing some problems that have to be faced when trying to develop an appropriate deductive machinery for a specification and verification environment. In Section 1.2, we will exemplify some of the solutions that are proposed in this thesis to achieve this aim. Since the implementation of the TYPELAB system [vHLS97] provided the initial stimulus for the present investigation, it will serve as an illustration of the methodology. A detailed outline of the contents of this thesis follows in Section 1.3.

In order to keep the presentation concrete, this thesis examines more closely the type theory on which TYPELAB is based, an extension of the Calculus of Constructions [CH88]. Some of the questions and problems raised in the following reappear in a similar form in related theories, possibly even in much less complex variants such as simply-typed higher order logic. The applicability of the solutions proposed below thus extends well beyond a narrow range of specialized logics.

1.1.1. The Essence of Type Theory

Type theory studies the assignment of *types* to *terms*. Terms are the expressions of an (idealized) programming language, usually the λ-calculus, whereas types denote collections of terms. There is a wide variety of type theories, see the textbooks [Tho91, Mit96] for a general introduction and [NPS90, Luo94] for more specialized material.

In some of the more elementary type theories, such as the simply-typed λ-calculus [Hin97], there is a neat syntactic distinction between terms and types, which is blurred in more complex calculi such as the one which is the object of study of this thesis. In any case, type theories usually contain expressions such as function application $(f\ a)$ and λ-abstraction $\lambda\, x : A.\ M$ on the term level and function types $A \to B$ on the type level.

Typing judgements of the form $\Gamma \vdash M : A$ express that term M has type A in context Γ, where Γ is a list $x_1 : T_1, \ldots x_n : T_n$ of variable declarations. Typing judgements are generated by typing derivations like the following which express, respectively, that the application of a function f of type $A \to B$ to an argument a of type A yields a term $(f\ a)$ of type B and that $\lambda\, x : A.\ M$ is a function of type $A \to B$ if variable x is of type A and the function body M is of type B:

$$\frac{\Gamma \vdash f : A \to B \quad \Gamma \vdash a : A}{\Gamma \vdash (f\ a) : B}\ (\to\text{-Elim}) \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda\, x : A.\ M : A \to B}\ (\to\text{-Intro})$$

The process of type inference (given Γ and M, determine A with $\Gamma \vdash M : A$) and type checking (given Γ, M and A, determine whether $\Gamma \vdash M : A$ holds

or not) is well understood. Type inference, for example, can be accomplished efficiently by first decomposing the term M, applying the type inference rules backwards in a syntax-directed fashion, and then building up the type of M.

However, the complementary question of *inhabitation* of a type (given Γ and A, determine M with $\Gamma \vdash M : A$) is still a topic of active research and constitutes a central theme of this thesis. Given that types can convey an informative description of a collection of objects (see Sections 1.1.2 and 1.1.4 below), it is not only of theoretical interest to be able to effectively produce objects satisfying such a description. Obviously, a brute force method which interleaves enumeration of terms M and type checking is of no practical use, even though it demonstrates that the set of inhabited types is recursively enumerable, since type checking is decidable. Inhabitation of types in simply-typed λ-calculus is decidable (see [Kle52b]), whereas any of the extensions discussed in Section 1.1.3 makes it undecidable.

1.1.2. Intuitionistic Logic and Type Theory

The rule for typing function applications, shown above, resembles the rule of *modus ponens* in predicate logic. In general, by interpreting the function space constructor $A \to B$ as implication, a natural correspondence can be established between propositions of *intuitionistic logic* and types. When adding further type constructors, such as Cartesian product $A \times B$ and disjoint sum $A + B$, other connectives such as conjunction and disjunction can be represented. (Alternatively, a higher-order encoding as in Section 2.1.4 is possible). It should be emphasized that it is not reasonable to conceive every type, such as the specification types discussed further below, as a proposition, but only certain types belonging to a collection of "propositional" types.

Elements of these types can be understood as proofs of the corresponding propositions. For example, a proof of $A \times B$ is a pair $\langle p_A, p_B \rangle$, where p_A is a proof of A and p_B a proof of B. Similarly, a proof of $C \to D$ is a function $\lambda\, x : C.p_D$ which transforms any proof x of C into a proof p_D of D.

This *Heyting interpretation* of propositions as types, also known as *Curry-Howard-isomorphism*, provides a semantics for intuitionistic logic, similar in spirit to Kleene's realizability interpretation [Kle52b]. Here, term models replace the usual set-theoretic models, so that "provability of A under hypotheses Γ" is equated to "existence of a term M such that $\Gamma \vdash M : A$ holds". An advantage of this semantics over a Kripke-style semantics of intuitionistic logic is that it directly carries over to type theories for which no classical set theoretic semantics can be constructed [Rey84, Pit87]. Semantic issues will mostly be disregarded in the following. It should however be kept in mind that the completeness results of proof search presented in Section 3.2, although given

as "syntactic" equivalence between two calculi, can in this sense be interpreted as completeness with respect to the above semantics.

The propositions-as-types-correspondence is often cited as one of the principles underlying *program extraction* techniques. The general idea is to carry out a constructive proof of the proposition $\forall\, x : I.\, \exists\, y : O.P(x, y)$, where $P(x, y)$ specifies the input-output behaviour of an algorithm. The resulting proof object $M : (\forall\, x : I.\, \exists\, y : O.P(x, y))$ can be decomposed into a function $f : I \to O$ and a proof object of the proposition $\forall\, x : I.P(x, f(x))$ which states the correctness of f with respect to the original specification. Here, we will not pursue this issue further, see for example [PM89, PMW93, HN88], but also [BBS+98] for an alternative approach.

The existence of proof objects increases the confidence in the "correctness" of a proof. Indeed, a simple validation of a proof can be carried out by type checking the proof object with respect to the proposition it is supposed to prove. As an extension of this idea, it is possible to attach a certification in the form of a proof that certain correctness requirements are met to documents or to software – see for example the concept of "proof carrying code" in [NL96].

The most immediate consequence of the propositions-as-types interpretation is that proof methods developed for logics can be adapted and extended to type theories, even for non-propositional types. This correspondence will be elaborated in detail in this thesis.

1.1.3. Dependent types

Adding type dependencies is one of the principal ways of extending the simply-typed λ-calculus. The most prominent forms are dependencies of types on types and of types on terms, which can be combined in various ways. In the following, some of these dependencies will be discussed. A systematization is provided by the λ-cube, which is recapitulated in Section 3.3.1 with the purpose of estimating the strength of the methods of proof search developed in Chapter 3.

The *dependency of types on types* can be used to model polymorphism. A standard example of a polymorphic function is the (higher-order) *map* function which takes a function f and a list *lst* and applies f to all elements of *lst*. As long as the domain type of f agrees with the type of the elements of *lst*, it is immaterial what the exact types of f and *lst* are. We can therefore abstract over them, giving *map* the type $\Pi A, B : \mathit{Type}.(A \to B) \to (\mathit{List}\ A) \to (\mathit{List}\ B)$. In this type expression, Π may be read as a universal quantification over types.

Before applying *map* to values such as a *length* function on strings and a list of strings *strlst*, the type parameters have to be instantiated appropriately. The function $(\mathit{map}\ \mathit{String}\ \mathit{Nat})$ of type $(\mathit{String} \to \mathit{Nat}) \to (\mathit{List}\ \mathit{String}) \to (\mathit{List}\ \mathit{Nat})$ can then be applied to arguments *length* and *strlst*. Since it is

cumbersome to write down type parameters and type arguments, explicit type information may be omitted in polymorphic programming languages like ML [Pau91], Gofer and Haskell [Has97]. It can be recovered through a *type inference* algorithm whose main ingredient is a unification procedure on type expressions. A related idea is implemented in proof assistants such as Lego [LP92, Pol94] and Typelab. For the types to be inferred, *metavariables* are introduced and later solved by unification. More on this topic can be found in Section 4.1.

Girard's System F [Gir72, GLT89], a subsystem of the polymorphic type system considered in this thesis, is more complex than the ML type system since abstraction over type variables can occur "on the left of an implication", as in $(\Pi A : \mathit{Type}.A \to A) \to B$, thus essentially permitting existential quantification over types. Indeed, it can be shown that type inference for System F is not decidable [Wel94]. Furthermore, it is known that classical first-order predicate logic can be embedded into higher-order intuitionistic propositional logic [Löb76], the logical counterpart of System F. As a consequence, the problem of whether an arbitrary type of System F is inhabited is not decidable.

The *dependency of types on values* is most prominent in the coding of intuitionistic predicate logic according to the propositions-as-types principle. For example, in the proposition $\forall\, x : T.P(x)$, the type $P(x)$ depends on the value of x. ($\forall$-abstraction is a notational variant of Π-abstraction that emphasizes the logical aspect as opposed to the typing aspect).

In the realm of data types, dependency on values can be used to define families of types like $\mathit{Vect} : \mathit{Nat} \to \mathit{Type}$, where $\mathit{Vect}\ n$ is, for example, the type of bit-vectors of length n.

Various embeddings of classical first order into intuitionistic first order logic are known, such as Gödel's double-negation embedding [TS96]. Consequently, intuitionistic predicate logic is undecidable and therefore also the problem of inhabitation of types corresponding to propositions of intuitionistic predicate logic.

1.1.4. Specifications and mathematical theories

The Extended Calculus of Constructions (*ECC*) [Luo90] enriches the Calculus of Constructions with Sigma types which permit to encode specifications and mathematical theories. Specifications are not grafted onto the logic by an external mechanism, but are types, that is, internal objects of the logic. Elements of specification types can be understood as structures satisfying the corresponding specification.

A Sigma type $\Sigma x : A.B$ is a generalized Cartesian product $A \times B$ in which B may depend on x. Take, for example, the following description of a monoid: A monoid is given by a carrier set T, a binary operation *op* and a constant *unit*.

The binary operation is associative and has *unit* as neutral element. These requirements can be encoded by the following Sigma type

$$\begin{array}{lllll} \mathit{MONOID} & := & \Sigma & T: & \mathit{Type}, \\ & & \Sigma & \mathit{op}: & T \to T \to T, \\ & & \Sigma & \mathit{unit}: & T, \\ & & \Sigma & \mathit{assoc\_ax}: & \mathit{associative\_p}\ \mathit{op}. \\ & & & & \mathit{neutral\_p}\ \mathit{op}\ \mathit{unit} \end{array}$$

or, similarly, by the following TYPELAB specification

```
MONOID :=
  SPEC
     T : Type,
     op : T -> T -> T,
     unit : T,
     AXIOM assoc_ax : associative_p op,
     AXIOM unit_ax: neutral_p op unit
  END-SPEC
```

Here, `associative_p op` stands for `all(x,y,z:T) (op (op x y) z) = (op x (op y z))`, and similarly `neutral_p op unit`. Note that this specification combines type dependencies (*op* and *unit* depend on T) and value dependencies (the axioms depend on *op* and *unit*).

This description is satisfied by the natural numbers together with addition and 0 as neutral element. Together with terms *assoc_pr* and *neutr_pr* which encode proofs of the properties of associativity and neutrality, an element of type *MONOID* can be constructed:

$$\langle \mathit{Nat},\ +,\ 0,\ \mathit{assoc\_pr},\ \mathit{neutr\_pr} \rangle$$

Again, TYPELAB offers a more readable notation, which in particular permits to avoid an explicit mention of proof objects:

```
STRUCT  T:= Nat, op:= +, unit:= 0  END-STRUCT  :: MONOID
```

The coercion of this structure to type `MONOID` leads to the generation of *proof obligations* which can be discharged later on. Proof obligations are represented by *metavariables*, introduced in Section 1.2.1 below.

Since specifications are internal objects of the logic, it is possible to parameterize over specifications by simple function abstraction. For example, a specification of lists, `LIST`, can be parameterized over structures with a non-empty carrier, `ELEM`, as follows:

```
ELEM :=
  SPEC
     T : Type,
     t : T
  END-SPEC;

LIST := fun(E:ELEM)
  SPEC
     List : Type,
     nil : List,
     cons: E.T -> List -> List,
     ...
  END-SPEC;
```

When strengthening the parameter theory, for example by abstracting over `MONOID` instead of `ELEM`, specialized theorems can be stated and subsequently be proved. For example, the following theory states that for lists made up of elements of monoids, the fold function over lists "distributes" over the append function `++`:

```
LIST_FOLD := fun(M:MONOID)
  SPEC IMPORT LIST (MONOID_to_ELEM M) END-IMPORT
   THEOREM fold_distr_append:
    all(l1,l2:List)
      (fold M.op M.unit (l1 ++ l2))
    = (M.op (fold M.op M.unit l1) (fold M.op M.unit l2))
  END-SPEC;
```

The use of a `THEOREM` statement leads to the generation of a proof obligation, which in turn is represented by a metavariable. The mechanisms described in Sections 1.2.1 and 1.2.2 below can then be used to discharge the proof obligation.

Even though specification and refinement of programs is not the central concern of this thesis, it was one of the driving forces behind the conception of the TYPELAB system. In order to put the above example into perspective, let us briefly compare TYPELAB with similar language and system developments.

- Like many specification formalisms [Wir86, KBS91, Spe92], TYPELAB offers constructs for defining elementary specifications and operators for manipulating specifications, such as renaming, combination of specifications and extension of specifications with additional components. The latter is accomplished with an `IMPORT .. END-IMPORT` statement, as shown above.

- In this example, the specification of lists has been parameterized over structures of type `ELEM`. It should be noted that lists can as well be defined as polymorphic datatypes, see for example the use in Section 1.1.3. For a discussion of different kinds of parameterization, see [KBS91]. Parameterization over structures has the advantage that additional information can be conveyed in the parameter type. It has the disadvantage that coercion functions (such as `MONOID_to_ELEM`) have to be supplied, which adapt parameters (such as `M`, a `MONOID`) to the appropriate type (such as `ELEM`, the parameter type of `LIST`).

- Several other systems implement notions such as parameterized specification, refinement and realization of specifications etc., for example IMPS [FGT93], KIV [RSSB98], OBJ [GW88] and Specware [SJ94]. We are not aware of an implementation other than Typelab where these concepts have been internalized in the type system or logic.

1.1.5. Proof Assistants

In the following, we will briefly review the development of proof assistants for type theory. In a broad sense, systems such as PVS [ORS92], IMPS [FGT93] and logical frameworks in the LCF tradition (Isabelle [Pau94], HOL [GM93]) can be subsumed under this concept, since they are based on a typed lambda calculus, usually the simply-typed lambda calculus, possibly with some extensions such as predicate subtypes in PVS or type classes in Isabelle. Even though these extensions can make the calculi quite complex in that proof obligations have to be generated during type checking (as in the presence of semantic subtypes) or specialized type checking algorithms are employed [NP95], the user of such a system is not directly confronted with this complexity. Whereas much emphasis is laid on theorem proving techniques, for which ample support is provided, the construction of terms of a particular type is usually not addressed. Instantiation of existentially quantified variables has to be performed manually or is handled by (possibly higher-order) unification procedures, but not by explicit construction.

In a more restricted sense, type theoretic proof assistants aim at supporting construction of terms of a given type. Usually, the calculi under consideration here are quite complex, so that the activity of proof term construction encompasses standard theorem proving by the propositions-as-types principle mentioned above, but may go well beyond that, as for example the incremental development of a realization of a given specification (see Section 1.2.1).

The first prominent proof assistant for typed lambda calculi was implemented in the Automath project [dB70]. Even though this research made fun-

damental contributions such as a nameless representation of binding structures (de Bruijn indices, employed in many proof assistants, see also Section A.1) and a coding of logic in type theory (further developed in the logical framework LF [HHP87]), there currently do not seem to be any direct successors of Automath.

The Nuprl system [Con86] is an implementation of a predicative version of Martin-Löf's type theory. Apart from additions such as quotient types and semantic subtypes which make the type system inherently undecidable, the typing rules are presented in a form which is appropriate for proof development, but not for type checking, since type checking cannot proceed by structural decomposition of terms. Thus, type checking may lead to proof obligations requiring the well-formedness of terms or types to be shown. In fact, using the terminology of Chapter 3, the Nuprl type system is formulated as a sequent calculus. To each such sequent rule, an extract term is associated which gives a description of how to build up the witness term of the entire proof once the proof is finished. We will come back to a discussion of this topic in Chapter 3 and thus will not go into details here. Nuprl provides decision procedures for specialized theories such as integer arithmetic. General proof search is currently not supported.

The proof assistants Constructor [Hel91], Coq [Bar98], Lego [LP92] are based on variants of the Calculus of Constructions. They employ typing rules in the form of "introduction" and "elimination" rules, as exemplified by the ($\rightarrow$-Intro) and ($\rightarrow$-Elim) rules of Section 1.1.1, which directly yield an algorithm that decides type correctness of a term. Proof development mostly proceeds by successive application of "introduction" and "resolution" tactics. The introduction tactic corresponds to a backwards application of the introduction rule, it moves assumptions into the current context. Thus, a goal of the form $\Gamma \vdash t_0 : A \rightarrow B$ is transformed into $\Gamma, x : A \vdash t_1 : B$, where t_0 has the form $\lambda\, x : A.t_1$. The resolution tactic can best be compared to a single step in a Prolog-style proof derivation: Given a goal of the form $\Gamma \vdash t_0 : G$, where Γ contains a hypothesis of the form $h : A \rightarrow B$, an attempt is made to resolve $A \rightarrow B$ against G, by unifying B and G. If unification succeeds, a new goal $\Gamma \vdash t_1 : A$ is created.

Since the Π type constructor, the dependent analogue of the function space constructor, permits to encode all logical connectives, the seemingly elementary introduction-resolution proof style is in fact quite expressive, for a sufficiently powerful unification procedure. However, this method of proof search is often unsatisfactory from a practical viewpoint. Therefore, the proof search used in TYPELAB (described in Section 4.4) is based on the standard connectives of logic.

In addition to the introduction and resolution tactics, the Coq system con-

tains several specialized tactics, for example for induction, integer arithmetic etc.

Constructor, Lego and an earlier version of Coq support proof term construction with the aid of metavariables, which are not secure, however, in that they do not cope with the problems raised in Section 2.2. A mechanism for securely integrating metavariables in Coq is under development, based on the theory elaborated by Muñoz [Muñ97] and discussed more in detail further below.

Whereas the degree of automation of proof search is rather low in the systems mentioned so far, the "introduction" and "resolution" proof style is taken as the basis for automated proof search in the Elf system [Pfe91a], in extension of concepts from higher-order logic programming [NM88]. Elf provides a specialized unification procedure [Ell89] for its underlying logic LF, a dependently-typed calculus where values may depend on terms, but there is no abstraction over types.

The Alf system [MN94] is based on a monomorphic type theory, where, however, variables in abstractions do not carry type annotations, as in the term $\lambda\, x.x$. This "Curry-style" type theory permits a limited form of polymorphism, but also raises particular problems, since types have to be reconstructed during type checking. Apart from Typelab, Alf is so far the only type theoretic proof assistant which incorporates metavariables for proof construction in a theoretically sound way. The primary method of proof construction is successive instantiation of metavariables by terms which possibly contain fresh metavariables; there is however hardly any automation available in the form of proof search procedures.

1.2. Methods

The above sections have given a motivation for the interest in examining complex type structures, and they have delineated some of the theoretical boundaries, such as undecidability of type inhabitation. In order to convey a feeling for the applicability of the methods developed in the sequel, the following two subsections will informally discuss some of the techniques that facilitate construction of objects and that are used to carry out proof search in type theories. These techniques will be analyzed in detail in the remainder of this thesis, with a stronger emphasis on difficulties and problems from which we largely abstract in this introduction.

1.2.1. "Construction": Metavariables

In order to construct an element of a given type, a *metavariable* is introduced which stands for an as yet unknown object of that type. The metavariable can be instantiated with an appropriate term which is either a complete solution or, more interestingly, a term which itself contains metavariables. The process continues until all metavariables of a partial solution have been instantiated. Most frequently, an instantiation of a metavariable is not performed explicitly, but rather occurs behind the scenes as the result of carrying out an operation on the current proof goal.

We illustrate the principle by an "interactive" construction of an element of type `MONOID`, as it would be carried out in TYPELAB (cf. Section 1.1.4). This kind of activity is rather far removed from what can be done with a traditional theorem prover, because the proof obligation is not a proposition, but a type (in this case a specification). In the course of the development, however, some propositional subgoals have to be solved.

The initial proof obligation is:

```
  ...
|---------------------------------------
  ?M:MONOID
```

As a syntactical convention, metavariables are named by identifiers that start with a question mark. Assumptions are displayed above the stylized turnstile (currently there are none), the goal itself is shown below it. The global context, indicated by `...` and not shown here, contains definitions such as `MONOID`. After unfolding this definition and decomposing the specification, five new goals are created, one for each component of the specification. The first goal, for example, is

```
  ...
|---------------------------------------
  ?T:Type
```

An intermediate solution of the original metavariable is then given by

```
STRUCT
    T:= ?T, op:= ?op, unit:= ?unit,
    assoc_ax:= ?assoc_ax, unit_ax:= ?unit_ax
END-STRUCT
```

Instead of attempting to solve the goals in the given order, we turn to the second goal:

```
  ...
  T := ?T
|-------------------------------------
  ?op: T -> T -> T
```

This second goal contains the definition `T := ?T` in its local context, so after expansion of the definition, one obtains the goal `?op: ?T -> ?T -> ?T`. The dependency on the level of types leads to a dependency of metavariables: Metavariable `?op` depends on metavariable `?T`. Conversely, a solution of `?op` (for example with addition `+` on natural numbers) determines the solution of `?T`.

Thus, the following command, which directly instantiates metavariable `?op` with `+`, also leads to an instantiation of `?T` with `Nat`. The next open goal requires construction of the unit:

```
tlab? axiom + ;
  ...
  T  := Nat
  op := +
|-------------------------------------
  ?unit: T
```

We can now instantiate `?unit` with `0`, which gives

```
STRUCT
    T:= Nat, op:= +, unit:= 0,
    assoc_ax:=?assoc_ax, unit_ax:=?unit_ax
END-STRUCT
```

as a partially instantiated solution of the original metavariable `?M`. Thus, we still have the obligation to prove associativity of addition and neutrality of 0.

Here, we will not carry out these proofs in detail. Instead, assume that we delay the instantiation of `?unit` and directly attack the problem of proving neutrality of a still unknown unit with respect to the binary operation `+`. We will see that after a few transformations, this proof leads to a state which directly permits to read off an appropriate instantiation for `?unit`. Thus, the initial proof obligation is (in the following, we omit some local context entries):

```
  ...
|-------------------------------------
  ?unit_ax: neutral_p + ?unit
```

After simple manipulations like expansion of definitions of `neutral_p` and `+`, we reach the goal

```
  ...
|--------------------------------------
  ?1: all(x:Nat) (iter_Nat ?unit succ x) = x
```

Here, `iter_Nat` is an iterator over the natural numbers with a behaviour analogous to primitive recursion:

- `(iter_Nat f0 fs 0)` reduces to `f0`
- `(iter_Nat f0 fs (succ n))` reduces to `(fs (iter_Nat f0 fs n))`

Induction on the variable `x` gives the following base case:

```
  ...
|--------------------------------------
  ?2: (iter_Nat ?unit succ 0) = 0
```

Reduction of the iterator leaves the goal

```
  ...
|--------------------------------------
  ?3: ?unit = 0
```

which can immediately be solved by reflexivity of equality. It is equally simple to deal with the remaining proof obligations.

1.2.2. "Deduction": Proof Search

In the following, we will present an example which corresponds more closely to what is traditionally understood by proof search. The example has been chosen so that it highlights some peculiar aspects of the logic and their consequences for proof search:

- Even though the logic which is embedded into the Calculus of Constructions is intuitionistic, classical logic can be obtained by assuming that a principle of excluded middle holds for all propositions. This principle, which in weaker logics could only be described schematically, can here be expressed by the formula $\forall\, P : \mathit{Prop}.P \vee \neg P$. In order to take effect, an appropriate instantiation of this formula has to be found during proof search, which will again be achieved through the use of metavariables.

- The provability of a proposition can depend on the existence of elements of types. Even if proof search has been completed for the "propositional" part of a proof goal, proof search may have to continue to construct elements of certain types. This part of proof search obeys the same principles as propositional proof search.

The (intuitionistically not valid) proposition we want to prove now is

$$(\forall\, x : T_1.\, \exists\, y : T_2.(P_1\ x) \vee (P_2\ y)) \ \rightarrow\ (\exists\, y : T_2.\, \forall\, x : T_1.(P_1\ x) \vee (P_2\ y))$$

under the assumption that the principle of excluded middle holds and that there is an element a of type T_1 and a function $f : T_1 \rightarrow T_2$.

Thus, omitting the declarations of `T1`, `T2`, `P1` and `P2`, the initial goal is:

```
  h_EM: all(P:Prop) P or not P
  a: T1
  f: T1 -> T2
|--------------------------------------
  ?TH: (all(x:T1) exists(y:T2) (P1 x) or (P2 y))
        -> (exists(y:T2) all(x:T1) (P1 x) or (P2 y))
```

The local context of this goal contains not only propositions (e.g. excluded middle), but also declarations of constants, whose purpose in the proof process will become clear further below.

During proof search, the antecedent of the implication is first moved into the context. As mentioned above, the hypothesis `h_EM` has to be instantiated appropriately, so in a next step, the universally quantified variable `P` of hypothesis `h_EM` is replaced by a metavariable `?P`:

```
  h_EM : all(P:Prop) P or not P
  a: T1
  f: T1 -> T2
  h: all(x:T1) exists(y:T2) (P1 x) or (P2 y)
  h_EM_1: ?P or not ?P
|--------------------------------------
  ?1: exists(y:T2) all(x:T1) (P1 x) or (P2 y)
```

A case distinction on hypothesis `h_EM_1` yields two new subgoals, one for `?P`, the other for its negation. The first branch of the proof

```
  ...
  h_P: ?P
|--------------------------------------
  ?2: exists(y:T2) all(x:T1) (P1 x) or (P2 y)
```

can immediately be solved by unification, leading to an instantiation of metavariable `?P`. Note that the instantiation term for `?P` is a complex proposition and not just an ordinary term. The proof then continues with the second branch, which (under the instantiation for `?P`) reads as follows:

```
  h_EM : all(P:Prop) P or not P
  a: T1
  f: T1 -> T2
  h: all(x:T1) exists(y:T2) (P1 x) or (P2 y)
  h_n_P: not (exists(y:T2) all(x:T1) (P1 x) or (P2 y))
|--------------------------------------
  ?3: exists(y:T2) all(x:T1) (P1 x) or (P2 y)
```

Now, the existentially quantified variable `y` of the proof goal is replaced by a metavariable `?y`. The first of the resulting subgoals

```
  h_EM : all(P:Prop) P or not P
  a: T1
  f: T1 -> T2
  h: all(x:T1) exists(y:T2) (P1 x) or (P2 y)
  h_n_P: not (exists(y:T2) all(x:T1) (P1 x) or (P2 y))
|--------------------------------------
  ?4: all(x:T1) (P1 x) or (P2 ?y)
```

can now be solved by standard reasoning, but does not lead to an instantiation of `?y`, so the proof is not finished yet. We are left with the goal

```
  h_EM : all(P:Prop) P or not P
  a: T1
  f: T1 -> T2
  h: all(x:T1) exists(y:T2) (P1 x) or (P2 y)
  h_n_P: not (exists(y:T2) all(x:T1) (P1 x) or (P2 y))
|--------------------------------------
  ?y: T2
```

Implication (as in the original goal formula) and the function type constructor (as in the type of `f`) are identified in the underlying logic and can therefore be handled similarly: By applying rules reminiscent of propositional reasoning to the hypotheses `a` and `f`, we succeed in proving the goal `T2`, at the same time constructing the term `(f a)` as instantiation for `?y`.

A precise definition of the rules and some more examples of proof search will be given in Chapter 4, after some technical terminology has been introduced.

1.3. Survey of this thesis

The main body of this thesis is divided into three chapters, a concluding chapter summarizes the most important results and suggests topics of future research, and an appendix collects some rather involved proofs. Let us now give an outline of these chapters, with an emphasis on the main contributions and a comparison with related work. A more detailed discussion will be given in conjunction with the technical presentations in later sections.

1.3.1. Metavariables

Chapter 2 is devoted to the study of a calculus with metavariables. A metavariable $?n$ is a placeholder for a term t to be constructed during a derivation or a proof. The expected solution term t is constrained by two factors: by a context Γ in which t has to be well-typed, and by the type T of t. Other constraints on solutions are conceivable, but are not an object of study in this thesis, because they can to a large extent be expressed within the logic itself.

There are mainly two difficulties that have to be faced when introducing metavariables in type theory:

- The computational behaviour of the λ-calculus has to be accounted for. In particular, β-reduction of a term containing a metavariable should not lead to an incorrect term. First solving a metavariable and then reducing a term should yield essentially the same result as first reducing the term and then solving the metavariable.

- The type dependencies are reflected by the fact that the type of one metavariable may depend on the value of another metavariable. Differently said, the solution of one metavariable may constrain the set of solutions of other metavariables.

Needless to say, these problems do not occur in logics with a simple type structure and without a notion of reduction, as for example many-sorted first-order predicate logic. The first of these problems is common to all type theories, including the simply-typed λ-calculus, the second problem only arises in dependently typed calculi.

In order to provide the terminology for a technical discussion, we begin with a short summary of the Extended Calculus of Constructions on which the further development is based (Section 2.1). The questions sketched above are then further illustrated with some examples in Section 2.2.

The calculus with metavariables is introduced in two stages, which separates the concerns for the computational behaviour (term calculus, Section 2.3)

from issues related to typing. Computation, i.e. β-reduction, is essentially performed by substituting the arguments of a function into the body of the function. Substitutions into metavariables cannot be carried out immediately, but have to be delayed until a solution for the metavariable is available. For this purpose, the notion of substitution is internalized in the calculus as a separate term constructor, in a form which is tailored to our specific needs. In particular, "delayed substitutions" can only be attached to metavariables, in the form $?n^\frown\sigma$, where $?n$ is a metavariable and σ a substitution, and not to arbitrary terms, as for example in $(f\ a)\sigma$. This distinguishes our calculus from other calculi with explicit substitutions (see [Les94] for a survey) which have been defined with the intention of providing an abstract machine model for evaluation of functional programs. Due to the limitations we impose, properties such as confluence of reduction hold in our calculus, which are impossible or difficult to achieve in a more general setting.

As remarked before, metavariables can depend on one another in that a metavariable occurs in the type or context of another metavariable. Typing rules can sensibly only be defined if circular dependencies between metavariables are excluded, that is, if there is a well-founded partial order among the metavariables. Sets of metavariables satisfying this property, so-called *valid proof problems*, are examined in Section 2.4. For valid proof problems, typing rules are introduced in Section 2.5. It is not immediately obvious that type checking under these rules is decidable, because at any given moment, several rule applications are possible, some of which lead into dead ends. By applying rules in a predetermined order, however, this difficulty can be avoided, type checking is shown to be decidable indeed.

A metavariable $?n$ cannot be solved by an arbitrary term, but only by a term satisfying the typing constraints imposed by the context Γ and the type T of $?n$. This idea is made precise in Section 2.6, and it is shown that with this notion of "solution", the problems identified at the outset disappear. In particular, instantiation of metavariables and reduction commute and typecorrect instantiations cannot lead to ill-typed terms.

The chapter concludes with a discussion of a different encoding of scopes, which can be found in some proof assistants essentially based on the simply-typed λ-calculus, for example Isabelle [Pau94] or HOL [GM93]. Instead of having a metavariable $?n$ depend on a context Γ, the general idea of this encoding is to represent $?n$ by a function taking the variables declared in Γ as formal parameters. It is shown that both representations are equivalent from a theoretical perspective, but it is argued that, in practice, the functional encoding is less appropriate in our particular setting. The discussion yields, as a side-effect, a rather short proof of strong normalization of the calculus with metavariables and explicit substitutions.

Comparison: There is a plethora of calculi with explicit substitutions, and it would not be sensible to attempt an individual appreciation of all of them. As briefly mentioned above, most of these calculi have been designed with the intention of providing abstract machine models of functional programs, but not with the intention of coping with problems arising in proof assistants based on typed λ-calculi. Consequently, most of the calculi with explicit substitutions are framed in the setting of an untyped or simply-typed language, which considerably facilitates all questions related to typing. Conversely, the term calculus becomes more complex if explicit substitutions can occur in arbitrary term positions, and are not only attached to metavariables. In this setting, terms have the form $t\sigma$, where σ is a substitution and t is a variable, an application or λ-abstraction or itself of the form $t'\sigma'$.

The calculi differ in the precise definition of terms, substitutions and reduction. Questions of central concern are confluence (can different sequences of reduction be joined to yield the same result?) and strong normalization (do all sequences of reduction terminate?). One of the earliest calculi, $\lambda\sigma$ [ACCL91], was shown to be confluent, but only for closed terms. Furthermore, a variant corresponding to the simply-typed λ-calculus was shown not to be strongly normalizing [Mel95]. The calculus $\lambda\nu$ [Les94], for example, avoids this defect by introducing different operators to build up substitutions, but it is still not confluent on open terms. The last step in a series of gradual improvements is $\lambda\zeta$[Muñ96], a calculus which is both confluent on open terms and which preserves strong normalization. A disadvantage of the calculus is that before propagating a substitution in a term, the term has to be traversed in order to be "marked" in a certain fashion. This destroys most efficiency gains of explicit substitutions.

Two investigations are more closely related to ours, since they develop a calculus with explicit substitutions in a type theory: The calculus of Magnusson [Mag95] is the basis of the Alf system, the calculus of Muñoz [Muñ97] will apparently be integrated into Coq. These two approaches will be discussed in the following.

As mentioned in Section 1.1.5, the Alf system is based on Martin-Löf's monomorphic type theory, presented in a Curry-style calculus. In addition to β-reduction, there is a notion of reduction of functions defined by primitive recursion over inductive datatypes. The primary method of proof construction is successive instantiation of metavariables by terms which possibly contain other metavariables. In [Mag95], a calculus with explicit substitutions is presented. In this calculus, variable identities are not expressed by a nameless representation in the style of de Bruijn indices, but by variables with names. In order to avoid name clashes, explicit substitutions are not moved below abstractions, thus terms of the form $(\lambda\, x.b)\sigma$, with σ a substitution, are irreducible.

The main contribution of Magnusson's thesis [Mag95] is to operationalize the abstract presentation of the calculus, by deriving algorithms such as type checking, unification and instantiation of metavariables, and proving them correct. However, metatheoretic properties of the calculus itself are not investigated. Indeed, it seems that the calculus is not confluent – for example, the term $(\lambda f.f\{x := a\})\ (\lambda z.z)$ can be reduced to $\lambda z.z$ by first reducing the substitution, or to $(\lambda z.z)\{x := a\}$ by β-reduction, which, as remarked above, is a normal form.

Magnusson's work provides interesting results for the style of proof construction supported by Alf. For example, a "local undo" operation – as opposed to a chronological undo – permits to retract instantiations of metavariables by taking into account their logical interdependence. It is however not clear how one of the main objectives of this thesis, namely a greater degree of automation of proof construction, could be achieved in Magnusson's framework.

The calculus of Muñoz [Muñ97], developed in parallel with and independently from this work, is based on the same logic as this thesis (up to some differences which are negligible in this context), namely the Calculus of Constructions. The main differences between Muñoz's calculus (not to be confused with [Muñ96], an untyped calculus) and the one presented in the following are:

- Explicit substitutions can be attached to arbitrary term constructors, not only to metavariables as in our case. For type checking to be decidable, typing information has to be attached to substitutions, which is not necessary in our case (see the discussion in Section 2.5).

- The term calculus is confluent, but the construction of a reduction sequence which does not terminate, even for well-typed terms, can be adapted from [Mel95]. However, it can be shown that reduction terminates when following a certain strategy, which gives the resulting calculus a weak normalization property. Even though our calculus is not constrained to one particular reduction strategy, the accumulation of substitutions within substitutions, one of the main reason for nontermination in the construction by Melliès [Mel95], is excluded.

- As mentioned above and further elaborated in Section 2.4, circular dependencies between metavariables have to be excluded when defining typing rules for terms containing metavariables. In our calculus, this requirement is imposed by postulating a partial order between metavariables. In the calculus of Muñoz, metavariables are even ordered linearly in a so-called "signature", which gives a very elegant formulation of typing rules and a comparatively easy criterion for verifying the correctness of instantiations (cf. our approach in Section 2.6). In particular, a solution

term t for a metavariable $?n$ is only acceptable if all metavariables $?m$ occurring in t have been declared before $?n$ in the signature.

One of the drawbacks of this method is that completeness of proof search is unnecessarily restricted by the somewhat arbitrary order of metavariable declarations in the signature. For example, when trying to prove $\exists\, n : T.\, \exists\, m : T.n = (f\ m)$, the first step is to replace existentially quantified variables by metavariables, leading to the goal $?n = (f\ ?m)$, with metavariable $?m$ occurring behind $?n$ in the signature. According to the correctness criterion stated above, an instantiation $?n := (f\ ?m)$ is then impossible. Of course, by an "appropriate" permutation of the metavariables in the signature, the instantiation can still be carried out. Since proof search as envisaged in this thesis is not the primary objective of [Muñ97], these implications are not discussed, but it may be conjectured that the notion of appropriateness would resemble the criteria developed in Section 2.6.

1.3.2. Theory of proof search

Chapter 3 takes a first step towards automation of proof search, by converting the natural deduction-style definition of *ECC* into the form of a sequent calculus. For several reasons, the resulting calculus is not immediately practically applicable. Further modifications, described in Chapter 4, yield search procedures such as the ones implemented in the Typelab system.

The kind of proof problems to be dealt with in Chapters 3 and 4 are "type inhabitation problems". The question is to produce, for given context Γ and type A, a term M such that $\Gamma \vdash M : A$ holds. Here, A can be a "genuine" type or, by the propositions-as-types principle, a type encoding a proposition. As mentioned further above (Section 1.1.3), this question is in general not decidable.

Usually, proof search proceeds by applying rules backwards, starting from the goal, until some form of axiom is reached. This method is not practicable for the standard definition of *ECC* in the form of a natural deduction calculus, because the rules corresponding to elimination rules, such as ($\rightarrow$-Elim), do not have a subformula property:

$$\frac{\Gamma \vdash f : A \rightarrow B \quad \Gamma \vdash a : A}{\Gamma \vdash (f\ a) : B}\ (\rightarrow\text{-Elim})$$

When applying the rule backwards, the type A would have to be guessed, which would lead to an infinite branching factor at this node of the search tree.

The idea of a sequent-style calculus is to replace the scheme of introduction and elimination rules by a scheme of left- and right-rules, which permit to

decompose formulae both in the antecedent and the succedent of a judgement. For example, the left-rule corresponding to (→-Elim) is

$$\frac{\Gamma, h : A \to B, \Gamma' \vdash a : A \quad \Gamma, h : A \to B, \Gamma', h' : B \vdash M : G}{\Gamma, h : A \to B, \Gamma' \vdash M\{h' := (h\ a)\} : G}\ (\to L)$$

A sequent-style calculus corresponding to *ECC* is formally defined in Section 3.1.

The general idea of transforming a natural deduction into a sequent calculus goes back to Gentzen [Gen34], and so does the general technique of demonstrating the equivalence of both calculi, by first introducing an intermediate calculus with the "cut" rule and then showing how this rule can be eliminated from derivations (see the more detailed introduction in Section 3.1.1). In the given situation, matters are complicated by at least the following factors:

- The logic is higher-order, with, among others, functions that produce types. Computation steps have to be interleaved appropriately with the rules that decompose type constructors, since type constructors may be hidden below β-redexes, as for example in $(\lambda\, X : \mathit{Prop}.X \to B)\ A$.

- The logic is dependently typed. This makes the cut-elimination proof more complex, as some of the structural rules of predicate logic are not available. In particular, hypotheses $x : T_1, y : T_2$ in a sequent cannot be exchanged, since T_2 may depend on x.

Section 3.2 examines more closely the relation between the natural deduction version ECC_N of *ECC* and the sequent-style version (called ECC_G for historical reasons). It should be emphasized (cf. the remarks in Section 1.1.2) that the calculus ECC_N can be understood as providing a particular form of realizability semantics for ECC_G, and the aim of Section 3.2 is to investigate correctness and completeness of ECC_G with respect to this semantics.

Correctness of ECC_G is rather easy to show for the whole calculus, by a simulation of ECC_G-proofs in ECC_N. In order to establish the completeness of ECC_G, a cut elimination proof is carried out (see Section 3.2.2 and details of the proof in Appendix A.2). Cut elimination is a procedure which pushes applications of the cut formula towards the leaves of a proof tree. To show that this procedure terminates, terms have to become "smaller" (in a well-defined sense) with each cut elimination step. The proof is first carried out for an abstract measure function m on terms which has the desired properties.

Unfortunately, it turns out to be impossible to effectively define such an m for the whole calculus *ECC*. Section 3.3 is devoted to identifying appropriate fragments of *ECC* for which a measure m can be defined. A classification of

subsystems of the Calculus of Constructions (essentially *ECC* without Σ-types) is provided by the "λ-cube", and a closer analysis of these systems shows that for all *predicative* systems of the λ-cube, a measure function on terms having the required properties is definable. Thus, to summarize, the sequent calculus ECC_G is correct for all of *ECC*, but complete only for predicative subsystems.

All of Chapter 3 examines a calculus without metavariables, as this permits a better comparison of calculi. Metavariables are introduced again in Chapter 4 as a device to delay the choice of how to instantiate existentially quantified variables.

Comparison: Proof search in type theory has been a research topic for quite a while [Hue73, Hue75, And86, Koh98]. The object of study is usually the simply-typed λ-calculus, and since most of the difficulties of the present investigation arise from dependent typing, a direct comparison with these approaches is not appropriate.

For the type theory of Nuprl, Harper [Har85] develops several calculi, starting from a natural deduction calculus close to a formulation originally given by Martin-Löf and ending with a sequent calculus which has subsequently been adopted as the "standard" Nuprl logic [Con86]. In [Har85], only one part of the equivalence proof (corresponding to "correctness") is given. The motivation for defining this calculus is, similar as in our case, to obtain a calculus which is appropriate for proof development. However, since Nuprl does not support metavariables, the calculus is not used for automated proof search, where the instantiation of existentially quantified variables can be delayed, but only for interactive proof refinement, where instantiations are provided explicitly.

In [Pym90, PW91], Pym examines proof search in the type theory LF [HHP87], one of the subsystems of the Calculus of Constructions (see Section 3.3.1) in which types can depend on terms but which does not allow quantification over types. Apart from the natural-deduction style definition of LF, Pym presents a sequent calculus subject to the restriction that (ΠL)-rules operate on types, but not on kinds (type universes in the terminology of Section 2.1). Thus, the (ΠL)-rule can decompose $\Pi x : A.B$, with $A : \mathit{Type}$ and $B : \mathit{Type}$, but not $\Pi x : A.\mathit{Type}$. Completeness of the sequent calculus is proved by a "direct" method without an intermediate system containing the cut rule, by examining the structure of terms derived in the sequent calculus.

(An aside concerning the proof search techniques as further developed in [Pym90]: For carrying out proof search, metavariables are introduced, which can be solved by a higher-order unification procedure extended to LF. Instead of the notion of correctness of instantiations presented in Section 2.6, a criterion is developed which is based on permutability of inferences in the sequent calculus. This approach has the advantage of not making premature decisions on the order in which rules are applied, thus leading to less backtracking in

proof search and a smaller search space. It would be interesting to empirically evaluate this procedure; however, we are not aware of an implementation. Furthermore, we are not sure how substitutions into terms containing metavariables are handled, since there is no notion of explicit substitutions.)

Dowek [Dow93] gives a complete proof search procedure for the type systems of the λ-cube and thus in particular for the Calculus of Constructions. It owes much to the methods developed by Huet [Hue73, Hue75] for the simply-typed λ-calculus, and it has found its way into some proof assistants by means of the "introduction" and "resolution" proof tactics mentioned in Section 1.1.5. Completeness of proof search is shown by induction on the size, a complex measure not coinciding with the number of symbols, of the proof term. (Again a comment on a technicality: In [Dow93], a functional encoding of scopes of existential variables, as described in Section 2.7, is used.)

Returning to the sequent system presented in this thesis: The fact that completeness is only shown for a fragment of the calculus in the proof of Section 3.2.2 is no evidence that the sequent system is indeed incomplete for the whole calculus. It is interesting to observe that the completeness proofs of the two approaches mentioned above are based on the *term* constructed during proof search and not on the *types* occurring in the antecedent and succedent of the proof goal, as in the cut elimination argument, and it might be worthwhile to attempt a completeness proof along these lines. The motivation for carrying out a cut elimination proof was its easy scalability to other type constructors. In particular, a sequent system of practical relevance that includes an encoding of the usual logical connectives, such as the one in Chapter 4, could be dealt with by a cut elimination proof, whereas it is hard to see how an argument involving proof terms could be set up.

1.3.3. Pragmatics of proof search

Chapter 4 is a synthesis of the preceding chapters, i.e. a combination of the calculus with metavariables of Chapter 2 and the proof search methods developed in Chapter 3. Altogether, the style of this chapter is less formal, its purpose is to show how proof search in type theory can be put to work.

Given a goal G, to be proved in a context Γ, the most immediate first step is to introduce a metavariable $?n_0$ for the proof term to be constructed and then apply rules of the sequent calculus in a backward-chaining fashion to the judgement $\Gamma \vdash ?n_0 : G$. Each rule application leads to an instantiation of the metavariable with a proof term which possibly contains new metavariables. This process continues until all open metavariables have been instantiated, either directly or through unification.

The examples presented in Section 4.1 demonstrate the general procedure,

but also illustrate that the proof rules of Chapter 3 generate a search space which is difficult to control. The remainder of Chapter 4 will be concerned with refining the general procedure by analyzing the behaviour of individual rules, until practically useful search algorithms are obtained.

Proof search has two complementary aspects: Application of sequent rules and unification. These aspects are intertwined in the calculus of Chapter 3. In Section 4.2, the original calculus is transformed in such a way that the role of unification becomes apparent and requirements for a unification procedure can be stated.

Section 4.3 examines unification more closely, beginning with a unification procedure which is "first-order" in that it structurally compares terms without taking into account β-equality. However, it respects binding structure (α-equality) and permits to solve equations not only between terms, but also between types, and thus differs considerably from standard first-order unification. Successful unification of two terms produces an instantiation, which is shown to have the properties postulated in Chapter 2, so unification (and similarly the other proof methods, for which analogous results are shown) is indeed a correct implementation of the proof construction methods of Chapter 2.

In Sections 4.3.3 and 4.3.4, higher-order extensions are outlined: Whereas the first-order unification procedure can only handle "simple" equations $?n \stackrel{?}{\simeq} t$ between a metavariable and a term, it cannot deal with equations of the form $?n^\frown\sigma \stackrel{?}{\simeq} t$, where a delayed substitution σ is attached to metavariable $?n$. By a translation of metavariables into a representation with functional encoding of scopes, as in Section 2.7, it will be shown that this latter kind of unification problem is equivalent to a standard higher-order unification problem. By a similar translation, a correspondence with a special case, so-called "patterns", is established in Section 4.3.4. With the mappings defined in these sections, it is in principle possible to reduce unification problems in our representation of metavariables to a form where standard higher-order unification algorithms can be applied. The discussion in Section 4.3.5 identifies problems specific to dependent typing which cannot be solved by moderate adaptations of existing algorithms. The algorithms presented in Section 4.3 are therefore not complete in a global sense, but only when restricted to certain fragments (which, apart from the case of patterns, are not formally made precise, however).

Section 4.4 focuses on proof search in a more traditional sense. Proof rules that have been recognized as leading to an uncontrollable growth of the search space in Section 4.1 are discarded, rules are introduced that permit a direct decomposition of the standard logical connectives (conjunction, disjunction) and quantifiers (existential) rather than requiring manipulation of their encodings as Π-types. The resulting system (see Section 4.4.1) bears strong resemblance

to Tableau calculi, even though the type-theoretic background imposes some adaptations. For example, the proof of an existential statement $\exists\, x : T.P(x)$ explicitly involves the construction of an element $?x$ of type T (although a solution for $?x$ is usually found by unification), and not only hypotheses, but also variables are recorded in the context. Tableau calculi are usually stated with an eigenvariable condition. In Section 4.4.2, it is shown that the notion of typecorrect instantiation, developed in Section 2.6, is an adequate alternative to Skolemization, which is the standard technique to enforce this condition. Some optimizations of the proof rules are presented in Section 4.4.3. Even though these optimizations are well-known in presentations of intuitionistic sequent systems, they are interesting because they are applicable at all in our context and because it can be shown by a rather simple reasoning involving proof terms that they preserve completeness.

Comparison: The combination of an explicit substitution calculus with sequent style proof search in a type theory seems to be novel. We are not aware of a similar approach, neither presented in a theoretical study nor realized in an implementation. There is, however, a larger body of work on unification in typed λ-calculi and on proof search in intuitionistic logic, which we will review now.

Pym [Pym90] and Elliot [Ell89] describe higher-order unification in LF, and Pfenning [Pfe91b] outlines unification for "patterns" in the Calculus of Constructions. All these algorithms use a functional encoding of scopes and are similar in spirit to Huet's algorithm [Hue75] for the simply-typed λ-calculus. The algorithms are claimed to be complete, but these results apparently only encompass equations between terms, but not between types. A detailed discussion of this question follows in Section 4.3 and need not be repeated here.

A higher-order unification algorithm for the simply-typed calculus with explicit substitutions, $\lambda\sigma$, is described in [DHK95], a unification procedure for the analogue of "patterns", again for $\lambda\sigma$, in [DHKP96]. The unification algorithm obtained in [DHK95] is put into correspondence with Huet's algorithm for the simply typed λ-calculus by a mapping which resembles the inverse of our functional translation of metavariables in Section 2.7.

Recently, there is a surge of interest in theorem proving in (untyped) intuitionistic logic [Sha92, OK95, Tam96], however often with a restriction to the propositional fragment. Optimization of proof search is not the primary focus of this thesis, but constitutes an interesting topic for future research (see also Chapter 5).

We are only aware of two publicly accessible implementations [SFH92, Ott97], which are admittedly much more performant on standard benchmark problems than the current implementation in Typelab. Apart from the constant overhead incurred by an interactive system (e.g. for proof presentation),

this can be attributed to the rather weak strategies that are currently built into TYPELAB's proof search procedure. The discussion of Section 4.4.3 gives reason to believe that at least some of the optimizations developed for untyped intuitionistic logic can be incorporated in a type theoretic setting.

2. A Calculus with Metavariables

2.1. The Extended Calculus of Constructions

This section summarizes the most important facts about the Extended Calculus of Constructions, *ECC*, which is the formal basis for the following investigations. *ECC* [Luo90, Luo94] is an extension of the Calculus of Constructions, *CC* [CH88], with dependent Σ-types which are useful for representing specifications and mathematical theories (see Section 1.1.4 for a motivation). The results derived in the following sections for *ECC* could also be obtained for the subsystem *CC*. Conversely, passing from *CC* to *ECC* does not lead to a significant increase in complexity. We will restrict attention to *CC* alone whenever more appropriate for the purpose of comparison with other logical theories, as for example in Chapter 3.

For readability, we have chosen in this text a term representation which establishes identity of variables via names. There are well-known shortcomings of this representation, such as free variable capture (for example, the term $(\lambda\, x.\, \lambda\, y.x)\ y$ should not β-reduce to $\lambda\, y.y$). In order to avoid naming problems, we will always rename variables appropriately. A conceptually cleaner nameless representation of the calculus with de Bruijn-indices is given in Appendix A.1.

The material in this Section 2.1 is not new, apart from some minor definitions and propositions needed for later reference. It has mostly been taken from [Luo90], as well as from other sources cited throughout the text, which should also be consulted for proofs of propositions.

2.1.1. Base calculus – Term language

The terms of *ECC* are generated by the grammar of Figure 2.1. In this definition, $\mathcal{V}$ is a set of variables, *Prop* and Type_i, for $0 \leq i$, are type universes, which can be understood as collections of types. Dependent function spaces are formed by Π-abstraction, dependent record types by Σ-abstraction. The variables bound by Π or Σ are subject to the same scoping rules and naming conventions as variables bound by λ-abstraction. In particular, in $\Pi x : M.N$,

$$
\begin{array}{rcl}
\mathcal{T} & ::= & \mathcal{V} \\
& | & \mathit{Prop} \mid \mathit{Type}_i \\
& | & \Pi\mathcal{V} : \mathcal{T}.\mathcal{T} \mid \lambda\,\mathcal{V} : \mathcal{T}.\mathcal{T} \mid (\mathcal{T}\ \mathcal{T}) \\
& | & \Sigma\mathcal{V} : \mathcal{T}.\mathcal{T} \mid \mathit{pair}_{\mathcal{T}}(\mathcal{T},\ \mathcal{T}) \mid \pi_1(\mathcal{T}) \mid \pi_2(\mathcal{T})
\end{array}
$$

Figure 2.1.: Grammar defining the language of *ECC*

$\Sigma x : M.N$ and $\lambda\, x : M.N$, the variable x is bound in N, but not in M, and x can be renamed to y, so as to yield the terms $\Pi y : M.N\{x := y\}$ etc., in case y does not occur in N. Application of function f to argument a is written as $(f\ a)$. A pair of elements e_1 and e_2 is written as $\mathit{pair}_T(e_1,\ e_2)$, the type T of the resulting pair has to be added for reasons of type inference. The first and second component of a pair p can be obtained by projection $\pi_1(p)$ and $\pi_2(p)$, respectively.

Notational Conventions:

- Small letters from the end of the alphabet, such as x, y, z, usually stand for term variables, whereas capital letters S, T, X, Y, Z stand for type variables or types. Capital letters L, M, N from the middle of the alphabet denote arbitrary terms, whereas capital letters A, B, C often denote propositions or type terms.

- Consecutive λ- and Π-abstractions are abbreviated as usual. For example, $\lambda\, x_1 : T_1.\ \lambda\, x_2 : T_2.\ B$ becomes $\lambda\, x_1 : T_1, x_2 : T_2.M$. This term is sometimes still further abbreviated as $\lambda\,\Gamma.M$, if Γ is the list of variable declarations $x_1 : T_1, x_2 : T_2$.

- We omit redundant parentheses. The function type constructor $\rightarrow$ associates to the right, application to the left. In particular, repeated applications $(\cdots((M\ N_1)\ N_2)\cdots N_n)$ are simplified to $(M\ N_1\ N_2 \cdots N_n)$, sometimes the notation $f(a_1, \ldots, a_n)$ is used instead of $(f\ a_1 \ldots a_n)$.

- Whenever x does not occur in B, the function space $\Pi x : A.\ B$ is written as $A \rightarrow B$. Similarly, $\Sigma x : A.\ B$ is then written as the Cartesian product $A \times B$. Conversely, to emphasize that a variable x occurs in B, we write $B[x]$, for example in $\Sigma x : A.\ B[x]$. Substituting all occurrences of x by a term N is then written as $B[N]$.

- Whenever the exact level j of a type universe Type_j is of no interest, we simply write *Type*. Indeed, implementations of type checkers can deal

with this kind of ambiguity by building up a set of inequalities among type universe variables, as dictated by the typing rules, and raising a typing error if a consistent assignment of levels to these universe variables is not possible [HP91].

Definition 2.1 (Free Variables)
The set of *free variables* of a term M, $FV(M)$, is defined as follows:

$$\begin{array}{lcll}
FV(x) & = & \{x\} & \text{for } x \text{ a variable} \\
FV(Prop) & = & \varnothing & \\
FV(Type_i) & = & \varnothing & \\
FV(\mathcal{Q}x : A.\ M) & = & FV(A) \cup (FV(M) \backslash \{x\}) & \text{for } \mathcal{Q} \in \Pi, \lambda, \Sigma \\
FV((M\ N)) & = & FV(M) \cup FV(N) & \\
FV(pair_T(M,\ N)) & = & FV(T) \cup FV(M) \cup FV(N) & \\
FV(\pi_i(M)) & = & FV(M) & \text{for } i = 1, 2
\end{array}$$

The definition of *subterm* is standard; in particular, T and M are subterms of $\lambda\, x : T.M$, similarly for Π- and Σ-abstractions. The term constructors $Prop$, $Type_i$ and Π- and Σ-abstractions can be understood as types. For a structural analysis of proof systems (see in particular Chapter 3), it is interesting to determine, for a given type, the set of subterms that again are types. The adequacy of this definition is justified by the typing rules.

Definition 2.2 (Type Subterm)
The set of *type subterms* $TSubt(T)$ of a type term T is defined as follows:

$$\begin{array}{lcl}
TSubt(Prop) & = & \{Prop\} \\
TSubt(Type_i) & = & \{Type_i\} \\
TSubt(\Pi x : A.\ B) & = & \{\Pi x : A.\ B\} \cup TSubt(A) \cup TSubt(B) \\
TSubt(\Sigma x : A.\ B) & = & \{\Sigma x : A.\ B\} \cup TSubt(A) \cup TSubt(B) \\
TSubt(T) & = & \{T\} \text{ for all other term constructors}
\end{array}$$

The set of *proper type subterms* of T is defined as $TSubt(T) \backslash \{T\}$.

For example, in the term $\Sigma x : A.\ (f\ (\lambda\, y : C.\ D))$, the types A and $f\ (\lambda\, y : C.\ D)$ are type subterms, but not the type C.

Computational behaviour is achieved by β- and π-reduction, defined by $(\lambda\, x : T.\ M)\ N \rightarrow_\beta M\{x := N\}$ and $\pi_i(pair_T(M_1,\ M_2)) \rightarrow_\pi M_i\ (i = 1, 2)$, respectively. The definition of β-reduction is based on the concept of substitution $M\{x := N\}$ of a term N for variable x in term M. Since this topic is covered in depth in Section 2.3, we will refrain from further formalizations here.

Two terms M and N are called *convertible*, written as $M \simeq N$, if they can be transformed into one another by repeated applications of β- and π-reduction (see Definition 2.29). Given a notion of convertibility $\simeq$, we can define the notion of *cumulativity*, which will be used further below in the typing rules:

Definition 2.3 (Cumulativity)
The cumulativity relation $\preceq$ is defined as the smallest relation over terms such that:

1. If $M \simeq N$, then $M \preceq N$
2. If $M \preceq N$ and $N \preceq M$, then $M \simeq N$
3. If $L \preceq M$ and $M \preceq N$, then $L \preceq N$
4. $\mathit{Prop} \preceq \mathit{Type}_0$ and $\mathit{Type}_i \preceq \mathit{Type}_{i+1}$ for $i \geq 0$
5. If $A_1 \simeq B_1$ and $A_2 \preceq B_2$, then $\Pi x : A_1.\ B_1 \preceq \Pi x : A_2.\ B_2$
6. If $A_1 \preceq B_1$ and $A_2 \preceq B_2$, then $\Sigma x : A_1.\ B_1 \preceq \Sigma x : A_2.\ B_2$

Loosely speaking, cumulativity formalizes a subset relation between type universes and function spaces and Cartesian products built on top of universes. Clauses 1, 2 and 3 state that $\preceq$ is a partial order based on $\simeq$. Clause 4 formalizes the containment relation between universes: When viewed as sets, $\mathit{Prop} \subseteq \mathit{Type}_0 \subseteq \mathit{Type}_1 \ldots$. Clauses 5 and 6 extend this notion to Π- and Σ-types.

2.1.2. Base calculus – Typing

Definition 2.4 (Context)
A context is a finite list of declarations of the form $x_i : A_i$, where each x_i is a variable, each A_i a term and the x_i are mutually distinct. The empty context is denoted by $\langle\rangle$. The concatenation of two contexts Γ and Δ is written as Γ, Δ, adding a declaration $x : A$ to a context Γ as $\Gamma, x : A$.

The set $FV(\Gamma)$ of free variables of a context $\Gamma \equiv x_1 : A_1, \ldots, x_n : A_n$ is defined to be $\bigcup_{1 \leq i \leq n}(\{x_i\} \cup FV(A_i))$

The *domain* of a context $x_1 : A_1, \ldots, x_n : A_n$ is, depending on the situation under consideration, the list $[x_1, \ldots, x_n]$ or the set $\{x_1, \ldots, x_n\}$.

We sometimes used standard set notation in connection with contexts. For example, context Δ is a *subcontext* of context Γ, written $\Delta \subseteq \Gamma$, if every $x_i : A_i$ occurring in Δ occurs in Γ. If $x : T$ is a declaration in Γ, we write $(x : T) \in \Gamma$.

Definition 2.5 (Judgement)
A judgement is an expression of the form $\Gamma \vdash M : A$, where Γ is a context and M and A are terms, or it is an expression of the form $\Gamma\ \mathit{valid}_c$.

The judgement form $\Gamma \vdash M : A$ expresses that term M has type A in context Γ. Since the well-formedness of contexts cannot be assured by purely syntactic means, as for example in the simply-typed λ-calculus, a separate judgement form $\Gamma\ \mathit{valid}_c$ is required. Its main purpose is to ensure that for all the declarations $x : A$ in Γ, A is indeed a type. An alternative formulation of typing rules codes the judgement $\Gamma\ \mathit{valid}_c$ as $\Gamma \vdash \mathit{Prop} : \mathit{Type}_0$. This version is, in the author's opinion, less intuitive, but sometimes more suitable for proof theoretic studies and will be used in Section 3.3.

The typing rules of *ECC* are shown in Figure 2.2. The rules can be categorized as follows:

- Rules (Cempty) and (Cvalid) formalize the construction of contexts.
- Rules (UProp) and (UType) define how type universes can be typed.
- Rule (var) defines typing of variables.
- Rule ($\preceq$) formalizes the intuition of relation $\preceq$ as a generalization of set containment already mentioned above (following Definition 2.3).
- Rules (Π-Form$_1$), (Π-Form$_2$) and (Σ-Form) are *type formation* rules, defining how Π- and Σ-types can be constructed. Alternatively, they can be understood as defining how elements of the type universes *Prop* and Type_i are built.
- Rules (λ) and (pair) are *introduction* rules, describing how canonical elements of Π- and Σ-types are formed.
- Rules (app), (π_1) and (π_2) are *elimination* rules, defining how elements of Π- and Σ-types can be applied.

Considering, in addition, the β- and π-reduction relations given above, there is an obvious parallel between these rules and standard expositions of abstract data types. In particular, λ-abstraction and pairing can be conceived as the "constructors" of the "data types" Π and Σ, whereas application and projection are the "selectors". The reduction rules describe the equalities arising from applying selectors to constructors. The implications of defining the typing relation in the style of introduction / elimination rules are further explored in Chapter 3.

$$\frac{}{\langle\rangle\ \mathit{valid}_c}\ (\text{Cempty})$$

$$\frac{\Gamma \vdash A : \mathit{Type}_j \qquad x \notin FV(\Gamma)}{\Gamma, x : A\ \mathit{valid}_c}\ (\text{Cvalid})$$

$$\frac{\Gamma\ \mathit{valid}_c}{\Gamma \vdash \mathit{Prop} : \mathit{Type}_0}\ (\text{UProp}) \qquad \frac{\Gamma\ \mathit{valid}_c}{\Gamma \vdash \mathit{Type}_j : \mathit{Type}_{j+1}}\ (\text{UType})$$

$$\frac{\Gamma, x : A, \Gamma'\ \mathit{valid}_c}{\Gamma, x : A, \Gamma' \vdash x : A}\ (\text{var})$$

$$\frac{\Gamma, x : A \vdash P : \mathit{Prop}}{\Gamma \vdash \Pi x : A.P : \mathit{Prop}}\ (\Pi\text{-Form}_1)$$

$$\frac{\Gamma \vdash A : \mathit{Type}_j \quad \Gamma, x : A \vdash B : \mathit{Type}_j}{\Gamma \vdash \Pi x : A.B : \mathit{Type}_j}\ (\Pi\text{-Form}_2)$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda\, x : A.M : \Pi x : A.B}\ (\lambda)$$

$$\frac{\Gamma \vdash M : \Pi x : A.B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B\,\{x := N\}}\ (\text{app})$$

$$\frac{\Gamma \vdash A : \mathit{Type}_j \quad \Gamma, x : A \vdash B : \mathit{Type}_j}{\Gamma \vdash \Sigma x : A.B : \mathit{Type}_j}\ (\Sigma\text{-Form})$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B\,\{x := M\} \quad \Gamma, x : A \vdash B : \mathit{Type}_j}{\Gamma \vdash \mathit{pair}_{\Sigma x : A.\ B}(M,\ N) : \Sigma x : A.\ B}\ (\text{pair})$$

$$\frac{\Gamma \vdash M : \Sigma x : A.B}{\Gamma \vdash \pi_1(M) : A}\ (\pi_1) \qquad \frac{\Gamma \vdash M : \Sigma x : A.B}{\Gamma \vdash \pi_2(M) : B\,\{x := \pi_1(M)\}}\ (\pi_2)$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash A' : \mathit{Type}_j \quad A \preceq A'}{\Gamma \vdash M : A'}\ (\preceq)$$

Figure 2.2.: Rules of the calculus *ECC*

According to rule ($\preceq$), a term M may have several types A and A'. However, it can be shown that whenever M is typeable, it has a uniquely determined "principal type":

Definition 2.6
A is called a *principal type* of M in Γ if and only if

- $\Gamma \vdash M : A$ and
- for any A', $\Gamma \vdash M : A'$ if and only if $A \preceq A'$ and A' is a type in Γ

Proposition 2.7
Every term M typeable in a context Γ has a principal type, which is the minimum type of M with respect to $\preceq$.

For use in later sections, we now prove a statement analogous to rule ($\preceq$), showing that a type can be replaced in the context by a type which is smaller (contravariance !) with respect to the order $\preceq$.

Lemma 2.8
Let $A \preceq A'$ and $\Gamma \vdash A : \mathit{Type}_j$.

- If $\Gamma, x : A', \Gamma'$ valid_c, then $\Gamma, x : A, \Gamma'$ valid_c.
- If $\Gamma, x : A', \Gamma' \vdash M : B$, then $\Gamma, x : A, \Gamma' \vdash M : B$.

Proof: By induction on the derivation of $\Gamma, x : A', \Gamma'$ valid_c and $\Gamma, x : A', \Gamma' \vdash M : B$, respectively. The only interesting case is a derivation ending with the (var) rule, which can be dealt with by applying the rule (var) to the induction hypothesis and then reasoning with the rule ($\preceq$). □

2.1.3. Properties of the base calculus

We will now state some of the main questions of interest in the study of the calculus *ECC* and summarize its most important properties. The concepts used here should be intuitively clear, even if some of them have not been formally introduced yet. A detailed exposition will follow when presenting the calculus with metavariables.

The Church-Rosser theorem ensures that evaluating a program by using different reduction strategies has no effect on the result. This theorem is also of relevance for type checking, to be described below.

Proposition 2.9 (Church-Rosser property of *ECC*)
Reduction has the Church-Rosser property: Whenever two terms M_1, M_2 are convertible, there is a term M such that both M_1 and M_2 reduce to M.

Another essential property of the calculus is termination of reduction for well-typed terms.

Definition 2.10 (Weak and Strong Normalization)

- A term is *weakly normalizing* for a reduction relation if for every term t, there exists a finite sequence of reduction steps starting from t and leading to a normal form of t.

- A term is *strongly normalizing* for a reduction relation if for every term t, every sequence of reduction steps starting from t is finite.

Proposition 2.11 (Strong Normalization)
The calculus *ECC* is strongly normalizing.

Definition 2.12 (Type inference, -checking and -inhabitation)

- Given a context Γ and a term t, *type inference* is the problem of either determining a term T such that $\Gamma \vdash t : T$ or showing that no such term exists.

- Given a context Γ and terms t and T, *type checking* is the problem of determining whether $\Gamma \vdash t : T$ holds.

- Given a context Γ and a type T such that $\Gamma \vdash T : \mathit{Type}_i$, *type inhabitation* is the problem of either determining a term t such that $\Gamma \vdash t : T$ or showing that no such term exists.

Type inference and type checking are decidable (see [Luo90], Section 6.2). The question of type inhabitation will be dealt with at length in Chapter 3.

Proposition 2.13

- There exists a type inference algorithm $\mathcal{TI}(\Gamma, t)$ which determines the principal type of t under Γ.

- There exists a type checking algorithm $\mathcal{TC}(\Gamma, t, T)$.

For determining a $?T$ with $\Gamma \vdash t :? T$, the type inference algorithm $\mathcal{TI}$ essentially applies the typing rules backwards, selecting an appropriate rule depending on the outermost term constructor of t. For determining whether $\Gamma \vdash t : T$ holds, the type checking algorithm $\mathcal{TC}$ first computes the principal type T_p by

means of the type inference algorithm and then compares T_p (if existent) and T with respect to $\preceq$.

When literally following the type inference algorithm, the validity of contexts has to be checked repeatedly, as the result of applying the rules (UProp), (UType) and (var). It is easy to construct examples in which type inference then has exponential complexity. Therefore, it has to be asked whether recomputation of essentially the same type information cannot be avoided.

An incremental typing algorithm assumes that it works with valid contexts, i.e. contexts which are known to be correct by construction. An incremental algorithm essentially prunes the derivation tree at subderivations of the form Γ *valid*$_c$.

Definition 2.14 (Incremental type inference / type checking)

- An *incremental type inference* algorithm for context Γ and term t is an algorithm that does not check the validity of contexts.
- An *incremental type checking* algorithm for Γ, t and T is a type checking algorithm based on an incremental type inference algorithm for Γ and t.

The cost of incremental type inference $\mathcal{TI}(\Gamma, t)$, measured as the number of backwards applications of typing rules, is linear in the size of the term t.

Usually, type inference for a term t is only invoked in a context Γ which is *a priori* known to be type correct. With a slight modification, a non-incremental type inference algorithm can be transformed into an equivalent incremental one:

Proposition 2.15
Given a non-incremental type checking algorithm $\mathcal{TI}$, define $\mathcal{TI}_i$ as follows:

- Add the judgement $\Gamma \vdash A : \mathit{Type}_j$ as precondition of rules (Π-Form$_1$) and (λ).
- Discard the preconditions of rules (UProp), (UType) and (var).
- Include all other rules without modification.

$\mathcal{TI}_i$ is an incremental type inference algorithm which, for every context Γ with Γ *valid*$_c$ and for every term t, is equivalent to $\mathcal{TI}$.

Proof: Verify by an induction on the structure of the term t that only valid contexts occur in the derivation of the type of t. Addition of the side conditions $\Gamma \vdash A : \mathit{Type}_j$ in the rules (Π-Form$_1$) and (λ) does not diminish the number of derivable judgements, as these conditions would have to be verified in any case in a subderivation (cf. Lemma 3.2.3 of [Luo90]). □

In the following, we will take a phrase such as " ... can be checked with an incremental algorithm" to mean that there exists an incremental type checking algorithm, which is equivalent to the non-incremental version under discussion and which can be obtained from it by an analogous transformation as above.

For the base calculus presented in this section, the above remarks concerning incremental type checking are rather evident. They are thought to provide the terminology for a discussion of similar questions that arise when trying to ascertain that a solution for a proof obligation is indeed typecorrect (see Section 2.6). The validity of a context can be compromised by a possibly incorrect solution, and an efficient verification of the correctness of a solution relies on the validity of contexts. Thus, it is far from obvious that this vicious circle can be avoided and incremental type checking is always applicable.

2.1.4. Base calculus – Encodings

In *ECC*, the standard logical connectives and equality can be encoded by use of Π-abstraction only.

The encoding of the logical connectives from universal quantification and implication in second-order logic dates back to Prawitz [Pra65]. The formulation using the type constructors of *ECC* is shown in Figure 2.3. In particular, logical implication coincides with non-dependent function type, and universal quantification with Π-abstraction. Accordingly, we will use the symbols Π and ∀ interchangeably, the latter mostly in conjunction with propositions.

$$
\begin{array}{rcl}
\mathit{True} & := & \Pi X : \mathit{Prop}.X \to X \\
\mathit{False} & := & \Pi X : \mathit{Prop}.X \\
A \to B & := & A \to B \\
A \wedge B & := & \Pi R : \mathit{Prop}.(A \to B \to R) \to R \\
A \vee B & := & \Pi R : \mathit{Prop}.(A \to R) \to (B \to R) \to R \\
\neg A & := & A \to \mathit{False} \\
\forall\, x : A.P(x) & := & \Pi x : A.P(x) \\
\exists\, x : A.P(x) & := & \Pi R : \mathit{Prop}.(\Pi x : A.(P(x) \to R)) \to R
\end{array}
$$

Figure 2.3.: Coding of logical connectives in *ECC*

It can be shown that these encodings are adequate with respect to the usual definition of logical connectives via introduction and elimination rules

(see [Pra65], p. 67 or [Luo90], p. 118). For example, in standard predicate logic, the introduction and elimination rules for conjunction are:

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ (\wedge\mathrm{El}) \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ (\wedge\mathrm{Er}) \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ (\wedge\mathrm{I})$$

When expanding the definitions of the connectives, these rules are derivable using rules involving only the rules for universal quantification and implication.

2.2. Introducing Metavariables

In Chapter 1, some applications of metavariables have been informally presented. Now that the most important concepts of the base calculus *ECC* have been introduced, it is possible to describe more precisely the requirements that metavariables are supposed to fulfill. In this section, we will identify some difficulties that arise in a naive approach of handling metavariables, and suggest possible solutions. The rigorous treatment following in the remaining sections of this chapter shows that a calculus incorporating metavariables can indeed be defined without sacrificing essential properties of the base calculus.

As mentioned before, a metavariable $?n$ is a placeholder for a term. Since the formal framework considered here is strongly typed, any solution term is expected to have a certain type T in a context Γ, so it makes sense to constrain $?n$ itself accordingly. We write $\Gamma \vdash ?n : T$ to indicate that any solution of $?n$ has to be of type T in context Γ. Conversely, any term having this property is acceptable as solution for $?n$; we do not impose additional requirements. – As long as no solution for $?n$ has been determined, we want to manipulate $?n$ just like any other term, so $\Gamma \vdash ?n : T$ can alternatively be interpreted as a type assignment for metavariable $?n$. The resulting type system is examined in Section 2.5.

The process of assigning a solution term s to a metavariable $?n$ within a term t containing $?n$ is called *instantiation*. In order to make the formal definition of Section 2.6.1 more comprehensible, we will point out one peculiarity of treating bound variables, which is motivated by the following example.

Example 2.16
Assume we want to prove $\exists f : T \to T.(f\ t) = t$, where T is a type and t a term of type T. Introducing a metavariable for the existential quantifier, we are left with the goal $(?f\ t) = t$, where $\Gamma \vdash ?f : T \to T$ (Γ being the context under consideration). When instantiating the goal with $?f := \lambda\, x : T.\ x$, we obtain $((\lambda\, x : T.\ x)\ t) = t$, which is easy to prove. By the above criterion, this instantiation is acceptable, because $\Gamma \vdash (\lambda\, x : T.\ x) : T \to T$.

The solution provided for $?f$ is a function such that given a parameter $x : T$, x is returned. Instead of producing this function in one step, it should be possible to synthesize it: When applying the rule (λ) backwards to $\Gamma \vdash ?f : T \to T$, we still have to construct $?b$ with $\Gamma, x : T \vdash ?b : T$, where the partial solution for $?f$ is $\lambda\, x : T.\ ?b$. Following the intuition that the function body $?b$ should just return the parameter x, we instantiate $?b := x$. In order to obtain the solution $\lambda\, x : T.\ x$ from the partial solution $\lambda\, x : T.\ ?b$, we have to map the instantiation $?b := x$ over $\lambda\, x : T.\ ?b$ *without renaming bound variables.*

This definition of instantiation has been chosen because it models best the fact that the notion of "local" and "global" variable is relative: Given the defining context $\Gamma, x : T$ of metavariable $?b$, x is global, whereas it is locally bound in the expression $\lambda\, x : T.\ ?b$. Nevertheless, x should in both cases be regarded as the same object.

There are mainly two problems when dealing with context-dependent metavariables, illustrated by the following examples:

Example 2.17
Commutativity of instantiation and reduction: Assume that metavariable $?n_1$ is defined to be of type T in a context containing $x : T$, that is, $T : \mathit{Type}, x : T \vdash ?n_1 : T$. In a naive approach, first reducing the term $\mathit{trm}_1 := (\lambda\, x : T.\ ?n_1)\ t$ to $?n_1$ and then instantiating the result with term x yields the result x. First instantiating $?n_1$ to x and then reducing yields t (as remarked above, the variable x bound by λ-abstraction is the same object as the variable x bound in the context of the metavariable).

$$\begin{array}{ccc} (\lambda\, x : T.\ ?n_1)\ t & \xrightarrow{\{?n_1 := x\}} & (\lambda\, x : T.\ x)\ t \\ \beta \downarrow & & \beta \downarrow \\ ?n_1 & \xrightarrow{\{?n_1 := x\}} & x \qquad\qquad t \end{array}$$

Note that this problem is not caused by a particular type system, but arises in any calculus in which there is a notion of β-reduction and in which metavariables depend on a context.

This problem will be solved by explicitly recording the substitutions that have been performed on a metavariable. This leads to a concept of *explicit substitutions*. In the above example, the term $(\lambda\, x : T.\ ?n_1)\ t$ is then not simply reduced to $?n_1$, but to $?n_1^\frown[x := t]$, where the delayed assignment $x := t$ is carried out as soon as $?n_1$ is instantiated. After having formally defined a

calculus with explicit substitution, we will again look at this particular problem (see Example 2.68).

Example 2.18
Keeping track of type information: Consider a metavariable $?n_2$ defined with the following context and type:

$$A : \mathit{Type}, T : \mathit{Type}, x : T \vdash ?n_2 : T$$

Consider the term $\mathit{trm}_2 := (\lambda\, T : \mathit{Type}.\ \lambda\, x : T.\ ?n_2)\ A$ in context $A : \mathit{Type}$. When first instantiating $?n_2$ with x and then reducing, the resulting $\lambda\, x : A.\ x$ is easily seen to have type $A \to A$. When first reducing trm_2, however, the question arises what the type of the resulting term $\lambda\, x : A.\ ?n_2$ should be. $A \to T$ is certainly not correct, as T does not even occur in context $A : \mathit{Type}$. Claiming that $?n_2$ has type A is also problematic, since then, the term $?n_2$ would have different types (A resp. T) in different contexts. As opposed to the first problem, this difficulty is directly related to the type system and arises in a similar form in any calculus with dependent types.

Again, explicit substitutions provide a solution. When reducing the term $(\lambda\, T : \mathit{Type}.\ \lambda\, x : T.\ ?n_2)\ A$, we do not obtain $\lambda\, x : A.\ ?n_2$, as suggested above, but $\lambda\, x : A.\ ?n_2\widehat{}[T := A]$. According to the typing rules developed in Section 2.5.1, the term $?n_2\widehat{}[T := A]$ does not have the same type as $?n_2$. More generally, it will be shown that terms keep their type when being reduced ("subject reduction", Proposition 2.47) and types remain consistent under instantiation (Proposition 2.72).

2.3. Term calculus with Metavariables

2.3.1. Metavariables

In this section, we formally introduce metavariables and describe properties of the term calculus, without making assumptions about the well-typedness of terms (cf. Section 2.5). In general, separating these issues makes the calculus easier to analyze. However, in the specific case of a calculus with explicit substitutions, we have to impose some requirements which anticipate some properties of well-typed terms.

Definition 2.19 (Metavariables)
$\mathcal{M}$ is an infinite set of objects called *metavariables* disjoint from the set $\mathcal{V}$ of variables. As a notational convention, we always write metavariables as an identifier prefixed by a question mark.

Now, the syntax can be extended to accommodate metavariables and explicit substitutions: Two productions are added to the grammar of core *ECC* (Figure 2.1) to yield the grammar of $\mathcal{L}_{\mathcal{M}}$, the language of *ECC* with metavariables (Figure 2.4).

$$
\begin{array}{rcl}
\mathcal{T} & ::= & \mathcal{V} \\
& | & \mathit{Prop} \mid \mathit{Type}_i \\
& | & \Pi\mathcal{V} : \mathcal{T}.\mathcal{T} \mid \lambda\,\mathcal{V} : \mathcal{T}.\mathcal{T} \mid (\mathcal{T}\ \mathcal{T}) \\
& | & \Sigma\mathcal{V} : \mathcal{T}.\mathcal{T} \mid \mathit{pair}_{\mathcal{T}}(\mathcal{T},\ \mathcal{T}) \mid \pi_1(\mathcal{T}) \mid \pi_2(\mathcal{T}) \\
& | & \mathcal{M}^\frown\mathcal{S} \\
\\
\mathcal{S} & ::= & [] \mid [\mathcal{V} := \mathcal{T}] :: \mathcal{S}
\end{array}
$$

Figure 2.4.: Grammar defining the language of *ECC* with metavariables

A new production for terms permits to build terms of the form $?n^\frown\sigma$ that apply substitutions σ to metavariables $?n$. Substitutions are generated by production $\mathcal{S}$. By this construction, a substitution effectively becomes a part of a term and can be reasoned about in the calculus. This notion of *explicit substitutions* has to be distinguished from the traditional notion of substitutions, which are defined as meta-operations on terms (such as in the following Section 2.3.2).

In order to model the fact that a metavariable depends on a context and that only variables declared in this context can be substituted in the metavariable, all term operations are parameterized by a function $\mathit{svars} : \mathcal{M} \to \mathit{List}\ \mathcal{V}$ which associates to each $?n \in \mathcal{M}$ a list of variables that may be substituted in $?n$. The function *svars* will later be defined to coincide with the function that yields the domain of the context in which the metavariable is declared. Alternatively, it would have been possible to define proof problems (Section 2.4) first and parameterize term operations over proof problems. For reasons of clarity, we have chosen the present order.

More formally, we define explicit (or *internal*) substitutions as follows:

Definition 2.20 (Explicit Substitution)
An explicit substitution is a list of the form $[x_1 := t_1, \ldots x_k := t_k]$ associating terms to variables. A substitution $\sigma = [x_1 := t_1, \ldots x_k := t_k]$ is *valid* for a metavariable $?n$ if:

1. All the variables x_i are distinct.
2. All the variables x_i are contained in $\mathit{svars}(?n)$.

3. If x_i occurs before x_j in σ, then x_i occurs before x_j in $svars(?n)$.

The *domain* of a substitution is defined to be the set $dom(\sigma) := \{x_1, \ldots x_k\}$.

Henceforth, we will assume that all the internal substitutions are valid for the metavariables to which they are attached, and we will enforce this invariant for all operations on terms.

It should be stressed that, in contrast to the view of substitutions as possibly infinite mappings from variables to terms, explicit substitutions are here taken to be finite lists. This is necessary for obtaining a finite representation of terms. The notion of *subterm* can then be extended naturally to metavariables with substitutions: the terms t_i and their subterms are subterms of $?n^\frown[x_1 := t_1, \ldots x_k := t_k]$.

Variables in a valid substitution applied to metavariable $?n$ are always kept in a unique order determined by the variable order of $svars(?n)$ (requirement 3. in Definition 2.20). This requirement is necessary for syntactically comparing terms which are essentially equal. This is further illustrated by Example 2.25 below, the property is used for the proof of the Church-Rosser theorem in Section 2.3.3, in particular Lemma 2.34.

Only variables declared in $svars(?n)$ may occur in the domain of valid substitutions for $?n$ (requirement 2. in Definition 2.20). This postulate can best be motivated when considering typing rules (Section 2.5) and solutions for metavariables (Section 2.6). The general idea is that a term containing a free variable, say x, cannot be the solution of a metavariable $?n$ whose context does not contain a declaration of x. Therefore, any substitution having x as its domain cannot become effective in a solution of $?n$. Apart from being merely plausible, this requirement also facilitates the proof of strong normalization, see Section 2.7.2.

In a strict sense, "pure" metavariables without attached substitutions are not valid terms. Actually, we do not want to distinguish between a metavariable $?n$ and the same metavariable with an empty substitution, $?n^\frown[]$, and so we establish this equality as a notational convention, rather than formalizing it in the calculus. Also note that substitutions can only be attached to metavariables and not to arbitrary terms – for example, $(f\ a)\sigma$ is not a valid term. This distinguishes our calculus from (to our best knowledge) all calculi of explicit substitutions presented in the literature.

In a language with metavariables, one has to distinguish between an actual occurrence of a variable in a term and a potential occurrence, that is, an occurrence after an appropriate instantiation has been carried out. For some of our considerations, a straightforward extension of the standard notion of "free variable" as given by Definition 2.1 is not always sufficient (see for example

the remark following Lemma 2.23), as it would only take actual, but not potential occurrences into account. We therefore extend the definition as follows, however keeping the notation FV:

Definition 2.21 (Free Variables - Terms with Metavariables)
For terms M containing metavariables, the inductive Definition 2.1 of function FV is extended by the clause:

$$FV(?n^\frown[x_1 := t_1, \ldots x_k := t_k]) = (svars(?n) \backslash \{x_1, \ldots x_k\}) \cup \bigcup_{i=1,\ldots,k} FV(t_i)$$

To illustrate this definition, consider a metavariable $?n$ with $svars(?n) \mathrel{\widehat{=}} [x, y, z]$ and a term $t \mathrel{\widehat{=}} (f\ ?n^\frown[x := y])$. Even though x has a syntactic appearance in t, any correct instantiation for $?n$ (in the sense of Section 2.6) such as $?n := x$ will make x disappear, leading to a term $(f\ y)$. Even though z does not occur syntactically in t, it may be introduced by a correct instantiation such as $?n := z$, leading to a term $(f\ z)$.

2.3.2. Reduction Relations

This section is concerned with extending the usual notions of reduction to the language with metavariables. Since substitutions in our calculus are only internalized as attachment to metavariables to express that a substitution is delayed, we cannot completely dispense with an external notion of substitution.

Definition 2.22 (External Substitutions)
Let s be a term, x a variable with $x \notin FV(s)$. A substitution $\sigma := \{x := s\}$ is a mapping from terms to terms, defined by the mapping $\overset{s}{\rightarrow}$ as follows:

Variable

$$y\sigma \overset{s}{\rightarrow} \begin{cases} s & \text{if } x = y \\ y & \text{otherwise} \end{cases}$$

Constant $(Prop)\sigma \overset{s}{\rightarrow} Prop$, $(Type_i)\sigma \overset{s}{\rightarrow} Type_i$

Quantifier For $\mathcal{Q} \in \{\lambda, \Pi, \Sigma\}$: $(\mathcal{Q}z : T.M)\sigma \overset{s}{\rightarrow} \mathcal{Q}z : T\sigma.M\sigma$
provided that $x \neq z$

Application $(f\ a)\sigma \overset{s}{\rightarrow} (f\sigma\ a\sigma)$

Pair $(pair_T(t_1,\ t_2))\sigma \overset{s}{\rightarrow} pair_{T\sigma}(t_1\sigma,\ t_2\sigma)$

Projection $\pi_i(t)\sigma \overset{s}{\rightarrow} \pi_i(t\sigma)$ for $i = 1, 2$.

Metavariable Provided that $x \notin \{y_1 \dots y_k\}$:

- If $x \in svars(?n)$ and x occurs between y_i and y_{i+1} in $svars(?n)$:

$$(?n^\frown[y_1 := t_1, \dots y_k := t_k])\sigma \xrightarrow{s}$$

$$?n^\frown[y_1 := t_1\sigma, \dots y_i := t_i\sigma, x := s, y_{i+1} := t_{i+1}\sigma, \dots y_k := t_k\sigma]$$

- If $x \notin svars(?n)$:

$$(?n^\frown[y_1 := t_1, \dots y_k := t_k])\sigma \xrightarrow{s}$$

$$?n^\frown[y_1 := t_1\sigma, \dots y_k := t_k\sigma]$$

The definition can be extended to contexts as follows:

- $\langle\rangle\{x := s\} \xrightarrow{s} \langle\rangle$
- $(y : T, \Gamma)\{x := s\} \xrightarrow{s} (y : T\{x := s\},\ \Gamma\{x := s\})$, provided $x \neq y$.

In the sequel, we will sometimes use a *parallel* substitution of several variables in a term: For $\sigma := \{x_1 := t_1 \dots x_k := t_k\}$, $t\sigma$ means that the substitution is applied once, but is not to be re-applied to the term resulting from a substitution. In particular, for variables y, $y\sigma \xrightarrow{s} t_i$ (and not $t_i\sigma$) for $y = x_i$. The definition for the other term constructors is analogous to the definition above.

In order to distinguish external substitutions from the internalized explicit substitutions, we write the former in braces like $\{\dots\}$ and the latter in brackets like $[\dots]$. However, the letters σ and τ will be used indiscriminately for both kinds of substitutions.

The mapping $\xrightarrow{s}$ pushes a substitution inside a term, until it can either be applied directly to a variable or has to be recorded for later substitution in a metavariable. Only expressions irreducible with respect to $\xrightarrow{s}$ are correct terms of $\mathcal{L}_{\mathcal{M}}$. An expression like $(\lambda\, x : M.\ N)\sigma$, even though it is not a term, will be taken to stand for the term obtained by moving substitutions inside the term as far as possible, as defined by $\xrightarrow{s}$.

The rules are stated with some provisos concerning the occurrence of variables. When abstracting away from the particular presentation of variables as named objects chosen here (also see Section A.1), the logical essence of these provisos is the following:

- The Quantifier rule postulates that the variables in the domain of a substitution must not occur as bound variables in a term to which the substitution is applied. This is reasonable to assume, since a variable should not be accessible outside its scope.

- The Metavariable rule requires that the variable to be substituted does not yet occur in the list of substitutions of a metavariable. Variables in the substitution list have been used previously for substitution and therefore cannot serve as substitution variables a second time.

Lemma 2.23

- For terms M, N, if $x \notin FV(M)$, then $M\{x := N\} \equiv M$

- For context Γ and term N, if $x \notin FV(\Gamma)$, then $\Gamma\{x := N\} \equiv \Gamma$

This lemma provides a further motivation for the definition of FV given in Definition 2.21, as this lemma would be invalidated by a simple notion of "free variable occurrence": Consider the term $?n$ with $x \in svars(?n)$ and the substitution $?n\{x := N\} \equiv ?n^\frown[x := N] \not\equiv ?n$.

Proof: By induction on the structure of M (the extension to contexts Γ is straightforward).

- M is a variable. $M \equiv x$ is impossible, so $M \equiv y \neq x$, and $y\{x := N\} \equiv y$

- M is $\lambda\, z : A.\ B$. (By our variable convention, $z \neq x$). We have $x \notin FV(A)$ and $x \notin FV(B)$ and, by induction hypothesis, $A\{x := N\} \equiv A$ and $B\{x := N\} \equiv B$, thus

$$(\lambda\, z : A.\ B)\{x := N\} \equiv \lambda\, z : A\{x := N\}.\ B\{x := N\} \equiv \lambda\, z : A.\ B$$

- Similarly for Π- and Σ-abstraction.

- M is $?n^\frown[x_1 := t_1, \ldots x_k := t_k]$. By convention, we assume that $x \notin \{x_1, \ldots, x_k\}$. By $x \notin FV(M)$ and definition of FV, we have $x \notin svars(?n)$ and $x \notin FV(t_j)$ for $j = 1, \ldots, k$.

  By definition of substitution, taking into account that $x \notin svars(?n)$ and that $t_j\{x := N\} \equiv t_j$ by induction hypothesis, we have $M\{x := N\} \equiv ?n^\frown[x_1 := t_1\{x := N\}, \ldots x_k := t_k\{x := N\}] \equiv ?n^\frown[x_1 := t_1, \ldots x_k := t_k] \equiv M$.

- All other cases require a straightforward application of the induction hypothesis.

□

Lemma 2.24
Let Γ be a context and M and N terms.

- If $x \in FV(M)$, then $FV(M\{x := N\}) = FV(M)\backslash\{x\} \cup FV(N)$
- If $x \in FV(\Gamma)$, then $FV(\Gamma\{x := N\}) = FV(\Gamma)\backslash\{x\} \cup FV(N)$

Proof: Induction on the structure of M (resp. Γ). We only consider the following cases, the other ones are similar:

- M is a variable. In this case, $M \equiv x$, so $FV(M\{x := N\}) = FV(N)$.
- M is a λ-abstraction of the form $\lambda\, z : A.\ B$. Since $x \in FV(\lambda\, z : A.\ B)$, either $x \in FV(A), x \notin FV(B)$ or $x \in FV(B), x \notin FV(A)$ or $x \in FV(A), x \in FV(B)$. Let us assume the first, the other cases are similar. By induction hypothesis, $FV(A\{x := N\}) = FV(A)\backslash\{x\} \cup FV(N)$ and by Lemma 2.23, $FV(B\{x := N\}) = FV(B)$, so by the definition of FV, $FV((\lambda\, z : A.\ B)\{x := N\}) = FV(A\{x := N\}) \cup FV(B\{x := N\}) = FV(A)\backslash\{x\} \cup FV(N) \cup FV(B) = FV(M)\backslash\{x\} \cup FV(N)$.
- M is a metavariable of the form $?n^\frown[x_1 := t_1, \ldots x_k := t_k]$.

  Case $x \in svars(?n)$*:*
  $FV((?n^\frown[x_1 := t_1, \ldots x_k := t_k])\{x := N\}) =$
  (by definition of substitution, similarly for all other variable orders:)
  $FV(?n^\frown[x_1 := t_1\{x := N\}, \ldots x_k := t_k\{x := N\}, x := N]) =$
  (by definition of FV:)
  $(svars(?n)\backslash\{x_1, \ldots x_k, x\}) \cup \bigcup_{i=1,\ldots,k} FV(t_i\{x := N\}) \cup FV(N) =$
  (by induction hypothesis, Lemma 2.23 and set theory:)
  $((svars(?n)\backslash\{x_1, \ldots x_k\}) \cup \bigcup_{i=1,\ldots,k} FV(t_i))\backslash\{x\} \cup FV(N) =$
  (by definition of FV:)
  $FV(?n^\frown[x_1 := t_1, \ldots x_k := t_k])\backslash\{x\} \cup FV(N)$

  Case $x \notin svars(?n)$*:* Similar. Note that $x := N$ will not be added to the substitution. Since $x \notin svars(?n)$, we have $(svars(?n)\backslash\{x_1, \ldots x_k\}) = (svars(?n)\backslash\{x_1, \ldots x_k\})\backslash\{x\}$.

□

The following example illustrates why the variables in a substitution have to be ordered:

Example 2.25
Consider a variant of the above Metavariable reduction rule in which the substitution list is not ordered, but the new substitution variable is simply added at the front of the list:

$$(?n^\frown[y_1 := t_1, \ldots y_k := t_k])\sigma \xrightarrow{s} ?n^\frown[x := s, y_1 := t_1\sigma, \ldots y_k := t_k\sigma]$$

Then, the term

$$(\lambda\, x_1 : T_1.\ (\lambda\, x_2 : T_2.\ ?n)\ a_2)\ a_1$$

can be reduced by first contracting the innermost redex, yielding the term $?n^\frown[x_1 := a_1, x_2 := a_2]$, or by first contracting the outermost redex, yielding $?n^\frown[x_2 := a_2, x_1 := a_1]$. Clearly, both terms are not equal. With the original rule, the variables being ordered as x_1 before x_2, both reductions lead to the term $?n^\frown[x_1 := a_1, x_2 := a_2]$.

Example 2.26
It may be wondered whether the substitution in metavariables could not be "optimized" in the following sense: Replace case $x \notin \mathit{svars}(?n)$ of the definition by:

$$(?n^\frown[y_1 := t_1, \ldots y_k := t_k])\{x := t\} \xrightarrow{s} ?n^\frown[y_1 := t_1, \ldots y_k := t_k]$$

that is, do not propagate $\{x := t\}$ in the substitution, because at first glance, it may seem that x cannot occur in any t_i if it is not in the context of $?n$. Consider, however, the following example: $a : T, y : T \vdash ?n : T$ and compare the reductions of the term $(\lambda\, x : T.\ ((\lambda\, y : T.\ ?n)\ x))\ a$. When first reducing the inner, then the outer redex with the "optimized" rewriting relation, we obtain the term $?n^\frown[y := x]$, otherwise (i.e. first outer, then inner redex) we obtain $?n^\frown[y := a]$ and thus have lost confluence.

In the Section 2.3.3, the question of uniqueness of normal forms will be treated on a more formal basis. First, however, some standard definitions will be given, which have mainly been taken from [Bar84].

The base reductions we are interested in are β-reduction, which computes the application of a λ-abstraction to an argument, and π-reduction, which computes the projection from a pair:

Definition 2.27 (Base Reduction Relations)
- β-reduction: $(\lambda\, x : T.\ M)\ N \ \rightarrow_\beta\ M\{x := N\}$
- π-reduction: $\pi_i(\mathit{pair}_T(M_1,\ M_2)) \ \rightarrow_\pi\ M_i$ for $i = 1, 2$

- Redex and contractum: The term $(\lambda\, x : T.\ M)\ N$ resp. $\pi_i(pair_T(M_1,\ M_2))$ is called a *redex*, the term $M\{x := N\}$ resp. M_i the *contractum*.

For example, $(\lambda\, x : T.\ x)\ N\ \rightarrow_\beta\ N$ and $(\lambda\, x : T.\ ?n^\frown[\ ])\ N\ \rightarrow_\beta\ ?n^\frown[x := N]$, if $x \in svars(?n)$.

From these base relations, more complex relations can be constructed. If a relation R holds between two terms M and N, then the compatible closure R^c essentially extends R to terms in which M and N are embedded.

Definition 2.28 (Derived Reduction Relations)

- The compatible closure R^c of a relation R on terms is the smallest relation such that $(C[M]\ R^c\ C[N])$ holds whenever $(M\ R\ N)$, for all terms M, N and term contexts $C[.]$.

- One-step reduction $\rightarrow_1$ is defined as the compatible closure of $\rightarrow_\beta$ and $\rightarrow_\pi$:

$$\rightarrow_1 := (\ \rightarrow_\beta\ \cup\ \rightarrow_\pi\ )^c$$

- Reduction $\twoheadrightarrow$ is defined as the reflexive-transitive closure of $\rightarrow_1$:

$$\twoheadrightarrow := {\rightarrow_1}^*$$

The *reduct* of a term M is any term N such that $M \twoheadrightarrow N$.

We assume that the notion of "term context" (not to be confused with typing contexts) is sufficiently clear. For details, consult [Bar84].

Definition 2.29 (Convertibility and Normal Form)

- Convertibility: Terms M and N are convertible (written $M \simeq N$) if there exists a sequence of terms $M \equiv M_0 \ldots M_n \equiv N$ such that $M_i \twoheadrightarrow M_{i+1}$ or $M_{i+1} \twoheadrightarrow M_i$.

- Normal form: A term is in *normal form* if it contains no redex.

Occasionally, we will use some of the above notions, which have been defined for terms, in an obvious extension to contexts. For example, a context is in normal form if in all of its declarations $x : T$, the term T is in normal form.

Definition 2.30 (Weak head normal form)

- A term M is in *weak head normal form* (whnf) if it is not of the form $(\lambda\, x : A.M_1)\ M_2$ or $\pi_i(pair_T(M_1,\ M_2))$ (for $i = 1, 2$).

- A whnf of M is a (not necessarily unique) term M' such that $M \twoheadrightarrow M'$ and M' is in whnf.

Thus, a term is in weak head normal form if its outermost term position is not a redex. The outermost term constructor of the whnf of a term is uniquely determined, even though different reduction strategies may produce different whnf's of a term. For proof search, it is often sufficient to compute weak head normal forms, because once the whnf of a term is known, it is clear which inference rule (if any) can be applied to it.

2.3.3. Properties of the Term Calculus

In this section, we will show that the reduction relation $\twoheadrightarrow$ has the Church-Rosser property. This property ensures that two diverging computation paths can always be joined again. If reductions of the strict part of $\twoheadrightarrow$ always terminate (which indeed they do, see Section 2.7.2), then normal forms are unique. By this means, convertibility of two terms can be decided, by reducing them to normal form and comparing the normal forms syntactically.

Definition 2.31 (Diamond and Church-Rosser property)

- A relation R on terms satisfies the Diamond property, if for all terms M, M_1, M_2 such that $(M\ R\ M_1)$ and $(M\ R\ M_2)$ hold, there exists a term M_3 such that $(M_1\ R\ M_3)$ and $(M_2\ R\ M_3)$ hold.

- A relation R on terms has the Church-Rosser property, if the compatible, reflexive-transitive closure of R satisfies the Diamond property.

Proposition 2.32 (Reduction is Church-Rosser)
The reduction relation $\twoheadrightarrow$ satisfies the Church-Rosser property.

Proof: The proof is an extension of the proof of the Church-Rosser property of the untyped λ-calculus that can be found for example in [Bar84], Chapter 3. The proof method has originally been developed by Martin-Löf and Tait and proceeds along the following steps:

- Define a parallel one-step reduction relation $\Rightarrow_1$ and show that it is preserved under substitutions (see Lemma 2.35).

- Show that $\Rightarrow_1$ has the Diamond property (see Lemma 2.36).

- The inclusions $\rightarrow_1 \subseteq \Rightarrow_1$ and $\Rightarrow_1 \subseteq \twoheadrightarrow$ hold among the reduction relations. With $\twoheadrightarrow$ defined as the reflexive-transitive closure of $\rightarrow_1$, it is easy to show that the reflexive-transitive closure of $\Rightarrow_1$ is equal to $\twoheadrightarrow$.

- For an arbitrary relation R, it can be shown that if R has the Diamond property, then also its reflexive-transitive closure R^*. In particular, the

relation $\twoheadrightarrow$, being equal to ${\Rightarrow_1}^*$, has the Diamond property, and since it is its own compatible, reflexive-transitive closure, it is Church-Rosser.

□

Definition 2.33 (Parallel one-step reduction)
Parallel one-step reduction $\Rightarrow_1$ is defined as the simultaneous application of the reduction relation $\rightarrow_1$ at several subterms of a term:

- $M \Rightarrow_1 M$
- If $T \Rightarrow_1 T'$ and $M \Rightarrow_1 M'$, then $(\mathcal{Q}x : T.M) \Rightarrow_1 (\mathcal{Q}x : T'.M')$ for $\mathcal{Q} \in \{\lambda, \Pi, \Sigma\}$
- If $M_1 \Rightarrow_1 M_1'$ and $M_2 \Rightarrow_1 M_2'$, then $(M_1\ M_2) \Rightarrow_1 (M_1'\ M_2')$. Similarly for pairs and projections.
- If $P \Rightarrow_1 P'$ and $N \Rightarrow_1 N'$, then $(\lambda\, x : T.\ P)\ N \Rightarrow_1 P'\{x := N'\}$.
- If $T \Rightarrow_1 T'$ and $M_i \Rightarrow_1 M_i'$ $(i = 1, 2)$, then $\pi_i(pair_T(M_1,\ M_2)) \Rightarrow_1 M_i'$.
- If $M_i \Rightarrow_1 M_i'$ $(i = 1 \ldots n)$, then
  $?n^\frown[x_1 := M_1 \ldots x_k := M_k] \Rightarrow_1 ?n^\frown[x_1 := M_1' \ldots x_k := M_k']$

Parallel one-step reduction allows to reduce several redexes in different subterms at once, as in:

$$(f\ ((\lambda\, x : T.\ x)\ a)\ ((\lambda\, y : T.\ y)\ b)) \Rightarrow_1 (f\ a\ b)$$

which is not possible in simple one-step reduction. Thus,

$$(f\ ((\lambda\, x : T.\ x)\ a)\ ((\lambda\, y : T.\ y)\ b)) \rightarrow_1 (f\ a\ b)$$

does not hold.

The Substitution Lemma shows that the application of substitutions can be permuted in a certain sense:

Lemma 2.34 (Substitution Lemma)
Consider the substitutions $\{x := N\}$ and $\{y := L\}$, assume that $x \neq y$ and $x \notin FV(L)$. Then $M\{x := N\}\{y := L\} = M\{y := L\}\{x := N\{y := L\}\}$

Proof: The proof is by induction on the structure of M and runs essentially along the same lines as in the untyped λ-calculus. Only the variable and metavariable cases are non-trivial. For all other term constructors, draw substitutions inside the term and apply induction hypothesis.

Variable case: Either $M = x$ or $M = y$ or $M = z$ with $z \neq x, y$:

- $M = x$: Then $x\{x := N\}\{y := L\} = N\{y := L\} = x\{y := L\}\{x := N\{y := L\}\}$
- $M = y$: Then $y\{x := N\}\{y := L\} = L = L\{x := N\{y := L\}\}$ (by Lemma 2.23, since $x \notin FV(L)$).
- $M = z$: Then $z\{x := N\}\{y := L\} = z = z\{y := L\}\{x := N\{y := L\}\}$.

Metavariable case: Assume that M is of the form $?n^\frown[z_1 := t_1, \ldots z_k := t_k]$. Then
$?n^\frown[z_1 := t_1, \ldots z_k := t_k]\{x := N\}\{y := L\} \equiv$
(by definition of substitution)
$?n^\frown[x := N\{y := L\}, y := L, z_1 := t_1\{x := N\}\{y := L\}, \ldots] \equiv$
(induction hypothesis applied to the t_i)
$?n^\frown[x := N\{y := L\}, y := L, z_1 := t_1\{y := L\}\{x := N\{y := L\}\}, \ldots] \equiv$
(by definition of substitution)
$?n^\frown[z_1 := t_1, \ldots z_k := t_k]\{y := L\}\{x := N\{y := L\}\}$

This line of reasoning supposes that $x, y \in svars(?n)$. If x, y or both do not occur in $svars(?n)$, a similar proof is possible, with x resp. y resp. both missing in the substitution for $?n$.

Note that the correctness of this argument critically depends on an appropriate ordering of the variables in the substitution. Here, we have assumed, without loss of generality, that $x < y < z_1 < \ldots < z_k$ if $<$ is the order of the variables in $svars(?n)$. □

Lemma 2.35
If $M \Rightarrow_1 M'$ and $N \Rightarrow_1 N'$, then $M\{x := N\} \Rightarrow_1 M'\{x := N'\}$

Proof: By induction on the definition of relation $\Rightarrow_1$. For convenience, the substitution $\{x := N\}$ will be called σ, the substitution $\{x := N'\}$ will be called σ'. We will only consider selected cases, other cases being similar:

- Assume $M \Rightarrow_1 M'$ is $M \Rightarrow_1 M$. Then one has to show that $M\sigma \Rightarrow_1 M\sigma'$. This is done by induction on the structure of M:
  - $M \equiv x$. Then $M\sigma \equiv N \Rightarrow_1 N' \equiv M\sigma'$
  - $M \equiv y$ for an $y \neq x$. Then $M\sigma \equiv y \Rightarrow_1 y \equiv M\sigma'$
  - *Prop*, $Type_i$: Trivial
  - Abstraction, application, pairing, projections: apply induction hypothesis

  - Metavariables: also apply induction hypothesis, as in the following (assuming $x \in svars(?n)$):
    $M\sigma \equiv (?v^\frown[y_1 := t_1 \ldots y_k := t_k])\sigma \equiv$
    $?v^\frown[x := N, y_1 := t_1\sigma \ldots y_k := t_k\sigma] \Rightarrow_1$
    (by induction hypothesis and the fact that $N \Rightarrow_1 N'$):
    $?v^\frown[x := N', y_1 := t_1\sigma' \ldots y_k := t_k\sigma'] \equiv$
    $(?v^\frown[y_1 := t_1 \ldots y_k := t_k])\sigma' \equiv M\sigma'$
    If $x \notin svars(?n)$, reason analogously with $x := N$ omitted from the substitution.

- Assume $M \Rightarrow_1 M'$ is $(\lambda\, y : T.\ B)\ A \Rightarrow_1 B'\{y := A'\}$, with $A \Rightarrow_1 A'$ and $B \Rightarrow_1 B'$. Then
  $M\sigma \equiv ((\lambda\, y : T.\ B)\ A)\sigma \equiv ((\lambda\, y : T\sigma.\ B\sigma)\ A\sigma) \Rightarrow_1$
  (by induction hypothesis)
  $(B'\sigma')\{y := A'\sigma'\}$
  (since y does not occur in N', by the substitution lemma 2.34)
  $\equiv (B'\{y := A'\})\{x := N'\} \equiv M'\sigma'$

- Reduction of applications other than the above case, of pairs, projections, quantification, metavariables: Simply apply induction hypothesis.

□

Lemma 2.36
$\Rightarrow_1$ satisfies the diamond property.

Proof: It has to be shown for all M, M_1, M_2, that whenever $M \Rightarrow_1 M_1$ and $M \Rightarrow_1 M_2$, there exists a term M_3 such that $M_1 \Rightarrow_1 M_3$ and $M_2 \Rightarrow_1 M_3$. This is done by induction on the derivation $M \Rightarrow_1 M_1$. The only interesting cases arise from reductions that can be applied at overlapping redexes; these situations already occur in the untyped λ-calculus and can be handled by standard methods. For all other reductions, a straightforward application of the induction hypothesis is possible. □

This concludes the proof of the Church-Rosser property of the term calculus.

2.4. Proof Problems

As opposed to the situation encountered in less complex calculi (such as the simply-typed λ-calculus), there can be intricate dependencies among metavariables in calculi with dependent types. In particular, the type of one metavariable can depend on the value assigned to another one, and the well-typedness of a context can depend on the value assigned to a metavariable.

Before stating typing rules and examining their properties, some restrictions on dependencies among metavariables have to be imposed which are strong enough to make verification of the correctness of solutions for metavariables possible. The restrictions should be sufficiently liberal so that dependencies among metavariables can be exploited for proof search, as in the following example.

Example 2.37
Consider the $(\exists\, R)$ rule (cf. Section 4.4.1):

$$\frac{\Gamma \vdash ?n_1 : T \qquad \Gamma \vdash ?n_2 : P(?n_1)}{\Gamma \vdash ?n_0 : \exists\, x : T.P(x)}\ (\exists\, R)$$

Application of this rule introduces two metavariables $?n_1$ and $?n_2$, where $?n_2$ depends on $?n_1$ since $?n_1$ occurs in $P(?n_1)$. If there is a declaration of the form $h : P(t)$ in Γ, then $?n_2$ can be solved with h, leading to an assignment $\{?n_1 := t\}$ as a side-effect.

The next example illustrates that a seemingly innocent formula can give rise to proof situations that are awkward, if not impossible to handle:

Example 2.38
Starting from the (academic) formula $\exists\, T : \mathit{Type}, Q : T \to \mathit{Prop}, x : T.(T \to Q(x))$, eliminating existential quantifiers (as in the example above) and introducing assumptions, one obtains the following set of metavariables:
$\vdash ?T : \mathit{Type}$
$\vdash ?Q : ?T \to \mathit{Prop}$
$\vdash ?x : ?T$
$h : ?T \vdash ?n : ?Q(?x)$.

One step that suggests itself now is to equate $?n$ with h and consequently $?T$ with $?Q(?x)$, leading to a new proof problem with $\vdash ?Q : ?Q(?x) \to \mathit{Prop}$ and $\vdash ?x : ?Q(?x)$, in which the dependency of metavariables is cyclic. There is no intuitive interpretation of such a proof problem, nor is it clear how to type-check terms containing $?Q$ and $?x$ and how to effectively verify tentative solutions of such a proof problem.

In the sequel, we will permit proof problems with metavariable dependencies of the first kind, but will exclude circularities of the second kind. As demonstrated by the above examples, a context or a type can again contain metavariables. The set *MVars* of metavariables of a term (a context, a substitution) is given by the following definition:

Definition 2.39
For a term t, context Δ respectively substitution σ, let $MVars(t)$, $MVars(\Delta)$ and $MVars(\sigma)$ be the set of metavariables occurring in t, Δ respectively σ.

A metavariable depends on a context Γ and has a type T, as expressed by the more suggestive notation $\Gamma \vdash ?n : T$. Since, in the course of a proof, metavariables occurring in Γ or T can be instantiated, a context and a type are not invariantly assigned to a metavariable $?n$ by functions depending only on $?n$, but they are also determined by other metavariables occurring in Γ and T. Likewise, the set of metavariables on which $?n$ depends changes as metavariables are instantiated by terms possibly containing new metavariables.

These considerations lead to the notion of *proof problem* which will capture more formally the idea of a set of metavariables and the defining contexts and types of the metavariables contained in this set.

Definition 2.40 ((Valid) Proof Problem)
A proof problem $\mathcal{P}$ is a triple $(M_\mathcal{P}, ctxt_\mathcal{P}, type_\mathcal{P})$ consisting of:

- A finite set of metavariables $M_\mathcal{P}$
- A function $ctxt_\mathcal{P}$ assigning a context to each $?n \in M_\mathcal{P}$, such that $dom(ctxt_\mathcal{P}(?n)) = svars(?n)$.
- A function $type_\mathcal{P}$ assigning a term to each $?n \in M_\mathcal{P}$

For a proof problem $\mathcal{P}$ and $?n_1, ?n_2 \in M_\mathcal{P}$, the relation $<_\mathcal{P}$ is defined as:
$?n_1 <_\mathcal{P} ?n_2$ iff $?n_1 \in MVars(ctxt_\mathcal{P}(?n_2))$ or $?n_1 \in MVars(type_\mathcal{P}(?n_2))$.
Let $\ll_\mathcal{P}$ be defined as the transitive closure of $<_\mathcal{P}$.
A proof problem $\mathcal{P}$ is called *valid* if $\ll_\mathcal{P}$ is an irreflexive partial order.

Remarks:

1. The condition $dom(ctxt_\mathcal{P}(?n)) = svars(?n)$ establishes a link between aspects of the syntax of metavariables (the list of variables that may appear in substitutions, $svars(?n)$) and aspects of the semantics (the typing constraints of metavariables, given by the functions $ctxt_\mathcal{P}$ and $type_\mathcal{P}$, see Section 2.5).

2. The above definition has to be made more precise in the following sense: The relation $<_\mathcal{P}$ as defined above may change when terms are reduced. Assume, for example, that $type_\mathcal{P}(?n_2) = (\lambda\, x :?n_1.\ x)\ A$. Here, $?n_1 <_\mathcal{P} ?n_2$, even though there is no genuine dependency between $?n_1$ and $?n_2$ – it disappears when reducing $(\lambda\, x :?n_1.\ x)\ A$. Altogether, we require the relation $<_\mathcal{P}$ to be invariant under reduction of terms. Otherwise,

$<_{\mathcal{P}}$ is subject to the contingencies of a particular presentation of terms, which is confusing in situations where terms are routinely reduced, for example in type checking. Thus, we should like to say that we compute the relation $<_{\mathcal{P}}$ as defined above, however with $ctxt_{\mathcal{P}}(?n_2)$ and $type_{\mathcal{P}}(?n_2)$ in normal form. The fact that normal forms exist will not be proved until Section 2.7.2, and so we have to express this requirement more clumsily as: $<_{\mathcal{P}}$ is the least relation such that $?n_1 <_{\mathcal{P}} ?n_2$ iff $?n_1 \in MVars(C)$ or $?n_1 \in MVars(T)$, for all reducts C, T of $ctxt_{\mathcal{P}}(?n_2)$ resp. $type_{\mathcal{P}}(?n_2)$.

Notation:

- The subscripts in $M_{\mathcal{P}}, ctxt_{\mathcal{P}}, type_{\mathcal{P}}$ will be omitted whenever $\mathcal{P}$ is clear from the context.
- Instead of $?n \in M_{\mathcal{P}}$, we often write $?n \in \mathcal{P}$
- Analogously, we define $\mathcal{P}\backslash\{?n_1, \ldots ?n_k\}$ as the proof problem

  $$(M_{\mathcal{P}}\backslash\{?n_1, \ldots ?n_k\}, ctxt', type')$$

  where the functions $ctxt'$ and $type'$ are the restrictions of $ctxt_{\mathcal{P}}$ and $type_{\mathcal{P}}$ to the set $M_{\mathcal{P}}\backslash\{?n_1, \ldots ?n_k\}$. Similarly, we write $\mathcal{P} \cup \{?n_1, \ldots ?n_k\}$ if the contexts and types of $?n_1, \ldots ?n_k$ are understood.

Intuitively speaking, a proof problem is just a set of metavariables, whereas a valid proof problem is one where the dependency relation $\ll_{\mathcal{P}}$ among metavariables is acyclic. For example, the proof problem $\mathcal{P}_1$ with the set of metavariables $\{?n_1, ?n_2\}$ as given in Example 2.37 is valid, with $?n_1 \ll_{\mathcal{P}_1} ?n_2$ since $?n_1 \in MVars(P(?n_1))$. Regarding Example 2.38, the proof problem $\mathcal{P}_2 = (M_{\mathcal{P}_2}, ctxt_{\mathcal{P}_2}, type_{\mathcal{P}_2})$ has $M_{\mathcal{P}_2} = \{?x, ?Q\}$, the contexts $ctxt_{\mathcal{P}_2}(?x)$ and $ctxt_{\mathcal{P}_2}(?Q)$ are empty and $type_{\mathcal{P}_2}(?x) = ?Q(?x)$ and $type_{\mathcal{P}_2}(?Q) = ?Q(?x) \to Prop$. Proof problem $\mathcal{P}_2$ is not valid, since $?x \ll_{\mathcal{P}_2} ?x$.

Since, for every valid proof problem $\mathcal{P}$, the set $M_{\mathcal{P}}$ is finite and the order $\ll_{\mathcal{P}}$ is transitive and irreflexive, it is also well-founded on $M_{\mathcal{P}}$. The order $\ll_{\mathcal{P}}$ can therefore be used in inductive proofs.

As an application, consider the following proposition, which is motivated by the fact that in subsequent chapters (see for example Section 2.7), we will have to define functions f that are essentially homomorphisms over the term structure and over the structure of contexts, with the exception of metavariables, where some additional computations are performed by functions g. It is not immediately evident that these functions f are well-defined, even if the functions g are strict, i.e. yield defined results for defined arguments.

Proposition 2.41
Let $\mathcal{P}$ be a valid proof problem, and assume that all terms and contexts under consideration only contain metavariables from $\mathcal{P}$.

Let f be a function defined on terms as follows:

- $f(x) = x$ for variables x
- $f(\mathit{Prop}) = \mathit{Prop}$, $f(\mathit{Type}_i) = \mathit{Type}_i$
- $f(\mathcal{Q}x : T.M) = \mathcal{Q}x : f(T).f(M)$ for $\mathcal{Q} \in \{\lambda, \Pi, \Sigma\}$
- $f(M\ N) = (f(M)\ f(N))$
- $f(\mathit{pair}_T(t_1,\ t_2)) = \mathit{pair}_{f(T)}(f(t_1),\ f(t_2))$
- $f(\pi_i(t)) = \pi_i(f(t))$ for $i = 1, 2$
- $f(?n^\frown[x_1 := t_1, \ldots, x_k := t_k]) =$
  $g(\hat{f}(\mathit{ctxt}(?n)), f(\mathit{type}(?n)), [f(t_1) \cdots f(t_k)])$,
  where the function $g(\Gamma, T, \mathit{trmlist})$ is defined whenever context Γ, term T and the list of terms *trmlist* are defined.

The extension $\hat{f}$ of f to contexts is defined by:

- $\hat{f}(\langle\rangle) = \langle\rangle$
- $\hat{f}(x : A, \Gamma) = x : f(A),\ \hat{f}(\Gamma)$

Then $f(t)$ is well-defined for every term t and $\hat{f}(\Gamma)$ for every context Γ.

Proof: Termination of the functions f resp. $\hat{f}$ is shown by giving measure functions *ord* resp. $\widehat{ord}$ which decrease for recursive calls of f resp. $\hat{f}$.

Define functions *ord* on terms and $\widehat{ord}$ on contexts as follows:
For terms t, let $\mathit{ord}(t)$ be the pair (m, s) where:

- m is the multiset of metavariables occurring in t
- s is the size of term t

For contexts $\Gamma \mathrel{\widehat{=}} x_1 : A_1, \ldots x_n : A_n$, let $\widehat{ord}(\Gamma) = (m, s)$ where:

- m is the multiset of metavariables occurring in Γ
- s is the sum of the term sizes of the A_i

The multisets m are ordered by the multiset order induced by $\ll_{\mathcal{P}}$ (cf. [Der85]). A term t_1 is smaller than t_2 if $ord(t_1)$ is lexicographically smaller than $ord(t_2)$ – similarly for contexts.

It is easy to check that for recursive applications of f resp. $\hat{f}$, the parameters get strictly smaller with respect to ord resp. $\widehat{ord}$. In particular, $\widehat{ord}(ctxt(?n))$ is smaller than $ord(?n^\frown\sigma)$ because only metavariables smaller than $?n$ (with respect to $\ll_{\mathcal{P}}$) occur in $ctxt(?n)$. For the same reason, $ord(type(?n))$ is smaller than $ord(?n^\frown\sigma)$. □

2.5. Typing

In this section, typing rules for the language with metavariables are presented, and some properties of the resulting calculus ECC_M are proved. Just as the language $\mathcal{L}_{\mathcal{M}}$ is an extension of the language $\mathcal{L}$ of ECC, so new typing rules are added in a "modular" fashion to the calculus with metavariables, i.e. without modifying existing rules. As a consequence, exactly the same derivations are possible in the fragment of ECC_M not involving metavariables as in pure ECC. This is a useful property in view of the fact that a calculus with metavariables is introduced as a convenience for carrying out proofs, but is supposed not to display a behaviour differing from the original calculus after the proofs are finished.

In the following, typing rules will be presented (Section 2.5.1), some properties of typing will be proved (Section 2.5.2), and it will be shown how these rules give rise to a type checking algorithm for the extended language, which is not immediately evident (Section 2.5.3).

Before embarking on a detailed analysis of typing, it should be remarked that typing of terms involving metavariables always has to be understood relative to a proof problem, even a valid proof problem in order to ensure termination of type inference and type checking (see for example the proof of Proposition 2.56, which depends on the validity of the underlying proof problem). When a proof problem is modified by instantiating some of its metavariables, the typing relation changes as well. Assume, for example, that a metavariable-free term t has type $P(?n)$. After instantiating $?n$ to a, term t will have type $P(a)$, which is not convertible to $P(?n)$ (see however Proposition 2.72). It is noteworthy that this situation only arises in dependently-typed calculi. In the simply-typed λ-calculus, instantiation does not affect the type of terms.

2.5.1. Typing: Rules and Definitions

As mentioned above, typing is relative to a particular proof problem $\mathcal{P}$. To emphasize this dependence on $\mathcal{P}$, we will in the following define a derivability relation $\vdash_{\mathcal{P}}$ indexed by valid proof problems $\mathcal{P}$. For proof problems $\mathcal{P}$ which are not valid, $\vdash_{\mathcal{P}}$ is undefined. For most of the investigations, we will fix a valid proof problem $\mathcal{P}$ and consider derivability with respect to this $\mathcal{P}$. We will then drop the index from $\vdash_{\mathcal{P}}$.

Three typing rules for metavariables, displayed in Figure 2.5, are added to the base calculus.

$$\frac{ctxt_{\mathcal{P}}(?n) \vdash_{\mathcal{P}} type_{\mathcal{P}}(?n) : Type_j}{ctxt_{\mathcal{P}}(?n) \vdash_{\mathcal{P}} ?n^\frown[] : type_{\mathcal{P}}(?n)} \text{ (MV-base)}$$

$$\frac{z \notin FV(\Gamma) \cup FV(\Delta) \cup dom(\sigma) \qquad \Gamma \vdash_{\mathcal{P}} T : Type_j \qquad \Gamma, \Delta \vdash_{\mathcal{P}} ?n^\frown\sigma : N}{\Gamma, z : T, \Delta \vdash_{\mathcal{P}} ?n^\frown\sigma : N} \text{ (MV-weak)}$$

$$\frac{\Gamma \vdash_{\mathcal{P}} t : T \qquad \Gamma, x : T, \Delta \vdash_{\mathcal{P}} ?n^\frown\sigma : N}{\Gamma, \Delta\{x := t\} \vdash_{\mathcal{P}} (?n^\frown\sigma)\{x := t\} \ : \ N\{x := t\}} \text{ (MV-}\beta\text{-Red)}$$

Figure 2.5.: Typing rules for Metavariables

These rules can be motivated as follows:

MV-base A metavariable $?n$ with empty substitution is typecorrect (with respect to proof problem $\mathcal{P}$) in case its defining type $type_{\mathcal{P}}(?n)$ and, consequently, its defining context $ctxt_{\mathcal{P}}(?n)$ are well-typed.

MV-weak The weakening rule, which is admissible for the base logic, is explicitly added for metavariables. Note that in the presentation chosen here, an application of this rule does not leave any trace in $?n^\frown\sigma$ and its type N. This should be compared to the presentation with de Bruijn indices in Section A.1.3, where application of this rule leads to a shift of indices.

MV-β-Red This rule simulates the behaviour of β-reduction. To illustrate its effect, we resume Example 2.18 which leads to a type-incorrect term when treated naively.

Assume, then, that the term $((\lambda\, T : Type.\ \lambda\, x : T.\ ?n)\ A)$ has to be reduced to normal form, where $A : Type, T : Type, x : T \vdash ?n : T$. Note

that the type of this term is $A \to A$, since $\lambda\, T : \mathit{Type}.\ \lambda\, x : T.\ ?n$ is of type $\Pi\, T : \mathit{Type}.\ T \to T$. Reduction yields the term $\lambda\, x : A.\ ?n^\frown[T := A]$. The derivation of its type reflects the procedure of β-reduction – with the sole difference that $T : \mathit{Type}$ and $x : T$ are not bound locally by λ-abstraction, but globally in the context:

$$\dfrac{\dfrac{A : \mathit{Type} \vdash A : \mathit{Type} \quad A : \mathit{Type}, T : \mathit{Type}, x : T \vdash ?n^\frown[\,] : T}{A : \mathit{Type}, x : A \vdash ?n^\frown[T := A] : A}\ (\text{MV-}\beta\text{-Red})}{A : \mathit{Type} \vdash \lambda\, x : A.\ ?n^\frown[T := A] : A \to A}\ (\lambda)$$

The observation suggested by this example – the type of terms is invariant under reduction – is confirmed by Proposition 2.47 below.

Terminology: We say that rule (MV-weak) *introduces* variable z and that rule (MV-β-Red) is *applied at* (or *eliminates*) variable x.

In the sequel, we are mainly interested in proof problems $\mathcal{P}$ that are well-typed in the sense that the typing constraints imposed by $\mathit{ctxt}_\mathcal{P}$ and $\mathit{type}_\mathcal{P}$ are "internally consistent". This notion is best illustrated by the following counterexample: Assume that the valid proof problem $\mathcal{P}$ contains metavariables $?n_0, ?n_1$ with $A : \mathit{Type} \vdash ?n_0 : A$ and $B : \mathit{Type}, P : B \to \mathit{Prop} \vdash ?n_1 : P(?n_0)$. Obviously, the expression $P(?n_0)$ is ill-typed, as P expects an argument of type B and not of type A. In particular, such a proof problem does not admit a well-typed ground instantiation which satisfies the above typing constraint. Note, however, that even a well-typed proof problem may possibly be unsolvable.

Definition 2.42 (Well-Typed Proof Problem)
A valid proof problem $\mathcal{P}$ is called *well-typed* if for every $?n \in \mathcal{P}$, $\mathit{ctxt}_\mathcal{P}(?n) \vdash_\mathcal{P} \mathit{type}_\mathcal{P}(?n) : \mathit{Type}_j$ holds for $j \geq 0$.

Remember that validity (Definition 2.40) is a purely structural property of a proof problem, namely the property of being equipped with an irreflexive dependency order $\ll_\mathcal{P}$ on metavariables. Well-typedness is a stronger criterion which is only satisfied if the type constraints on metavariables are internally consistent.

2.5.2. Some properties of typing

In the following, some propositions will be proved that hold in an analogous form for the calculus *ECC*. These propositions are proved by an induction on derivations. Since the typing relation for the calculus with metavariables has been defined by simply adding rules to the existing calculus, we will present the proofs of the propositions below in a simplified form: After checking that

the validity of the proofs for the rules of the original *ECC* is not influenced by the new rules, it is sufficient to verify the new rules only.

In the proofs of the following propositions, we sometimes make reference to propositions of Section 2.5.3, in which it is shown (among others) that inferences in derivations can be ordered in a certain form. The proofs are not mutually dependent and so the arrangement of the propositions could be linearized accordingly. In order to maintain a coherent presentation, we have chosen not to do so.

Proposition 2.43 (Weakening)
If $\Gamma \vdash M : A$ and Γ' is a valid context containing every element of Γ, then $\Gamma' \vdash M : A$ holds.

Proof: By induction on the structure of the derivation of $\Gamma \vdash M : A$:

- For derivations ending in rule applications with a conclusion of the form $\Gamma \vdash ?n^\frown\sigma : A$, add an appropriate number of applications of the rule (MV-weak).
- For derivations ending in rule applications with a conclusion of the form $\Gamma \vdash M : A$, with M being no metavariable, take into account the induction hypothesis and re-apply the rule. □

The strengthening rule states that unused assumptions can be removed without affecting typeability. Recall that the concept of "free variable" (Definition 2.1) comprises not only variables which actually occur in a term, but also those variables which potentially occur in it. Without this extended notion, a strengthening rule is not admissible. Indeed, assume we were allowed to use a metavariable $?n$ with $\mathit{ctxt}(?n) \mathrel{\widehat{=}} \Gamma, y : A$ and $\mathit{type}(?n) \mathrel{\widehat{=}} A$ in context Γ alone, such that $\Gamma \vdash ?n : A$ would be derivable. Even though the solution $?n := y$ is correct since the solution term has the right type in the defining context of $?n$, we would obtain an incorrect term in context Γ where y is not typeable. This runs counter to our intention that well-typed instantiations should preserve typing (see Proposition 2.72).

Proposition 2.44 (Strengthening)
If $\Gamma, y : Y, \Gamma' \vdash M : A$ and $y \notin FV(M) \cup FV(A) \cup FV(\Gamma')$, then $\Gamma, \Gamma' \vdash M : A$.

Proof: As noted in [Luo90], a slightly stronger version of the proposition has to be proved, because the rules (app) and ($\preceq$) lose information about variable occurrences and so the induction hypothesis cannot be applied directly. The following proposition entails the desired statement:

> (*) If $\Gamma, y : Y, \Gamma' \vdash M : A$ and $y \notin FV(M) \cup FV(\Gamma')$, then there exists a $A' \preceq A$ such that $\Gamma, \Gamma' \vdash M : A'$.

The proof of (*) is by induction on the length of the derivation of $\Gamma, y : Y, \Gamma' \vdash M : A$ and, except for the metavariable rules, can be adapted from [Luo90]. We assume, furthermore, that derivations are in standard form (Definition 2.54 and Proposition 2.55). Thus, we only consider the cases in which the derivation ends in an application of:

- (MV-base): The judgement that is derived is of the form $ctxt(?n) \vdash ?n : type(?n)$. If $(y : Y) \in ctxt(?n)$, then $y \in FV(?n)$, so the claim vacuously holds.

- (MV-weak): The derivation is of the form:

$$\frac{\Gamma_1 \vdash T : Type_j \qquad \Gamma_1, \Delta_1 \vdash ?n^\frown\sigma : N}{\Gamma_1, z : T, \Delta_1 \vdash ?n^\frown\sigma : N} \text{ (MV-weak)}$$

  If y of (*) is the variable z introduced by this application, then this application can simply be dropped. Otherwise, apply induction hypothesis (after case distinction $(y : Y) \in \Gamma_1$ or $(y : Y) \in \Delta_1$).

- (MV-β-Red): The derivation is of the form:

$$\frac{\Gamma_1 \vdash t : T \qquad \Gamma_1, x : T, \Delta_1 \vdash ?n^\frown\sigma : N}{\Gamma_1, \Delta_1\{x := t\} \vdash (?n^\frown\sigma)\{x := t\} \ : \ N\{x := t\}} \text{ (MV-}\beta\text{-Red)}$$

  Since the derivation under consideration is in standard form, we can conclude that $x \in dom(ctxt(?n))$ and, in particular, that $x \in FV(?n)$. *Assume* $(y : Y) \in \Gamma_1$*:* Thus, the derivation is of the form

$$\frac{\Gamma_1', y : Y, \Gamma_1'' \vdash t : T \qquad \Gamma_1', y : Y, \Gamma_1'', x : T, \Delta_1 \vdash ?n^\frown\sigma : N}{\Gamma_1', y : Y, \Gamma_1'', \Delta_1\{x := t\} \vdash (?n^\frown\sigma)\{x := t\} : N\{x := t\}} \text{ (MV-}\beta\text{-Red)}$$

  Since $y \notin FV(\Gamma_1'', \Delta_1\{x := t\}) \cup FV((?n^\frown\sigma)\{x := t\}) \cup FV(N\{x := t\})$, we conclude by Lemma 2.24 that $y \notin FV(\Gamma_1'')$ and $y \notin FV(t)$, thus by induction hypothesis

  (1) $\Gamma_1', \Gamma_1'' \vdash t : T'$

  for a $T' \preceq T$. In particular, $y \notin FV(T')$ by Lemma 2.57. By Lemma 2.8, $\Gamma_1', y : Y, \Gamma_1'', x : T', \Delta_1 \vdash ?n^\frown\sigma : N$. Now, we can apply the induction hypothesis to obtain:

  (2) $\Gamma_1', \Gamma_1'', x : T', \Delta_1 \vdash ?n^\frown\sigma : N$

  Application of (MV-β-Red) to (1) and (2) gives the desired result.

Assume $(y : Y) \in \Delta_1\{x := t\}$*:* Thus, the derivation is of the form

$$\frac{\Gamma_1 \vdash t : T \qquad \Gamma_1, x : T, \Delta_1', y : Y', \Delta_1'' \vdash ?n^\frown\sigma : N}{\Gamma_1, \Delta_1'\{x := t\}, y : Y, \Delta_1''\{x := t\} \vdash (?n^\frown\sigma)\{x := t\} \ : \ N\{x := t\}} \text{ (MV-}\beta\text{-Red)}$$

for a Y' such that $Y'\{x := t\} \equiv Y$. By Lemma 2.24, $y \notin FV(\Delta_1'') \cup FV(?n^\frown\sigma) \cup FV(N)$, thus we can apply the induction hypothesis to obtain: $\Gamma_1, x : T, \Delta_1', \Delta_1'' \vdash ?n^\frown\sigma : N$. Together with $\Gamma_1 \vdash t : T$, application of (MV-β-Red) gives the desired result.

□

Proposition 2.45 (Cut)
If $\Gamma, x : T, \Delta \vdash M : A$ and $\Gamma \vdash t : T$ are derivable, then so is
$\Gamma, \Delta\{x := t\} \vdash M\{x := t\} : A\{x := t\}$

Proof: By induction on the structure of the derivation of $\Gamma, x : T, \Delta \vdash M : A$.

- For derivations ending in rule applications with a conclusion of the form $\Gamma, x : T, \Delta \vdash ?n^\frown\sigma : A$, apply rule (MV-$\beta$-Red).
- For derivations not ending with a metavariable, compare with Lemma 3.2.6 in [Luo90].

□

Proposition 2.46 (Principal Type)
For any context Γ and term M well-typed in Γ, there exists a principal type, i.e. a type A such that $\Gamma \vdash M : A$ and for all A', $\Gamma \vdash M : A'$ if and only if $A \preceq A'$ and A' is a type in Γ.

Proof: The proof can to a large extent be adapted from [Luo90]. The proof makes use of a diamond property of $\preceq$, which states that if $\Gamma \vdash M : A$ and $\Gamma \vdash M : B$, then there exists a C such that $\Gamma \vdash M : C$. This can be shown by a straightforward induction over the sum of the lengths of the two derivations; no essential adaptations are necessary for the metavariable rules. □

Proposition 2.47 (Subject Reduction)
If $\Gamma \vdash M : A$ and $M \twoheadrightarrow N$, then $\Gamma \vdash N : A$.

Proof: The property is shown for parallel one-step reduction $\Rightarrow_1$by induction on the derivation of $\Gamma \vdash M : A$ (the claim for $\twoheadrightarrow$ then follows by an induction on the length of a reduction by $\Rightarrow_1$). Again, only the derivations ending with a metavariable rule are considered, since the other cases can be dealt with as in the metavariable-free calculus.

The case (MV-base) is trivial, the case (MV-weak) requires a straightforward application of the induction hypothesis.

Assume, then, that the last rule is an application of (MV-β-Red). By Lemma 2.48, applications of the rule (MV-β-Red) can be ordered in such a way that a variable occurring before other variables in the context is reduced first. In a derivation rearranged in this manner, an application of (MV-β-Red) has the following form, where the substitution $\{x := t\}$ does not affect the terms $t_1, \ldots t_k$ because x does not occur in $t_1, \ldots t_k$:

$$\frac{\Gamma \vdash t : T \qquad \Gamma, x : T, \Delta \vdash ?n^\frown[x_1 := t_1, \ldots, x_k := t_k] : N}{\Gamma, \Delta\{x := t\} \vdash ?n^\frown[x_1 := t_1, \ldots, x_k := t_k, x := t] \ : \ N\{x := t\}} \text{ (MV-}\beta\text{-Red)}$$

By induction hypothesis, if $t \Rightarrow_1 t'$ and $t_i \Rightarrow_1 t'_i$ (for $i = 1, \ldots k$), then $\Gamma \vdash t' : T$ and $\Gamma, x : T, \Delta \vdash ?n^\frown[x_1 := t'_1, \ldots, x_k := t'_k] : N$. By application of rule (MV-β-Red), we obtain $\Gamma, \Delta\{x := t'\} \vdash ?n^\frown[x_1 := t'_1, \ldots, x_k := t'_k, x := t'] : N\{x := t'\}$, which by typing rule ($\preceq$) and by Lemma 2.8 can be converted into a derivation of $\Gamma, \Delta\{x := t\} \vdash ?n^\frown[x_1 := t'_1, \ldots, x_k := t'_k, x := t'] : N\{x := t\}$. □

2.5.3. Type inference algorithm

It is not immediately obvious from the typing rules for metavariables that type inference for metavariable terms is decidable. In particular, the rule (MV-β-Red) is problematic because it does not have a genuine subterm property for the term to be type-checked: There is no deterministic algorithm which computes $?n^\frown\sigma$ and t from the term $?n^\frown\sigma\{x := t\}$.

In the following, it will be shown that type inference is nevertheless decidable. In order to type-check a metavariable term $?n^\frown\sigma$, the typing rules of Figure 2.5 have to be applied in the forward direction, starting from $?n^\frown[\,]$ and incrementally building up the substitution σ. Even in the forward direction, there is a considerable degree of nondeterminism as to which rule has to be applied, and at which position. We will show that applications of the rules (MV-β-Red) and (MV-weak) can be permuted in a certain fashion, so that all judgements $\Gamma \vdash ?n^\frown\sigma : N$ that are derivable at all can be derived by following a "standard derivation" (Definition 2.54 and Proposition 2.55).

Lemma 2.48 (Permutation (MV-β-Red) / (MV-β-Red))
There is a derivation of the form:

$$\dfrac{\Gamma_1 \vdash t_1 : T_1 \qquad \dfrac{\dfrac{\Gamma_1, x_1 : T_1, \Gamma_2 \vdash t_2 : T_2 \qquad \Gamma_1, x_1 : T_1, \Gamma_2, x_2 : T_2, \Gamma_3 \vdash ?n^\frown\sigma : N}{\cdots}\,(\text{MV-}\beta\text{-Red})}{\Gamma \vdash ?n^\frown\sigma' : N'}\,(\text{MV-}\beta\text{-Red})$$

iff there is a derivation of the same judgement from the same premisses in which the order of application of (MV-β-Red) is reversed:

$$\dfrac{\Gamma_1, x_1 : T_1, \Gamma_2 \vdash t_2 : T_2 \qquad \dfrac{\dfrac{\Gamma_1 \vdash t_1 : T_1 \qquad \Gamma_1, x_1 : T_1, \Gamma_2, x_2 : T_2, \Gamma_3 \vdash ?n^\frown\sigma : N}{\cdots}\,(\text{MV-}\beta\text{-Red})}{\Gamma \vdash ?n^\frown\sigma' : N'}\,(\text{MV-}\beta\text{-Red})$$

This lemma is not surprising in view of the confluence results proved in Section 2.3.3. It states, in a modified form, that two different sequences of β-reductions can be joined. As opposed to the results proved for terms, β-reductions are here not entirely carried out within terms, but with some λ-abstractions shifted into the context.
Proof: When starting to reduce x_2 first, the following steps are carried out (where σ_2 is $\{x_2 := t_2\}$):

$$\dfrac{\Gamma_1, x_1 : T_1, \Gamma_2 \vdash t_2 : T_2 \qquad \Gamma_1, x_1 : T_1, \Gamma_2, x_2 : T_2, \Gamma_3 \vdash ?n^\frown\sigma : N}{\Gamma_1, x_1 : T_1, \Gamma_2, \Gamma_3\sigma_2 \vdash (?n^\frown\sigma)\sigma_2 : N\sigma_2}$$

Another application of (MV-β-Red) yields (with $\sigma_1 \equiv \{x_1 := t_1\}$):

$$\dfrac{\Gamma_1 \vdash t_1 : T_1 \qquad \Gamma_1, x_1 : T_1, \Gamma_2, \Gamma_3\sigma_2 \vdash (?n^\frown\sigma)\sigma_2 : N\sigma_2}{\Gamma_1, \Gamma_2\sigma_1, \Gamma_3\sigma_2\sigma_1 \vdash (?n^\frown\sigma)\sigma_2\sigma_1 : N\sigma_2\sigma_1}$$

Conversely, when reducing x_1 first, one obtains:

$$\dfrac{\Gamma_1 \vdash t_1 : T_1 \qquad \Gamma_1, x_1 : T_1, \Gamma_2, x_2 : T_2, \Gamma_3 \vdash ?n^\frown\sigma : N}{\Gamma_1, \Gamma_2\sigma_1, x_2 : T_2\sigma_1, \Gamma_3\sigma_1 \vdash (?n^\frown\sigma)\sigma_1 : N\sigma_1}$$

From $\Gamma_1, x_1 : T_1, \Gamma_2 \vdash t_2 : T_2$ and $\Gamma_1 \vdash t_1 : T_1$, it can be concluded that $\Gamma_1, \Gamma_2\sigma_1 \vdash t_2\sigma_1 : T_2\sigma_1$. By renewed application of (MV-β-Red), one obtains

$$\dfrac{\Gamma_1, \Gamma_2\sigma_1 \vdash t_2\sigma_1 : T_2\sigma_1 \qquad \Gamma_1, \Gamma_2\sigma_1, x_2 : T_2\sigma_1, \Gamma_3\sigma_1 \vdash (?n^\frown\sigma)\sigma_1 : N\sigma_1}{\Gamma_1, \Gamma_2\sigma_1, \Gamma_3\sigma_1\sigma_2' \vdash (?n^\frown\sigma)\sigma_1\sigma_2' : N\sigma_1\sigma_2'}$$

where $\sigma_2' \equiv \{x_2 := t_2\sigma_1\}$. With Lemma 2.34, it can be concluded that the judgement thus obtained is the same as the judgement resulting from applying the rules in inverse order. □

An application a_1 of rule (MV-weak) can be permuted above another application a_2 of (MV-weak) whenever a_1 introduces a variable z_1 occurring before the variable z_2 introduced by a_2. Of course, the converse does not hold, since z_2 may depend on z_1.

Lemma 2.49 (Permutation (MV-weak) / (MV-weak))
Assume $\Gamma_1, z_1 : Z_1, \Gamma_2, z_2 : Z_2, \Gamma_3 \vdash ?n^\frown\sigma : N$ is derivable by a derivation

$$\dfrac{\dfrac{\Gamma_1 \vdash Z_1 : \mathit{Type} \qquad \dfrac{\Gamma_1, \Gamma_2 \vdash Z_2 : \mathit{Type} \quad \Gamma_1, \Gamma_2, \Gamma_3 \vdash ?n^\frown\sigma : N}{\Gamma_1, \Gamma_2, z_2 : Z_2, \Gamma_3 \vdash ?n^\frown\sigma : N}\,(\text{MV-weak})}{\Gamma_1, z_1 : Z_1, \Gamma_2, z_2 : Z_2, \Gamma_3 \vdash ?n^\frown\sigma : N}\,(\text{MV-weak})$$

then the same judgement is derivable by a derivation in which z_1 is introduced first.

Proof: The side condition $\Gamma_1, z_1 : Z_1, \Gamma_2 \vdash Z_2 : \mathit{Type}$ which is required when introducing z_2 in the second step follows from $\Gamma_1, \Gamma_2 \vdash Z_2 : \mathit{Type}$ by Lemma 2.43. □

Applications of the rules (MV-weak) and (MV-β-Red) cannot be permuted uniformly. Indeed, there are examples when (MV-weak) has to be applied before (MV-β-Red) and vice versa in order to derive that a certain term is well-typed.

Example 2.50
In the following derivation, (MV-weak) has to be applied before (MV-β-Red) in order to obtain $A : \mathit{Type}, z : A \vdash ?n^\frown[x := z] : A$ from $A : \mathit{Type}, x : A \vdash ?n : A$.

$$\dfrac{\dfrac{A : \mathit{Type}, z : A \vdash z : A \qquad \dfrac{A : \mathit{Type} \vdash A : \mathit{Type} \quad A : \mathit{Type}, x : A \vdash ?n : A}{A : \mathit{Type}, z : A, x : A \vdash ?n : A}\,(\text{MV-weak})}{A : \mathit{Type}, z : A \vdash ?n^\frown[x := z] : A}\,(\text{MV-}\beta\text{-Red})$$

Example 2.51
In the following derivation, (MV-β-Red) has to be applied before (MV-weak) in order to obtain $Z : \mathit{Type}_0, z : Z \vdash ?n^\frown[X := \mathit{Type}_0] : \mathit{Type}_0$ from a metavariable defined by $X : \mathit{Type}_1, Z : X \vdash ?n : X$:

$$\dfrac{\dfrac{Z : \mathit{Type}_0 \vdash Z : \mathit{Type}_0 \qquad \dfrac{\vdash \mathit{Type}_0 : \mathit{Type}_1 \quad X : \mathit{Type}_1, Z : X \vdash ?n : X}{Z : \mathit{Type}_0 \vdash ?n^\frown[X := \mathit{Type}_0] : \mathit{Type}_0}\,(\text{MV-}\beta\text{-Red})}{Z : \mathit{Type}_0, z : Z \vdash ?n^\frown[X := \mathit{Type}_0] : \mathit{Type}_0}\,(\text{MV-weak})$$

Note that a permutation of rules is not possible here, since in the original sequent, Z is not a type.

In spite of these examples, the rules (MV-weak) and (MV-β-Red) can be permuted in case the variables introduced (by weakening) resp. eliminated (by (MV-β-Red)) occur in certain positions, as shown by the following Lemma 2.52 and Lemma 2.53.

Lemma 2.52 (Permutation of (MV-β-Red) above (MV-weak))
An application of (MV-β-Red) can be permuted above an application of (MV-weak) if the variable introduced by (MV-weak) occurs behind the variable eliminated by (MV-β-Red).

Contrast this with Example 2.50.
Proof: If there is a derivation of the form (where $\tau \mathrel{\widehat{=}} \{x := t\}$):

$$\dfrac{\Gamma \vdash t : T \qquad \dfrac{\Gamma, x : T, \Delta' \vdash Z : \mathit{Type} \quad \Gamma, x : T, \Delta', \Delta'' \vdash ?n^\frown\sigma : N}{\Gamma, x : T, \Delta', z : Z, \Delta'' \vdash ?n^\frown\sigma : N}\,(\text{MV-weak})}{\Gamma, \Delta'\tau, z : Z\tau, \Delta''\tau \vdash (?n^\frown\sigma)\tau : N\tau}\,(\text{MV-}\beta\text{-Red})$$

then there is a derivation of the form:

$$\dfrac{\Pi \qquad \dfrac{\Gamma \vdash t : T \quad \Gamma, x : T, \Delta', \Delta'' \vdash ?n^\frown\sigma : N}{\Gamma, \Delta'\tau, \Delta''\tau \vdash (?n^\frown\sigma)\tau : N\tau}\,(\text{MV-}\beta\text{-Red})}{\Gamma, \Delta'\tau, z : Z\tau, \Delta''\tau \vdash (?n^\frown\sigma)\tau : N\tau}\,(\text{MV-weak})$$

where Π is the following subproof (see Proposition 2.45)

$$\dfrac{\Gamma \vdash t : T \quad \Gamma, x : T, \Delta' \vdash Z : \mathit{Type}}{\Gamma, \Delta'\tau \vdash Z\tau : \mathit{Type}}$$

□

Lemma 2.53 (Permutation of (MV-weak) above (MV-β-Red))
An application of (MV-weak) can be permuted above an application of (MV-β-Red) if the variable introduced by (MV-weak) occurs before the variable eliminated by (MV-β-Red).

Contrast this with Example 2.51.
Proof: If there is a derivation of the form (where $\tau \mathrel{\widehat{=}} \{x := t\}$):

$$\dfrac{\Gamma' \vdash Z : \mathit{Type} \qquad \dfrac{\Gamma', \Gamma'' \vdash t : T \quad \Gamma', \Gamma'', x : T, \Delta \vdash ?n^\frown\sigma : N}{\Gamma', \Gamma'', \Delta\tau \vdash (?n^\frown\sigma)\tau : N\tau}\,(\text{MV-}\beta\text{-Red})}{\Gamma', z : Z, \Gamma'', \Delta\tau \vdash (?n^\frown\sigma)\tau : N\tau}\,(\text{MV-weak})$$

then there is a derivation of the form:

$$\dfrac{\Gamma', z : Z, \Gamma'' \vdash t : T \qquad \dfrac{\Gamma' \vdash Z : \mathit{Type} \quad \Gamma', \Gamma'', x : T, \Delta \vdash ?n^\frown\sigma : N}{\Gamma', z : Z, \Gamma'', x : T, \Delta \vdash ?n^\frown\sigma : N}\,(\text{MV-weak})}{\Gamma', z : Z, \Gamma'', \Delta\tau \vdash (?n^\frown\sigma)\tau : N\tau}\,(\text{MV-}\beta\text{-Red})$$

where $\Gamma', z : Z, \Gamma'' \vdash t : T$ can be derived from $\Gamma', \Gamma'' \vdash t : T$ by Proposition 2.43. □

We will now examine how judgements of the form $\Gamma \vdash ?n^\frown\sigma$ can be derived. Obviously, apart from applications of the rule $(\preceq)$, this can only be achieved by a series of applications of the three metavariable rules. In the following, let us call the *main branch* of a derivation of the judgement $\Gamma \vdash ?n^\frown\sigma : N$ the branch obtained by following the main premiss of the rules (MV-weak) and (MV-β-Red) up to and including rule (MV-base). The *main premiss* of (MV-weak) and (MV-β-Red) is the premiss in which a judgement of the form $\Gamma' \vdash ?n^\frown\sigma' : N'$ is derived.

Definition 2.54 (Standard derivation)
Assume that, for a metavariable $?n$, $\mathit{ctxt}(?n) \vdash \mathit{type}(?n) : \mathit{Type}_j$ is derivable by a derivation $\mathcal{D}$. A *standard derivation* of $\Gamma \vdash ?n^\frown\sigma : N$ is a derivation in which $\mathcal{D}$ is followed by an application of (MV-base) and in which all subsequent rule applications fulfill the following conditions:

- The rule (MV-β-Red) is only applied to variables $x \in \mathit{dom}(\mathit{ctxt}(?n))$.
- If rule (MV-β-Red) is applied to a variable x_i, then there is no previous application of (MV-β-Red) to a variable x_j with $i < j$ (that is, variables are eliminated from the front to the end).
- If rule (MV-weak) introduces a variable z, then no previous application of (MV-weak) has introduced a variable in the context segment behind z (that is, variables are introduced from the front to the end).

For example, the following derivation (shown without side conditions) is not standard, because (MV-β-Red) is applied to b at β_1 with substitution $\{b := z\}$ before it is applied to z, and because (MV-β-Red) is applied to $z \notin \mathit{dom}(\mathit{ctxt}(?n))$ at β_2 with substitution $\{z := a\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{T : Type, a, b : T \vdash T : Type}{T : Type, a, b : T \vdash ?n : T}\,(\text{MV-base}) - b}{T : Type, a : T, z : T, b : T \vdash ?n : T}\,(\text{MV-weak}) - w}{T : Type, a : T, z : T \vdash ?n^\frown[b := z] : T}\,(\text{MV-}\beta\text{-Red}) - \beta_1}{T : Type, a : T \vdash ?n^\frown[b := a] : T}\,(\text{MV-}\beta\text{-Red}) - \beta_2$$

When looking at the sequence of non-standard applications $b - w - \beta_1 - \beta_2$, we see that we can eliminate the wrong order of applications of (MV-β-Red) by permuting β_1 and β_2 according to Lemma 2.48, which gives us:

$$\dfrac{\dfrac{\dfrac{\dfrac{T : Type, a, b : T \vdash T : Type}{T : Type, a, b : T \vdash ?n : T}\,(\text{MV-base}) - b}{T : Type, a : T, z : T, b : T \vdash ?n : T}\,(\text{MV-weak}) - w}{T : Type, a : T, b : T \vdash ?n : T}\,(\text{MV-}\beta\text{-Red}) - \beta_2}{T : Type, a : T \vdash ?n^\frown[b := a] : T}\,(\text{MV-}\beta\text{-Red}) - \beta_1$$

Now, the sequence of applications $b - w - \beta_2$ is not in standard form because, in β_2, (MV-β-Red) is applied to $z \notin dom(ctxt(?n))$. Obviously, the applications w and β_2 are an unnecessary detour and can be dropped, which gives the standard application:

$$\dfrac{\dfrac{\dfrac{T : Type, a, b : T \vdash T : Type}{T : Type, a, b : T \vdash ?n : T}\,(\text{MV-base}) - b}{T : Type, a : T \vdash ?n^\frown[b := a] : T}\,(\text{MV-}\beta\text{-Red}) - \beta_1$$

The following proposition can be proved by a generalization of this procedure. The proof is quite involved; details can be found in Appendix A.3, page 175.

Proposition 2.55 (Existence of standard derivation)
Whenever $\Gamma \vdash ?n^\frown\sigma : N$ is derivable, then it is derivable by a standard derivation.

Proposition 2.56 (Decidability of Type Inference/ Type Checking)
Let $\mathcal{P}$ be a valid proof problem, Γ a context and M a term.

- In the calculus with metavariables, it is effectively decidable whether there exists a term A such that $\Gamma \vdash_{\mathcal{P}} M : A$ is derivable (*type inference*).
- Given a term A, it is effectively decidable whether $\Gamma \vdash_{\mathcal{P}} M : A$ holds or not (*type checking*).

Proof: Decidability of type checking follows from decidability of type inference by Proposition 2.46: Given Γ, M and A, use the type inference algorithm to compute a type A' such that $\Gamma \vdash M : A'$ holds, and then check that A and A' are related by $\preceq$, which is decidable.

For decidability of type inference, assume that the following statement holds:

> (*) For all Γ', M', whenever $\mathit{ctxt}(?n) \vdash \mathit{type}(?n) : \mathit{Type}$ is decidable for all $?n$ in Γ' and M', then it is decidable whether there is an A such that $\Gamma' \vdash M' : A$.

Note that $\mathcal{P}$ is a valid proof problem and thus $\ll_{\mathcal{P}}$ is well-founded. Then, by well-founded induction and the fact that $\mathit{ctxt}(?n)$ and $\mathit{type}(?n)$ only contain metavariables $?m \ll_{\mathcal{P}} ?n$, we can show that (*) entails decidability of type inference for arbitrary Γ and M.

Thus, it remains to show (*). If M' is not of the form $?n^\frown\sigma$, then decompose M' until reaching a variable or a term of the form $?n^\frown\sigma$. This is essentially the type inference algorithm for the metavariable-free calculus. If M' is of the form $?n^\frown\sigma$, then try to find a derivation of $\Gamma' \vdash M' : A$ (for an A to be determined), starting from $\mathit{ctxt}(?n) \vdash \mathit{type}(?n) : \mathit{Type}$ which, by assumption of (*), is derivable. By Proposition 2.55, it is sufficient to look for a standard derivation. In particular, (MV-weak) only has to be applied to introduce a variable $z \in \mathit{dom}(\Gamma')\backslash\mathit{dom}(\mathit{ctxt}(?n))$ and (MV-β-Red) only has to be applied to eliminate a variable $x \in \mathit{dom}(\sigma)$. Since there is only a finite number of such choices, it is decidable whether there exists an A such that $\Gamma' \vdash M' : A$ holds. □

The above results are relevant in the context of software development environments like TYPELAB, where it is desirable to be able to read in and check all expressions that are generated and eventually printed by the system.

However, for the applications in theorem proving that we have in mind, full-fledged type inference or type checking as presented above need not even be performed. Usually, metavariables are not entered afresh, but are manipulated internally by the theorem prover. The substitutions attached to a metavariable are built up incrementally, starting with a metavariable which initially contains not substitutions and whose type is known. The type of a term $?n^\frown\sigma$ can be stored and recomputed as new substitutions are performed. A problem that we will have to deal with in the following is, in this sense, a very restricted form of type inference, namely to determine whether a term M which is known to have type A in a context Γ keeps this type in a context $\Delta \subset \Gamma$. The following Lemmas 2.57 and 2.58 develop necessary and sufficient criteria for moving a term into a "smaller" context. For an implementation, this procedure can be

advantageous in that type checking can be reduced to manipulation of sets of variables and no term structures need to be built up.

Lemma 2.57
If $\Gamma \vdash M : A$, then $FV(M) \subseteq dom(\Gamma)$ and $FV(A) \subseteq dom(\Gamma)$.

Proof: Simple induction on the derivation of $\Gamma \vdash M : A$. Consider, for example, the rule (MV-β-Red):

$$\frac{\Gamma \vdash t : T \qquad \Gamma, x : T, \Delta \vdash ?n^\frown\sigma : N}{\Gamma, \Delta\{x := t\} \vdash (?n^\frown\sigma)\{x := t\} \ : \ N\{x := t\}} \text{ (MV-}\beta\text{-Red)}$$

By induction hypothesis, $FV(t) \subseteq dom(\Gamma)$, $FV(T) \subseteq dom(\Gamma)$, $FV(?n^\frown\sigma) \subseteq dom(\Gamma, x : T, \Delta)$ and $FV(N) \subseteq dom(\Gamma, x : T, \Delta)$.

Depending on whether $x \in FV(?n^\frown\sigma)$, by Lemma 2.23 and 2.24,

$$\begin{aligned} FV(?n^\frown\sigma\{x := t\}) &\subseteq FV(?n^\frown\sigma)\backslash\{x\} \cup FV(t) \\ &\subseteq dom(\Gamma, \Delta) = dom(\Gamma, \Delta\{x := t\}) \end{aligned}$$

Similarly, $FV(N\{x := t\}) \subseteq dom(\Gamma, \Delta)$. □

In particular, this lemma expresses that if $\Gamma \vdash M : A$ and Δ is a valid context with $\Delta \subset \Gamma$ and there is a variable x such that ($x \in FV(M)$ or $x \in FV(A)$) and $x \notin dom(\Delta)$, then $\Delta \nvdash M : A$.

Lemma 2.58
If $\Gamma \vdash M : A$, Δ is a valid context, $\Delta \subseteq \Gamma$ and $FV(M) \cup FV(A) \subseteq dom(\Delta)$, then $\Delta \vdash M : A$.

Proof: We remove the variable declarations in Γ that are not in Δ from right to left, which permits to use strengthening to show that typeability is preserved. More precisely, let n be $length(\Gamma) - length(\Delta)$. Then we show the statement by induction on n. For $n = 0$, the statement is obviously true. For the inductive step, assume that $\Gamma \equiv x_1 : T_1, \ldots x_k : T_k, \ldots x_m : T_m$, and x_k is the greatest index such that $x_k \notin dom(\Delta)$. Since Δ is a valid context, $x_k \notin FV(T_{k+1}, \ldots, T_m)$. Let Γ' be $x_1 : T_1, \ldots, x_{k-1} : T_{k-1}, x_{k+1} : T_{k+1}, \ldots x_m : T_m$. By the fact that $x_k \notin FV(M) \cup FV(A)$, we can conclude with Proposition 2.44 that $\Gamma' \vdash M : A$, and by application of the induction hypothesis, we have $\Delta \vdash M : A$. □

The following Lemma 2.59 shows that when strictly following the typing rules of Figure 2.5, type checking and type inference become inefficient. This calls for an optimization by an incremental type checking algorithm in the sense of Definition 2.14.

Lemma 2.59
Let Γ be a context and t and T terms, $\mathcal{P}$ a well-typed proof problem. Define M to be the set $MVars(\Gamma) \cup MVars(t) \cup MVars(T)$. Every derivation of $\Gamma \vdash t : T$ contains a subderivation of $ctxt(?k) \vdash ?k : type(?k)$ if and only if ($?k \in M$ or there exists an $?m \in M$ such that $?k \ll_{\mathcal{P}} ?m$).

Proof: Simple induction on the derivation of $\Gamma \vdash t : T$. Consider, in particular, a derivation of $ctxt(?n) \vdash ?n : type(?n)$ by an application of the rule (MV-base). If $M \triangleq MVars(ctxt(?n)) \cup MVars(type(?n))$, then by induction hypothesis, any derivation of $ctxt(?n) \vdash type(?n) : Type$ contains a subderivation of $ctxt(?k) \vdash ?k : type(?k)$ iff $?k$ is smaller than or equals a metavariable in M. By the definition of $\ll_{\mathcal{P}}$, the set of metavariables $\{?k \mid ?k \in M \text{ or } \exists ?m \in M . ?k \ll_{\mathcal{P}} ?m\}$ is just the set $\{?k \mid ?k \ll_{\mathcal{P}} ?n\}$. Thus, a derivation of $ctxt(?n) \vdash ?n : type(?n)$ contains a subderivation of $ctxt(?k) \vdash ?k : type(?k)$ iff ($?k \in M \cup \{?n\}$ or there exists an $?m \in M \cup \{?n\}$ such that $?k \ll_{\mathcal{P}} ?m$).
□

For a well-typed proof problem $\mathcal{P}$, it is redundant to verify that $ctxt(?n) \vdash type(?n) : Type$ for $?n \in \mathcal{P}$, since, by definition of well-typedness, this judgement is derivable.

Definition 2.60 (Incremental type inference with metavariables)
The incremental type inference algorithm for the calculus with metavariables is based on the set of rules of the calculus with metavariables in which the rule (MV-base) is replaced by the rule

$$\frac{}{ctxt_{\mathcal{P}}(?n) \vdash_{\mathcal{P}} ?n^\frown[] : type_{\mathcal{P}}(?n)} \text{ (MV-base)}$$

Corollary 2.61
For a well-typed proof problem $\mathcal{P}$, a judgement is derivable with standard type inference if and only if it is derivable with the incremental type inference algorithm of Definition 2.60.

Again, as already noted at the end of Proposition 2.15, this result is not particularly interesting. However, incremental type checking becomes relevant when considering instantiations which possibly lead to ill-typed proof problems, such that the above corollary is not applicable.

2.6. Solutions of Metavariables

2.6.1. Instantiations

Definition 2.62 (Instantiation)
An instantiation[1] ι is a function mapping a finite set of metavariables to terms, subject to the requirement that for every metavariable $?n$, the following occurs check condition holds:

- if $\iota(?n) = t$ and $?m \in \mathit{MVars}(t)$, then $\iota(?m) = ?m$

If an instantiation ι maps metavariables $?n_1, \ldots ?n_k$ to terms $t_1, \ldots t_k$, then ι is also written as $\{?n_1 := t_1, \ldots ?n_k := t_k\}$.

Instantiations are inductively extended to terms as follows:

- $\iota(x) = x$ for variables x.
- $\iota(\mathit{Prop}) = \mathit{Prop}$, $\iota(\mathit{Type}_i) = \mathit{Type}_i$
- $\iota(\mathcal{Q}x : T.M) = \mathcal{Q}x : \iota(T).\iota(M)$ for $\mathcal{Q} \in \{\lambda, \Pi, \Sigma\}$
- $\iota(f\ a) = (\iota(f)\ \iota(a))$
- $\iota(\mathit{pair}_T(t_1,\ t_2)) = \mathit{pair}_{\iota(T)}(\iota(t_1),\ \iota(t_2))$
- $\iota(\pi_i(t)) = \pi_i(\iota(t))$ for $i = 1, 2$
- $\iota(?n^\frown[x_1 := t_1 \ldots x_k := t_k]) = \iota(?n)\{x_1 := \iota(t_1) \ldots x_k := \iota(t_k)\}$

Instantiations can be extended to contexts by the following inductive definition:

- $\iota(\langle\rangle) = \langle\rangle$
- $\iota(x : T, \Gamma) = x : \iota(T), \iota(\Gamma)$

Note that, by Proposition 2.41, the definition of ι is well-founded.
Let us spell out some of the conditions of the definition in more detail:

- The requirement imposed on instantiations ι provides a strengthened form of an "occurs check". As a consequence of this requirement, an instantiation ι is idempotent, i.e. for every metavariable $?n \in \mathcal{P}$, $\iota(\iota(?n)) = \iota(?n)$. (The converse does not hold in a higher-order logic, i.e. idempotency does not imply the occurs-check condition. With our definition, we exclude an "instantiation" ι with $\iota(?f) = \lambda\, x : T.\ (?f\ a)$, which is idempotent modulo β-equivalence.)

[1] The term *instantiation* has been chosen to distinguish instantiation of metavariables from substitution of variables

- Instantiations map over terms without renaming bound variables. (cf. the concept of "grafting" in [DHK95]). Consider, for example, the following proof situation, in which a proof rule derived from the typing rule (λ) is applied:

$$\frac{\Gamma, x : A \vdash ?n_1 : A}{\Gamma \vdash ?n_0 : \Pi x : A.A}$$

  A solution of $?n_0$ is given by the instantiation ι_0 with $\iota_0(?n_0) := \lambda\, x : A.?n_1$. An appropriate instantiation of $?n_1$ is ι_1 with $\iota_1(?n_1) := x$. Then, $\iota_1(\iota_0(?n_0)) = \lambda\, x : A.x$ provides a solution for the original proof problem.

- When an instantiation maps over a metavariable, the substitution saved up so far is carried out on the solution of the metavariable (see Example 2.68 below).

For an exposition in a names-free setting, see the definitions with de Bruijn indices (Section A.1.4).

Definition 2.63 (Domain of an instantiation)
The domain of an instantiation ι is defined as the set

$$dom(\iota) := \{?n \mid \iota(?n) \neq ?n\}$$

Definition 2.64 (Ground instantiation)
A *ground instantiation* of a proof problem $\mathcal{P}$ is an instantiation ι such that for all $?n \in \mathcal{P}$, $\iota(?n)$ is a ground term, i.e. a term not containing metavariables.

In the following, we will often have to combine two instantiations ι_1 and ι_2 to produce a new one. For this, we will use the notation $\iota_1 \uplus \iota_2$. In general, a simple union of the instantiations considered as sets does not produce an instantiation satisfying the occurs check condition. Even computing a seemingly obvious "closure" can be problematic, since some higher-order effects interfere. Consider, for example, the combination $\{?n_1 := (?n_2\ x)\} \uplus \{?n_2 := \lambda\, y :?n_1.\ T\}$. First applying the solution of $?n_2$ to $(?n_2\ x)$ yields an instantiation which reduces to $\{?n_1 := T, ?n_2 := \lambda\, y : T.\ T\}$. First applying the solution of $?n_1$ to the solution of $?n_2$ leads to an instantiation containing $?n_2 := \lambda\, y : (?n_2\ x).\ T$, which is an invalid assignment. To avoid these difficulties, we impose some restrictions on instantiations for the definition of $\uplus$:

Definition 2.65 (Combination of instantiations)
Let ι_1, ι_2 be two instantiations with $dom(\iota_1) \cap dom(\iota_2) = \varnothing$ and such that whenever $(?n := t) \in \iota_2$ and $?m \in MVars(t)$, then $\iota_1(?m) = ?m$. Then $\iota_1 \uplus \iota_2$ is defined and equal to $(\iota_2 \circ \iota_1) \cup \iota_2$, otherwise it is undefined. The symbol $\uplus$ associates to the left.

Thus, computing $\iota_1 \uplus \iota_2$ consists in applying the solutions for metavariables in ι_2 to the solutions for metavariables in ι_1 and then adding the metavariable assignments of ι_2. Thus, $\uplus$ is in general not commutative. Obviously, the resulting instantiation fulfills the occurs check condition.

Definition 2.66 (Instantiation of proof problems)
Assume that $\mathcal{P} = (M_\mathcal{P}, ctxt_\mathcal{P}, type_\mathcal{P})$ is a valid proof problem and ι an instantiation. The instantiation $\iota(\mathcal{P}) = (M'_\mathcal{P}, ctxt'_\mathcal{P}, type'_\mathcal{P})$ is defined to be the proof problem consisting of:

- $M'_\mathcal{P} := M_\mathcal{P} \backslash dom(\iota)$
- $ctxt'_\mathcal{P}$ is defined as the function which, for $?n \in M'_\mathcal{P}$, yields $\iota(ctxt_\mathcal{P}(?n))$
- $type'_\mathcal{P}$ is defined as the function which, for $?n \in M'_\mathcal{P}$, yields $\iota(type_\mathcal{P}(?n))$

Intuitively, if $\mathcal{P}$ is a proof problem, then $\iota(\mathcal{P})$ is the proof problem that remains after providing a partial solution ι, the metavariables $?n \in dom(\iota)$ being removed from $\mathcal{P}$.

Example 2.67
Assume the proof problem $\mathcal{P}_1$ is given by:

$$A : Type, P : A \to Prop, a : A, h : (P\ a) \vdash ?n_1 : A$$

$$A : Type, P : A \to Prop, a : A, h : (P\ a) \vdash ?n_2 : (P\ ?n_1)$$

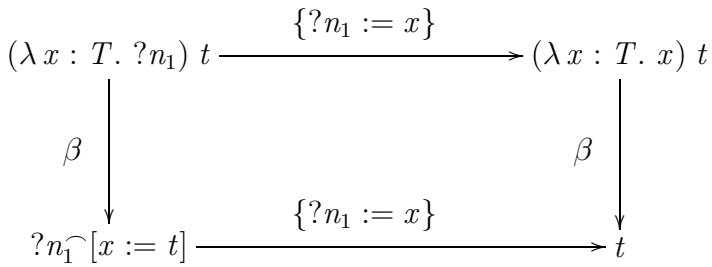
and the instantiation ι_1 by $\{?n_1 := a\}$. Then $\mathcal{P}_2 := \iota_1(\mathcal{P}_1)$ is the proof problem consisting of:

$$A : Type, P : A \to Prop, a : A, h : (P\ a) \vdash ?n_2 : (P\ a)$$

It can be solved by another instantiation $\iota_2 := \{?n_2 := h\}$.

After these definitions, we will first examine some syntactic properties of instantiations, i.e. properties of the term calculus that are not related to notions of typecorrectness.

Example 2.68
We resume the introductory Example 2.17, which lead to problems when treated naively, and show how the machinery developed so far permits to deal with it satisfactorily. Remember that metavariable $?n_1$ was defined by the following context and type: $T : Type, x : T \vdash ?n_1 : T$. First reducing the term $(\lambda\, x : T.\ ?n_1)\ t$ yields $?n_1\widehat{\ }[x := t]$, instantiating $?n_1$ with x and then carrying out the explicit substitution produces t, just the same as first providing the instantiation and then reducing.



Before generalizing this observation to Proposition 2.70 below, we need the following lemma:

Lemma 2.69
For terms P, N and instantiation ι: $\iota(P\{x := N\}) \equiv \iota(P)\{x := \iota(N)\}$

Proof: By induction on the structure of P. Most cases require a straightforward application of the definitions of substitutions and instantiations and of the induction hypothesis. Just consider the case where P is a metavariable:
$\iota((?n^\frown[x_1 := t_1 \dots x_n := t_n])\{x := N\}) =$
(by definition of substitution)
$\iota(?n^\frown[x_1 := t_1\{x := N\} \dots x := N \dots x_n := t_n\{x := N\}]) =$
(by definition of instantiation)
$\iota(?n)\{x_1 := \iota(t_1\{x := N\}) \dots x := \iota(N) \dots x_n := \iota(t_n\{x := N\})\} =$
(by induction hypothesis)
$\iota(?n)\{x_1 := \iota(t_1)\{x := \iota(N)\} \dots x := \iota(N) \dots x_n := \iota(t_n)\{x := \iota(N)\}\} =$
(properties of substitution)
$(\iota(?n)\{x_1 := \iota(t_1) \dots x_n := \iota(t_n)\})\{x := \iota(N)\} =$
(by definition of instantiation)
$\iota(?n^\frown[x_1 := t_1 \dots x_n := t_n])\{x := \iota(N)\}$ □

Proposition 2.70 (Commutativity of Instantiation and Reduction)
For terms M, N and instantiation ι: If $M \Rightarrow_1 N$, then $\iota(M) \Rightarrow_1 \iota(N)$.

Actually, this statement only expresses one direction of commutativity, see the remark further below.
Proof: By induction on the generation of $\Rightarrow_1$. We only consider the cases which require more than a simple application of the induction hypothesis:

- *Case:*
  $P \Rightarrow_1 P'$ and $N \Rightarrow_1 N'$ implies $(\lambda\, x : T.\ P)\ N \Rightarrow_1 P'\{x := N'\}$
  Assume $P \Rightarrow_1 P'$ and $N \Rightarrow_1 N'$, thus by induction hypothesis $\iota(P) \Rightarrow_1 \iota(P')$ and $\iota(N) \Rightarrow_1 \iota(N')$. Then:
  $\iota((\lambda\, x : T.\ P)\ N) = (\lambda\, x : \iota(T).\ \iota(P))\ \iota(N)$

(by induction hypothesis and definition of $\Rightarrow_1$)
$\Rightarrow_1 \iota(P')\{x := \iota(N')\}$
(by Lemma 2.69)
$= \iota(P'\{x := N'\})$

- *Case:*
  $M_i \Rightarrow_1 M_i'$ implies $?n^\frown[x_1 := M_1 \ldots x_n := M_n] \Rightarrow_1 ?n^\frown[x_1 := M_1' \ldots x_n := M_n']$
  Assume $M_i \Rightarrow_1 M_i'$, thus by induction hypothesis $\iota(M_i) \Rightarrow_1 \iota(M_i')$. Then:
  $\iota(?n^\frown[x_1 := M_1 \ldots x_n := M_n]) = \iota(?n)\{x_1 := \iota(M_1) \ldots x_n := \iota(M_n)\}$
  (by Lemma 2.35 and induction hypothesis)
  $\Rightarrow_1 \iota(?n)\{x_1 := \iota(M_1') \ldots x_n := \iota(M_n')\}$
  (by definition of instantiation)
  $= \iota(?n^\frown[x_1 := M_1' \ldots x_n := M_n'])$ □

Remark: Proposition 2.70 only expresses that if term M is reduced to term N, then $\iota(M)$ can be reduced to $\iota(N)$ (this corresponds to saying that the "lower triangle" in the figure of Example 2.68 implies the "upper triangle").

The other direction does not hold, i.e. if for terms M, N and instantiation ι, we have $\iota(M) \Rightarrow_1 \iota(N)$, we do not necessarily have $M \Rightarrow_1 N$. Take for example $M \mathrel{\widehat{=}} (?n\ t)$, $N \mathrel{\widehat{=}} t$ and $\iota \mathrel{\widehat{=}} \{?n := \lambda x : T.\ x\}$. After all, this result is not very surprising: Instantiations add information which allows for a more complex computational behaviour.

The notion of instantiation is not related to type correctness. This is remedied by the following definition:

Definition 2.71 (Properties of Instantiations)
Let $\mathcal{P} = (M_\mathcal{P}, ctxt_\mathcal{P}, type_\mathcal{P})$ be a proof problem.

- Assume $\mathcal{P}$ is valid. An instantiation ι is *valid* if $\iota(\mathcal{P})$ is a valid proof problem.
- Assume $\mathcal{P}$ is well-typed. An instantiation ι *preserves well-typedness* if $\iota(\mathcal{P})$ is a well-typed proof problem.
- Assume $\mathcal{P}$ is well-typed. An instantiation ι is *well-typed*, if ι is valid and, for every $?n \in M_\mathcal{P}$,

  $$\iota(ctxt_\mathcal{P}(?n)) \vdash_{\iota(\mathcal{P})} \iota(?n) : \iota(type_\mathcal{P}(?n))$$

  holds.

In the following, it will be shown that well-typed instantiations preserve well-typedness. This result is not particularly exciting for those term constructors over which instantiations map homomorphically. As far as metavariables

are concerned, however, the proof of the proposition provides an argument for the adequacy of the typing rules for metavariables.

Proposition 2.72 (Instantiation preserves typing)
Let $\mathcal{P}$ be a well-typed proof problem, Γ, t and T be a context resp. terms in which at most metavariables from $\mathcal{P}$ occur, and ι a well-typed instantiation for $\mathcal{P}$.

- If Γ $valid_c$ holds, then also $\iota(\Gamma)$ $valid_c$.
- If $\Gamma \vdash_{\mathcal{P}} t : T$ holds, then also $\iota(\Gamma) \vdash_{\iota(\mathcal{P})} \iota(t) : \iota(T)$

Proof: The proof is by induction on the structure of the derivation of $\Gamma \vdash_{\mathcal{P}} t : T$. Since the proof is evident for most of the term constructors, we will only consider the metavariable rules and, by means of example, the λ-rule.

- Suppose the last rule of the derivation is the λ-rule:

$$\frac{\begin{array}{c}\mathcal{D}\\ \Gamma, x : A \vdash_{\mathcal{P}} M : B\end{array}}{\Gamma \vdash_{\mathcal{P}} \lambda\, x : A.M : \Pi x : A.B}\ (\lambda)$$

  Then, by induction hypothesis, there exists a derivation $\mathcal{D}'$ ending with $\iota(\Gamma), x : \iota(A) \vdash_{\iota(\mathcal{P})} \iota(M) : \iota(B)$. Application of the λ-rule yields the desired result.

- Suppose the last rule of the derivation is (MV-base):

$$\frac{ctxt(?n) \vdash_{\mathcal{P}} type(?n) : Type_j}{ctxt(?n) \vdash_{\mathcal{P}} ?n^\frown[] : type(?n)}\ \text{(MV-base)}$$

  Since ι is a well-typed instantiation, $\iota(ctxt(?n)) \vdash_{\iota(\mathcal{P})} \iota(?n) : \iota(type(?n))$ holds.

- Suppose the last rule of the derivation is (MV-weak):

$$\frac{\Gamma \vdash_{\mathcal{P}} T : Type_j \qquad \Gamma, \Delta \vdash_{\mathcal{P}} ?n^\frown\sigma : N}{\Gamma, z : T, \Delta \vdash_{\mathcal{P}} ?n^\frown\sigma : N}\ \text{(MV-weak)}$$

  By induction hypothesis, there are derivations ending in:

  $$\iota(\Gamma) \vdash_{\iota(\mathcal{P})} \iota(T) : Type_j$$

  and

  $$\iota(\Gamma, \Delta) \vdash_{\iota(\mathcal{P})} \iota(?n^\frown\sigma) : \iota(N)$$

By the Weakening Lemma (Proposition 2.43) these derivations can be combined to a derivation of:

$$\iota(\Gamma), z : \iota(T), \iota(\Delta) \vdash_{\iota(\mathcal{P})} \iota(?n^\frown\sigma) : \iota(N)$$

- Suppose the last rule of the derivation is (MV-β-Red):

$$\frac{\Gamma \vdash_{\mathcal{P}} t : T \qquad \Gamma, x : T, \Delta \vdash_{\mathcal{P}} ?n^\frown\sigma : N}{\Gamma, \Delta\{x := t\} \vdash_{\mathcal{P}} (?n^\frown\sigma)\{x := t\} \ : \ N\{x := t\}} \text{ (MV-}\beta\text{-Red)}$$

By induction hypothesis, there are derivations ending in:

$$\iota(\Gamma) \vdash_{\iota(\mathcal{P})} \iota(t) : \iota(T)$$

and

$$\iota(\Gamma), x : \iota(T), \iota(\Delta) \vdash_{\iota(\mathcal{P})} \iota(?n^\frown\sigma) : \iota(N)$$

Then, by the Cut rule (Proposition 2.45), there is a derivation of:

$$\iota(\Gamma), \iota(\Delta)\{x := \iota(t)\} \vdash_{\iota(\mathcal{P})} \iota(?n^\frown\sigma)\{x := \iota(t)\} : \iota(N)\{x := \iota(t)\}$$

By definition of ι and Lemma 2.69, this is equal to:

$$\iota(\Gamma, \Delta\{x := t\}) \vdash_{\iota(\mathcal{P})} \iota(?n^\frown\sigma\{x := t\}) : \iota(N\{x := t\})$$

□

Corollary 2.73
If $\mathcal{P}$ is a well-typed proof problem and ι is a well-typed instantiation of $\mathcal{P}$, then ι preserves well-typedness.

Proof: It has to be shown that $\iota(\mathcal{P})$ is a well-typed proof problem, which means that $\mathit{ctxt}_{\iota(\mathcal{P})}(?n) \vdash_{\iota(\mathcal{P})} \mathit{type}_{\iota(\mathcal{P})}(?n) : \mathit{Type}$ for every $?n \in \iota(\mathcal{P})$. This follows from the fact that $\mathit{ctxt}_{\mathcal{P}}(?n) \vdash_{\mathcal{P}} \mathit{type}_{\mathcal{P}}(?n) : \mathit{Type}$ for every $?n \in \mathcal{P}$ and Proposition 2.72. □

Let us comment on our terminology, which has been chosen in accordance with the terminology for proof problems (cf. the remarks following Definition 2.42): The concept of "valid instantiation" refers to a structural property, namely the property of preserving the validity of a proof problem. A "well-typed instantiation" additionally satisfies the intended semantics of an instantiation, since it meets the typing constraints imposed by $\mathit{ctxt}_{\mathcal{P}}$ and $\mathit{type}_{\mathcal{P}}$. The notion of "preservation of well-typedness" is of intermediate strength, since, by the above corollary, it is implied by the notion of well-typed instantiation, whereas there are obviously instantiations that preserve well-typedness without being well-typed. Since this latter kind of instantiation is of no interest, we will in the following focus on valid and well-typed instantiations and analyze how they interact.

2.6.2. Verifying instantiations

Given a well-typed proof problem $\mathcal{P}$ and an instantiation ι for $\mathcal{P}$, two undesirable situations may occur:

- ι is not a valid instantiation, that is, $\iota(\mathcal{P})$ is not a valid proof problem.
- Even though ι is a valid instantiation, ι is not well-typed, violating some typing constraints.

Checking that an instantiation ι is valid may be expensive, if the metavariable dependencies of $\iota(\mathcal{P})$ have to be computed afresh, without taking into account the dependencies of $\mathcal{P}$. Likewise, verifying that an instantiation ι is well-typed can be intricate, since the metavariables which are solved by ι may depend on one another. Testing whether $\iota(\mathit{ctxt}(?n)) \vdash \iota(?n) : \iota(\mathit{type}(?n))$ holds for all metavariables $?n$ of a proof problem $\mathcal{P}$ can certainly be achieved by completely checking the judgement, including the context $\iota(\mathit{ctxt}(?n))$ whose validity may have been compromised by the instantiation.

Thus, altogether, it may be computationally expensive to globally verify properties of proof problems, such as validity and well-typedness. This section explores under which conditions these properties can be ensured locally:

- The (direct) metavariable dependency order $<_{\mathcal{P}}$ can be approximated by a kind of "finite differencing" (see Lemma 2.75). The validity of instantiations can be ensured when only admitting certain kinds of instantiation (see Proposition 2.76).
- For these instantiations, well-typedness can be established by only checking the correctness of solution terms with respect to their purported type without verifying the validity of contexts, i.e. by using incremental type checking.

Before presenting results, let us first motivate the relevance of these questions by an example. Proposition 2.72 seems to suggest that incremental type checking (in the sense of Definition 2.60) is possible, since a well-typed instantiation does not affect the type correctness of a context. However, even an incorrect instantiation can, by a self-sustaining effect, appear to be well-typed. The following example illustrates this problem:

Example 2.74
Assume the proof problem $\mathcal{P}$ is given by:

$$A, B : \mathit{Type}, a : A, b : B, P : A \to \mathit{Prop} \vdash ?n_1 : A$$

$$A, B : \mathit{Type}, a : A, b : B, P : A \to \mathit{Prop}, h : (P\ ?n_1) \vdash ?n_2 : (P\ ?n_1)$$

The instantiation $\iota_{\mathcal{P}}$ with $\{?n_1 := b, ?n_2 := h\}$ is not typecorrect, because any solution of $?n_1$ has to be of type A. However, the assignment $?n_2 := h$ is accepted by an incremental algorithm that checks whether $h : (P\ b)$ in context $A, B : \mathit{Type}, a : A, b : B, P : A \to \mathit{Prop}, h : (P\ b)$, where the validity of the context is not verified.

In this example, even an incremental type checking algorithm will ultimately reject the instantiation ι, because it correctly recognizes that $\iota(?n_1)$ does not have the required type. Quite in general, it is not clear under which conditions an incremental verification of an arbitrary instantiation ι for a proof problem $\mathcal{P}$ is possible. Instead of attempting to solve the problem in full generality, we restrict attention to *elementary* instantiations $\{?n := t\}$ which contain assignments to only one metavariable. They are of great practical interest, since all algorithms to be considered in the sequel, such as unification and tableau calculi, build up complex instantiations by a series of such elementary instantiations.

The following lemma characterizes how, for an arbitrary instantiation ι, the dependency order $\ll_{\iota(\mathcal{P})}$ of metavariables of $\iota(\mathcal{P})$ can be computed as a function of the metavariable order $\ll_{\mathcal{P}}$ and the metavariables occurring in the assignments $?n := t$ of ι.

Lemma 2.75
Let $\mathcal{P}$ be a valid proof problem, $<_{\mathcal{P}}$ its associated immediate dependency order (cf. Definition 2.40) and ι an instantiation. Then $<_{\iota(\mathcal{P})}$ can be computed as:

$$\begin{array}{rcl} <_{\iota(\mathcal{P})} & \subseteq & <_{\mathcal{P}} \setminus \{(?m, ?n) \mid \iota(?m) \neq ?m \text{ or } \iota(?n) \neq ?n\} \\ & & \cup\ \{(?k, ?m) \mid ?n <_{\mathcal{P}} ?m, \iota(?m) = ?m, ?k \in \mathit{MVars}(\iota(?n))\} \end{array}$$

Proof: Solved metavariables do not occur in $\iota(\mathcal{P})$ any more. This accounts for the pairs removed from $<_{\mathcal{P}}$. If a proof obligation $?m$ depends on $?n$, as in $\Gamma[?n] \vdash ?m : T[?n]$, and $?n$ is solved by a term $t[?k]$ containing $?k$, then a dependency from $?m$ on $?k$ is established. No other direct dependencies than those mentioned can arise. □

Thus, we have a "local" criterion which permits to compute the immediate dependency relation $<_{\mathcal{P}}$ by examining solution terms $\iota(?n)$ only, without traversing entire contexts. It has to be remarked that this is only an approximation (note the $\subseteq$ in the lemma), since metavariable dependencies may disappear by reduction. Take for example a proof problem containing $\Gamma \vdash ?m : ?P(?x)$ with a metavariable order $?P \ll_{\mathcal{P}} ?m$ and $?x \ll_{\mathcal{P}} ?m$. After an instantiation $\{?P := \lambda\, y :? T.\ Q(y)\}$, we obtain $\Gamma \vdash ?m : Q(?x)$, and the metavariable order

is simply $?x \ll_{\mathcal{P}} ?m$ and not also $?T \ll_{\mathcal{P}} ?m$. Altogether, a procedure using this approximation to calculate $\ll_{\mathcal{P}}$ is "sound" in that it recognizes all invalid instantiations, but not "complete" since it may reject valid instantiations. In such an event, it would be necessary to compute the correct order $\ll_{\mathcal{P}}$ by a global algorithm.

Proposition 2.76
Let $\mathcal{P}$ be a well-typed proof problem and $\iota \mathrel{\widehat{=}} \{?n := t\}$ an instantiation. If, for all $?k \in MVars(t)$, not $?n \ll_{\mathcal{P}} ?k$, then:

- $\iota(\mathcal{P})$ is a valid proof problem.
- ι can be verified by an incremental type checking algorithm.

Proof:

- It has to be shown that the order $\ll_{\iota(\mathcal{P})}$ is irreflexive. Assume, for a contradiction, that $\ll_{\iota(\mathcal{P})}$ is reflexive. Since proof problems have a finite set of metavariables, we may then assume that there are metavariables $?p_0, \ldots ?p_{n-1}$ such that $?p_0 <_{\iota(\mathcal{P})}, \ldots ?p_{n-1} <_{\iota(\mathcal{P})} ?p_0$.
  - Either, for all $i = 0, \ldots, n-1$, we have $?p_i <_{\mathcal{P}} ?p_{i+1}$ (indices modulo n). This contradicts the irreflexivity of $\ll_{\mathcal{P}}$.
  - Otherwise, there is an index i such that $?p_i <_{\iota(\mathcal{P})} ?p_{i+1}$ but not $?p_i <_{\mathcal{P}} ?p_{i+1}$. By Lemma 2.75, $?p_i \in MVars(t)$ and $?n <_{\mathcal{P}} ?p_{i+1}$. Again by Lemma 2.75 and an easy inductive argument, for all $m > i$, $?n \ll_{\mathcal{P}} ?p_m$, and for all $k \leq i$, $?p_k <_{\mathcal{P}} ?p_{k+1}$. Thus, $?p_m \ll_{\iota(\mathcal{P})} ?p_i$ for $m > i$ implies $?n <_{\mathcal{P}} ?p_m <_{\mathcal{P}} \ldots ?p_0 <_{\mathcal{P}} \ldots ?p_i$ and thus $?n \ll_{\mathcal{P}} ?p_i$ for $?p_i \in MVars(t)$, contradicting the fact that for all $?k \in MVars(t)$, not $?n \ll_{\mathcal{P}} ?k$.
- Define the set M as $\{?k \mid ?k \in MVars(t) \text{ and not } ?n \ll_{\mathcal{P}} ?k\}$. By well-founded induction on the order $\ll_{\mathcal{P}}$, it can be shown that for all $?k \in M$, $\iota(ctxt_{\mathcal{P}}(?k)) = ctxt_{\mathcal{P}}(?k)$ and $\iota(type_{\mathcal{P}}(?k)) = type_{\mathcal{P}}(?k)$. Therefore, whenever $ctxt_{\mathcal{P}}(?k) \vdash_{\mathcal{P}} type_{\mathcal{P}}(?k) : Type$ for $?k \in M$, then also $ctxt_{\iota(\mathcal{P})}(?k) \vdash_{\iota(\mathcal{P})} type_{\iota(\mathcal{P})}(?k) : Type$, therefore these subderivations can be pruned.

□

Proposition 2.76 is even a "tight fit" in that any elementary instantiation not satisfying the precondition of the proposition is not valid (neglecting the effects described in the remark following Lemma 2.75) and, consequently, not

well-typed. Indeed, assume that $\{?n := t[?k]\}$ is an instantiation and $?n \ll_{\mathcal{P}} ?k$. To keep the argument simple, even assume that $?n <_{\mathcal{P}} ?k$ – the more general argument is similar. Thus, $?k$ has a defining context $ctxt_{\mathcal{P}}(?k)$ of the form $\Gamma[?n]$ or, similarly, a defining type of the form $T[?n]$. After applying the above instantiation, one obtains $\Gamma[t[?k]] \vdash ?k : T[t[?k]]$, and the metavariable dependency order becomes cyclic. This happens, for instance, in Example 2.38.

2.7. Functional Representation of Metavariables

In the following, a functional representation of metavariables will be examined. The general idea is to replace a metavariable $?n$ of type T which depends on assumptions $x_1 : T_1, \ldots, x_k : T_k$ by a metavariable $?F$ which is of functional type $\Pi x_1 : T_1 \ldots \Pi x_k : T_k . T$ and which does not depend on assumptions.

Example 2.77
This procedure can best be illustrated by an example. Consider the following proof problem:

$$A : \mathit{Type}, P : A \to \mathit{Prop} \vdash ?n_0 : \forall\, a : A. \exists\, x : A.(P\ a) \to (P\ x)$$

Moving the universally quantified variable into the context and dissolving the existential quantifier, this problem can be decomposed into the subproblems:

$$A : \mathit{Type}, P : A \to \mathit{Prop}, a : A \vdash ?n_1 : A$$

$$A : \mathit{Type}, P : A \to \mathit{Prop}, a : A \vdash ?n_2 : (P\ a) \to (P\ ?n_1)$$

It can easily be verified that $\{?n_1 := a, ?n_2 := \lambda\, y : (P\ a).\ y\}$ is a typecorrect solution. The problem can also be stated with metavariables $?F_1$ and $?F_2$ which are the functional analogues of $?n_1$ and $?n_2$ and which are defined by:

$$\vdash ?F_1 : \Pi A : \mathit{Type}, P : A \to \mathit{Prop}, a : A.A$$

$$\vdash ?F_2 : \Pi A : \mathit{Type}, P : A \to \mathit{Prop}, a : A.(P\ a) \to (P\ (?F_1\ A\ P\ a))$$

The solution of this proof problem is $\{?F_1 := \lambda\, A : \mathit{Type}, P : A \to \mathit{Prop}, a : A.a, ?F_2 := \lambda\, A : \mathit{Type}, P : A \to \mathit{Prop}, a : A.\lambda\, y : (P\ a).\ y\}$.

The functional translation as defined more formally in Section 2.7.1 below is interesting in its own right, since it shows that, at least in principle, it is possible to do completely without metavariables depending on assumptions. A benefit

of this purely functional representation is that substitutions cannot take effect in metavariables: $?F\sigma$ will always be the same as $?F$, since all the variables in the domain of σ cannot occur free in a solution of $?F$. Thus, under a functional representation, there is no need to explicitly record substitutions applied to metavariables – altogether, the calculus essentially behaves like a calculus without metavariables. By showing that the reduction relations are preserved under the functional translation, this observation will be used in Section 2.7.2 to prove that since *ECC* without metavariables is strongly normalizing, so is the calculus ECC_M with metavariables and explicit substitutions.

Due to the advantages mentioned above, a functional encoding of metavariables is used in many algorithms and systems dealing with proof search in higher-order logic. For example, Huet's unification algorithm [Hue75] relies on a functional representation to avoid variable capture, and most variants and refinements follow in this tradition (see [Pre95] for a survey). Only recently have there been attempts to restate unification algorithms for the simply typed lambda calculus [DHK94, DHK95] and some of its refinements [DHKP96] in terms of calculi of explicit substitutions.

The transformation of metavariables into a functional encoding is closely related to Skolemization, as discussed in more depth in Section 4.4, and to "lifting" as presented by Paulson in [Pau89]. Miller [Mil92] examines methods of exchanging existential and universal quantifiers and develops a similar technique called "raising".

In the Isabelle system [Pau94], in λProlog [NM88] and the Elf system [Pfe89], proof obligations are represented in a restricted form of functional encoding. In Isabelle, the example problem given above would, upon refinement, yield something like the (single) subgoal

$$A : \mathit{Type}, P : A \to \mathit{Prop}, a : A \vdash (P\ a) \to (P\ (?f_1\ a))$$

with the metavariable $?f_1$ "normalized" to the context in which the proof started (of course, in Isabelle, the context with declarations of types and propositions is not made explicit). By the use of higher-order unification, the solution $?f_1 := \lambda\, x : A.\ x$ is established.

Although a functional representation of metavariables is preferable in some respects, it has some pragmatic drawbacks which have led us to adopt the kind of representation described in the previous sections:

- It is often more intuitive to work with a first-order than with a higher-order representation. Most users of an interactive prover find it easier to instantiate a metavariable $?n_1$ with a term a (as in the example above) than to instantiate the corresponding functional metavariable $?f_1$ with a projection function $\lambda\, x : A.\ x$.

- This problem is aggravated if proof obligations occur deeply nested inside a term and depend on several local assumptions. Then, the notation of a functional metavariable plus arguments tends to become quite clumsy. This situation notoriously arises if metavariables are generated as proof obligations for theorems inside a specification (cf. Section 1.1.4), as in the following example:

```
SPEC
     T: Type,
     x: T,
     AXIOM ax: (P_1 x),
     THEOREM th: (P_2 x)
END-SPEC
```

  The metavariable `?th: (P_2 x)` is generated in the context of the declarations for `T`, `x` and `ax`. The corresponding metavariable in functional representation would have to abstract over these declarations, like in: `?F_th:∀ T: Type, x: T, ax: (P_1 x). (P_2 x)`

2.7.1. Definition of the Functional Translation

This section gives a formal definition of the functional translation $\overline{t}$ of a term t. It is assumed throughout this section that the terms considered here only contain metavariables from a valid proof problem $\mathcal{P}$.

Definition 2.78 (Terms and contexts in functional representation) The translation of a term t to a term $\overline{t}$ with metavariables in functional representation is defined as follows:

- $\overline{x} = x$ for variables x
- $\overline{Prop} = Prop$, $\overline{Type_i} = Type_i$
- $\overline{\mathcal{Q}x : T.M} = \mathcal{Q}x : \overline{T}.\overline{M}$ for $\mathcal{Q} \in \{\lambda, \Pi, \Sigma\}$
- $\overline{(f\ a)} = (\overline{f}\ \overline{a})$
- $\overline{pair_T(t_1,\ t_2)} = pair_{\overline{T}}(\overline{t_1},\ \overline{t_2})$
- $\overline{\pi_i(t)} = \pi_i(\overline{t})$ for $i = 1, 2$
- If $ctxt(?n) \equiv \Gamma \equiv x_1 : T_1, \ldots x_k : T_k$ and $type(?n) \equiv T$, then $\overline{?n^\frown\sigma} = (?F\ x_1\overline{\sigma} \ldots x_k\overline{\sigma})$, where $?F$ is a unique fresh metavariable associated to $?n$ such that:

- $ctxt(?F) = \langle\rangle$
- $type(?F) = \Pi\overline{\Gamma}.\overline{T}$[2]

and where for a substitution $\sigma \equiv \{x'_1 := t_1, \ldots x'_m := t_m\}$, the substitution $\overline{\sigma}$ is defined as $\{x'_1 := \overline{t_1}, \ldots x'_m := \overline{t_m}\}$.

The translation of a context Γ to a context $\overline{\Gamma}$ is an obvious extension of the translation of terms.

It follows from Proposition 2.41 that the translation is well-defined.

Example 2.79
Consider metavariable $?n$ with

$$ctxt(?n) = A, B : Type, f : A \to A, P : B \to Prop, x_1 : A, x_2 : B$$

and $type(?n) = (P\ x_2)$. Then

$$\overline{?n^\frown[B := A, x_2 := (f\ x_1)]} = ?F\ A\ A\ f\ P\ x_1\ (f\ x_1)$$

with

$$?F : \Pi A, B : Type, f : A \to A, P : B \to Prop, x_1 : A, x_2 : B.(P\ x_2)$$

Note that the type of $?n^\frown[B := A, x_2 := (f\ x_1)]$ is $(P\ (f\ x_1))$, which is also the translation of the type of $?F\ A\ A\ f\ P\ x_1\ (f\ x_1)$.

In general, it can be shown that the translation preserves typing:

Proposition 2.80
If $\Gamma \vdash t : T$, then $\overline{\Gamma} \vdash \overline{t} : \overline{T}$.

Proof: The proof is by induction on the structure of the derivation $\Gamma \vdash t : T$.

The proof is routine for the rules of the base calculus ECC. Assume, for example, that the last rule of the derivation is the λ-rule:

$$\frac{\begin{array}{c}\mathcal{D}\\ \Gamma, x : A \vdash M : B\end{array}}{\Gamma \vdash \lambda\, x : A.M : \Pi x : A.B}\ (\lambda)$$

Then, by induction hypothesis, there exists a derivation $\overline{\mathcal{D}}$ ending with $\overline{\Gamma, x : A} \vdash \overline{M} : \overline{B}$, from which one can conclude by the definition of the translation and an application of the (λ) rule that $\overline{\Gamma} \vdash \overline{\lambda\, x : A.\ M} : \overline{\Pi x : A.\ B}$.

The only non-trivial cases are the applications of the metavariable rules:

[2]Remember here and in the following the convention of writing $\Pi\Gamma.T$ instead of $\Pi x_1 : T_1, \ldots \Pi x_k : T_k.T$, if $\Gamma \equiv x_1 : T_1, \ldots x_k : T_k$ (and similarly for λ-abstractions).

- Application of the rule (MV-base):
  Assume the last rule in a derivation is:

$$\frac{ctxt(?n) \vdash type(?n) : Type_j}{ctxt(?n) \vdash ?n^\frown[] : type(?n)} \text{ (MV-base)}$$

  Assume that $ctxt(?n)$ has the form $x_1 : T_1, \ldots x_k : T_k$. Then, it has to be shown that $\overline{ctxt(?n)} \vdash (?F\ x_1 \ldots x_k) : \overline{type(?n)}$.

  By induction hypothesis, there is a derivation for $\overline{ctxt(?n)} \vdash \overline{type(?n)} : Type_j$, and by repeated application of the (Π-Form$_2$) rule, we can conclude that $\langle\rangle \vdash \Pi\overline{ctxt(?n)}.\overline{type(?n)} : Type_i$ (for an i possibly different from j). This shows that $\langle\rangle \vdash ?F : \Pi\overline{ctxt(?n)}.\overline{type(?n)}$ is a valid metavariable.

  Repeated application of weakening yields (renaming bound variables):

$$\overline{ctxt(?n)} \vdash ?F : \Pi\overline{ctxt(?n)}.\overline{type(?n)}$$

  After repeated application of the (app) rule, we obtain:

$$\overline{ctxt(?n)} \vdash (?F\ x_1 \ldots x_k) : \overline{type(?n)}$$

  which had to be shown.

- Application of the rule (MV-weak):
  Assume the last rule in a derivation is:

$$\frac{\Gamma \vdash T : Type_j \quad \Gamma, \Delta \vdash ?n^\frown\sigma : N}{\Gamma, z : T, \Delta \vdash ?n^\frown\sigma : N} \text{ (MV-weak)}$$

  Then, by induction hypothesis, there are derivations for:

  - $\overline{\Gamma} \vdash \overline{T} : Type_j$
  - $\overline{\Gamma}, \overline{\Delta} \vdash ?F\ x_1\overline{\sigma} \ldots x_k\overline{\sigma} : \overline{N}$

  The weakening lemma (Proposition 2.43) then permits to conclude that $\overline{\Gamma}, z : \overline{T}, \overline{\Delta} \vdash ?F\ x_1\overline{\sigma} \ldots x_k\overline{\sigma} : \overline{N}$.

- Application of the rule (MV-β-Red):
  Assume the last rule in a derivation is:

$$\frac{\Gamma \vdash t : T \qquad \Gamma, x : T, \Delta \vdash ?n^\frown\sigma : N}{\Gamma, \Delta\{x := t\} \vdash (?n^\frown\sigma)\{x := t\} \ : \ N\{x := t\}} \text{ (MV-}\beta\text{-Red)}$$

  By induction hypothesis, there are derivations for:

– $\overline{\Gamma} \vdash \overline{t} : \overline{T}$

– $\overline{\Gamma}, x : \overline{T}, \overline{\Delta} \vdash \overline{(?n^\frown\sigma)} : \overline{N}$.

With the Cut rule (Proposition 2.45), it can be concluded that $\overline{\Gamma}, \overline{\Delta}\{x := \overline{t}\} \vdash \overline{(?n^\frown\sigma)}\{x := \overline{t}\} : \overline{N}\{x := \overline{t}\}$. By noting that substitutions can be pulled inside the translation (see Lemma 2.81 below), we obtain $\overline{\Gamma, \Delta\{x := t\}} \vdash \overline{(?n^\frown\sigma)\{x := t\}} : \overline{N\{x := t\}}$, which had to be shown.

□

Lemma 2.81
For all terms M and N, $\overline{M\{x := N\}} \equiv \overline{M}\{x := \overline{N}\}$

Proof: By induction on the structure of M. Only the case with M of the form $?n^\frown\sigma$ is interesting, the other cases require a straightforward application of the induction hypothesis.

Assume in the following that $\sigma = \{x'_1 := t_1, \ldots x'_m := t_m\}$, $ctxt(?n) = x_1 : T_1, \ldots x_k : T_k$ and $\{x'_1, \ldots x'_m\} \subseteq \{x_1, \ldots x_k\}$, so $\overline{\sigma} = \{x'_1 := \overline{t_1}, \ldots x'_m := \overline{t_m}\}$.

- If $x \in \{x_1, \ldots, x_k\}$: By definition of substitution:
  $\overline{(?n^\frown\sigma)\{x := N\}}$
  $= \overline{?n^\frown[x'_1 := t_1\{x := N\}, x := N, \ldots x'_m := t_m\{x := N\}]}$
  $= ?F\ x_1\overline{\tau} \ldots x_k\overline{\tau}$
  (for $\overline{\tau} = \{x'_1 := \overline{t_1\{x := N\}}, x := \overline{N}, \ldots x'_m := \overline{t_m\{x := N\}}\}$, by definition of translation).

  Applying the induction hypothesis, $\overline{\tau}$ can be rewritten as
  $\{x'_1 := \overline{t_1}\{x := \overline{N}\}, x := \overline{N}, \ldots x'_m := \overline{t_m}\{x := \overline{N}\}\}$

  By definition of substitution:
  $?F\ x_1\overline{\tau} \ldots x_k\overline{\tau} = (?F\ (x_1\overline{\sigma}\{x := \overline{N}\}) \ldots (x_k\overline{\sigma}\{x := \overline{N}\}))$
  $= (?F\ x_1\overline{\sigma} \ldots x_k\overline{\sigma})\{x := \overline{N}\}$

  Applying the translation backwards, we obtain $\overline{?n^\frown\sigma}\{x := \overline{N}\}$.

- If $x \notin \{x_1, \ldots, x_k\}$: Similar to the above reasoning.

□

Definition 2.82 (Proof problems in functional representation)
A proof problem

$$\mathcal{P} := \{(ctxt(?n_i), ?n_i, type(?n_i)) \mid i \in \mathcal{I}\}$$

(where $\mathcal{I}$ is a finite index set) is translated to a proof problem

$$\overline{\mathcal{P}} := \{(\langle\rangle, ?F_i, \Pi\overline{ctxt(?n_i)}.\overline{type(?n_i)}) \mid i \in \mathcal{I}\}$$

with metavariables in functional representation.

Proposition 2.80 implies that each $(\langle\rangle, ?F_i, \Pi\overline{ctxt(?n_i)}.\overline{type(?n_i)})$ is a well-typed metavariable if $?n_i$ is one. Furthermore, assume that $\mathcal{P}$ is a valid proof problem with associated order $\ll_{\mathcal{P}}$, which is the transitive closure of $<_{\mathcal{P}}$ (see Definition 2.40). Then

$$\begin{array}{rll} ?n_i <_{\mathcal{P}} ?n_j & \text{iff} & ?n_i \in MVars(ctxt(?n_j)) \text{ or } ?n_i \in MVars(type(?n_j)) \\ & \text{iff} & ?F_i \in MVars(\overline{ctxt(?n_j)}) \text{ or } ?F_i \in MVars(\overline{type(?n_j)}) \\ & \text{iff} & ?F_i \in MVars(ctxt(?F_j)) \text{ or } ?F_i \in MVars(type(?F_j)) \\ & \text{iff} & ?F_i <_{\overline{\mathcal{P}}} ?F_j \end{array}$$

Since $\ll_{\mathcal{P}}$ is an irreflexive partial order, so is $\ll_{\overline{\mathcal{P}}}$, the transitive closure of $<_{\overline{\mathcal{P}}}$, and therefore $\overline{\mathcal{P}}$ is also a valid proof problem.

Proposition 2.83
The proof problem $\mathcal{P}$ has a solution iff the proof problem $\overline{\mathcal{P}}$ has one.

Proof: Assume that $\iota_{\mathcal{P}}$ is a solution of $\mathcal{P}$. Then, for every metavariable $?n_i \in \mathcal{P}$, $\iota(ctxt(?n_i)) \vdash \iota(?n_i) : \iota(type(?n_i))$. Therefore,

$$\vdash \lambda\, \iota(ctxt(?n_i)).\iota(?n_i) : \Pi\iota(ctxt(?n_i)).\iota(type(?n_i))$$

For $?F_i \in \overline{\mathcal{P}}$ corresponding to $?n_i \in \mathcal{P}$, define $\iota_{\overline{\mathcal{P}}}(?F_i)$ as $\lambda\,\overline{\iota(ctxt(?n_i))}.\overline{\iota(?n_i)}$. By preservation of typing (Proposition 2.80),

$$\vdash \iota_{\overline{\mathcal{P}}}(?F_i) : \Pi\overline{\iota(ctxt(?n_i))}.\overline{\iota(type(?n_i))}$$

thus $\iota_{\overline{\mathcal{P}}}$ is a typecorrect solution of the proof problem $\overline{\mathcal{P}}$.

Conversely, assume that $\iota_{\overline{\mathcal{P}}}$ is a solution of $\overline{\mathcal{P}}$. Thus, for each $?F_i \in \overline{\mathcal{P}}$,

$$\overline{\iota_{\overline{\mathcal{P}}}(ctxt(?n_i))} \vdash \iota_{\overline{\mathcal{P}}}(?F_i\, x_1, \ldots x_k) : \overline{\iota_{\overline{\mathcal{P}}}(type(?n_i))}$$

if $x_1, \ldots x_k$ are the variables of $\overline{\iota_{\overline{\mathcal{P}}}(ctxt(?n_i))}$. For $?n_i$ corresponding to $?F_i$, one can therefore choose $\iota(?n_i) := \iota_{\overline{\mathcal{P}}}(?F_i\, x_1, \ldots x_k)$. By an induction over the order $\ll_{\mathcal{P}}$, it can then be shown that $\iota(ctxt(?n_i)) \vdash \iota(?n_i) : \iota(type(?n_i))$. □

For an illustration of the "if" direction of this proposition, we refer the reader back to Example 2.77. For the "only if" direction, consider the following:

Example 2.84
We resume Example 2.77 to show how any solution of:

$$\vdash ?F_1 : \Pi A : \mathit{Type}, P : A \to \mathit{Prop}, a : A.A$$

$$\vdash ?F_2 : \Pi A : \mathit{Type}, P : A \to \mathit{Prop}, a : A.(P\ a) \to (P\ (?F_1\ A\ P\ a))$$

can be converted into a solution of:

$$A : \mathit{Type}, P : A \to \mathit{Prop}, a : A \vdash ?n_1 : A$$

$$A : \mathit{Type}, P : A \to \mathit{Prop}, a : A \vdash ?n_2 : (P\ a) \to (P\ ?n_1)$$

The construction given in the proof of Proposition 2.83 permits to convert any solution f_1 resp. f_2 of $?F_1$ resp. $?F_2$ to solutions $(f_1\ P\ a)$ resp. $(f_2\ P\ a)$ of $?n_1$ resp. $?n_2$. In particular, if $f_1 := \lambda\, A' : \mathit{Type}, P' : A' \to \mathit{Prop}, a' : A'.a'$ and $f_2 := \lambda\, A' : \mathit{Type}, P' : A' \to \mathit{Prop}, a' : A'.\lambda\, y : (P'\ a').\ y$, as suggested in Example 2.77, then we get back the solutions $?n_1 := a$ and $?n_2 := \lambda\, y : (P\ a).\ y$.

2.7.2. Strong Normalization of the Calculus with Metavariables

In this section, strong normalization of the calculus with metavariables ECC_M will be proved by mapping reductions in ECC_M to reductions in the calculus ECC without metavariables. In order to do this, a mapping from terms containing metavariables to metavariable-free terms has to be provided. The translation defined in the last section is already quite a good approximation, because no substitutions can be performed inside the context-independent metavariables $?F$ and therefore the calculus almost behaves like a calculus without metavariables, at least as far as reductions are concerned.

To formalize this idea, we define a new translation that makes the metavariables disappear. We keep the notation $\overline{t}$ and only indicate where this translation differs from the one previously defined:

> If $\mathit{ctxt}(?n) = \Gamma = x_1 : T_1, \ldots x_k : T_k$ and $\mathit{type}(?n) = T$, then $\overline{?n^\frown\sigma} = (F\ x_1\overline{\sigma} \ldots x_k\overline{\sigma})$, where F of type $\Pi\overline{\Gamma}.\overline{T}$ is a unique *constant* associated to $?n$.

When typing the terms obtained by this translation, the new constants F have to be added at the front of the typing contexts. Here again, we assume that all metavariables are stemming from a valid proof problem $\mathcal{P}$, and so the order

$\ll_{\mathcal{P}}$ can be used to arrange the new constants in such a way that no mutual dependencies arise. The statements made above (in particular Proposition 2.80 and Lemma 2.81) now carry over to the new translation.

Rather than spelling out the formalization in more detail, we present the following example to illustrate the concepts:

Example 2.85
Consider the proof problem $\mathcal{P}$ with metavariables $?n_1$ and $?n_2$ of Example 2.77. The translation generates the constants

$$F_1 : \Pi A : \mathit{Type}, P : A \to \mathit{Prop}, a : A.A$$

$$F_2 : \Pi A : \mathit{Type}, P : A \to \mathit{Prop}, a : A.(P\ a) \to (P\ (F_1\ A\ P\ a))$$

Note that $?n_1 \ll_{\mathcal{P}} ?n_2$, so F_1 has to be placed before F_2 in the augmented contexts used for typing. For example, the translation of the term $(P\ ?n_1)$ yields the term $(P\ (F_1\ A\ P\ a))$, which is of type Prop in context

$$F_1 : \ldots, F_2 : \ldots, A : \mathit{Type}, P : A \to \mathit{Prop}, a : A$$

The following lemma makes precise the idea of simulating reductions in the calculus with metavariables by reductions in the metavariable-free calculus.

Lemma 2.86
If $M \Rightarrow_1 M'$ and $M \neq M'$, then $\overline{M} \Rightarrow_1 \overline{M'}$ and $\overline{M} \neq \overline{M'}$.

Proof: The proof is by induction on the generation of $\Rightarrow_1$. All the cases not dealt with in the following require a straightforward application of the induction hypothesis.

- Assume M is of the form $(\lambda\, x : T.\ P)\ N$, and M' of the form $P'\{x := N'\}$, where $P \Rightarrow_1 P'$ and $N \Rightarrow_1 N'$. By induction hypothesis, $\overline{P} \Rightarrow_1 \overline{P'}$ and $\overline{N} \Rightarrow_1 \overline{N'}$, so $\overline{M} = (\lambda\, x : \overline{T}.\ \overline{P})\ \overline{N} \Rightarrow_1 \overline{P'}\{x := \overline{N'}\} = \overline{P'\{x := N'\}} = \overline{M'}$ by an application of Lemma 2.81.

- Assume M is of the form $?n^\frown[x'_1 := t_1, \ldots, x'_m := t_m]$, and M' is of the form $?n^\frown[x'_1 := t'_1, \ldots, x'_m := t'_m]$, with $t_i \Rightarrow_1 t'_i$. We define σ as $\{x'_1 := t_1, \ldots, x'_m := t_m\}$ and σ' as $\{x'_1 := t'_1, \ldots, x'_m := t'_m\}$.

  For a metavariable $?n$ with $\mathit{ctxt}(?n) = x_1 : T_1, \ldots x_k : T_k$, the term $\overline{M}$ is of the form $F\ x_1\overline{\sigma} \ldots x_k\overline{\sigma}$. By parallel reduction, using the induction hypothesis, $\overline{M} = F\ x_1\overline{\sigma} \ldots x_k\overline{\sigma} \Rightarrow_1 F\ x_1\overline{\sigma'} \ldots x_k\overline{\sigma'} = \overline{M'}$.

  Since $M \neq M'$, there has to be at least one t_i with $t_i \neq t'_i$. Because the variables $x'_1, \ldots x'_m$ are all contained in the domain of the context $x_1 : T_1, \ldots x_k : T_k$, for at least one i, $x_i\overline{\sigma} \neq x_i\overline{\sigma'}$ and therefore also $\overline{M} \neq \overline{M'}$. □

Proposition 2.87 (Strong Normalization)
The calculus ECC_M is strongly normalizing for reduction $\Rightarrow_1$.

Proof: Assume, to the contrary, that there is a well-typed term M which permits an infinite sequence of reduction steps: $M \equiv M_0 \Rightarrow_1 M_1 \Rightarrow_1 \ldots$, with each $M_i \neq M_{i+1}$. Then, by Proposition 2.80, the term $\overline{M}$ is well-typed in the calculus ECC without metavariables, and by Lemma 2.86, there is also an infinite sequence of terms $\overline{M} \equiv \overline{M_0} \Rightarrow_1 \overline{M_1} \Rightarrow_1 \ldots$, with each $\overline{M_i} \neq \overline{M_{i+1}}$. This contradicts the strong normalization property of ECC (Proposition 2.11).
□

3. A Sequent Calculus Characterization

3.1. Natural Deduction and Sequent Systems

3.1.1. Survey

The calculus presented in Section 2.1 is adequate for type inference and type checking, since it allows for derivations which are driven by the structure of the term under consideration: When inferring the type of a term M in context Γ, i.e. when trying to find a term $?A$ such that $\Gamma \vdash M :?A$, it is sufficient to choose a rule according to the outermost constructor of M and to apply this rule backwards. This gives rise to new type inference problems $\Gamma_i \vdash M_i :?A_i$ which are entirely determined by M since the M_i are subterms of M. Thus, this calculus, to which we will henceforth refer as ECC_N, has a subterm property with respect to the *terms* of its typing judgements.

However, there is no such subterm property for *types*, which makes the calculus inappropriate for proof search, i.e. for determining a term $?M$ such that $\Gamma \vdash ?M : A$ holds. In particular, the rules (app), (π_1), (π_2) and $(\preceq)$ destroy the subterm property, since their premisses contain types that do not occur in their conclusions. The purpose of this section is to develop calculi having a subterm property for types, and which permit proof search by structural decomposition of types. Not surprisingly, the subterm property for terms gets lost, so these calculi are not useful for type inference.

The following considerations are inspired by the work of Gentzen [Gen34], who introduces *natural deduction calculi* with a schema of introduction / elimination rules and *sequent calculi* with left / right rules and proves their equivalence. The procedure roughly works as follows:

- To establish a subterm property on types, define a calculus ECC_G[1] (Def-

[1] A remark on notation: The N in ECC_N stands for *natural deduction*. The G in ECC_G has become rather common for the designation of sequent systems to commemorate *Gentzen*, even though he has actually introduced both kinds of system.

inition 3.2) which abandons the rules (app), (π_1) and (π_2) in favour of rules (ΠL) and (ΣL) that decompose Π- and Σ-abstractions on the left side, i.e. in the antecedent of a typing judgement. To compensate for the lack of a subterm property of rule ($\preceq$), a new rule having a similar effect in the antecedent of sequents is introduced; together with this new rule, ($\preceq$) is retained in ECC_G. Even though this does not seem to be an improvement, it can be shown (Section 4.2) that applications of these two rules can be split into a deterministic reduction to weak head normal form in the "interior" of the proof tree to be constructed and a higher-order unification taking into account the cumulativity relation at the leaves of the proof tree.

- It is easy to simulate derivations of ECC_G in the calculus ECC_N (Proposition 3.4). This result can be interpreted as a proof of "correctness" of ECC_G with respect to ECC_N.

- For showing "completeness", an auxiliary system ECC_{G+cut} is introduced, in which the cut rule, which was shown to be admissible in ECC_N, is explicitly added to ECC_G. Derivations in ECC_N can be simulated in ECC_{G+cut} (Proposition 3.5).

- By a process of *cut elimination* (Proposition 3.7), it is shown that applications of the cut rule in derivations of ECC_{G+cut} can be removed, thus yielding derivations in ECC_G. This establishes the equivalence of the three calculi. Cut elimination will not be proved in full generality, but only for a fragment in which terms become smaller under substitution for an appropriately defined measure.

Even the restricted fragment turns out to be too large to permit a well-focussed proof search, and too unwieldy because of the meager set Π, Σ of type constructors. Therefore, in Chapter 4, a calculus more appropriate for practical concerns will be developed.

In the following considerations, no reference will be made to metavariables. In particular, we will examine the relation between natural deduction calculi (ECC_N) and sequent calculi (ECC_G and ECC_{G+cut}) in a language without metavariables. The reason is that the metavariable rules of Figure 2.5 only permit to derive judgements of the form $\Gamma \vdash ?n^\frown\sigma : A$ for metavariables $?n$ and substitutions σ. However, metavariables are not a purpose in themselves, and we are not aiming at synthesizing metavariables as proof terms, even though proof terms arising during a proof may contain metavariables.

Metavariables will be reintroduced as devices for facilitating proof search in Chapter 4. The resulting framework will be hybrid in the sense that proof

search by decomposition of formulae will be carried out with a sequent calculus, whereas verification of type correctness of solutions is obtained via type checking solutions in a natural deduction calculus, in the style elaborated in Section 2.6.

3.1.2. Definition of the systems

In the calculus ECC_N, the following cut rule is admissible (see Proposition 2.45):

$$\frac{\Gamma, z : C, \Delta \vdash M : A \qquad \Gamma \vdash c : C}{\Gamma, \Delta\{z := c\} \vdash M\{z := c\} : A\{z := c\}}$$

In subsequent proofs, it will be practical to use a slightly extended version of this rule, which incorporates the cumulativity relation $\preceq$ and which expresses that elements c which are of a smaller type C (with respect to $\preceq$) can be substituted for parameters z of larger type C'.

Definition 3.1 (Cut Rule)

$$\frac{\Gamma, z : C', \Delta \vdash M : A \quad \Gamma \vdash c : C \quad C \preceq C'}{\Gamma, \Delta\{z := c\} \vdash M\{z := c\} : A\{z := c\}} \text{ (cut)}$$

$$\frac{\Gamma, z : C', \Delta \ \mathit{valid}_c \quad \Gamma \vdash c : C \quad C \preceq C'}{\Gamma, \Delta\{z := c\} \ \mathit{valid}_c} \text{ (cut)}$$

Let us formally define the systems referred to in the following:

Definition 3.2 (Systems ECC_N, ECC_G and ECC_{G+cut})
The system ECC_N is the same as the previously defined system ECC, given by the rules of Figure 2.2.

The system ECC_G differs from ECC_N as follows:

- The rules (app), (π_1) and (π_2) are removed.
- The following rules are added:

$$\frac{\begin{array}{c}\Gamma, p : \Pi x : A.\ B[x], \Gamma' \vdash N_1 : A \\ \Gamma, p : \Pi x : A.\ B[x], \Gamma', p' : B[N_1] \vdash N_2 : G\end{array}}{\Gamma, p : \Pi x : A.\ B[x], \Gamma' \vdash N_2\{p' := (p\ N_1)\} : G} \ (\Pi L)$$

$$\frac{\Gamma, p : \Sigma x : A.\ B[x], \Gamma', p_1 : A, p_2 : B[\pi_1(p)] \vdash N : G}{\Gamma, p : \Sigma x : A.\ B[x], \Gamma' \vdash N\{p_1 := \pi_1(p), p_2 := \pi_2(p)\} : G} \ (\Sigma L)$$

$$\frac{\Gamma, p : T', \Gamma' \vdash M : A \quad \Gamma \vdash_N T : \mathit{Type}_j \quad T \preceq T'}{\Gamma, p : T, \Gamma' \vdash M : A} \ (\preceq L)$$

$$\frac{}{\langle\rangle\ \mathit{valid}_c}\ (\text{Cempty})$$

$$\frac{\Gamma \vdash_N A : \mathit{Type}_j \quad x \notin FV(\Gamma)}{\Gamma, x : A\ \mathit{valid}_c}\ (\text{Cvalid})$$

$$\frac{\Gamma\ \mathit{valid}_c}{\Gamma \vdash \mathit{Prop} : \mathit{Type}_0}\ (\text{UProp}) \qquad \frac{\Gamma\ \mathit{valid}_c}{\Gamma \vdash \mathit{Type}_j : \mathit{Type}_{j+1}}\ (\text{UType})$$

$$\frac{\Gamma, x : A, \Gamma'\ \mathit{valid}_c}{\Gamma, x : A, \Gamma' \vdash x : A}\ (\text{var})$$

$$\frac{\Gamma \vdash A : \mathit{Type}_j \quad \Gamma, x : A \vdash P : \mathit{Prop}}{\Gamma \vdash \Pi x : A.P : \mathit{Prop}}\ (\Pi\text{-Form}_1)$$

$$\frac{\Gamma \vdash A : \mathit{Type}_j \quad \Gamma, x : A \vdash B : \mathit{Type}_j}{\Gamma \vdash \Pi x : A.B : \mathit{Type}_j}\ (\Pi\text{-Form}_2)$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda\, x : A.M : \Pi x : A.B}\ (\Pi R)$$

$$\frac{\begin{array}{c}\Gamma, p : \Pi x : A.\ B[x], \Gamma' \vdash N_1 : A \\ \Gamma, p : \Pi x : A.\ B[x], \Gamma', p' : B[N_1] \vdash N_2 : G\end{array}}{\Gamma, p : \Pi x : A.\ B[x], \Gamma' \vdash N_2\{p' := (p\ N_1)\} : G}\ (\Pi L)$$

$$\frac{\Gamma \vdash A : \mathit{Type}_j \quad \Gamma, x : A \vdash B : \mathit{Type}_j}{\Gamma \vdash \Sigma x : A.B : \mathit{Type}_j}\ (\Sigma\text{-Form})$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B\{x := M\} \quad \Gamma, x : A \vdash B : \mathit{Type}_j}{\Gamma \vdash \mathit{pair}_{\Sigma x : A.\ B}(M,\ N) : \Sigma x : A.\ B}\ (\Sigma R)$$

$$\frac{\Gamma, p : \Sigma x : A.\ B[x], \Gamma', p_1 : A, p_2 : B[\pi_1(p)] \vdash N : G}{\Gamma, p : \Sigma x : A.\ B[x], \Gamma' \vdash N\{p_1 := \pi_1(p), p_2 := \pi_2(p)\} : G}\ (\Sigma L)$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash_N A' : \mathit{Type}_j \quad A \preceq A'}{\Gamma \vdash M : A'}\ (\preceq R)$$

$$\frac{\Gamma, p : T', \Gamma' \vdash M : A \quad \Gamma \vdash_N T : \mathit{Type}_j \quad T \preceq T'}{\Gamma, p : T, \Gamma' \vdash M : A}\ (\preceq L)$$

Figure 3.1.: Rules of the calculus ECC_G

- The premiss $A : \mathit{Type}$ is added to rule (Π-Form$_1$).
- In the rules (Cvalid) and ($\preceq$), the precondition of the form "$\Gamma \vdash T : \mathit{Type}_j$" is changed to the form "$\Gamma \vdash_N T : \mathit{Type}_j$" (see remarks below).
- The rules (λ), (pair) and ($\preceq$) are renamed to (ΠR), (ΣR) and ($\preceq\ R$), respectively.
- All other rules remain unchanged.

The system ECC_G is summarized in Figure 3.1.

The system ECC_{G+cut} is defined as the system ECC_G together with the (cut) rule.

Each of the systems ECC_N, ECC_G and ECC_{G+cut} defines a different derivability relation $\vdash_N$, $\vdash_G$ and $\vdash_C$. For readability, we omit the index and simply use the symbol $\vdash$ whenever the system under consideration is clear.

Definition 3.3 (Principal Formula)
The formula to which (ΠL) resp. (ΣL) is applied is called the *principal formula* of the application. The premiss $\Gamma, p : \Pi x : A.\ B[x], \Gamma', p' : B[N_1] \vdash N_2 : G$ of (ΠL) is called its *main premiss*, the branch of a proof tree following the main premiss the *main branch*.

The adequacy of the rules introduced above will ultimately become apparent from the equivalence results for the systems ECC_N, ECC_G and ECC_{G+cut} proved in the following. The rules (ΠL) and (ΣL) have already been motivated above. The rule ($\preceq\ L$) can be justified by Lemma 2.8. In order to give an intuition for situations where an application of ($\preceq\ L$) is required, consider the following proof goal: It has to be shown that B holds in context $\Gamma \mathrel{\widehat{=}} a : A, h : (\lambda\, P : \mathit{Prop}.P \to B)\ A$. In the system ECC_N, this proof can be carried out as follows:

$$\dfrac{\dfrac{\Gamma \vdash h : (\lambda\, P : \mathit{Prop}.P \to B)\ A}{\Gamma \vdash h : A \to B}\ (\preceq) \qquad \Gamma \vdash a : A}{\Gamma \vdash (h\ a) : B}\ (\text{app})$$

A similar proof in the system ECC_G can only succeed with an application of ($\preceq\ L$), since no other rules are applicable to the initial goal:

$$\dfrac{\dfrac{a : A, h : A \to B \vdash a : A \quad a : A, h : A \to B, h' : B \vdash h' : B}{a : A, h : A \to B \vdash (h\ a) : B}\ (\Pi L)}{a : A, h : (\lambda\, P : \mathit{Prop}.P \to B)\ A \vdash (h\ a) : B}\ (\preceq\ L)$$

Hypotheses like $h : (\lambda\, P : \mathit{Prop}.P \to B)\ A$ which are not in normal form may seem to be artificial and therefore situations like the above easily avoidable by stipulating that hypotheses in a proof have to be normalized. However, redexes in the list of hypotheses may arise during proof search by an application of rule (ΠL), whenever x in $B[x]$ is applied to arguments and the term N_1 is a λ-abstraction (see Example 4.2 for an illustration).

Let us now comment on the judgement form "$\Gamma \vdash_N T : \mathit{Type}$" in the rules (Cvalid), ($\preceq R$) and ($\preceq L$): For applying the rules (Cvalid), ($\preceq R$) and ($\preceq L$), it has to be ensured that A, A' resp. T are types. Assume these rules had been stated with the judgement form $\Gamma \vdash_G T : \mathit{Type}$ instead of $\Gamma \vdash_N T : \mathit{Type}$. Then, an easy inductive argument shows that whenever $\Gamma \vdash_G M : P$ is derivable, M is in normal form (in particular, the substitutions $p' := (p\ N_1)$, $p_1 := \pi_1(p)$, $p_2 := \pi_2(p)$ do not introduce redexes since p is a variable). This would have the following consequences:

- Whenever a context contains a declaration $x : A$ with A not in normal form, this context cannot be shown to be valid, because, by rule (Cvalid), this would require a derivation of $\Gamma \vdash_G A : \mathit{Type}$. As remarked above, contexts with non-normalized terms arise naturally in proof search, so proof search would lead to invalid contexts.

- It is not possible to derive $\Gamma \vdash_G A' : \mathit{Type}$ or $\Gamma \vdash_G T : \mathit{Type}$ whenever A' or T are not in normal form, so backwards applications of rules ($\preceq R$) and ($\preceq L$) will be blocked in these cases, leading to an unnecessary loss of completeness. However, since the types A' resp. T are known when the rules ($\preceq R$) and ($\preceq L$) are applied in proof search, there is no need to use the proof search calculus ECC_G to check these side conditions.

Altogether, this discussion suggests that the test for being a type is better not performed in the calculus ECC_G, but rather in the calculus ECC_N. By inspection of the rules in Figure 3.1, it is obvious that the judgement $\Gamma\ \mathit{valid}_c$ is then derivable in ECC_G iff it is derivable in ECC_N. All this makes the calculus ECC_G hybrid in the sense that ECC_G-derivations "invoke" subderivations in ECC_N. In proof search, the test for being a valid type in a context or a valid context is however not made explicit: A more detailed analysis of ECC_G-derivations in Section 4.2 will show that these conditions are either implicitly ensured (in applications of ($\preceq R$) and ($\preceq L$) that reduce types to weak head normal form) or will be imposed by the well-typedness conditions that have been developed for metavariables in Section 2.6.

3.2. Properties of Sequent Systems

3.2.1. Correctness

Proposition 3.4 (Derivability in ECC_G implies derivability in ECC_N) If a judgement $\Gamma \vdash M : A$ (or $\Gamma\ \mathit{valid}_c$) is derivable in system ECC_G, then it is derivable in system ECC_N.

Proof: By induction on the structure of the derivation in ECC_G. We will first show that the rules (ΠL) and (ΣL) can be simulated by applications of rules of ECC_N:

- Rule (ΠL): Given the following derivations $\mathcal{D}_1$ and $\mathcal{D}_2$:

$$\frac{\Gamma, p : \Pi x : A.\ B[x], \Gamma', p' : B[N_1] \vdash N_2 : G}{\Gamma, p : \Pi x : A.\ B[x], \Gamma' \vdash \lambda\, p' : B[N_1].\ N_2 : \Pi p' : B[N_1].\ G}\ (\lambda)$$

$$\frac{\Gamma, p : \Pi x : A.\ B[x], \Gamma' \vdash p : \Pi x : A.\ B[x] \quad \Gamma, p : \Pi x : A.\ B[x], \Gamma' \vdash N_1 : A}{\Gamma, p : \Pi x : A.\ B[x], \Gamma' \vdash (p\ N_1) : B[N_1]}\ (\text{app})$$

  Form the following derivation (note that p' does not occur in G):

$$\frac{\mathcal{D}_1 \qquad \mathcal{D}_2}{\Gamma, p : \Pi x : A.\ B[x], \Gamma' \vdash (\lambda\, p' : B[N_1].\ N_2)\ (p\ N_1)\ : G}\ (\text{app})$$

  Then use the subject reduction property of ECC_N to obtain the judgement

$$\Gamma, p : \Pi x : A.\ B[x], \Gamma' \vdash N_2\{p' := (p\ N_1)\} : G$$

- Rule (ΣL): From the judgement

$$\Gamma, p : \Sigma x : A.\ B[x], \Gamma', p_1 : A, p_2 : B[\pi_1(p)] \vdash N : G$$

  derive

$$\lambda\, p_1 : A.\ \lambda\, p_2 : B[\pi_1(p)].\ N\ :\ \Pi p_1 : A.\ \Pi p_2 : B[\pi_1(p)].\ G$$

  in context $\Gamma, p : \Sigma x : A.\ B[x], \Gamma'$ by two applications of rule (λ). Then, in this context, derive $\pi_1(p) : A$ and $\pi_2(p) : B[\pi_1(p)]$ with rules (π_1) and (π_2). Twofold application of rule (app) and subject reduction gives the desired result

$$\Gamma, p : \Sigma x : A.\ B[x], \Gamma' \vdash N\{p_1 := \pi_1(p), p_2 := \pi_2(p)\} : G$$

Thus, (ΠL) and (ΣL) can be conceived as derived rules of the calculus ECC_N. By Lemma 2.8, the rule $(\preceq L)$ is admissible in ECC_N and so does not permit the derivation of new judgements, and, clearly, neither does adding a precondition to rule (Π-Form$_1$). □

3.2.2. Completeness

As mentioned in the introduction, completeness will be proved in two steps: First by showing that derivability in ECC_N implies derivability in ECC_{G+cut}, and then, by a cut elimination proof, that ECC_{G+cut} and ECC_G are equivalent.

Proposition 3.5 (ECC_N implies ECC_{G+cut})
If a judgement $\Gamma \vdash M : A$ (or Γ *valid*$_c$) is derivable in system ECC_N, then it is derivable in system ECC_{G+cut}.

Proof: By induction on the structure of the derivation in ECC_N. Since the system ECC_N only differs from ECC_{G+cut} in the rules (app), (π_1) and (π_2) and the strengthened premiss of (Π-Form$_1$), only these rules have to be considered:

- Assume the last rule of the derivation is (app):

$$\frac{\begin{array}{cc}\mathcal{D}_0 & \mathcal{D}_1\\ \Gamma \vdash M : \Pi x : A.B[x] & \Gamma \vdash N : A\end{array}}{\Gamma \vdash M\ N : B[N]}\ (\text{app})$$

  By induction hypothesis, there are derivations $\mathcal{D}_0^*$ and $\mathcal{D}_1^*$ in ECC_{G+cut} corresponding to $\mathcal{D}_0$ and $\mathcal{D}_1$. Weaken the derivation $\mathcal{D}_1^*$ to derivation $\mathcal{D}_1'$ by introducing the assumption $p : \Pi x : A.\ B[x]$ in the antecedents of $\mathcal{D}_1^*$. Call the following derivation $\mathcal{D}_2$:

$$\frac{\begin{array}{cc}\mathcal{D}_1' & \\ \Gamma, p : \Pi x : A.\ B[x] \vdash N : A & \Gamma, p : \Pi x : A.\ B[x], p' : B[N] \vdash p' : B[N]\end{array}}{\Gamma, p : \Pi x : A.\ B[x] \vdash p\ N : B[N]}\ (\Pi L)$$

  Then, form the derivation:

$$\frac{\begin{array}{cc}\mathcal{D}_2 & \Gamma \vdash M : \Pi x : A.\ B[x]\end{array}}{\Gamma \vdash M\ N : B[N]}\ (\text{cut})$$

- Assume the last rule is (π_1):

$$\frac{\begin{array}{c}\mathcal{D}\\ \Gamma \vdash M : \Sigma x : A.B[x]\end{array}}{\Gamma \vdash \pi_1(M) : A}\ (\pi_1)$$

Then form the ECC_{G+cut}-derivation:

$$\frac{\mathcal{D}_1 \quad \Gamma \vdash M : \overset{\mathcal{D}^*}{\Sigma x : A.\ B[x]}}{\Gamma \vdash \pi_1(M) : A}\ (\text{cut})$$

with $\mathcal{D}_1$ defined as:

$$\frac{\Gamma, p : \Sigma x : A.\ B[x], p_1 : A, p_2 : B[\pi_1(p)] \vdash p_1 : A}{\Gamma, p : \Sigma x : A.\ B[x] \vdash \pi_1(p) : A}\ (\Sigma L)$$

- Assume the last rule is (π_2):

$$\frac{\Gamma \vdash M : \overset{\mathcal{D}}{\Sigma x : A.B}}{\Gamma \vdash \pi_2(M) : B[\pi_1(M)]}\ (\pi_2)$$

Then form the ECC_{G+cut}-derivation:

$$\frac{\mathcal{D}_1 \quad \Gamma \vdash M : \overset{\mathcal{D}^*}{\Sigma x : A.\ B[x]}}{\Gamma \vdash \pi_2(M) : B[\pi_1(M)]}\ (\text{cut})$$

with $\mathcal{D}_1$ defined as:

$$\frac{\Gamma, p : \Sigma x : A.\ B[x], p_1 : A, p_2 : B[\pi_1(p)] \vdash p_2 : B[\pi_1(p)]}{\Gamma, p : \Sigma x : A.\ B[x] \vdash \pi_2(p) : B[\pi_1(p)]}\ (\Sigma L)$$

- Assume the last rule is (Π-Form$_1$) of calculus ECC_N. The premiss $\Gamma, x : A \vdash P : \mathit{Prop}$ is derivable in ECC_N if and only if $\Gamma \vdash A : \mathit{Type}_j$ and $\Gamma, x : A \vdash P : \mathit{Prop}$ are both derivable in ECC_N, because the latter derivation contains $\Gamma \vdash A : \mathit{Type}_j$ as a subderivation. Therefore, by induction hypothesis, both premisses of rule (Π-Form$_1$) of calculus ECC_{G+cut} are derivable.

□

Cut Elimination

As announced above, we will restrict cut elimination to a fragment in which "terms become smaller under substitution", for an appropriate measure on terms. The reason for this restriction is a technicality in the proof of cut elimination as presented below. For being able to apply an induction hypothesis,

terms of the form $B\{x := M\}$ have to be smaller than terms of the form $\Pi x : A.\ B$ and terms of the form $\Sigma x : A.\ B$, where M is a term of type A. Possibly, a completely different setup of the proof would permit to eliminate this requirement, see the discussion in Section 1.3.2. In any case, there are certain choices of M, A and B that violate this constraint, take for example $M \mathrel{\widehat{=}} \Pi X : \mathit{Prop}.\ X \to X$, $A \mathrel{\widehat{=}} \mathit{Prop}$ and $\Pi x : A.\ B \mathrel{\widehat{=}} \Pi X : \mathit{Prop}.\ X \to X$. Then $B\{X := M\} \equiv (\Pi X : \mathit{Prop}.\ X \to X) \to (\Pi X : \mathit{Prop}.\ X \to X)$, which is greater than $\Pi X : \mathit{Prop}.\ X \to X$ for any reasonable term order, i.e. a term order respecting subterms.

Let us now define properties of a measure function on terms which are required to let the cut elimination proof go through.

Definition 3.6 (Measure for cut elimination)
A measure m mapping terms to a well-founded total order $<$ is a *term measure for cut elimination* if it fulfills the following requirements:

- If t' and t are well-typed terms (w.r.t. ECC_N) in normal form and t' is a proper subterm of t, then $m(t') < m(t)$.

- For all contexts Γ and terms M, B and A: whenever $\Gamma \vdash M : A$ is derivable in ECC_N, $B\{x := M\}$ and $\Pi x : A.\ B$ are well-typed terms in normal form, then $m(B\{x := M\}) < m(\Pi x : A.\ B)$. Similarly, $m(B\{x := M\}) < m(\Sigma x : A.\ B)$.

- If t' and t are well-typed terms (w.r.t. ECC_N) *not* in normal form, then $m(t') < m(t)$ if $m(nf(t')) < m(nf(t))$ for the respective normal forms.

It is intuitively clear that such a measure can be defined for a simple first-order language. A quantified formula like $\forall\, x : A.B$ is then greater than a formula $B\{x := M\}$ having less quantifiers (since M does not contain quantifiers), even though the term size $\mid B\{x := M\} \mid$ may be greater than the term size $\mid \forall\, x : A.B \mid$. This statement will be made more precise in Section 3.3 and generalized to all predicative subsystems of CC.

Apart from the restriction to a fragment for which a measure for cut elimination can be defined, some care has to be taken as to how an "equivalence" between derivations in ECC_{G+cut} and ECC_G can be stated: As observed at the end of Section 3.1.2, whenever $\Gamma \vdash M' : A'$ is derivable in ECC_G and M' is not a type, then M' is at least in weak head normal form. However, a derivation of $\Gamma \vdash M : A$ in ECC_{G+cut} can produce a topmost redex in M, for example if the cut term is a λ-abstraction and the variable for which it is substituted occurs in the functional position of an application. Thus, ECC_G and ECC_{G+cut} do not

derive exactly the same judgements, but only judgements equal up to conversion. Since ECC_G only serves as a calculus for proof search and is not intended to be used for type checking, this restriction is of no practical significance.

Proposition 3.7 (Cut Elimination)
Let $\mathcal{F}$ be a fragment in which a term measure for cut elimination can be defined. If the judgement $\Gamma \vdash M : A$ is derivable in the system ECC_{G+cut} (restricted to $\mathcal{F}$), then $\Gamma \vdash M' : A$ is derivable in ECC_G (restricted to $\mathcal{F}$), where M' is a term convertible to M.

Proof: The proof is by induction on derivations, using a measure whose main ingredients are the *rank* and the *level* of an applications of (cut):

- The *rank* of an application of (cut) is $m(C)$, where m is the measure over which the cut elimination theorem is parameterized and C is the formula to which (cut) is applied.
- The *level* of an application of the cut rule is the sum of the depths of the deductions of its premisses.

For a given application of (cut) in a derivation, the derivations leading to its premisses are examined. It is then shown that the application of (cut) can either be removed altogether, or can be moved further upwards (thus decreasing its level) or can be replaced by cuts of lower rank and/or level. The proof is carried out in detail in Appendix A.2. □

With respect to the original proof by Gentzen (and variants), this proof is complicated considerably by two factors, apart from the more complex measure: Firstly, proof terms have to be taken into account. More seriously, formulae in the antecedent (context) are labeled by variables, variables may occur in subsequent formulae. This requires more attention when applying operations such as weakening and contraction, and some of the standard structural rules (like exchange of formulae) are not valid.

3.3. A Measure for Cut Elimination

3.3.1. The Lambda-Cube

The proof of cut elimination in Section 3.2.2 is parameterized by a measure which has been called "term measure for cut elimination". In that section, an example is given which shows that no such measure can be defined for the full Extended Calculus of Constructions ECC. In order to assess the strength of

the cut elimination proof, we will in the following examine subsystems of the Calculus of Constructions CC. A classification of such subsystems is given by the λ-cube. The analysis of these subsystems will reveal that a measure for cut elimination can be defined for all predicative systems of the cube, whereas the counterexample of Section 3.2.2 can be reproduced for all impredicative systems of the cube. Consequently, completeness of proof search extends to all predicative systems of the λ-cube.

The discussion in this section takes as basis the Calculus of Constructions CC and not its 'extended' version ECC, because it permits a cleaner identification of subsystems. However, as we will see below, the difference between CC and ECC is not essential for the question under consideration, and a similar analysis could be performed for the calculus ECC.

The material in this Section 3.3.1, which has mostly been taken from the survey article [Bar92], is not new and is thought to provide the necessary background for an understanding of the following sections.

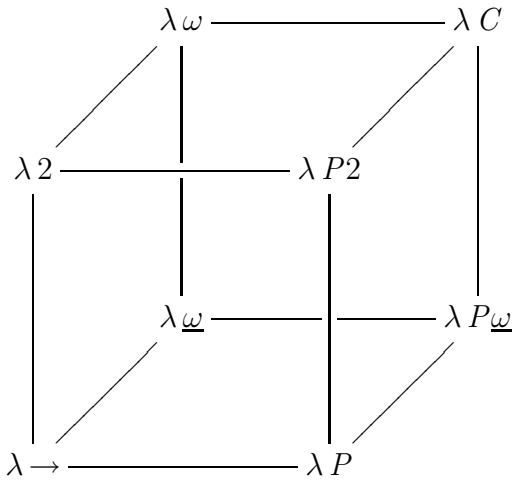


Figure 3.2.: The Lambda Cube

The λ-cube has been introduced by Barendregt to elucidate the fine structure of the Calculus of Constructions. The systems of the cube are often referred to as "Pure Type Systems" (PTS), a denotation which we occasionally use in the following. The corners of the cube (Figure 3.2) are marked with subsystems of the Calculus of Constructions. The inclusion relation between systems is depicted by an edge leading to the right, upwards or backwards. Some of the systems are artificial in the sense that they have been defined with the λ-cube in mind. However, most systems historically predate the Calculus of Constructions and have been known under different names than the ones

given in the cube, for example:

- $\lambda\!\rightarrow$ is the simply typed λ-calculus as defined by Church.
- $\lambda 2$ is Girard's System F, $\lambda\omega$ Girard's System Fω [Gir72].
- λP is the system LF as proposed in [HHP87]. This also corresponds to the logic used in a version of the Automath system [dB70].
- λC is the Calculus of Constructions CC which is a synthesis of all the systems of the λ-cube.

$$\langle\rangle \vdash \mathsf{Prop} : \mathsf{Type}$$

$$\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}\ (\text{var})$$

$$\frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B}\ (\text{weak})$$

$$\frac{\Gamma \vdash M : \Pi x : A.B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B\{x := N\}}\ (\text{app})$$

$$\frac{\Gamma, x : A \vdash M : B \quad \Gamma \vdash \Pi x : A.\ B : s}{\Gamma \vdash \lambda x : A.M : \Pi x : A.B}\ (\lambda)$$

$$\frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s \quad B \simeq B'}{\Gamma \vdash A : B'}\ (\text{conv})$$

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x : A.B : s_2}\ (\Pi)$$

Figure 3.3.: Typing rules for the systems of the Lambda Cube

The systems of the λ-cube are generated by a scheme of typing rules, depicted in Figure 3.3. The systems of the cube have all the rules except for (Π) in common. The systems differ in that the pair of sort variables (s_1, s_2) in the (Π) rule may only be instantiated with the sorts Prop or Type according to the sort combinations specified in Figure 3.4.

The systems of the upper plane of the λ-cube (those in which the sort pair $(\mathsf{Type}, \mathsf{Prop})$ is admissible) are *impredicative*: A type can be formed by quantifying over a domain to which the type under consideration itself belongs. For

$\lambda\!\rightarrow$	(Prop, Prop)			
$\lambda 2$	(Prop, Prop)	(Type, Prop)		
λP	(Prop, Prop)		(Prop, Type)	
$\lambda P2$	(Prop, Prop)	(Type, Prop)	(Prop, Type)	
$\lambda \underline{\omega}$	(Prop, Prop)			(Type, Type)
$\lambda \omega$	(Prop, Prop)	(Type, Prop)		(Type, Type)
$\lambda P\underline{\omega}$	(Prop, Prop)		(Prop, Type)	(Type, Type)
λC	(Prop, Prop)	(Type, Prop)	(Prop, Type)	(Type, Type)

Figure 3.4.: Sort combinations for the systems of the Lambda Cube

example, one such type is the term $\Pi X : \mathsf{Prop}.\ X \rightarrow X$, whose typing derivation is:

$$\dfrac{\vdash \mathsf{Prop} : \mathsf{Type} \qquad \dfrac{X : \mathsf{Prop} \vdash X : \mathsf{Prop} \quad X : \mathsf{Prop}, x : X \vdash X : \mathsf{Prop}}{X : \mathsf{Prop} \vdash X \rightarrow X : \mathsf{Prop}}\ (\Pi)}{\vdash \Pi X : \mathsf{Prop}.\ X \rightarrow X : \mathsf{Prop}}\ (\Pi)$$

The lower inference uses an "impredicative" application of the (Π) rule, with combination (Type, Prop), the upper inference a "predicative" application with combination (Prop, Prop).

Formally, the rules of Figure 3.3 are not a subset of the rules of *ECC* on which previous considerations were based. In particular, the weakening rule of the λ-cube systems is not made explicit in the rules of *ECC*, but can be shown to be admissible. The most notable difference is the use of type universes: In *ECC*, there is a hierarchy of universes Prop, Type_0, $\mathit{Type}_1, \ldots$, where $\vdash \mathit{Prop} : \mathit{Type}_i$ and $\vdash \mathit{Type}_i : \mathit{Type}_{i+1}$. In the λ-cube, there are only two universes (traditionally called *sorts* and denoted by s, s_1, s_2 in the rules of Figure 3.3) Prop and Type, where $\vdash \mathsf{Prop} : \mathsf{Type}$ and Type is not a typeable term. Since there is no cumulativity between sorts (i.e., a type A of Prop is not also in Type), the rule (conv) directly corresponds to $(\preceq)$ in *ECC*. Altogether, the system *ECC* and the systems of the λ-cube are sufficiently close to justify a comparison. In particular, the cut elimination proof of Section 3.2.2 carries over with only minor modifications.

In the same way that the systems of the λ-cube are fragments of the Calculus of Constructions *CC*, it is possible to define fragments of the calculus *ECC* by replacing the rules (Π-Form$_1$) and (Π-Form$_2$) of *ECC* by the rule scheme (Π) with its corresponding sort constraints. In this sense, impredicative subsystems of ECC_G (resp. ECC_{G+cut}) are the counterparts of impredicative subsystems of

the Calculus of Constructions. By taking into account the results established in the following, Proposition 3.7 can be interpreted as saying that corresponding predicative subsystems of ECC_G resp. ECC_{G+cut} are equivalent.

3.3.2. Facts about Pure Type Systems

In the following, some facts and lemmas about *predicative* Pure Type Systems will be proved. The purpose of this section is to show that, for predicative PTSs, a measure for cut elimination can indeed be defined. Note that some of the properties do not also hold in impredicative systems.

Lemma 3.8 (Facts about predicative PTSs)
The following facts hold in predicative PTSs, considering terms in normal form:

1. $\Gamma \not\vdash \mathsf{Type} : T$ for all Γ, T.
2. $\Pi x : \mathsf{Type}.\ B$ is not typeable for any term B (follows from rules (Π) and (var)).
3. If $\Gamma \vdash T : \mathsf{Type}$, then T has the form $\Pi x_1 : C_1 \ldots x_n : C_n.\mathsf{Prop}$, where each C_i is of type Prop or Type (follows from rule (Π)).
4. If $\Gamma \vdash T : \mathsf{Prop}$, then T has one of the following forms (follows from induction over structure of T):
   - T is a variable
   - T is of the form $(f\ a)$ with $f : \Pi x : A.\ \mathsf{Prop}$ and $A : \mathsf{Prop}$ and $a : A$.
   - T is of the form $\Pi x_1 : C_1 \ldots x_n : C_n.B$, with $C_i : \mathsf{Prop}$ and $B : \mathsf{Prop}$.

The notion of "level" of a term can be found under various guises in the literature. We define it as follows:

Definition 3.9 (Level)
Assume that t is a term which is well-typed in a context Γ. The level $level_\Gamma(t)$ of t is defined inductively on the structure of t as follows:

- $level_\Gamma(t) = 3$ if $t \equiv \mathsf{Type}$.
- $level_\Gamma(t) = 2$ if $\Gamma \vdash t : \mathsf{Type}$.
- $level_\Gamma(t) = 1$ if there exist a T such that $\Gamma \vdash t : T$ and $\Gamma \vdash T : \mathsf{Type}$.
- $level_\Gamma(t) = 0$ if there exist a T such that $\Gamma \vdash t : T$ and $\Gamma \vdash T : \mathsf{Prop}$.

The index Γ is usually omitted from $level_\Gamma$.

The function *level* is well-defined. Note, in particular, that in the λ-cube, there is no such "inclusion" as $\mathsf{Prop} \subset \mathsf{Type}$. Thus, we do not have $T : \mathsf{Prop}$ and $T : \mathsf{Type}$ at the same time.

A term of level 2 is usually called a *kind*, a term of level 1 a *constructor* (if it is of functional type) or a *type*, and a term of level 0 an *object*.

The Π-depth counts the number of Π-abstractions of a term:

Definition 3.10 (Π-depth)
The Π-depth $depth(t)$ of term t is defined by:

- $depth(t) = 0$ for all term constructors except for Π
- $depth(\Pi x : A.\ B) := depth(B) + 1$

Definition 3.11 (PTS Measure)
Assume that t is a well-typed term. The PTS measure $m(t)$ yields a multiset of pairs (l, d), where l and d are natural numbers. For terms t in normal form, $m(t)$ is defined inductively on the structure of t as follows:

- $m(t) = \{(level(t), 0)\}$ for all term constructors except for Π-abstractions
- $m(\Pi x : A.\ B) = \{(level(\Pi x : A.\ B), depth(\Pi x : A.\ B))\} \cup m(A) \cup m(B)$[2]

For t not in normal form, $m(t)$ is defined as $m(nf(t))$, the result of computing the measure for the normal form of t.

A term t_1 is smaller than term t_2 with respect to the PTS measure (written as $t_1 < t_2$) if $m(t_1)$ is less than $m(t_2)$ with respect to the multiset order induced by a lexicographic ordering of the pairs (l, d).

Lemma 3.12
Let $\Gamma \vdash T : \mathsf{Type}$. Then for any term M, $depth(T\{x := M\}) = depth(T)$.

Proof: By Lemma 3.8, T has the form $\Pi x_1 : C_1 \ldots x_n : C_n.\mathsf{Prop}$. Then $T\{x := M\}$ has the form $\Pi x_1 : C_1\{x := M\} \ldots x_n : C_n\{x := M\}.\mathsf{Prop}$. Both terms have depth n. □

Lemma 3.13

1. For all typecorrect terms t, M, if $\{x := M\}$ is a type-respecting substitution (i.e., x has the same type as M), then $level(t) = level(t\{x := M\})$.
2. For all typecorrect terms t_1, t_2: If t_1 β-reduces to t_2, then $level(t_1) = level(t_2)$.

[2]Here and in the following, $\cup$ stands for the union of multisets

Proof:

1. By induction on the structure of t.
2. By induction on the length of the reduction, using the subject reduction property of PTSs.

□

In the following, we will restrict the discussion to type-correct terms.

Lemma 3.14

1. For all applications $(f\ a)$, either $level(f\ a) = 0$ or $level(f\ a) = 1$.
2. If $level(f\ a) = 1$, then $level(f) = 1$.

Proof:

1. $level(t) = 3$ implies $t = \mathsf{Type}$, $level(t) = 2$ implies that t is of the form $\Pi x_1 : T_1, \ldots, x_n : T_n.\mathsf{Prop}$ (by Lemma 3.8).
2. Assume $level(f) = 0$. Then there exist context Γ and types T_1, T_2 such that[3] $\Gamma \vdash f : (\Pi x : T_1.\ T_2) : \mathsf{Prop}$. In predicative systems, T_1 and T_2 then even have to be of type Prop. Then, for any a such that $\Gamma \vdash a : T_1$, $\Gamma \vdash (f\ a) : T_2 : \mathsf{Prop}$, thus $level(f\ a) = 0$. The statement follows with 1.

□

The following lemma gives bounds for $m(T)$ for *type terms* T, that is, terms T for which there exists a context Γ such that $\Gamma \vdash T : \mathsf{Type}$ or $\Gamma \vdash T : \mathsf{Prop}$.

Lemma 3.15

1. For all type terms T: If $level(T) = n$, then $\{(n, 0)\} \leq m(T)$
2. For all type terms T: If $level(T) = n$, then $m(T) < \{(n + 1, 0)\}$
3. For all terms M, A: If $\Gamma \vdash M : A$, then $m(M) < m(A)$.

Proof: 1. and 2.: By induction on the structure of T.
3.: If $\Gamma \vdash M : A : \mathsf{Type}$, then $level(M) = 1$ and $level(A) = 2$, thus $m(M) < \{(2, 0)\} \leq m(A)$. Analogously for $\Gamma \vdash M : A : \mathsf{Prop}$. □

Note that if m respects subterm structure, the third condition fails for impredicative systems – take for example $M \equiv \Pi X : \mathsf{Prop}.\ X$ and $A \equiv \mathsf{Prop}$. For this choice of M and A, we have $\vdash M : A$, so by Lemma 3.15, $m(M) < m(A)$. However, since A is a type subterm of M, we would also have $m(A) < m(M)$.

[3] Here and in the following, $\Gamma \vdash M : A : s$ abbreviates $\Gamma \vdash M : A$ and $\Gamma \vdash A : s$

Lemma 3.16 (Measure under substitution)
Assume $\Gamma \vdash M : A$ and $\Gamma \vdash A : s$, with $s \in \{\mathsf{Prop}, \mathsf{Type}\}$. Assume that T is Type or a type term which is well-typed in Γ and possibly has free occurrences of a variable x of type A. Then:

- $m(T\{x := M\}) \leq m(T)$ or $m(T\{x := M\}) < m(A)$
- If $T \equiv \mathsf{Type}$ or $\Gamma \vdash T : \mathsf{Type}$, then even $m(T\{x := M\}) \leq m(T)$ alone holds.
- If $\Gamma \vdash T : \mathsf{Prop}$ and $\Gamma \vdash M : A : \mathsf{Prop}$, then even $m(T\{x := M\}) \leq m(T)$ alone holds.

Proof: By induction on the structure of T.

- The cases $T \equiv \mathsf{Type}$ or $T \equiv \mathsf{Prop}$ are obvious.
- Assume T is a variable y.
  - If $y \neq x$, then $m(y\{x := M\}) = m(y)$.
  - If $y = x$, then $m(y\{x := M\}) = m(M) < m(A)$ by Lemma 3.15

  Note in particular that there are no variables of Type (which handles the second statement of the lemma) and that if $y : \mathsf{Prop}$ and also $M : A : \mathsf{Prop}$ (and consequently also $x : A : \mathsf{Prop}$), then $y \neq x$ and therefore $m(T\{x := M\}) \leq m(T)$ (third statement of lemma).
- Assume T is of the form $(f\ a)$. By Lemma 3.14, we only have to examine the cases that $\mathit{level}(f\ a) = 0$ or $\mathit{level}(f\ a) = 1$, and since T is assumed to be a type term, only the case $\mathit{level}(f\ a) = 1$ remains.
  - Assume $\mathit{level}(f\ a) = 1$ and $\mathit{level}(M) \geq 1$. Then, by Lemma 3.13, also $\mathit{level}((f\ a)\{x := M\}) = 1$. Since $\mathit{level}(A) \geq 2$, we have $m((f\ a)\{x := M\}) < m(A)$.
  - Assume $\mathit{level}(f\ a) = 1$ and $\mathit{level}(M) = 0$. Since the term $(f\ a)$ is assumed to be in normal form, it can be written as $(h\ a_1 \ldots a_n\ a)$, where h is a variable. By Lemma 3.14, $\mathit{level}(h) = 1$. In particular, h is not x, since $\mathit{level}(x) = \mathit{level}(M) = 0$. The normal form of $(f\ a)\{x := M\}$ then has the form $(h\ a'_1 \ldots a'_n\ a')$, where $a'_1, \ldots a'_n, a'$ are the normal forms of $a_1\{x := M\}, \ldots a_n\{x := M\}, a\{x := M\}$. Thus, $m((f\ a)\{x := M\}) = \{(1, 0)\} = m(f\ a)$.
- Assume $T \equiv \Pi z : T_1.\ T_2$.

– If $T_1 : \mathsf{Type}$ and $T_2 : \mathsf{Type}$, then $(\Pi z : T_1.\ T_2) : \mathsf{Type}$.
By induction hypothesis, $m(T_1\{x := M\}) \leq m(T_1)$ and $m(T_2\{x := M\}) \leq m(T_2)$. Furthermore, by Lemma 3.12, $\mathit{depth}(\Pi z : T_1.\ T_2) = \mathit{depth}((\Pi z : T_1.\ T_2)\{x := M\})$, thus

$$\begin{array}{ll} & m((\Pi z : T_1.\ T_2)\{x := M\}) \\ = & \{(2, \mathit{depth}((\Pi z : T_1.\ T_2)\{x := M\}))\} \\ & \cup m(T_1\{x := M\}) \cup m(T_2\{x := M\}) \\ \leq & \{(2, \mathit{depth}(\Pi z : T_1.\ T_2))\} \cup m(T_1) \cup m(T_2) \\ = & m(\Pi z : T_1.\ T_2) \end{array}$$

– If $T_1 : \mathsf{Prop}$ and $T_2 : \mathsf{Type}$, then $(\Pi z : T_1.\ T_2) : \mathsf{Type}$.
By induction hypothesis, $m(T_2\{x := M\}) \leq m(T_2)$. Furthermore, $\mathit{level}(T_1) < 2$ and by Lemma 3.13, $\mathit{level}(T_1\{x := M\}) < 2$, thus according to Lemma 3.15, $m(T_1\{x := M\}) < \{(2, 0)\} \leq \{(2, \mathit{depth}(\Pi z : T_1.\ T_2))\}$. By Lemma 3.12, $\mathit{depth}(\Pi z : T_1.\ T_2) = \mathit{depth}((\Pi z : T_1.\ T_2)\{x := M\})$. Altogether:

$$\begin{array}{ll} & m((\Pi z : T_1.\ T_2)\{x := M\}) \\ = & \{(2, \mathit{depth}((\Pi z : T_1.\ T_2)\{x := M\}))\} \\ & \cup m(T_1\{x := M\}) \cup m(T_2\{x := M\}) \\ \leq & \{(2, \mathit{depth}(\Pi z : T_1.\ T_2))\} \cup m(T_2) \\ \leq & m(\Pi z : T_1.\ T_2) \end{array}$$

– If $T_1 : \mathsf{Prop}$ and $T_2 : \mathsf{Prop}$, then $(\Pi z : T_1.\ T_2) : \mathsf{Prop}$.
Either, $M : A : \mathsf{Type}$ or $M : A : \mathsf{Prop}$.
If $M : A : \mathsf{Type}$, then, since

$$\mathit{level}(\Pi z : T_1.\ T_2) = \mathit{level}((\Pi z : T_1.\ T_2)\{x := M\}) = 1$$

but $\mathit{level}(A) = 2$, we have by Lemma 3.15 that

$$m((\Pi z : T_1.\ T_2)\{x := M\}) < \{(2, 0)\} \leq m(A)$$

If $M : A : \mathsf{Prop}$, then $m(T_1\{x := M\}) \leq m(T_1)$ and $m(T_2\{x := M\}) \leq m(T_2)$ and $\mathit{depth}(\Pi z : T_1.\ T_2) = \mathit{depth}((\Pi z : T_1.\ T_2)\{x := M\})$, thus

$$\begin{array}{ll} & m((\Pi z : T_1.\ T_2)\{x := M\}) \\ = & \{(1, \mathit{depth}((\Pi z : T_1.\ T_2)\{x := M\}))\} \\ & \cup m(T_1\{x := M\}) \cup m(T_2\{x := M\}) \\ \leq & \{(1, \mathit{depth}(\Pi z : T_1.\ T_2))\} \cup m(T_1) \cup m(T_2) \\ = & m(\Pi z : T_1.\ T_2) \end{array}$$

□

Proposition 3.17
For the measure m defined in Definition 3.11, the following holds:

1. If T and T' are well-typed type terms and T' is a type subterm (cf. Definition 2.2) of T, then $m(T') < m(T)$.

2. If $\Gamma \vdash M : A$ is derivable, M and A are in normal form, and $\Pi x : A.\ B$ is a term in normal form and well-typed in Γ, then $m(B\{x := M\}) < m(\Pi x : A.\ B)$.

Therefore, m is a term measure for cut elimination.

Proof:

1. Obvious from the definition of m.

2. By definition of m,

   $$m(\Pi x : A.\ B) = \{(\mathit{level}(\Pi x : A.\ B), \mathit{depth}(\Pi x : A.\ B))\} \cup m(A) \cup m(B)$$

   By 1. and Lemma 3.16, we have:

   - $m(B\{x := M\}) \leq m(B) < m(\Pi x : A.\ B)$, or
   - $m(B\{x := M\}) < m(A) < m(\Pi x : A.\ B)$. □

4. Methods of Proof Search

4.1. Introduction

This chapter ties together the results developed in the two previous chapters, by showing how the mechanisms for securely manipulating metavariables, developed in Chapter 2, can be combined with the sequent calculus of Chapter 3 to produce practically useful proof search algorithms.

The idea of using some kind of "metavariable" to postpone construction of unknown terms is not new. However, it will be demonstrated in the following how the notion of well-typed instantiation (Definition 2.71) ensures the usual side-conditions of sequent calculi (see in particular Section 4.4.2). Conversely, the typing rules for a calculus with metavariables (Section 2.5) yield a criterion for showing that the proof terms built up during proof search are correct by construction, thus no *a posteriori* type checking of proof terms is necessary to ascertain that "nothing went wrong". By this means, proof terms can be understood as a conceptual device for establishing soundness of proof search, but their construction is often not actually required.

In the following, we will present two examples, one of them rather simple, the other more involved, which, for one part, give a general idea of how proof search proceeds, and which, for another part, demonstrate how the behaviour of proof search is influenced by the choice of rules of the calculus ECC_G that have to be applied. This dependency of the "complexity" (in a non-technical sense) of proof search on fragments of the language in which proof problems are expressed will subsequently be further analyzed, and specialized proof search procedures will be presented.

Example 4.1
We will show that, for a given type T and propositional functions $P, Q : T \to Prop$, the implication $(\forall\, x : T.(P\ x)) \wedge (\forall\, y : T.(Q\ y)) \to \forall\, z : T.(P\ z) \wedge (Q\ z)$ holds. For the proof term to be constructed, a metavariable $?n_0$ is introduced. We start by decomposing the connectives $\to$ and $\wedge$ on the left side (the formal

rules and the proof terms associated with them will be given in Section 4.4):

$$\dfrac{\dfrac{h_1 : \forall\, x : T.(P\ x), h_2 : \forall\, y : T.(Q\ y) \vdash ?n_2 : \forall\, z : T.(P\ z) \wedge (Q\ z)}{h_0 : (\forall\, x : T.(P\ x)) \wedge (\forall\, y : T.(Q\ y)) \vdash ?n_1 : \forall\, z : T.(P\ z) \wedge (Q\ z)}\ (\wedge L)}{\vdash ?n_0 : (\forall\, x : T.(P\ x)) \wedge (\forall\, y : T.(Q\ y)) \rightarrow \forall\, z : T.(P\ z) \wedge (Q\ z)}\ (\rightarrow R)$$

Application of these rules yields solutions $?n_0 := \lambda\, h_0 : (\forall\, x : T.(P\ x)) \wedge (\forall\, y : T.(Q\ y)).?n_1$ and $?n_1 := ?n_2\frown[h_1 := (andEl\ h_0), h_2 := (andEr\ h_0)]$, respectively.

The next step is the elimination of the universal quantifier on the right side. As opposed to traditional tableau calculi, the universally quantified variable is not simply dropped or converted into a "constant", but moved into the context.

$$\dfrac{h_1 : \forall\, x : T.(P\ x), h_2 : \forall\, y : T.(Q\ y), z : T \vdash ?n_3 : (P\ z) \wedge (Q\ z)}{h_1 : \forall\, x : T.(P\ x), h_2 : \forall\, y : T.(Q\ y) \vdash ?n_2 : \forall\, z : T.(P\ z) \wedge (Q\ z)}\ (\forall\, R)$$

$$\vdots$$

Of course, this can be interpreted as fixing the previously variable z so that it becomes a constant. It can also be interpreted as stating that the type T, when understood as a set, is non-empty (*a priori*, no assumptions are made about the emptiness or non-emptiness of types).

After $\wedge$-splitting on the right side, a metavariable $?x$ is introduced for variable x, by application of $(\forall\, L)$ at position h_1.

$$\dfrac{\dfrac{\dfrac{h_1 : \forall\, x : T.(P\ x), h_2 : \ldots, z : T \vdash ?x : T}{h_1 : \forall\, x : T.(P\ x), h_2 : \ldots, z : T, h_3 : (P\ ?x) \vdash ?n_7 : (P\ z)}\ (\forall\, L)}{h_1 : \forall\, x : T.(P\ x), h_2 : \ldots, z : T \vdash ?n_5 : (P\ z)} \qquad \ldots \vdash ?n_6 : (Q\ z)\ (\wedge R)}{\vdots}$$

Unification of $?n_7$ with h_3 succeeds and at the same time solves $?x$ with z. Here, it is important to note that some subgoals are solved by unification and not necessarily by application of sequent rules. The proof of subgoal $?n_5$ is now completely finished, with proof term $h_1\ z$. The branch of goal $?n_6$ can be handled in a similar manner, by application of $(\forall\, L)$ at position h_2 and subsequent unification.

Altogether, when reconstructing the proof term of the original goal from the individual steps, we obtain:

$$\lambda\, h_0 : (\forall\, x : T.(P\ x)) \wedge (\forall\, y : T.(Q\ y)).\, \lambda\, z : T.andI\ (andEl\ h_0\ z)\ (andEr\ h_0\ z)$$

This example is comparatively simple, in the following sense:

- Proof search proceeds deterministically, by decomposition of connectives.

- For the "existential" variables x and y (universally quantified on the left of $\vdash$), metavariables are introduced.
- It is essential that a solution of the goals $?x : T$ and $?y : T$ is not attempted directly, but that these goals are delayed and appropriate instantiations are obtained by a (first-order) unification procedure.

Even though unification restricted to a "first-order" fragment is non-trivial since type information has to be taken into account (see Section 4.3), no search space is generated during unification.

The second example below is more complex because goals of the form $\Gamma \vdash ?n : \mathit{Prop}$ have to be dealt with. Apart from left-rules, the only rule of the calculus ECC_G that is applicable to such a goal is (Π-Form$_1$):

$$\frac{\Gamma \vdash A : \mathit{Type}_j \quad \Gamma, v : A \vdash B : \mathit{Prop}}{\Gamma \vdash \Pi v : A.B : \mathit{Prop}} \ (\Pi\text{-Form}_1)$$

which solves $?n$ by a term of the form $\Pi v :?A.?B$ and creates new subgoals $\Gamma \vdash ?A : \mathit{Type}_j$ and $\Gamma, v :?A \vdash ?B : \mathit{Prop}$.

Example 4.2
In this example, it will be proved that Leibniz equality is symmetric, where Leibniz equality $=_T$ for a type T is defined by

$$x =_T y \ \widehat{=} \ \forall P : T \to \mathit{Prop}.(P\ x) \to (P\ y)$$

Thus, we have to show $\forall T : \mathit{Type}, x, y : T.x =_T y \to y =_T x$. Let us sketch the main steps in the proof. Expansion of the definition of $=_T$ gives the following initial goal:

$$\vdash ?n_0 : \ \forall T : \mathit{Type}, x, y : T. \\ (\forall P : T \to \mathit{Prop}.(P\ x) \to (P\ y)) \to (\forall P : T \to \mathit{Prop}.(P\ y) \to (P\ x))$$

Introduction of hypotheses by repeated application of the (ΠR) rule leaves a goal $\Gamma \vdash ?n_1 : (P\ x)$, where the context Γ is

$$T : \mathit{Type},\ x : T,\ y : T,\ h_0 : (\forall P : T \to \mathit{Prop}.(P\ x) \to (P\ y)), \\ P : T \to \mathit{Prop},\ h_1 : (P\ y)$$

(Remember that Π-rules are applicable to $\forall$-quantified formulae and to implications, which are both syntactic variants of Π-abstractions.)

To this goal, we apply rule (ΠL) at position h_0, and since the resulting sequents are not immediately solvable, we repeat application of (ΠL), this

time at position h_2.

$$\dfrac{\Gamma \vdash ?P_0 : T \to \mathit{Prop} \quad \dfrac{\dfrac{\Gamma, h_2 : (?P_0\ x) \to (?P_0\ y) \vdash ?n_3 : (?P_0\ x) \quad \Gamma, h_2 : (?P_0\ x) \to (?P_0\ y), h_3 : (?P_0\ y) \vdash ?n_4 : (P\ x)}{\Gamma, h_2 : (?P_0\ x) \to (?P_0\ y) \vdash ?n_2 : (P\ x)}\ (\Pi L)}{\Gamma \vdash ?n_1 : (P\ x)}\ (\Pi L)$$

Even though (higher-order) unification of $(?P_0\ x)$ with $(P\ y)$ or of $(?P_0\ y)$ with $(P\ x)$ is possible, leading to instantiations $?P_0 := \lambda z : T.(P\ y)$ and $?P_0 := \lambda z : T.(P\ x)$, respectively, these solutions do not advance the state of affairs. We therefore turn to the goal $\Gamma \vdash ?P_0 : T \to \mathit{Prop}$, which so far has only served as a side condition, and apply the following rules:

$$\dfrac{\dfrac{\Gamma, t : T \vdash ?A : \mathit{Type} \quad \Gamma, t : T, v :?A \vdash ?B : \mathit{Prop}}{\Gamma, t : T \vdash ?P_1 : \mathit{Prop}}\ (\Pi\text{-Form}_1)}{\Gamma \vdash ?P_0 : T \to \mathit{Prop}}\ (\Pi R)$$

By these rule applications, the metavariable $?P_0$ is instantiated to the term $\lambda t : T.\Pi v :?A.\ ?B$. With this instantiation, we can return to goal $?n_4$ and continue as follows:

$$\dfrac{\dfrac{\Gamma, h_2 : \ldots, h_3 : \Pi v : ?A^\frown[t := y].\ ?B^\frown[t := y] \vdash ?n_6 : ?A^\frown[t := y] \quad \Gamma, h_2 : \ldots, h_3 : \ldots, h_4 : ?B^\frown[t := y, v :=?n_6] \vdash ?n_7 : (P\ x)}{\Gamma, h_2 : \ldots, h_3 : \Pi v : ?A^\frown[t := y].\ ?B^\frown[t := y] \vdash ?n_5 : (P\ x)}\ (\Pi L)}{\Gamma, h_2 : \ldots, h_3 : ((\lambda t : T.\Pi v :?A.\ ?B)\ y) \vdash ?n_4 : (P\ x)}\ (\preceq L)$$

One solution of unifying $?n_6 : ?A^\frown[t := y]$ with $h_1 : (P\ y)$ is the instantiation $\{?n_6 := h_1, ?A := (P\ t)\}$. Similarly, unifying $?n_7 : (P\ x)$ with $h_4 : ?B^\frown[t := y, v :=?n_6]$ yields $\{?n_7 := h_4, ?B := (P\ x)\}$.

Altogether, we have synthesized the term $\lambda t : T.(P\ t) \to (P\ x)$ as a solution of $?P_0$. Propagating this solution to the only remaining subgoal $?n_3$, we obtain $\Gamma, h_2 : \ldots \vdash ?n_3 : ((\lambda t : T.(P\ t) \to (P\ x))\ x)$, which, after reduction of the goal formula, is trivial to prove.

The search space generated by this proof is hard to control, mainly because:

- Higher-order unification does not produce a single most general unifier, but there are possibly several independent unifiers. For example, unification of $?A^\frown[t := y]$ and $(P\ y)$ has the two solutions $?A := (P\ y)$ and $?A := (P\ t)$.

- Application of the rule (Π-Form$_1$) (similarly (Π-Form$_2$)) to a proof obligation $?P$ creates two new proof obligations, each of which has the same "complexity" as $?P$ itself. It is not evident how to interleave the rule (Π-Form$_1$) with other rules during proof search.

After these illustrations of some of the specificities of proof search in type theory, the following sections concentrate on techniques which help to make proof search practicable, at least for restricted fragments. Unification is one such technique which plays a major role in proof search, but which is also used for other purposes in verification environments like TYPELAB.

Example 4.3
In TYPELAB, as well as in other proof assistants like Lego [LP92], function expressions can be written in a style of "implicit polymorphism", which avoids making type arguments of functions explicit whenever they can be inferred. (This is not to be confused with an ML-style type inference, which is undecidable in the presence of dependent types [Wel94]. Even though incomplete, the algorithm sketched below works well in practice.)

For example, the explicit notation $map\ (List\ Nat)\ Nat\ sum\ [[1,2],[3,4,5]]$ can be abbreviated to $map\ sum\ [[1,2],[3,4,5]]$. Here, the function map has type $\Pi A, B \mid Type.(A \to B) \to (List\ A) \to (List\ B)$ and sum (adding up the elements of a list) the type $(List\ Nat) \to Nat$. In $\Pi A, B \mid Type. \ldots$, the vertical bar indicates that the types A and B have to be inferred.

To type-check the incomplete expression $map\ sum\ [[1,2],[3,4,5]]$ in a context Γ, the algorithm proceeds as follows: For the hidden type variables A and B, metavariables $\Gamma \vdash ?A : Type$ and $\Gamma, ?A : Type \vdash ?B : Type$ are generated. Now, type checking of the "explicit" expression $map\ ?A\ ?B\ sum\ [[1,2],[3,4,5]]$ is carried out. For the subexpression $map\ ?A\ ?B$, the type $(?A \to ?B) \to (List\ ?A) \to (List ?B)$ is determined. Unifying the domain type $(?A \to ?B)$ with the type $(List\ Nat) \to Nat$ of sum yields the substitution $\{?A := (List\ Nat), ?B := Nat\}$. Applying the resulting $map\ (List\ Nat)\ Nat\ sum$ to argument $[[1,2],[3,4,5]]$ of type $(List\ Nat)$ leads to no further instantiation of type parameters.

In Section 4.3, we will take a closer look at unification. Even in the "first-order" case, unification is not trivial, because several issues related to typing have to be taken into account. As a motivation, we show in Section 4.2 how some of the properties unification has to satisfy can be derived from the calculus ECC_G presented in Chapter 3, in particular from the rules $(\preceq L)$ and $(\preceq R)$, which are directly related to conversion. This gives a basis for defining which problems have to be solved by unification and what constitutes an acceptable solution (Section 4.3.1). We then proceed to discuss procedures resembling standard first-order unification (Section 4.3.2), and we establish a correspondence to general higher-order unification (Section 4.3.3) and to a special kind of higher-order unification problems, so-called patterns (Section 4.3.4). We finally discuss why it is does not appear to be useful to devise a complete unification algorithm for the whole language under consideration (Section 4.3.5).

Having introduced unification, we extend the calculus ECC_G to the standard logical connectives (Section 4.4). The resulting calculus strongly bears resemblance to traditional tableau calculi, and we will discuss to which degree some of the techniques developed for tableau calculi are adequate for our calculus.

4.2. The Structure of Proofs

In Chapter 3, a sequent system ECC_G has been presented and shown to be (in a certain sense) "equivalent" to the natural deduction system ECC_N of Chapter 2. The system ECC_G is not directly usable for proof search, for the following reasons:

- When trying to prove whether type A is inhabited in context Γ, the proof term M with $\Gamma \vdash M : A$ would have to be known in advance in order to apply the rules of system ECC_G. Of course, as illustrated by the examples of Section 4.1, the solution to this problem is to introduce a metavariable $?M$ for the term to be constructed, to carry out the proof and to instantiate $?M$ with a term according to the rule that has been applied to the goal $\Gamma \vdash ?M : A$. The search space is further pruned by determining appropriate instantiations of metavariables with a unification procedure.

- Because of the rules $(\preceq L)$ and $(\preceq R)$, the calculus ECC_G still suffers from a lack of subformula property. For example, in the rule $(\preceq R)$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash_N A' : \mathit{Type} \quad A \preceq A'}{\Gamma \vdash M : A'} \ (\preceq R)$$

  the term A is not known in advance.

The purpose of this section is to deal with the second problem, by showing that in a derivation, applications of a $\preceq$-rule can be split into two parts, where the first part is an innocent deterministic reduction of A' to its weak head normal form A_w, and the second, more difficult part is shifted towards the leaves of the proof tree, where it can be incorporated into the unification procedure.

Example 4.4
Given the goal $a : A, h : (\lambda\, X : \mathit{Prop}.X \to B \to \mathit{Prop})\ A \vdash ?M_0 : B \to \mathit{Type}_0$,

the following derivation can be carried out with the rules of ECC_G:

$$\dfrac{\dfrac{\dots \vdash a : A \quad \dfrac{\vdots}{a : A, h : A \to B \to \mathit{Type}_0, h' : B \to \mathit{Type}_0 \vdash ?M_2 : B \to \mathit{Type}_0}\ (\mathrm{var})}{a : A, h : A \to B \to \mathit{Type}_0 \vdash ?M_1 : B \to \mathit{Type}_0}\ (\Pi L)}{a : A, h : (\lambda\, X : \mathit{Prop}.X \to B \to \mathit{Prop})\ A \vdash ?M_0 : B \to \mathit{Type}_0}\ (\preceq L)$$

(with $?M_2 := h'$ and $?M_1 := ?M_0 := (h\ a)$).

In this derivation, the conversion of $(\lambda\, X : \mathit{Prop}.X \to B \to \mathit{Prop})\ A$ to $A \to B \to \mathit{Type}_0$ does not have an obvious motivation. Compare this with the following derivation:

$$\dfrac{\dfrac{\dots \vdash a : A \quad \dfrac{\dfrac{\vdots}{a : A, h : A \to B \to \mathit{Prop}, h' : B \to \mathit{Type}_0 \vdash ?M_3 : B \to \mathit{Type}_0}\ (\mathrm{var})}{a : A, h : A \to B \to \mathit{Prop}, h' : B \to \mathit{Prop} \vdash ?M_2 : B \to \mathit{Type}_0}\ (\preceq L)}{a : A, h : A \to B \to \mathit{Prop} \vdash ?M_1 : B \to \mathit{Type}_0}\ (\Pi L)}{a : A, h : (\lambda\, X : \mathit{Prop}.X \to B \to \mathit{Prop})\ A \vdash ?M_0 : B \to \mathit{Type}_0}\ (\text{whnf } L)$$

This derivation is demand-driven in that the weak head normal form reduction of $(\lambda\, X : \mathit{Prop}.X \to B \to \mathit{Prop})\ A$ to $A \to B \to \mathit{Prop}$ by (whnf L) is required for an application of the (ΠL)-rule, and the conversion of $B \to \mathit{Prop}$ to $B \to \mathit{Type}_0$ is required for an application of the (var) rule.

The unification procedure developed below will permit to perform applications of $(\preceq L)$, $(\preceq R)$ and (var) in one step, so that the derivation becomes

$$\dfrac{\dfrac{\dots \vdash a : A \quad \dfrac{\vdots}{a : A, h : A \to B \to \mathit{Prop}, h' : B \to \mathit{Prop} \vdash ?M_2 : B \to \mathit{Type}_0}\ (\mathrm{var}^*)}{a : A, h : A \to B \to \mathit{Prop} \vdash ?M_1 : B \to \mathit{Type}_0}\ (\Pi L)}{a : A, h : (\lambda\, X : \mathit{Prop}.X \to B \to \mathit{Prop})\ A \vdash ?M_0 : B \to \mathit{Type}_0}\ (\text{whnf } L)$$

where (var*) is the (var) rule comprising a generalized unification which permits to equate $h' : B \to \mathit{Prop}$ and $?M_2 : B \to \mathit{Type}_0$ because $B \to \mathit{Prop} \preceq B \to \mathit{Type}_0$.

Let us now formally introduce the rules (whnf L) and (whnf R) which will supplant applications of the $\preceq$-rules in the interior of proof trees:

$$\dfrac{\Gamma \vdash M : A \quad \Gamma \vdash_N A' : \mathit{Type} \quad A \text{ whnf of } A'}{\Gamma \vdash M : A'}\ (\text{whnf } R)$$

$$\dfrac{\Gamma, p : T, \Gamma' \vdash M : A \quad \Gamma \vdash_N T : \mathit{Type} \quad T \text{ whnf of } T'}{\Gamma, p : T', \Gamma' \vdash M : A}\ (\text{whnf } L)$$

Since these rules are weakened versions of $(\preceq L)$ and $(\preceq R)$, their correctness is obvious. Also note that the side conditions $\Gamma \vdash_N A' : \mathit{Type}$ and $\Gamma \vdash_N T : \mathit{Type}$ are required when reading the rules from premiss to conclusion, because the converse of a subject reduction theorem ("subject expansion") does not hold. However, they do not lead to proof obligations when the rule is applied backwards in proof search, because Proposition 2.47 insures that the condition is verified.

Proposition 4.5
If there is a derivation of $\Gamma \vdash M : A$ in system ECC_G, then there is a derivation $\Gamma \vdash M' : A$ in the system with the rules of ECC_G and (whnf L), (whnf R), such that:

- $M \simeq M'$
- The rule $(\preceq L)$ is only applied directly below the (var) rule, and only at the position which is the subject of the (var) rule (i.e. the variable x with $\Gamma, x : T, \Gamma' \vdash x : T$).
- The rule $(\preceq R)$ is only applied directly below (UProp), (UType), (var) and $(\preceq L)$

Proof: The proof proceeds by induction on derivations, showing that applications of $(\preceq L)$ and $(\preceq R)$ can be permuted upwards in the derivation tree. The proof is lengthy, but not difficult. Details are given in Appendix A.3, page 178. □

Proposition 4.5 has the following consequences for proof search: A proof can be transformed in such a way that the only occurrences of the nondeterministic rules $(\preceq R)$ and $(\preceq L)$ are as follows:

$$\dfrac{\dfrac{\Gamma \ \mathit{valid}_c}{\Gamma \vdash \mathit{Prop} : \mathit{Type}_0}\ (\text{UProp})}{\Gamma \vdash \mathit{Prop} : U}\ (\preceq R) \qquad \dfrac{\dfrac{\Gamma \ \mathit{valid}_c}{\Gamma \vdash \mathit{Type}_j : \mathit{Type}_{j+1}}\ (\text{UType})}{\Gamma \vdash \mathit{Type}_j : U}\ (\preceq R)$$

for U convertible to an appropriate type universe, and

$$\dfrac{\dfrac{\dfrac{\Gamma, x : A, \Gamma' \ \mathit{valid}_c}{\Gamma, x : A, \Gamma' \vdash x : A}\ (\text{var})}{\Gamma, x : A', \Gamma' \vdash x : A}\ (\preceq L)}{\Gamma, x : A', \Gamma' \vdash x : A''}\ (\preceq R)$$

where $A' \preceq A \preceq A''$.

For the purpose of proof search, the combination of $(\preceq R)$ and universe rules will be merged into a rule

$$\frac{U \in \{Prop, Type_0, \ldots Type_{j-1}\}}{\Gamma \vdash U : Type_j} \text{ (Univ)}$$

The combination of $(\preceq R)$, $(\preceq L)$ and the (var) rule will be merged into

$$\frac{A' \preceq A''}{\Gamma, x : A', \Gamma' \vdash x : A''} \text{ (var*)}$$

It is obvious that with these new rules (that is, (Univ), (var*) and the whnf-rules), applications of $(\preceq R)$ and $(\preceq L)$ can completely be dispensed with. In the practice of proof search, all applications of the (Univ) rule can be delayed and solved at the end of the proof as a set of constraints between universes, comparable to the approach in [HP91] (this will not be described in detail below). The precondition $A' \preceq A''$ in rule (var*) will be ensured by a special unification procedure taking into account cumulativity. This issue will be examined in detail in the following sections. Note that in proof search, we assume that we manipulate valid contexts, so the corresponding validity conditions have been omitted from the above rules.

4.3. Unification

4.3.1. Unification Problems

A unification problem consists in finding an instantiation ι which, when applied to two terms t_1 and t_2, makes both terms equal. In this informal definition, the concepts of "equality" and of "instantiation" have to be made more precise.

Equality is taken modulo convertibility $\simeq$, thus the unification problems we obtain will in general be higher-order. The discussion of the previous section even suggests that it is useful to consider a unification which equates terms modulo cumulativity $\preceq$.

Furthermore, we have the choice between arbitrary instantiations and instantiations fulfilling some additional requirements, such as being well-typed in the sense of Definition 2.71, that is, producing only typecorrect terms. In the first case, solutions of unification problems may be easier to obtain, but they are practically useless for proof development. We will therefore only regard instantiations as acceptable solutions to unification problems if they are well-typed. This introduces some additional complexity in the definition, since a unification problem is then defined relative to a context and a proof problem.

Definition 4.6 (Unification Problem)
Let $\mathcal{P}$ be a well-typed proof problem, Γ be a valid context, and s and t terms which are well-typed in Γ.

- A *unification problem* is a pair of the form $\langle \mathcal{P} \ ; \ \Gamma \vdash s \stackrel{?}{\simeq} t \rangle$.
- A *cumulative unification problem* is a pair of the form $\langle \mathcal{P} \ ; \ \Gamma \vdash s \stackrel{?}{\preceq} t \rangle$.

The terms s, t of a unification problem have to be well-typed in a context Γ (with metavariables taken from a proof problem $\mathcal{P}$). This is required because on some occasions, we have to compute the types of terms – see for example rule (MV-term) below – or reduce terms to normal form, which can only securely be done for well-typed terms. However, the types of the terms do not have to be equal. On the contrary, it is one of the main purposes of a unification procedure to determine an instantiation that makes two terms and their types agree. Alternatively, one could postulate for $\Gamma \vdash s \stackrel{?}{\simeq} t$ that s and t not only have to be well-typed in Γ, but even have to be of the same type (this is for example the approach taken in most higher-order unification algorithms). However, if two terms have the same type, their subterms do not necessarily share this property, so a unification algorithm would have to enforce this condition at each step – take for example $f : A \to B \to C, g : A \to C, a : A, b : B \vdash (f \ a \ b) \stackrel{?}{\simeq} (g \ a)$, where $(f \ a \ b)$ and $(g \ a)$ have the same type, but their respective subterms $(f \ a)$ and g do not. For simply-typed calculi, another approach than a decomposition of an application into function and argument is conceivable (see Section 4.3.3), which does not seem to scale up well to dependently typed calculi (see Section 4.3.5).

Example 4.7
Consider the unification problem $\langle \mathcal{P} \ ; \ \Gamma \vdash ?n_1 \stackrel{?}{\simeq} h \rangle$, where $\mathcal{P}$ is:

$$\{\Gamma \vdash ?n_1 : (P \ ?n_2), \quad \Gamma \vdash ?n_2 : A\}$$

and the context Γ:

$$A : \mathit{Type}, P : A \to \mathit{Prop}, a : A, h : (P \ a)$$

The unification problem can easily be solved by an instantiation $\{?n_1 := h\}$, which, however, is not typecorrect, since $?n_1$ is of type $(P \ ?n_2)$ and h of type $(P \ a)$. The instantiation $\{?n_1 := h, ?n_2 := a\}$ also solves the equation and, in addition, is typecorrect. Only instantiations of this kind will be accepted as solutions of unification problems.

Definition 4.8 (Solution of a unification problem)
A *solution* of a unification problem $\langle \mathcal{P} \ ; \ \Gamma \vdash s \stackrel{?}{\simeq} t \rangle$ (resp. of a cumulative unification problem $\langle \mathcal{P} \ ; \ \Gamma \vdash s \stackrel{?}{\preceq} t \rangle$) is an instantiation ι with the following properties:

- ι is well-typed for $\mathcal{P}$.
- $\iota(s) \ \simeq \ \iota(t)$ (resp. $\iota(s) \preceq \iota(t)$).

It is important to note that a unification problem depends on a proof problem, but not vice versa. This is in contrast to the representation proposed by Dowek [Dow93], where unification equations are kept in a generalized context together with variable and metavariable declarations and are simplified as metavariables are instantiated. As a consequence, typecorrectness of terms can depend on the solvability of unification equations, type checking is therefore not decidable in general. Whereas this is acceptable for a proof search method which exhaustively generates solutions for metavariables, possibly delaying a verification of their typecorrectness, such a procedure is not appropriate for an interactive proof development system.

In order to define a correctness criterion, unification problems and their solutions are stated relative to a fixed proof problem $\mathcal{P}$, which permits to assess the validity of instantiations. It might be objected that these definitions are too restrictive, in that they do not allow for the creation of new metavariables. Thus, for example, some standard methods for higher-order unification seem to be excluded. They can, however, be accommodated in our framework by slightly extending the notion of solution given in Definition 4.8, see Section 4.3.3.

4.3.2. First-order unification

In the following, an algorithm which essentially carries out first-order unification will be presented. Roughly speaking, we are aiming at a unification that equates terms modulo α-convertibility, but not modulo β-convertibility. The notion of "essentially first-order" does not imply that the language is restricted to a fragment that does not comprise features such as λ-abstraction. Operationally, first-order unification is a structural comparison of two terms; a metavariable $?n$ and a term t can be compared by adding an appropriate assignment $?n := t$ to the instantiation to be constructed.

The interest in first-order unification derives from the fact that it is rather straightforward to relate correctness criteria of unification to notions such as validity and well-typedness of instantiations which were examined in Section 2.6 (see also Definition 4.8). Completeness criteria are more difficult to state. For

reasons laid out more fully in Section 4.3.3, explicit substitutions induce a higher-order aspect which makes it harder to delimit an appropriate fragment and to develop a complete algorithm for it. In Section 4.3.4, a comparison is made with unification algorithms of "patterns" which are sometimes claimed to display an essentially first-order behaviour.

The rules defining the unification algorithm (Figure 4.1) use a judgement of the form $\langle \mathcal{P}_0 \; ; \; \Gamma \vdash t_1 \stackrel{?}{\simeq} t_2 \rangle \Rightarrow \mathcal{P}_1; \; \iota_1$, which expresses that the unification problem $\langle \mathcal{P}_0 \; ; \; \Gamma \vdash t_1 \stackrel{?}{\simeq} t_2 \rangle$ can be solved by instantiation ι_1, leaving open the metavariables of the proof problem $\mathcal{P}_1$.

Let us comment on some peculiarities of the rules:

- The preconditions of some of the rules have a sequential reading, even though in most cases, there is no intrinsic necessity to impose any particular order on how subterms are unified. For example, applications could be decomposed in the style of a Martelli-Montanari unification algorithm. However, the presentation chosen here simplifies reasoning about the behaviour of the algorithm, as it keeps track of which proof problems and which instantiations are generated at each stage.

- In the quantifier rule ($\mathcal{Q}$-$\mathcal{Q}$), the types of the abstraction variables are unified before the bodies. An α-conversion is carried out by renaming the binder variable of the second term to the binder variable in the first term. In this particular case, the order in which subterms are unified does matter. For example, if $f_{Nat} : Nat \to Nat$ and $f_{Bool} : Bool \to Bool$, then each of the terms is typecorrect in the unification problem $\vdash \lambda\, x_1 : Nat.\ (f_{Nat}\ x_1) \stackrel{?}{\simeq} \lambda\, x_2 : Bool.\ (f_{Bool}\ x_2)$, although the types are incompatible. If not solving the preconditions in the indicated order, one would obtain a type-incorrect intermediate unification problem $x_1 : Nat \vdash (f_{Nat}\ x_1) \stackrel{?}{\simeq} (f_{Bool}\ x_1)$.

- Some of the rules use the combination $\iota_1 \uplus \iota_2$ of instantiations. The constraint imposed by Definition 2.65 to make the resulting instantiation defined is satisfied, since if $\iota_1(?m) \neq ?m$, then the proof problem $\mathcal{P}_1$ from which instantiation ι_2 is computed does not contain $?m$ any more, thus whenever $(?n := t) \in \iota_2$, then $?m \notin MVars(t)$.

Rule (MV-term) deserves some closer attention. In order to unify a metavariable $?n$ and an arbitrary term t, the type of $?n$ and the type T of t are first unified, yielding an instantiation ι_1 (remember that in a unification problem with $\Gamma \vdash t_1 \stackrel{?}{\simeq} t_2$, it is not known whether the types of t_1 and t_2 agree). It now still has to be verified that the instantiation $\{?n := \iota_1(t)\}$ is valid. If all

$$\frac{}{\langle \mathcal{P} \;;\; \Gamma \vdash x \overset{?}{\simeq} x \rangle \Rightarrow \mathcal{P};\; \{\}} \text{ (var-var)}$$

$$\frac{}{\langle \mathcal{P} \;;\; \Gamma \vdash \mathit{Prop} \overset{?}{\simeq} \mathit{Prop} \rangle \Rightarrow \mathcal{P};\; \{\}} \text{ (Prop-Prop)}$$

$$\frac{}{\langle \mathcal{P} \;;\; \Gamma \vdash \mathit{Type}_i \overset{?}{\simeq} \mathit{Type}_i \rangle \Rightarrow \mathcal{P};\; \{\}} \text{ (Type-Type)}$$

$$\frac{\begin{array}{c}\langle \mathcal{P}_0 \;;\; \Gamma \vdash f_1 \overset{?}{\simeq} f_2 \rangle \Rightarrow \mathcal{P}_1;\; \iota_1 \\ \langle \mathcal{P}_1 \;;\; \iota_1(\Gamma) \vdash \iota_1(a_1) \overset{?}{\simeq} \iota_1(a_2) \rangle \Rightarrow \mathcal{P}_2;\; \iota_2\end{array}}{\langle \mathcal{P}_0 \;;\; \Gamma \vdash (f_1\ a_1) \overset{?}{\simeq} (f_2\ a_2) \rangle \Rightarrow \mathcal{P}_2;\; \iota_1 \uplus \iota_2} \text{ (app-app)}$$

$$\frac{\langle \mathcal{P}_0 \;;\; \Gamma \vdash t_1 \overset{?}{\simeq} t_2 \rangle \Rightarrow \mathcal{P}_1;\; \iota_1}{\langle \mathcal{P}_0 \;;\; \Gamma \vdash \pi_i(t_1) \overset{?}{\simeq} \pi_i(t_2) \rangle \Rightarrow \mathcal{P}_1;\; \iota_1} \text{ (π-π)}$$

$$\frac{\begin{array}{c}\langle \mathcal{P}_0 \;;\; \Gamma \vdash A_1 \overset{?}{\simeq} A_2 \rangle \Rightarrow \mathcal{P}_1;\; \iota_1 \\ \langle \mathcal{P}_1 \;;\; \iota_1(\Gamma, x_1 : A_1) \vdash \iota_1(B_1) \overset{?}{\simeq} \iota_1(B_2\{x_2 := x_1\}) \rangle \Rightarrow \mathcal{P}_2;\; \iota_2 \\ \mathcal{Q} \in \{\lambda, \Pi, \Sigma\}\end{array}}{\langle \mathcal{P}_0 \;;\; \Gamma \vdash \mathcal{Q} x_1 : A_1.B_1 \overset{?}{\simeq} \mathcal{Q} x_2 : A_2.B_2 \rangle \Rightarrow \mathcal{P}_2;\; \iota_1 \uplus \iota_2} \text{ ($\mathcal{Q}$-$\mathcal{Q}$)}$$

$$\frac{\begin{array}{c}\langle \mathcal{P}_0 \;;\; \Gamma \vdash T_1 \overset{?}{\simeq} T_2 \rangle \Rightarrow \mathcal{P}_1;\; \iota_1 \\ \langle \mathcal{P}_1 \;;\; \iota_1(\Gamma) \vdash \iota_1(s_1) \overset{?}{\simeq} \iota_1(s_2) \rangle \Rightarrow \mathcal{P}_2;\; \iota_2 \\ \langle \mathcal{P}_2 \;;\; \iota_2(\iota_1(\Gamma)) \vdash \iota_2(\iota_1(t_1)) \overset{?}{\simeq} \iota_2(\iota_1(t_2)) \rangle \Rightarrow \mathcal{P}_3;\; \iota_3\end{array}}{\langle \mathcal{P}_0 \;;\; \Gamma \vdash \mathit{pair}_{T_1}(s_1,\ t_1) \overset{?}{\simeq} \mathit{pair}_{T_2}(s_2,\ t_2) \rangle \Rightarrow \mathcal{P}_3;\; \iota_1 \uplus \iota_2 \uplus \iota_3} \text{ (pair-pair)}$$

$$\frac{\begin{array}{c}\mathit{ctxt}_{\mathcal{P}_0}(?n) \vdash t : T \\ \langle \mathcal{P}_0 \;;\; \mathit{ctxt}_{\mathcal{P}_0}(?n) \vdash T \overset{?}{\preceq} \mathit{type}_{\mathcal{P}_0}(?n) \rangle \Rightarrow \mathcal{P}_1;\; \iota_1 \\ \forall ?k \in \mathit{MVars}(T).?k \ll_{\mathcal{P}_0} ?n \\ \mathit{valid}(\{?n := \iota_1(t)\}, \mathcal{P}_1) \\ \iota_2 := \iota_1 \uplus \{?n := \iota_1(t)\} \quad \mathcal{P}_2 := \mathcal{P}_1\{?n := \iota_1(t)\}\end{array}}{\langle \mathcal{P}_0 \;;\; \Gamma \vdash ?n \overset{?}{\simeq} t \rangle \Rightarrow \mathcal{P}_2;\; \iota_2} \text{ (MV-term)}$$

Figure 4.1.: First-order unification

these conditions are satisfied, it can be concluded that this instantiation is well-typed (this will be proved in Proposition 4.10). By definition, the predicate $\mathit{valid}(\iota, \mathcal{P})$ used in rule (MV-term) is satisfied if:

- the instantiation ι passes the occurs check implicit in Definition 2.62 and
- the instantiation ι is valid in the sense of Definition 2.71, i.e., $\iota(\mathcal{P})$ is a valid proof problem, that is, a proof problem without circular metavariable dependencies.

Section 2.6.2 has discussed methods which permit to efficiently compute the predicate *valid*.

The side condition $\forall ?k \in \mathit{MVars}(T).?k \ll_{\mathcal{P}_0} ?n$ serves two purposes: It is used in the proof of termination of the unification algorithm (see proof of Proposition 4.11) and it ensures that $\iota_1 \uplus \{?n := \iota_1(t)\}$ is well-defined (see proof of Proposition 4.10). To see which kind of a situation this condition is thought to prevent, consider the following example derivation which violates this condition: Assume that metavariable $?n$ is defined by $\Gamma \vdash ?n : A$ for an (unspecified) context Γ and there is a term s such that $\Gamma \vdash s : ?n$. Unification of $\Gamma \vdash ?n \stackrel{?}{\simeq} s$ is first reduced to $\Gamma \vdash ?n \stackrel{?}{\preceq} A$ by application of rule (MV-term), and then further to $\Gamma \vdash ?n \stackrel{?}{\simeq} A$. One potential problem of such a derivation is non-termination, since situations of the form $?n \stackrel{?}{\simeq} s$ and $?n \stackrel{?}{\simeq} A$ reoccur and there is no obvious measure which decreases in the course of the derivation. Secondly, if an assignment $?n := A$ were possible (which it is not in this case), two conflicting assignments $?n := A$ and $?n := s$ would result. Note that this example is not a conclusive argument for the necessity of including the above side condition in rule (MV-term). In fact, we conjecture that the properties of unification stated in the following theorems would remain valid even if this condition were dropped. On the other hand, adding this restriction does not do any harm in applications such as proof search in sequent calculi, where it can be shown to be satisfied a priori.

In rule (MV-term), the problem of testing whether the types of metavariable $?n$ and term t are compatible is reduced to a cumulative unification problem. This is justified by the discussion in Section 4.2, in particular the rule

$$\frac{A' \preceq A''}{\Gamma, x : A', \Gamma' \vdash x : A''} \ (\text{var}^*)$$

derived there. To see the practical implications, consider the problem of unifying a metavariable $?x$ of type Type_1 with a variable x of type Type_0 (in this case, A' is Type_0 and A'' is Type_1). Obviously, the instantiation $\{?x := x\}$ is well-typed, since by cumulativity, x is also of type Type_1. Thus, it is not

sufficient to simply unify the type of $?x$ and the type of x in order to find out whether x is a valid solution of $?x$.

The rules for cumulative unification are given in Figure 4.2. They follow the generation of the cumulativity relation $\preceq$ (see Definition 2.3). The rule (cu-term-term) handles all the cases not covered by the other rules.

$$\frac{\kappa \in \{\mathit{Prop}, \mathit{Type}_i\}}{\langle \mathcal{P} \;;\; \Gamma \vdash \mathit{Prop} \overset{?}{\preceq} \kappa \rangle \Rightarrow \mathcal{P};\; \{\}} \text{ (cu-Prop-Kind)}$$

$$\frac{i \leq j}{\langle \mathcal{P} \;;\; \Gamma \vdash \mathit{Type}_i \overset{?}{\preceq} \mathit{Type}_j \rangle \Rightarrow \mathcal{P};\; \{\}} \text{ (cu-Type-Kind)}$$

$$\frac{\begin{array}{c}\langle \mathcal{P}_0 \;;\; \Gamma \vdash A_1 \overset{?}{\simeq} A_2 \rangle \Rightarrow \mathcal{P}_1;\; \iota_1 \\ \langle \mathcal{P}_1 \;;\; \iota_1(\Gamma, x_1 : A_1) \vdash \iota_1(B_1) \overset{?}{\preceq} \iota_1(B_1\{x_2 := x_1\}) \rangle \Rightarrow \mathcal{P}_2;\; \iota_2\end{array}}{\langle \mathcal{P}_0 \;;\; \Gamma \vdash \Pi x_1 : A_1 . B_1 \overset{?}{\preceq} \Pi x_2 : A_2 . B_2 \rangle \Rightarrow \mathcal{P}_2;\; \iota_1 \uplus \iota_2} \text{ (cu-}\Pi\text{-}\Pi\text{)}$$

$$\frac{\begin{array}{c}\langle \mathcal{P}_0 \;;\; \Gamma \vdash A_1 \overset{?}{\preceq} A_2 \rangle \Rightarrow \mathcal{P}_1;\; \iota_1 \\ \langle \mathcal{P}_1 \;;\; \iota_1(\Gamma, x_1 : A_1) \vdash \iota_1(B_1) \overset{?}{\preceq} \iota_1(B_1\{x_2 := x_1\}) \rangle \Rightarrow \mathcal{P}_2;\; \iota_2\end{array}}{\langle \mathcal{P}_0 \;;\; \Gamma \vdash \Sigma x_1 : A_1 . B_1 \overset{?}{\preceq} \Sigma x_2 : A_2 . B_2 \rangle \Rightarrow \mathcal{P}_2;\; \iota_1 \uplus \iota_2} \text{ (cu-}\Sigma\text{-}\Sigma\text{)}$$

$$\frac{\langle \mathcal{P}_0 \;;\; \Gamma \vdash t_1 \overset{?}{\simeq} t_2 \rangle \Rightarrow \mathcal{P}_1;\; \iota_1}{\langle \mathcal{P}_0 \;;\; \Gamma \vdash t_1 \overset{?}{\preceq} t_2 \rangle \Rightarrow \mathcal{P}_1;\; \iota_1} \text{ (cu-term-term)}$$

Figure 4.2.: First-order cumulative unification

After these remarks, let us point out one technicality: Rule (MV-term) has a companion rule in which the roles of $?n$ and t are reversed, that is, which solves a unification problem of the form $\Gamma \vdash t \overset{?}{\simeq} ?n$ instead of $\Gamma \vdash ?n \overset{?}{\simeq} t$.

Example 4.9
Unification is extensively used in proof search. A frequently occurring situation is the following: given a hypothesis $h : \forall\, x : T . P(x)$ and a constant $z : T$, one tries to establish that $P(?y)$ holds. Thus, the proof goal is:

$\Gamma \vdash ?n : P(?y)$, where the context Γ contains the hypothesis h and the constant z. We can attempt to solve this goal by unifying $(h\ z)$ and $?n$. Let us spell out in detail how the unification algorithm proceeds. The original unification problem is $\langle \mathcal{P}_0 \ ; \ \Gamma \vdash ?n \stackrel{?}{\simeq} (h\ z) \rangle$, which is transformed to the problem $\langle \mathcal{P}_0 \ ; \ \mathit{ctxt}_{\mathcal{P}_0}(?n) \vdash P(z) \stackrel{?}{\preceq} P(?y) \rangle$ by application of the rule (MV-term), because $\mathit{ctxt}_{\mathcal{P}_0}(?n) \vdash (h\ z) : P(z)$. After applications of the rules (cu-term-term), (app-app), (var-var) and after switching sides, we are left with the problem $\langle \mathcal{P}_0 \ ; \ \mathit{ctxt}_{\mathcal{P}_0}(?n) \vdash ?y \stackrel{?}{\simeq} z \rangle$. Let us examine the following situations:

- $\mathit{ctxt}(?y)$ does not contain the declaration $z : T$. In this case, computing the type of z in $\mathit{ctxt}(?y)$ fails and the rule (MV-term) is not applicable. This ultimately leads to a failure of $?n \stackrel{?}{\simeq} (h\ z)$.

- $\mathit{ctxt}(?y)$ contains the declaration $z : T$, thus $\mathit{ctxt}(?y) \vdash z : T$. Since the type of z is compatible with $\mathit{type}(?y) = T$, and assuming that the validity of the instantiation can be established, we obtain $\langle \mathcal{P}_0 \ ; \ \Gamma \vdash ?y \stackrel{?}{\simeq} z \rangle \Rightarrow \mathcal{P}_1; \ \{?y := z\}$, where $\mathcal{P}_1$ contains the proof problem $\Gamma\{?y := z\} \vdash ?n : P(z)$. Since the types of $h\ z$ and $?n$ have now been shown to be unifiable, we can equate $?n$ and $h\ z$. Thus, in the end, the original unification problem is solved by the instantiation $\{?y := z, ?n := (h\ z)\}$.

The first kind of situation can arise when trying to prove $(\forall\, x : T.P(x)) \rightarrow (\exists\, y : T.\forall\, z : T.P(y))$, which does not hold, because there is no witness of type T for variable y. The second situation arises when trying to prove $(\forall\, x : T.P(x)) \rightarrow (\forall\, z : T.\exists\, y : T.P(y))$, which does hold. Section 4.4.2 will discuss in depth how unification and our particular presentation of proof rules interact to deal with questions of quantifier alternations and eigenvariable conditions, of which these two formulae are reminiscent.

Altogether, the rules satisfy the following invariants, which can be understood as stating the correctness of the unification algorithm.

Proposition 4.10 (Invariants of Unification)
Assume $\langle \mathcal{P}_0 \ ; \ \Gamma \vdash t_1 \stackrel{?}{\simeq} t_2 \rangle$ is a unification problem (resp. $\langle \mathcal{P}_0 \ ; \ \Gamma \vdash t_1 \stackrel{?}{\preceq} t_2 \rangle$ a cumulative unification problem). If $\langle \mathcal{P}_0 \ ; \ \Gamma \vdash t_1 \stackrel{?}{\simeq} t_2 \rangle \Rightarrow \mathcal{P}_1; \ \iota_1$ is derivable with the rules of Figure 4.1 (resp. $\langle \mathcal{P}_0 \ ; \ \Gamma \vdash t_1 \stackrel{?}{\preceq} t_2 \rangle \Rightarrow \mathcal{P}_1; \ \iota_1$ is derivable with the rules of Figure 4.2), then:

- $\iota_1(t_1) \ \simeq \ \iota_1(t_2)$ (resp. $\iota_1(t_1) \preceq \iota_1(t_2)$)

- ι_1 is a valid, well-typed instantiation for $\mathcal{P}_0$.

- $\mathcal{P}_1 = \iota_1(\mathcal{P}_0)$. This entails that $\mathcal{P}_1$ is a valid, well-typed proof problem.

Proof: Induction on the structure of the derivation of the unification judgement. The condition $\iota_1(t_1) \simeq \iota_1(t_2)$ (resp. $\iota_1(t_1) \preceq \iota_1(t_2)$) is immediately verified by inspection of the rules.

The fact that $\mathcal{P}_1 = \iota_1(\mathcal{P}_0)$ is also established by a simple inductive argument, observing the following 'transitivity' in rules (MV-term), (app-app) etc.: if $\mathcal{P}_1 = \iota_1(\mathcal{P}_0)$ and $\mathcal{P}_2 = \iota_2(\mathcal{P}_1)$, then $\mathcal{P}_2 = (\iota_1 \uplus \iota_2)(\mathcal{P}_0)$.

To show that ι_1 is a valid, well-typed instantiation for $\mathcal{P}_0$, we only consider the rule (MV-term). For the other rules, the claim follows from the induction hypothesis.

The following conditions have to be verified in order to show that the resulting instantiation is valid and well-typed:

- The unification problem in the precondition of the rule has to be valid in the sense that both $type_{\mathcal{P}_0}(?n)$ and T are well-typed in context $ctxt_{\mathcal{P}_0}(?n)$. Since $\mathcal{P}_0$ is a valid proof problem and $?n \in \mathcal{P}_0$, we have that $type_{\mathcal{P}_0}(?n)$ is well-typed in $ctxt_{\mathcal{P}_0}(?n)$. The fact that T is well-typed in $ctxt_{\mathcal{P}_0}(?n)$ is enforced by the first premiss of rule (MV-term).

- The resulting instantiation (in particular the assignment of $\iota_1(t)$ to $?n$) must be well-typed. If $\langle \mathcal{P}_0 \ ; \ \Gamma \vdash T \stackrel{?}{\preceq} type_{\mathcal{P}_0}(?n) \rangle \Rightarrow \mathcal{P}_1; \ \iota_1$, we can conclude by induction hypothesis that $\iota_1(T) \preceq \iota_1(type(?n))$. The fact that ι_1 is a well-typed instantiation permits to show (cf. Proposition 2.72) that $\iota_1(ctxt(?n)) \vdash \iota_1(t) : \iota_1(T)$ and thus, by cumulativity, that $\iota_1(ctxt(?n)) \vdash \iota_1(t) : \iota_1(type(?n))$. Therefore, $?n := \iota_1(t)$ is a typecorrect assignment.

- The validity of the resulting instantiation has to be ensured. This follows immediately from the definition of the predicate *valid*.

- The instantiation $\iota_1 \uplus \{?n := \iota_1(t)\}$ is defined (in the sense of Definition 2.65): By the precondition $\forall ?k \in MVars(T).?k \ll_{\mathcal{P}_0} ?n$, $dom(\iota_1)$ only contains metavariables $?k \ll_{\mathcal{P}_0} ?n$, thus $dom(\iota_1) \cap \{?n\} = \varnothing$ and by the definition of instantiation, for all $?m \in MVars(\iota_1(t))$, $\iota_1(?m) = ?m$.

□

There is a natural interpretation of the set of rules of Figures 4.1 and 4.2 as a unification algorithm. We will now show that this algorithm always terminates.

Proposition 4.11 (Termination of Unification Algorithm)
The unification algorithm obtained by applying the rules of Figure 4.1 (resp. Figure 4.2) backwards always terminates when invoked with a unification problem $\langle \mathcal{P}_0 \ ; \ \Gamma \vdash t_1 \stackrel{?}{\simeq} t_2 \rangle$ (a cumulative unification problem $\langle \mathcal{P}_0 \ ; \ \Gamma \vdash t_1 \stackrel{?}{\preceq} t_2 \rangle$).

Proof: As far as termination is concerned, this unification algorithm mainly differs from traditional first-order unification by the recursive "call" in rule (MV-term) that is required to unify the types of $?n$ and t in order to solve $?n \stackrel{?}{\simeq} t$. For showing termination, we define a measure as the lexicographic order on the quadruple $(\mid \mathcal{P} \mid,\ MVars(t_1) \cup MVars(t_2),\ \mid t_1 \mid + \mid t_2 \mid,\ j)$, where $\mid \mathcal{P} \mid$ is the number of metavariables of $\mathcal{P}$, $MVars(t_1) \cup MVars(t_2)$ is the set of metavariables occurring in t_1 and t_2, ordered by the multiset order induced by $\ll_{\mathcal{P}}$, $\mid t_1 \mid + \mid t_2 \mid$ is the sum of the term sizes of t_1 and t_2 and $j = 1$ if the unification problem under consideration is of the form $\langle \mathcal{P}\ ;\ \Gamma \vdash t_1 \stackrel{?}{\preceq} t_2 \rangle$ and $j = 0$ if it is of the form $\langle \mathcal{P}\ ;\ \Gamma \vdash t_1 \stackrel{?}{\simeq} t_2 \rangle$ (in order to take the rule (cu-term-term) into account). We verify that this measure decreases for the rules (MV-term) and (app-app), the other cases being similar:

- For (MV-term): by the side condition $\forall ?k \in MVars(T).?k \ll_{\mathcal{P}_0} ?n$ and the fact that $\forall ?k \in MVars(type_{\mathcal{P}_0}(?n)).?k \ll_{\mathcal{P}_0} ?n$, it can be concluded that $MVars(T) \cup MVars(type_{\mathcal{P}_0}(?n))$ is smaller with respect to the multiset order than $MVars(?n) \cup MVars(t)$. Therefore,
  $(\mid \mathcal{P}_0 \mid,\ MVars(T) \cup MVars(type_{\mathcal{P}_0}(?n)),\ \mid T \mid + \mid type_{\mathcal{P}_0}(?n) \mid,\ 0)$
  $< (\mid \mathcal{P}_0 \mid,\ MVars(?n) \cup MVars(t),\ \mid ?n \mid + \mid t \mid,\ 0)$

- For (app-app), we trivially have:
  $(\mid \mathcal{P}_0 \mid,\ MVars(f_1) \cup MVars(f_2),\ \mid f_1 \mid + \mid f_2 \mid,\ 0)$
  $< (\mid \mathcal{P}_0 \mid,\ MVars(f_1\ a_1) \cup MVars(f_2\ a_2),\ \mid (f_1\ a_1) \mid + \mid (f_2\ a_2) \mid,\ 0)$
  We have
  $(\mid \mathcal{P}_1 \mid,\ MVars(\iota_1(a_1)) \cup MVars(\iota_1(a_2)),\ \mid \iota_1(a_1) \mid + \mid \iota_1(a_2) \mid,\ 0)$
  $< (\mid \mathcal{P}_0 \mid,\ MVars(f_1\ a_1) \cup MVars(f_2\ a_2)\ \mid (f_1\ a_1) \mid + \mid (f_2\ a_2) \mid,\ 0)$
  since either $\mid \mathcal{P}_1 \mid < \mid \mathcal{P}_0 \mid$ or $\mid \mathcal{P}_1 \mid = \mid \mathcal{P}_0 \mid$, in which case it can shown by an easy argument that ι_1 must be the identity instantiation, thus
  $MVars(\iota_1(a_1)) \cup MVars(\iota_1(a_2)) \leq MVars(f_1\ a_1) \cup MVars(f_2\ a_2)$
  and
  $\mid \iota_1(a_1) \mid + \mid \iota_1(a_2) \mid = \mid a_1 \mid + \mid a_2 \mid < \mid (f_1\ a_1) \mid + \mid (f_2\ a_2) \mid$.

□

4.3.3. Higher-order unification

The unification algorithm of Section 4.3.2 has been stated for substitution-free metavariables only. This algorithm will fail for unification problems of the form $?n^\frown[y_1 := t_1, \ldots y_m := t_m] \stackrel{?}{\simeq} t$, so it is definitely not complete. As we will see further below, unification involving metavariables with substitutions

is equivalent to full higher-order unification. We will not attempt to treat higher-order unification in full generality here. Instead, we will sketch how unification problems for terms containing metavariables with substitutions can be translated to traditional higher-order unification problems. This opens the way to adapt standard higher-order unification algorithms to the problems dealt with here (however, see the discussion in Section 4.3.5). A special case of higher-order unification problems, involving only so-called patterns, will be examined in Section 4.3.4.

Higher-order unification is concerned with solving term equations modulo β-conversion. A simple structural comparison of terms is not sufficient, since β-reduction can completely change the structure of a term. For unifying two simply-typed λ-terms in $\beta\eta$ normal form, Huet [Hue75] has developed an algorithm, which we sketch here for later reference: common λ-abstractions are removed, that is, $\lambda\, x\, :\, T.s \stackrel{?}{\simeq} \lambda\, x\, :\, T.t$ is simplified to $s \stackrel{?}{\simeq} t$. Similarly, if both terms to be unified are applications whose head symbols are variables or constants (*rigid-rigid* equations), then the arguments are unified or unification fails, depending on whether the head symbols are equal or not. The fundamental difference with respect to first order unification is the treatment of applications where one of the heads (*flex-rigid, rigid-flex*) or both (*flex-flex*) are metavariables. Here, sequences of applications are not decomposed into function and argument, as in our (app-app) rule, but are handled 'en bloc'. Flex-rigid (and similarly rigid-flex) equations $(?F\ s_1, \ldots s_m) \stackrel{?}{\simeq} (g\ t_1, \ldots t_n)$ can be solved by:

- *imitation* of the form $\lambda\, x_1, \ldots x_p.(g\ (?H_1\ x_1, \ldots x_p), \ldots (?H_n\ x_1, \ldots x_p))$. Here, the $?H_i$ are new metavariables and p and the types of the $?H_i$ are determined by m, n and the types of $?F$ and g.

- *projection* of the form $\lambda\, x_1, \ldots x_p.\ (x_i\ (?H_1\ x_1, \ldots x_p), \ldots\ (?H_n\ x_1, \ldots x_p))$, with $1 \leq i \leq p$.

In both of the above cases, there may be several appropriate choices for p, so in general the tree constructed in search of a unifier has branching points with a finite number of successors. Flex-flex equations $(?Fs_1, \ldots s_m) \stackrel{?}{\simeq} (?G\ t_1, \ldots t_n)$ can always be solved by a uniform procedure, therefore the unification algorithm does not compute solutions for them, but leaves them as constraints for a postprocessing phase. In calculi in which the η rule does not hold, the algorithm becomes more complex as still more cases have to be taken into account.

The algorithm sketched above uses a functional encoding of scopes. For example, the fact that the new metavariables $?H_i$ created in the imitation and

projection steps possibly depend on the variables $x_1, \ldots x_p$ is indicated by the applications $(?H_i\ x_1, \ldots x_p)$.

The transformation given in Section 2.7.1 is the key to translating any equation $s \stackrel{?}{\simeq} t$ in which s or t contain metavariables with substitutions to an equation only containing substitution-free metavariables: Just substitute every $?n^\frown[y_1 := t_1, \ldots y_m := t_m]$ in s resp. t by its translation $(?F\ a_1 \ldots a_k)$, as specified in Definition 2.78, and then unify $\overline{s} \stackrel{?}{\simeq} \overline{t}$, following the above algorithm, for example.

Before formalizing this observation, let us note that when applying the inverse of the above translation (implicit in the proof of Proposition 2.83), we can transform any equation containing a metavariable in functional position of an application, as in $(?F\ a_1 \ldots a_k)$, to an equation in which metavariables (possibly with substitutions) only occur in non-functional position (cf. the treatment of higher-order patterns in Section 4.3.4). This approach has been taken in [DHK95].

The transformation of an equation $s \stackrel{?}{\simeq} t$ to an equation $\overline{s} \stackrel{?}{\simeq} \overline{t}$ according to the translation of Section 2.7.1 is a viable approach, but often too drastic: Usually, an explicit substitution attached to a metavariable $?n$ does not affect all the variables of $ctxt(?n)$, but only some of them, say up to some variable x_i. Thus, the metavariable term is of the form $?n^\frown[x_i := t_i, \ldots x_j := t_j]$. We will say that we *lift* such a term up to some variable x_i if we produce an instantiation for $?n$ that makes the substitution $[x_i := t_i, \ldots x_j := t_j]$ disappear. The lifted term contains a functional metavariable $?F$ that depends on all variables of $ctxt(?n)$ up to, but not including x_i. More precisely:

Definition 4.12 (Lift)
Let $?n$ be a metavariable with $ctxt(?n) = x_1 : T_1, \ldots, x_i : T_i, \ldots x_k : T_k$ and $type(?n) = T$. Then, $lift(?n, x_i)$ is defined as the pair $\langle ?F, \iota \rangle$, where $?F$ is a new metavariable with $ctxt(?F) = x_1 : T_1, \ldots, x_{i-1} : T_{i-1}$ and $type(?F) = \Pi x_i : T_i, \ldots x_k : T_k.\ T$ and ι is the instantiation $\{?n := (?F\ x_i, \ldots x_k)\}$.

The lift operation is related to the translation into functional representation of Section 2.7.1 as follows: If $ctxt(?n) = x_1 : T_1, \ldots x_k : T_k$ and $lift(?n, x_1) = \langle ?F, \iota \rangle$, and substitution σ does not contain $?n$, then $\overline{?n^\frown\sigma} = \iota(?n^\frown\sigma)$.

Lifting can be carried out during a preprocessing phase before unification. The introduction of superfluous metavariables resulting from consecutive applications of lifting to the same metavariable can then be avoided. More precisely, if $ctxt(?n_0) = x_1 : T_1, \ldots, x_i : T_i, \ldots, x_j : T_j, \ldots, x_k : T_k$ and $type(?n_0) = T$, then $lift(?n_0, x_j) = \langle ?n_1, \iota_1 \rangle$ and $lift(?n_1, x_i) = \langle ?n_2, \iota_2 \rangle$, with $\iota_1 = \{?n_0 := (?n_1\ x_j, \ldots x_k)\}$ and $\iota_2 = \{?n_1 := (?n_2\ x_i, \ldots x_{j-1})\}$. Instead of carrying out these lifts in two steps, one could directly apply $lift(?n_0, x_i) = \langle ?n, \iota \rangle$,

where $?n$ would have the same type and context as $?n_2$ and ι would be the composition of ι_1 and ι_2.

Example 4.13
Consider a term $?n^\frown[x_2 := (f\ x_1), x_3 := x_1]$, where $?n$ is defined by:

$$A, B : \mathit{Type}, f : A \to B, x_1 : A, x_2 : B, x_3 : A \vdash ?n : A$$

Then $\mathit{lift}(?n,\ x_2) = \langle ?F, \{?n :=?F\ x_2\ x_3\}\rangle$, where

$$A, B : \mathit{Type}, f : A \to B, x_1 : A \vdash ?F : \Pi x_2 : B, x_3 : A.\ A$$

Note that $(?n^\frown[x_2 := (f\ x_1), x_3 := x_1])\{?n :=?F\ x_2\ x_3\} =?F\ (f\ x_1)\ x_1$, thus contains no more explicit substitution.

$$\frac{\mathit{lift}(?n, y_1) = \langle ?F, \iota_1\rangle \quad \mathcal{P}_1 := \iota_1(\mathcal{P}_0) \cup \{?F\} \qquad \langle \mathcal{P}_1\ ;\ \iota_1(\Gamma) \vdash \iota_1(?n^\frown[y_1 := t_1, \ldots y_m := t_m]) \stackrel{?}{\simeq} \iota_1(t)\rangle \Rightarrow \mathcal{P}_2;\ \iota_2}{\langle \mathcal{P}_0\ ;\ \Gamma \vdash ?n^\frown[y_1 := t_1, \ldots y_m := t_m] \stackrel{?}{\simeq} t\rangle \Rightarrow \mathcal{P}_2;\ \iota_1 \uplus \iota_2}\ (\text{lift})$$

Figure 4.3.: Lift rule

When trying to unify $s \stackrel{?}{\simeq} t$, where s or t contains metavariables with explicit substitutions, we can apply the (lift) rule of Figure 4.3, possibly repeatedly if there are several such metavariables, to obtain terms s' and t' containing no more metavariables with explicit substitutions. The terms s' and t' can then submitted to a higher-order unification algorithm. Standard algorithms such as Huet's have to be adapted with care. We will not expand on this issue here – a discussion of possible pitfalls can be found in Section 4.3.5.

Example 4.14
Continuing with Example 4.13, assume that we want to solve the following unification problem: $?n^\frown[x_2 := (f\ x_1), x_3 := x_1] \stackrel{?}{\simeq} (f\ x_1)$. After lifting, we obtain the unification problem $?F\ (f\ x_1)\ x_1 \stackrel{?}{\simeq} (f\ x_1)$. A traditional higher-order unification algorithm yields, among others, the following solutions:

- Projection solution: $?F := \lambda\, u : B.\ \lambda\, v : A.\ u$. Composed with the solution $?n :=?F\ x_2\ x_3$, one obtains $?n := x_2$.
- Imitation solution: $?F := \lambda\, u : B.\ \lambda\, v : A.\ (f\ x_1)$. Composed with the solution for $?n$, one obtains $?n := (f\ x_1)$.

Invariants of the resulting unification algorithm are more difficult to state than in the first-order case, since new metavariables are generated and possibly instantiated during unification. Thus, if $\langle \mathcal{P}_0 \ ; \ \Gamma \vdash t_1 \stackrel{?}{\simeq} t_2 \rangle \Rightarrow \mathcal{P}_1; \ \iota_1$, then the domain and the solution terms of ι_1 may contain metavariables which are not in $\mathcal{P}_0$. One may be tempted to take into account the metavariables added during unification as follows: Assume $\mathcal{P}_1$ contains metavariables $?n_1, \ldots ?n_k$ not in $\mathcal{P}_0$. Then add $?n_1, \ldots ?n_k$ to $\mathcal{P}_0$ and claim that ι_1 is a valid and well-typed instantiation for this extension of $\mathcal{P}_0$. Unfortunately, metavariables added later during unification or during a proof may not be well-typed with respect to a previous proof problem.

Example 4.15
Assume $\mathcal{P}_0 = \{\Gamma \vdash ?n_0 : \mathit{Type}, \Gamma \vdash ?n_1 : ?n_0\}$ is a valid, well-typed proof problem. Then, $\iota = \{?n_0 := A\}$, with $A : \mathit{Type}$, is a type-correct instantiation for $\mathcal{P}_0$, and $\mathcal{P}_1 := \iota(\mathcal{P}_0) = \{?n_1 : A\}$. Assume that an (unspecified) proof rule adds a metavariable $?n_2 : P(?n_1)$ to $\mathcal{P}_1$, for $P : A \to \mathit{Prop}$. This can legally be done, the resulting proof problem $\{?n_1 : A, ?n_2 : P(?n_1)\}$ is valid and type-correct. However, adding $?n_2 : P(?n_1)$ to $\mathcal{P}_0$ would lead to an ill-typed proof problem, since in $\mathcal{P}_0$, $\Gamma \vdash ?n_1 : ?n_0$, thus $P(?n_1)$ is not well-typed.

The following proposition states invariants of the extended unification algorithm with a criterion that avoids these difficulties:

Proposition 4.16 (Invariants of Unification with Lift Rule)
Assume that $\langle \mathcal{P}_0 \ ; \ \Gamma \vdash t_1 \stackrel{?}{\simeq} t_2 \rangle$ is a unification problem ($\langle \mathcal{P}_0 \ ; \ \Gamma \vdash t_1 \stackrel{?}{\preceq} t_2 \rangle$ a cumulative unification problem).
If $\langle \mathcal{P}_0 \ ; \ \Gamma \vdash t_1 \stackrel{?}{\simeq} t_2 \rangle \Rightarrow \mathcal{P}_1; \ \iota_1$ (resp. $\langle \mathcal{P}_0 \ ; \ \Gamma \vdash t_1 \stackrel{?}{\preceq} t_2 \rangle \Rightarrow \mathcal{P}_1; \ \iota_1$) is derivable with the rules of Figures 4.1, 4.2 and 4.3, then:

- $\iota_1(t_1) \ \simeq \ \iota_1(t_2)$ (resp. $\iota_1(t_1) \preceq \iota_1(t_2)$)
- $\mathcal{P}_1$ is a valid, well-typed proof problem.
- For every $?m \in \mathcal{P}_0$, there is a derivation of $\iota_1(\mathit{ctxt}_{\mathcal{P}_0}(?m)) \vdash_{\mathcal{P}_1} \iota_1(?m) : \iota_1(\mathit{type}_{\mathcal{P}_0}(?m))$

Proof: Similar to the proof of Proposition 4.10. □

This proposition has the following limit case: Whenever the set of metavariables of $\mathcal{P}_1$ is contained in the set of metavariables of $\mathcal{P}_0$, then the validity and well-typedness of the resulting instantiation can be characterized more directly, as in Proposition 4.10, by $\mathcal{P}_1 = \iota_1(\mathcal{P}_0)$.

Corollary 4.17
If $\langle \mathcal{P}_0 \ ; \ \Gamma \vdash t_1 \stackrel{?}{\simeq} t_2 \rangle \Rightarrow \mathcal{P}_1; \ \iota_1$ (resp. $\langle \mathcal{P}_0 \ ; \ \Gamma \vdash t_1 \stackrel{?}{\preceq} t_2 \rangle \Rightarrow \mathcal{P}_1; \ \iota_1$) is derivable with the rules of Figures 4.1, 4.2 and 4.3 and $\mathcal{P}_1 \subseteq \mathcal{P}_0$, then ι_1 is a valid, well-typed instantiation of $\mathcal{P}_0$.

With minor modifications, the termination proof of Proposition 4.11 carries over to the system of rules of Figure 4.1, 4.2 and 4.3. Instead of a lexicographic order on a quadruple as defined there, we define a lexicographic order on a quintuple $(\mid \mathcal{P} \mid, \ \mid\mid \mathcal{P} \mid\mid, \ \mathit{MVars}(t_1) \cup \mathit{MVars}(t_2), \ \mid t_1 \mid + \mid t_2 \mid, \ j)$, where $\mid\mid \mathcal{P} \mid\mid := \sum_{?n \in \mathcal{P}} \mathit{length}(\mathit{ctxt}(?n))$ adds up the lengths of the contexts of the metavariables in $\mathcal{P}$. Lifting a metavariable $?n$ leaves the total number of metavariables unchanged, but introduces a metavariable $?F$ instead of $?n$ with $\mathit{length}(\mathit{ctxt}(?F)) < \mathit{length}(\mathit{ctxt}(?n))$.

4.3.4. Unification of Higher-Order "Patterns"

As already mentioned in Section 2.7, most existing proof assistants which have a notion of metavariables use a functional encoding of scopes, as opposed to a dependence on contexts, as in our case. In particular, whenever an existential variable in the scope of universal variables is converted to a metavariable, as in $\forall\, a : A. \exists\, x : A. (P\ a) \to (P\ x)$ (compare with Example 2.77 and the following remarks), a "raising" step is performed which turns the existential variable into a metavariable of functional type and the universal variables into its arguments, as in $(P\ a) \to (P\ (?x\ a))$.

It has been observed by Miller [Mil91] that this raising operation only generates terms of a certain form, so-called *patterns*, and that unification for these patterns resembles first-order unification in that unification is decidable and most general unifiers exist. In the framework of a simply-typed calculus, as examined in [Mil91], a pattern is a term in which metavariables $?F$ only occur in subterms of the form $?F\ v_1 \ldots v_n$ (with $n \geq 0$), where the v_i are distinct variables. For example, $(g\ (?F_1\ x\ y)\ ?F_2)$ is a pattern, but not $(?F\ (\lambda\, x.y))$ or $(?F_1\ (?F_2\ x))$. The notion of pattern directly carries over to the Calculus of Constructions.

It is easy to see that the translation of terms t into functional representation $\overline{t}$ of Section 2.7.1 only produces patterns when applied to terms t that only contain metavariables of the form $?n^\frown[\,]$, i.e. without an explicit substitution. The unification procedure of Section 4.3.2 above is essentially specialized to terms of this form. As will be seen below, it turns out that in the converse direction, patterns describe a slightly more general class of terms than our metavariables of the form $?n^\frown[\,]$. However, it may be questioned whether this increased generality is ever exploited to a significant extent, that is, whether

patterns not in the image of our translation function $t \mapsto \overline{t}$ (for terms t with substitution-free metavariables) arise in practice very often.

The interest of this section is to take a closer look at the correspondence between patterns and certain terms of the calculus with explicit substitutions and to investigate the consequences for unification. In [DHKP96], a unification algorithm for the simply-typed explicit substitution calculus $\lambda\sigma$ [ACCL91] is given. Pfenning [Pfe91b] describes pattern unification in the Calculus of Constructions, however not for a calculus with explicit substitutions. The algorithm in [Pfe91b] is incomplete because it cannot deal with unification problems like $(?F\ A) \stackrel{?}{\simeq} (A \to A)$. Neither is completeness achieved with the rules presented below, although for different reasons. Some of the obstacles to obtaining completeness are discussed in Section 4.3.5.

Definition 4.18 (Pattern)
A *pattern metavariable* is a metavariable of the form $?n^\frown[x_1 := v_1, \ldots x_n := v_n]$, where $v_i \notin dom(ctxt(?n))$ for all $v_i \in \{v_1, \ldots v_n\}$ and all v_i are mutually distinct.

A *pattern* is a term all of whose metavariable occurrences are pattern metavariables.

Patterns in the standard sense can easily be converted to patterns according to the above definition: If $?f\ v_1 \ldots\ v_k$ is such a standard pattern, where $\vdash ?f : \Pi x_1 : T_1 \ldots \Pi x_m : T_m.T$, then $?f$ can be instantiated with $\lambda\, x_1 : T_1, \ldots x_k : T_k.?n$, where $?n$ is defined by $x_1 : T_1, \ldots x_k : T_k \vdash ?n : \Pi x_{k+1} : T_{k+1} \ldots \Pi x_m : T_m.T$, which, after reduction, yields a pattern metavariable $?n^\frown[x_1 := v_1, \ldots x_k := v_k]$. The v_i differ from the x_j, and from the fact that $?f\ v_1 \ldots\ v_k$ is a pattern, it follows that the v_i are mutually different, so $?n^\frown[x_1 := v_1, \ldots x_k := v_k]$ is indeed a pattern in the above sense.

Conversely, when subjected to the translation of Section 2.7.1, a pattern as defined above yields a pattern in the standard sense. This is not difficult to see for most of the term constructors, so we only examine the translation $\overline{?n^\frown[x_1 := v_1, \ldots x_k := v_k]}$ of a pattern metavariable. If $ctxt(?n) = y_1 : T_1, \ldots y_n : T_n$ and $type(?n) = T$, and $x_1, \ldots x_k$ are among $y_1, \ldots y_n$, then for a metavariable $\vdash ?F : \Pi y_1 : T_1, \ldots y_n : T_n.T$, we obtain $\overline{?n^\frown[x_1 := v_1, \ldots x_k := v_k]}$ $=?F\ a_1 \ldots a_n$, where each a_i is either one of the v_i or a variable $y \in \{y_1, \ldots, y_n\} \setminus \{x_1, \ldots, x_k\}$. Each v_i differs from all variables of $ctxt(?n)$, by condition $v_i \notin dom(ctxt(?n))$. Since all v_i and all y_i are mutually distinct, we can conclude that in $?F\ a_1 \ldots a_n$, the a_i are mutually distinct, so this term indeed is a pattern in the standard sense.

Since in pattern metavariable $?n^\frown\sigma$, the substitution $\sigma = \{x_1 := v_1, \ldots x_n := v_n\}$ in fact defines a renaming of variables, i.e. a one-to-one mapping from vari-

ables x_i to variables v_i, we can define a substitution $\sigma^{-1} \mathrel{\widehat{=}} \{v_1 := x_1, \ldots v_n := x_n\}$. A unification problem of the form $\Gamma \vdash ?n^\frown\sigma \stackrel{?}{\simeq} t$ can then be reduced to $\Gamma\sigma^{-1} \vdash ?n \stackrel{?}{\simeq} t\sigma^{-1}$, which can be described by the following rule:

$$\frac{\langle \mathcal{P}_0 \ ; \ \Gamma\sigma^{-1} \vdash ?n \stackrel{?}{\simeq} t\sigma^{-1} \rangle \Rightarrow \mathcal{P}_1; \ \iota_1}{\langle \mathcal{P}_0 \ ; \ \Gamma \vdash ?n^\frown\sigma \stackrel{?}{\simeq} t \rangle \Rightarrow \mathcal{P}_1; \ \iota_1} \text{ (pattern)}$$

Even though it is obvious that this rule leads to structurally equal terms, it is not evident that the unification problem $\langle \mathcal{P}_0 \ ; \ \Gamma\sigma^{-1} \vdash ?n \stackrel{?}{\simeq} t\sigma^{-1} \rangle$ fulfills the precondition of Definition 4.6, viz. that the terms $?n$ and $t\sigma^{-1}$ are well-typed in context $\Gamma\sigma^{-1}$ (which again implies that the context $\Gamma\sigma^{-1}$ is valid at all). We conjecture that this is indeed the case, but make no attempt to prove this here. As will be discussed below, this fact is not trivial, and it depends on the metavariable $?n^\frown\sigma$ being a pattern metavariable. For substitutions $\{x_1 := v_1, \ldots x_k := v_k\}$ which violate the requirement $v_i \notin dom(ctxt(?n))$, even if they define a bijection between variables, ill-typed terms may result, as shown by Example 4.19 below.

4.3.5. Discussion

The unification rules presented in the preceding sections are "modular" in that they can be combined to approximate completeness to varying degrees. A question that comes to mind is why no complete method is presented that subsumes the individual methods given so far. In the following, we discuss some of the reasons why a complete procedure is hard to achieve. In particular, they indicate that a complete general-purpose procedure would be of no practical use in a theorem proving environment. Therefore, composing a unification algorithm of building blocks that can be tailored to specific needs is of greater interest.

It is well known (see for example [Hue75]) that already in the simply-typed λ-calculus,

- unification is undecidable
- no most general unifiers exist, i.e., there are terms t_1, t_2 such that for each unifier ι of t_1 and t_2, there is a more general unifier ι'.

Given these facts, it is nevertheless possible to define algorithms, such as Huet's [Hue75], which enumerate a possibly infinite *complete set of unifiers*, that is, a set C of instantiations which is correct (for all $\iota \in C$, $\iota(t_1) \simeq \iota(t_2)$) and complete in that any unifier ι' is an instance of a $\iota \in C$.

In the simply-typed λ-calculus, the type level and the term level are clearly separated. This has, among others, the following consequences for unification algorithms:

- The type of terms is not affected by instantiations. When unifying two terms, it can be tested in advance whether their type is identical. If this is so, the algorithm proceeds by structural manipulations, as sketched in Section 4.3.3, relying on fixed type information. In dependently typed calculi, some seemingly obvious structural manipulations may lead to type-incorrect instantiations. This problem is illustrated by Example 4.19.

- In dependently-typed calculi, it may be necessary to unify types, and not only to compare them for testing whether they are identical. Since there are functions yielding types as result, there is a much greater number of cases to be considered than in the simply-typed calculus (see Example 4.20).

- There is possibly no upper bound on the number of arguments a function may have. A unification procedure in the style of Huet's algorithm, as described in Section 4.3.3, would in some cases have to build an infinitely branching search tree, as in Example 4.21.

The following example illustrates the problem of dependent typing in unification:

Example 4.19
Consider the problem of unifying $A : \mathit{Type}, g : A \to A \vdash ?n^\frown[T := A] \stackrel{?}{\simeq} \lambda\, x : A.(g\ x)$. An instantiation that comes to mind is $?n := \lambda\, x : T.\ (g\ x)$, which corresponds to the solution suggested by the (pattern) rule of Section 4.3.4. Whether this instantiation is correct or not depends on the definition of metavariable $?n$.

- Suppose $?n$ is defined by $T : \mathit{Type}, g : T \to T \vdash ?n : T \to T$, then the instantiation is correct. The context $A : \mathit{Type}, g : A \to A$ of $?n^\frown[T := A]$ may for example result from a derivation in which the original $T : \mathit{Type}, g : T \to T \vdash ?n : T \to T$ is first weakened to $A : \mathit{Type}, T : \mathit{Type}, g : T \to T \vdash ?n : T \to T$ and then reduced by rule (MV-β-Red) to $A : \mathit{Type}, g : A \to A \vdash ?n^\frown[T := A] : A \to A$, which also gives a correct typing to $\lambda\, x : A.\ (g\ x)$. In this case, the metavariable $?n^\frown[T := A]$ is indeed a pattern metavariable.

- Suppose $?n$ is defined by $A : \mathit{Type}, T : \mathit{Type}, g : A \to A \vdash ?n : A \to A$. In this case, the instantiation $?n := \lambda\, x : T.\ (g\ x)$ is not typecorrect, since

g expects an argument of type A and not type T. Note that we can again derive $A : \mathit{Type}, g : A \to A \vdash ?n^\frown[T := A] : A \to A$ by an application of (MV-β-Red), but this time, $?n^\frown[T := A]$ is no pattern metavariable, since $A \in \mathit{dom}(\mathit{ctxt}(?n))$.

Even though this example is taken from a polymorphic calculus, a similar problem can also be stated in a calculus with type dependencies on terms, but not on types, as for example LF, for which Elliot [Ell89] and Pym [Pym90] have devised unification algorithms. The (almost identical) algorithms [Ell89] and [Pym90] use metavariables without explicit substitutions, but with the aid of the lifting operation described in Section 4.3.3, an adaptation of their algorithms to our calculus should be possible.

Even though [Ell89] and [Pym90] claim completeness of their algorithms, just as [Pfe91b] for pattern unification in the Calculus of Constructions, their approaches do not deal with cases as the following. Since proofs of these claims are direct adaptations of [Hue75], it is difficult to assess how the completeness statement should be interpreted. Probably, a tacit assumption of [Ell89] and [Pym90] is that metavariables only stand for elements of types (that is, $?n : A$ with $A : \mathit{Type}$), but not for types or type constructors (that is, $?n : A_1 \to \ldots \to \mathit{Type}$).

Example 4.20
In many calculi of the λ-cube, functions can be defined whose range are propositions or types. Assume we want to unify $T : \mathit{Type}, x : T, P : T \to \mathit{Prop} \vdash ?f\ x \stackrel{?}{\simeq} (P\ x) \to (P\ x)$, with $T : \mathit{Type}, x : T, P : T \to \mathit{Prop} \vdash ?f : T \to \mathit{Type}$. Obviously, $?f := \lambda z : T.\ (P\ z) \to (P\ x)$ and $?f := \lambda z : T.\ (P\ z) \to (P\ z)$ are typecorrect solutions to this problem. Notice that Huet's algorithm does not apply here, since a combination of outermost term constructors such as flexible application – implication (strictly speaking, Π-abstraction) does not arise in the simply-typed λ-calculus.

The above two examples describe situations that occur in all dependently typed calculi. When a metavariable $?A$ can stand for types and not only elements of types, there is an additional difficulty: The number of Π-abstractions of a solution of $?A$ is not bounded, and neither is the number of arguments a function of type $?A$ has.

Example 4.21
Assume Γ is a valid context, and the proof problem $\mathcal{P}$ is given by metavariables $\Gamma \vdash ?A : \mathit{Type}$ and $\Gamma \vdash ?f : ?A \to ?A$. The unification problem we want to solve is $\langle \mathcal{P}\ ;\ \Gamma \vdash (?f\ a) \stackrel{?}{\simeq} b \rangle$, where a and b are two terms such that $\Gamma \vdash a : ?A$ and $\Gamma \vdash b : ?A$. In its "match" phase, an adaptation of Huet's algorithm would have

to generate, among others, all of the following "projection" instantiations for $?f$ (for simplicity, we do not give the more general dependently-typed solution terms):

- $\iota_0 \mathrel{\widehat{=}} \{?f := \lambda\, z :?A.\ z\}$
- $\iota_1 \mathrel{\widehat{=}} \{?f := \lambda\, z :?A_1 \to ?A'.\ (z\ ?a_1), ?A :=?A_1 \to ?A'\}$ with $\Gamma \vdash ?A_1 : \mathit{Type}$, $\Gamma \vdash ?A' : \mathit{Type}$ and $\Gamma \vdash ?a_1 :?A_1$.
- $\iota_2 \mathrel{\widehat{=}} \{?f := \lambda\, z :?A_2 \to ?A_1 \to ?A'.\ (z\ ?a_1\ ?a_2), ?A :=?A_2 \to ?A_1 \to ?A'\}$, with $\Gamma \vdash ?A_2 : \mathit{Type}$, ..., $\Gamma \vdash ?a_2 :?A_2$.
- and so forth.

The $\iota_1, \iota_2, \ldots$ are mutually independent, i.e. none of these instantiations can be represented as an instance of the other ones.

4.4. Tableau-Style Proof Search

In Chapter 3, it has been argued that the natural deduction presentation ECC_N of the calculus ECC is unsuitable for proof search, and that a sequent style calculus, baptized ECC_G, is more appropriate because it permits a structural decomposition of formulae. However, for practical concerns, even the calculus ECC_G is not a good choice, mainly for two reasons:

- The language characterized by ECC_G is very expressive, and consequently, inferences in this language can be expected to be quite expensive. In spite of the subformula property of ECC_G, some of its rules generate a search space that is hard to keep under control, for example formation rules such as (Π-Form$_1$), which are applicable to goals of the form $\Gamma \vdash ?P : \mathit{Prop}$. The utility of these rules for a particular proof problem has been demonstrated in Example 4.2, but since this kind of problem does not arise often in practice, these rules should not be available for general proof search, but be relegated to specialized procedures.
- The language is very minimalistic, imposing an awkward encoding of natural notions such as conjunction, disjunction, existence etc. As mentioned in Section 2.1.4, the usual logical connectives can be expressed in ECC, and a proof search calculus should provide appropriate rules for manipulating them.

In the following, we will examine a calculus resembling the usual predicate logic tableau calculi. In Section 4.4.1, we will first give its rules and

discuss differences with respect to standard presentations of tableau calculi. In Section 4.4.2, it will be shown that the mechanisms developed in Chapter 2 can cope with the eigenvariable provisos of sequent calculi and thus offer an alternative to Skolemization. The raw form of the calculus is amenable to optimizations (Section 4.4.3), which brings an implementation into the realm of dedicated theorem provers, in spite of some insufficiencies that persist (cf. the discussion in Section 5.2).

Even though equational and inductive reasoning are interesting topics which raise particular problems, especially in polymorphic calculi (see [Sor96]), they will not be considered here.

4.4.1. Sequent Calculus Rules

In Section 2.1.4, an encoding of the standard connectives of predicate logic in the calculus *ECC* has been given, and it has been verified that this encoding is adequate in the sense that the usual introduction and elimination rules of a natural deduction calculus can be shown to hold. Figure 4.4 lists the corresponding sequent calculus rules. As usual, these rules are supposed to be applied backwards, starting with a judgement $\Gamma \vdash ?n : G$, where G is the goal formula to be proved under hypotheses Γ. Each backwards application of a rule gives rise to an instantiation of the current metavariable $?n_0$ with a proof term containing the metavariables of the new subgoals, as displayed in Figure 4.5. These proof terms use the definitions of proof elements of Figure 4.6, which are shown together with their types. In order to keep proof terms small, we will omit type arguments as long as these can be synthesized from the remaining arguments (compare with Example 4.3 to see how this is done). For example, we usually write $(andI\ p_A\ p_B)$ instead of $(andI\ A\ B\ p_A\ p_B)$.

The rules marked with an asterisk – $(FalseL)$, $(\vee L)$, $(\neg L)$, $(\exists\, L)$ – are subject to the proviso that the goal formula G is of type *Prop*, i.e. $ctxt(?n_0) \vdash G : Prop$, for a reason to be explained in a moment. However, the eigenvariable condition need not be enforced explicitly. A discussion of this question is deferred to Section 4.4.2.

A statement of *correctness* of the rules of Figure 4.4 can be given in a similar fashion as for the rules of the calculus ECC_G in Section 3.2.1, and a proof of correctness proceeds along the same lines, once the definitions of the connectives have been expanded. As an example, we now give a derivation of the proof term of rule $(\neg L)$ (surrounding contexts Γ, Γ' are omitted for better readability), which also elucidates the requirement that G is of type *Prop*. Derivations of the remaining rules can be found in [Wag95].

$$\dfrac{\dfrac{\dfrac{}{p : \neg P \vdash p : \neg P}\,(\text{var})}{p : \neg P \vdash p : (P \to \Pi X : Prop.X)} \quad p : \neg P \vdash ?n_1 : P}{p : \neg P \vdash (p\ ?n_1) : (\Pi X : Prop.X)}\,(\text{app}) \quad p : \neg P \vdash G : Prop}{p : \neg P \vdash (p\ ?n_1\ G) : G}\,(\text{app})$$

The double lines in this derivation symbolize an unfolding of definitions: $\neg P \equiv P \to False \equiv P \to \Pi X : Prop.X$.

The Right rules of the logical connectives are standard and deserve no special mention, but there are some peculiarities about the other rules setting them off from standard sequent calculus rules as found, for example, in [Gal87]:

- The Γ and Γ' occurring in the rules are valid contexts and not only sets of hypotheses. Thus, apart from formulae, they contain variable declarations. In order to preserve well-typing, the variables x and y of rules $(\forall\, R)$ and $(\exists\, L)$ cannot simply be fixed as "constants", but have to be added to the context explicitly.

- Even though certain antecedent formulae in the premiss of some rules are "redundant" (such as $p : A \wedge B$ in $(\wedge L)$), they cannot simply be dropped, because this might lead to ill-typed contexts or terms. In Section 4.4.3, we make this notion of redundancy more precise and show how to deal with it.

- The rules for the quantifier $\forall$ are directly obtained from the (ΠL) and (ΠR) rules of Section 3.1.2. Similarly, the rules for implications are just the non-dependent versions of the Π rules, added for comparison with standard presentations of sequent calculi.

- The rules $(\exists\, R)$ and $(\forall\, L)$ have two premisses, one requiring the construction of a witness $?n_1$, the other, depending on $?n_1$, pursuing the proof of the goal formula. When the rules are operationalized for proof search, it is reasonable to delay the subgoal $?n_1$, with the intention of finding a solution through unification in the branch of subgoal $?n_2$. However, if a proof does not lead to an instantiation of $?n_1$, search has to be resumed for subgoal $?n_1$ (cf. Example 4.23).

- The (axiom) rule makes appeal to unification: x is acceptable as solution for $?n_0$, if x and $?n_0$ can be unified, i.e. if $\Gamma, x : T_1, \Gamma' \vdash x \stackrel{?}{\simeq} ?n_0$, using unification rules as those presented in Section 4.3. By unification rule (MV-term), this again leads to a (cumulative) unification of the types

$$\frac{\Gamma, x : T_1, \Gamma' \vdash x \stackrel{?}{\simeq} ?n_0}{\Gamma, x : T_1, \Gamma' \vdash ?n_0 : T_2}\ (\text{axiom})$$

$$\frac{}{\Gamma \vdash ?n_0 : \mathit{True}}\ (\mathit{TrueR}) \qquad \frac{}{\Gamma, p : \mathit{False}, \Gamma' \vdash ?n_0 : G}\ (\mathit{FalseL})^*$$

$$\frac{\Gamma \vdash ?n_1 : A \quad \Gamma \vdash ?n_2 : B}{\Gamma \vdash ?n_0 : A \wedge B}\ (\wedge R)$$

$$\frac{\Gamma, p : A \wedge B, \Gamma', p_A : A, p_B : B \vdash ?n_1 : G}{\Gamma, p : A \wedge B, \Gamma' \vdash ?n_0 : G}\ (\wedge L)$$

$$\frac{\Gamma \vdash ?n_1 : A}{\Gamma \vdash ?n_0 : A \vee B}\ (\vee Rl) \qquad \frac{\Gamma \vdash ?n_1 : B}{\Gamma \vdash ?n_0 : A \vee B}\ (\vee Rr)$$

$$\frac{\Gamma, p : A \vee B, \Gamma', p_A : A \vdash ?n_1 : G \qquad \Gamma, p : A \vee B, \Gamma', p_B : B \vdash ?n_2 : G}{\Gamma, p : A \vee B, \Gamma' \vdash ?n_0 : G}\ (\vee L)^*$$

$$\frac{\Gamma, x : A \vdash ?n_1 : B}{\Gamma \vdash ?n_0 : A \to B}\ (\to R)$$

$$\frac{\Gamma, p : A \to B, \Gamma' \vdash ?n_1 : A \qquad \Gamma, p : A \to B, \Gamma', p' : B \vdash ?n_2 : G}{\Gamma, p : A \to B, \Gamma' \vdash ?n_0 : G}\ (\to L)$$

$$\frac{\Gamma, p : A \vdash ?n_1 : \mathit{False}}{\Gamma \vdash ?n_0 : \neg A}\ (\neg R) \qquad \frac{\Gamma, p : \neg P, \Gamma' \vdash ?n_1 : P}{\Gamma, p : \neg P, \Gamma' \vdash ?n_0 : G}\ (\neg L)^*$$

$$\frac{\Gamma \vdash ?n_1 : T \qquad \Gamma \vdash ?n_2 : P[?n_1]}{\Gamma \vdash ?n_0 : \exists\, x : T.P[x]}\ (\exists\, R)$$

$$\frac{\Gamma, p : \exists\, x : T.P[x], \Gamma', y : T, p' : P[y] \vdash ?n_1 : G}{\Gamma, p : \exists\, x : T.P[x], \Gamma' \vdash ?n_0 : G}\ (\exists\, L)^*$$

$$\frac{\Gamma, x : T \vdash ?n_1 : P[x]}{\Gamma \vdash ?n_0 : \forall\, x : T.P[x]}\ (\forall\, R)$$

$$\frac{\begin{array}{c}\Gamma, p : \forall\, x : T.P[x], \Gamma' \vdash ?n_1 : T \\ \Gamma, p : \forall\, x : T.P[x], \Gamma', p' : P[?n_1] \vdash ?n_2 : G\end{array}}{\Gamma, p : \forall\, x : T.P[x], \Gamma' \vdash ?n_0 : G}\ (\forall\, L)$$

Figure 4.4.: Rules of the Tableau calculus

(axiom)	$?n_0 := x$
$(TrueR)$	$?n_0 := \lambda\, X : Prop.\ \lambda\, x : X.\ x$
$(FalseL)$	$?n_0 := p\ \ G$
$(\wedge R)$	$?n_0 := andI(A, B, ?n_1, ?n_2)$
$(\wedge L)$	$?n_0 := ?n_1\frown[p_a := andEl(A, B, p), p_b := andEr(A, B, p)]$
$(\vee Rl)$	$?n_0 := orIl(A, B, ?n_1)$
$(\vee Rr)$	$?n_0 := orIr(A, B, ?n_1)$
$(\vee L)$	$?n_0 := orE(A, B, G,\ \lambda\, p_a : A.?n_1,\ \lambda\, p_b : B.?n2,\ p)$
$(\rightarrow R)$	$?n_0 := \lambda\, x : A.\ ?n_1$
$(\rightarrow L)$	$?n_0 := ?n_2\frown[p' := (p\ ?n_1)]$
$(\neg R)$	$?n_0 := \lambda\, p : A.\ ?n_1$
$(\neg L)$	$?n_0 := p\ ?n_1\ \ G$
$(\exists\, R)$	$?n_0 := exI(T, P, ?n_1, ?n_2)$
$(\exists\, L)$	$?n_0 := p\ \ G\ (\lambda\, y : T.\lambda\, p' : P[y].?n_1)$
$(\forall\, R)$	$?n_0 := \lambda\, x : T.\ ?n_1$
$(\forall\, L)$	$?n_0 := ?n_2\frown[p' := (p\ ?n_1)]$

Figure 4.5.: Solutions associated with the Tableau rules

of x and $?n_0$, so the type T_1 of x and T_2 of $?n_0$ will be compared with respect to $\preceq$.

In the rules of Figure 4.4, the information about proof problems remains implicit. In analogy to unification, we can restate the rules in a somewhat less readable, but more precise form as transformation rules which convert a proof problem $\mathcal{P}_0$ to a new proof problem $\mathcal{P}_1$ and an instantiation ι_1. By this means, we can relate the type correctness of rules, as stated above, to the notion of well-typed instantiation as given by Definition 2.71 and by this means arrive at a formal correctness criterion for the proof terms that are constructed during proof search.

The proof transformation relation $\mathcal{P}_0 \Longrightarrow \mathcal{P}_1;\ \iota_1$ can be obtained by a canonical translation of the rules of Figure 4.4. The transformation rules for (axiom), $(\wedge R)$ and $(\wedge L)$ are displayed in Figure 4.7, the remaining rules should then be obvious. Side conditions of the transformation rules are indented, disjoint set union is denoted by $\dot\cup$.

Based on the relation $\Longrightarrow$, the relation $\Longrightarrow^*$ is defined inductively by:

$$
\begin{array}{lcl}
\mathit{andI} & := & \lambda\, A, B : \mathit{Prop}, a : A, b : B. \\
 & & \lambda\, R : \mathit{Prop}.\, \lambda\, h : (A \to B \to R).(h\ a\ b) \\
 & : & \Pi A, B : \mathit{Prop}.\ A \to B \to A \wedge B \\
\mathit{andEl} & := & \lambda\, A, B : \mathit{Prop}, p : A \wedge B.(p\ A\ (\lambda\, p_1 : A, p_2 : B.p_1)) \\
 & : & \Pi A, B : \mathit{Prop}.\ A \wedge B \to A \\
\mathit{andEr} & := & \lambda\, A, B : \mathit{Prop}, p : A \wedge B.(p\ B\ (\lambda\, p_1 : A, p_2 : B.p_2)) \\
 & : & \Pi A, B : \mathit{Prop}.\ A \wedge B \to B \\
\mathit{orIl} & := & \lambda\, A, B : \mathit{Prop}, a : A. \\
 & & \lambda\, R : \mathit{Prop}, f_1 : A \to R, f_2 : B \to R.(f_1\ a) \\
 & : & \Pi A, B : \mathit{Prop}.\ A \to A \vee B \\
\mathit{orIr} & := & \lambda\, A, B : \mathit{Prop}, b : B. \\
 & & \lambda\, R : \mathit{Prop}, f_1 : A \to R, f_2 : B \to R.(f_2\ b) \\
 & : & \Pi A, B : \mathit{Prop}.\ B \to A \vee B \\
\mathit{orE} & := & \lambda\, A, B, G : \mathit{Prop}, f_1 : A \to G, f_2 : B \to G. \\
 & & \lambda\, p : A \vee B.(p\ G\ f_1\ f_2) \\
 & : & \Pi A, B, G : \mathit{Prop}.\ (A \to G) \to (B \to G) \to (A \vee B) \to G \\
\mathit{exI} & := & \lambda\, T : \mathit{Type}, P : T \to \mathit{Prop}, x : T, p : P(x). \\
 & & \lambda\, R : \mathit{Prop}.\, \lambda\, h : (\Pi y : T.\ (P(y) \to R)).(h\ x\ p) \\
 & : & \Pi\, T : \mathit{Type}, P : T \to \mathit{Prop}, x : T, p : P(x).\ \exists\, x : T.P(x)
\end{array}
$$

Figure 4.6.: Proof elements of logical connectives and quantifiers

- $\mathcal{P}_0 \Longrightarrow^* \mathcal{P}_0;\ \{\}$, where $\{\}$ is the identity instantiation.
- $\mathcal{P}_0 \Longrightarrow^* \mathcal{P}_{n+1};\ \iota_{n+1}$ if there is a proof problem $\mathcal{P}_n$ and an instantiation ι_n such that $\mathcal{P}_0 \Longrightarrow^* \mathcal{P}_n;\ \iota_n$ and $\mathcal{P}_n \Longrightarrow \mathcal{P}_{n+1};\ \iota'$ and $\iota_{n+1} = \iota_n \uplus \iota'$.

During proof search, new metavariables are generated, so some care has to be taken when stating a correctness criterion for the proof rules (cf. the discussion in Section 4.3.3, in particular the remarks preceding Proposition 4.16).

Proposition 4.22 (Correctness of Proof Transformations)
Assume $\mathcal{P}_0$ is a well-typed proof problem. If $\mathcal{P}_0 \Longrightarrow \mathcal{P}';\ \iota'$ and $\mathcal{P}_0 \Longrightarrow^* \mathcal{P}_n;\ \iota_n$, then:

- $\mathcal{P}'$ and $\mathcal{P}_n$ are well-typed proof problems

$$\begin{array}{ll} & \mathcal{P} \dot\cup \{\Gamma, x : T_1, \Gamma' \vdash ?n_0 : T_2\} \\ & \quad \langle \mathcal{P} \ ; \ \Gamma, x : T_1, \Gamma' \vdash x \stackrel{?}{\simeq} ?n_0 \rangle \Rightarrow \mathcal{P}_1; \ \iota_1 \\ \Longrightarrow & \mathcal{P}_1; \ \iota_1 \end{array}$$

$$\begin{array}{ll} & \mathcal{P} \dot\cup \{\Gamma \vdash ?n_0 : A \wedge B\} \\ & \quad \iota_1 := \{ \ ?n_0 := andI(A, B, ?n_1, ?n_2) \ \} \\ \Longrightarrow & \iota_1(\mathcal{P} \cup \{\Gamma \vdash ?n_1 : A, \ \Gamma \vdash ?n_2 : B\}); \ \iota_1 \end{array}$$

$$\begin{array}{ll} & \mathcal{P} \dot\cup \{\Gamma, p : A \wedge B, \Gamma' \vdash ?n_0 : G\} \\ & \quad \iota_1 := \{ \ ?n_0 := ?n_1^\frown[p_a := andEl(A, B, p), p_b := andEr(A, B, p)] \ \} \\ \Longrightarrow & \iota_1(\mathcal{P} \cup \{\Gamma, p : A \wedge B, \Gamma', p_A : A, p_B : B \vdash ?n_1 : G\}); \ \iota_1 \end{array}$$

Figure 4.7.: Rules of Proof Transformation System

- For every $?m \in \mathcal{P}_0$, there are derivations of:
  $\iota'(ctxt_{\mathcal{P}_0}(?m)) \vdash_{\mathcal{P}'} \iota'(?m) : \iota'(type_{\mathcal{P}_0}(?m))$ and
  $\iota_n(ctxt_{\mathcal{P}_0}(?m)) \vdash_{\mathcal{P}_n} \iota_n(?m) : \iota_n(type_{\mathcal{P}_0}(?m))$

Proof: For the statements about relation $\Longrightarrow$, the proof is by induction on the generation of $\Longrightarrow$, and for the statements about relation $\Longrightarrow^*$, the proof is by induction on n.

As to relation $\Longrightarrow$, the statement for the transformation corresponding to the (axiom) proof rule follows directly from similar properties of the relation $\Rightarrow$ of the unification algorithm (see Propositions 4.10 and 4.16). For the other proof rules, the statement follows from the typecorrectness of the proof terms and Proposition 2.72.

As to relation $\Longrightarrow^*$: The base case for $n = 0$ is trivial. Assume that the properties stated in the proposition have been shown for $\mathcal{P}_0 \Longrightarrow^* \mathcal{P}_n; \ \iota_n$, so $\mathcal{P}_n$ is a well-typed proof problem and $\iota_n(ctxt_{\mathcal{P}_0}(?m)) \vdash_{\mathcal{P}_n} \iota_n(?m) : \iota_n(type_{\mathcal{P}_0}(?m))$ is well-typed. Then, by the definition of $\Longrightarrow^*$ and the properties of $\Longrightarrow$, also $\mathcal{P}_{n+1}$ is a well-typed proof problem. Furthermore:
$\iota_n(ctxt_{\mathcal{P}_0}(?m)) \vdash_{\mathcal{P}_n} \iota_n(?m) : \iota_n(type_{\mathcal{P}_0}(?m))$ for all $?m \in \mathcal{P}_0$
(by induction hypothesis) entails:
$\iota'(\iota_n(ctxt_{\mathcal{P}_0}(?m))) \vdash_{\mathcal{P}_{n+1}} \iota'(\iota_n(?m)) : \iota'(\iota_n(type_{\mathcal{P}_0}(?m)))$ for all $?m \in \mathcal{P}_0$
(using properties of $\Longrightarrow$ and Proposition 2.72) entails:
$\iota_{n+1}(ctxt_{\mathcal{P}_0}(?m)) \vdash_{\mathcal{P}_{n+1}} \iota_{n+1}(?m) : \iota_{n+1}(type_{\mathcal{P}_0}(?m))$ for all $?m \in \mathcal{P}_0$
(by $\iota_{n+1} \equiv \iota_n \uplus \iota'$ and definition of $\uplus$) $\square$

We conclude this section with an example which demonstrates the necessity

of having two premisses in the rules $(\exists\, R)$ and $(\forall\, L)$, which is a significant departure from standard (untyped) logic, where the same rules would be stated as

$$\frac{\Gamma \vdash P[t]}{\Gamma \vdash \exists\, x.P[x]}\ (\exists\, R) \qquad \frac{\Gamma, \forall\, x : T.P[x], \Gamma', P[t] \vdash G}{\Gamma, \forall\, x : T.P[x], \Gamma' \vdash G}\ (\forall\, L)$$

where t is any well-formed term. Since the domain of interpretation (for example in classical first-order logic) is assumed to be non-empty, the term t can always be taken to be a variable y, which stands for an arbitrary element of the domain.

Example 4.23
We compare a proof of the proposition $\exists\, x_1, x_2.P(x_2) \to P(f(x_1))$ in classical logic with a proof of $\Gamma \vdash \exists\, x_1 : A, x_2 : B.P(x_2) \to P(f(x_1))$ in type theory, where $(f : A \to B) \in \Gamma$.

A derivation of the first proposition in classical logic is:

$$\cfrac{\cfrac{\cfrac{}{P(y_2) \vdash P(f(y_1))}\ (\text{axiom})}{\vdash P(y_2) \to P(f(y_1))}\ (\to R)}{\vdash \exists\, x_1, x_2.P(x_2) \to P(f(x_1))}\ (\exists\, R)$$

This proof is finished by unification of $P(y_2)$ and $P(f(y_1))$, thus an assignment $y_2 := f(y_1)$.

Compare this with the following proof in type theory:

$$\cfrac{\Gamma \vdash ?y_1 : A \quad \Gamma \vdash ?y_2 : B \quad \cfrac{\cfrac{}{\Gamma, h : P(?y_2) \vdash ?n_3 : P(f(?y_1))}\ (\text{axiom})}{\Gamma \vdash ?n_1 : P(?y_2) \to P(f(?y_1))}\ (\to R)}{\Gamma \vdash ?n_0 : \exists\, x_1 : A, x_2 : B.P(x_2) \to P(f(x_1))}\ (\exists\, R)$$

The instantiation $?y_2 := f(?y_1)$ is typecorrect (given that $?y_1 : A$ and $f : A \to B$) and solves the subgoal $\Gamma \vdash ?y_2 : B$. However, this does not yet finish the proof. If the context Γ contains no other declaration than $f : A \to B$, then the goal is not provable, because no element of type A can be constructed. If, for example, the context Γ contains declarations $c : C, g : C \to A$, then the proof search continues with goal $\Gamma \vdash ?y_1 : A$, eventually yielding $?y_1 := (g\ c)$ as a solution.

This example seems to suggest that establishing the non-emptiness of a given type can be reduced to theorem proving in the propositional fragment of type theory, which is decidable (see [Kle52b] and [Wag95] for an implementation). However, this is not so – as mentioned in Section 1.1.3, adding polymorphic types makes the question of emptiness of a type undecidable. Apart from that, propositions themselves may contain statements about the emptiness of

types, so deciding whether a type is non-empty is as hard as general theorem proving.

4.4.2. Eigenvariable Conditions

In traditional presentations of sequent calculi, the rules $(\forall R)$ and $(\exists L)$ are usually stated with an "eigenvariable condition". Typically, the rule $(\forall R)$ is then given by:

$$\frac{\Gamma \vdash B(c)}{\Gamma \vdash \forall x.\ B(x)}\ (\forall R)$$

under the proviso that the newly introduced constant c does not occur in Γ. In this section, we will examine how the eigenvariable condition is enforced in different calculi, in particular the calculus presented above. Since it has already been shown that the rules of Figure 4.4 and the associated proof transformation system are correct, this section does not provide an additional correctness proof, but rather informal evidence for one aspect of correctness. Since most of the argument is independent of typing, we will omit type information whenever convenient.

Example 4.24
As an illustration, we will use the following formulae throughout this section, the first of which: $\forall x. \exists y. x = y$ is valid, the second of which: $\exists x. \forall y. x = y$ is not.

In traditional tableau calculi, Skolemization is used to eliminate a universal quantifier by keeping track of existential quantifiers on which it depends. Skolemization expresses that the formula $\exists x. \forall y. P(x, y)$ is valid iff $\exists x. P(x, f(x))$ is valid, where f is a fresh function constant. (NB: There are two variants of Skolemization, one of which eliminates existential quantifiers and preserves *satisfiability*. Since we are interested in proving, not refuting, a formula, the dual variant preserving *validity* is required here). In classical logic, Skolemization is justified by semantic arguments. Similar arguments can certainly be provided for intuitionistic logic (though a standard reference like [Fit83] does not even mention the techniques of Skolemization and unification to delay the instantiation of existential quantifiers).

When Skolemizing the above examples, the first formula is reduced to $c =?y$ the second formula to $?x = f(?x)$. Obviously, unification succeeds in the first case, but an occurs check makes it fail in the second case.

There are two impediments to using Skolemization in our framework. The first is that it is hard to justify the introduction of a new function constant f proof-theoretically. When considering, in a typed calculus, the transition

from $\exists\, x : A.\forall\, y : B.P(x, y)$ to $\exists\, x : A.P(x, f(x))$, we make a claim as to the existence of a function $f : A \to B$, for which we have no direct evidence. The second reason for not using Skolemization, related to the first but of a more practical nature, is that it can blow up formulae and make them difficult to understand.

The method that is implicit in our approach is to describe the dependence of existential variables on universal variables. A proof obligation $x_1 : T_1 \ldots x_k : T_k \vdash ?n : T$ expresses that the existential variable $?n$ occurs in the scope of the universal variables $x_1, \ldots x_k$ and can only be solved by terms containing at most $x_1, \ldots x_k$. The examples demonstrate the procedure: In the first example, the proof succeeds because $?y$ can be unified with x.

$$\dfrac{\dfrac{x \;:\; T \vdash ?y \;:\; T \qquad x \;:\; T \vdash ?n_2 \;:\; x = ?y}{x \;:\; T \vdash ?n_1 \;:\; \exists\, y : T.x = y}\,(\exists\, R)}{\vdash ?n_0 \;:\; \forall\, x : T.\exists\, y : T.x = y}\,(\forall\, R)$$

In the second example, however, $?x$ does not unify with y because y does not occur in the context of $?x$ (cf. the description of unification in Example 4.9).

$$\dfrac{\vdash ?x \;:\; T \qquad \dfrac{y : T \vdash ?n_2 \;:\; ?x = y}{\vdash ?n_1 \;:\; \forall\, y : T.?x = y}\,(\forall\, R)}{\vdash ?n_0 \;:\; \exists\, x : T.\forall\, y : T.x = y}\,(\exists\, R)$$

Again, this dependence could be made explicit by a functional encoding of scopes. In the Isabelle system, for example, the two formulae would be simplified to $\bigwedge x.x = (?y\ x)$ and $\bigwedge y.?x = y$, respectively, where $\bigwedge$ is the universal quantifier of Isabelle's meta logic. Unification would succeed in the first case with the function metavariable solved by $?y := \lambda\, z.z$, whereas it would fail in the second case, since $?x$ has global scope and y is bound locally.

These informal observations are corroborated by the following proposition:

Proposition 4.25 (Eigenvariable Condition)
Assume $\Gamma, x : A$ is a valid context with occurrences of metavariables $?m_1, \ldots ?m_k$ in Γ. Then there is no well-typed instantiation ι of $?m_1, \ldots ?m_k$ such that x occurs free in $\iota(\Gamma)$

Proof: Since $\Gamma, x : A$ is a valid context and ι a well-typed instantiation, by Proposition 2.72, $\iota(\Gamma)$ is a valid context and thus cannot contain a free occurrence of x. □

This proposition expresses that instantiations of metavariables do not permit to introduce variables in places where they are out of scope. This is a

re-statement of the eigenvariable condition which says that, after finishing a proof with an instantiation ι, the variable x does not occur free in the hypotheses $\iota(\Gamma)$:

$$\frac{\iota(\Gamma), x : \iota(A) \vdash \iota(?n_1) : \iota(B(x))}{\iota(\Gamma) \vdash \iota(?n_0) : \iota(\Pi x : A.\ B(x))}\ (\forall\, R)$$

A similar remark holds for the rule $(\exists\, L)$.

4.4.3. Optimizations of Proof Search

In this section, we will examine the following two optimizations that lead to a noticeable reduction of the search space when carrying out proof search with the rules of Figure 4.4:

- Omission of the formulae to which some of the Left-rules have been applied.
- Giving preference to "invertible" rules in proof search.

Both optimizations are *per se* no novelties. What makes them interesting for us is that they are applicable even in our framework, and that they can be justified by an elementary reasoning about proof terms (and not about derivations), given some very liberal assumptions about the structure of proof goals.

For the proofs below, we need the following lemma, which states that under certain conditions, hypotheses can be exchanged:

Lemma 4.26
If $\Gamma, \Gamma', h : H \vdash M : A$ (or $\Gamma, \Gamma', h : H\ \mathit{valid}_c$) is derivable in ECC_G and $\mathit{dom}(\Gamma') \cap FV(H) = \varnothing$, then $\Gamma, h : H, \Gamma' \vdash M : A$ (or $\Gamma, h : H, \Gamma'\ \mathit{valid}_c$) is derivable.

Proof: The proof is by induction on derivations in ECC_G. The only interesting case is the application of the rule (Cvalid), the other cases only require application of the induction hypothesis. Assume, then, that $\Gamma, \Gamma', h : H\ \mathit{valid}_c$ has been derived as follows:

$$\frac{\Gamma, \Gamma' \vdash_N H : \mathit{Type}}{\Gamma, \Gamma', h : H\ \mathit{valid}_c}\ (\text{Cvalid})$$

Since $\mathit{dom}(\Gamma') \cap FV(H) = \varnothing$, we can by strengthening (Proposition 2.44) conclude that $\Gamma \vdash H : \mathit{Type}$, and with a renewed application of (Cvalid), that $\Gamma, h : H\ \mathit{valid}_c$. With weakening (Proposition 2.43), we obtain $\Gamma, h : H, \Gamma'\ \mathit{valid}_c$. □

We now state the first optimization: For proof-term free goals, the principal formula (cf. Definition 3.3) can be omitted from the premisses of rules $(\wedge L)$, $(\vee L)$, $(\exists\, L)$ and from the right branch of rule $(\to L)$ without sacrificing completeness, provided the variable declaring the principal formula does not occur elsewhere in the sequent. More precisely:

Proposition 4.27 (Redundant Premisses)
If $\Delta \vdash M : P$ can be derived with the proof rules of Figure 4.4, then there exists a term M' and a derivation of $\Delta \vdash M' : P$ in which the rules $(\wedge L)$, $(\vee L)$, $(\exists\, L)$ and $(\to L)$ are modified as follows (assuming that $p \notin FV(G)$ and $p \notin FV(T)$ for all $(x : T) \in \Gamma'$):

$$\frac{\Gamma, \Gamma', p_A : A, p_B : B \vdash ?n_1 : G}{\Gamma, p : A \wedge B, \Gamma' \vdash ?n_0 : G}\ (\wedge L)$$

$$\frac{\Gamma, \Gamma', p_A : A \vdash ?n_1 : G \qquad \Gamma, \Gamma', p_B : B \vdash ?n_2 : G}{\Gamma, p : A \vee B, \Gamma' \vdash ?n_0 : G}\ (\vee L)$$

$$\frac{\Gamma, \Gamma', y : T, p' : P[y] \vdash ?n_1 : G}{\Gamma, p : \exists\, x : T.P[x], \Gamma' \vdash ?n_0 : G}\ (\exists\, L)$$

$$\frac{\Gamma, p : A \to B, \Gamma' \vdash ?n_1 : A \qquad \Gamma, \Gamma', p' : B \vdash ?n_2 : G}{\Gamma, p : A \to B, \Gamma' \vdash ?n_0 : G}\ (\to L)$$

This presentation of the rules is the one usually found in textbooks (for example [TS96], p. 65).
Proof: The proof is by induction on the derivation of $\Delta \vdash M : P$, with a case distinction on the last proof rule that has been applied. Instead of carrying out a subinduction on derivations for each of the cases, we can show that the principal formula is indeed superfluous in the premisses of the rules, by using propositions previously proved for the term calculus:

- Rule $(\wedge L)$: Assume the premiss $\Gamma, p : A \wedge B, \Gamma', p_A : A, p_B : B \vdash M_1 : G$ has been derived with the original set of rules, for a term M_1, so by the exchange lemma (4.26) above, also $\Gamma, p_A : A, p_B : B, p : A \wedge B, \Gamma' \vdash M_1 : G$ is derivable. We can derive $\Gamma, p_A : A, p_B : B \vdash \mathit{andI}(p_A, p_B) : A \wedge B$, so by the cut rule, $\Gamma, p_A : A, p_B : B, \Gamma' \vdash M_1\{p := \mathit{andI}(p_A, p_B)\} : G$. Another application of the exchange lemma to the declarations in Γ' permits to obtain a derivation of $\Gamma, \Gamma', p_A : A, p_B : B \vdash M_1\{p := \mathit{andI}(p_A, p_B)\} : G$.

- Rule $(\vee L)$: Similar. The derivation $\Gamma, p : A \vee B, \Gamma', p_A : A \vdash M_1 : G$ of the left premiss is turned into $\Gamma, \Gamma', p_A : A \vdash M_1\{p := \mathit{orIl}(p_A)\} : G$, and analogously for the right premiss, using a substitution $p := \mathit{orIr}(p_B)$.

- Rule $(\exists\, L)$: The derivation $\Gamma, p : \exists\, x : T.P[x], \Gamma', y : T, p' : P[y] \vdash M_1 : G$ can be turned into $\Gamma, \Gamma', y : T, p' : P[y] \vdash M_1\{p := exiI(y, p')\} : G$
- Rule $(\rightarrow\, L)$: In this case, we only consider a derivation $\Gamma, p : A \rightarrow B, \Gamma', p' : B \vdash M_2 : G$ of the right premiss. This derivation can be turned into $\Gamma, \Gamma', p' : B \vdash M_2\{p := \lambda\, x : A.p'\} : G$, if x is a fresh variable. It is worth noting that this kind of reasoning does not succeed in the dependent case, i.e. for the rule $(\forall\, L)$, because the proof term obtained for the left premiss, which also occurs in the context of the right premiss, may depend on p, so strengthening does not apply here.

□

The second optimization is based on the notion of invertible rule.

Definition 4.28
A rule of the form

$$\frac{\Gamma_1 \vdash M_1 : A_1 \;\; \ldots \;\; \Gamma_n \vdash M_n : A_n}{\Gamma_0 \vdash M_0 : A_0}$$

is called *invertible* if $\Gamma_0 \vdash M_0 : A_0$ is provable if and only if all $\Gamma_i \vdash M_i : A_i$ $(i = 1, \ldots, n)$ are provable.

By applying an invertible rule backwards, "no information is lost". If several rules are applicable to a goal $\Gamma_0 \vdash M_0 : A_0$, an invertible rule can safely be chosen first. If one of the $\Gamma_i \vdash M_i : A_i$ turns out to be unprovable, the original goal is unprovable, so no backtracking with other rules is required.

The concept of invertibility is illustrated best with a counterexample. The intuitionistic rule $(\vee Rl)$ is not invertible, as can be seen when applying it to the goal $A, B : Prop \vdash ?n : A \vee (B \rightarrow B)$. The classical counterpart

$$\frac{\Gamma \vdash P, Q}{\Gamma \vdash P \vee Q}\ (\vee R)$$

of this rule is invertible, so the greater proof-theoretic complexity of intuitionistic logic (intuitionistic propositional logic is PSPACE-complete [Sta79]) becomes plausible.

Proposition 4.29 (Invertible Rules)
The rules $(\wedge R)$, $(\wedge L)$, $(\vee L)$, $(\rightarrow R)$, $(\forall\, R)$, $(\neg R)$ and $(\exists\, L)$ are invertible.

Proof: We will illustrate the idea of the proof for the rule $(\wedge R)$, the proof is similar for the other rules. Assume that $\Gamma \vdash pt : A \wedge B$, for a proof term pt. Then the premisses $\Gamma \vdash pt_A : A$ and $\Gamma \vdash pt_B : B$ are provable with proof terms $pt_A := andEl(pt)$ and $pt_B := andEr(pt)$. □

Apart from the rules mentioned in Proposition 4.29, the rules $(\to L)$ and $(\forall L)$ are *semi-invertible* in the sense that provability of the conclusion of the rule implies provability of its right premiss.

Let us emphasize again that we do not claim originality as far as the above optimizations are concerned, but proving their adequacy is facilitated by the type-theoretic apparatus developed so far. Historically, Kleene's [Kle52a] "permutability" of proof rules anticipates most of the concept of invertibility examined above, but is more difficult to handle. The idea has later been taken up in various forms, see for example the notion of "uniform proof" [MNPS91], the optimizations described in [Sha92] and [Wei95] and implemented in the intuitionistic theorem prover described in [SFH92]. Altogether, there is reason to hope that these refinements can be transferred into a type-theoretic setting without major difficulties.

5. Conclusions

5.1. Summary of Results

This thesis has presented methods underlying machine assisted proof construction in type theory, which are applicable both in interactive proof development and in the form of automated proof search procedures.

The main contributions of this thesis are:

- Analysis of situations commonly encountered when carrying out proofs in type theory; identification of difficulties arising with a naive approach of manipulating incomplete proof objects.

- Presentation of a calculus with explicit substitutions which solves these difficulties. A detailed investigation of its properties confirms that desirable meta-theoretic properties of type theoretic calculi (confluence, decidability of type inference, subject reduction, strong normalization) are preserved.

- Development of a sequent calculus having a subterm property for types; the sequent calculus is correct with respect to the original natural deduction calculus and complete for predicative fragments.

- Gradual transformation and refinement of a general proof search procedure, resulting in a unification algorithm, primarily for a "first-order" fragment, but also allowing for higher-order extensions, and a first-order proof search procedure including some optimizations.

Calculi with explicit substitutions and proof search procedures based on sequent systems are not an invention of this thesis and have been used elsewhere, however mostly in conjunction with weaker logics and for a different purpose. Dependent types often lead to a perceptible increase in complexity (e.g. typing rules; proof of cut elimination) and require non-standard adaptations of some algorithms (e.g. unification; proof search). To the best of our knowledge, a similar combination of a calculus with explicit substitutions and methods of

proof search has not been described elsewhere and thus constitutes a novelty of this thesis.

A recurring theme is the comparison of our calculus with approaches based on a functional encoding of scopes, which helps to gain insight into properties of our calculus (strong normalization, Section 2.7.2) and permits to establish a correspondence with some standard algorithms (e.g. higher-order unification, Section 4.3.3), but also shows that a functional encoding of scopes is inappropriate for practical proof development in the given logic.

A technique employed throughout this thesis is the constructive transformation of a calculus into a form which makes it more suitable for proof search (cut elimination proof in Section 3.2.2, motivation of unification in Section 4.2, optimizations of proof search in Section 4.4.3). A closer analysis of these transformations has permitted to compare properties of the target with properties of the source calculus (e.g. completeness proof for predicative subsystems of the λ-cube in Section 3.3). It has helped to track down the origin of excessive branching in proof search (e.g. synthesis of types in goals of the form $\Gamma \vdash ? T : \mathit{Type}$ in Section 4.1) and lead to the proposal of a practically useful proof search calculus.

Apart from the theoretical results showing that the calculi under investigation have the desired properties, the viability of the approach has been demonstrated by an implementation of the key algorithms in the TYPELAB system.

5.2. Evaluation and Perspectives

In this section, we will assess the results obtained in this thesis and outline future work, first discussing some issues related to theory and then addressing aspects of an implementation.

The theory of a calculus with metavariables and explicit substitutions presented here can be considered as completed in the sense that the properties proved in this thesis show that the calculus is adequate for the purpose it was designed for. In practice, this view is confirmed by the (admittedly still not very numerous) external users of the TYPELAB system who perceive working with the calculus as intuitive. However, we feel that the presentation of the calculus together with all the accompanying side conditions (proof problems, order on metavariables etc.) is still too complex. This is an obstacle to a completely formal reasoning about the calculus or to an integration into a reflexive architecture [Rue95, Pfe95], and it also has negative effects on an efficient implementation. A topic of future research is therefore to find out whether a streamlined presentation of the calculus is possible without impos-

ing restrictions which make the calculus useless for program development or proof search.

Even though of purely theoretical interest, another question that remains to be solved is whether the cut elimination proof for predicative fragments of *CC* can be extended to the whole calculus. A direct adaptation of the method employed in Section 3.2.2 is unlikely to succeed, so it is tempting to exploit the structure of normal terms to guide cut elimination, as suggested in the more detailed discussion in Section 1.3.2.

Several practical issues have to be settled to make automated proof search satisfactory in realistic applications in verification and software development. Currently, proof search in TYPELAB performs well on medium-sized problems not depending on many assumptions, but often fails in more complex environments. To use proof search techniques efficiently in a larger context, users have to split propositions into sufficiently small portions in order to benefit from the automation provided by the system. In the following, we will discuss inherent problems and possible improvements.

The problem of how to deal with *definitions and computational behaviour* is one of the most prominent problems from which we have largely abstracted in this thesis. For a practically useful system, it is indispensable to have a means of binding terms to an identifier by a definition. In general, this leads to an extension of the notion of context where contexts may consist of declarations like $x : T$, as used before, and definitions $d := t$. It is not obvious how to handle definitions during proof search, and, in particular, when to expand them. The following algorithms have to be revised in the presence of definitions:

- Unification: First expanding all definitions and then applying an algorithm in the style of Section 4.3 is practically infeasible, in particular for complex or deeply nested definitions, mostly because it destroys the abstraction mechanism which definitions provide in the first place. For example, solving the unification problem $(d\ a_1) \stackrel{?}{\simeq} (d\ a_2)$ may be inexpensive (provided a_1 and a_2 are easy to unify), even if d represents a large term. For this reason, unification has to proceed incrementally, expanding definitions only if non-expansion has lead to a failure. For example, even if a_1 and a_2 are not unifiable, $(d\ a_1) \stackrel{?}{\simeq} (d\ a_2)$ may still succeed if d expands to a function that forgets its first argument. This induces a simple form of search with backtracking even in apparently "deterministic" fragments like first-order logic. An interesting question is whether there are optimal unification strategies which are complete (for a given fragment), but only expand a minimal number of definitions.

- Proof search: In a similar fashion, proof rules may become applicable

after definitions have been expanded. For example, a higher-order predicate d can take an arbitrarily complex proposition p as argument. The term $(d\ p)$ may then expand to an expression whose structure has little in common with p.

Developing term indexing techniques which take into account definitions may improve the current situation considerably.

In this thesis (and in particular in Chapter 4), theorem proving has primarily been considered from a "logical" angle. For realistic applications, dedicated techniques for *equality reasoning and induction* are required. Experiments with interfacing an external untyped, first-order term rewriting system and TYPELAB [SS97] have shown that the gap between the logics involved is too large to permit a direct adaptation of traditional rewriting techniques to type theory. Not surprisingly, typing information is relevant. In particular, dependent typing prohibits a simple structural replacement of terms, because this may produce ill-typed terms. Some of the fundamental algorithms developed in this thesis (such as unification) are currently used in TYPELAB's rewrite package, but specialized techniques have to be integrated to make rewriting more efficient. The same holds true for induction. It can currently be invoked manually in TYPELAB, but so far no further automation in the spirit of [BM79, BSvH+93] is available.

A. Appendix

A.1. A formulation with de Bruijn Indices

A.1.1. Elementary concepts of de Bruijn Indices

In order to make some of the concepts developed in the previous sections clearer, some of the definitions will be restated using de Bruijn indices. De Bruijn indices [dB72] were devised to provide a representation of λ-terms (and, more generally, of quantification) in which variables bear no names and in which the usual difficulties related to variable names can be avoided.

A term in de Bruijn notation can be understood as a representative of a class of mutually α-equivalent terms. The general idea of de Bruijn indices is to indicate the binding position of a variable occurrence in a term not by its name, but by its distance from the binding position, a natural number. More precisely, the de Bruijn index of a variable occurrence in a term is the number of quantifiers that have to be crossed when moving from the occurrence towards the root of the term tree, plus 1[1]. For example, the untyped λ-term $(\lambda\, x.(\lambda\, y.(y\ x))x)$ is represented as $(\lambda(\lambda(1\ 2))1)$. The advantage of de Bruijn's notation is its precision, which also makes it an adequate internal representation for system implementations, in particular in purely functional programming languages. However, the fact that it is difficult to read makes it inappropriate for textual presentation. Besides, programming languages offering some notion of "reference" (such as object-oriented languages) may permit other means to represent bindings, thus de Bruijn indices are not an inevitable choice.

This section is concerned with the representation of terms and contexts in de Bruijn's notation (Section A.1.2), a definition of the typing rules (Section A.1.3) and of instantiation (Section A.1.4). The proofs given for the calculus with names can be transcribed in an obvious manner, so they are omitted here.

[1] Some alternative presentations are 0-based and not 1-based.

A.1.2. Term calculus with de Bruijn Indices

Terms in nameless representation have almost the same structure as terms with named variables. The main difference is that named references to variables are replaced by numbers and that in an abstraction, the name behind the quantifier (λ, Π or Σ) is cancelled, only the type remains. Thus, the term $\lambda\, T : \mathit{Type}_1.\ \lambda\, x : T.\ x$ is turned into $\lambda\, \mathit{Type}_1.\ \lambda 1.\ 1.$ In general, terms are generated by the grammar of Figure A.1, $\mathcal{N}$ being the syntactic category of natural numbers.

$$
\begin{array}{rcl}
\mathcal{T} & ::= & \mathcal{N} \\
 & | & \mathit{Prop} \mid \mathit{Type}_i \\
 & | & \Pi\mathcal{T}.\mathcal{T} \mid \lambda\,\mathcal{T}.\mathcal{T} \mid (\mathcal{T}\ \mathcal{T}) \\
 & | & \Sigma\mathcal{T}.\mathcal{T} \mid \mathit{pair}_{\mathcal{T}}(\mathcal{T},\ \mathcal{T}) \mid \pi_1(\mathcal{T}) \mid \pi_2(\mathcal{T}) \\
 & | & \mathcal{M}^\frown\mathcal{S} \\
 & & \\
\mathcal{S} & ::= & [] \mid [\mathcal{N} := \mathcal{T}] :: \mathcal{S}
\end{array}
$$

Figure A.1.: Grammar defining the language with de Bruijn Indices

We use the same abbreviations as in the named version, notably for Π types. Consequently, a term to the left of an arrow $\rightarrow$ counts like a quantifier. For example, $\Pi 1.\ 2$ becomes $1 \rightarrow 2$ and not $1 \rightarrow 1$.

Contexts undergo similar transformations: A context now is a list of terms, not a list of declarations. Thus, contexts are built up by $\langle\rangle$, the empty context, and (T, Γ), where T is a term and Γ a context. The function *length* computes the length of a context.

Remember that according to the formalization given in Section 2.3, the variable-term pair list σ attached to a metavariable $?n^\frown\sigma$ records all the substitutions that can possibly become effective in any solution for $?n$. This substitution list is gradually built up, starting from an empty list, as in $?n^\frown[]$. Depending on whether x occurs in the defining context of $?n$ or not, the variable assignment $\{x := t\}$ is added to substitution σ in a metavariable term $?n^\frown\sigma$ or simply propagated (see Definitions 2.22 and 2.40).

Although awkward in general, it will be more convenient in the following to slightly shift the perspective and to assume that the initial substitution of a metavariable is a substitution in which each variable of the metavariable context is mapped to itself. Thus, if $x_1, \ldots x_k$ are the variables of $\mathit{ctxt}(?n)$, then $?n^\frown[]$ becomes $?n^\frown[x_1 := x_1, \ldots x_k := x_k]$. In a nameless representation, the list

of variables produced by the function *svars* of Section 2.3.1 is no longer interesting. Rather, it is the length of $svars(?n)$ (and thus the length of $ctxt(?n)$) that matters. Accordingly, Definition 2.19 is modified as follows:

Definition A.1 (Metavariables – de Bruijn version)
$\mathcal{M}$ is an infinite set of objects called *metavariables* disjoint from the set $\mathcal{V}$ of variables. The function $svars\# : \mathcal{M} \to Nat$ associates to each $?n \in \mathcal{M}$ the number of variables that may be substituted in $?n$. The identity substitution $idSubst(?n)$ of metavariable $?n$ is the substitution $[k := k, \ldots 1 := 1]$, provided $svars\#(?n) = k$. A metavariable term $?n^\frown\sigma$ is *well-formed* if σ is of the form $[k := t_k, \ldots 1 := t_1]$ provided $svars\#(?n) = k$.

In the following, it will be assumed that all metavariable terms are well-formed. Note that Definition 2.40 has to be adapted in an obvious manner to the above terminology.

Some simple arithmetic has to be performed to shift indices in terms as substitutions are carried out. The following is an auxiliary definition to Definition A.3.

Definition A.2 (Variable shifting)
The *shift* operation s^{+i} increments by i the indices of all free variables in term s and leaves indices of bound variables unchanged. The shift operation is defined as $s^{+i} := inc(s, 0, i)$ by means of the function $inc(s, k, i)$ which increments the free variables with number $> k$ in s by i. The function inc is defined recursively by:

- For variables m:

$$inc(m, k, i) := \begin{cases} m + i & \text{if } m > k \\ m & \text{if } m \leq k \end{cases}$$

- $inc(Prop, k, i) := Prop$, $inc(Type_j, k, i) := Type_j$
- $inc((\mathcal{Q}\ T.\ M), k, i) := (\mathcal{Q}\ inc(T, k, i).\ inc(M, k + 1, i + 1))$ for $\mathcal{Q} \in \{\lambda, \Pi, \Sigma\}$
- $inc((f\ a), k, i) := (inc(f, k, i)\ inc(a, k, i))$
- $inc(pair_T(t_1,\ t_2), k, i) := pair_{inc(T,k,i)}(inc(t_1, k, i),\ inc(t_2, k, i))$
- $inc(\pi_j(t), k, i) := \pi_j(inc(t, k, i))$ for $j = 1, 2$
- $inc(?M^\frown[y_1 := t_1, \ldots y_n := t_n], k, i) :=$
  $?M^\frown[y_1 := inc(t_1, k, i), \ldots y_n := inc(t_n, k, i)]$

The definitions are extended to contexts Γ by: $\Gamma^{+i} := inc(\Gamma, 0, i)$, where:

- $inc(\langle\rangle, k, i) := \langle\rangle$
- $inc((T, \Gamma), k, i) := (inc(T, k, i),\ inc(\Gamma, k+1, i+1))$

Definition A.3 (External Substitutions with de Bruijn Indices) Substitution is defined by the mapping $\xrightarrow{s}$ as follows:

Variable

$$m\{i := s\} \xrightarrow{s} \begin{cases} m-1 & \text{if } m > i \\ s & \text{if } m = i \\ m & \text{if } m < i \end{cases}$$

Constant $(Prop)\{i := s\} \xrightarrow{s} Prop$, $(Type_j)\{i := s\} \xrightarrow{s} Type_j$

Quantifier For $\mathcal{Q} \in \{\lambda, \Pi, \Sigma\}$:
$(\mathcal{Q}T.\ M)\{i := s\} \xrightarrow{s} \mathcal{Q}T\{i := s\}.\ M\{i+1 := s^{+1}\}$

Application $(f\ a)\{i := s\} \xrightarrow{s} (f\{i := s\}\ a\{i := s\})$

Pair $(pair_T(t_1,\ t_2))\{i := s\} \xrightarrow{s} pair_{T\{i:=s\}}(t_1\{i := s\},\ t_2\{i := s\})$

Projection $\pi_j(t)\{i := s\} \xrightarrow{s} \pi_j(t\{i := s\})$ for $j = 1, 2$.

Metavariable

$$(?M^\frown[y_1 := t_1, \ldots y_n := t_n])\{i := s\} \xrightarrow{s}$$

$$?M^\frown[y_1 := t_1\{i := s\}, \ldots y_n := t_n\{i := s\}]$$

Substitution is extended to contexts as follows:

- $\langle\rangle\{i := s\} := \langle\rangle$
- $(T, \Gamma)\{i := s\} := (T\{i := s\},\ \Gamma\{i+1 := s^{+1}\})$

As noted above, any metavariable $?M$ is equipped with a full substitution whose domain already contains all the variables of the domain of $ctxt(?M)$. For this reason, when substituting $\{i := s\}$, the distinction of whether i denotes a variable of $ctxt(?M)$ or not, necessary in Definition 2.22, now becomes immaterial.

The above definition of substitution can again be extended to *parallel* substitutions. Since it is not clear how the "variable" case of the definition can be stated for arbitrary parallel substitutions, we only consider parallel substitutions for a consecutive interval of indices, that is, for substitutions $\sigma := \{i := s_i, \ldots j := s_j\}$ where $i \leq j$ and there are only assignments $k := s_k$ with $i \leq k \leq j$ in σ, and there is an assignment for each such k. We can then define the variable case by:

$$m\{i := s_i, \ldots j := s_j\} \stackrel{s}{\rightarrow} \begin{cases} m - (j - i + 1) & \text{if } m > j \\ s_k & \text{if } m = k \text{ for } i \leq k \leq j \\ m & \text{if } m < i \end{cases}$$

Given the above definitions of substitution, *Beta reduction* is defined as: $(\lambda\, T.M)\ N \ \rightarrow_\beta \ M\{1 := N\}$.

Example A.4
Assume, in a named calculus, a metavariable $?n$ defined by:

$$T : \mathit{Type}, a : T, f : T \rightarrow T, x : T \vdash ?n^\frown[] : T$$

Then, it can be derived that:

$$T : \mathit{Type}, a : T, f : T \rightarrow T \vdash (\lambda\, x : T.\ ?n^\frown[])\ a : T$$

Reduction of the application yields $?n^\frown[x := a]$.

In the nameless calculus, the metavariable is defined by:

$$\mathit{Type}, 1, 2 \rightarrow 3, 3 \vdash ?n^\frown[4 := 4, \ldots 1 := 1] : 4$$

Using the rules of Figure A.2, one can derive:

$$\mathit{Type}, 1, 2 \rightarrow 3 \vdash ((\lambda\, 3.\ ?n^\frown[4 := 4, \ldots 1 := 1])\ 2) : 4\{1 := 2\}$$

Computation of $4\{1 := 2\}$ yields 3, reduction of $((\lambda\, 3.\ ?n^\frown[4 := 4, \ldots 1 := 1])\ 2)$ yields $?n^\frown[4 := 3, 3 := 2, 2 := 1, 1 := 2]$. This result has to be interpreted as follows: the variable with index 4 in the original metavariable context, $T : \mathit{Type}$, is mapped to term 3 in the current context (also T), ..., variable 1 in the metavariable context, $x : T$, is mapped to term 2 in the current context (a). Translating this back to a named representation yields the same result as above.

$$\frac{}{\langle\rangle\ \mathit{valid}_c}\ (\text{Cempty})$$

$$\frac{\Gamma \vdash A : \mathit{Type}_j}{\Gamma, A\ \mathit{valid}_c}\ (\text{Cvalid})$$

$$\frac{\Gamma\ \mathit{valid}_c}{\Gamma \vdash \mathit{Prop} : \mathit{Type}_0}\ (\text{UProp}) \qquad \frac{\Gamma\ \mathit{valid}_c}{\Gamma \vdash \mathit{Type}_j : \mathit{Type}_{j+1}}\ (\text{UType})$$

$$\frac{\Gamma, A, \Gamma'\ \mathit{valid}_c \qquad \mathit{length}(\Gamma') = n}{\Gamma, A, \Gamma' \vdash n + 1 : A}\ (\text{var})$$

$$\frac{\Gamma, A \vdash P : \mathit{Prop}}{\Gamma \vdash \Pi A.P : \mathit{Prop}}\ (\Pi\text{-Form}_1)$$

$$\frac{\Gamma \vdash A : \mathit{Type}_j \quad \Gamma, A \vdash B : \mathit{Type}_j}{\Gamma \vdash \Pi A.B : \mathit{Type}_j}\ (\Pi\text{-Form}_2)$$

$$\frac{\Gamma, A \vdash M : B}{\Gamma \vdash \lambda\, A.M : \Pi A.B}\ (\lambda)$$

$$\frac{\Gamma \vdash M : \Pi A.B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B\,\{1 := N\}}\ (\text{app})$$

$$\frac{\Gamma \vdash A : \mathit{Type}_j \quad \Gamma, A \vdash B : \mathit{Type}_j}{\Gamma \vdash \Sigma A.B : \mathit{Type}_j}\ (\Sigma\text{-Form})$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B\,\{1 := M\} \quad \Gamma, A \vdash B : \mathit{Type}_j}{\Gamma \vdash \mathit{pair}_{\Sigma A.B}(M,\ N) : \Sigma A.B}\ (\text{pair})$$

$$\frac{\Gamma \vdash M : \Sigma A.B}{\Gamma \vdash \pi_1(M) : A}\ (\pi_1) \qquad \frac{\Gamma \vdash M : \Sigma A.B}{\Gamma \vdash \pi_2(M) : B\,\{1 := \pi_1(M)\}}\ (\pi_2)$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash A' : \mathit{Type}_j \quad A \preceq A'}{\Gamma \vdash M : A'}\ (\preceq)$$

Figure A.2.: Typing rules using de Bruijn Indices

A.1.3. Typing with de Bruijn Indices

A translation of the typing rules of the base logic is rather straightforward. It is given in Figure A.2. Some remarks concerning the metavariable rules (Figure A.3):

- Rule (MV-base): Instead of a metavariable with an empty substitution list, the base case is now given by a variable with an identity substitution of the form $?n^\frown[k := k, \ldots 1 := 1]$.
- Rule (MV-weak): Whereas the named variant of this rule (see Figure 2.5) does not leave any traces when it is applied, a shift of the indices of variables declared in Γ becomes necessary here which takes into account that a new declaration is inserted between Γ and Δ.
- Rule (MV-β-Red): Note again the close correspondence between β-reduction which takes place in a term and application of this rule which is carried out on part of the context. Again, this rule can be simulated on terms: Assume $\Gamma \vdash t : T$ and $\Gamma \vdash \lambda\, T.\, \lambda\, \Delta.\ ?n^\frown\sigma \ : \ \Pi T.\Pi\Delta.\ N$. Then applying rule ($\lambda$), performing β-reduction and moving substitutions inside yields the term $\lambda\, \Delta\{1 := t\}.((?n^\frown\sigma)\{k+1 := t^{+k}\})$ of type $\Pi\Delta\{1 := t\}.(N\{k+1 := t^{+k}\})$ in context Γ, where k is the length of Δ.

$$\frac{ctxt_{\mathcal{P}}(?n) \vdash_{\mathcal{P}} type_{\mathcal{P}}(?n) : Type_j}{ctxt_{\mathcal{P}}(?n) \vdash_{\mathcal{P}} ?n^\frown idSubst(?n) : type_{\mathcal{P}}(?n)}\ \text{(MV-base)}$$

$$\frac{\Gamma \vdash_{\mathcal{P}} T : Type_j \qquad \Gamma, \Delta \vdash_{\mathcal{P}} ?n^\frown\sigma : N \qquad length(\Delta) = k}{\Gamma,\, T,\, \Delta^{+1} \vdash_{\mathcal{P}} inc(?n^\frown\sigma, k, 1) :\ inc(N, k, 1)}\ \text{(MV-weak)}$$

$$\frac{\Gamma \vdash_{\mathcal{P}} t : T \qquad \Gamma, T, \Delta \vdash_{\mathcal{P}} ?n^\frown\sigma : N \qquad length(\Delta) = k}{\Gamma, \Delta\{1 := t\} \vdash_{\mathcal{P}} (?n^\frown\sigma)\{k+1 := t^{+k}\} \ :\ N\{k+1 := t^{+k}\}}\ \text{(MV-}\beta\text{-Red)}$$

Figure A.3.: Typing rules for Metavariables using de Bruijn Indices

A.1.4. Definition of instantiations using de Bruijn Indices

The definition of instantiations using de Bruijn indices is almost literally the same as the one given in Definition 2.62, using named variables. Note how the

requirement that bound variables are not renamed translates to a purely homomorphic mapping of ι over quantifiers and contexts in the following definition:

Definition A.5 (Instantiation using de Bruijn Indices)
The applicability conditions and notation for instantiations are the same as in Definition 2.62. The definition for the behaviour of instantiations on terms is as follows:

- $\iota(x) = x$ for variables x.
- $\iota(\mathit{Prop}) = \mathit{Prop}$, $\iota(\mathit{Type}_i) = \mathit{Type}_i$
- $\iota(\mathcal{Q}T.M) = \mathcal{Q}\iota(T).\iota(M)$ for $\mathcal{Q} \in \{\lambda, \Pi, \Sigma\}$
- $\iota(f\ a) = (\iota(f)\ \iota(a))$
- $\iota(\mathit{pair}_T(t_1,\ t_2)) = \mathit{pair}_{\iota(T)}(\iota(t_1),\ \iota(t_2))$
- $\iota(\pi_i(t)) = \pi_i(\iota(t))$ for $i = 1, 2$
- $\iota(?n^\frown[x_1 := t_1 \ldots x_k := t_k]) = \iota(?n)\{x_1 := \iota(t_1) \ldots x_k := \iota(t_k)\}$

Instantiations can be extended to contexts by the following inductive definition:

- $\iota(\langle\rangle) = \langle\rangle$
- $\iota(T, \Gamma) = \iota(T), \iota(\Gamma)$

Example A.6
Continuing with Example A.4, we instantiate $?n$ with the following terms and observe the effect on $t \mathrel{\widehat{=}} ?n^\frown[4 := 3, 3 := 2, 2 := 1, 1 := 2]$, where $\mathit{Type}, 1, 2 \to 3 \vdash t : 3$.

- $\iota_1 := \{?n := x\}$, which translates to $\{?n := 1\}$. Then $\iota_1(t) = 2$, which corresponds to the term a
- $\iota_2 := \{?n := a\}$, which translates to $\{?n := 3\}$. Then $\iota_2(t) = 2$, which corresponds to the term a
- $\iota_3 := \{?n := (f\ a)\}$, which translates to $\{?n := (2\ 3)\}$. Then $\iota_3(t) = (1\ 2)$, which corresponds to the term $(f\ a)$

A.2. Proof of Cut Elimination

We recall the cut elimination theorem (Proposition 3.7):

> Let $\mathcal{F}$ be a fragment in which a term measure for cut elimination can be defined. If the judgement $\Gamma \vdash M : A$ is derivable in the system ECC_{G+cut} (restricted to $\mathcal{F}$), then $\Gamma \vdash M' : A$ is derivable in ECC_G (restricted to $\mathcal{F}$), where M' is a term convertible to M.

A.2.1. Proof

Proof: Before getting into details of the proof, we state the following facts about derivations in ECC_{G+cut}, which can be proved by easy inductions on derivations:

1. Every derivation of $\Gamma \vdash M : A$ contains a subderivation of Γ $valid_c$.
2. Weakening: If $\Gamma \vdash c : C$ and Γ, Δ $valid_c$ are derivable without use of (cut), then also $\Gamma, \Delta \vdash c : C$ is derivable without (cut).
3. Contraction: If $\Gamma_1, x : T, \Gamma_2, y : T, \Gamma_3 \vdash M : A$ is derivable without (cut), then $\Gamma_1, x : T, \Gamma_2, \Gamma_3\{y := x\} \vdash M\{y := x\} : A\{y := x\}$ is derivable without cut.

Recall the following definitions from Proposition 3.7:

- The *rank* of an application $\mathcal{A}$ of the cut rule is $m(C)$, where m is the measure over which the cut elimination theorem is parameterized (Definition 3.6) and C is the cut formula to which the rule is applied.
- The *level* of an application $\mathcal{A}$ of the cut rule is the sum of the depths of the deductions of its premisses.

Given a derivation $\mathcal{D}$, the general idea of the proof is to eliminate cuts in $\mathcal{D}$ by selecting an application of the cut rule having no application of cut above it, and replacing it by cuts of lower rank or level. More precisely, we measure the weight of an application $\mathcal{A}$ of the cut rule by $(rank(\mathcal{A}), level(\mathcal{A}))$, ordered lexicographically. For derivation $\mathcal{D}$, we take the multiset of weights of its cuts, and we show that this multiset becomes smaller by each of the transformations described below. In practice, this amounts to showing that each of the transformations introduces cuts which are of lower rank or which have the same rank but lower level.

For derivations ending with (cut), we make a case distinction on the rule applied in the left premiss of (cut). The following situations may arise:

1. The application of the cut rule can be removed altogether.
2. The application of (cut) can be moved upwards in the left premiss of the derivation, which leaves the rank constant, but decreases the level.
3. The cut can neither be removed nor moved upwards in the left premiss in case the cut rule is applied to the principal formula of (ΠL) or (ΣL). In this case, the cut is replaced with several cuts of lower rank and/or lower level. This case is technically quite involved and will be handled separately.

Moving the cut into the left premiss

We assume that the derivation is of the form

$$\dfrac{\dfrac{\cdots \vdash \cdots}{\Gamma, z : C', \Delta \vdash t : T}\,(R_l) \quad \Gamma \vdash c : C \quad C \preceq C'}{\Gamma, \Delta\{z := c\} \vdash t\{z := c\} : T\{z := c\}}\,(\text{cut})$$

or

$$\dfrac{\dfrac{\cdots \vdash \cdots}{\Gamma, z : C', \Delta\ \mathit{valid}_c}\,(R_l) \quad \Gamma \vdash c : C \quad C \preceq C'}{\Gamma, \Delta\{z := c\}\ \mathit{valid}_c}\,(\text{cut})$$

and make a case distinction on (R_l). Whenever the side condition $C \preceq C'$ is not essential, it will be omitted.

- Rule (Cvalid): We have the following two cases:
  - Application of (cut) is redundant because the derivation of $\Gamma \vdash_N C : \mathit{Type}$ already contains a subderivation of $\Gamma\ \mathit{valid}_c$:

$$\dfrac{\dfrac{\Gamma \vdash_N C : \mathit{Type}}{\Gamma, z : C\ \mathit{valid}_c}\,(\text{Cvalid}) \quad \Gamma \vdash c : C}{\Gamma\ \mathit{valid}_c}\,(\text{cut})$$

  - Assume the derivation is as follows:

$$\dfrac{\dfrac{\Gamma, z : C, \Delta \vdash_N A : \mathit{Type}}{\Gamma, z : C, \Delta, x : A\ \mathit{valid}_c}\,(\text{Cvalid}) \quad \Gamma \vdash c : C}{\Gamma, \Delta\{z := c\}, x : A\{z := c\}\ \mathit{valid}_c}\,(\text{cut})$$

    If $\Gamma \vdash c : C$ in calculus ECC_G, then by the correctness of ECC_G (Proposition 3.4), also $\Gamma \vdash_N c : C$, so by the admissibility of (cut) in ECC_N, we can form the derivation:

$$\dfrac{\dfrac{\Gamma, z : C, \Delta \vdash_N A : \mathit{Type}_j \quad \Gamma \vdash_N c : C}{\Gamma, \Delta\{z := c\} \vdash_N A\{z := c\} : \mathit{Type}_j}\,(\text{cut})}{\Gamma, \Delta\{z := c\}, x : A\{z := c\}\ \mathit{valid}_c}\,(\text{Cvalid})$$

- Rule (var): Distinguish the following cases:
  - Variable z occurs before variable x:

$$\dfrac{\dfrac{\Gamma, z : C, \Delta', x : A, \Delta'' \ \mathit{valid}_c}{\Gamma, z : C, \Delta', x : A, \Delta'' \vdash x : A}\ (\text{var}) \qquad \Gamma \vdash c : C}{\Gamma, \Delta'\{z := c\}, x : A\{z := c\}, \Delta''\{z := c\} \vdash x : A\{z := c\}}\ (\text{cut})$$

    This can be transformed into:

$$\dfrac{\dfrac{\Gamma, z : C, \Delta', x : A, \Delta'' \ \mathit{valid}_c \quad \Gamma \vdash c : C}{\Gamma, \Delta'\{z := c\}, x : A\{z := c\}, \Delta''\{z := c\} \ \mathit{valid}_c}\ (\text{cut})}{\Gamma, \Delta'\{z := c\}, x : A\{z := c\}, \Delta''\{z := c\} \vdash x : A\{z := c\}}\ (\text{var})$$

  - Variable z occurs after variable x: Similar.
  - Variables z and x agree:

$$\dfrac{\dfrac{\Gamma, z : C', \Delta \ \mathit{valid}_c}{\Gamma, z : C', \Delta \vdash z : C'}\ (\text{var}) \qquad \Gamma \vdash c : C \quad C \preceq C'}{\Gamma, \Delta\{z := c\} \vdash c : C'}\ (\text{cut})$$

    The derivation

$$\dfrac{\Gamma, z : C', \Delta \ \mathit{valid}_c \quad \Gamma \vdash c : C \quad C \preceq C'}{\Gamma, \Delta\{z := c\} \ \mathit{valid}_c}\ (\text{cut})$$

    has a lower cut level than the original derivation, so by induction hypothesis it can be converted into a cut-free derivation. From this and from the assumption $\Gamma \vdash c : C$, we can, by an application of weakening (assumption 2. above) derive $\Gamma, \Delta\{z := c\} \vdash c : C$ without use of (cut). Applying rule $(\preceq R)$ yields the desired judgement $\Gamma, \Delta\{z := c\} \vdash c : C'$.

- All other rules except for the Left-rules: We illustrate the principle by considering the rule (Π-Form$_2$), where an application of the (cut) rule:

$$\dfrac{\dfrac{\Gamma, z : C, \Delta \vdash A : \mathit{Type}_j \qquad \Gamma, z : C, \Delta, x : A \vdash B : \mathit{Type}_j}{\Gamma, z : C, \Delta \vdash \Pi x : A.B : \mathit{Type}_j}\ (\Pi\text{-Form}_2) \qquad \Gamma \vdash c : C}{\Gamma, \Delta\{z := c\} \vdash (\Pi x : A.B)\{z := c\} : \mathit{Type}_j}\ (\text{cut})$$

  can be permuted to: (with $\sigma \mathrel{\widehat{=}} \{z := c\}$)

$$\dfrac{\dfrac{\Gamma, z : C, \Delta \vdash A : \mathit{Type}_j \qquad \Gamma \vdash c : C}{\Gamma, \Delta\sigma \vdash A\sigma : \mathit{Type}_j}\ (\text{cut}) \qquad \dfrac{\Gamma, z : C, \Delta, x : A \vdash B : \mathit{Type}_j \qquad \Gamma \vdash c : C}{\Gamma, \Delta\sigma, x : A\sigma \vdash B\sigma : \mathit{Type}_j}\ (\text{cut})}{\Gamma, \Delta\sigma \vdash \Pi x : A\sigma.\ B\sigma : \mathit{Type}_j}\ (\Pi\text{-Form}_2)$$

The rules with one resp. three premisses are treated similarly. For rule $(\preceq R)$, note that $A \preceq A'$ implies $A\{z := c\} \preceq A'\{z := c\}$. For rule (pair), apply the Substitution Lemma 2.34.

The Left-Rules – Survey:
We have the following two cases:

- The cut is applied to a formula other than the principal formula of $(\preceq L)$, (ΠL) or (ΣL). In this case, the cut can be moved upwards in a way similar to rule (Π-Form$_2$), see above.

- The cut is applied to the principal formula. The case of rule $(\preceq L)$ can be handled with the extended (cut) rule. Assume that the original derivation has the form:

$$\dfrac{\dfrac{\Gamma, z : C'', \Delta \vdash N : G \quad C' \preceq C''}{\Gamma, z : C', \Delta \vdash N : G}\,(\preceq L) \qquad \Gamma \vdash c : C \quad C \preceq C'}{\Gamma, \Delta\{z := c\} \vdash N\{z := c\} : G\{z := c\}}\,(\text{cut})$$

  By transitivity of $\preceq$, this can be transformed into:

$$\dfrac{\Gamma, z : C'', \Delta \vdash N : G \quad \Gamma \vdash c : C \quad C \preceq C''}{\Gamma, \Delta\{z := c\} \vdash N\{z := c\} : G\{z := c\}}\,(\text{cut})$$

- If the cut is applied to the principal formula of (ΠL) or (ΣL), the situation is more complex. In the case of (ΠL), for example, we have the following situation:

$$\dfrac{\dfrac{\Gamma, z : \cdots \Gamma' \vdash \cdots \quad \Gamma, z : \cdots, \Gamma', z' : \cdots \vdash \cdots}{\Gamma, z : \Pi x : A.\ B, \Gamma' \vdash N : G}\,(\Pi L) \qquad \dfrac{\vdots}{\Gamma \vdash c : \Pi x : A.\ B}\,(R_r)}{\Gamma, \Gamma'\{z := c\} \vdash N\{z := c\} : G\{z := c\}}\,(\text{cut})$$

  Moving the cut upwards towards the left premiss cannot be done in the same straightforward fashion as in the preceding cases, since this would make a renewed application of (ΠL) impossible. To deal with this case, we make a case distinction on (R_r) in order to find out how $\Gamma \vdash c : \Pi x : A.\ B$ has been generated. There are essentially the following possibilities:

  - (R_r) is the (var) rule: This case can be dealt with by the third assumption stated at the beginning of this proof.
  - (R_r) is the $(\preceq R)$ rule: Using transitivity of $\preceq$ and applying (cut) directly to the premiss of (R_r), the application of (R_r) can be dropped altogether.

- (R_r) is the (λ) rule: See case "Cut with formula generated by (λ)".
- (R_r) is a left rule: See case "Cut with formula generated by a left rule". Eventually, this amounts to moving the cut upwards towards the right premiss. However, this is far more complex than, for example, in the case of standard predicate logic, due to dependent context entries. See the remarks following this proof in A.2.2.

An analogous reasoning has to be applied for Σ types, where we have a similar case distinction:

- (R_r) is the (var) or ($\preceq R$) rule: As for Π types.
- (R_r) is the (pair) rule: See case "Cut with formula generated by (pair)".
- (R_r) is a left rule: The reasoning is as spelled out under "Cut with formula generated by a left rule" for Π types, with obvious adaptations to Σ types. However, we do not treat this case explicitly here.

Cut with formula generated by (λ)

Assume that the derivations of the right and left premiss of the (cut) rule (called $\mathcal{D}_0^\lambda$ and $\mathcal{D}^{\Pi L}$, respectively) have the following form:

$$\frac{\begin{array}{c}\mathcal{D}_0\\ \Gamma, x : A \vdash b : B\end{array}}{\Gamma \vdash \lambda\, x : A.\ b : \Pi x : A.\ B}\ (\lambda)$$

$$\frac{\begin{array}{c}\mathcal{D}_1\\ \Gamma, z : \Pi x : A.\ B, \Gamma' \vdash N_1 : A\end{array} \quad \begin{array}{c}\mathcal{D}_2\\ \Gamma, z : \Pi x : A.\ B, \Gamma', z' : B\{x := N_1\} \vdash N_2 : G\end{array}}{\Gamma, z : \Pi x : A.\ B, \Gamma' \vdash N_2\{z' := z\ N_1\} : G}\ (\Pi L)$$

Thus, altogether, the derivation $\mathcal{D}_{cut}$ in ECC_{G+cut} has the following form (the substitution $\{z := \lambda\, x : A.\ b\}$ is called σ_1):

$$\frac{\begin{array}{c}\mathcal{D}^{\Pi L}\\ \Gamma, z : \Pi x : A.\ B, \Gamma' \vdash N_2\{z' := z\ N_1\} : G\end{array} \quad \begin{array}{c}\mathcal{D}_0^\lambda\\ \Gamma \vdash \lambda\, x : A.\ b : \Pi x : A.\ B\end{array}}{\Gamma, \Gamma'\sigma_1 \vdash N_2\{z' := z\ N_1\}\sigma_1 : G\sigma_1}\ (\text{cut})$$

To obtain a proof of lower complexity, construct the derivation $\mathcal{D}$ consisting of the following series of applications of the cut rule:

$$\dfrac{\dfrac{A \qquad B}{\cdots \vdash \cdots}\, c_3 \qquad \dfrac{C \qquad \dfrac{D \qquad E}{\cdots \vdash \cdots}\, c_1}{\cdots \vdash \cdots}\, c_2}{\cdots \vdash \cdots}\, c_4$$

In detail, the derivations are as follows:

$$\dfrac{\Gamma, z : \Pi x : A.\ B, \Gamma' \vdash N_1 : A \quad \Gamma \vdash \lambda\, x : A.\ b : \Pi x : A.\ B}{\Gamma, \Gamma'\sigma_1 \vdash N_1\sigma_1 : A}\ (\mathrm{cut})_1$$

Note that z does not occur in A, therefore $A\sigma_1 \equiv A$. The level of $(\mathrm{cut})_1$ is lower than the level of (cut) in $\mathcal{D}_{cut}$, whilst the cut rank remains the same or decreases. Thus, by induction hypothesis, $(\mathrm{cut})_1$ can be eliminated.

$$\dfrac{\dfrac{\Gamma, x : A \vdash b : B}{\Gamma, \Gamma'\sigma_1, x : A \vdash b : B}\ (\mathrm{weak}) \qquad \dfrac{\vdots}{\Gamma, \Gamma'\sigma_1 \vdash N_1\sigma_1 : A}\, c_1}{\Gamma, \Gamma'\sigma_1 \vdash b\sigma_2 : B\sigma_2}\ (\mathrm{cut})_2$$

Here, the substitution $\{x := N_1\sigma_1\}$ is called σ_2. The cut rank of $(\mathrm{cut})_2$ is lower than the cut rank of (cut) in $\mathcal{D}_{cut}$. Thus, by induction hypothesis, $(\mathrm{cut})_2$ can be eliminated.

$$\dfrac{\Gamma, z : \Pi x : A.\ B, \Gamma', z' : B\{x := N_1\} \vdash N_2 : G \quad \Gamma \vdash \lambda\, x : A.\ b : \Pi x : A.\ B}{\Gamma, \Gamma'\sigma_1, z' : B\{x := N_1\}\sigma_1 \vdash N_2\sigma_1 : G\sigma_1}\ (\mathrm{cut})_3$$

Again, with respect to the cut in $\mathcal{D}_{cut}$, the level of $(\mathrm{cut})_3$ decreases, the cut rank remains the same. Thus, $(\mathrm{cut})_3$ can be eliminated.

$$\dfrac{\dfrac{\vdots}{\Gamma, \Gamma'\sigma_1, z' : B\{x := N_1\}\sigma_1 \vdash N_2\sigma_1 : G\sigma_1}\, c_3 \qquad \dfrac{\vdots}{\Gamma, \Gamma'\sigma_1 \vdash b\sigma_2 : B\sigma_2}\, c_2}{\Gamma, \Gamma'\sigma_1 \vdash N_2\sigma_1\{z' := b\sigma_2\} : G\sigma_1}\ (\mathrm{cut})_4$$

Note that the last application of (cut) is typecorrect since $B\{x := N_1\}\sigma_1$ and $B\sigma_2$ are the same term: Indeed, by the Substitution Lemma (Lemma 2.34), $B\{x := N_1\}\sigma_1 \equiv B\{x := N_1\}\{z := \lambda\, x : A.\ b\}$, which, by the fact that z does not occur in B, equals $B\{x := N_1\{z := \lambda\, x : A.\ b\}\} \equiv B\{x := N_1\sigma_1\} \equiv B\sigma_2$.

Similarly, $G\sigma_1\{z' := b\sigma_2\} \equiv G\sigma_1$. Again by the Substitution Lemma, the term $N_2\{z' := z\ N_1\}\sigma_1 \equiv N_2\{z' := z\ N_1\}\{z := \lambda\, x : A.\ b\}$ resulting from the

derivation $\mathcal{D}_{cut}$ is β-convertible to (but not the same as!) the term $N_2\{z := \lambda\, x : A.\ b\}\{z' := b\{x := N_1\sigma_1\}\} \equiv N_2\sigma_1\{z' := b\sigma_2\}$ resulting from the cut-free derivation.

The cut rank decreases for $(\text{cut})_4$, provided that for every $M : A$, $m(B\{x := M\}) < m(\Pi x : A.\ B)$, in particular for $N_1\sigma_1$. Thus, by induction hypothesis, $(\text{cut})_4$ can be eliminated.

Cut with formula generated by a left rule

The application of (cut) has the following form:

$$\dfrac{\dfrac{\begin{matrix}\mathcal{D}_0\\ \Gamma, z : \Pi x : A.\ B, \Gamma' \vdash N_1 : A\end{matrix} \quad \begin{matrix}\mathcal{D}_1\\ \Gamma, z : \Pi x : A.\ B, \Gamma', z' : B[N_1] \vdash N_2 : G\end{matrix}}{\Gamma, z : \Pi x : A.\ B, \Gamma' \vdash N_2\{z' := z\ N_1\} : G}\,(\Pi L) \qquad \mathcal{R}}{\Gamma, \Gamma'\sigma \vdash N_2\{z' := z\ N_1\}\sigma : G\sigma}\,(\text{cut})$$

The derivation $\mathcal{R}$ terminates with a judgement $\Gamma \vdash c : \Pi x : A.\ B$ (so $\sigma \equiv \{z := c\}$), which has been generated by a left rule ((ΠL) or (ΣL)). Following the main premiss of this application, there is a sequence of other applications of left rules which is eventually terminated by an application of rule (λ) or rule (var).

- If the sequence of successive applications of left rules terminates with rule (λ), the application of (λ) can be permuted below the applications of (ΠL) and (ΣL). Therefore, this case can be reduced to the case "Cut with formula generated by (λ)" handled above.
- Otherwise, the sequence of successive applications of (ΠL) terminates with rule (var), which we will examine in the following.

Thus, assume that derivation $\mathcal{R}$ has the following form. In order not to clutter up notation, we present the derivation as if only (ΠL)-rules had been applied. The reasoning is the same for any combination of (ΠL)- and (ΣL)-rules.

$$\dfrac{\dfrac{\Gamma_0 \vdash t_0 : T_0 \qquad \dfrac{\Gamma_1 \vdash c_1 : \Pi x : A.\ B \qquad \dfrac{\vdots}{\Gamma_{n-1} \vdash t_{n-1} : T_{n-1} \quad \dfrac{\vdots}{\Gamma_n \vdash c_n : \Pi x : A.\ B}\,(var)}\,(\Pi L)_{n-1}}{\vdots}}{\Gamma_0 \vdash c_0 : \Pi x : A.\ B}\,(\Pi L)_0$$

where $c_0 \equiv c$ and where $\Gamma_0 \equiv \Gamma$ and $\Gamma_{i+1} \equiv \Gamma_i, x_i : P_i$ for a P_i which need not be specified in detail here. For $1 \leq i < n$, define the substitution γ_i to be

$\{x_{i+1} := (a_i\ t_i)\}$, where a_i is the variable where $(\Pi L)_i$ has been applied. Then $c_n \in dom(\Gamma_n)$ and for $1 \leq i < n$, we have $c_i \equiv c_{i+1}\gamma_i$, thus $c_0 \equiv x_n\gamma_{n-1}\ldots\gamma_0$.

As a preparation, form new derivations $\mathcal{A}$:

$$\dfrac{\dfrac{\overset{\mathcal{D}_0}{\Gamma, z : \Pi x : A.\ B, \Gamma' \vdash N_1 : A} \quad \overset{\mathcal{R}}{\Gamma \vdash c_0 : \Pi x : A.\ B}}{\Gamma, \Gamma'\sigma \vdash N_1\sigma : A}\ (\text{cut})_1}{\Gamma, \Gamma'\sigma, x_1 : P_1, \ldots x_n : P_n \vdash N_1\sigma : A}\ (\text{weak})$$

and $\mathcal{B}$:

$$\dfrac{\overset{\mathcal{D}_1}{\Gamma, z : \Pi x : A.\ B, \Gamma', z' : B[N_1] \vdash N_2 : G} \quad \overset{\mathcal{R}}{\Gamma \vdash c_0 : \Pi x : A.\ B}}{\Gamma, \Gamma'\sigma, z' : B[N_1]\sigma \vdash N_2\sigma : G\sigma}\ (\text{cut})_2$$

Note that the cut level of $\mathcal{A}$ and $\mathcal{B}$ is lower than the level of the original cut, the rank remains the same.

Since $c_n \in \Gamma_n$ and $c_n : \Pi x : A.\ B$, the rule (ΠL) can be applied once more at position c_n, which gives the following subderivation:

$$\dfrac{\begin{array}{c}\overset{\mathcal{A}}{\Gamma, \Gamma'\sigma, x_1 : P_1, \ldots x_n : P_n \vdash N_1\sigma : A} \\ \Gamma, \Gamma'\sigma, x_1 : P_1, \ldots x_n : P_n, x_{n+1} : B[N_1]\sigma \vdash x_{n+1} : B[N_1]\sigma\end{array}}{\Gamma, \Gamma'\sigma, x_1 : P_1, \ldots x_n : P_n \vdash (x_n\ N_1\sigma) : B[N_1]\sigma}\ (\Pi L)_n$$

Call this derivation $\mathcal{V}$.

We can now modify $\mathcal{R}$ as follows, by replacing the derivation above (var) by $\mathcal{V}$ and weakening the contexts Γ_i by defining: $\Gamma'_0 \mathrel{\widehat{=}} \Gamma, \Gamma'\sigma$ and $\Gamma'_{i+1} \mathrel{\widehat{=}} \Gamma'_i, x_i : P_i$.

$$\dfrac{\Gamma'_0 \vdash t_0 : T_0 \qquad \dfrac{\dfrac{\overset{\mathcal{V}}{\Gamma'_{n-1} \vdash t_{n-1} : T_{n-1} \quad \Gamma'_n \vdash (x_n\ N_1\sigma) : B[N_1]\sigma}\ (\Pi L)_{n-1}}{\vdots}}{\Gamma'_1 \vdash c'_1 : B[N_1]\sigma}}{\Gamma'_0 \vdash c'_0 : B[N_1]\sigma}\ (\Pi L)_0$$

Note that the validity of each Γ'_i can be shown by a cut of the contexts $\Gamma, z : \Pi x : A.\ B, \Gamma', x_1 : P_1 \ldots x_{i-1} : P_{i-1}$ with $\Gamma \vdash c_0 : \Pi x : A.\ B$. These cuts can be eliminated, since they have lower level than the original cut.

With γ_i as defined above, we now have $c'_n \equiv (x_n\ N_1\sigma)$ and $c'_i \equiv c'_{i+1}\gamma_i$, which gives $c'_0 \equiv (x_n\ N_1\sigma)\gamma_{n-1}\ldots\gamma_0$.

With the result of derivation $\mathcal{B}$ and the judgement just derived, we can perform a cut, whose rank is lower than the rank of the original cut (note that $B[N_1]\sigma \equiv B[N_1\sigma]$ and thus $m(B[N_1\sigma]) < m(\Pi x : A.\ B)$.

$$\frac{\Gamma_0', z' : B[N_1]\sigma \vdash N_2\sigma : G\sigma \quad \Gamma_0' \vdash c_0' : B[N_1]\sigma}{\Gamma_0' \vdash N_2\sigma\{z' := c_0'\} : G\sigma\{z' := c_0'\}}\ (\text{cut})$$

It remains to be shown that the resulting judgement is the same as the judgement obtained by the original derivation. Note, first, that for all substitutions $\gamma_i \equiv \{x_{i+1} := (a_i\ t_i)\}$, we have $z' \notin FV((a_i\ t_i))$ and therefore $z' \notin FV(c_0)$. Besides, $z' \notin FV(G)$, and therefore $\Gamma\sigma \equiv \Gamma\{z := c_0\} \equiv \Gamma\{z := c_0\}\{z' := c_0'\}$.

Furthermore, $N_2\{z' := zN_1\}\sigma \equiv N_2\{z' := zN_1\}\{z := c_0\} \equiv N_2\{z := c_0\}\{z' := c_0\ N_1\sigma\} \equiv N_2\{z := c_0\}\{z' := c_0'\}$ by the Substitution Lemma and the following reasoning: Expanding the definitions of c_0 and c_0', we obtain $(c_0\ N_1\sigma) \equiv (x_n\gamma_{n-1}\dots\gamma_0\ N_1\sigma) \equiv (x_n\ N_1\sigma)\gamma_{n-1}\dots\gamma_0 \equiv c_0'$ (observe that the domain of none of the γ_i occurs in $N_1\sigma$).

Cut with formula generated by (pair)

Assume the derivation $\mathcal{D}_{cut}$ in ECC_{G+cut} has the following form:

$$\dfrac{\dfrac{\overset{\mathcal{D}_2}{\Gamma, z : \Sigma x : A.\ B, \Gamma', z_1 : A, z_2 : B\{x := \pi_1(z)\} \vdash N_2 : G}\ (\Sigma L) \quad \dfrac{\vdots\ \ \vdots}{\dots \vdash \dots}\ (\text{pair})}{\Gamma, z : \Sigma x : A.\ B, \Gamma' \vdash N_2\sigma_1 : G}\ (\text{cut})}{\Gamma, \Gamma'\sigma_2 \vdash N_2\sigma_1\sigma_2 : G\sigma_2}$$

with the subderivation

$$\frac{\overset{\mathcal{D}_0}{\Gamma \vdash N_0 : A} \quad \overset{\mathcal{D}_1}{\Gamma \vdash N_1 : B\{x := N_0\}}}{\Gamma \vdash \mathit{pair}(N_0, N_1) : \Sigma x : A.\ B}\ (\text{pair})$$

inserted at the indicated position, and where the substitution σ_1 is defined as $\{z_1 := \pi_1(z), z_2 := \pi_2(z)\}$, the substitution σ_2 as $\{z := \mathit{pair}(N_0, N_1)\}$.

This derivation can be turned into a derivation $\mathcal{D}$ of lower complexity, having the following form:

$$\dfrac{\dfrac{\dfrac{A \quad B}{\dots \vdash \dots}\ c_1 \qquad C}{\dots \vdash \dots}\ c_2 \qquad D}{\dots \vdash \dots}\ c_3$$

where the subderivations are given by:

$$\dfrac{\overset{\mathcal{D}_2}{\Gamma, z, \Gamma', z_1, z_2 \vdash N_2 : G} \qquad \dfrac{\overset{\mathcal{D}_0}{\Gamma \vdash N_0 : A} \quad \overset{\mathcal{D}_1}{\Gamma \vdash N_1 : B\{x := N_0\}}}{\Gamma \vdash \mathit{pair}(N_0, N_1) : \Sigma x : A.\ B}\ (\text{pair})}{\Gamma, \Gamma'\sigma_2, z_1 : A, z_2 : B\{x := N_0\} \vdash N_2\sigma_2 : G\sigma_2}\ (\text{cut})_1$$

The type of z_2, $B\{x := N_0\}$, is the result of applying a π-reduction to $B\{x := \pi_1(z)\}\{z := pair(N_0, N_1)\}$. The level of $(\text{cut})_1$ is lower than the level of (cut) in $\mathcal{D}_{cut}$ and thus, by induction hypothesis, $(\text{cut})_1$ can be eliminated.

$$\dfrac{\dfrac{\vdots}{\Gamma, \Gamma'\sigma_2, z_1 : A, z_2 : B\{x := N_0\} \vdash N_2\sigma_2 : G\sigma_2}\,(\text{cut})_1 \quad \begin{matrix}\mathcal{D}_0\\ \Gamma \vdash N_0 : A\end{matrix}}{\Gamma, \Gamma'\sigma_2, z_2 : B\{x := N_0\} \vdash N_2\sigma_2\{z_1 := N_0\} : G\sigma_2}\,(\text{cut})_2$$

Note that z_1 does not occur in B, G, N_0 or N_1, thus $B\{x := N_0\}\{z_1 := N_0\} \equiv B\{x := N_0\}$, similarly for $\Gamma'\sigma_2$, $G\sigma_2$.

The rank of $(\text{cut})_2$ is lower than the rank of (cut) in $\mathcal{D}_{cut}$ and thus, by induction hypothesis, $(\text{cut})_2$ can be eliminated.

$$\dfrac{\dfrac{\vdots}{\Gamma, \Gamma'\sigma_2, z_2 : B\{x := N_0\} \vdash \cdots}\,(\text{cut})_2 \quad \begin{matrix}\mathcal{D}_1\\ \Gamma \vdash N_1 : B\{x := N_0\}\end{matrix}}{\Gamma, \Gamma'\sigma_2 \vdash N_2\sigma_2\{z_1 := N_0\}\{z_2 := N_1\} : G\sigma_2}\,(\text{cut})_3$$

Note that by the Substitution Lemma, the terms $N_2\sigma_2\{z_1 := N_0\}\{z_2 := N_1\} \equiv N_2\{z := pair(N_0, N_1)\}\{z_1 := N_0\}\{z_2 := N_1\}$ and $N_2\sigma_1\sigma_2 \equiv N_2\{z_1 := \pi_1(z), z_2 := \pi_2(z)\}\{z := pair(N_0, N_1)\}$ (the term resulting from derivation $\mathcal{D}_{cut}$) are the same modulo π-conversion.

The rank of $(\text{cut})_3$ is lower than the rank of (cut) in $\mathcal{D}_{cut}$, provided $m(B\{x := N_0\}) < m(\Sigma x : A.\ B)$. Thus, by induction hypothesis, $(\text{cut})_3$ can be eliminated.

□

A.2.2. Discussion

The procedure taken in the case "Cut with formula generated by a left rule" may seem overly complicated. In the following, we will see where an attempt to adopt a method suitable for standard predicate logic breaks down.

So assume the situation is the following:

$$\dfrac{\dfrac{\Gamma, z : \cdots \Gamma' \vdash \cdots \quad \Gamma, z : \cdots, \Gamma', z' : \cdots \vdash \cdots}{\Gamma, z : \Pi x : A.\ B, \Gamma' \vdash N : G}\,(\Pi L)_1 \quad \dfrac{\vdots \quad \vdots}{\Gamma \vdash c : \Pi x : A.\ B}\,(\Pi L)_2}{\Gamma, \Gamma'\{z := c\} \vdash N\{z := c\} : G\{z := c\}}\,(\text{cut})$$

where the right subderivation is

$$\dfrac{\begin{matrix}\mathcal{D}_1\\ \Delta, q : \Pi y : D.\ E, \Delta' \vdash M_1 : D\end{matrix} \quad \begin{matrix}\mathcal{D}_2\\ \Delta, q : \Pi y : D.\ E, \Delta', q' : E[M_1] \vdash M_2 : \Pi x : A.\ B\end{matrix}}{\Delta, q : \Pi y : D.\ E, \Delta' \vdash M_2\{q' := (q\ M_1)\} : \Pi x : A.\ B}\,(\Pi L)_2$$

and thus $\Gamma \equiv \Delta, q : \Pi y : D.\ E, \Delta'$ and $c \equiv M_2\{q' := (q\ M_1)\}$. Let us define $\Gamma_0 \mathrel{\widehat{=}} \Delta, q : \Pi y : D.\ E, \Delta', q' : E[M_1]$.

In order to move the cut into the right premiss, we first weaken the assumptions throughout the left branch of the cut rule, which leads to a derivation:

$$\frac{\Gamma_0, z : \cdots \Gamma' \vdash \cdots \quad \Gamma_0, z : \cdots, \Gamma', z' : \cdots \vdash \cdots}{\Gamma_0, z : \Pi x : A.\ B, \Gamma' \vdash N : G}\ (\Pi L)_1$$

We then move the cut upwards into $\mathcal{D}_2$ as follows (note that weakening in the left branch of the cut rule is a necessary prerequisite for being able to do this):

$$\frac{\dfrac{\cdots}{\Gamma_0, z : \Pi x : A.\ B, \Gamma' \vdash N_1 : G}\ (\Pi L)_1 \quad \begin{matrix}\mathcal{D}_2\\ \Gamma_0 \vdash M_2 : \Pi x : A.\ B\end{matrix}}{\Gamma_0, \Gamma'\{z := M_2\} \vdash N_1\{z := M_2\} : G\{z := M_2\}}\ (\text{cut})$$

Now, it would be necessary to apply (ΠL) again to this judgement and a judgement resulting from (a weakening of) derivation $\mathcal{D}_1$. However, this turns out to be impossible, as the context entry $q' : E[M_1]$ occurs buried inside the antecedent and cannot be moved to the right, since context entries in $\Gamma'\{z := M_2\}$ possibly depend on it.

A.3. Proofs of Miscellaneous Theorems

Proof of Proposition 2.55:

> Whenever $\Gamma \vdash ?n^\frown\sigma : N$ is derivable, then it is derivable by a standard derivation.

Proof: We show that whenever $\Gamma \vdash ?n^\frown\sigma : N$ is derivable by a derivation $\mathcal{D}$ with main branch of length k, then there is a derivation of the same judgement whose main branch has length at most k. The proof is by induction on k.

Assume that $k = 1$. This is only possible if for an application of (MV-base), which makes the derivation standard.

In order to conclude from k to $k+1$ for $k > 1$, we examine the last rule that has been applied, either (MV-β-Red) or (MV-weak), and exploit permutability of rules to obtain a standard derivation. Assume that derivation $\mathcal{D}$ has the form:

$$\dfrac{S_1 \qquad \dfrac{\begin{matrix}\mathcal{D}'\\ S_2\end{matrix} \quad M_2}{M_1}\ (U)}{M_0}\ (L)$$

where, by induction hypothesis,

$$\dfrac{\begin{array}{cc} & \mathcal{D}' \\ S_2 & M_2 \end{array}}{M_1}\,(U)$$

is in standard form and the upper application (U) is one of the metavariable rules (side condition S_2 does not exist if (U) is (MV-base)).

Case 1: Last rule application is (MV-β-Red), thus (L) is (MV-β-Red).

We further make a case distinction on the rule (U):

- (U) is (MV-base): In this case, the derivation is in standard form, as the variable x to which (MV-β-Red) is applied has to be in $dom(ctxt(?n))$, and there are no other applications of metavariable rules above (U).
- (U) is (MV-weak): If $\mathcal{D}$ is not a standard derivation, the following can go wrong:

  Case (a): (L) is applied to a variable $x \notin dom(ctxt(?n))$

  Either the variable introduced by (U) occurs before or behind the variable eliminated by (L), or the variable introduced is immediately eliminated again. The first two cases are covered by the methods of Case (b). In the third case, the derivation is of the following form:

$$\dfrac{\Gamma \vdash t : T \qquad \dfrac{\Gamma \vdash T : \mathit{Type} \quad \Gamma, \Delta \vdash ?n^\frown\sigma : N \;\; \mathcal{D}'}{\Gamma, x : T, \Delta \vdash ?n^\frown\sigma : N}\,(\text{MV-weak})}{\Gamma, \Delta\{x := t\} \vdash (?n^\frown\sigma)\{x := t\} : N\{x := t\}}\,(\text{MV-}\beta\text{-Red})$$

  Since $x \notin FV(\Delta) \cup FV(?n^\frown\sigma) \cup FV(N)$, we conclude that the judgement derived by (U) and (L) is the same as $\Gamma, \Delta \vdash ?n^\frown\sigma : N$, thus the shorter derivation $\mathcal{D}'$ derives the same judgement.

  Case (b): (L) is applied to x_i and there is an application (A) of (MV-β-Red) in $\mathcal{D}'$ applied to an x_j, where x_i occurs before x_j.

  The derivation can be of two forms: Either, the variable z introduced by (U) occurs behind x_i, or z occurs before x_i (because of Case (a), we may exclude that $z = x_i$).

  In the first case, the derivation has the following form ($\tau \mathrel{\widehat{=}} \{x := t\}$):

$$\dfrac{\Gamma \vdash t : T \qquad \dfrac{\Gamma, x_i : T, \Delta' \vdash Z : \mathit{Type} \quad \Gamma, x_i : T, \Delta', \Delta'' \vdash ?n^\frown\sigma : N \;\; \mathcal{D}'}{\Gamma, x_i : T, \Delta', z : Z, \Delta'' \vdash ?n^\frown\sigma : N}\,(\text{MV-weak})}{\Gamma, \Delta'\tau, z : Z\tau, \Delta''\tau \vdash (?n^\frown\sigma)\tau : N\tau}\,(\text{MV-}\beta\text{-Red})$$

By Lemma 2.52, we may permute the application of (MV-β-Red) above the application of (MV-weak) and apply the induction hypothesis above the application of (MV-β-Red) to the derivation tree obtained after permutation. The resulting derivation is a standard derivation, since by induction hypothesis, the two conditions on (MV-β-Red) are met and the relative position of z with respect to other variables introduced by (MV-weak) has not been affected by the permutation.

In the second case, the derivation has the form

$$\dfrac{\Gamma', z : Z, \Gamma'' \vdash t : T \qquad \dfrac{\Gamma' \vdash Z : \mathit{Type} \quad \Gamma', \Gamma'', x_i : T, \overset{\mathcal{D}'}{\Delta} \vdash ?n^\frown\sigma : N}{\Gamma', z : Z, \Gamma'', x_i : T, \Delta \vdash ?n^\frown\sigma : N}\,(\text{MV-weak})}{\Gamma', z : Z, \Gamma''\Delta\tau \vdash (?n^\frown\sigma)\tau : N\tau}\,(\text{MV-}\beta\text{-Red})$$

By assumption, there is an application (A) of (MV-β-Red) in $\mathcal{D}'$ which is applied to an x_j with $x_i < x_j$ (where $<$ denotes position in the context). Since the derivation above (U) is in standard form, we know that all variables z' introduced by an application of (MV-weak) between (L) and (A) have $z' < z < x_i < x_j$. By Lemma 2.53, we can permute application (A) below all weakenings and by Lemma 2.48, we can then permute (L) above (A). By application of induction hypothesis, we obtain a standard derivation.

(*Note:* This argument is not completely formal. A complete proof would require induction on the number of occurrences of (MV-β-Red) in $\mathcal{D}'$, and nested therein, an induction on the number of applications of (MV-weak) between applications of (MV-β-Red)).

- (U) is (MV-β-Red): If $\mathcal{D}$ is not a standard derivation, (L) is applied to a variable $x \notin \mathit{dom}(\mathit{ctxt}(?n))$, or (L) is applied to x_i and (U) is applied to x_j, where x_i occurs before x_j in $\mathit{dom}(\mathit{ctxt}(?n))$ (this may be assumed, since the derivation above (U) is in standard form). In both cases, we can, by Lemma 2.48, permute the applications of (L) and (U) and apply induction hypothesis to obtain a derivation in standard form.

Case 2: Last rule application is (MV-weak), thus (L) is (MV-weak). Again, we make a case distinction on the rule (U):

- (U) is (MV-base): In this case, the derivation is in standard form, as there are no other applications of metavariable rules above (U).
- (U) is (MV-weak): If $\mathcal{D}$ is not a standard derivation, (L) introduces a variable before the variable introduced by (U) (by induction hypothesis,

the derivation above (U) is in standard form). In this case, we can, by Lemma 2.49, permute (L) and (U) and apply induction hypothesis.

- (U) is (MV-β-Red): The only reason why $\mathcal{D}$ may fail to be a standard derivation is that there is an application (A) of (MV-weak) in $\mathcal{D}'$ which introduces a variable z_j such that z_j appears in the context segment behind the variable z_i introduced by application (L). With respect to the relative variable orders of z_i and the variable x eliminated by application (U), we can distinguish the following situations:

  If the derivation has the form:

$$\dfrac{\Gamma' \vdash Z_i : \mathit{Type} \qquad \dfrac{\Gamma', \Gamma'' \vdash t : T \quad \overset{\mathcal{D}'}{\Gamma', \Gamma'', x : T, \Delta \vdash ?n^\frown\sigma : N}}{\Gamma', \Gamma'', \Delta\tau \vdash (?n^\frown\sigma)\tau : N\tau}\ \text{(MV-}\beta\text{-Red)}}{\Gamma', z_i : Z_i, \Gamma'', \Delta\tau \vdash (?n^\frown\sigma)\tau : N\tau}\ \text{(MV-weak)}$$

  and z_j is introduced in the context segment $\Gamma'', x : T, \Delta$, we may apply Lemma 2.53 to permute (U) and (L) and apply induction hypothesis. The resulting derivation is in standard form, because the properties postulated for applications of (MV-β-Red) are not affected by the permutation.

  If the derivation has the form:

$$\dfrac{\Gamma' \vdash Z_i : \mathit{Type} \qquad \dfrac{\Gamma \vdash t : T \quad \overset{\mathcal{D}'}{\Gamma, x : T, \Delta', \Delta'' \vdash ?n^\frown\sigma : N}}{\Gamma, \Delta'\tau, \Delta''\tau \vdash (?n^\frown\sigma)\tau : N\tau}\ \text{(MV-}\beta\text{-Red)}}{\Gamma, \Delta'\tau, z_i : Z_i, \Delta''\tau \vdash (?n^\frown\sigma)\tau : N\tau}\ \text{(MV-weak)}$$

  and z_j is introduced in the context segment Δ'', we remark that since x precedes z_j, we can apply Lemma 2.52 to permute (A) below all applications of (MV-β-Red) and Lemma 2.49 to permute (A) and (L). After application of the induction hypothesis, we obtain a standard derivation. (cf. Case 1(b) above, which uses a similar line of reasoning, and the accompanying note).

□

Proof of Proposition 4.5:

If there is a derivation of $\Gamma \vdash M : A$ in system ECC_G, then there is a derivation $\Gamma \vdash M' : A$ in the system with the rules of ECC_G and (whnf L), (whnf R), such that:

- $M \simeq M'$
- The rule $(\preceq L)$ is only applied directly below the (var) rule, and only at the position which is the subject of the (var) rule (i.e. the variable x with $\Gamma, x : T, \Gamma' \vdash x : T$).
- The rule $(\preceq R)$ is only applied directly below (UProp), (UType), (var) and $(\preceq L)$

Proof: It has to be shown that applications of $(\preceq L)$ and $(\preceq R)$ can be moved upwards in the derivation tree, possibly leaving behind a "residue" in the form of an application of (whnf L) or (whnf R). The proof is by induction on derivations. We distinguish the following cases:

- $(\preceq L)$ occurs below (UProp) or (UType), for example as follows:

$$\dfrac{\dfrac{\Gamma, p : T', \Gamma' \ \mathit{valid}_c}{\Gamma, p : T', \Gamma' \vdash \mathit{Prop} : \mathit{Type}_0}\ (\text{UProp}) \quad \Gamma \vdash_N T : \mathit{Type} \quad T \preceq T'}{\Gamma, p : T, \Gamma' \vdash \mathit{Prop} : \mathit{Type}_0}\ (\preceq L)$$

  Using the assumption $\Gamma \vdash_N T : \mathit{Type}$, the derivation of $\Gamma, p : T', \Gamma' \ \mathit{valid}_c$ can be converted into a derivation of $\Gamma, p : T, \Gamma' \ \mathit{valid}_c$ in system ECC_N (Proposition 2.8), from which $\Gamma, p : T, \Gamma' \vdash \mathit{Prop} : \mathit{Type}_0$ can be derived by (UProp). Thus, $(\preceq L)$ can be eliminated altogether.

- An application of $(\preceq L)$ below (var) at a position p which is not the position of the subject x of the (var) rule can, by a similar reasoning, be eliminated. The derivation

$$\dfrac{\dfrac{\Gamma, p : T', x : A, \Gamma' \ \mathit{valid}_c}{\Gamma, p : T', x : A, \Gamma' \vdash x : A}\ (\text{var}) \quad \Gamma \vdash_N T : \mathit{Type} \quad T \preceq T'}{\Gamma, p : T, x : A, \Gamma' \vdash x : A}\ (\preceq L)$$

  becomes

$$\dfrac{\Gamma, p : T, x : A, \Gamma' \ \mathit{valid}_c}{\Gamma, p : T, x : A, \Gamma' \vdash x : A}\ (\text{var})$$

- If $(\preceq L)$ occurs below a Right-rule or a formation-rule, $(\preceq L)$ can directly be permuted above it.

- Assume $(\preceq L)$ occurs directly below another application of $(\preceq L)$. By induction hypothesis, it may be assumed that the upper application of $(\preceq L)$ is only at the "subject" position of a (var) rule. If both applications of $(\preceq L)$ are at the same position, these applications can be combined into one, by transitivity of $\preceq$. Otherwise, the lower application of $(\preceq L)$ can be permuted above the upper one.

- Assume $(\preceq L)$ occurs directly below (ΠL), and both rules are applied at the same position (otherwise, the rule applications can be permuted):

$$\dfrac{\dfrac{\Gamma, p : \Pi x : A.\ B[x], \Gamma' \vdash N_1 : A \quad \Gamma, p : \Pi x : A.\ B[x], \Gamma', p' : B[N_1] \vdash N_2 : G}{\Gamma, p : \Pi x : A.\ B[x], \Gamma' \vdash N_2\{p' := (p\ N_1)\} : G}\ (\preceq L)}{\Gamma, p : T, \Gamma' \vdash N_2\{p' := (p\ N_1)\} : G}\ (\Pi L)$$

  Here, $T \preceq \Pi x : A.\ B[x]$. By the definition of $\preceq$, T is of the form $\Pi x : A'.\ B'[x]$ for $A' \simeq A$ and $B'[x] \preceq B[x]$, or T is convertible to such a Π-abstraction. In the latter case, we may assume that $\Pi x : A'.\ B'[x]$, with the given properties, is a whnf of T.

  The above derivation can therefore be transformed to:

$$\dfrac{\dfrac{\dfrac{\Gamma, p : \Pi x : A.\ B[x], \Gamma' \vdash N_1 : A}{\Gamma, p : \Pi x : A'.\ B'[x], \Gamma' \vdash N_1' : A'} \quad \dfrac{\Gamma, p : \Pi x : A.\ B[x], \Gamma', p' : B[N_1] \vdash N_2 : G}{\Gamma, p : \Pi x : A'.\ B'[x], \Gamma', p' : B'[N_1'] \vdash N_2' : G}}{\Gamma, p : \Pi x : A'.\ B'[x], \Gamma' \vdash N_2'\{p' := (p\ N_1')\} : G}\ (\Pi L)}{\Gamma, p : T, \Gamma' \vdash N_2'\{p' := (p\ N_1')\} : G}\ (\text{whnf } L)$$

  If T is already a Π-abstraction, application of (whnf L) can be omitted. The inferences above the double lines in the left branch are applications of $(\preceq L)$ at position p and $(\preceq R)$. By the induction hypothesis, the $\preceq$-rule applications in the left branch can be propagated upwards in the derivation tree, yielding a derivation of $\Gamma, p : \Pi x : A'.\ B'[x], \Gamma' \vdash N_1' : A'$ having the properties stated in the proposition, where $N_1 \simeq N_1'$.

  Similarly, propagating application of $(\preceq L)$ at positions p and p' in the right branch, the derivation of $\Gamma, p : \Pi x : A.\ B[x], \Gamma', p' : B[N_1] \vdash N_2 : G$ can be turned into a derivation of $\Gamma, p : \Pi x : A'.\ B'[x], \Gamma', p' : B'[N_1'] \vdash N_2' : G$, with $N_2 \simeq N_2'$. Renewed application of (ΠL) and (whnf L) gives the desired result.

  The argument for an occurrence of $(\preceq L)$ below (ΣL) is similar.

- Applications of $(\preceq R)$ can obviously be permuted above all applications of Left-rules.

- As to permutations of $(\preceq R)$ above formation rules, we consider the following case, all other formation rules being similar:

$$\dfrac{\dfrac{\Gamma \vdash A : \mathit{Type}_j \quad \Gamma, x : A \vdash B : \mathit{Type}_j}{\Gamma \vdash \Pi x : A.B : \mathit{Type}_j}\ (\Pi\text{-Form}_2)}{\Gamma \vdash \Pi x : A.B : T}\ (\preceq R)$$

Here, $\mathit{Type}_j \preceq T$, thus T is Type_i for a $i > j$ or can be reduced to a term having this form. We assume the latter case, and by a similar reasoning as further above, transform the derivation into

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash A : \mathit{Type}_j}{\Gamma \vdash A : \mathit{Type}_i}\,(\preceq R) \quad \dfrac{\Gamma, x : A \vdash B : \mathit{Type}_j}{\Gamma, x : A \vdash B : \mathit{Type}_i}\,(\preceq R)}{\Gamma \vdash \Pi x : A.B : \mathit{Type}_i}\,(\Pi\text{-Form}_2)}{\Gamma \vdash \Pi x : A.B : T}\,(\text{whnf } R)$$

- For permutations of $(\preceq R)$ above Right-rules, we consider the case of (ΠR), the rule (ΣR) is treated similarly. Assume the deduction has the form

$$\dfrac{\dfrac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda\, x : A.M : \Pi x : A.B}\,(\Pi R)}{\Gamma \vdash \lambda\, x : A.M : T}\,(\preceq R)$$

  where $\Pi x : A.B \preceq T$. By a similar reasoning as above, this derivation can be converted into

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, x : A \vdash M : B}{\Gamma, x : A \vdash M : B'}\,(\preceq R)}{\Gamma, x : A' \vdash M : B'}\,(\preceq L)}{\Gamma \vdash \lambda\, x : A'.M : \Pi x : A'.B'}\,(\Pi R)}{\Gamma \vdash \lambda\, x : A'.M : T}\,(\text{whnf } R)$$

  Here, $\Pi x : A'.B' \preceq \Pi x : A.B$. Note that the term $\lambda\, x : A'.M$ resulting from this derivation is convertible to the term $\lambda\, x : A.M$ of the original derivation, because $A \simeq A'$.

□

Bibliography

[ACCL91] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, October 1991. 1.3.1, 4.3.4

[And86] Peter B. Andrews. *An introduction to mathematical logic and type theory: To truth through proof*. Academic Press, 1986. 1.3.2

[Bar84] H.P. Barendregt. *The Lambda Calculus*. Elsevier Science Publishers, 1984. 2.3.2, 2.3.2, 2.3.3

[Bar92] Henk Barendregt. *Handbook of Logic in Computer Science*, chapter Lambda Calculi with Types, pages 117–309. Clarendon Press, 1992. 3.3.1

[Bar98] B. Barras et al. *The Coq Proof Assistant Reference Manual, Version 6.2*. INRIA Rocquencourt – CNRS - ENS Lyon, May 1998. 1.1.5

[BBS+98] H. Benl, U. Berger, H. Schwichtenberg, M. Seisenberger, and W. Zuber. Proof theory at work: program development in the Minlog system. In W. Bibel and P. Schmitt, editors, *Automated Deduction — A Basis for Applications*. Kluwer Academic Publishers, 1998. 1.1.2

[BM79] R.S. Boyer and J.S. Moore. *A Computational Logic*. Academic Press, 1979. 5.2

[BSvH+93] Alan Bundy, Andrew Stevens, Frank van Harmelen, Andrew Ireland, and Alan Smaill. Rippling: a heuristic for guiding inductive proofs. *Artificial Intelligence*, (62):185–253, 1993. 5.2

[CH88] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, 1988. 1.1, 2.1

[Con86] R.L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986. 1.1.5, 1.3.2

[dB70] N. G. de Bruijn. The mathematical language AUTOMATH, its usage and some of its extensions. In *Symposium on automatic demostration*, volume 125 of *Lecture Notes in Mathematics*, pages 29–61, 1970. 1.1.5, 3.3.1

[dB72] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag. Math.*, 34:381–392, 1972. A.1.1

[Der85] Nachum Dershowitz. Termination. In *Proceedings of the First International Conference on Rewriting Techniques and Applications*, volume 202 of *Lecture Notes in Computer Science*, pages 180–224, Berlin, May 1985. Springer Verlag. 2.4

[DHK94] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Higher-order unification via explicit substitutions. Technical Report 94R243, CRIN, 1994. 2.7

[DHK95] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Higher-order unification via explicit substitutions. In Dexter Kozen, editor, *Proceedings of the 10th IEEE Symposium on Logic in Computer Science*, pages 366–374, 1995. Extended abstract. 1.3.3, 1.3.3, 2.6.1, 2.7, 4.3.3

[DHKP96] Gilles Dowek, Thérèse Hardin, Claude Kirchner, and Frank Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In *Joint International Conference and Symposium on Logic Programming (JICSLP'96)*, 1996. 1.3.3, 2.7, 4.3.4

[Dow93] Gilles Dowek. A complete proof synthesis method for the cube of type systems. *Journal of Logic and Computation*, 3(3):287–315, 1993. 1.3.2, 1.3.2, 4.3.1

[Ell89] Conal M. Elliott. Higher-order unification with dependent function types. In N. Dershowitz, editor, *Proc. 3rd Intl. Conf. on Rewriting Techniques and Applications*, volume 355 of *Lecture Notes in Computer Science*, pages 121–136. Springer Verlag, 1989. 1.1.5, 1.3.3, 4.3.5, 4.3.5, 4.3.5, 4.3.5

[FGT93] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. IMPS: An interactive mathematical proof system. *Journal of Automated Reasoning*, 11:213–248, 1993. 1.1.4, 1.1.5

[Fit83] Melvin Fitting. *Proof Methods for Modal and Intuitionistic Logics*. D. Reidel Publishing Company, 1983. 4.4.2

[Gal87] Jean H. Gallier. *Logic for Computer Science*. John Wiley & Sons, 1987. 4.4.1

[Gen34] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210 and 405–431, 1934. 1.3.2, 3.1.1

[Gir72] J.Y. Girard. *Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris 7, 1972. 1.1.3, 3.3.1

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, 1989. 1.1.3

[GM93] M.J.C. Gordon and T.F. Melham. *Introduction to HOL – A theorem proving environment for higher order logic*. Cambridge University Press, 1993. 1.1.5, 1.3.1

[GW88] Joseph Goguen and Timothy Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, August 1988. 1.1.4

[Har85] Robert Harper. *Aspects of the Implementation of Type Theory*. PhD thesis, Cornell University, April 1985. 1.3.2, 1.3.2

[Has97] *The Haskell Report, Version 1.4*, April 1997. Available from `http://www.haskell.org/`. 1.1.3

[Hel91] Leen Helmink. Resolution and type theory. *Science of Computer Programming*, 17:119–138, 1991. 1.1.5

[HHP87] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science*, pages 194–204, June 1987. 1.1.5, 1.3.2, 3.3.1

[Hin97] J. Roger Hindley. *Basic Simple Type Theory*, volume 42 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1997. 1.1.1

[HN88] S. Hayashi and H. Nakano. *PX: A Computational Logic*. MIT Press, 1988. 1.1.2

[HP91] Robert Harper and Robert Pollack. Type checking with universes. *Theoretical Computer Science*, 89:107–136, 1991. 2.1.1, 4.2

[Hue73] Gérard Huet. A mechanization of type theory. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, pages 139–146, 1973. 1.3.2, 1.3.2

[Hue75] Gérard Huet. A unification algorithm for typed lambda-calculus. *Theoretical Computer Science*, pages 27–57, 1975. 1.3.2, 1.3.2, 1.3.3, 2.7, 4.3.3, 4.3.5, 4.3.5, 4.3.5

[KBS91] Bernd Krieg-Brückner and Donald Sannella. Structuring specifications in-the-large and in-the-small: Higher-order functions, dependent types and inheritance in SPECTRAL. In *Proceedings of TAPSOFT'91*, volume 494 of *Lecture Notes in Computer Science*, pages 313–336, 1991. 1.1.4, 1.1.4

[Kle52a] S. C. Kleene. Permutability of inferences in Gentzen's Calculi LK and LJ. *Memoirs of the A.M.S*, 10, 1952. 4.4.3

[Kle52b] S.C. Kleene. *Introduction to Meta-Mathematics*. North Holland, 1952. 1.1.1, 1.1.2, 4.23

[Koh98] Michael Kohlhase. Higher-order automated theorem proving. In W. Bibel and P. Schmitt, editors, *Automated Deduction — A Basis for Applications*. Kluwer Academic Publishers, 1998. 1.3.2

[Les94] Pierre Lescanne. From $\lambda\sigma$ to $\lambda\nu$ a journey through calculi of explicit substitutions. In *Proceedings of POPL'94*, pages 60–69, January 1994. 1.3.1, 1.3.1

[Löb76] M. H. Löb. Embedding first order predicate logic in fragments of intuitionistic logic. *Journal of Symbolic Logic*, 41(4):705–718, Dec. 1976. 1.1.3

[LP92] Zhaohui Luo and Robert Pollack. *Lego Proof Development System: User's Manual*. University of Edinburgh, May 1992. 1.1.3, 1.1.5, 4.3

[Luo90] Zhaohui Luo. *An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, July 1990. 1.1.4, 2.1, 2.1, 2.1.3, 2.1.3, 2.1.4, 2.5.2, 2.5.2, 2.5.2, 2.5.2

[Luo94] Zhaohui Luo. *Computation and Reasoning*. Oxford University Press, 1994. 1.1.1, 2.1

[Mag95] Lena Magnusson. *The Implementation of ALF - a Proof Editor based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution*. PhD thesis, Chalmers University of Technology, 1995. 1.3.1, 1.3.1, 1.3.1

[Mel95] Paul-André Melliès. Typed λ-calculi with explicit substitutions may not terminate. In *Typed Lambda Calculi and Applications*, volume 902 of *Lecture Notes in Computer Science*, 1995. 1.3.1, 1.3.1, 1.3.1

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991. 4.3.4, 4.3.4

[Mil92] Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14:321–358, 1992. 2.7

[Mit96] John Mitchell. *Foundations for Programming Lanugages*. MIT Press, 1996. 1.1.1

[MN94] Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs*, volume 806 of *Lecture Notes in Computer Science*, pages 213–237, 1994. 1.1.5

[MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991. 4.4.3

[Muñ96] Cézar Muñoz. Confluence and preservation of strong normalisation in an explicit substitutions calculus (extended abstract). In *Proceedings of the 11th IEEE Symposium on Logic in Computer Science*, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press. 1.3.1, 1.3.1

[Muñ97] Cézar Muñoz. *Un calcul de substitutions pour la représentation de preuves partielles en théorie de types*. Thèse de doctorat, Université Paris 7, 1997. 1.1.5, 1.3.1, 1.3.1, 1.3.1

[NL96] George Necula and Peter Lee. Proof-carrying code. Technical Report CMU-CS-96-165, Carnegie Mellon University, September 1996. 1.1.2

[NM88] Gopalan Nadathur and Dale Miller. An overview of λProlog. In *Proc. Fifth International Logic Programming Conference*, pages 810–827, 1988. 1.1.5, 2.7

[NP95] Tobias Nipkow and Christian Prehofer. Type reconstruction for type classes. *Journal of Functional Programming*, 5(2):201–224, 1995. 1.1.5

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory*. Oxford University Press, 1990. 1.1.1

[OK95] Jens Otten and Christoph Kreitz. A connection based proof method for intuitionistic logic. In P. Baumgartner, R. Hähnle, and J. Posegga, editors, *Proceedings of TABLEAUX'95*, volume 918 of *Lecture Notes in Artificial Intelligence*, pages 122–137. Springer Verlag, 1995. 1.3.3

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *Proceedings of CADE-11*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer Verlag, October 1992. 1.1.5

[Ott97] Jens Otten. ileanTAP: An intuitionistic theorem prover. In D. Galmiche, editor, *Proceedings of TABLEAUX'97*, volume 1227 of *Lecture Notes in Artificial Intelligence*, pages 307–312. Springer Verlag, 1997. 1.3.3

[Pau89] Lawrence Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, 1989. 2.7

[Pau91] Larry Paulson. *ML for the working programmer*. Cambridge University Press, 1991. 1.1.3

[Pau94] Lawrence Paulson. *Isabelle - a generic theorem prover*, volume 828 of *Lecture Notes in Computer Science*. Springer Verlag, 1994. 1.1.5, 1.3.1, 2.7

[Pfe89] Frank Pfenning. Elf: A language for logic definition and verified meta-programming. In *Proceedings of the 4th IEEE Symposium on Logic in Computer Science*, pages 313–322. IEEE Computer Society Press, June 1989. 2.7

[Pfe91a] Frank Pfenning. Logic programming in the LF logical framework. In Huet and Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991. 1.1.5

[Pfe91b] Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Proceedings of the 6th IEEE Symposium on Logic in Computer Science*, pages 74–85. IEEE Computer Society Press, July 1991. 1.3.3, 4.3.4, 4.3.4, 4.3.5

[Pfe95] Holger Pfeifer. Eine reflexive Architektur zur Darstellung von Beweis- und Softwareentwicklungsschritten in Typentheorie. Diplomarbeit, Universität Ulm, 1995. 5.2

[Pit87] A. M. Pitts. Polymorphism is set theoretic, constructively. In D. H. Pitt, A. Poigné, and D. E. Rydeheard, editors, *Category Theory and Computer Science*, volume 283 of *Lecture Notes in Computer Science*, pages 12–39. Springer Verlag, 1987. 1.1.2

[PM89] Christine Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions*. PhD thesis, Université Paris 7, 1989. 1.1.2

[PMW93] Christine Paulin-Mohring and Benjamin Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 11:1–34, 1993. 1.1.2

[Pol94] Robert Pollack. *The Theory of LEGO – A proof checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1994. 1.1.3

[Pra65] Dag Prawitz. *Natural Deduction – A proof-theoretic study*. Almqvist & Wiksells, 1965. 2.1.4, 2.1.4

[Pre95] Christian Prehofer. *Solving Higher-Order Equations: From Logic to Programming*. PhD thesis, TU München, 1995. 2.7

[PW91] David Pym and Lincoln Wallen. *Logical Frameworks*, chapter Proof-search in the $\lambda\Pi$-calculus, pages 311–340. Cambridge University Press, 1991. 1.3.2

[Pym90] David Pym. *Proofs, Search and Computation in General Logic*. PhD thesis, University of Edinburgh, 1990. 1.3.2, 1.3.2, 1.3.3, 4.3.5, 4.3.5, 4.3.5, 4.3.5

[Rey84] John Reynolds. Polymorphism is not set-theoretic. In *Int. Symp. on Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 145–156. Springer Verlag, 1984. 1.1.2

[RSSB98] W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction — A Basis for Applications*. Kluwer Academic Publishers, 1998. 1.1.4

[Rue95] Harald Rueß. *Formal Meta-Programming in the Calculus of Constructions*. PhD thesis, Universität Ulm, 1995. 5.2

[SFH92] Dan Sahlin, Torkel Franzén, and Seif Haridi. An intuitionistic predicate logic theorem prover. *Journal of Logic and Computation*, 2(5):619–656, 1992. 1.3.3, 4.4.3

[Sha92] N. Shankar. Proof search in the intuitionistic sequent calculus. In *CADE-11*, volume 607 of *Lecture Notes in Computer Science*, 1992. 1.3.3, 4.4.3

[SJ94] Y. V. Srinivas and R. Jüllig. Specware: Formal support for composing software. Technical Report KES.U.94.5, Kestrel Institute, 1994. Updated version appeared in Proc. of the Conference on Mathematics of Program Construction, Kloster Irsee, Germany, July 1995. 1.1.4

[Sor96] Maria Sorea. Integration von Gleichheitsbeweisen in einen typentheoretischen Beweiser. Diplomarbeit, Universität Ulm, 1996. 4.4

[Spe92] The Munich Spectrum Group. The requirement and design specification language Spectrum. An informal introduction. Version 0.3. Technical report, TU München, May 1992. 1.1.4

[SS97] Martin Strecker and Maria Sorea. Integrating an equality prover into a software development system based on type theory. In G. Brewka, Ch. Habel, and B. Nebel, editors, *Proceedings KI'97*, volume 1303 of *Lecture Notes in Artificial Intelligence*, pages 147–158, 1997. 5.2

[Sta79] R. Statman. Intuitionistic propositional logic is polynomial-space complete. *Theoretical Computer Science*, 9:67–72, 1979. 4.4.3

[Tam96] Tanel Tammet. A resolution theorem prover for intuitionistic logic. In *Proceedings of CADE-13*, volume 1104 of *Lecture Notes in Computer Science*, pages 2–16. Springer Verlag, 1996. 1.3.3

[Tho91] Simon Thompson. *Type Theory and Functional Programming*. Addison-Wesley, 1991. 1.1.1

[TS96] A.S. Troelstra and H. Schwichtenberg. *Basic Proof Theory*, volume 43 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1996. 1.1.3, 4.4.3

[vHLS97] F.W. von Henke, M. Luther, and M. Strecker. TYPELAB: An environment for modular program development. In M. Dauchet M. Bidoit, editor, *Proceedings of TAPSOFT'97*, volume 1214 of *Lecture Notes in Computer Science*, pages 851–854, 1997. 1.1

[Wag95] Matthias Wagner. Entwicklung und Implementierung eines Beweisers für konstruktive Logik. Diplomarbeit, Universität Ulm, 1995. 4.4.1, 4.23

[Wei95] Klaus Weich. Beweissuche in intuitionistischer Logik. Diplomarbeit, Universität München, 1995. 4.4.3

[Wel94] Joe Wells. Typability and type checking in the second-order lambda calculus are equivalent and undecidable. In *Proceedings of the 9th IEEE Symposium on Logic in Computer Science*, pages 176–185, 1994. 1.1.3, 4.3

[Wir86] Martin Wirsing. Structured algebraic specifications: A kernel language. *Theoretical Computer Science*, 42:123–249, 1986. 1.1.4

Index

Zusammenfassung (Abstract in German)

Diese Dissertation beschäftigt sich mit interaktiver Beweiskonstruktion und automatisierter Beweissuche in Typentheorien, insbesondere dem Konstruktionskalkül (*Calculus of Constructions*) und Teilsystemen davon.

Typentheorien kombinieren eine funktionale Sprache, ein starkes Typsystem und eine Logik höherer Ordnung in einem einheitlichen Rahmen und eignen sich somit als Grundlage für Spezifikations- und Verifikationswerkzeuge. So lassen sich unter anderem Spezifikationen als Typen darstellen und Realisierungen und Verfeinerungen von Spezifikationen innerhalb des Kalküls selbst entwickeln und als korrekt nachweisen.

Dieser Ausdrucksmächtigkeit steht die Schwierigkeit gegenüber, geeignete Mechanismen zur Unterstützung von Deduktionsprozessen in Typentheorien bereitzustellen. Zur Lösung dieser Problematik leistet die vorliegende Dissertation einen zweifachen Beitrag: Es werden allgemeine Methoden untersucht, mit denen Beweise in Typentheorien entwickelt werden können; daraufhin werden spezielle Verfahren vorgestellt, die eine Automatisierung der Beweissuche in eingeschränkten Fragmenten des Konstruktionskalküls erlauben.

Im allgemeinen erfordert Beweisentwicklung in Typentheorien die Konstruktion eines Terms M, der in einem Kontext Γ von einem Typ A ist. In logischem Zusammenhang kann Γ als Menge von Hypothesen aufgefaßt werden und A als zu beweisende Aussage. In Erweiterung dessen enthält Γ in Typentheorien Variablendeklarationen, wodurch Bindungsstrukturen wie in funktionalen Programmiersprachen entstehen; zudem können Typen von vorher deklarierten Variablen abhängen (*dependent types*).

Für den zu konstruierenden Term M kann eine *Metavariable* eingeführt werden, die im Verlauf des Beweises sukzessive instanziiert wird. Es werden einige Probleme identifiziert, die bei einer naiven Verwendung von Metavariablen auftreten. Insbesondere führen Instanziierung von Metavariablen und Reduktion von Termen, je nach Reihenfolge der Anwendung, zu unterschiedlichen Ergebnissen und möglicherweise zu typ-inkorrekten Termen. Zur Lösung der

Probleme wird ein Kalkül mit Metavariablen und *expliziten Substitutionen* vorgeschlagen, dessen wesentlichste Idee es ist, die bei einer Reduktion anfallenden Substitutionen an Metavariablen aufzubewahren und sie erst bei deren Instanziierung auszuführen. Es wird nachgewiesen, daß der entstehende Kalkül wünschenswerte Eigenschaften wie Konfluenz und Termination der Reduktionsrelation sowie Entscheidbarkeit der Typisierung besitzt und zudem die erwähnten Probleme beseitigt.

Einer unmittelbaren Automatisierung der Beweisentwicklung steht im Wege, daß die Typisierungsrelation in Typentheorien üblicherweise in Form eines Kalküls des Natürlichen Schließens durch Angabe von Einführungs- und Eliminationsregeln definiert wird. Bei einer vom Beweisziel ausgehenden Rückwärtsanwendung dieser Regeln müssen unter Umständen Terme erraten werden, so daß ein auf diesen Regeln basierendes Verfahren impraktikabel ist. Stattdessen wird ein Sequenzenkalkül definiert, der die Einführungs- und Eliminationsregeln ersetzt durch Regeln, die (Typ-)Terme im Sukzedens bzw. Antezedens zerlegen. Dieses Vorgehen ist angelehnt an entsprechende Verfahren in der Prädikatenlogik, wird jedoch erschwert dadurch, daß die Reduktionsrelation in die Betrachtungen einbezogen werden muß. Der Sequenzenkalkül wird als korrekt in Bezug auf den Kalkül des Natürlichen Schließens nachgewiesen. Der Beweis der Vollständigkeit verwendet ein Schnitteliminations-Verfahren, das gegenüber dem aus der Prädikatenlogik bekannten substantiell erweitert werden muß, um das Fehlen bestimmter struktureller Regeln (wie z.B. Vertauschung von Hypothesen) zu kompensieren. Der Vollständigkeitsbeweis wird nicht für den gesamten Konstruktionskalkül durchgeführt, sondern nur für Fragmente, für die eine Maßfunktion auf Termen mit bestimmten Eigenschaften existiert. Es wird gezeigt, daß eine solche Maßfunktion gerade für die prädikativen Systeme des Lambda-Würfels, einer Klassifikation der Subsysteme des Konstruktionskalküls, angegeben werden kann.

Die zuvor beschriebenen Konzepte – Metavariablen, Kalkül mit expliziten Substitutionen und Sequenzenkalkül – bilden die Basis für praktikable Beweisverfahren, wie sie in dem vom Autor mitentwickelten Typelab-System implementiert sind. Durch Einführung von Metavariablen für existentiell quantifizierte Variablen und anschließende Unifikation kann ein noch bestehender Nichtdeterminismus des reinen Sequenzenkalküls abgemildert werden. Es werden durch Transformation des Sequenzenkalküls Anforderungen abgeleitet, die eine Unifikationsprozedur zu erfüllen hat, und es wird ein in diesem Sinne korrekter Unifikationsalgorithmus angegeben. Insbesondere wird sichergestellt, daß die durch Unifikation erhaltenen Lösungen typkorrekt sind. Der Algorithmus ist nicht vollständig bezüglich β-Äquivalenz, jedoch läßt sich eine Analogie herstellen zu den aus dem einfachen Lambda-Kalkül bekannten "Pattern". Die hier entwickelten Mechanismen eignen sich unter anderem für Beweissuche in

der intuitionistischen Prädikatenlogik, die sich direkt in den Konstruktionskalkül einbetten läßt. Vor allem werden die bei der Anwendung von Quantorenregeln relevanten Eigenvariablenbedingungen erfüllt von typkorrekten Instanziierungen von Metavariablen, wie sie durch Unifikation entstehen.

Lebenslauf Martin Strecker

17.3.1966		Geboren in Darmstadt
1972	- 1976	Grundschule
1976	- 1985	Gymnasium
1985	- 1990	Studium der Informatik mit Nebenfach Mathematik an der Technischen Hochschule Darmstadt
1990	- 1991	Studienjahr an dem Institut National Polytechnique de Grenoble / Frankreich
1991	- 1999	Wissenschaftlicher Angestellter an der Universität Ulm