Metamathematics in Coq

Metamathematica in Coq
(met een samenvatting in het Nederlands)

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit Utrecht
op gezag van de Rector Magnificus,
Prof. dr. W.H. Gispen,
ingevolge het besluit van het College voor Promoties
in het openbaar te verdedigen
op vrijdag 31 oktober 2003 des middags te 14:30 uur
door
Roger Dimitri Alexander Hendriks
geboren op 6 augustus 1973 te IJsselstein.

Promotoren:

Prof. dr. J.A. Bergstra (Faculteit der Wijsbegeerte, Universiteit Utrecht)

Prof. dr. M.A. Bezem (Institutt for Informatikk, Universitetet i Bergen)

Voor Nelleke en Ole.

Preface

The ultimate arbiter of correctness is *formalisability*. It is a widespread view amongst mathematicians that *correct* proofs can be written out completely formally. This means that, after 'unfolding' the layers of abbreviations and conventions on which the presentation of a mathematical proof usually depends, the validity of every inference step should be completely perspicuous by a presentation of this step in an appropriate formal language. When descending from the informal heat to the formal cold,[1] we can rely less on intuition and more on formal rules. Once we have convinced ourselves that those rules are sound, we are ready to believe that the derivations they accept are correct. Formalising reduces the reliability of a proof to the reliability of the means of verifying it ([60]).

Formalising is more than just filling-in the details, it is a creative and challenging job. It forces one to make decisions that informal presentations often leave unspecified. The search for formal definitions that, on the one hand, convincingly represent the concepts involved and, on the other hand, are convenient for formal proof, often elucidates the informal presentation.

Checking conformity to formal rules is something computers are good at and with their arrival the old dream of formalising mathematics has become feasible, at least in principle. Even though a proof may be large, a small verification program can check each inference step locally. Besides such *proof checkers*, there are systems that support (interactive) proof development. A *proof assistant* consists of both, it checks and supports. Theorem proving using a proof assistant is the interactive construction of explicit *proof objects*, which can be verified independently.

I will exploit the proof assistant Coq as a tool for the development of logic and metamathematics. Three aspects are thematic:

- Incorporating the logical technique of resolution to support reasoning in type-theoretical systems.

- Using reflection to enable manipulation of proof objects.

- A complete formalisation of new meta-theory.

[1] A refreshing trip!

My PhD research started five years ago. First I continued the work initiated in my master's thesis [31] on incorporating resolution based theorem proving in Coq. The research that led to an implementation of a tool which enables the use of Bliksem in Coq, is joint work with Marc Bezem and Hans de Nivelle. The results are presented in Chapter 1, a copy of our article [14] (though slightly modified to fit in the present thesis), which in turn is a modified and extended version of our conference paper [13]. We describe techniques to integrate resolution logic in type theory. Refutation proofs obtained by resolution are translated into λ-terms, using reflection and an encoding of resolution proofs in minimal logic. Thereby we obtain a verification procedure for resolution proofs, and, more importantly, we add the power of resolution theorem provers to interactive proof construction systems based on type theory. We introduce a novel representation of clauses in minimal logic such that the λ-representation of resolution steps is linear in the size of the premisses. A clausification algorithm, equipped with a correctness proof, is encoded in Coq.

After this project was finished, we learned from Gilles Dowek that Skolem function symbols can be eliminated from refutation proofs.[2] This follows from the conservativity of the Axiom of Choice over first-order classical logic, see [63] and [30]. In order to deal with proof transformations, I formalised predicate logic with explicit proof terms; the results are described in Chapter 2, which is a modified and extended version of [32]. Natural deduction for first-order logic is formalised in the proof assistant Coq, using De Bruijn indices ([19]) for variable binding. The main judgement is of the form $\Gamma \vdash d\ [:]\ \phi$, stating that d is a proof term of formula ϕ under hypotheses Γ; it can be viewed as a typing relation by the Curry–Howard–De Bruijn isomorphism. This relation is proved sound with respect to Coq's native logic and is amenable to the manipulation of both formulas and derivations. As an illustration, I define a reduction relation on proof terms with permutative conversions and prove the property of subject reduction.

I spent quite some time on the problem of implementing a 'deskolemiser', but did not manage to reach that goal. The invitation of Vincent van Oostrom to collaborate on new research concerning explicit scoping mechanisms in the λ-calculus, came as a welcome alternative. I decided to put the project of deskolemising aside, and spent the remaining time of my PhD scholarship on what we baptised the $\curlywedge$*-calculus*. Chapter 3 has been submitted for publication in the *Journal of Functional Programming*, and is the full version of the conference paper [34]. Central to this chapter is the reification of the notion of *scope* in the λ-calculus. To this end we extend the syntax of the λ-calculus with an end-of-scope operator $\curlywedge$. The idea is that an $\curlywedge x$ ends the scope of the *matching* λx above it (in the term tree). Accordingly, β-reduction is extended to the set of scoped λ-terms by performing *minimal* scope extrusion before performing replication as usual. We show confluence of the resulting scoped β-reduction. Confluence of β-reduction for the ordinary λ-calculus is obtained as a corollary,

[2] As a result, proofs obtained via the proposed method of refutation, clausification and resolution, would no longer depend on (instances of) the Axiom of Choice.

by extruding scopes *maximally* before forgetting them altogether. Only in this final forgetful step, α-equivalence is needed. All our proofs have been verified in Coq.

In the following sections we briefly introduce type theory and the system Coq, explain the idea of reflection, and motivate the design choices made in the first two chapters with respect to variable binding mechanisms and the format of hypothetical judgements.

Type Theory and Coq Type theory offers a powerful formalism for formalising mathematics and, in particular, for formalising meta-theory of calculi and deduction systems. Definitions, reasoning and computation are captured in an integrated way. The level of detail is such that the well-formedness of definitions and the correctness of derivations can be verified automatically. In a type-theoretical system, formalised mathematical statements are represented by types, and their proofs are represented by λ-terms. This strong correspondence between proofs and typed λ-terms is referred to as the Curry–Howard–De Bruijn isomorphism. The relation between a proof and the statement it verifies, can be viewed as the membership of an object in a set. The problem whether a is a proof of statement A reduces to checking whether the term a has type A. A constructive proof is, in effect, a program annotated with additional information (types), which is used for verification (type checking).

The logical framework of the proof assistant Coq ([66]) is the calculus of inductive constructions ([69]). Useful are the common proof techniques of structural induction, pattern matching and primitive recursion. The user is allowed to extend the type theory with inductive types. Dually, the reduction rules can be extended in a flexible way. An inductive type provides a principle of structural induction (inhabited by a λ-term automatically generated by the system). Functions whose domain is an inductive type, can be defined using case analysis over the possible constructors of the object and recursion.

The basic sorts in Coq are $*^p$ and $*^s$. An object M of type $*^p$ is a logical proposition and objects of type M are proofs of M. Objects of type $*^s$ are usual sets such as the set of natural numbers or lists. The typing relation is expressed by $t : T$, to be interpreted as 't belongs to set T' when $T : *^s$, and as 't is a proof of proposition T' when $T : *^p$. The primitive type constructor is the constructor of the product type $\Pi x{:}T.\,U$ and is called *dependent* if x occurs in U; if not, we write $T \to U$. The product type is used for logical quantification (implication) as well as for function spaces. This overloading witnesses the Curry–Howard–De Bruijn isomorphism. Scopes of Πs and λs extend to the right as far as brackets allow ($\to$ associates to the right). Furthermore, well-typed application is denoted by $(M\ N)$ and associates to the left.

In Coq, connectives are defined as inductive types, the constructors being the proof formators. For example, conjunction $A \wedge B$ is defined as the inductive type inhabited by pairs $\langle a, b\rangle$, where $a : A$ and $b : B$. The corresponding induction principle is inhabited by $\wedge_{\mathsf{ind}}$, a λ-term generated by the system:

$$\wedge_{\mathsf{ind}} : \Pi A, B, P{:}*^p.\,(A \to B \to P) \to A \wedge B \to P$$

which can be used to eliminate the $\wedge$. For instance, a proof of $A \wedge B \to B \wedge A$ can be constructed as follows:

$$(\wedge_{\mathsf{ind}}\ A\ B\ (B \wedge A)\ (\lambda a\!:\!A.\, \lambda b\!:\!B.\, \langle b, a\rangle))$$

Two-level Approach, Reflection In Chapters 1 and 2 we choose for a deep embedding in adopting a two-level approach for the treatment of arbitrary first-order languages. The idea is to represent first-order formulas as objects in an inductive set $o : *^s$, accompanied by an interpretation function $[\![_]\!]$ that maps these objects into $*^p$. The next paragraphs explain why we distinguish a higher (*meta-*, *logical*) level $*^p$ and a lower (*object-*, *computational*) level o.

The universe $*^p$ includes higher-order propositions; in fact it encompasses full impredicative type theory. As such, it is too large for our purposes. Moreover, Coq supplies only limited computational power on $*^p$; every connective is defined as the inductive set of proofs of propositions with that connective in the head. We need a way to grasp first-order formulas (Chapters 1 and 2) and natural deduction proofs (Chapter 2), so that they can be subject to syntactical manipulation. Moreover, we want the ability to reason about such objects, and prove logical properties about them.

A natural choice, then, is to define formulas and proof terms as inductive objects, equipped with the powerful computational device of higher-order primitive recursion.

Object-level formulas (type o) are related to the meta-level by means of an interpretation function $[\![_]\!] : o \to *^p$. Given a suitable signature, any first-order proposition $\phi : *^p$ will have a formal counterpart $p : o$ such that ϕ is convertible with $[\![p]\!]$, the interpretation of p. Thus, the first-order fragment of $*^p$ can be identified as the collection of interpretations of objects in o.

In Chapter 1, *reflection* is used for the proof construction of first-order formulas in $*^p$ in the following way. Let $\varphi : *^p$ be a first-order formula. Then there is some $\dot\varphi : o$ such that $[\![\dot\varphi]\!]$ is convertible with φ. Moreover, suppose we have proved:

$$T_{\mathsf{sound}} : \Pi p\!:\!o.\, [\![(T\ p)]\!] \to [\![p]\!]$$

for some function $T : o \to o$, typically a transformation to clausal form. Then, to prove φ it suffices to prove $[\![(T\ \dot\varphi)]\!]$. Matters are presented schematically in Figure 1.1 on page 4. We will discuss a concrete function T, for which we have proved the above. For this T, proofs of $[\![(T\ \dot\varphi)]\!]$ will be generated automatically.

In Chapter 2, proof terms are defined as syntactical objects in an inductive set. There, the main judgement is of the form $\Gamma \vdash d\ [:]\ \phi$; it is of type $*^p$. The structure of the proof of $\Gamma \vdash d\ [:]\ \phi$ is similar to the structure of d, as will be pointed out in the sequel. Furthermore, we prove that if $\Gamma \vdash d\ [:]\ \phi$, then $[\![\Gamma]\!] \to [\![\phi]\!]$, in other words we construct a λ-term M of the following type:

$$M : (\Gamma \vdash d\ [:]\ \phi) \to [\![\Gamma]\!] \to [\![\phi]\!]$$

One could say that an object d *reflects* the λ-term $(M\ H_d\ H_\Gamma) : [\![\phi]\!]$, where $H_d : (\Gamma \vdash d\ [:]\ \phi)$ and $H_\Gamma : [\![\Gamma]\!]$.

Deep versus shallow embeddings One of the design choices to be made is whether to use a deep or shallow embedding of the objects we need. When syntax and meaning of a language are described separately, the language is said to be deeply embedded. Sometimes it is more economic to use a shallow embedding, where representation and denotation of objects are identified (in other words: the interpretation function is the identity function). The disadvantage of a shallow embedding is that the syntactic structure cannot be exploited. In Chapter 1, first-order formulas are deeply embedded, whilst a shallow embedding is used for first-order terms. In combination with the use of higher-order abstract syntax to represent quantifiers (see the paragraph on variable binding below), this gives rise to several difficulties. For example, it's not possible to prove syntactical correctness of the described formula transformation in a formal way. In Chapter 2, we choose for a deep embedding of terms, formulas, and derivation terms, giving us full control over the defined constructs.

Variable Binding Several ways exist for representing binding operators (e.g. quantification over first-order terms, binding of assumption variables), of which we mention formalising binding with the use of *named variables*, *higher-order abstract syntax* and *De Bruijn indices*.

In informal practice, the so-called *variable convention* plays a crucial role; expressions that differ only in the names assigned to their bound variables are to be identified; $\forall x.\, \phi(x)$ is said to be *α-equivalent* to $\forall y.\, \phi(y)$. In mathematical contexts bound variables are chosen different from free variables. In the process of substitution this means the (often silent) renaming of bound variables.

Using names (e.g., natural numbers) to encode the link between a binder and the variable it binds, is technically hard work. On top of the 'natural' definition of formulas one needs to define explicitly α-equality. As pointed out in [57], the (unavoidable) use of side-conditions in the definition of substitution is problematic when it comes to computation. As the unfolding of definitions proceeds, the number of side-conditions increases exponentially. Another difficulty is that there is no canonical choice of a fresh variable, necessary, for example for satisfying the eigenvariable condition in the inference rule for introducing a universal quantifier. Moreover, for many applications one needs a way to distinguish free and bound variables.

The advantage of representing binders by the use of higher-order abstract syntax is that several binding mechanisms are handled by λ-abstraction. This is the approach taken in Chapter 1. Identification of α-convertible formulas now comes for free. Substitution on the object level is supported by β-reduction in the meta-language. One problem of this representation[3] is that it generates a class of terms that contains *too much*. In Chapter 1, first-order terms are shallowly embedded, the domain of discourse A, being a parameter set. Object-level quantifiers are $\dot{\exists}, \dot{\forall}$ mapping propositional functions of type $A \to o$ to

[3] A related problem is the conflict between higher-order abstract syntax and inductive definitions. A constructor of type $(o \to o) \to o$ cannot be accepted in an inductive definition, because of the negative (leftmost) occurrence of o. This problem is absent in the case of representing a first-order language.

propositions of type o. If we instantiate A with an inductive set, it is possible to construct anomalous objects (that no longer fit in the intended language) by making use of a case construct, e.g., $\dot\forall\,(\lambda x : A.\, \mathsf{Case}\ x\ \mathsf{of}\ \ldots)$. Several possibilities have been explored to overcome this problem (apart from rejecting higher-order abstract syntax altogether), but many of them seem to harm the 'directness' of induction principles.

In Chapter 2, variable bindings are formalised by the use of De Bruijn indices. The major advantage is that inductive definitions can be used in a direct way. The freely generated (structural) equality of inductively defined objects is the natural equality satisfying α-convertibility. Instead of static scoping as in named calculi, De Bruijn indexing provides a dynamic counting scheme. The involved algorithms are of a computational nature. Surely, there's more work for the programmer, but that's no reason not to do it.[4] The idea is to get rid of names altogether and replace a variable occurrence by a pointer to the corresponding binder. A variable is represented by a natural number which indicates the number of binders between the variable and its binder. In Chapter 2, we have constructors $\dot\forall\,, \dot\exists$ of type $o \to o$; the operational semantics prescribes that, e.g., $\dot\forall\,\dot\exists\,\phi(v_1, v_0)$ reads as $\forall x.\, \exists y.\, \phi(x, y)$.[5]

Analytic versus Synthetic Judgements Another design choice to be made, for the purpose of Chapter 2, is whether to localise derivations themselves. In the terminology of Martin-Löf, this is the distinction between *analytic* and *synthetic* judgements. Synthetic judgements are of the form $\Gamma \vdash \phi$ as opposed to analytic judgements $\Gamma \vdash d : \phi$, which carry their own evidence d. Objects d can be seen as λ-terms and formulas ϕ as classifying those λ-terms. Given our objective of building a 'tool' for manipulation of first-order proofs, the choice for analytic judgements is obvious. The advantage of analytic judgements is that we get more control over proofs and that such judgements are decidable, as will be shown in the sequel. We are able to perform computational proofs of lemmas about judgements, because instead of induction over a logical hypothesis $\Gamma \vdash \phi$, we can use structural recursion on a proof object. It has to be noted that, in the case of synthetic judgements, it's possible to view the constructors of $\vdash$ as constituting a λ-calculus. But those constructors have Γ and ϕ as arguments, which make them less practical to reason about or to manipulate.

[4] On the contrary, Coq is the best game in town; it's fun!

[5] De Bruijn counts from 1, we start counting from 0, consistently with the definition of $\mathbb{N}$.

Acknowledgements

The writing of this thesis has been a period of ups and downs. On the whole, it has been an enjoyable enterprise for which I am beholden to several people:

I am indebted to Jan Bergstra, my first promotor, whose support has been of vital importance. I recall him saying "You and your thesis are different entities", a freeing remark.[6] I enjoyed sharing the office with Jan, and our late night chess games were lots of fun.

I wish to express my gratitude to Marc Bezem, my second promotor and supervisor during the first two years of my PhD scholarship. Marc was my guide when I took my first steps in Academia. He suggested the topic of our joint research. He insisted, over and over again, to work out examples first, an invaluable strategy for the subject. His precision and clear presentation have been exemplary.

Cooperating with Hans de Nivelle and my visit to Saarbrücken have been both fruitful and pleasant.

I am grateful to Vincent van Oostrom for a very effective and pleasant cooperation during the last year of my PhD scholarship. I learnt a great deal from him.

I acknowledge Jaco van de Pol, who was willing to work through all the details of my thesis, in various stages of its completion. He provided constructive criticism.

I very much appreciate the supporting role of Freek Wiedijk, who has always been prepared to discuss my work.

I am thankful to Gilles Dowek for hosting my visit to Rocquencourt; it has been a useful and instructive fortnight.

Finally, I thank Herman Geuvers, Kees Middelburg, and Albert Visser for their valuable suggestions and comments on draft versions of this thesis.

Dimitri Hendriks, September 9, 2003.

[6] Actually he said: "Jij *bent* niet je proefschrift. Punt.".

Contents

1 Automated Proof Construction in Type Theory using Resolution **1**
1.1 Introduction 1
1.2 A Two-level Approach 3
1.3 Clausification and Correctness 6
1.3.1 Negation Normal Form 6
1.3.2 Skolemisation 7
1.3.3 Composing the Modules 9
1.4 Minimal Resolution Logic 10
1.5 Lifting to Predicate Logic 15
1.6 Examples 16
1.6.1 A small example 16
1.6.2 A medium scale example: Newman's Lemma 17

2 Proof Reflection in Coq **23**
2.1 Introduction 23
2.2 Objects 24
2.3 Recursive Patterns 27
2.4 Lifting and Substitution 29
2.5 Properties of De Bruijn Operations 32
2.6 Judgements 33
2.7 Admissible Rules 36
2.8 Translation to Coq's Native Logic 37
2.9 Free Variables 39
2.9.1 Free Variables, Finitely versus Infinitely Many 39
2.9.2 Free Variables and Substitution 39
2.10 Thinning and Substitution Lemmas 40
2.11 Soundness with respect to the Native Logic 40
2.12 Type Checking Function 42
2.13 Correctness of Type Checking Function 42
2.14 Unique Types 43
2.15 Proof Reduction 43
2.16 Subject Reduction 45
2.17 Conclusion and Future Research 47

3 ƛ **49**
3.1 Introduction 49
3.2 Preliminaries 54
3.3 α 55
3.3.1 $\lambda\alpha$ 55
3.3.2 ƛα 58
3.4 β 64
3.4.1 $\lambda\beta$ 64
3.4.2 ƛβ 65
3.4.3 α and β 74
3.4.4 Confluence of λ-calculus 76
3.5 Applications 81
3.6 Conclusion and Discussion 82

Chapter 1

Automated Proof Construction in Type Theory using Resolution

We provide techniques to integrate resolution logic with equality in type theory. The results may be rendered as follows.

- A clausification procedure in type theory, equipped with a correctness proof, all encoded using higher-order primitive recursion.
- A novel representation of clauses in minimal logic such that the λ-representation of resolution steps is linear in the size of the premisses.
- A translation of resolution proofs into lambda terms, yielding a verification procedure for those proofs.
- The power of resolution theorem provers becomes available in interactive proof construction systems based on type theory.

Authors: Marc Bezem, Dimitri Hendriks and Hans de Nivelle

1.1 Introduction

Type theory (= typed lambda calculus, with dependent products as most relevant feature) offers a powerful formalism for formalising mathematics. Strong points are: the logical foundation, the fact that proofs are first-class citizens and the generality which naturally facilitates extensions, such as inductive types. Type theory captures definitions, reasoning and computation at various levels in an integrated way. In a type-theoretical system, formalised mathematical statements are represented by types, and their proofs are represented by λ-terms. The problem whether a is a proof of statement A reduces to checking whether the term a has type A. Computation is based on a simple notion of rewriting. The level of detail is such that the well-formedness of definitions and the correctness of derivations can automatically be verified.

However, there are also weak points. It is exactly the appraised expressivity and the level of detail that makes automation at the same time necessary and difficult. Automated deduction appears to be mostly successful in weak systems, such as propositional logic and predicate logic, systems that practically fall short of formalising a larger body of mathematics. Apart from the problem of the expressivity of these systems, only a minor part of the theorems that can be expressed can actually be proved automatically. Therefore it is necessary to combine automated theorem proving with interactive theorem proving. Recently a number of proposals in this direction have been made. In [18] a two-level approach (called *reflection*) is used to develop in Coq a certified decision procedure for equations in abelian rings. In the same vein, [53] certifies ELAN traces in Coq. In [49] Otter is combined with the Boyer-Moore theorem prover. (A verified program rechecks proofs generated by Otter.) In [41] Gandalf is linked to HOL. (The translation generates scripts to be run by the HOL-system.) In [64], proofs are translated into Martin-Löf's type theory, for the Horn clause fragment of first-order logic. In the Omega system [38, 29] various theorem provers have been linked to a natural deduction proof checker. The purpose there is to automatically generate proofs from so called *proof plans*. Our approach is different in that we generate complete proof objects for both the clausification and the refutation part.

Resolution theorem provers, such as Bliksem [54], are powerful, but have the drawback that they work with normal forms of formulas, so-called clausal forms. Clauses are (universally closed) disjunctions of literals, and a literal is either an atom or a negated atom. The clausal form of a formula is essentially its Skolem-conjunctive normal form, which need not be exactly logically equivalent to the original formula. This makes resolution proofs hard to read and understand, and makes interactive navigation of the theorem prover through the search space very difficult. Moreover, optimised implementations of proof procedures are error-prone. It has occurred that systems that took part in the yearly theorem prover competition CASC had to withdraw afterwards, due to the fact that the system turned out unsound. In 1999 the system that otherwise would have won the MIX category was withdrawn, see [65].

In type theory, the proof generation capabilities suffer from the small granularity of the inference steps and the corresponding astronomic size of the search space. Typically, one hyperresolution step requires a few dozens of inference steps in type theory. In order to make the formalisation of a large body of mathematics feasible, the level of automation of interactive proof construction systems such as Coq [66], based on type theory, has to be improved.

We propose the following proof procedure. Identify a non-trivial step in a Coq session that amounts to a first-order tautology. Export this tautology to Bliksem, and delegate the proof search to the Bliksem inference engine. Convert the resolution proof to type theoretic format and import the result back in Coq. We stress the fact that the above procedure is as secure as Coq. Hypothetical errors (e.g. the clausification procedure not producing clauses, possible errors in the resolution theorem prover or the erroneous formulation of the lambda terms corresponding to its proofs) are intercepted because the resulting proofs

are type-checked by Coq. The security could be made independent of Coq by using another type-checker.

Most of the necessary meta-theory is already known. The negation normal form transformation can be axiomatised by classical logic. The prenex and conjunctive normal form transformations require that the domain is non-empty. Skolemisation can be axiomatised by so-called Skolem axioms, which can be viewed as specific instances of the Axiom of Choice. Higher-order logic is particularly suited for this axiomatisation: we get logical equivalence modulo classical logic plus the Axiom of Choice, instead of awkward invariants as equiconsistency or equisatisfiability in the first-order case.

Following the proof of the conservativity of the Axiom of Choice over first-order logic (without equality), see e.g. [63] (elaborated in [30]) and [58], Skolem functions and –axioms could be eliminated from resolution proofs, which would allow us to obtain directly a proof of the original formula, but currently we still make use of the Axiom of Choice.

This chapter is organised as follows. In Section 1.2 we set out a two-level approach and define a deep embedding to represent first-order logic.[1] Section 1.3 describes a uniform clausification procedure. We explain how resolution proofs are translated into λ-terms in Sections 1.4 and 1.5. Finally, the outlined constructions are demonstrated in Section 1.6.

1.2 A Two-level Approach

We choose for a deep embedding in adopting a two-level approach for the treatment of arbitrary first-order languages. The idea is to represent first-order formulas as objects in an inductive set $o : *^s$, accompanied by an interpretation function $[\![_]\!]$ that maps these objects into $*^p$.[2] The next paragraphs explain why we distinguish a higher (*meta-*, *logical*) level $*^p$ and a lower (*object-*, *computational*) level o.

The universe $*^p$ includes higher-order propositions; in fact it encompasses full impredicative type theory. As such, it is too large for our purposes. Given a suitable signature, any first-order formula $\varphi : *^p$ will have a formal counterpart $p : o$ such that φ equals $[\![p]\!]$, the interpretation of p. Thus the first-order fragment of $*^p$ can be identified as a collection of interpretations of objects in o.

Secondly, Coq supplies only limited computational power on $*^p$, whereas o, as every inductive set, is equipped with the powerful computational device of higher-order primitive recursion. This enables the syntactical manipulation of object-level propositions.

Reflection is used for the proof construction of first-order formulas in $*^p$ in the following way. Let $\varphi : *^p$ be a first-order formula. Then there is some $\dot{\varphi} : o$ such that $[\![\dot{\varphi}]\!]$ is convertible with φ.[3] Moreover, suppose we have proved:

$$T_{\mathsf{sound}} : \Pi p\!:\!o.\, [\![(T\ p)]\!] \rightarrow [\![p]\!]$$

[1] Cf. the discussion on deep vs. shallow embeddings in the preface.
[2] Both o as well as $[\![_]\!]$ depend on a fixed but arbitrary signature.
[3] The mapping $\dot{_}$ is a syntax-based translation outside Coq.

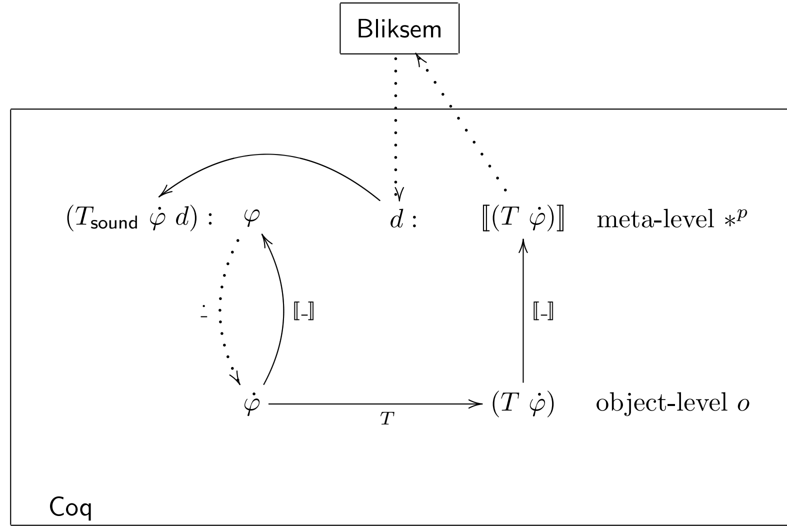

Figure 1.1: Schematic overview of the general procedure. Arrows correspond to application in Coq, dotted arrows are not performed by Coq. The term $[\![(T\ \dot\varphi)]\!]$ is computed by Coq and exported to Bliksem. Bliksem is to return a proof term d, which is imported back in Coq. Then $(T_{\mathsf{sound}}\ \dot\varphi\ d)$ is a proof of $[\![\dot\varphi]\!]$, and hence of φ.

for some function $T : o \to o$, typically a transformation to clausal form. Then, to prove φ it suffices to prove $[\![(T\ \dot\varphi)]\!]$. Matters are presented schematically in Figure 1.1. In Section 1.3 we discuss a concrete function T, for which we have proved the above. For this T, proofs of $[\![(T\ \dot\varphi)]\!]$ will be generated automatically, as will be described in Sections 1.4 and 1.5.

Object-level Propositions and the Reflection Operation

In Coq, we have constructed a general framework to represent first-order languages with multiple sorts. Bliksem is one-sorted, so we describe the setup for one-sorted signatures only.

The set o (formulas) defined in the present section depends on the signature, constituted by an arbitrary but fixed list of natural numbers, representing relation arities. This dependence remains implicit in the sequel. We start by giving some preliminary definitions.

Definition 1.2.1 *Given a set A,* lists *of type* (list A) *are defined by* $\square$ *and* $[a|l]$*, where* $a : A$ *and* l : (list A)*. Given a list* l : (list A)*, its* index set *is defined by*

$I_l = \{0, \ldots, |l| - 1\}$, where we write $|l|$ to denote the length of l.[4] *Furthermore, we write $l(i)$ for the element indexed by $i \in I_l$. The cartesian product A^n of n copies of a set A is defined by:*

$$A^0 = \mathbf{1} \qquad A^{n+1} = A \times A^n$$

where $\mathbf{1}$ is the unit set with sole inhabitant $\bullet$.

Note that the product $A \times B$ is the set of pairs (a, b) with $a : A$ and $b : B$. We shall use the following notational conventions regarding lists and tuples. Let $a, a_1, a_2, \ldots, a_n : A$. The sugared version of a list $[a_1|[a_2|\cdots|[a_n|\Box]\cdots]]$ is $[a_1, a_2, \ldots, a_n]$. Similarly, tuples $(a_1, (a_2, \ldots, (a_n, \bullet)\ldots))$ of type A^n are written $(a_1, a_2, \ldots, a_n)$; also, we simply use a instead of $(a, \bullet)$ of type A^1.

Next, we define object-level propositions.

Definition 1.2.2 *Assume a domain of discourse $A : *^s$ and let l_{rel} be a list of natural numbers representing arities. The set o of objects representing propositions is inductively defined as follows, where $p, q : o$, $p' : A \to o$, $x_1, \ldots, x_k : A$, $i : I_{l_{\mathsf{rel}}}$, and $l_{\mathsf{rel}}(i) = k$.*

$$o := R_i(x_1, \ldots, x_k) \mid \dot{\neg} p \mid p \dot{\to} q \mid p \dot{\wedge} q \mid p \dot{\vee} q \mid (\dot{\forall}\ p') \mid (\dot{\exists}\ p')$$

Note that $R : \Pi i : I_{l_{\mathsf{rel}}}.\, A^{l_{\mathsf{rel}}(i)} \to o$, we write R_i instead of $(R\ i)$. We use the dot-notation $\dot{}$ to distinguish the object-level constructors from Coq's predefined connectives. The constructors $\dot{\forall}$, $\dot{\exists}$ are typed $(A \to o) \to o$; they map propositional functions of type $A \to o$ to propositions of type o. This representation has the advantage that binding and predication are handled by λ-abstraction and λ-application. On the object-level, existential quantification of x in p (of type o, possibly containing occurrences of x) is written as $(\dot{\exists}\ (\lambda x\!:\!A.\, p))$. Although this representation suffices for our purposes, it causes some well-known difficulties. See [52, Sections 8.3, 9.2] and the preface for a further discussion.

For our purposes, a shallow embedding of function symbols is sufficient. We have not defined an inductive set term representing the first-order terms in A like we have defined o representing the first-order fragment of $*^p$. Instead, 'meta-level' terms of type A are taken as arguments of object-level predicates. Due to this shallow embedding, we cannot check whether variables have occurrences in a given term. Because of that, e.g., distributing universal quantifiers over conjuncts can yield dummy abstractions. These problems could be overcome by using De Bruijn indices (see [19]) for a deep embedding of terms in Coq, cf. Chapter 2.

Definition 1.2.3 *The* interpretation *function $[\![_]\!]$ is a canonical homomorphism recursively defined as follows. Assume a family of relations indexed over $I_{l_{\mathsf{rel}}}$.*

$$\mathcal{R} : \Pi i\!:\!I_{l_{\mathsf{rel}}}.\, A^{l_{\mathsf{rel}}(i)} \to *^p$$

[4] For a more formal definition (i.e. closer to the actual Coq implementation) of list indices, the reader is referred to Chapter 2, Definition 2.2.1.

We write $\mathcal{R}_i$ for $(\mathcal{R}\ i)$.

$$
\begin{array}{rcl}
[\![R_i(t_1,\ldots,t_k)]\!] & = & \mathcal{R}_i(t_1,\ldots,t_k) \\
[\![\dot{\neg} p]\!] & = & \neg[\![p]\!] \\
[\![p \mathrel{\dot{\rightarrow}} q]\!] & = & [\![p]\!] \rightarrow [\![q]\!] \\
[\![p \mathrel{\dot{\wedge}} q]\!] & = & [\![p]\!] \wedge [\![q]\!] \\
[\![p \mathrel{\dot{\vee}} q]\!] & = & [\![p]\!] \vee [\![q]\!] \\
[\![(\dot{\forall}\ p')]\!] & = & \Pi x\!:\!A.\, [\![(p'\ x)]\!] \\
[\![(\dot{\exists}\ p')]\!] & = & \exists x\!:\!A.\, [\![(p'\ x)]\!]
\end{array}
$$

We use $\wedge, \vee, \exists$ for Coq's predefined logical connectives. Note that '$\rightarrow$' (and 'Π') is used for both (dependent) function space as well as for logical implication (quantification); this overloading witnesses the Curry–Howard–De Bruijn isomorphism.

We do not have to worry about name conflicts when introducing a new $x : A$ for interpretation of formulas whose head constructor is a quantifier. Coq's binding mechanisms are internally based on De Bruijn indices (with a user-friendly tool showing named variables on top of it). In the above definitions of o, its constructors and of $[\![_]\!]$, the dependency on the signature (constituted by A, l_{rel} and $\mathcal{R}$) has been suppressed.

1.3 Clausification and Correctness

We describe the transformation to *clausal form* (see Section 1.4), which is realised on both levels. On the object-level, we define an algorithm $\mathsf{mcf} : o \rightarrow o$ that converts object-level propositions into their clausal form. On the meta-level, clausification is realised by a term $\mathsf{mcf}_{\mathsf{sound}}$, which (given the axiom of excluded middle and the axiom of choice) transforms a proof of $[\![(\mathsf{mcf}\ p)]\!]$ into a proof of $[\![p]\!]$.

The algorithm mcf consists of the subsequent application of the following functions: $\mathsf{nnf}, \mathsf{pnf}, \mathsf{cnf}, \mathsf{sklm}, \mathsf{duqc}, \mathsf{impf}$ standing for transformations to negation, prenex and conjunctive normal form, Skolemisation, distribution of universal quantifiers over conjuncts and transformation to implicational form, respectively. As an illustration, we describe the functions nnf and sklm.

1.3.1 Negation Normal Form

Concerning negation normal form, a recursive call like:

$$(\mathsf{nnf}\ \dot{\neg}(p \mathrel{\dot{\wedge}} q)) = (\mathsf{nnf}\ \dot{\neg}p) \mathrel{\dot{\vee}} (\mathsf{nnf}\ \dot{\neg}q)$$

is not primitive recursive, since $\dot{\neg}p$ and $\dot{\neg}q$ are not subformulas of $\dot{\neg}(p \mathrel{\dot{\wedge}} q)$. Such a call requires general recursion. Coq's computational mechanism is higher-order primitive recursion, which is weaker than general recursion but ensures universal termination.

Definition 1.3.1 *The function* $\mathsf{nnf} : o \to \mathsf{pol} \to o$ *makes use of the so-called polarity (⊕ or ⊖) of an input formula.*

$$
\begin{aligned}
(\mathsf{nnf}\ R_i(t_1,\ldots,t_k)\ \oplus) &= R_i(t_1,\ldots,t_k)\\
(\mathsf{nnf}\ R_i(t_1,\ldots,t_k)\ \ominus) &= \dot{\neg} R_i(t_1,\ldots,t_k)\\
(\mathsf{nnf}\ \dot{\neg} p\ \oplus) &= (\mathsf{nnf}\ p\ \ominus)\\
(\mathsf{nnf}\ \dot{\neg} p\ \ominus) &= (\mathsf{nnf}\ p\ \oplus)\\
(\mathsf{nnf}\ p_1 \dot{\to} p_2\ \oplus) &= (\mathsf{nnf}\ p_1\ \ominus)\ \dot{\vee}\ (\mathsf{nnf}\ p_2\ \oplus)\\
(\mathsf{nnf}\ p_1 \dot{\to} p_2\ \ominus) &= (\mathsf{nnf}\ p_1\ \oplus)\ \dot{\wedge}\ (\mathsf{nnf}\ p_2\ \ominus)\\
(\mathsf{nnf}\ p_1 \dot{\wedge} p_2\ \oplus) &= (\mathsf{nnf}\ p_1\ \oplus)\ \dot{\wedge}\ (\mathsf{nnf}\ p_2\ \oplus)\\
(\mathsf{nnf}\ p_1 \dot{\wedge} p_2\ \ominus) &= (\mathsf{nnf}\ p_1\ \ominus)\ \dot{\vee}\ (\mathsf{nnf}\ p_2\ \ominus)\\
(\mathsf{nnf}\ p_1 \dot{\vee} p_2\ \oplus) &= (\mathsf{nnf}\ p_1\ \oplus)\ \dot{\vee}\ (\mathsf{nnf}\ p_2\ \oplus)\\
(\mathsf{nnf}\ p_1 \dot{\vee} p_2\ \ominus) &= (\mathsf{nnf}\ p_1\ \ominus)\ \dot{\wedge}\ (\mathsf{nnf}\ p_2\ \ominus)\\
(\mathsf{nnf}\ (\dot{\forall}\ p')\ \oplus) &= (\dot{\forall}\ (\lambda x\!:\!A.\,(\mathsf{nnf}\ (p'\ x)\ \oplus)))\\
(\mathsf{nnf}\ (\dot{\forall}\ p')\ \ominus) &= (\dot{\exists}\ (\lambda x\!:\!A.\,(\mathsf{nnf}\ (p'\ x)\ \ominus)))\\
(\mathsf{nnf}\ (\dot{\exists}\ p')\ \oplus) &= (\dot{\exists}\ (\lambda x\!:\!A.\,(\mathsf{nnf}\ (p'\ x)\ \oplus)))\\
(\mathsf{nnf}\ (\dot{\exists}\ p')\ \ominus) &= (\dot{\forall}\ (\lambda x\!:\!A.\,(\mathsf{nnf}\ (p'\ x)\ \ominus)))
\end{aligned}
$$

In order to prove soundness of nnf we need the principle of excluded middle PEM, which we define in such a way that it affects the first-order fragment only (like o, PEM depends on the signature):

Definition 1.3.2

$$\mathsf{PEM} := \Pi p\!:\!o.\, [\![p]\!] \vee \neg [\![p]\!]$$

Lemma 1.3.1 *Assume* PEM*, then we have for all* $p : o$*:*

$$
\begin{aligned}
[\![p]\!] &\leftrightarrow [\![(\mathsf{nnf}\ p\ \oplus)]\!]\\
\neg[\![p]\!] &\leftrightarrow [\![(\mathsf{nnf}\ p\ \ominus)]\!]
\end{aligned}
$$

1.3.2 Skolemisation

Skolemisation of a formula means the removal of all existential quantifiers and the replacement of the variables that were bound by the removed existential quantifiers by new terms, that is, Skolem functions applied to the universally quantified variables whose quantifier had the existential quantifier in its scope. Instead of quantifying each of the Skolem functions, we introduce an index type $\mathcal{S}$, which may be viewed as a type for families of Skolem functions:

Definition 1.3.3

$$\mathcal{S} := \mathbb{N} \to \mathbb{N} \to \Pi n\!:\!\mathbb{N}.\, A^n \to A$$

A Skolem function, then, is a term $(f\ i\ j\ n) : A^n \to A$ with $f : \mathcal{S}$ and $i, j, n : \mathbb{N}$. Here, i and j are indices that distinguish the family members. If the output of nnf yields a conjunction, the remaining clausification steps are performed separately on the conjuncts. (This yields a significant speed-up in performance.) Index i denotes the position of the conjunct, j denotes the number of the replaced existentially quantified variable in that conjunct.

Definition 1.3.4 *The function* sklm *is defined as follows.*

$$\begin{aligned}
(\mathsf{sklm}\ f\ i\ j\ n\ t\ (\dot\forall\ p')) &= (\dot\forall\ (\lambda x{:}A.\, (\mathsf{sklm}\ f\ i\ j\ n+1\ (t,x)\ (p'\ x))))\\
(\mathsf{sklm}\ f\ i\ j\ n\ t\ (\dot\exists\ p')) &= (\mathsf{sklm}\ f\ i\ j+1\ n\ t\ (p'\ (f\ i\ j\ n\ t)))\\
(\mathsf{sklm}\ f\ i\ j\ n\ t\ p) &= p, \textit{ if } p \textit{ is neither } (\dot\forall\ p') \textit{ nor } (\dot\exists\ p')
\end{aligned}$$

Here and below (t, x) denotes the tuple typed A^{n+1} obtained by appending x to t. If the input formula is of the form $(\dot\forall\ p')$, then the quantified variable is added at the end of the so far constructed tuple t of universally quantified variables. In case the input formula matches $(\dot\exists\ p')$ with $p' : A \to o$ the term $(f\ i\ j\ n\ t)$ is substituted for the existentially quantified variable (the 'hole' in p') and index j is incremented. This substitution comes for free and is performed on the meta-level by β-reducing $(p'\ (f\ i\ j\ n\ t))$. The third case exhausts the five remaining cases. As we enforce input formulas of sklm to be in prenex normal form (via the definition of mcf), nothing remains to be done.

Lemma 1.3.2 *For all* $i : \mathbb{N}$ *and* $p : o$ *we have:*

$$A \to \mathsf{AC}_{\mathcal{S}} \to [\![p]\!] \to \exists f{:}\mathcal{S}.\, [\![(\mathsf{sklm}\ f\ i\ 0\ 0\ \bullet\ p)]\!]$$

In the above lemma, $A \to \cdots$ expresses the condition that A is non-empty, and below $a : A$ denotes a canonical inhabitant. $\mathsf{AC}_{\mathcal{S}}$ is a specific formulation of the Axiom of Choice, which allows us to form Skolem functions. Like PEM, $\mathsf{AC}_{\mathcal{S}}$ implicitly depends on the signature, that is, on A, l_{rel} and $\mathcal{R}$.

Definition 1.3.5

$$\begin{aligned}
\mathsf{AC}_{\mathcal{S}} := \ & \Pi\alpha{:}A \to \mathcal{S} \to o.\\
& (\Pi x{:}A.\, \exists f{:}\mathcal{S}.\, [\![(\alpha\ x\ f)]\!])\\
& \to \exists F{:}A \to \mathcal{S}.\, \Pi x{:}A.\, [\![(\alpha\ x\ (F\ x))]\!]
\end{aligned}$$

Note that $\mathsf{AC}_{\mathcal{S}}$ indeed follows from the more general:

$$\begin{aligned}
\mathsf{AC} := \ & \Pi A, B{:}*^s.\\
& \Pi P{:}A \to B \to *^p.\\
& (\Pi x{:}A.\, \exists y{:}B.\, (P\ x\ y))\\
& \to \exists f{:}A \to B.\, \Pi x{:}A.\, (P\ x\ (f\ x))
\end{aligned}$$

Let us inspect a crucial step in the proof of this lemma, which proceeds by induction on $p : o$. Consider the case that p is of the form $(\dot\forall\ p')$. Our induction hypothesis is:

$$\Pi x{:}A.\, [\![(p'\ x)]\!] \to \exists f{:}\mathcal{S}.\, [\![(\mathsf{sklm}\ f\ i\ 0\ 0\ \bullet\ (p'\ x))]\!]$$

Assume $\Pi x{:}A.\,[\![(p'\ x)]\!]$. Then we have:

$$\Pi x{:}A.\,\exists f{:}\mathcal{S}.\,[\![(\mathsf{sklm}\ f\ i\ 0\ 0\ \bullet\ (p'\ x))]\!]$$

By application of $\mathsf{AC}_{\mathcal{S}}$ we get:

$$\Pi x{:}A.\,[\![(\mathsf{sklm}\ (F\ x)\ i\ 0\ 0\ \bullet (p'\ x))]\!]$$

for some function $F : A \to \mathcal{S}$. Our goal is:

$$\exists g{:}\mathcal{S}.\,\Pi x{:}A.\,[\![(\mathsf{sklm}\ g\ i\ 0\ 1\ x\ (p'\ x))]\!]$$

The witnessing g is given by:

$$\begin{array}{rcl}(g\ i\ j\ 0\ \bullet) & = & a \\ (g\ i\ j\ n+1\ (x,t)) & = & (F\ x\ i\ j\ n\ t)\end{array}$$

Now

$$[\![(\mathsf{sklm}\ g\ i\ 0\ 1\ x\ (p'\ x))]\!]$$

follows from

$$[\![(\mathsf{sklm}\ (F\ x)\ i\ 0\ 0\ \bullet (p'\ x))]\!]$$

via Lemma 1.3.3, as for any $n : \mathbb{N}$, g behaves like $(F\ x)$ on any tail $t : A^n$.

Lemma 1.3.3 *For all $i, j_f, j_g, n_f, n_g : \mathbb{N}$, $t_f : A^{n_f}$, $t_g : A^{n_g}$, $p : o$, we have: if for all $m, n : \mathbb{N}, t : A^n$*

$$(f\ i\ j_f + m\ n_f + n\ (t_f, t)) = (g\ i\ j_g + m\ n_g + n\ (t_g, t))$$

then

$$[\![(\mathsf{sklm}\ f\ i\ j_f\ n_f\ t_f\ p)]\!] \to [\![(\mathsf{sklm}\ g\ i\ j_g\ n_g\ t_g\ p)]\!]$$

Here tuples $(t_f, t) : A^{n_f+n}$ and $(t_g, t) : A^{n_g+n}$ are the result of appending t to t_f and t_g, respectively.

1.3.3 Composing the Modules

Reconsider Figure 1.1 and substitute mcf for T. Given a suitable signature, from any first-order formula $\varphi : *^p$, we can compute the clausal form $[\![(\mathsf{mcf}\ \dot\varphi)]\!]$.

Theorem 1.3.1 *There exists a proof term* $\mathsf{mcf}_{\mathsf{sound}}$ *which validates clausification on the meta-level. More precisely:*

$$\mathsf{mcf}_{\mathsf{sound}} : \mathsf{PEM} \to \mathsf{AC}_{\mathcal{S}} \to A \to \Pi p{:}o.\,[\![(\mathsf{mcf}\ p)]\!] \to [\![p]\!]$$

The term $[\![(\mathsf{mcf}\ \dot\varphi)]\!]$ computes a format $C_1 \to \cdots \to C_n \to \bot$. Here $C_1, \ldots, C_n : *^p$ are universally closed clauses that will be exported to Bliksem, which constructs the proof term d representing a resolution refutation of these clauses (see Sections 1.4 and 1.5). Finally, d is type-checked in Coq. Section 1.6 demonstrates the outlined constructions.

The complete Coq-script generating the correctness proof of the clausification algorithm comprises ± 65 pages. It is available at [12].

1.4 Minimal Resolution Logic

There exist many representations of clauses and corresponding formulations of resolution rules. The traditional form of a clause is a disjunction of literals, that is, of atoms and negated atoms. Another form which is often used is that of a sequent, that is, the implication of a disjunction of atoms by a conjunction of atoms.

Here we propose yet another representation of clauses, as far as we know not used before. There are three main considerations.

- A structural requirement is that the representation of clauses is closed under the operations involved, such as instantiation and resolution.
- The Curry–Howard–De Bruijn correspondence is most direct between minimal logic ($\rightarrow$,$\forall$) and a typed lambda calculus with product types (with $\rightarrow$ as a special, non-dependent, case of Π). Conjunction and disjunction in the logic require either extra type-forming primitives and extra terms to inhabit these, or impredicative encodings.
- The λ-representation of resolution steps should preferably be linear in the size of the premisses.

These considerations have led us to represent a clause like:

$$L_1 \vee \cdots \vee L_p$$

by the following classically equivalent implication in minimal logic:

$$\overline{L}_1 \rightarrow \cdots \rightarrow \overline{L}_p \rightarrow \bot$$

Here $\overline{L}_i$ is the complement of L_i in the classical sense (i.e. double negations are removed). If C is the disjunctive form of a clause, then we denote its implicational form by $[C]$. As usual, these expressions are implicitly or explicitly universally closed.

A resolution refutation of given clauses $C_1, \ldots, C_n$ proves their inconsistency, and can be taken as a proof of the following implication in minimal logic:

$$C_1 \rightarrow \cdots \rightarrow C_n \rightarrow \bot$$

Here and below, 'minimal' refers to minimal logic, as we use no particular properties of $\bot$. In particular, 'minimal clause' refers to the representation in minimal logic, and not to any other kind of minimality. We are now ready for the definition of the syntax of minimal resolution logic.

Definition 1.4.1 *Let $\forall\vec{x}.\,\phi$ denote the universal closure of ϕ. Let* Atom *be the set of atomic propositions. We define the sets* Literal, Clause *and* MCF *of, respectively, literals, clauses and minimal clausal forms by the following abstract syntax:*

$$\begin{array}{rcl} \mathsf{Literal} & ::= & \mathsf{Atom} \mid \mathsf{Atom} \rightarrow \bot \\ \mathsf{Clause} & ::= & \bot \mid \mathsf{Literal} \rightarrow \mathsf{Clause} \\ \mathsf{MCF} & ::= & \bot \mid (\forall\vec{x}.\,\mathsf{Clause}) \rightarrow \mathsf{MCF} \end{array}$$

Next we elaborate the familiar inference rules for factoring, permuting and weakening clauses, as well as the binary resolution rule.

Factoring, Permutation, Weakening

Let C and D be clauses, such that C subsumes D propositionally, that is, any literal in C also occurs in D. Let $A_1, \ldots, A_p, B_1, \ldots, B_q$ be literals ($p, q \geq 0$) and write

$$[C] = A_1 \to \cdots \to A_p \to \bot$$

and

$$[D] = B_1 \to \cdots \to B_q \to \bot$$

assuming that for every $1 \leq i \leq p$ there is $1 \leq j \leq q$ such that $A_i = B_j$.

A proof of $[C] \to [D]$ is the following λ-term:

$$\lambda c\!:\![C].\, \lambda b_1\!:\!B_1 \ldots \lambda b_q\!:\!B_q.\, (c\ \pi_1\ \ldots\ \pi_p)$$

with $\pi_i = b_j$, where j is such that $B_j = A_i$.

Binary Resolution

In the traditional form of the binary resolution rule for disjunctive clauses we have premisses C_1 and C_2, containing one or more occurrences of a literal L and of $\overline{L}$, respectively. The conclusion of the rule, the resolvent, is then a clause D consisting of all literals of C_1 different from L joined with all literals of C_2 different from $\overline{L}$. This rule is completely symmetric with respect to C_1 and C_2.

For clauses in implicational form there is a slight asymmetry in the formulation of binary resolution. Let $A_1, \ldots, A_p, B_1 \ldots, B_q$ be literals ($p, q \geq 0$) and write

$$[C_1] = A_1 \to \cdots \to A_p \to \bot,$$

with one or more occurrences of the negated atom $A \to \bot$ among the A_i and

$$[C_2] = B_1 \to \cdots \to B_q \to \bot,$$

with one or more occurrences of the atom A among the B_j. Write the resolvent D as

$$[D] = D_1 \to \cdots \to D_r \to \bot$$

consisting of all literals of C_1 different from $A \to \bot$ joined with all literals of C_2 different from A. A proof of $[C_1] \to [C_2] \to [D]$ is the following λ-term:

$$\lambda c_1\!:\![C_1].\, \lambda c_2\!:\![C_2].\, \lambda d_1\!:\!D_1 \ldots \lambda d_r\!:\!D_r.\, (c_1\ \pi_1\ \ldots\ \pi_p)$$

For $1 \leq i \leq p$, π_i is defined as follows. If $A_i \neq (A \to \bot)$, then $\pi_i = d_k$, where k is such that $D_k = A_i$. If $A_i = A \to \bot$, then we put

$$\pi_i = \lambda a\!:\!A.\, (c_2\ \pi'_1\ \ldots\ \pi'_q),$$

with π'_j ($1 \leq j \leq q$) defined as follows. If $B_j \neq A$, then $\pi'_j = d_k$, where k is such that $D_k = B_j$. If $B_j = A$, then $\pi'_j = a$. It is easily verified that $\pi_i : (A \to \bot)$ in this case.

If $(A \to \bot)$ occurs more than once among the A_i, then $(c_1\ \pi_1\ \ldots\ \pi_p)$ need not be linear. This can be avoided by factoring timely. Even without factoring, a linear proof term is possible: by taking the following β-expansion of $(c_1\ \pi_1\ \ldots\ \pi_p)$ (with a' replacing copies of proofs of $(A \to \bot)$):

$$(\lambda a' \!:\! A \to \bot.\, (c_1\ \pi_1\ \ldots\ a'\ \ldots\ a'\ \ldots\ \pi_p))(\lambda a \!:\! A.\, (c_2\ \pi'_1\ \ldots\ \pi'_q))$$

This remark applies to the rules in the next subsections as well.

Paramodulation

Paramodulation combines equational reasoning with resolution. For equational reasoning we use the inductive equality of Coq. In order to simplify matters, we assume a fixed domain of discourse A, and denote equality of $s_1, s_2 \in A$ by $s_1 \approx s_2$.

Coq supplies us with the following terms:

$$\begin{array}{rcl} \mathsf{eqrefl} & : & \forall s \!:\! A.\ (s \approx s) \\ \mathsf{eqsubst} & : & \forall s \!:\! A. \forall P \!:\! A \to *^p.\ (P\ s) \to \forall t \!:\! A.\ (s \approx t) \to (P\ t) \\ \mathsf{eqsym} & : & \forall s_1, s_2 \!:\! A.\ (s_1 \approx s_2) \to (s_2 \approx s_1) \end{array}$$

As an example we define eqsym from eqsubst, eqrefl:

$$\lambda s_1, s_2 \!:\! A.\, \lambda h \!:\! (s_1 \approx s_2).\, (\mathsf{eqsubst}\ s_1\ (\lambda s \!:\! A.\, (s \approx s_1))\ (\mathsf{eqrefl}\ s_1)\ s_2\ h)$$

Paramodulation for disjunctive clauses is the rule with premiss C_1 containing the equality literal $t_1 \approx t_2$ and premiss C_2 containing literal $L[t_1]$. The conclusion is then a clause D containing all literals of C_1 different from $t_1 \approx t_2$, joined with C_2 with $L[t_2]$ instead of $L[t_1]$.

Let $A_1, \ldots, A_p, B_1 \ldots, B_q$ be literals ($p, q \geq 0$) and write

$$[C_1] = A_1 \to \cdots \to A_p \to \bot,$$

with one or more occurrences of the equality atom $t_1 \approx t_2 \to \bot$ among the A_i, and

$$[C_2] = B_1 \to \cdots \to B_q \to \bot,$$

with one or more occurrences of the literal $L[t_1]$ among the B_j. Write the conclusion D as

$$[D] = D_1 \to \cdots \to D_r \to \bot$$

and let l be such that $D_l = L[t_2]$. A proof of $[C_1] \to [C_2] \to [D]$ can be obtained as follows:

$$\lambda c_1 \!:\! [C_1].\, \lambda c_2 \!:\! [C_2].\, \lambda d_1 \!:\! D_1 \ldots \lambda d_r \!:\! D_r.\, (c_1\ \pi_1\ \ldots\ \pi_p)$$

If $A_i \neq (t_1 \approx t_2 \to \bot)$, then $\pi_i = d_k$, where k is such that $D_k = A_i$. If $A_i = (t_1 \approx t_2 \to \bot)$, then we want again that $\pi_i : A_i$ and therefore put

$$\pi_i = \lambda e\!:\!(t_1 \approx t_2).\,(c_2\ \pi'_1\ \ldots\ \pi'_q).$$

If $B_j \neq L[t_1]$, then $\pi'_j = d_k$, where k is such that $D_k = B_j$. If $B_j = L[t_1]$, then we also want that $\pi'_j : B_j$ and put (with $d_l : D_l = L[t_2]$)

$$\pi'_j = (\mathsf{eqsubst}\ t_2\ (\lambda s\!:\!A.\,L[s])\ d_l\ t_1\ (\mathsf{eqsym}\ t_1\ t_2\ e))$$

The term π'_j has type $L[t_1]$ in the context $e : (t_1 \approx t_2)$. The term π'_j contains an occurrence of eqsym because of the fact that the equality $t_1 \approx t_2$ comes in the wrong direction for proving $L[t_1]$ from $L[t_2]$. With this definition of π'_j, the term π_i has indeed type $A_i = (t_1 \approx t_2 \to \bot)$.

As an alternative, it is possible to expand the proof of eqsym in the proof of the paramodulation step.

Equality Factoring

Equality factoring for disjunctive clauses is the rule with premiss C containing equality literals $t_1 \approx t_2$ and $t_1 \approx t_3$, and conclusion D which is identical to C but for the replacement of $t_1 \approx t_3$ by $t_2 \not\approx t_3$. The soundness of this rule relies on $t_2 \approx t_3 \vee t_2 \not\approx t_3$.

Let $A_1, \ldots, A_p, B_1 \ldots, B_q$ be literals $(p, q \geq 0)$ and write

$$[C] = A_1 \to \cdots \to A_p \to \bot,$$

with equality literals $t_1 \approx t_2 \to \bot$ and $t_1 \approx t_3 \to \bot$ among the A_i. Write the conclusion D as

$$[D] = B_1 \to \cdots \to B_q \to \bot$$

with $B_{j'} = (t_1 \approx t_2 \to \bot)$ and $B_{j''} = (t_2 \approx t_3)$. We get a proof of $[C] \to [D]$ from

$$\lambda c\!:\![C].\,\lambda b_1\!:\!B_1 \ldots \lambda b_q\!:\!B_q.\,(c\ \pi_1\ \ldots\ \pi_p).$$

If $A_i \neq (t_1 \approx t_3 \to \bot)$, then $\pi_i = b_j$, where j is such that $B_j = A_i$. For $A_i = (t_1 \approx t_3 \to \bot)$, we put

$$\pi_i = (\mathsf{eqsubst}\ t_2\ (\lambda s\!:\!A.\,(t_1 \approx s \to \bot))\ b_{j'}\ t_3\ b_{j''}).$$

The type of π_i is indeed $t_1 \approx t_3 \to \bot$.

Note that the equality factoring rule is constructive in the implicational translation, whereas its disjunctive counterpart relies on the decidability of $\approx$. This phenomenon is well-known from the double negation translation.

Positive and Negative Equality Swapping

The positive equality swapping rule for disjunctive clauses simply swaps an atom $t_1 \approx t_2$ into $t_2 \approx t_1$, whereas the negative rule swaps the negated atom. Both versions are obviously sound, given the symmetry of $\approx$.

We give the translation for the positive case first and will then sketch the simpler negative case. Let C be the premiss and D the conclusion and write

$$[C] = A_1 \to \cdots \to A_p \to \bot,$$

with some of the A_i equal to $t_1 \approx t_2 \to \bot$, and

$$[D] = B_1 \to \cdots \to B_q \to \bot.$$

Let j' be such that $B_{j'} = (t_2 \approx t_1 \to \bot)$. The following term is a proof of $[C] \to [D]$.

$$\lambda c\!:\![C].\, \lambda b_1\!:\!B_1 \ldots \lambda b_q\!:\!B_q.\, (c\ \pi_1\ \ldots\ \pi_p)$$

If $A_i \neq (t_1 \approx t_2 \to \bot)$, then $\pi_i = b_j$, where j is such that $B_j = A_i$. Otherwise

$$\pi_i = \lambda e\!:\!(t_1 \approx t_2).\, (b_{j'}\ (\mathsf{eqsym}\ t_1\ t_2\ e))$$

such that also $\pi_i : (t_1 \approx t_2 \to \bot) = A_i$.

In the negative case the literals $t_1 \approx t_2$ in question are not negated, and we change the above definition of π_i into

$$\pi_i = (\mathsf{eqsym}\ t_2\ t_1\ b_{j'}).$$

In this case we have $b_{j'} : (t_2 \approx t_1)$ so that $\pi_i : (t_1 \approx t_2) = A_i$ also in the negative case.

Equality Reflexivity Rule

The equality reflexivity rule simply cancels a negative equality literal of the form $t \not\approx t$ in a disjunctive clause. We write once more the premiss

$$[C] = A_1 \to \cdots \to A_p \to \bot,$$

with some of the A_i equal to $t \approx t$, and the conclusion

$$[D] = B_1 \to \cdots \to B_q \to \bot.$$

The following term is a proof of $[C] \to [D]$:

$$\lambda c\!:\![C].\, \lambda b_1\!:\!B_1 \ldots \lambda b_q\!:\!B_q.\, (c\ \pi_1\ \ldots\ \pi_p).$$

If $A_i \neq (t \approx t)$, then $\pi_i = b_j$, where j is such that $B_j = A_i$. Otherwise $\pi_i = (\mathsf{eqrefl}\ t)$.

1.5 Lifting to Predicate Logic

Until now we have only considered inference rules without quantifications. In this section we explain how to lift the resolution rule to predicate logic. Lifting the other rules is very similar.

Recall that we must assume that the domain is not empty. Proof terms below may contain a variable $a : A$ as free variable. By abstraction $\lambda a : A$ we will close all proof terms. This extra step is necessary since $\forall a : A.\, \bot$ does not imply $\bot$ when the domain A is empty. This is to be compared to $\Box\bot$ being true in a blind world in modal logic.

Consider the following clauses

$$C_1 = \forall x_1, \ldots, x_p : A.\, [A_1 \vee R_1]$$

and

$$C_2 = \forall y_1, \ldots, y_q : A.\, [\neg A_2 \vee R_2]$$

and their resolvent

$$R = \forall z_1, \ldots, z_r : A.\, [R_1\theta_1 \vee R_2\theta_2]$$

Here θ_1 and θ_2 are substitutions such that $A_1\theta_1 = A_2\theta_2$ and $z_1, \ldots, z_r$ are all variables that actually occur in the resolvent, that is, in $R_1\theta_1 \vee R_2\theta_2$ after application of θ_1, θ_2. It may be the case that $x_i\theta_1$ and/or $y_j\theta_2$ contain other variables than $z_1, \ldots, z_r$; these are understood to be replaced by the variable $a : A$ (see above). It that case θ_1, θ_2 may not represent a *most general* unifier. For soundness this is no problem at all, but even completeness is not at stake since the resolvent is not affected. The reason for this subtlety is that the proof terms involved must not contain undeclared variables.

Using the methods of the previous sections we can produce a proof π that has the type

$$[A_1 \vee R_1]\theta_1 \to [\neg A_2 \vee R_2]\theta_2 \to [R_1\theta_1 \vee R_2\theta_2].$$

A proof of $C_1 \to C_2 \to R$ is obtained as follows:

$$\begin{array}{l} \lambda c_1 : C_1.\, \lambda c_2 : C_2.\, \lambda z_1 \ldots z_r : A. \\ \ (\pi\ (c_1\ (x_1\theta_1) \ldots (x_p\theta_1))\ (c_2\ (y_1\theta_2) \ldots (y_q\theta_2))) \end{array}$$

We finish this section by showing how to assemble a λ-term for an entire resolution refutation from the proof terms justifying the individual steps. Consider a Hilbert-style resolution derivation

$$C_1, \ldots, C_m, C_{m+1}, \ldots, C_n$$

with premisses $c_1 : C_1, \ldots, c_m : C_m$. Starting from n and going downward, we will define by recursion for every $m \le k \le n$ a term π_k such that

$$\pi_k[c_{m+1}, \ldots, c_k] : C_n$$

in the context extended with $c_{m+1} : C_{m+1}, \ldots, c_k : C_k$. For $k = n$ we can simply take $\pi_n = c_n$. Now assume π_{k+1} has been constructed for some $k \geq m$. The proof π_k is more difficult than π_{k+1} since π_k cannot use the assumption $c_{k+1} : C_{k+1}$. However, C_{k+1} is a resolvent, say of C_i and C_j for some $i, j \leq k$. Let d be the proof of $C_i \to C_j \to C_{k+1}$. Now define

$$\pi_k[c_{m+1}, \ldots, c_k] = (\lambda x{:}C_{k+1}.\pi_{k+1}[c_{m+1}, \ldots, c_k, x])(d\ c_i\ c_j) : C_n$$

The downward recursion yields a proof $\pi_m : C_n$ which is linear in the size of the original Hilbert-style resolution derivation. Observe that a forward recursion from m to n would yield the normal form of π_m, which could be exponential.

1.6 Examples

1.6.1 A small example

Let P be a property of natural numbers such that P holds for n if and only if P does not hold for any number greater than n. Does this sound paradoxical? It is contradictory. We have $P(n)$ if and only if $\neg P(n+1), \neg P(n+2), \neg P(n+3), \ldots$, which implies $\neg P(n+2), \neg P(n+3), \ldots$, so $P(n+1)$. It follows that $\neg P(n)$ for all n. However, $\neg P(0)$ implies $P(n)$ for some n, contradiction.

A closer analysis of this argument shows that the essence is not arithmetical, but relies on the fact that $<$ is transitive and serial. The argument is also valid in a finite cyclic structure, say $0 < 1 < 2 < 2$. This qualifies for a small refutation problem, which we formalise in Coq.

Let us adopt $\mathbb{N}$ as the domain of discourse. We declare a unary relation P and a binary relation $<$.

$$\begin{array}{rcl} P & : & \mathbb{N} \to *^p \\ < & : & \mathbb{N} \times \mathbb{N} \to *^p \end{array}$$

Let $l_{\mathsf{rel}} = [1, 2]$ be the corresponding list of arities. The relations are packaged by $\mathcal{R}$ of type $\Pi i{:}[0,1].\,\mathbb{N}^{l_{\mathsf{rel}}(i)} \to *^p$. We write $\mathcal{R}_i$ for $(\mathcal{R}\ i)$; note $I_{l_{\mathsf{rel}}} = [0, 1]$.

$$\mathcal{R}_0 = P \qquad \mathcal{R}_1 = <$$

We write $\dot{P}$ for R_0 and infix $\dot{<}$ for R_1 respectively.

Let us construct the formal propositions trans and serial, stating that $\dot{<}$ is serial and transitive. $\dot{\forall} x.\,\phi$ is syntactic sugar for $(\dot{\forall}\ (\lambda x{:}\mathbb{N}.\,\phi))$, likewise for $\dot{\exists}$.

$$\begin{array}{rcl} \mathsf{trans} & = & \dot{\forall} x, y, z.\,(x \mathrel{\dot{<}} y \mathrel{\dot{\wedge}} y \mathrel{\dot{<}} z) \mathrel{\dot{\to}} x \mathrel{\dot{<}} z \\ \mathsf{serial} & = & \dot{\forall} x.\,\dot{\exists} y.\,x \mathrel{\dot{<}} y \end{array}$$

We define foo.

$$\mathsf{foo} = \dot{\forall} x.\,(\dot{P}\ x) \mathrel{\dot{\leftrightarrow}} (\dot{\forall} y.\,x \mathrel{\dot{<}} y \mathrel{\dot{\to}} \dot{\neg}(\dot{P}\ y))$$

Furthermore, we define taut on the object-level, representing the example informally stated at the beginning of this section. (If the latter is denoted by φ, then $\mathsf{taut} = \dot{\varphi}$.)

$$\mathsf{taut} = (\mathsf{trans} \mathbin{\dot{\wedge}} \mathsf{serial}) \mathbin{\dot{\to}} \dot{\neg}\mathsf{foo}$$

Interpreting taut, that is $\beta\delta\iota$-normalising $[\![\mathsf{taut}]\!]$, results in 'taut without dots'.

We declare $\mathsf{pem} : \mathsf{PEM}$, $\mathsf{ac} : \mathsf{AC}_{\mathcal{S}}$ and use 0 to witness the non-emptiness of $\mathbb{N}$. We reduce the goal $[\![\mathsf{taut}]\!]$ using the result of Section 1.3, to the goal $[\![(\mathsf{mcf}\ \mathsf{taut})]\!]$. If we prove this latter goal, say by a term d, then

$$(\mathsf{mcf}_{\mathsf{sound}}\ \mathsf{pem}\ \mathsf{ac}\ 0\ \mathsf{taut}\ d) : [\![\mathsf{taut}]\!]$$

We compute the minimal clausal form (Definition 1.4.1) of taut by normalising the goal $[\![(\mathsf{mcf}\ \mathsf{taut})]\!]$.

$$\begin{array}{l}
[\![(\mathsf{mcf}\ \mathsf{taut})]\!] =_{\beta\delta\iota} \\
\quad (\Pi x, y, z : \mathbb{N}.\, x < y \to y < z \to (x < z \to \bot) \to \bot) \\
\quad\ \to (\Pi x : \mathbb{N}.\, (x < (f\ 1\ 0\ 1\ x) \to \bot) \to \bot) \\
\quad\ \ \to (\Pi x : \mathbb{N}.\, (x < (f\ 2\ 0\ 1\ x) \to \bot) \to ((P\ x) \to \bot) \to \bot) \\
\quad\ \ \ \to (\Pi x : \mathbb{N}.\, ((P\ (f\ 2\ 0\ 1\ x)) \to \bot) \to ((P\ x) \to \bot) \to \bot) \\
\quad\ \ \ \ \to (\Pi x, y : \mathbb{N}.\, (P\ x) \to x < y \to (P\ y) \to \bot) \\
\quad\ \ \ \ \ \to \bot
\end{array}$$

This is the minimal clausal form of the original goal. We refrained from exhibiting its proof d. All files can be found in [12].

1.6.2 A medium scale example: Newman's Lemma

A medium scale example is provided by the automation of Huet's [39] proof of Newman's Lemma (NL), a well known result in rewriting theory stating that a rewriting relation is confluent whenever it is both locally confluent and terminating. For a precise analysis we have to introduce some notions from rewriting theory.

Definition 1.6.1 *Let $\to$ be a binary relation on a set S and let $\twoheadrightarrow$ be the reflexive-transitive closure of $\to$.*

1. *We say that x is* confluent *if, for all $x_1, x_2 \in S$, $x \twoheadrightarrow x_1$ and $x \twoheadrightarrow x_2$ implies that $x_1 \twoheadrightarrow y$ and $x_2 \twoheadrightarrow y$ for some $y \in S$. In other words, any two diverging reductions starting from x can always be brought together. We say that $\to$ is* confluent *if every $x \in S$ is confluent.*

2. *We say that x is* locally confluent *if, for all x_1, x_2, $x \to x_1$ and $x \to x_2$ implies that $x_1 \twoheadrightarrow y$ and $x_2 \twoheadrightarrow y$ for some $y \in S$. Here the 'locality' lies in the fact that only diverging* one-step *reductions can be brought together. We say that $\to$ is* locally confluent *if every $x \in S$ is locally confluent.*

3. *We say that $\to$ is* terminating *if there is no infinite sequence $x_0 \to x_1 \to x_2 \to \cdots$ in S.*

NL provides an interesting test case for several reasons. First, it consists of a mix of first-order and higher-order aspects. The higher-order aspects are the transitive closure and the termination. This makes the identification of the first-order combinatorial core of the proof non-trivial. Second, the proof of Newman's Lemma is not completely trivial, as experienced by everybody seeing it for the first time. It will turn out to be a reasoning step that is just on the edge of what can be achieved by current theorem provers. As such the successful automation is very sensitive to the exact formalisation of the problem, the settings of the theorem prover and the machine on which one runs the proof. We admit that this is in some sense a disadvantage for an example. However, the aim of this example is to explore the borders of what is possible, and not to show-off how great the method is. It is to be expected that, with faster machines and better strategies for proof search, the automatic solution of problems of the size of NL will soon become routine. Moreover, the inductive approach to termination and the speed-up obtained by removing superfluous symmetries have a generality that goes beyond NL.

The classical proof of NL is by contradiction. Assume there is an x which is not confluent, that is, there exist $x_1, x_2 \in S$ such that $x \twoheadrightarrow x_1$ and $x \twoheadrightarrow x_2$ and no $y \in S$ exists such that $x_1 \twoheadrightarrow y$ and $x_2 \twoheadrightarrow y$. Since $\rightarrow$ is terminating, we may assume without loss of generality that x is an $\rightarrow$-maximal[5] non-confluent element. If not, there would be a non-confluent x' with $x \rightarrow x'$, and if that x' is not $\rightarrow$-maximal, then there would be a non-confluent x'' with $x' \rightarrow x''$ and so on, leading to a sequence contradicting the termination of $\rightarrow$. This part is difficult to explain, it actually uses the Axiom of Dependent Choice (DC). From the fact that x_1 and x_2 have no common reduct, it follows that we do not have $x = x_1$ or $x = x_2$, so there must exist intermediate points i_1, i_2 such that $x \rightarrow i_1 \twoheadrightarrow x_1$ and $x \rightarrow i_2 \twoheadrightarrow x_2$. To x and these intermediate points we can apply local confluence to obtain a common reduct of the intermediate points. By the maximality of x we can then complete the diagram in Figure 1.2 below. This is a contradiction and hence NL has been proved.

The formalisation of the classical argument requires higher-order logic (to express transitive closure) and three-sorted first-order logic: one sort for the set S, one for the natural numbers and one for infinite sequences of elements of S. An important improvement is obtained by taking the constructive reformulation of NL as point of departure. In this formulation the infinite sequences such as used in the definition of termination and in DC are avoided by using an inductively defined predicate called accessibility.

Definition 1.6.2 *Let $\rightarrow$ be a binary relation on a set S. The unary predicate $\mathsf{Acc}_{\rightarrow}$ is inductively defined as follows: if $\mathsf{Acc}_{\rightarrow}(y)$ for all $y \in S$ such that $x \rightarrow y$, then $\mathsf{Acc}_{\rightarrow}(x)$. By $\mathsf{Acc}_{\rightarrow}(S)$ we express that $\mathsf{Acc}_{\rightarrow}(x)$ for all $x \in S$.*

In other words, all $\rightarrow$-maximal elements are accessible, as well as all elements whose successors are all $\rightarrow$-maximal, and so on. An infinite sequence $x_0 \rightarrow$

[5] If the transitive closure of $\rightarrow$ is viewed as a *greater than* ordering, then it would be natural to speak of $\rightarrow$-*minimal* instead.

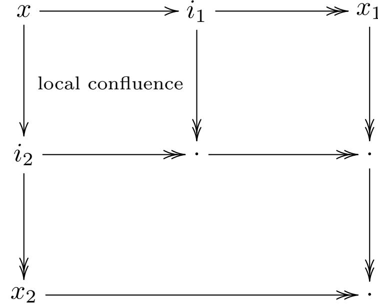

Figure 1.2: Diagram chase for confluence

$x_1 \to x_2 \to \cdots$ consists of elements that are not accessible. The reason is that they can be left out without violating the defining rule for Acc. In fact one can prove by classical logic and DC that $\to$ is terminating if and only if all elements of S are accessible, that is, if $\mathsf{Acc}_{\to}(S)$.

The advantages of using $\mathsf{Acc}_{\to}(S)$ instead of the traditional formulation of termination are three-fold.

- DC is not needed anymore in the proof of NL.
- The sorts for the natural numbers and for infinite sequences become obsolete.
- We can reason by induction on $\mathsf{Acc}_{\to}(x)$, the induction step being first-order.

These reasons above should motivate the following reformulation of NL: if $\mathsf{Acc}_{\to}(S)$, then confluence of $\to$ follows from local confluence.

We could have added a fourth advantage to the three advantages above, namely that the proof of NL in the formulation with the accessibility predicate can be done constructively. This would require resolution to be used bottom-up, in a forward reasoning style. We have not been able to generate a proof in this way. Instead, we had to appeal to classical logic by using resolution as a refutation procedure. The constructive proof is not more complicated than the classical one, it is actually shorter, but the relevant point here is that the search space for finding the proof in a bottom-up way appears to be larger than that for finding a proof in a more top-down, goal-oriented, way. We consider the situation in which there is a constructive proof, but for ill-understood reasons of efficiency only a classical proof can be found, as unsatisfactory.

We will sketch the constructive argument. By induction one proves that every accessible x is confluent. By $\mathsf{Acc}_{\to}(S)$ we then obtain confluence. The induction step we have to prove is that confluence is preserved under the inductive definition of $\mathsf{Acc}_{\to}$. In other words, we have to prove that x is confluent if the

induction hypothesis (IH) holds, that is, every y such that $x \to y$ is confluent. Assume IH and let $x_1, x_2 \in S$ such that $x \twoheadrightarrow x_1$ and $x \twoheadrightarrow x_2$. If $x = x_1$ or $x = x_2$ then x_2 or x_1 is a common reduct of x_1, x_2. Otherwise, actually appealing to the inductive definition of the reflexive–transitive closure, there exist intermediate points as in the classical proof above. Now a common reduct can be obtained in exactly the same way as in the classical proof, with IH replacing the $\to$-maximality of x. This proves the induction step.

The above proof of the induction step is completely first-order, provided that we replace the appeal to the inductive definition of $\twoheadrightarrow$ by some first-order sentences that trivially follow from the inductive definition of $\twoheadrightarrow$ and are sufficient for the proof.

$$\left.\begin{array}{l} = \text{ is reflexive and symmetric} \\ \twoheadrightarrow \text{ includes } = \text{ and } \to \text{ and is transitive} \\ \twoheadrightarrow \text{ is included in the union of } = \text{ and } \to\cdot\twoheadrightarrow \\ \to \text{ is locally confluent} \end{array}\right\} \Rightarrow \quad \begin{array}{l} \text{confluence} \\ \text{is } \mathsf{Acc}_{\to}\text{-inductive} \end{array}$$

Here the conclusion that confluence is $\mathsf{Acc}_{\to}$-inductive means that for all $x \in S$ confluence of x follows from confluence of all y such that $x \to y$. Note that we do not need transitivity of $=$. Moreover, $\to\cdot\twoheadrightarrow$ is the composition of $\to$ and $\twoheadrightarrow$.

We have formalised in Coq the proof of NL based on the above first-order tautology, with the intention to delegate the proof of the latter to a resolution theorem prover in the style of Section 1.6. The automatic clausification in Coq was a matter of seconds and resulted in 14 clauses. Both Otter and Bliksem were slow to refute the 14 clauses (without any tuning at least half an hour). The best results have been obtained with ordered hyperresolution in combination with unit-resulting resolution. The proof found by Otter is quite close to a 'human' proof by contradiction and the diagram chase in Figure 1.3. Bliksem managed to refute the corresponding set of clauses and to generate a proof object in the form of a lambda term. Although this lambda term has a considerable size (100 KByte), it could be type checked by Coq without any problem and included in a complete proof of NL in Coq. All files can be found in [12].

An obvious difficulty for proof search is the symmetry of the formulation of NL. Inspection of the proof shows that it is possible to distinguish between 'horizontal' and 'vertical' steps in the formulation of both confluence and local confluence. This leads to an asymmetrical version of Newman's Lemma (aNL), which can be proved by the same proof with all the steps properly labelled as either 'horizontal' or 'vertical'. NL can easily be recovered from aNL by removing the distinction. The advantage of the asymmetrical over the symmetrical formulation is that the search space for the proof is considerably reduced. For example, in the symmetrical case any step $x \to y$ leads to useless common reducts of y and y, which are avoided in the asymmetrical case. The asymmetrical analogues of confluence and local confluence are known in the literature as commutativity and weak commutativity, respectively.

Definition 1.6.3 *Let $\to_h$ and $\to_v$ be binary relations on a set S, with reflexive-transitive closures $\twoheadrightarrow_h$ and $\twoheadrightarrow_v$, respectively.*

1. *We say that x is* commutative *if, for all $x_1, x_2 \in S$, $x \twoheadrightarrow_h x_1$ and $x \twoheadrightarrow_v x_2$ implies that $x_1 \twoheadrightarrow_v y$ and $x_2 \twoheadrightarrow_h y$ for some $y \in S$. We say that $\to_h$ and $\to_v$* commute *if every $x \in S$ is commutative.*

2. *We say that x is* weakly commutative *if, for all $x_1, x_2 \in S$, $x \to_h x_1$ and $x \to_v x_2$ implies that $x_1 \twoheadrightarrow_v y$ and $x_2 \twoheadrightarrow_h y$ for some $y \in S$. We say that $\to_h$ and $\to_v$* commute weakly *if every $x \in S$ is weakly commutative.*

The precise statement of aNL is that $\to_h$ and $\to_v$ commute if they commute weakly, provided $\mathsf{Acc}_{\to_{hv}}(S)$. Here $\to_{hv}$ is the union of $\to_h$ and $\to_v$. A glance at Figure 1.3 tells us that we need the induction hypothesis both for i_1 with $x \to_h i_1$ and for i_2 with $x \to_v i_2$.

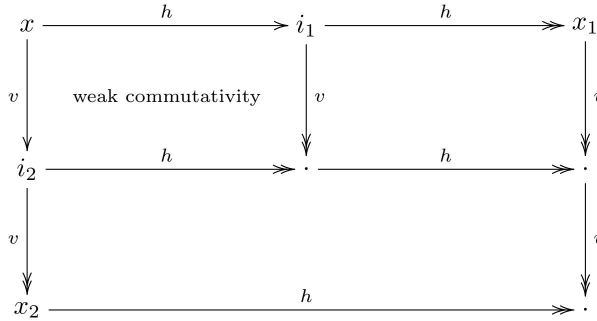

Figure 1.3: Diagram chase for commutativity

The proof of aNL follows the pattern of the proof of NL, but is based on the following first-order tautology:

$$\left.\begin{array}{l} = \text{ is reflexive and symmetric} \\ \twoheadrightarrow_h \text{ includes } = \text{ and } \to_h \text{ and is transitive} \\ \twoheadrightarrow_v \text{ includes } = \text{ and } \to_v \text{ and is transitive} \\ \twoheadrightarrow_h \text{ is included in the union of } = \text{ and } \to_h \cdot \twoheadrightarrow_h \\ \twoheadrightarrow_v \text{ is included in the union of } = \text{ and } \to_v \cdot \twoheadrightarrow_v \\ \to_h \text{ and } \to_v \text{ are weakly commutative} \end{array}\right\} \Rightarrow \begin{array}{l} \text{commutativity is} \\ \mathsf{Acc}_{\to_{hv}}\text{-inductive} \end{array}$$

Here the conclusion means that for all $x \in S$ commutativity of x follows from commutativity of all y such that $x \to_h y$ or $x \to_v y$.

We formalised in Coq the proof of aNL based on the above first-order tautology. Proof search in the asymmetrical case is about two orders of magnitude faster than in the symmetrical case. Again all files can be found in [12].

Summarising, the method can be put to work on medium scale examples. However, it is obvious that some human intelligence has been spent on stylising the proof before it could be automated. The techniques for proof search should be improved before the method can be scaled up any further.

Chapter 2

Proof Reflection in Coq

We formalise natural deduction for first-order logic in the proof assistant Coq, using De Bruijn indices for variable binding. The main judgement we model is of the form $\Gamma \vdash d$ [:] ϕ, stating that d is a proof term of formula ϕ under hypotheses Γ; it can be viewed as a typing relation by the Curry–Howard–De Bruijn isomorphism. This relation is proved sound with respect to Coq's native logic and is amenable to the manipulation of formulas and of derivations. As an illustration, we define a reduction relation on proof terms with permutative conversions and prove the property of subject reduction.

Author: Dimitri Hendriks

2.1 Introduction

We represent intuitionistic predicate logic in Coq [66], an interactive proof construction system that implements the calculus of inductive constructions [69], which is a type theory that provides inductive definitions. We adopt a two-level approach [8] in the sense that the native logic of the system is the meta-language in which we define and reason about our object-language. The object-language consists of a deep embedding of first-order terms, formulas and derivation terms. Derivation terms and formulas are related on the meta-level by definition of a deduction system for hypothetical judgements $\Gamma \vdash d$ [:] ϕ, that encapsulate their own evidence; d *inhabits* ϕ given context Γ. Several binding mechanisms are handled by De Bruijn indices [19].

The main contribution of our work is that we design an object language representing first-order logic, which can be used as a 'tool' for the manipulation of formulas and proofs. Moreover, via the so-called *reflection* operation [18] and the soundness result, it's possible to reason about the first-order fragment of the native logic itself.

The meta-theory of lambda calculi, type and proof systems of various kinds has already been treated quite extensively with the use of theorem provers, as shown, for example, in [50, 2, 10, 40, 6, 51, 7].

To our knowledge, this is the first complete (first-order) formalisation of

natural deduction for first-order logic using analytic judgements. Although we realise that our work is just standard first-order logic, and the results proved are the first basic ones, the strong point is the achievement of a Coq implementation, a library which can be reused in the future for several purposes, of which we mention:

- investigating the meta-theory of deduction systems, and
- proving correctness of proof-search algorithms.

Finally, we think that our work might serve as an overview on how to formalise logical systems in a theorem prover.

The complete development is formalised in Coq and can be retrieved from [33]; it's size is 116184 bytes, 4980 lines. The development time is approximately half a man-year.

For a brief introduction to type theory and the Coq proof assistant, an explanation of reflection and the two-level approach, and the motivation of our design choices with respect to variable binding mechanisms and the format of hypothetical judgements, the reader is referred to the preface.

This chapter is organised as follows. In Section 2.2 we introduce objects representing first-order terms, first-order formulas and derivation terms. In Sections 2.3 and 2.4, we define lifting and substitution. In Section 2.5 some basic algebraic properties of the defined De Bruijn operations are listed. The inference rules for hypothetical judgements are presented in Section 2.6. In Section 2.7 we show that the structural rules are admissible. In Section 2.8 we define the translation from object level formulas to their meta-level counterparts. In Section 2.9 we discuss an alternative set-up with a finite number of free variables instead of infinitely many; and we discuss an inconvenient aspect of substitution of free variables. Section 2.10 presents thinning and substitution lemmas about this translation function, necessary for the proof of soundness with respect to Coq's logic, given in Section 2.11. In Section 2.12, we specify a function which infers the type of (correct) proof terms. In Section 2.13 this function is proved correct with respect to the outlined inference system. As a corollary, derivation terms have unique types (Section 2.14). Section 2.15 serves as an example of how the defined machinery can be used to manipulate/transform proof terms; Prawitz's proof reduction rules are defined. In Section 2.16 we present soundness of types for the defined proof reduction, the property known as subject reduction. Finally, we conclude and discuss future work.

2.2 Objects

A logic is usually defined with respect to a signature determining its sorts, function symbols and predicate symbols. In our formalisation of intuitionistic predicate logic, we choose to deal with one sort only. We freed ourselves of the technical care multiple sorts would demand, simply for practical reasons.[1]

[1]It is well-known that sorts can be built-in artificially by using unary predicates.

The sets τ (terms), o (formulas) and π (proof terms), defined in the present section, depend on the signature—constituted by two arbitrary but fixed lists of natural numbers, representing function and relation arities. This dependence remains implicit in the sequel. We motivate this design choice.

A first (set-theoretical) attempt to formalise the dependency of an arbitrary signature would be to depart from an abstract set of function symbols, say F, along with an abstract function, say $\mathsf{arity} : F \to \mathbb{N}$. Given our aim to gain full control over the object language, however, this is unsatisfactory in several respects, of which we mention

- the undecidability of equality of terms, and
- the impossibility to check whether a function symbol occurs in a term.

Admittedly, one can add the necessary axioms. For example, we can assume the existence of a Boolean predicate $\mathsf{eqb} : F \to F \to \mathsf{bool}$. Of course, we then have to show consistency, but this doesn't seem to be problematic. What matters is the conceptual difference. With the approach chosen here, signatures are first class citizens and are finite, as opposed to the representation with F, arity and eqb. Such a representation of functions is what we called in the introduction a shallow embedding where the interpretation function of object-level function symbols to meta-level function symbols is the identity. As said before, the disadvantage of a shallow embedding is the impossibility to exploit the syntactical structure.[2]

Instead, we use an index set for function (as well as for relation) symbols.

Definition 2.2.1 *Given a set A, lists of type $\mathsf{list}(A)$ are defined by $\Box$ and $[a|l]$ where $a : A$ and $l : \mathsf{list}(A)$. Given a list $l : \mathsf{list}(A)$, its* index set *I_l is defined by the equations:*

$$I_\Box = \emptyset \qquad I_{[a|l']} = \mathbf{1} + I_{l'}$$

where $\emptyset$ is the empty set (i.e., without contructors), $\mathbf{1}$ the unit set (with sole inhabitant $\bullet$) and $A + B$ the disjoint sum of sets A and B, defined inductively by:

$$A + B := \mathsf{inl}(a) \mid \mathsf{inr}(b)$$

where $a : A$ and $b : B$. The application $l(i)$ computes the list element indexed by $i : I_l$, as defined by the following recursion:

$$\begin{array}{rcl} [a|l](\mathsf{inl}(\bullet)) & = & a \\ [a|l](\mathsf{inr}(i)) & = & l(i) \end{array}$$

For the sake of readability we set $I_l = \{0, \ldots, |l| - 1\}$, where $|l|$ denotes the length of l.

[2] A philosopher might raise his finger and swap things around: "A shallow embedding of objects in combination with full control over those objects, leads to well-known classical complications, such as diagonalisation, paradoxes and worse."

Definition 2.2.2 (Terms) *Assume a list of natural numbers, representing function arities.*

$$l_{\mathsf{fun}} : \mathsf{list}(\mathbb{N})$$

The set τ *of syntactic objects representing* first-order terms *is inductively defined by:*

$$\tau := v_n \mid f_i(t_1, \ldots, t_k)$$

where $n : \mathbb{N}$, $i : I_{l_{\mathsf{fun}}}$, $k = l_{\mathsf{fun}}(i)$ *and* $t_1, \ldots, t_k : \tau$. *It is to be understood that* $l_{\mathsf{fun}}(i)$ *computes the arity* k *of* f_i *(if* $k = 0$*, then* $f_i()$ *is a constant).*

Definition 2.2.3 (Formulas) *We assume a second list of natural numbers, representing relation arities.*

$$l_{\mathsf{rel}} : \mathsf{list}(\mathbb{N})$$

The set of objects o *representing* predicate logical formulas*, is defined by the following abstract syntax, where* $j : I_{l_{\mathsf{rel}}}$, $m = l_{\mathsf{rel}}(j)$ *and* $\phi, \chi : o$.

$$o := \dot{\top} \mid \dot{\bot} \mid R_j(t_1, \ldots, t_m) \mid \phi \dot{\to} \chi \mid \phi \dot{\wedge} \chi \mid \phi \dot{\vee} \chi \mid \dot{\forall} \phi \mid \dot{\exists} \phi$$

As usual, we write $\dot{\neg}\phi$ *as shorthand for* $\phi \dot{\to} \dot{\bot}$.

We use the following binding priorities for the connectives: $\dot{\forall}, \dot{\exists} > \dot{\wedge}, \dot{\vee} > \dot{\to}$ and let binary connectives associate to the right. For example, $\dot{\exists}\phi \dot{\vee} \dot{\exists}\chi \dot{\to} \dot{\exists}(\phi \dot{\vee} \chi)$ reads as $((\dot{\exists}\phi) \dot{\vee} (\dot{\exists}\chi)) \dot{\to} \dot{\exists}(\phi \dot{\vee} \chi)$.

In the sequel, when we write $f_i(t_1, \ldots, t_k)$ or $R_j(t_1, \ldots, t_m)$, we implicitly assume:

$$i : I_{l_{\mathsf{fun}}} \quad l_{\mathsf{fun}}(i) = k \qquad j : I_{l_{\mathsf{rel}}} \quad l_{\mathsf{rel}}(j) = m$$

We now turn to the definition of derivation terms, which can be seen as linear notations for two-dimensional proof trees.

Definition 2.2.4 (Derivations) *The syntactic class* π *of* proof terms *is defined by the grammar:*

$$\begin{aligned}\pi \quad := \quad & \top^+ \mid h_n \mid \bot^-(d, \phi) \mid \to^+(\phi, d) \mid \to^-(d, e) \\ & \mid \wedge^+(d, e) \mid \wedge_l^-(d) \mid \wedge_r^-(d) \mid \vee_l^+(\phi, d) \mid \vee_r^+(\phi, d) \mid \vee^-(d, e, f) \\ & \mid \forall^+(d) \mid \forall^-(t, d) \mid \exists^+(\phi, t, d) \mid \exists^-(d, e)\end{aligned}$$

where $n : \mathbb{N}$, $d, e, f : \pi$, $\phi : o$ *and* $t : \tau$. *Note that the* h_n *are assumption variables, as will become clear in the sequel.*

As an example, we depict the construction $\vee^-(d, e_1, e_2)$ in traditional natural deduction format:

$$\dfrac{\begin{array}{c} \\ \vdots\ (d) \\ \chi_1 \dot{\vee} \chi_2 \end{array} \quad \begin{array}{c} [\chi_1] \\ \vdots\ (e_1) \\ \phi \end{array} \quad \begin{array}{c} [\chi_2] \\ \vdots\ (e_2) \\ \phi \end{array}}{\phi}\ \vee^-$$

Some constructors ($\bot^-$, $\to^+$, $\vee_l^+$, $\vee_r^+$ and $\exists^+$) carry an argument of type o in order to have proof terms uniquely determine natural deductions, as will be shown in the sequel (see Section 2.12). Had we omitted the formula argument in, for example, $\to^+$, a term $\to^+(h_0)$ would be ambiguous in the sense that it serves as a proof term for $\phi \dot{\to} \phi$ for any $\phi : o$. Thus, we use explicit Church style typing. The formula argument in $\exists^+$ is required, because there is no inverse of substitution, that is, we cannot deduce ϕ from $\phi[t]$ (see Definition 2.6.1).

2.3 Recursive Patterns

Several object (of types o and π) transformations concerning (assumption as well as term) variables recursively descend in the same way. These recursive patterns are shared by abstracting from what should happen to terms or assumption variables.

The operations carry an argument storing the so-called *reference depth* of variables, because variables can only be 'grasped' (lifted, substituted, etc.) if we know at what reference depth they reside.

For objects in o, the reference depth increments when a quantifier is passed.

Definition 2.3.1 *Given $n : \mathbb{N}$, $g : \mathbb{N} \to \tau \to \tau$ and $\phi : o$, define* $\mathsf{map}_o(g, n, \phi)$ *as follows.*

$$\begin{array}{rcl}
\mathsf{map}_o(g, n, c) & = & c \textit{ for } c = \dot{\top}, \dot{\bot} \\
\mathsf{map}_o(g, n, R_j(t_1, \ldots, t_m)) & = & R_j(g(n, t_1), \ldots, g(n, t_m)) \\
\mathsf{map}_o(g, n, \phi \circ \chi) & = & \mathsf{map}_o(g, n, \phi) \circ \mathsf{map}_o(g, n, \chi) \textit{ for } \circ = \dot{\to}, \dot{\wedge}, \dot{\vee} \\
\mathsf{map}_o(g, n, \mathcal{Q}\phi) & = & \mathcal{Q}\mathsf{map}_o(g, n+1, \phi) \textit{ for } \mathcal{Q} = \dot{\forall}, \dot{\exists}
\end{array}$$

For proof terms, the reference depth of *term* variables v_i increments in the cases of $\forall^+$, $\exists^+$ (first argument) and $\exists^-$ (second argument).

Definition 2.3.2 *Given $g : \mathbb{N} \to \tau \to \tau$, $n : \mathbb{N}$ and $d : \pi$, the function* $\mathsf{map}^{\mathsf{v}}_\pi(g, n, d)$ *is defined by the following recursive equations.*

$$\begin{array}{rcl}
\mathsf{map}^{\mathsf{v}}_\pi(g, n, \top^+) & = & \top^+ \\
\mathsf{map}^{\mathsf{v}}_\pi(g, n, \bot^-(d, \phi)) & = & \bot^-(\mathsf{map}^{\mathsf{v}}_\pi(g, n, d), \mathsf{map}_o(g, n, \phi)) \\
\mathsf{map}^{\mathsf{v}}_\pi(g, n, h_i) & = & h_i \\
\mathsf{map}^{\mathsf{v}}_\pi(g, n, \to^+(\phi, d)) & = & \to^+(\mathsf{map}_o(g, n, \phi)), \mathsf{map}^{\mathsf{v}}_\pi(g, n, d)) \\
\mathsf{map}^{\mathsf{v}}_\pi(g, n, \to^-(d, e)) & = & \to^-(\mathsf{map}^{\mathsf{v}}_\pi(g, n, d), \mathsf{map}^{\mathsf{v}}_\pi(g, n, e)) \\
\mathsf{map}^{\mathsf{v}}_\pi(g, n, \wedge^+(d, e)) & = & \wedge^+(\mathsf{map}^{\mathsf{v}}_\pi(g, n, d), \mathsf{map}^{\mathsf{v}}_\pi(g, n, e)) \\
\mathsf{map}^{\mathsf{v}}_\pi(g, n, \wedge_l^-(d)) & = & \wedge_l^-(\mathsf{map}^{\mathsf{v}}_\pi(g, n, d)) \\
\mathsf{map}^{\mathsf{v}}_\pi(g, n, \wedge_r^-(d)) & = & \wedge_r^-(\mathsf{map}^{\mathsf{v}}_\pi(g, n, d)) \\
\mathsf{map}^{\mathsf{v}}_\pi(g, n, \vee_l^+(\phi, d)) & = & \vee_l^+(\mathsf{map}_o(g, n, \phi)), \mathsf{map}^{\mathsf{v}}_\pi(g, n, d)) \\
\mathsf{map}^{\mathsf{v}}_\pi(g, n, \vee_r^+(\phi, d)) & = & \vee_r^+(\mathsf{map}_o(g, n, \phi)), \mathsf{map}^{\mathsf{v}}_\pi(g, n, d))
\end{array}$$

$$\begin{array}{rcl}
\mathsf{map}^{\mathsf{v}}_{\pi}(g,n,\vee^{-}(d,e_1,e_2)) & = & \vee^{-}(\mathsf{map}^{\mathsf{v}}_{\pi}(g,n,d),\mathsf{map}^{\mathsf{v}}_{\pi}(g,n,e_1),\mathsf{map}^{\mathsf{v}}_{\pi}(g,n,e_2)) \\
\mathsf{map}^{\mathsf{v}}_{\pi}(g,n,\forall^{+}(d)) & = & \forall^{+}(\mathsf{map}^{\mathsf{v}}_{\pi}(g,n+1,d)) \\
\mathsf{map}^{\mathsf{v}}_{\pi}(g,n,\forall^{-}(t,d)) & = & \forall^{-}(g(n,t),\mathsf{map}^{\mathsf{v}}_{\pi}(g,n,d)) \\
\mathsf{map}^{\mathsf{v}}_{\pi}(g,n,\exists^{+}(\phi,t,d)) & = & \exists^{+}(\mathsf{map}_{o}(g,n+1,\phi)),g(n,t),\mathsf{map}^{\mathsf{v}}_{\pi}(g,n,d)) \\
\mathsf{map}^{\mathsf{v}}_{\pi}(g,n,\exists^{-}(d,e)) & = & \exists^{-}(\mathsf{map}^{\mathsf{v}}_{\pi}(g,n,d),\mathsf{map}^{\mathsf{v}}_{\pi}(g,n+1,e))
\end{array}$$

Note the increment of the reference depth of the formula argument in $\exists^+$. Consider the inference rule corresponding to $\exists^+$ given in Definition 2.6.1. The argument ϕ in term $\exists^+(\phi, t, d)$ has free variable v_0 ('from the outside'), for which the witnessing t is substituted in the type $\phi[t]$ of subterm d. This free variable should remain free; therefore the reference depth is incremented.

Also, the recursive pattern for proof term transformations concerning *assumption* variables will be reused several times in the sequel. The reference depth of assumption variables h_i is incremented in the cases of $\to^+$ (second argument), $\vee^-$ (second and third argument) and $\exists^-$ (second argument); that is, any time an extra hypothesis is added to the context (the inference rules of Definition 2.6.1 viewed bottom up).

Definition 2.3.3 *Let $g : \mathbb{N} \to \mathbb{N} \to \pi$, a function that returns a proof term given two natural numbers (reference depth, resp. index of assumption variable), $n : \mathbb{N}$, and $d : \pi$, then $\mathsf{map}^{\mathsf{h}}_{\pi}(g, n, d)$ is defined by the following recursive equations.*

$$\begin{array}{rcl}
\mathsf{map}^{\mathsf{h}}_{\pi}(g,n,\top^{+}) & = & \top^{+} \\
\mathsf{map}^{\mathsf{h}}_{\pi}(g,n,\perp^{-}(d,\phi)) & = & \perp^{-}(\mathsf{map}^{\mathsf{h}}_{\pi}(g,n,d),\phi) \\
\mathsf{map}^{\mathsf{h}}_{\pi}(g,n,h_i) & = & g(n,i) \\
\mathsf{map}^{\mathsf{h}}_{\pi}(g,n,\to^{+}(\phi,d)) & = & \to^{+}(\phi,\mathsf{map}^{\mathsf{h}}_{\pi}(g,n+1,d)) \\
\mathsf{map}^{\mathsf{h}}_{\pi}(g,n,\to^{-}(d,e)) & = & \to^{-}(\mathsf{map}^{\mathsf{h}}_{\pi}(g,n,d),\mathsf{map}^{\mathsf{h}}_{\pi}(g,n,e)) \\
\mathsf{map}^{\mathsf{h}}_{\pi}(g,n,\wedge^{+}(d,e)) & = & \wedge^{+}(\mathsf{map}^{\mathsf{h}}_{\pi}(g,n,d),\mathsf{map}^{\mathsf{h}}_{\pi}(g,n,e)) \\
\mathsf{map}^{\mathsf{h}}_{\pi}(g,n,\wedge^{-}_{l}(d)) & = & \wedge^{-}_{l}(\mathsf{map}^{\mathsf{h}}_{\pi}(g,n,d)) \\
\mathsf{map}^{\mathsf{h}}_{\pi}(g,n,\wedge^{-}_{r}(d)) & = & \wedge^{-}_{r}(\mathsf{map}^{\mathsf{h}}_{\pi}(g,n,d)) \\
\mathsf{map}^{\mathsf{h}}_{\pi}(g,n,\vee^{+}_{l}(\phi,d)) & = & \vee^{+}_{l}(\phi,\mathsf{map}^{\mathsf{h}}_{\pi}(g,n,d)) \\
\mathsf{map}^{\mathsf{h}}_{\pi}(g,n,\vee^{+}_{r}(\phi,d)) & = & \vee^{+}_{r}(\phi,\mathsf{map}^{\mathsf{h}}_{\pi}(g,n,d)) \\
\mathsf{map}^{\mathsf{h}}_{\pi}(g,n,\vee^{-}(d,e_1,e_2)) & = & \vee^{-}(\mathsf{map}^{\mathsf{h}}_{\pi}(g,n,d),\mathsf{map}^{\mathsf{h}}_{\pi}(g,n+1,e_1), \\
 & & \quad \mathsf{map}^{\mathsf{h}}_{\pi}(g,n+1,e_2)) \\
\mathsf{map}^{\mathsf{h}}_{\pi}(g,n,\forall^{+}(d)) & = & \forall^{+}(\mathsf{map}^{\mathsf{h}}_{\pi}(g,n,d)) \\
\mathsf{map}^{\mathsf{h}}_{\pi}(g,n,\forall^{-}(t,d)) & = & \forall^{-}(t,\mathsf{map}^{\mathsf{h}}_{\pi}(g,n,d)) \\
\mathsf{map}^{\mathsf{h}}_{\pi}(g,n,\exists^{+}(\phi,t,d)) & = & \exists^{+}(\phi,t,\mathsf{map}^{\mathsf{h}}_{\pi}(g,n,d)) \\
\mathsf{map}^{\mathsf{h}}_{\pi}(g,n,\exists^{-}(d,e)) & = & \exists^{-}(\mathsf{map}^{\mathsf{h}}_{\pi}(g,n,d),\mathsf{map}^{\mathsf{h}}_{\pi}(g,n+1,e))
\end{array}$$

2.4 Lifting and Substitution

The representation of variables by De Bruijn indices requires an extra operation called *lifting*.[3] Lifting increments the *free* variables in a formula.

We start with defining the operations of lifting and substitution, using side conditions. The implementation uses computationally more efficient definitions, as listed thereafter.

Definition 2.4.1 *We define* term lifting $\upharpoonleft_n t$ *by structural recursion on* $t : \tau$*, where* $n : \mathbb{N}$ *is the reference depth. The first* n *variables,* $v_0, \ldots, v_{n-1}$*, are assumed to be bound (this information being imported from functions calling* $\upharpoonleft_n t$*) and remain unchanged.*

$$\begin{array}{rcl} \upharpoonleft_n v_i & = & \begin{cases} v_i & \textit{if } i < n \\ v_{i+1} & \textit{if } i \geq n \end{cases} \\ \upharpoonleft_n f_i(t_1, \ldots, t_k) & = & f_i(\upharpoonleft_n t_1, \ldots, \upharpoonleft_n t_k) \end{array}$$

We write $\upharpoonleft t$ *to denote the lifting of all variables in* t*, shorthand for* $\upharpoonleft_0 t$*.*

Definition 2.4.2 Substitution of t' for v_n in t*, notation* $t[t']^n$*, is defined by recursion on the structure of* t*. Again,* n *is the reference depth, present in order to deal with substitution under binders. Thus, the first* n *variables should remain untouched. The term* t' *is lifted such that capture by binders is avoided. Indices greater than* n *are decremented, because substitution removes the original variable* v_n*.*

$$\begin{array}{rcl} v_i[t]^n & = & \begin{cases} v_i & \textit{if } i < n \\ \upharpoonleft_0^n t & \textit{if } i = n \\ v_{i-1} & \textit{if } i > n \end{cases} \\ f_i(t_1, \ldots, t_k)[t]^n & = & f_i(t_1[t]^n, \ldots, t_k[t]^n) \end{array}$$

where $\upharpoonleft_n^m t$ *is defined by* $\upharpoonleft_n^0 t = t$ *and* $\upharpoonleft_n^{m+1} t = \upharpoonleft_n^m (\upharpoonleft_n t)$*. We set* $t[t'] = t[t']^0$*.*

As mentioned, the side-conditions (if $i < n$, etc.) in the above definitions are inefficient. As the unfolding of definitions proceeds, the number of side-conditions increases exponentially. The implemented lifting and substitution functions are defined recursively and have no side-conditions $\upharpoonleft_n t$ is encoded as $\mathsf{lift_trm}(n, t)$ and $t[t']^n$ is encoded as $\mathsf{subst_trm}(n, t, t')$.[4]

$$\begin{array}{rcl} \mathsf{lift}(0, i) & = & i + 1 \\ \mathsf{lift}(n + 1, 0) & = & 0 \\ \mathsf{lift}(n + 1, i + 1) & = & \mathsf{lift}(n, i) + 1 \end{array}$$

[3] In the literature on explicit substitutions (e.g., [9]) the operation we call *lifting* here consists of two more primitive operations: *lifting* $\Uparrow$ of substitutions and the *shift* substitution $\uparrow$, which increments the indices in a term. Our $\upharpoonleft_n$ actually corresponds to $\Uparrow^n(\uparrow)$.

[4] We found these definitions in [57], where they are attributed to [2].

$$\begin{aligned}\mathsf{lift_trm}(n, v_i) &= v_{\mathsf{lift}(n,i)}\\ \mathsf{lift_trm}(n, f_j(t_1,\ldots,t_k)) &= f_j(\mathsf{lift_trm}(n,t_1),\ldots,\mathsf{lift_trm}(n,t_k))\end{aligned}$$

$$\begin{aligned}\mathsf{subst}(0,0,t) &= t\\ \mathsf{subst}(0,i+1,t) &= v_i\\ \mathsf{subst}(n+1,0,t) &= v_0\\ \mathsf{subst}(n+1,i+1,t) &= \mathsf{lift_trm}(0,\mathsf{subst}(n,i,t))\end{aligned}$$

$$\begin{aligned}\mathsf{subst_trm}(n, v_i, t) &= \mathsf{subst}(n,i,t)\\ \mathsf{subst_trm}(n, f_j(t_1,\ldots,t_k),t) &= f_j(\mathsf{subst_trm}(n,t_1,t),\\ &\qquad \ldots,\mathsf{subst_trm}(n,t_k,t))\end{aligned}$$

Next we define the lifting and substitution operations on formulas.

Definition 2.4.3 The lifting of $\phi : o$ for reference depth n, *notation* $\uparrow_n\phi$, *is defined as follows.*

$$\uparrow_n\phi = \mathsf{map}_o(\lambda m\!:\!\mathbb{N}.\,\lambda t\!:\!\tau.\uparrow_m t, n, \phi)$$

Let $\uparrow\phi$ *abbreviate* $\uparrow_0\phi$, *the increment of all* free *variables in* ϕ.

Definition 2.4.4 Substitution of $t : \tau$ for v_n[5] in $\phi : o$, *notation* $\phi[t]^n$, *is defined as follows.*

$$\phi[t]^n = \mathsf{map}_o(\lambda m\!:\!\mathbb{N}.\,\lambda u\!:\!\tau.\,u[t]^m, n, \phi)$$

For the inference system introduced in the next section, we also need the lifting and substitution operation on *contexts*. Contexts are defined by $\Box$ and $\Gamma;\phi$, where $\phi : o$ and Γ is a context.

Definition 2.4.5 Lifting of all free variables in context Γ, *given that the first* n *variables are bound, notation* $\uparrow_n\Gamma$, *is defined by:*

$$\begin{aligned}\uparrow_n\Box &= \Box\\ \uparrow_n(\Gamma;\phi) &= \uparrow_n\Gamma;\uparrow_n\phi\end{aligned}$$

Substitution of t for v_n in Γ, *written* $\Gamma[t]^n$, *is defined by:*

$$\begin{aligned}\Box[t]^n &= \Box\\ (\Gamma;\phi)[t]^n &= \Gamma[t]^n;\phi[t]^n\end{aligned}$$

Again, we write $\uparrow\Gamma$ *for* $\uparrow_0\Gamma$ *and* $\Gamma[t]$ *for* $\Gamma[t]^0$.

The type checking function, introduced in Section 2.12, requires the definition of the inverse of lifting: *projection*.

[5]The $n+1$-th free variable 'as seen from the outside'.

Definition 2.4.6 *We define* term projection, $\downarrow_n t$, *as follows.*

$$\begin{array}{rcl}\downarrow_n v_i & = & \begin{cases} v_i & \text{if } i \leq n \\ v_{i-1} & \text{if } i > n \end{cases} \\ \downarrow_n f_i(t_1, \ldots, t_k) & = & f_i(\downarrow_n t_1, \ldots, \downarrow_n t_k)\end{array}$$

Formula projection, $\downarrow_n \phi$, *is defined as follows.*

$$\downarrow_n \phi = \mathsf{map}_o(\lambda m : \mathbb{N}.\, \lambda t : \tau.\, \downarrow_m t, n, \phi)$$

Define $\downarrow \phi = \downarrow_0 \phi$.

Lemma 2.4.1 *For all* $n : \mathbb{N}$ *and* $\phi : o$, *we have that* $\downarrow_n \uparrow_n \phi = \phi$.

Also needed for Definition 2.12.1 is the ability to check whether a variable occurs free in a formula.

Definition 2.4.7 $v_n \in \mathsf{FV}(\phi)$ *is defined as follows.*[6]

$v_n \in \mathsf{FV}(R_j(t_1, \ldots, t_m))$	*if* $v_n \in t_i$ *for some* $1 \leq i \leq m$
$v_n \in \mathsf{FV}(\phi \circ \chi)$	*if* $v_n \in \mathsf{FV}(\phi)$ *or* $v_n \in \mathsf{FV}(\chi)$ *for* $\circ = \dot{\rightarrow}, \dot{\wedge}, \dot{\vee}$
$v_n \in \mathsf{FV}(\mathcal{Q}\phi)$	*if* $v_{n+1} \in \mathsf{FV}(\phi)$ *for* $\mathcal{Q} = \dot{\forall}, \dot{\exists}$

with $v_n \in t$ *defined by:*

$v_n \in v_m$	*if* $n = m$
$v_n \in f_i(t_1, \ldots, t_k)$	*if* $v_n \in t_j$ *for some* $1 \leq j \leq k$

Lemma 2.4.2 *For all* $n : \mathbb{N}$ *and* $\phi : o$, *we have that* $\uparrow_n \downarrow_n \phi = \phi$, *if* $v_n \notin \mathsf{FV}(\phi)$.

Lemma 2.4.3 *For all* $n : \mathbb{N}$, $t : o$, *we have* $v_n \notin \mathsf{FV}(\uparrow_n t)$.

Definition 2.4.8 *For* $n : \mathbb{N}$ *and* $d : \pi$, lifting of term variables in proof terms $\uparrow^{\mathsf{v}}_n d$ *is defined as follows.*

$$\uparrow^{\mathsf{v}}_n d = \mathsf{map}^{\mathsf{v}}_\pi(\lambda m : \mathbb{N}.\, \lambda t : \tau.\, \uparrow_m t, n, d)$$

Definition 2.4.9 *For* $n : \mathbb{N}$, $t : \tau$ *and* $d : \pi$, substitution of term variables in proof terms $d[t]^n_{\mathsf{v}}$ *is defined by*

$$d[t]^n_{\mathsf{v}} = \mathsf{map}^{\mathsf{v}}_\pi(\lambda m : \mathbb{N}.\, \lambda u : \tau.\, u[t]^m, n, d)$$

Definition 2.4.10 Lifting of assumption variables in proof terms *is defined by*

$$\uparrow^{\mathsf{h}}_n d = \mathsf{map}^{\mathsf{h}}_\pi(\lambda m : \mathbb{N}.\, \lambda i : \mathbb{N}.\, h_{\mathsf{lift}(m,i)}, n, d)$$

The function lift *is defined on page 29. Define* $\uparrow^{\mathsf{h}} d = \uparrow^{\mathsf{h}}_0 d$.

[6] We present $v_n \in \mathsf{FV}(\phi)$ as an inductive relation; it's implementation actually is a Boolean function.

Definition 2.4.11 Substitution of proof terms for assumption variables *is defined by*

$$\begin{array}{rcl}
\top^+[d']^n_{\mathsf{h}} & = & \top^+ \\
h_i[d']^n_{\mathsf{h}} & = & \begin{cases} h_i & \text{if } i < n \\ \uparrow^n_0 d' & \text{if } i = n \\ h_{i-1} & \text{if } i > n \end{cases} \\
\bot^-(d,\phi)[d']^n_{\mathsf{h}} & = & \bot^-(d[d']^n_{\mathsf{h}},\phi) \\
\to^+(\phi,d)[d']^n_{\mathsf{h}} & = & \to^+(\phi,d[d']^{n+1}_{\mathsf{h}}) \\
\to^-(d,e)[d']^n_{\mathsf{h}} & = & \to^-(d[d']^n_{\mathsf{h}},e[d']^n_{\mathsf{h}}) \\
\wedge^+(d,e)[d']^n_{\mathsf{h}} & = & \wedge^+(d[d']^n_{\mathsf{h}},e[d']^n_{\mathsf{h}}) \\
\wedge^-_l(d)[d']^n_{\mathsf{h}} & = & \wedge^-_l(d[d']^n_{\mathsf{h}}) \\
\wedge^-_r(d)[d']^n_{\mathsf{h}} & = & \wedge^-_r(d[d']^n_{\mathsf{h}}) \\
\vee^+_l(\phi,d)[d']^n_{\mathsf{h}} & = & \vee^+_l(\phi,d[d']^n_{\mathsf{h}}) \\
\vee^+_r(\phi,d)[d']^n_{\mathsf{h}} & = & \vee^+_r(\phi,d[d']^n_{\mathsf{h}}) \\
\vee^-(d,e_1,e_2)[d']^n_{\mathsf{h}} & = & \vee^-(d[d']^n_{\mathsf{h}},e_1[d']^{n+1}_{\mathsf{h}},e_2[d']^{n+1}_{\mathsf{h}}) \\
\forall^+(d)[d']^n_{\mathsf{h}} & = & \forall^+(d[\uparrow^{\mathsf{v}} d']^n_{\mathsf{h}}) \\
\forall^-(t,d)[d']^n_{\mathsf{h}} & = & \forall^-(t,d[d']^n_{\mathsf{h}}) \\
\exists^+(\phi,t,d)[d']^n_{\mathsf{h}} & = & \exists^+(\phi,t,d[d']^n_{\mathsf{h}}) \\
\exists^-(d,e)[d']^n_{\mathsf{h}} & = & \exists^-(d[d']^n_{\mathsf{h}},e[\uparrow^{\mathsf{v}} d']^{n+1}_{\mathsf{h}})
\end{array}$$

where $\uparrow^m_n d$ *is defined by* $\uparrow^0_n d = d$ *and* $\uparrow^{m+1}_n d = \uparrow^m_n(\uparrow^{\mathsf{h}}_n d)$*. It should be noted that* $h_i[d']^n_{\mathsf{h}}$ *is encoded without side-conditions, in a similar way as* $v_i[t]^n$ *(see Definition 2.4.2). Define* $d[d']_{\mathsf{h}} = d[d']^0_{\mathsf{h}}$*.*

Note that it is *not* correct to define $d[d']^n_{\mathsf{h}}$ by $\mathsf{map}^{\mathsf{h}}_{\pi}(\lambda m{:}\mathbb{N}.\,\lambda i{:}\mathbb{N}.\,h_i[d']^m_{\mathsf{h}},n,d)$[7], because all free variables v_i in d' have to be lifted to avoid capture of the first free variable by $\forall^+$ or $\exists^-$ (second argument).

2.5 Properties of De Bruijn Operations

We present some basic algebraic properties of the operations introduced in Section 2.4. Similar properties can be found in [7].[8] All five lemmas are proved for both $t:\tau$ as well as for $t:o$.[9] Furthermore $t',t_1,t_2:\tau$ and $n,m:\mathbb{N}$.

Lemma 2.5.1 (Permutation of lifting)

$$\uparrow_m(\uparrow_n t) = \uparrow_{n+1}(\uparrow_m t) \text{ if } m \le n$$

[7] As we did in [32] (though not in the Coq development).

[8] Where they are attributed to [40].

[9] Similar properties have been proved for lifting and substitution in proof terms, but these are not used in the sequel.

Lemma 2.5.2 (Simplification of substitution)

$$(\uparrow_n t)[t']^n = t$$

Lemma 2.5.3 (Commutation of lifting and substitution)

$$\uparrow_m (t[t']^n) = (\uparrow_m t)[t']^{n+1} \text{ if } m \leq n$$

Lemma 2.5.4 (Distribution of lifting over substitution)

$$\uparrow_{m+k} (t[t']^m) = (\uparrow_{m+k+1} t)[\uparrow_k t']^m$$

Lemma 2.5.5 (Distribution of substitution)

$$(t[t_1]^m)[t_2]^{m+k} = (t[t_2]^{m+k+1})[t_1[t_2]^k]^m$$

2.6 Judgements

We introduce *judgements* of the form $\Gamma \vdash d\ [:]\ \phi$, stating that d is a proof term of formula ϕ under hypotheses Γ.[10] Alternatively, the object d can be seen as a λ*-term of type* ϕ given variables h_i of type $\Gamma(i)$ for $0 \leq i < |\Gamma|$.

Definition 2.6.1 *The relation* $(\Gamma \vdash d\ [:]\ \phi) : *^p$ *is inductively defined by the following clauses. A context* Γ *is a list of formulas, where the rightmost element has index* 0*,* d, d_1, d_2, e_1, e_2 *are proof terms,* t *is a first-order term, and* $\phi, \phi_1, \phi_2, \chi$ *are formulas.*

$$\frac{}{\Gamma; \phi \vdash h_0\ [:]\ \phi} \qquad \frac{\Gamma \vdash h_i\ [:]\ \chi}{\Gamma; \phi \vdash h_{i+1}\ [:]\ \chi}$$

$$\frac{}{\Gamma \vdash \top^+\ [:]\ \dot{\top}} \qquad \frac{\Gamma \vdash d\ [:]\ \dot{\bot}}{\Gamma \vdash \bot^-(d, \phi)\ [:]\ \phi}$$

$$\frac{\Gamma; \phi \vdash d\ [:]\ \chi}{\Gamma \vdash \rightarrow^+(\phi, d)\ [:]\ \phi \dot{\rightarrow} \chi} \qquad \frac{\Gamma \vdash d\ [:]\ \phi \dot{\rightarrow} \chi \quad \Gamma \vdash e\ [:]\ \phi}{\Gamma \vdash \rightarrow^-(d, e)\ [:]\ \chi}$$

$$\frac{\Gamma \vdash d_1\ [:]\ \phi_1 \quad \Gamma \vdash d_2\ [:]\ \phi_2}{\Gamma \vdash \wedge^+(d_1, d_2)\ [:]\ \phi_1 \dot{\wedge} \phi_2}$$

$$\frac{\Gamma \vdash d\ [:]\ \phi_1 \dot{\wedge} \phi_2}{\Gamma \vdash \wedge_l^-(d)\ [:]\ \phi_1} \qquad \frac{\Gamma \vdash d\ [:]\ \phi_1 \dot{\wedge} \phi_2}{\Gamma \vdash \wedge_r^-(d)\ [:]\ \phi_2}$$

[10] We use this notation in order to distinguish '[:]' from ':', which is reserved for the typing relation of Coq.

$$\frac{\Gamma \vdash d\ [:]\ \phi_1}{\Gamma \vdash \vee_l^+(\phi_2, d)\ [:]\ \phi_1 \mathbin{\dot\vee} \phi_2} \qquad \frac{\Gamma \vdash d\ [:]\ \phi_2}{\Gamma \vdash \vee_r^+(\phi_1, d)\ [:]\ \phi_1 \mathbin{\dot\vee} \phi_2}$$

$$\frac{\Gamma \vdash d\ [:]\ \phi_1 \mathbin{\dot\vee} \phi_2 \quad \Gamma;\phi_1 \vdash e_1\ [:]\ \chi \quad \Gamma;\phi_2 \vdash e_2\ [:]\ \chi}{\Gamma \vdash \vee^-(d, e_1, e_2)\ [:]\ \chi}$$

$$\frac{\uparrow\Gamma \vdash d\ [:]\ \phi}{\Gamma \vdash \forall^+(d)\ [:]\ \dot\forall\phi} \qquad \frac{\Gamma \vdash d\ [:]\ \dot\forall\phi}{\Gamma \vdash \forall^-(t, d)\ [:]\ \phi[t]}$$

$$\frac{\Gamma \vdash d\ [:]\ \phi[t]}{\Gamma \vdash \exists^+(\phi, t, d)\ [:]\ \dot\exists\phi} \qquad \frac{\Gamma \vdash d\ [:]\ \dot\exists\chi \quad \uparrow\Gamma;\chi \vdash e\ [:]\ \uparrow\phi}{\Gamma \vdash \exists^-(d, e)\ [:]\ \phi}$$

In contrast to a formalisation with named variables (see [57]), there is a canonical choice of a fresh variable in the setting with De Bruijn indices, as, for example, needed in the rules $\forall^+$ and $\exists^-$. We simply lift all free variables (of Γ in the case of $\forall^+$, and of Γ and ϕ in the case of $\exists^-$), so that the first free variable becomes fresh.

The deduction system above defines the De Bruijn binding mechanism for assumption variables. Binders of assumption variables are $\to^+$, $\vee^-$ and $\exists^-$. For example, in $\to^+(\phi, \to^+(\chi, \wedge^+(h_1, h_0)))$, h_1 refers to the outer $\to^+$ and h_0 refers to the inner $\to^+$, as illustrated by the corresponding proof tree.

$$\cfrac{\cfrac{\cfrac{\phi;\chi \vdash h_1\ [:]\ \phi \quad \phi;\chi \vdash h_0\ [:]\ \chi}{\phi;\chi \vdash \wedge^+(h_1, h_0)\ [:]\ \phi \mathbin{\dot\wedge} \chi}}{\phi \vdash \to^+(\chi, \wedge^+(h_1, h_0))\ [:]\ \chi \mathbin{\dot\to} \phi \mathbin{\dot\wedge} \chi}}{\vdash \to^+(\phi, \to^+(\chi, \wedge^+(h_1, h_0)))\ [:]\ \phi \mathbin{\dot\to} \chi \mathbin{\dot\to} \phi \mathbin{\dot\wedge} \chi}$$

Note that $\vee^-$ and $\exists^-$ don't bind assumption variables in their first argument, since in the subtrees corresponding to those arguments no assumption is introduced (travelling bottom-up) into the context. For example, in

$$\to^+(\phi \mathbin{\dot\vee} \chi, \vee^-(h_0, \vee_r^+(\chi, \underline{h_0}), \vee_l^+(\phi, \underline{h_0})))$$

only the underlined ocurrences of h_0 are bound by the constructor $\vee^-$; the other one (referring to $\phi \mathbin{\dot\vee} \chi$) is bound by the constructor $\to^+$. The corresponding proof tree is:

$$\cfrac{\cfrac{\phi \mathbin{\dot\vee} \chi \vdash h_0\ [:]\ \phi \mathbin{\dot\vee} \chi \quad \mathcal{T}_1 \quad \mathcal{T}_2}{\phi \mathbin{\dot\vee} \chi \vdash \vee^-(h_0, \vee_r^+(\chi, h_0), \vee_l^+(\phi, h_0))\ [:]\ \chi \mathbin{\dot\vee} \phi}}{\vdash \to^+(\phi \mathbin{\dot\vee} \chi, \vee^-(h_0, \vee_r^+(\chi, h_0), \vee_l^+(\phi, h_0)))\ [:]\ \phi \mathbin{\dot\vee} \chi \mathbin{\dot\to} \chi \mathbin{\dot\vee} \phi}$$

where $\mathcal{T}_1$ denotes

$$\frac{\phi \mathbin{\dot\vee} \chi; \phi \vdash h_0 \,[:]\, \phi}{\phi \mathbin{\dot\vee} \chi; \phi \vdash \vee_r^+(\chi, h_0) \,[:]\, \chi \mathbin{\dot\vee} \phi}$$

and $\mathcal{T}_2$ is the analogous tree of $\phi \mathbin{\dot\vee} \chi; \chi \vdash \vee_l^+(\phi, h_0) \,[:]\, \chi \mathbin{\dot\vee} \phi$.

Some constructors also bind *term* variables. The constructor $\forall^+$ binds the first free variable in its argument; $\exists^-$ binds the first free variable in its second argument. These variables are called the *eigenvariables* of $\forall^+$ and $\exists^-$. The constructor $\exists^+$ binds the first free term variable in its first argument. We give a final example:

$$\dfrac{\dfrac{\dfrac{\dot\exists \uparrow_1\phi \mathbin{\dot\to} \uparrow\chi; \phi \vdash h_1 \,[:]\, \dot\exists \uparrow_1\phi \mathbin{\dot\to} \uparrow\chi \qquad \dfrac{\dot\exists \uparrow_1\phi \mathbin{\dot\to} \uparrow\chi; \phi \vdash h_0 \,[:]\, \phi}{\dot\exists \uparrow_1\phi \mathbin{\dot\to} \uparrow\chi; \phi \vdash \exists^+(\uparrow_1\phi, v_0, h_0) \,[:]\, \dot\exists \uparrow_1\phi}}{\dot\exists \uparrow_1\phi \mathbin{\dot\to} \uparrow\chi; \phi \vdash \to^-(h_1, \exists^+(\uparrow_1\phi, v_0, h_0)) \,[:]\, \uparrow\chi}}{\dot\exists \uparrow_1\phi \mathbin{\dot\to} \uparrow\chi \vdash \to^+(\phi, \to^-(h_1, \exists^+(\uparrow_1\phi, v_0, h_0))) \,[:]\, \phi \mathbin{\dot\to} \uparrow\chi}}{\dot\exists\, \phi \mathbin{\dot\to} \chi \vdash \forall^+(\to^+(\phi, \to^-(h_1, \exists^+(\uparrow_1\phi, v_0, h_0)))) \,[:]\, \dot\forall\,(\phi \mathbin{\dot\to} \uparrow\chi)}$$

The application of the $\exists^+$-rule is correct as $(\uparrow_1\phi)[v_0] = \phi$ by the following lemma. The application of $\forall^+$ is correct because $\uparrow(\dot\exists\,\phi \mathbin{\dot\to} \chi) = \dot\exists \uparrow_1\phi \mathbin{\dot\to} \uparrow\chi$, by definition of lifting. Note that, on the named level, for the formula $(\exists x.\phi \to \chi) \to \forall x.(\phi \to \chi)$ to be a tautology, the condition $x \notin \mathsf{FV}(\chi)$ is required to avoid capture by $\forall x$ in χ. This is expressed by $\uparrow\chi$ and can be compared to $\curlywedge x.\chi$ (see Chapter 3).

Lemma 2.6.1 *For $n : \mathbb{N}$, $t : \tau$ as well as for $t : o$, $(\uparrow_{n+1} t)[v_0]^n = t$.*

Note that, because intuitionistic predicate logic has the structural rules of weakening, exchange and contraction, the formulation of natural deduction above is logically equivalent to one that mentions (possibly) different contexts in rules with more than one premiss. Given the structural rules (shown to be derivable in the meta-theory in Section 2.7), for example, the following formulation of the rule for $\to^-$ is admissable, as can be shown by applying the weakening lemma (Lemma 2.7.2) $|\Gamma'|$ times. The assumption variables in d have to be lifted so that they still refer to the same assumptions in Γ as they originally did. Proof term e can be left unchanged, because there are no other assumption variables in e than those referring to Γ'.

$$\frac{\Gamma \vdash d \,[:]\, \phi \mathbin{\dot\to} \chi \qquad \Gamma' \vdash e \,[:]\, \phi}{\Gamma, \Gamma' \vdash \to^-(\uparrow_0^{|\Gamma'|} d, e) \,[:]\, \chi}$$

In Section 2.11 we show soundness of the deduction relation given in Definition 2.6.1, with respect to an interpetation function $[\![_]\!]$ mapping object-level formulas to Coq's native logic.

2.7 Admissible Rules

The following rules are *admissible*, that is, derivable in the meta-theory. In order to prove by induction, the statements are loaded appropriately (quantification over Γ, Δ, and so forth).

Lemma 2.7.1 (Lifting of judgement)

$$\frac{\Gamma \vdash d \;[:]\; \phi}{\uparrow_n \Gamma \vdash \uparrow^{\mathsf{v}}_n d \;[:]\; \uparrow_n \phi}$$

Proof. Induction on the proposition $\Gamma \vdash d \;[:]\; \phi$. The proofs of cases $\forall^+$ and $\exists^-$ require Lemma 2.5.1; cases $\forall^-$ and $\exists^+$ require Lemma 2.5.4.

Lemma 2.7.2 (Weakening)

$$\frac{\Gamma; \Delta \vdash d \;[:]\; \phi}{\Gamma; \chi; \Delta \vdash \uparrow^{\mathsf{h}}_{|\Delta|} d \;[:]\; \phi}$$

Proof. By induction on d and inverting the judgement.

Lemma 2.7.3 (Substitution of variables v_i in derivation terms)

$$\frac{\uparrow_n \Gamma; \Delta \vdash d \;[:]\; \phi}{\Gamma; \Delta[t]^n \vdash d[t]^n_{\mathsf{v}} \;[:]\; \phi[t]^n}$$

Proof. By induction on d and inversion. Case h_i is proved by induction over i and Lemma 2.5.2. Cases $\forall^+$ and $\exists^-$ require lemmas 2.5.3 and 2.5.1. Cases $\forall^-$ and $\exists^-$ require Lemma 2.5.5.

Lemma 2.7.4 (Substitution of variables h_i in derivation terms)

$$\frac{\Gamma \vdash d \;[:]\; \phi \quad \Gamma; \phi; \Delta \vdash e \;[:]\; \chi}{\Gamma; \Delta \vdash e[d]^{|\Delta|}_{\mathsf{h}} \;[:]\; \chi}$$

Proof. By induction on e and inversion. Case h_i is proved by induction over i and Lemma 2.7.2.

Exchange, contraction

The structural rules *exchange* and *contraction* are admissible, too.[11] First we need the functions exch and contr. The former swaps the indices n and $n+1$, while the latter decrements all indices greater than n, where n intends to be the reference depth of assumption variables ($n = |\Delta|$ in Lemmas 2.7.5 and 2.7.6).

$$\mathsf{exch}(n,i) = \begin{cases} h_{n+1} & \text{if } i = n \\ h_n & \text{if } i = n+1 \\ h_i & \text{otherwise} \end{cases} \qquad \mathsf{contr}(n,i) = \begin{cases} h_{i-1} & \text{if } i > n \\ h_i & \text{otherwise} \end{cases}$$

Again, the side conditions in the definitions above are avoided in the formalisation.

[11] These lemmas are not needed in the proof of Subject Reduction (Thm. 2.16.2).

Lemma 2.7.5 (Exchange)

$$\frac{\Gamma; \chi; \phi; \Delta \vdash d \;[:]\; \psi}{\Gamma; \phi; \chi; \Delta \vdash \mathsf{map}^{\mathsf{h}}_{\pi}(\mathsf{exch}, |\Delta|, d) \;[:]\; \psi}$$

Lemma 2.7.6 (Contraction)

$$\frac{\Gamma; \phi; \phi; \Delta \vdash d \;[:]\; \chi}{\Gamma; \phi; \Delta \vdash \mathsf{map}^{\mathsf{h}}_{\pi}(\mathsf{contr}, |\Delta|, d) \;[:]\; \chi}$$

2.8 Translation to Coq's Native Logic

We define the translation of object level statements (i.e., the objects defined in Definition 2.2.3) to meta-level statements (i.e., in the language of the framework itself). This translation will be referred to as *interpretation* and depends on a set A, the domain of discourse and parameters $\mathcal{V}$, $\mathcal{F}$, $\mathcal{R}$ for interpreting variables, function symbols and relation symbols respectively. As will be explained in Subsection 2.9.1, using $\mathbb{N}$ as an index set for variables, requires the domain to be non-empty; choose a_0 as the default value in A.

We introduce the operations of *shifting*, notation $\Uparrow_n \mathcal{V}$ and *inserting* terms $a : A$, notation $\mathcal{V}[a]^n$, in variable mappings, that is, λ-terms of type $\mathbb{N} \to A$.

Definition 2.8.1 *Given $\mathcal{V} : \mathbb{N} \to A$, $n : \mathbb{N}$, we define $\Uparrow_n \mathcal{V}$ as follows.*

$$\Uparrow_n \mathcal{V} = \lambda p{:}\mathbb{N}.\, \mathcal{V}(p+n)$$

Definition 2.8.2 *Given $\mathcal{V} : \mathbb{N} \to A$, $n : \mathbb{N}$ and $a : A$, $\mathcal{V}[a]^n$ is defined as follows.*

$$\begin{aligned} \mathcal{V}[a]^0(0) &= a \\ \mathcal{V}[a]^0(m+1) &= \mathcal{V}(m) \\ \mathcal{V}[a]^{n+1}(0) &= \mathcal{V}(0) \\ \mathcal{V}[a]^{n+1}(m+1) &= (\Uparrow_1 \mathcal{V})[a]^n(m) \end{aligned}$$

We write $\mathcal{V}[x]$ for $\mathcal{V}[x]^0$.

Term evaluation is defined as follows.

Definition 2.8.3 *Assume an arbitrary domain of discourse $A : *^s$ and a function $\mathcal{V} : \mathbb{N} \to A$ to interpret (free) variables. Declare a parameter $\mathcal{F}$, a family of functions indexed over $I_{l_{\mathsf{fun}}}$, used to interpret function symbols.*

$$\mathcal{F} : \Pi i{:}I_{l_{\mathsf{fun}}}.\, A^{l_{\mathsf{fun}}(i)} \to A$$

We write $\mathcal{F}_i$ for $(\mathcal{F}\ i)$. Given such a family, we define the evaluation function for terms of type τ.

$$\begin{aligned} [\![v_n]\!]^{\mathcal{V}} &= \mathcal{V}(n) \\ [\![f_i(t_1, \ldots, t_k)]\!]^{\mathcal{V}} &= \mathcal{F}_i([\![t_1]\!]^{\mathcal{V}}, \ldots, [\![t_k]\!]^{\mathcal{V}}) \end{aligned}$$

Next, we define the canonical interpretation of objects of type o.

Definition 2.8.4 *Again, let $A : *^s$ and $\mathcal{V} : \mathbb{N} \to A$. Assume a family of relations indexed over $I_{l_{\mathsf{rel}}}$.*

$$\mathcal{R} : \Pi j : I_{l_{\mathsf{rel}}} .\, A^{l_{\mathsf{rel}}(j)} \to *^p$$

We write $\mathcal{R}_i$ for $(\mathcal{R}\ i)$.

$$\begin{array}{rcl} [\![\dot{\top}]\!]^{\mathcal{V}} & = & \top \\ [\![\dot{\bot}]\!]^{\mathcal{V}} & = & \bot \\ [\![R_j(t_1,\ldots,t_m)]\!]^{\mathcal{V}} & = & \mathcal{R}_j([\![t_1]\!]^{\mathcal{V}},\ldots,[\![t_m]\!]^{\mathcal{V}}) \\ [\![\phi \mathbin{\dot{\wedge}} \chi]\!]^{\mathcal{V}} & = & [\![\phi]\!]^{\mathcal{V}} \wedge [\![\chi]\!]^{\mathcal{V}} \\ [\![\phi \mathbin{\dot{\vee}} \chi]\!]^{\mathcal{V}} & = & [\![\phi]\!]^{\mathcal{V}} \vee [\![\chi]\!]^{\mathcal{V}} \\ [\![\phi \mathbin{\dot{\to}} \chi]\!]^{\mathcal{V}} & = & [\![\phi]\!]^{\mathcal{V}} \to [\![\chi]\!]^{\mathcal{V}} \\ [\![\dot{\forall}\, \phi]\!]^{\mathcal{V}} & = & \Pi x : A.\, [\![\phi]\!]^{\mathcal{V}[x]} \\ [\![\dot{\exists}\, \phi]\!]^{\mathcal{V}} & = & \exists x : A.\, [\![\phi]\!]^{\mathcal{V}[x]} \end{array}$$

Initially (for closed formulas) we set $\mathcal{V}_0 = \lambda n : \mathbb{N}.\, a_0$, with a_0 the chosen default value in A, and define $[\![\phi]\!] = [\![\phi]\!]^{\mathcal{V}_0}$.

We use $\top, \bot, \wedge, \vee, \exists$ for Coq's predefined logical connectives. Note that '$\to$' (and 'Π') is used for both (dependent) function space as well as for logical implication (quantification); this overloading witnesses the Curry–Howard–De Bruijn isomorphism.

We don't have to worry about name conflicts when inserting a new $x : A$ to the variable interpretation function $\mathcal{V}$ (quantifier cases). Coq's binding mechanisms are internally based on De Bruijn indices (with a user-friendly tool showing named variables on top of it).

Definition 2.8.5 *The interpretation of a context is the conjunction of its interpreted elements.*

$$[\![\Box]\!]^{\mathcal{V}} = \top \qquad [\![\Gamma; \phi]\!]^{\mathcal{V}} = [\![\Gamma]\!]^{\mathcal{V}} \wedge [\![\phi]\!]^{\mathcal{V}}$$

Remark 2.8.1 *We stress the following analogies between the types of v, f, R, and the types of $\mathcal{V}$, $\mathcal{F}$, $\mathcal{R}$, respectively. First note that:*

$$\begin{array}{rccl} v_n & \textit{is syntactic sugar for} & & (v\ n) \\ f_i(t_1,\ldots,t_k) & '' & '' & (f\ i\ t_1\ \ldots\ t_k) \\ R_j(t_1,\ldots,t_m) & '' & '' & (R\ j\ t_1\ \ldots\ t_m) \end{array}$$

(Recall that $k = l_{\mathsf{fun}}(i)$ and $m = l_{\mathsf{rel}}(j)$.)

$$\begin{array}{rccl} v : \mathbb{N} \to \tau & \textit{analogous to} & & \mathcal{V} : \mathbb{N} \to A \\ f : \Pi i : I_{l_{\mathsf{fun}}} .\, \tau^k \to \tau & '' & '' & \mathcal{F} : \Pi i : I_{l_{\mathsf{fun}}} .\, A^k \to A \\ R : \Pi j : I_{l_{\mathsf{rel}}} .\, \tau^m \to o & '' & '' & \mathcal{R} : \Pi j : I_{l_{\mathsf{rel}}} .\, A^m \to *^p \end{array}$$

2.9 Free Variables

2.9.1 Free Variables, Finitely versus Infinitely Many

Note that, differently from type theory where variables have to be declared in the environment, in our representation we have infinitely many variables ($\mathbb{N}$ is the index set of variables). Therefore, we shall need a default value in order to have a total evaluation function (see Definition 2.8.3). Alternatively, we could have chosen to parameterise the sets of terms, formulas, and proof terms over a natural number n indicating the number of free variables an object is allowed to contain (enforced by definition via dependent types). Variables would then be indexed over $\mathbb{N}_n$, defined as follows (think of $\mathbb{N}_n$ as $\{0, \ldots, n-1\}$).

$$\mathbb{N}_0 = \emptyset \qquad \mathbb{N}_{n+1} = \mathbf{1} + \mathbb{N}_n$$

The set τ_n of first-order terms containing n free variables would then be defined as follows; let $m : \mathbb{N}_n$ and $t_1, \ldots, t_k : \tau_n$.

$$\tau_n := v_m \mid f_i(t_1, \ldots, t_k)$$

The constructors $\dot{\forall}$ and $\dot{\exists}$ of o_n then should be typed $o_{n+1} \to o_n$, as they bind the first free variable of their argument. The definition of lifting should be such that, given $t : \tau_n$, the application $\uparrow_m t$ is typed τ_{n+1} (a fresh variable v_m is introduced) and that $m \leq n$ is enforced. Given $k, m : \mathbb{N}$, $t : \tau_{k+m+1}$ and $t' : \tau_k$, $t[t']^m$ should be typed τ_{k+m}. Apparently, such an extra parameter means a considerable complication of matters and we chose to do without it. As a consequence, to be able to define a $\mathcal{V} : \mathbb{N} \to A$ for the evaluation of objects, one needs a default value in A.

2.9.2 Free Variables and Substitution

The De Bruijn representation works elegantly for bound variables, there is no renaming and the structural equality on De Bruijn terms corresponds to the intented identity of terms. As pointed out in [50], however, there is a slight inconvenience in the way free variables are treated. The point is that the order of free variables matters, not their names.

The subtle point about an expression $t[t']^n$ is that *the first n variables are assumed to be bound.* Let $\mathcal{V} : \mathbb{N} \to A$ be such that $\mathcal{V}(0) = y$ and $\mathcal{V}(1) = x$ (i.e., y is introduced later than x), then we can make a substitution that transforms, for example $\mathcal{R}_j(x, y)$ into $\mathcal{R}_j(x, x)$, as illustrated below. Note that, for any $\mathcal{V}'$, if t is interpreted under $\mathcal{V}'$, then $t[t']$ has to be interpreted under $\Uparrow_1 \mathcal{V}'$, because the original occurrences of v_0 in t that pointed to $\mathcal{V}'(0)$ have been removed, and the other variables have been decremented. We have $\Uparrow_1 \mathcal{V}(0) = \mathcal{V}(1) = x$ and

$$[\![R_j(v_1, v_0)]\!]^{\mathcal{V}} = \mathcal{R}_j(x, y)$$

$$[\![R_j(v_1, v_0)[v_0]]\!]^{\Uparrow_1 \mathcal{V}} = [\![R_i(v_0, v_0)]\!]^{\Uparrow_1 \mathcal{V}} = \mathcal{R}_j(x, x)$$

However, we *cannot* make a substitution that transforms $\mathcal{R}_j(x, y)$ into $\mathcal{R}_j(y, y)$. The reason for this is that x corresponds to v_1 and if you want to replace this, it is assumed that v_0 (pointing to y) is bound so that the variables in the substituent are lifted.

The substitution functions are meant for use only in combination with the removal of a binder; $\phi[t]$ is called to instantiate $\dot\forall\,\phi$ with t or to give t as a witness for $\dot\exists\,\phi$. Another possible (meta-level) binder is the variable mapping $\mathcal{V}$ as exemplified above. We maintain the term "substitution" *par abus de langage*.

2.10 Thinning and Substitution Lemmas

It is possible to insert free variables to the mapping $\mathcal{V}$ of the interpretation function given in definitions 2.8.3, 2.8.4 and, if the argument is appropriately lifted, keep the same interpretations. This is called *thinning* and can be compared to *weakening* (see Lemma 2.7.2); the latter is about assumption variables, the former about term variables. First we define some auxiliary lemmas.

Lemma 2.10.1 *For all $\mathcal{V}, \mathcal{V}' : \mathbb{N} \to A$, $x, y : A$, $n, m : \mathbb{N}$, $t : \tau$ and $\phi : o$, we have:*

$$\begin{array}{rcll} (\mathcal{V}[x]^n)[y](m) & = & (\mathcal{V}[y])[x]^{n+1}(m) & \textit{(permutation of insertion)} \\ \Uparrow_{n+1}(\mathcal{V}[x])(m) & = & \Uparrow_n \mathcal{V}(m) & \textit{(simplification of insertion)} \\ [\![\uparrow t]\!]^{\mathcal{V}} & = & [\![t]\!]^{\Uparrow_1 \mathcal{V}} & \textit{(lift-shift interchange)} \end{array}$$

Extensional equality of $\mathcal{V}$ and $\mathcal{V}'$, i.e. $\Pi n.\, \mathcal{V}(n) = \mathcal{V}'(n)$, implies $[\![t]\!]^{\mathcal{V}} = [\![t]\!]^{\mathcal{V}'}$ and $[\![\phi]\!]^{\mathcal{V}} \leftrightarrow [\![\phi]\!]^{\mathcal{V}'}$.

Lemma 2.10.2 (Thinning lemma) *Let $\mathcal{V} : \mathbb{N} \to A$, $a : A$ and $n : \mathbb{N}$. (Analogous to Lemma 2.7.2).*

$$\begin{array}{rcl} [\![t]\!]^{\mathcal{V}} & = & [\![\uparrow_n t]\!]^{\mathcal{V}[a]^n} \\ [\![\phi]\!]^{\mathcal{V}} & \leftrightarrow & [\![\uparrow_n \phi]\!]^{\mathcal{V}[a]^n} \end{array}$$

Similarly we need $[\![t[t']]\!]^{\mathcal{V}} = [\![t]\!]^{\mathcal{V}[[\![t']\!]^{\mathcal{V}}]}$. We need induction loading, no longer assuming that $[\![t']\!]^{\mathcal{V}}$ is the last added element.

Lemma 2.10.3 (Substitution lemma) *(Analogous to Lemma 2.7.4).*

$$\begin{array}{rcl} [\![t[t']^n]\!]^{\mathcal{V}} & = & [\![t]\!]^{\mathcal{V}[[\![t']\!]^{\Uparrow_n \mathcal{V}}]^n} \\ [\![\phi[t']^n]\!]^{\mathcal{V}} & \leftrightarrow & [\![\phi]\!]^{\mathcal{V}[[\![t']\!]^{\Uparrow_n \mathcal{V}}]^n} \end{array}$$

2.11 Soundness with respect to the Native Logic

We show that the deduction relation defined in Definition 2.6.1 is sound with respect to Coq's native logic.

Theorem 2.11.1 (Soundness) *For all contexts* Γ*, proof terms* d*, formulas* ϕ*, and variable mappings* $\mathcal{V}$ *we have that:*

$$(\Gamma \vdash d\ [:]\ \phi) \to [\![\Gamma]\!]^{\mathcal{V}} \to [\![\phi]\!]^{\mathcal{V}}$$

Proof. First the statement is loaded to $(\Gamma \vdash d\ [:]\ \phi) \to \Pi\mathcal{V}\!:\!\mathbb{N} \to A.\, [\![\Gamma]\!]^{\mathcal{V}} \to [\![\phi]\!]^{\mathcal{V}}$. Its proof proceeds by induction on the proposition $\Gamma \vdash d\ [:]\ \phi$. We sketch the proof for some representative cases.

(Case $\Gamma \vdash \to^{+}(\phi_1, d)\ [:]\ \phi_1 \dot{\to} \phi_2$) Assume $H_\Gamma : [\![\Gamma]\!]^{\mathcal{V}}$. We have the induction hypothesis $IH_d : [\![\Gamma;\phi_1]\!]^{\mathcal{V}} \to [\![\phi_2]\!]^{\mathcal{V}}$. Note that $[\![\Gamma;\phi_1]\!]^{\mathcal{V}} = [\![\Gamma]\!]^{\mathcal{V}} \wedge [\![\phi_1]\!]^{\mathcal{V}}$. It suffices to prove $[\![\phi_1]\!]^{\mathcal{V}} \to [\![\phi_2]\!]^{\mathcal{V}}$. Assume $H_{\phi_1} : [\![\phi_1]\!]^{\mathcal{V}}$, then IH_d applied to the pair $\langle H_\Gamma, H_{\phi_1}\rangle$, is a proof of $[\![\phi_2]\!]^{\mathcal{V}}$.

(Case $\Gamma \vdash \vee^{-}(d, e_1, e_2)\ [:]\ \phi$) Assume $H_\Gamma : [\![\Gamma]\!]^{\mathcal{V}}$. The proof obligation is $[\![\phi]\!]^{\mathcal{V}}$. Three induction hypotheses, corresponding to the three premisses of the $\vee^{-}$-rule are $IH_d : [\![\Gamma]\!]^{\mathcal{V}} \to [\![\chi_1 \dot{\vee} \chi_2]\!]^{\mathcal{V}}$ and $IH_{e_i} : [\![\Gamma;\chi_i]\!]^{\mathcal{V}} \to [\![\phi]\!]^{\mathcal{V}}$ $(i = 1,2)$. We get $[\![\chi_1]\!]^{\mathcal{V}} \vee [\![\chi_2]\!]^{\mathcal{V}}$ from IH_d and H_Γ.

- Suppose $H_{\chi_1} : [\![\chi_1]\!]^{\mathcal{V}}$, then $(IH_{e_1}\ \langle H_\Gamma, H_{\chi_1}\rangle) : [\![\phi]\!]^{\mathcal{V}}$.
- Suppose $H_{\chi_2} : [\![\chi_2]\!]^{\mathcal{V}}$, then $(IH_{e_2}\ \langle H_\Gamma, H_{\chi_2}\rangle) : [\![\phi]\!]^{\mathcal{V}}$.

(Case $\Gamma \vdash \forall^{+}(d)\ [:]\ \dot{\forall}\,\phi$) Let $H_\Gamma : [\![\Gamma]\!]^{\mathcal{V}}$. The induction hypothesis is $IH_d : \Pi\mathcal{V} : \mathbb{N} \to A.\, [\![\uparrow\!\Gamma]\!]^{\mathcal{V}} \to [\![\phi]\!]^{\mathcal{V}}$. We have to prove $\Pi x : A.\, [\![\phi]\!]^{\mathcal{V}[x]}$. Assume an arbitrary $x : A$. From Lemma 2.10.2 and H_Γ, it follows that $[\![\uparrow\!\Gamma]\!]^{\mathcal{V}[x]}$. Then, IH_d for $\mathcal{V}[x]$ and the proof of $[\![\uparrow\!\Gamma]\!]^{\mathcal{V}[x]}$, proves $[\![\phi]\!]^{\mathcal{V}[x]}$.

(Case $\Gamma \vdash \exists^{+}(\phi, t, d)\ [:]\ \dot{\exists}\,\phi$) Let $H_\Gamma : [\![\Gamma]\!]^{\mathcal{V}}$. We have $IH_d : [\![\Gamma]\!]^{\mathcal{V}} \to [\![\phi[t]]\!]^{\mathcal{V}}$. The proof obligation is $\exists x\!:\!A.\, [\![\phi]\!]^{\mathcal{V}[x]}$. Give $[\![t]\!]^{\mathcal{V}}$ as witness for this existential statement, so that our goal becomes $[\![\phi]\!]^{\mathcal{V}[[\![t]\!]^{\mathcal{V}}]}$, which, by Lemma 2.10.3, is implied by $[\![\phi[t]]\!]^{\mathcal{V}}$, which in turn follows directly from IH_d and H_Γ.

The following remark explains this chapter's title.

Remark 2.11.1 *As with all lemmas and theorems in this thesis, the proof of Theorem 2.11.1 is a formalised and verified* λ*-term in* Coq*:*

$$\mathsf{sound} : \Pi\Gamma, d, \phi.(\Gamma \vdash d\ [:]\ \phi) \to [\![\Gamma]\!] \to [\![\phi]\!]$$

Given H_d *of type* $\Gamma \vdash d\ [:]\ \phi$ *and* H_Γ *of type* $[\![\Gamma]\!]$ *for some context* Γ*, proof term* d*, and formula* ϕ*, define:*

$$M = (\mathsf{sound}\ \Gamma\ d\ \phi\ H_d\ H_\Gamma)$$

We say that d reflects *the* proof M *of the first-order proposition* $[\![\phi]\!]$*:*

$$(H_d\!:\!(\Gamma \vdash d\ [:]\ \phi); H_\Gamma\!:\![\![\Gamma]\!]) \vdash_{\mathsf{cic}} M : [\![\phi]\!]$$

where we use $\vdash_{\mathsf{cic}}$ *to denote derivability in the calculus of inductive constructions.*

For *correct* derivation terms d, the λ-term H_d of type $\Gamma \vdash d\ [:]\ \phi$ can be generated from d, as will be shown in the next two subsections.

2.12 Type Checking Function

Given a context Γ and a proof term d, it is possible to determine whether d reflects a correct proof and, if it does, to synthesise the type of d. First we define so-called options.

Definition 2.12.1 *The set* opt *of* options *is defined inductively as follows. Let* $\phi : o$*.*

$$\mathsf{opt} := \mathsf{val}(\phi) \mid \mathsf{err}$$

Definition 2.12.2 *We define the* type checking function $\mathsf{chk}(\Gamma, d) : \mathsf{opt}$ *by recursion on* d*.*

$$\begin{array}{rcll}
\mathsf{chk}(\Gamma, \top^+) & = & \mathsf{val}(\dot{\top}) & \\
\mathsf{chk}(\Gamma, h_i) & = & \mathsf{val}(\Gamma(i)) & \textit{if } i < |\Gamma| \\
\mathsf{chk}(\Gamma, \bot^-(d,\phi)) & = & \mathsf{val}(\phi) & \textit{if } \mathsf{chk}(\Gamma, d) = \mathsf{val}(\dot{\bot}) \\
\mathsf{chk}(\Gamma, \to^+(\phi, d)) & = & \mathsf{val}(\phi \dot{\to} \chi) & \textit{if } \mathsf{chk}([\Gamma;\phi], d) = \mathsf{val}(\chi) \\
\mathsf{chk}(\Gamma, \to^-(d, e)) & = & \mathsf{val}(\chi) & \textit{if } \left\{ \begin{array}{l} \mathsf{chk}(\Gamma, d) = \mathsf{val}(\phi \dot{\to} \chi) \\ \mathsf{chk}(\Gamma, e) = \mathsf{val}(\phi') \\ \phi = \phi' \end{array} \right. \\
\mathsf{chk}(\Gamma, \vee^-(d, e_1, e_2)) & = & \mathsf{val}(\phi) & \textit{if } \left\{ \begin{array}{l} \mathsf{chk}(\Gamma, d) = \mathsf{val}(\psi_1 \dot{\vee} \psi_2) \\ \mathsf{chk}([\Gamma;\psi_1], e_1) = \mathsf{val}(\phi) \\ \mathsf{chk}([\Gamma;\psi_2], e_2) = \mathsf{val}(\phi') \\ \phi' = \phi \end{array} \right. \\
\mathsf{chk}(\Gamma, \forall^+(d)) & = & \mathsf{val}(\dot{\forall}\,\phi) & \textit{if } \mathsf{chk}(\uparrow\Gamma, d) = \mathsf{val}(\phi) \\
\mathsf{chk}(\Gamma, \forall^-(t, d)) & = & \mathsf{val}(\phi[t]) & \textit{if } \mathsf{chk}(\Gamma, d) = \mathsf{val}(\dot{\forall}\,\phi) \\
\mathsf{chk}(\Gamma, \exists^+(\phi, t, d)) & = & \mathsf{val}(\dot{\exists}\,\phi) & \textit{if } \left\{ \begin{array}{l} \mathsf{chk}(\Gamma, d) = \mathsf{val}(\phi') \\ \phi' = \phi[t] \end{array} \right. \\
\mathsf{chk}(\Gamma, \exists^-(d, e)) & = & \mathsf{val}(\downarrow\phi) & \textit{if } \left\{ \begin{array}{l} \mathsf{chk}(\Gamma, d) = \mathsf{val}(\dot{\exists}\,\chi) \\ \mathsf{chk}([\uparrow\Gamma;\chi], e) = \mathsf{val}(\phi) \\ v_0 \notin \mathsf{FV}(\phi) \end{array} \right.
\end{array}$$

For any recursive call on a subterm, it is checked whether it gives a value or an error. Thus, unlike other programming languages, errors have to be propagated recursively. The proviso's are defined by case analysis on the recursive calls on substructures and by using a Boolean equality relation on formulas. If these conditions are not satisfied, err is returned. The canonical cases for the constructors $\wedge^+$, $\wedge^-_l$, $\wedge^-_r$, $\vee^+_l$, $\vee^+_r$ are left out.

2.13 Correctness of Type Checking Function

Type checking is sound and complete with respect to the deduction system of Definition 2.6.1.

Theorem 2.13.1 (Correctness of chk) *For all proof terms* d*, contexts* Γ *and formulas* ϕ*, we have that:*

$$\mathsf{chk}(\Gamma, d) = \mathsf{val}(\phi) \leftrightarrow \Gamma \vdash d\ [:]\ \phi$$

Proof. ($\rightarrow$) By induction on d.

(Case $\mathsf{chk}(\Gamma, \exists^-(d,e)) = \mathsf{val}(\downarrow\phi)$) From this it follows that $\mathsf{chk}(\Gamma, d) = \mathsf{val}(\dot{\exists}\,\chi)$ and $\mathsf{chk}([\uparrow\Gamma;\chi], e) = \mathsf{val}(\phi)$. By the induction hypotheses, we obtain $\Gamma \vdash d\ [:]\ \dot{\exists}\,\chi$ and $\uparrow\Gamma;\chi \vdash e\ [:]\ \phi$. Because e is a correct term, $v_0 \notin \mathsf{FV}(\phi)$; by Lemma 2.4.2 we get $\phi = \uparrow\downarrow\phi$. The proof obligation, then, is fulfilled by application of the inference rule for $\exists^-$.

$$\frac{\Gamma \vdash d\ [:]\ \dot{\exists}\chi \quad \uparrow\Gamma;\chi \vdash e\ [:]\ \uparrow\downarrow\phi}{\Gamma \vdash \exists^-(d,e)\ [:]\ \downarrow\phi}$$

($\leftarrow$) By induction on $\Gamma \vdash d\ [:]\ \phi$.

(Case $\Gamma \vdash \exists^-(d,e)\ [:]\ \phi$) We have $\Gamma \vdash d\ [:]\ \dot{\exists}\chi$ and $\uparrow\Gamma;\chi \vdash e\ [:]\ \uparrow\phi$. By the induction hypotheses, we obtain $\mathsf{chk}(\Gamma, d) = \mathsf{val}(\dot{\exists}\,\chi)$ and $\mathsf{chk}([\uparrow\Gamma;\chi], e) = \mathsf{val}(\uparrow\phi)$. We have $v_0 \notin \mathsf{FV}(\uparrow\phi)$ (Lemma 2.4.3) and so $\mathsf{chk}(\Gamma, \exists^-(d,e)) = \mathsf{val}(\downarrow\uparrow\phi)$; finally, $\downarrow\uparrow\phi = \phi$ by Lemma 2.4.1.

2.14 Unique Types

Proof terms have unique types.

Corollary 2.14.1 (Uniqueness of Types) *For all proof terms d, contexts Γ and formulas ϕ and χ, we have that:*

$$(\Gamma \vdash d\ [:]\ \phi) \rightarrow (\Gamma \vdash d\ [:]\ \chi) \rightarrow \phi = \chi$$

Proof. Direct from double application of Theorem 2.13.1.

2.15 Proof Reduction

To illustrate how the defined machinery can be used to manipulate proof objects, we define Prawitz's proof reduction rules [61].[12] The goal is to remove detours, as in the following tree.

$$\frac{\dfrac{\Gamma;\phi \vdash d\ [:]\ \chi}{\Gamma \vdash \rightarrow^+(\phi,d)\ [:]\ \phi \dot{\rightarrow} \chi} \quad \Gamma \vdash e\ [:]\ \phi}{\Gamma \vdash \rightarrow^-(\rightarrow^+(\phi,d),e)\ [:]\ \chi}$$

Instead of first assuming ϕ to build a proof d of χ, introduce the implication $\phi \dot{\rightarrow} \chi$, and then eliminate it immediately by plugging in derivation e, we can more directly replace the assumption ϕ in d (represented by the first free assumption variable) by e.

$$\frac{}{\Gamma \vdash d[e]_{\mathsf{h}}\ [:]\ \chi}$$

[12] We actually follow [59], pages 85–88.

The removal of such a direct detour is called a *proper reduction*. There are seven such rewrite rules, where on the left-hand side an introduction of a certain connective is immediately followed by an elimination of that connective. Sometimes, proper redexes are *hidden* by intermediate $\vee^-$ and/or $\exists^-$ rules. Such hidden detours are made direct by a sequence of so-called *permutative conversions*. These conversions pull out the $\vee^-$ and $\exists^-$ rules. After the following definition, we give an example of such a permutative conversion. The proof of Theorem 2.16.1 demonstrates why the various lifting operations are necessary to keep correct proofs.

Definition 2.15.1 Immediate proof reduction, *$d \mapsto e$, is defined by the following rewrite rules. The left-hand sides are called immediate (proper, permutative) redexes and the right-hand sides immediate (proper, permutative) reducts.*
Proper reductions.

$$\begin{array}{rcll}
\to^-(\to^+(\phi,d),e) & \mapsto & d[e]_{\mathsf{h}} & (\mathrm{PR}{\to}) \\
\wedge_l^-(\wedge^+(d_1,d_2)) & \mapsto & d_1 & (\mathrm{PR}{\wedge_1}) \\
\wedge_r^-(\wedge^+(d_1,d_2)) & \mapsto & d_2 & (\mathrm{PR}{\wedge_2}) \\
\vee^-(\vee_l^+(\phi,d),e_1,e_2) & \mapsto & e_1[d]_{\mathsf{h}} & (\mathrm{PR}{\vee_1}) \\
\vee^-(\vee_r^+(\phi,d),e_1,e_2) & \mapsto & e_2[d]_{\mathsf{h}} & (\mathrm{PR}{\vee_2}) \\
\forall^-(t,\forall^+(d)) & \mapsto & d[t]_{\mathsf{v}} & (\mathrm{PR}\forall) \\
\exists^-(\exists^+(\phi,t,d),e) & \mapsto & (e[t]_{\mathsf{v}})[d]_{\mathsf{h}} & (\mathrm{PR}\exists)
\end{array}$$

Permutative conversions.

$$\begin{array}{rcll}
\bot^-(\vee^-(d,e_1,e_2),\phi) & \mapsto & \vee^-(d,\bot^-(e_1,\phi),\bot^-(e_2,\phi)) & (\mathrm{PC}\vee\bot) \\
\to^-(\vee^-(d,e_1,e_2),g) & \mapsto & \vee^-(d,\to^-(e_1,\uparrow^{\mathsf{h}}g),\to^-(e_2,\uparrow^{\mathsf{h}}g)) & (\mathrm{PC}\vee{\to}) \\
\wedge_l^-(\vee^-(d,e_1,e_2)) & \mapsto & \vee^-(d,\wedge_l^-(e_1),\wedge_l^-(e_2)) & (\mathrm{PC}\vee\wedge_1) \\
\wedge_r^-(\vee^-(d,e_1,e_2)) & \mapsto & \vee^-(d,\wedge_r^-(e_1),\wedge_r^-(e_2)) & (\mathrm{PC}\vee\wedge_2) \\
\vee^-(\vee^-(d,e_1,e_2),g,h) & \mapsto & \vee^-(d,\vee^-(e_1,\uparrow_1^{\mathsf{h}}g,\uparrow_1^{\mathsf{h}}h), & \\
 & & \quad \vee^-(e_2,\uparrow_1^{\mathsf{h}}g,\uparrow_1^{\mathsf{h}}h)) & (\mathrm{PC}\vee\vee) \\
\forall^-(t,\vee^-(d,e_1,e_2)) & \mapsto & \vee^-(d,\forall^-(t,e_1),\forall^-(t,e_2)) & (\mathrm{PC}\vee\forall) \\
\exists^-(\vee^-(d,e_1,e_2),g) & \mapsto & \vee^-(d,\exists^-(e_1,\uparrow_1^{\mathsf{h}}g),\exists^-(e_2,\uparrow_1^{\mathsf{h}}g)) & (\mathrm{PC}\vee\exists) \\
\bot^-(\exists^-(d,e),\phi) & \mapsto & \exists^-(d,\bot^-(e,\uparrow\phi)) & (\mathrm{PC}\exists\bot) \\
\to^-(\exists^-(d,e),f) & \mapsto & \exists^-(d,\to^-(e,\uparrow^{\mathsf{h}}(\uparrow^{\mathsf{v}}f))) & (\mathrm{PC}\exists{\to}) \\
\wedge_l^-(\exists^-(d,e)) & \mapsto & \exists^-(d,\wedge_l^-(e)) & (\mathrm{PC}\exists\wedge_1) \\
\wedge_r^-(\exists^-(d,e)) & \mapsto & \exists^-(d,\wedge_r^-(e)) & (\mathrm{PC}\exists\wedge_2) \\
\vee^-(\exists^-(d,e),f,g) & \mapsto & \exists^-(d,\vee^-(e,\uparrow_1^{\mathsf{h}}(\uparrow^{\mathsf{v}}f),\uparrow_1^{\mathsf{h}}(\uparrow^{\mathsf{v}}g))) & (\mathrm{PC}\exists\vee) \\
\forall^-(t,\exists^-(d,e)) & \mapsto & \exists^-(d,\forall^-(\uparrow t,e)) & (\mathrm{PC}\exists\forall) \\
\exists^-(\exists^-(d,e),f) & \mapsto & \exists^-(d,\exists^-(e,\uparrow_1^{\mathsf{h}}(\uparrow_1^{\mathsf{v}}f))) & (\mathrm{PC}\exists\exists)
\end{array}$$

Definition 2.15.2 *We define $\rhd$ as the closure of $\mapsto$ under the construction rules of π. In other words, $d \rhd d'$ holds if d' can be obtained from d by replacing a subterm of d by an immediate reduct of it.*

As an example, consider the following reduction sequence, consisting of rules PC∨∃ and PR∃ respectively.

$$\exists^-(\vee^-(d,\exists^+(\phi,t,e_1),e_2),g)$$

$$\triangleright \quad \vee^-(d, \exists^-(\exists^+(\phi, t, e_1), \uparrow_1^{\mathsf{h}} g), \exists^-(e_2, \uparrow_1^{\mathsf{h}} g))$$
$$\triangleright \quad \vee^-(d, ((\uparrow_1^{\mathsf{h}} g)[t]_{\mathsf{v}})[e_1]_{\mathsf{h}}, \exists^-(e_2, \uparrow_1^{\mathsf{h}} g))$$

Let's depict the corresponding proof trees, starting with the permutative redex.

$$\dfrac{\dfrac{\Gamma \vdash d \ [:] \ \psi_1 \ \dot\vee \ \psi_2 \quad \dfrac{\Gamma; \psi_1 \vdash e_1 \ [:] \ \phi[t]}{\Gamma; \psi_1 \vdash \exists^+(\phi, t, e_1) \ [:] \ \dot\exists \phi} \quad \Gamma; \psi_2 \vdash e_2 \ [:] \ \dot\exists \phi}{\Gamma \vdash \vee^-(d, \exists^+(\phi, t, e_1), e_2) \ [:] \ \dot\exists \phi} \quad \uparrow\Gamma; \phi \vdash g \ [:] \uparrow\chi}{\Gamma \vdash \exists^-(\vee^-(d, \exists^+(\phi, t, e_1), e_2), g) \ [:] \ \chi}$$

The previously hidden detour is made direct, as shown in the following tree, corresponding to the permutative reduct.

$$\dfrac{\Gamma \vdash d \ [:] \ \psi_1 \ \dot\vee \ \psi_2 \quad \mathcal{T}_1 \quad \mathcal{T}_2}{\Gamma \vdash \vee^-(d, \exists^-(\exists^+(\phi, t, e_1), \uparrow_1^{\mathsf{h}} g), \exists^-(e_2, \uparrow_1^{\mathsf{h}} g)) \ [:] \ \chi}$$

Where $\mathcal{T}_1$ denotes

$$\dfrac{\Gamma; \psi_1 \vdash \exists^+(\phi, t, e_1) \ [:] \ \dot\exists \phi \quad \uparrow(\Gamma; \psi_1); \phi \vdash \uparrow_1^{\mathsf{h}} g \ [:] \uparrow\chi}{\Gamma; \psi_1 \vdash \exists^-(\exists^+(\phi, t, e_1), \uparrow_1^{\mathsf{h}} g) \ [:] \ \chi}$$

and $\mathcal{T}_2$ denotes

$$\dfrac{\Gamma; \psi_2 \vdash e_2 \ [:] \ \dot\exists \phi \quad \uparrow(\Gamma; \psi_2); \phi \vdash \uparrow_1^{\mathsf{h}} g \ [:] \uparrow\chi}{\Gamma; \psi_2 \vdash \exists^-(e_2, \uparrow_1^{\mathsf{h}} g) \ [:] \ \chi}$$

Now $\mathcal{T}_1$ contains a direct detour, which reduces to $\mathcal{T}_1'$:

$$\dfrac{}{\Gamma; \psi_1 \vdash ((\uparrow_1^{\mathsf{h}} g)[t]_{\mathsf{v}})[e_1]_{\mathsf{h}} \ [:] \ \chi}$$

The proof tree corresponding to the final term in the reduction sequence then reads:

$$\dfrac{\Gamma \vdash d \ [:] \ \psi_1 \ \dot\vee \ \psi_2 \quad \mathcal{T}_1' \quad \mathcal{T}_2}{\Gamma \vdash \vee^-(d, ((\uparrow_1^{\mathsf{h}} g)[t]_{\mathsf{v}})[e_1]_{\mathsf{h}}, \exists^-(e_2, \uparrow_1^{\mathsf{h}} g)) \ [:] \ \chi}$$

2.16 Subject Reduction

Theorem 2.16.1 (Subject Reduction ($\mapsto$))

$$(d \mapsto e) \to (\Gamma \vdash d \ [:] \ \phi) \to (\Gamma \vdash e \ [:] \ \phi)$$

Proof. By induction on the proposition $d \mapsto e$. The so obtained instances of $\Gamma \vdash d \ [:] \ \phi$ are inverted twice. We show some representative cases.

(PR→) The following tree is built bottom-up with the use of inversion, starting at the given judgement $\Gamma \vdash \to^-(\to^+(\phi, d), e)$ [:] χ in the root. Inverting the root gives $\Gamma \vdash \to^+(\phi, d)$ [:] $\phi \dot\to \chi$ and $\Gamma \vdash e$ [:] ϕ. Inverting the former judgement gives $\Gamma; \phi \vdash d$ [:] χ.

$$\dfrac{\dfrac{\Gamma; \phi \vdash d \text{ [:] } \chi}{\Gamma \vdash \to^+(\phi, d) \text{ [:] } \phi \dot\to \chi} \quad \Gamma \vdash e \text{ [:] } \phi}{\Gamma \vdash \to^-(\to^+(\phi, d), e) \text{ [:] } \chi}$$

We have to prove: $\Gamma \vdash d[e]_{\mathsf{h}}$ [:] χ, which follows from Lemma 2.7.4 by substituting the empty context for Δ:

$$\dfrac{\Gamma \vdash e \text{ [:] } \phi \quad \Gamma; \phi \vdash d \text{ [:] } \chi}{\Gamma \vdash d[e]_{\mathsf{h}} \text{ [:] } \chi}$$

(PR∀) Assume $\Gamma \vdash \forall^-(t, \forall^+(d))$ [:] $\phi[t]$. Using inversion, we build the following tree.

$$\dfrac{\dfrac{\uparrow\Gamma \vdash d \text{ [:] } \phi}{\Gamma \vdash \forall^+(d) \text{ [:] } \dot\forall \phi}}{\Gamma \vdash \forall^-(t, \forall^+(d)) \text{ [:] } \phi[t]}$$

We have to prove: $\Gamma \vdash d[t]_{\mathsf{v}}$ [:] $\phi[t]$, which follows from Lemma 2.7.3 and $\uparrow\Gamma \vdash d$ [:] ϕ (take Δ empty and $n = 0$).

(PC∨∨) Assume $\Gamma \vdash \vee^-(\vee^-(d, e_1, e_2), g, h)$ [:] ϕ. The proof obligation is:

$$\Gamma \vdash \vee^-(d, \vee^-(e_1, \uparrow^{\mathsf{h}}_1 g, \uparrow^{\mathsf{h}}_1 h), \vee^-(e_2, \uparrow^{\mathsf{h}}_1 g, \uparrow^{\mathsf{h}}_1 h)) \text{ [:] } \phi$$

We use the following abbreviations.

$$\begin{array}{lcl} \mathcal{J}_d & \equiv & \Gamma \vdash d \text{ [:] } \psi_1 \dot\vee \psi_2 \\ \mathcal{J}_g & \equiv & \Gamma; \chi_1 \vdash g \text{ [:] } \phi \\ \mathcal{J}_h & \equiv & \Gamma; \chi_2 \vdash h \text{ [:] } \phi \end{array} \qquad \begin{array}{lcl} \mathcal{J}_{e_1} & \equiv & \Gamma; \psi_1 \vdash e_1 \text{ [:] } \chi_1 \dot\vee \chi_2 \\ \mathcal{J}_{e_2} & \equiv & \Gamma; \psi_2 \vdash e_2 \text{ [:] } \chi_1 \dot\vee \chi_2 \end{array}$$

After inversion, we come to the following tree.

$$\dfrac{\dfrac{\mathcal{J}_d \quad \mathcal{J}_{e_1} \quad \mathcal{J}_{e_2}}{\Gamma \vdash \vee^-(d, e_1, e_2) \text{ [:] } \chi_1 \dot\vee \chi_2} \quad \mathcal{J}_g \quad \mathcal{J}_h}{\Gamma \vdash \vee^-(\vee^-(d, e_1, e_2), g, h) \text{ [:] } \phi}$$

The following tree demonstrates how the goal is deduced.

$$\dfrac{\mathcal{J}_d \quad \mathcal{T}_1 \quad \mathcal{T}_2}{\Gamma \vdash \vee^-(d, \vee^-(e_1, \uparrow^{\mathsf{h}}_1 g, \uparrow^{\mathsf{h}}_1 h), \vee^-(e_2, \uparrow^{\mathsf{h}}_1 g, \uparrow^{\mathsf{h}}_1 h)) \text{ [:] } \phi}$$

where $\mathcal{T}_1$ denotes the subtree:

$$\dfrac{\mathcal{J}_{e_1} \quad \Gamma; \psi_1; \chi_1 \vdash \uparrow^{\mathsf{h}}_1 g \text{ [:] } \phi \quad \Gamma; \psi_1; \chi_2 \vdash \uparrow^{\mathsf{h}}_1 h \text{ [:] } \phi}{\Gamma; \psi_1 \vdash \vee^-(e_1, \uparrow^{\mathsf{h}}_1 g, \uparrow^{\mathsf{h}}_1 h) \text{ [:] } \phi}$$

and $\mathcal{T}_2$ the analogous deduction of $\Gamma;\psi_2 \vdash \vee^-(e_2, \uparrow^{\mathsf{h}}_1 g, \uparrow^{\mathsf{h}}_1 h)$ [:] ϕ. Now it becomes clear why all free assumption variables except the first have to be lifted in, for example, proof term g: $\uparrow^{\mathsf{h}}_1 g$. In $\mathcal{T}_1$ the extra assumption ψ_1 is added to the context. The leafs $\Gamma;\psi_1;\chi_1 \vdash \uparrow^{\mathsf{h}}_1 g$ [:] ϕ and $\Gamma;\psi_1;\chi_2 \vdash \uparrow^{\mathsf{h}}_1 h$ [:] ϕ in $\mathcal{T}_1$ are implied by the judgements $\mathcal{J}_g$ and $\mathcal{J}_h$ respectively, via the weakening lemma (2.7.2).

(PC$\exists\rightarrow$) Assume $\Gamma \vdash \rightarrow^-(\exists^-(d,e), f)$ [:] χ.

$$\dfrac{\dfrac{\Gamma \vdash d \text{ [:] } \dot{\exists}\psi \quad \uparrow\Gamma;\psi \vdash e \text{ [:] } \uparrow(\phi \dot{\rightarrow} \chi)}{\Gamma \vdash \exists^-(d,e) \text{ [:] } \phi \dot{\rightarrow} \chi} \quad \Gamma \vdash f \text{ [:] } \phi}{\Gamma \vdash \rightarrow^-(\exists^-(d,e), f) \text{ [:] } \chi}$$

The conversion PC$\exists\rightarrow$ puts derivation f in the scope of the $\exists^-$. In the new situation, in order to obtain a correct deduction, f is lifted such that it no longers contains v_0 and h_0 (now referring to the new assumption ψ). We have to prove $\Gamma \vdash \exists^-(d, \rightarrow^-(e, \uparrow^{\mathsf{h}}(\uparrow^{\mathsf{v}} f)))$ [:] χ.

$$\dfrac{\Gamma \vdash d \text{ [:] } \dot{\exists}\psi \quad \dfrac{\uparrow\Gamma;\psi \vdash e \text{ [:] } \uparrow\phi \dot{\rightarrow} \uparrow\chi \quad \uparrow\Gamma;\psi \vdash \uparrow^{\mathsf{h}}(\uparrow^{\mathsf{v}} f) \text{ [:] } \uparrow\phi}{\uparrow\Gamma;\psi \vdash \rightarrow^-(e, \uparrow^{\mathsf{h}}(\uparrow^{\mathsf{v}} f)) \text{ [:] } \uparrow\chi}}{\Gamma \vdash \exists^-(d, \rightarrow^-(e, \uparrow^{\mathsf{h}}(\uparrow^{\mathsf{v}} f))) \text{ [:] } \chi}$$

Note that $\uparrow(\phi \dot{\rightarrow} \chi) = \uparrow\phi \dot{\rightarrow} \uparrow\chi$. Thus, all we have to show is that $\uparrow\Gamma;\psi \vdash \uparrow^{\mathsf{h}}(\uparrow^{\mathsf{v}} f)$ [:] $\uparrow\phi$ follows from $\Gamma \vdash f$ [:] ϕ. By Lemma 2.7.1, we have that $\uparrow\Gamma \vdash \uparrow^{\mathsf{v}} f$ [:] $\uparrow\phi$. Then our goal follows from the weakening lemma (2.7.2).

The following theorem, stating that $\triangleright$ preserves types, follows directly from Theorem 2.16.1. The proof proceeds by structural induction on the proposition $d \triangleright e$.

Theorem 2.16.2 (Subject Reduction ($\triangleright$))

$$(d \triangleright e) \rightarrow (\Gamma \vdash d \text{ [:] } \phi) \rightarrow (\Gamma \vdash e \text{ [:] } \phi)$$

2.17 Conclusion and Future Research

We described a formalisation of natural deduction for intuitionistic first-order logic in Coq. This formalisation provides an object language amenable to the manipulation of formulas and of proof objects, which is the objective of this study. In the meta-theory we are able to reason about these syntactical objects. The example of a proof reduction relation demonstrates how proof terms can be subject to manipulation and to reasoning. Via the soundness (Theorem 2.11.1) of the deduction system of hypothetical judgements (Definition 2.6.1), we are also able to lift object level proof terms to actual proof terms inhabiting propositions of type $*^p$. Thus we can reflect upon the first-order fragment of $*^p$.

We plan to use the described formalisation for a syntactical proof of conservativity of the Axiom of Choice over first-order intuitionistic logic without equality (see [63] and [30]). Also, proving termination of permutative conversions (along the lines of [42] or [59]) is challenging.

Acknowledgments

The author thanks Marc Bezem, Vincent van Oostrom, Jaco van de Pol and Freek Wiedijk for their critical remarks on draft versions of this chapter and for many fruitful discussions.

Chapter 3

$\text{\reflectbox{$\lambda$}}$

We make the notion of scope in the λ-calculus explicit. To that end, the syntax of the λ-calculus is extended with an end-of-scope operator $\text{\reflectbox{$\lambda$}}$, matching the usual opening of a scope due to λ. Accordingly, β-reduction is extended to the set of scoped λ-terms by performing *minimal* scope extrusion before performing replication as usual. We show confluence of the resulting scoped β-reduction. Confluence of β-reduction for the ordinary λ-calculus is obtained as a corollary, by extruding scopes *maximally* before forgetting them altogether. Only in this final forgetful step, α-equivalence is needed. All our proofs have been verified in Coq.

Authors: Dimitri Hendriks and Vincent van Oostrom

3.1 Introduction

Performing a substitution $M[x{:=}N]$ in the λ-calculus can be decomposed into two subtasks: replicating N an appropriate number of times, and renaming in M in order to prevent unintended capture of variables of N. Indeed, the defining clauses of Curry's definition of substitution, see e.g. C.1 DEFINITION of [4], can be neatly partitioned into those dealing with replication (the variable and application clauses) and those dealing with renaming (the abstraction clauses). In this chapter we will focus on trying to understand the latter subtask. We do so, by extending λ-calculus with an explicit operator representing the (end of the) scope of a name, while leaving replication implicit.

Abstractions in the λ-calculus can be viewed as being composed of two parts: one part which is dual to application, and another which causes the opening of the scope of the bound variable. The scope of the binder λx in $\lambda x.M$ is (implicitly) assumed to extend to the whole of M. Hence to make the notion of scope explicit, it suffices to introduce an operator expressing the end of the scope of λx. This operator is denoted by $\text{\reflectbox{$\lambda$}}$ (adbmal). $\text{\reflectbox{$\lambda$}}x.M$ expresses that the scope of x is ended 'above' M. For instance, in the $\text{\reflectbox{$\lambda$}}$-term $\lambda x.\text{\reflectbox{λ}}x.\underline{x}$ the underlined occurrence of the variable x is free, since the binding effect of the λx is undone by the subsequent $\text{\reflectbox{$\lambda$}}x$. For another example, only the underlined

occurrence of x is free in $\lambda x.x(\reflectbox{\lambda} x.\underline{x})x$; the first and third occurrences of x are in scope of the λx (see Figure 3.1).

Definition 3.1.1 *The set* $(M, N, P \in)\reflectbox{\Lambda}$ *of* $\reflectbox{\lambda}$-terms *is defined by:*

$$\reflectbox{\Lambda} ::= \mathcal{V} \mid \reflectbox{\lambda} x.\reflectbox{\Lambda} \mid \lambda x.\reflectbox{\Lambda} \mid \reflectbox{\Lambda}\reflectbox{\Lambda}$$

where $(x, y, z \in)\mathcal{V}$ is a collection of variable(name)s with decidable equality:

Axiom 3.1 (Names with decidable equality) $x = y \vee x \neq y$, *for all* $x, y : \mathcal{V}$

We stress that Definition 3.1.1 is an *inductive* one without any reference to α-equivalence. That is, we do *not* assume the variable convention (2.1.13 of [4]). In fact, we don't need it, since in the process of β-reduction, we will not rename offending binders to avoid capturing of free variables. Instead, $\reflectbox{\lambda}$s will be inserted in an appropriate way, as will become clear in the sequel.

We adopt the usual notational conventions for the λ-calculus [4] (but *not* the variable convention), treating $\reflectbox{\lambda}$ analogously to λ. We use the notation $\lambda X.M$ and $\reflectbox{\lambda} X.M$ where X is a stack of variables $x_0, \ldots, x_n$, to denote $\lambda x_0 \cdots x_n.M$ resp. $\reflectbox{\lambda} x_0 \cdots x_n.M$.

Remark 3.1.1 *One way in which the usefulness of the $\reflectbox{\lambda}$-calculus is shown, is by deriving confluence of the standard λ-calculus from confluence of the $\reflectbox{\lambda}$-calculus. One way to do this would be to define standard λ-terms as $\reflectbox{\lambda}$-terms without occurrences of $\reflectbox{\lambda}$, and then prove some kind of conservativity result. Instead, in our* Coq *implementation λ-terms are defined separately from $\reflectbox{\lambda}$-terms and we use a canonical embedding from the former to the latter. In this way we hope to make it clear that it is really the* standard *λ-calculus we are proving confluent, and also to take away any suspicion of cheating (e.g. employing notions for $\reflectbox{\lambda}$-terms in the λ-calculus). In order to improve readability, mention of the function embedding λ-terms into $\reflectbox{\lambda}$-terms will be suppressed.*

In order to extend the notions of α-equivalence and β-reduction, we should first try to make some semantic sense of $\reflectbox{\lambda}$s. Thinking of λx and $\reflectbox{\lambda} x$ as (named) opening '$[_x$' and closing '$]_x$' brackets,[1] it is clear that $\reflectbox{\lambda}$-terms may come in different degrees of balancedness. For instance, scopes could seemingly be crossing one another as indicated by the boxes in:

$$P = \boxed{\lambda x.\boxed{\lambda y.\reflectbox{\lambda} x.}\reflectbox{\lambda} y.}Q$$

This would obviously cause semantical problems (try to define substitution). To overcome this problem we assume a simple minded jump semantics: an occurrence of $\reflectbox{\lambda} x.M$ implicitly ends the scopes of all (non-matching) λs inbetween that occurrence and its matching λx, just as the occurrence of the variable x in $\lambda x.\lambda y.x$ can be thought of as implicitly ending the scope of the λy. Hence P is

[1] But note that brackets (parentheses) usually apply 'horizontally' to the textual representation of terms, whereas λ and $\reflectbox{\lambda}$ apply 'vertically' to their abstract syntax trees (where brackets do not even occur).

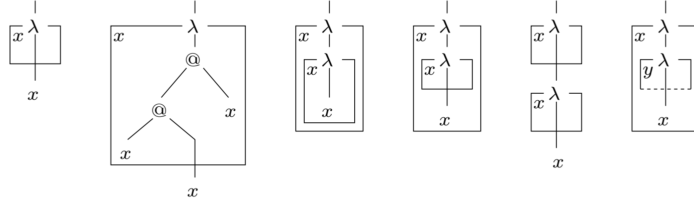

Figure 3.1: $\lambda x.\reflectbox{\ensuremath{\lambda}} x.x$, $\lambda x.x(\reflectbox{\ensuremath{\lambda}} x.x)x$, $\lambda x.\lambda x.x$, $\lambda x.\lambda x.\reflectbox{\ensuremath{\lambda}} x.x$, $\lambda x.\reflectbox{\ensuremath{\lambda}} x.\lambda x.\reflectbox{\ensuremath{\lambda}} x.x$, and $\lambda x.\lambda y.x$.

semantically equivalent to $\lambda x.\lambda y.\reflectbox{\ensuremath{\lambda}} y.\reflectbox{\ensuremath{\lambda}} x.\reflectbox{\ensuremath{\lambda}} y.Q$. Our definitions of α-equivalence and β-reduction and hence our definition of substitution, as will be presented below, are meant to reflect this intuitive (operational) semantics.

Apart from such *jump* terms we identify the useful subclasses of scope-balanced and balanced terms, both of which are closed under α-equivalence and β-reduction. Balanced terms can be used to represent nameless λ-terms using De Bruijn indices, by using only a single name (see the discussion in the paragraph on related work below). Ordinary λ-terms are not (necessarily) balanced, however they always are scope-balanced.

Definition 3.1.2 *A term is* scope-balanced *if it is scope-balanced* under *some stack X. A term M being* scope-balanced under *a stack X is denoted by $\langle X\rangle M$ and defined by:*

$$\frac{}{\langle X\rangle x} \qquad \frac{\langle xX\rangle M}{\langle X\rangle \lambda x.M} \qquad \frac{\langle X\rangle M}{\langle xX\rangle \reflectbox{\ensuremath{\lambda}} x.M} \qquad \frac{\langle X\rangle M \quad \langle X\rangle N}{\langle X\rangle MN}$$

Balancedness *is defined as scope-balancedness restricting the first clause to*

$$\frac{}{\langle xX\rangle x}$$

Here xX is the result of pushing x on the stack X.

For instance, $\lambda x.\lambda y.\reflectbox{\ensuremath{\lambda}} x.M$ is not scope-balanced (λy not closed before $\reflectbox{\ensuremath{\lambda}} x$), $\lambda y.x$ is scope-balanced but not balanced (λy not closed before x), and $\lambda x.\reflectbox{\ensuremath{\lambda}} x.M$ and $\lambda x.x$ are balanced (if M is in the former case).

It is easy to see that closed λ-terms are scope-balanced under any stack, hence in particular under the empty stack $\Box$. Scopes in balanced $\reflectbox{\ensuremath{\lambda}}$-terms can be neatly visualised as boxes in their abstract syntax tree, as shown in Figure 3.1.[2] Vice versa, in the term representation of a box, only its 'doors' are kept. That is, λs and $\reflectbox{\ensuremath{\lambda}}$s are used to demarcate all places where the boundary of the box is crossed by the abstract syntax tree. In fact, there is a strong similarity (see Figure 3.1) between balanced terms and the context-free string language of *matching brackets* as presented by the grammar:

$$P \quad ::= \quad \epsilon \mid [P] \mid PP$$

[2]Scopes in non-balanced terms can be drawn as floorless boxes ($\lambda x.\lambda y.x$ in Figure 3.1).

- Scopes can be nested (similar to $[P]$). In the λ-term $\lambda x.\lambda x.x$, the occurrence of x is implicitly assumed to be bound by the rightmost λx. Similarly, the scope of the rightmost λx is ended by the $\adbmal x$ in $\lambda x.\lambda x.\adbmal x.x$.
- Scopes can be concatenated (similar to PP). In the $\adbmal$-term $\lambda x.\adbmal x.\lambda x.\adbmal x.x$, the scopes of the two λxs do not have overlap/are not nested, in spite of the latter being 'to the right' of the former.

Indeed, the set of balanced $\adbmal$-terms can be generated by a so-called context-free term grammar, where context-free term grammars are the natural generalisation of context-free string grammars, see e.g. Section 2.5 of [23]. A difference between matching bracket strings and balanced $\adbmal$-terms is that, due to the branching structure of terms, *several* $\adbmal$'s may match the same λ as in $\lambda x.(\adbmal x.\underline{x})(\adbmal x.\underline{x})$, with both underlined occurrences of x free.

Related work This chapter is the full version of [34], and is under consideration for publication in the *Journal of Functional Programming*. Compared to [34] the results in the present chapter are more general, in particular the key substitution lemmas. Moreover, we have supplied more formal definitions and some typical proof ideas. For the complete proof development we refer the reader to [35].

When application of $\adbmal x$ is restricted to variables (and end-of-scopes), it corresponds to Berkling's lambda-bar [11], which is in turn seen to be a named version of the successor operator in De Bruijn's nameless (more precisely: single name) calculus [19]. Their calculi do not allow successions of boxes, only nestings of boxes. This corresponds to the sublanguage of the language of matching brackets (see above) generated by the grammar: $B ::= \epsilon \mid [B]$.

Restricting to a single name, i.e. to De Bruijn indices, $\adbmal x$ corresponds to the shift substitution $[\uparrow]$ in the λ-calculus with explicit substitutions $\lambda\sigma$ of [1], or the shift operation *Shi* of [21]. The earliest generalisation of De Bruijn indices seems to be due to Paterson [56]. The idea is to allow the successor S to appear on subterms, instead of just on indices; as it is written in [15]:

> Substitution on de Bruijn terms transforms arguments as well as function bodies, thus precluding sharing. Consider the example term from Section 1, with the variables written in unary notation:
>
> $$\lambda.0\,(\lambda.\mathsf{S}0\,0\,(\lambda.\mathsf{SS}0\,\mathsf{S}0\,0))$$
>
> If this term is applied to the term $\lambda.0\,\mathsf{S}0$, the result is
>
> $$(\underline{\lambda.0\,\mathsf{S}0})\,(\lambda.(\underline{\lambda.0\,\mathsf{SS}0})\,0\,(\lambda.(\underline{\lambda.0\,\mathsf{SSS}0})\,\mathsf{S}0\,0))$$
>
> where the three versions of the argument are underlined. There is a generalisation of de Bruijn notation in which S can be applied to any term, not just a variable (Paterson, 1991). Its effect is to escape the scope of the matching λ. With this looser representation of terms, one can avoid transforming arguments while substituting. In the above example, substitution yields
>
> $$(\underline{\lambda.0\,\mathsf{S}0})\,(\lambda.\mathsf{S}(\underline{\lambda.0\,\mathsf{S}0})\,0\,(\lambda.\mathsf{SS}(\underline{\lambda.0\,\mathsf{S}0})\,\mathsf{S}00))$$
>
> In effect, we have postponed pushing the S's down to the variables.

where the 'example term from Section 1' is the λ-term $\lambda x.x\ (\lambda y.x\ y\ (\lambda z.x\ y\ z))$ which translates to $\lambda 0.\ (\lambda.1\ 0\ (\lambda.2\ 1\ 0))$ in (their) De Bruijn notation. Applying the λ-term to a named version, say $\lambda x.xy$ of $\lambda.0\,\mathsf{S}0$, yields (see Subsection 3.4.2) $(\underline{\lambda x.xy})\ (\lambda y.(\reflectbox{\lambda} y.\underline{\lambda x.xy})\ y\ (\lambda z.(\reflectbox{\lambda} z.\reflectbox{\lambda} y.\underline{\lambda x.xy}))\ y\ z)$. So obviously $\reflectbox{\lambda}$ corresponds to the successor S in De Bruijn notation.

Bird and Paterson go on to show that in the balanced (single name) case the term language of the $\reflectbox{\lambda}$-calculus is context-free by presenting it by means of the following context-free term grammar:

$$\begin{array}{rcl} \mathit{Term}\ a & ::= & \mathit{Var}\ a \mid \mathit{App}(\mathit{Term}\ a, \mathit{Term}\ a) \mid \mathit{Abs}(\mathit{Term}(\mathit{Incr}(\mathit{Term}\ a))) \\ \mathit{Incr}\ a & ::= & \mathit{Zero} \mid \mathit{Succ}\ a \end{array}$$

the idea being that *Term*s are balanced by generating *Incr*s, i.e. variables (*Zero*s) or end-of-scopes (*Succ*s), at the same time as their matching *Abs* (abstraction).[3]

When restricting to the balanced case, our boxes correspond closely to boxes in MELL proof nets for linear logic, see e.g. [46]. In fact, in our optimal implementation (see Section 3.5) λx disintegrates into a λ (a par in MELL) and (part of the boundary of) an x-box ((Asperti's version of) a box in MELL), upon encountering an application. One can think of these two phases of abstraction as turning a free variable x into a bound one by closing it off from the outside world inside an x-box, but providing a handle to x to the outside world again in the form of the λ. Many proposals for decomposing abstraction into more elementary notions can be found in the literature, a recent one being [5]. Similarly, notions of enclosure abound. Analogous to the conflation of the enclosure with the enclosed as found in (the etymology of) words such as town, garden, park and paradise, these formalisations may or may not make the boundary explicit, see e.g. [20, 55, 16] for some recent ones.

In the area of dynamic semantics for natural language, a stack-based semantics for a variant of predicate logic is presented in [37]. Although, the exact relationship is not clear to us yet, a difference seems to be that in their semantics every variable has its own stack, whereas we have a single stack. However, also in [11] variables have their own stack.

Implementation All results informally presented here are formalised in Coq. The source files are available from [35]. The size of the development is 9543 lines of Coq-code, 259160 bytes; 324 lemmas are proved. The Coq proofs and the $\reflectbox{\lambda}$-calculus were developed concurrently. The total development time is estimated one man-year approximately.

Outline The outline of the rest of this chapter is as follows. In Section 3.2 we define some preliminary notions on abstract rewriting systems. We provide several definitions of α-equivalence for $\reflectbox{\lambda}$-terms in Section 3.3, extending classical

[3] This does not work (directly) for non-balanced terms in the many-variable case.

definitions as found in the literature on the λ-calculus, prove them to be decidable congruence relations, and show them to be equivalent. Then we present a definition of β-reduction for ƛ-terms in Section 3.4, extending the usual definition for the λ-calculus, and prove this notion of reduction to be confluent *without* α-equivalence. In both (α and β) cases it is shown how the results on the ƛ-calculus entail the corresponding results for the ordinary λ-calculus, e.g. confluence of β-reduction up to α-equivalence. Applications are presented in Section 3.5. Finally, in Section 3.6, we conclude, and discuss upon the relationship of the ƛ-calculus to explicit substitution calculi having the property of preservation of normalisation.

Acknowledgments We would like to thank the participants of the TCS seminar at the Vrije Universiteit Amsterdam, PAM and the 7th Dutch Proof Tools Day both at CWI, Amsterdam, ZIC at the Technische Universiteit Eindhoven, the CS seminar at the University of Leicester, and the TF lunch seminar at the Universiteit Utrecht, for feedback. Eduardo Bonelli, Marko van Eekelen, Joost Engelfriet, Stefan Kahrs, Kees Vermeulen, Albert Visser, and the CADE referees provided useful comments and pointers to the literature.

3.2 Preliminaries

Definition 3.2.1 *We define the n-fold composition $\rightarrow^n$ of a binary relation $\rightarrow$ as the smallest relation satisfying the following clauses:*

$$\frac{}{x \rightarrow^0 x}\ \textit{reflexivity} \qquad \frac{x \rightarrow y}{x \rightarrow^1 y}\ \textit{embedding} \qquad \frac{x \rightarrow^n y \quad y \rightarrow^m z}{x \rightarrow^{n+m} z}\ \textit{transitivity}$$

The reflexive–transitive closure $x \rightarrow^* y$ of $\rightarrow$ is defined as $\exists n.x \rightarrow^n y$. The equivalence closure *$\leftrightarrow^*$ of $\rightarrow$ is defined as $\exists n.x \leftrightarrow^n y$, where $\leftrightarrow^n$ is defined inductively by the former three clauses (replacing all occurrences $\rightarrow^k$ with $\leftrightarrow^k$) plus the following one:*

$$\frac{y \leftrightarrow^n x}{x \leftrightarrow^n y}\ \textit{symmetry}$$

Definition 3.2.2 *A binary relation R on a set A has the* diamond property*, if for all $a, b, c : A$, $a\ R\ b$ and $a\ R\ c$ implies there exists $d : A$, such that $c\ R\ d$ and $b\ R\ d$. R is* confluent *if its reflexive–transitive closure R^* has the diamond property. We say R has the* diamond property up to *S if for all $a, b, c : A$, $a\ R\ b$ and $a\ R\ c$ implies there exist $d, d' : A$, such that $c\ R\ d$, $b\ R\ d'$, and $d\ S\ d'$. R is* confluent up to *S if R^* has the diamond property up to S.*

Note that the ordinary diamond property is equivalent to the diamond property up to identity.

Definition 3.2.3 *We say R* transits *S if $R \subseteq S \subseteq R^*$, where $R_1 \subseteq R_2$ is defined by $\forall xy.x\ R_1\ y \Rightarrow x\ R_2\ y$.*

Lemma 3.2.1 *If R has the diamond property, then it is confluent.*

Proof. (See, for example, [67].) By induction on the complexity of the diverging steps, loading it by: converging steps have the same complexity as opposite diverging steps. Here we express the complexity by the number of R-steps. The following statement is proved by induction on the diverging steps:

$$x \; R^n \; y \wedge x \; R^m \; z \Rightarrow \exists u.y \; R^m \; u \wedge z \; R^n \; u$$

Lemma 3.2.2 *If R transits S and S has the diamond property, then R is confluent.*

Proof. By monotonicity (if $R \subseteq R'$, then $R^* \subseteq R'^*$), idempotence ($R^{**} = R^*$) of *, and the first assumption, we have $R^* \subseteq S^* \subseteq R^*$. We conclude from the previous lemma and the second assumption.

3.3 α

We present three distinct definitions of α-equivalence for the λ-calculus known from the literature, in historical order. We then compare these notions, present our adaptations of each of them to the ʎ-calculus, and prove them to be equivalent. For this the existence of fresh variables is required:

Axiom 3.2 (Fresh variable) *For any given (finite) stack of variables, there is a variable not among these, i.e. a fresh variable, $\forall X.\exists x.x \notin X$.*

3.3.1 $\lambda\alpha$

Church

Our first notion of α-equivalence is the usual one based on Church's Postulate I for the λ-calculus [22], which reads (page 355):

> *If* J *is true, if* L *is well-formed, if all the occurrences of the variable* x *in* L *are occurrences as a bound variable, and if the variable* y *does not occur in* L*, then* K*, the result of substituting* $\mathsf{S}^{\mathsf{x}}_{\mathsf{y}}\,\mathsf{L}|$ *for a particular occurrence of* L *in* J*, is also true.*

where $\mathsf{S}^{\mathsf{X}}_{\mathsf{Y}}\,\mathsf{U}|$ *represents the formula which results when we operate on the formula* U *by replacing* X *by* Y *throughout, where* Y *may be any symbol or formula but* X *must be a single symbol, not a combination of symbols* (page 350 of [22]).

Due to Curry, Postulate I is nowadays known as the α-conversion rule. An α-conversion *step* is obtained from the α-conversion rule by allowing its application to any subterm of a term. An α-*conversion* consists of a sequence of α-conversion steps. Finally, a term is said to be α-equivalent to another one, if there exists an α-conversion relating the former to the latter.

An advantage of this definition is that it is operational and fine-grained; each α-conversion step itself is easy to understand since it does only little work. A disadvantage of this fine-grainedness is that it is at first sight not clear whether structural properties such as symmetry and decidability of α-conversion hold. Moreover, it needs the Fresh variable axiom due to the *Extra-hand principle*: if both your hands are full, you need a third hand in order to swap their contents.[4]

Example 3.3.1 *The terms $\lambda x.\lambda y.xy$ and $\lambda y.\lambda x.yx$ are α-equivalent. However, both α-conversion steps replacing x by y and vice versa are forbidden. Hence, an α-conversion needs to introduce a third, fresh, variable, say z, first:*

$$\lambda \underline{x}.\lambda y.\underline{x}y \to_\alpha \lambda z.\lambda \underline{y}.z\underline{y} \to_\alpha \lambda \underline{z}.\lambda x.\underline{z}x \to_\alpha \lambda y.\lambda x.yx$$

where we have underlined in each term the variables that are converted in the subsequent step.

Schroer

In order to prove symmetry and decidability of α-equivalence as defined in the previous paragraph, one may try to find a strategy for α-conversion such that the number of α-conversion steps needed in a conversion from s to t is bounded by, say, the sum of the sizes of s and t. An obvious way to bound the number of steps is by restricting α-conversion by:

> Never rename twice.

However, from Example 3.3.1 we immediately see that this is too strict a restriction; the leftmost λ-abstraction needs to be renamed twice. Hence renaming once is not enough, but, as the example suggests our assumption may be replaced by:

> Never rename thrice.

Such an idea appears at least as early as Schoer's PhD thesis, see page 384 of his [62]:

```
        Scholium 3.44. The proof of Theorem 3.44 below
has as its germ the following procedure to determine of
A,B ε Wocc  whether or not  A adj B: Let Z₁, Z₂,... be
singleton expressions of the alphabetically earliest
variables not occurring at all in either of  A,B ,
enumerated without repetitions.  In each of  A,B , change
quantifiers from left to right, replacing the given variables
by the  Z's  in order.  There will result  A',B'  such that
A adj A' . B adj B' , and such that  A adj B . ≡ .  A' = B' .
```

where `adj` is his notion of α-equivalence and `Theorem 3.44` states decidability.

[4]There is the well-known way to swap the contents of two registers *in situ* by performing three exclusive-or's (xor); in Java: `r1 ^= r2; r2 ^= r1; r1 ^= r2` where `op1 ^= op2` is equivalent to `op1 = op1 ^ op2` and `^` is bitwise xor. Here, we will *not* assume the structure needed for this, e.g. a Boolean ring on the variables.

Example 3.3.2 *Applied to Example 3.3.1 Schroer's procedure yields:*

$$\lambda x.\lambda y.xy \to_\alpha \lambda z_1.\lambda y.z_1 y \to_\alpha \lambda z_1.\lambda z_2.z_1 z_2 \leftarrow_\alpha \lambda z_1.\lambda x.z_1 x \leftarrow_\alpha \lambda y.\lambda x.yx$$

Of course, to prove that this is an α-conversion one needs to prove that the last two backward α-steps are forward α-steps as well; they are.

Symmetry of a definition based on Schroer's procedure is trivial, decidability and reflexivity are also not too difficult, but now transitivity is not so simple because of the choosing of the `alphabetically earliest variables not occurring at all in either of  A,B` which may differ for `A,B` and `B,C`, when proving `A adj C`.[5] Also note that the procedure is *not* very parsimonious; it allocates as many fresh variables as there are λ-abstractions (quantifiers) in a term, where a single one (one extra hand) would suffice, as noted, e.g., by [27]. This fact may be seen by proceeding in a top-down fashion, the only interesting case being abstraction:

(abstraction) Suppose $s = \lambda x.s'$ and $t = \lambda y.t'$, such that the variables 'above' them have already been made identical. We proceed by first converting *every* x in s into z. Next, *every* y is converted into x and finally, *every* z is converted into y, resulting in, say, $\hat{s}$. Now $\hat{s}$ and t have the same initial binder, and we proceed on the respective subterms. To prove that this procedure is correct, one uses that y does not occur free in s' since otherwise s and t would not be α-equivalent. Symmetrically, x does not occur free in t'.

Example 3.3.3 *α-converting, say, $\lambda x_1 x_2 x_3.x_1 x_2 x_3$ into $\lambda x_2 x_3 x_1.x_2 x_3 x_1$ using this procedure proceeds as follows. First, we swap, using a fresh variable y, x_1 with x_2 yielding $\lambda x_2 x_1 x_3.x_2 x_1 x_3$. Hence the first variable has been appropriately renamed, and we may proceed on the respective subterms. In order to α-convert $\lambda x_1 x_3.x_2 x_1 x_3$ into $\lambda x_3 x_1.x_2 x_3 x_1$, we swap, using the same y, x_1 and x_3 in the former yielding the latter and we are done (formally one needs to continue with twice the subterm $\lambda x_1.x_2 x_3 x_1$, but nothing 'happens' anymore.)*

Kahrs

Both the problem of showing transitivity and the need for the Fresh variable axiom can be overcome by making renaming implicit. That is, instead of explicitly relating terms by explicitly renaming variables, one may set up an (implicit) correspondence between their respective variables. For instance, the two terms in Example 3.3.1 are shown α-equivalent by letting x and y in the first correspond to y and x in the second. However, the correspondence needs more structure than just a bijection between the sets of variables in both terms.

Example 3.3.4 *The terms $\lambda x.x\lambda y.y$ and $\lambda x.x\lambda x.x$ are α-equivalent, but this cannot be shown by means of a bijection between variables.*

[5] Compared to Church's α-conversion Schroer's procedure needs variables to be alphabetically sorted. Here, we will *not* assume the structure needed for this (e.g. a well-order) on the collection of variables.

$$\frac{}{\epsilon \vdash x = x} \quad \frac{}{\Gamma, x = y \vdash x = y} \quad \frac{v \neq x \quad y \neq z \quad \Gamma \vdash v = z}{\Gamma, x = y \vdash v = z}$$

$$\frac{x, y \in \mathit{Var} \quad \Gamma \vdash x = y}{\Gamma \vdash x \equiv y} \quad \frac{F \in \mathit{Sym}}{\Gamma \vdash F \equiv F} \quad \frac{\Gamma, x = y \vdash t \equiv u}{\Gamma \vdash [x]t \equiv [y]u} \quad \frac{\Gamma \vdash A \equiv C \quad \Gamma \vdash B \equiv D}{\Gamma \vdash AB \equiv CD}$$

Figure 3.2: Proof rules for α-congruence [43].

To define α-equivalence inductively, one has to set up a correspondence between *stacks* of variables. Such an idea appears in Kahrs' paper [43]; to quote from it:

> We also define a notion of α-congruence for our terms. It is the usual one, but we shall use it in a slightly more general setting, based on proof rules.
>
> **Definition 11**. *Sentences* are of the form $\Gamma \vdash t \equiv u$ or $\Gamma \vdash x = y$, where x and y are variables, t and u are terms of the same type and arity, and Γ is an *environment*. An environment is a list $x_1 = y_1, \cdots, x_n = y_n$ of equations between variables. We write ϵ for the empty environment ($n = 0$). A sentence *holds*, if it can be derived by the proof rules in figure 2.

where we present the proof rules of 'figure 2' in Figure 3.2. It is to be understood that two terms A and B are α-equivalent, if $\epsilon \vdash A \equiv B$ can be derived by the proof rules in Figure 3.2. One easily proves by induction that α-congruence defined in this way, has all the desired structural properties, e.g. transitivity and decidability. But, of course, it is less clear how to decompose α-equivalence into 'atomic' renaming steps.

3.3.2 ʎα

We show that each of the three definitions of α-equivalence can be straightforwardly extended from λ-terms to ʎ-terms. In each case, we highlight the key aspect of our formalisation in Coq. We start with some necessary technicalities. Apart from the 'renaming of boxes' (Definition 3.3.2), these may be skipped by the experienced reader who is referred to page 61.

We shall need to test whether a variable is fresh with respect to a term.

Definition 3.3.1 *The relation $x \in M$, saying whether x occurs in M (free or bound) is defined by:*

$$\begin{array}{l} x \in y, \text{ if } x = y \\ x \in \lambda y.M, \text{ if } x = y \text{ or } x \in M \\ x \in \text{ʎ} y.M, \text{ if } x = y \text{ or } x \in M \\ x \in MN, \text{ if } x \in M \text{ or } x \in N \end{array}$$

We say x is fresh for *M if $x \notin M$.*

As usual, α-conversion is defined using a renaming function to identify expressions that differ only in the names assigned to their bound variables. For $\reflectbox{\ensuremath{\lambda}}$-terms this means the renaming of *boxes* (viz. Figure 3.1). Renaming the (outer) x-box in $\lambda x.M$ into a y-box, means to replace λx by λy and to replace all *matching* occurrences of variables x and end-of-scopes $\reflectbox{\ensuremath{\lambda}} x$ in M by y and $\reflectbox{\ensuremath{\lambda}} y$ respectively. The first argument of renaming ($x{:=}y$) is fixed (as usual). Matching is performed using a stack (the second argument of renaming, initially empty) to record the binders encountered while descending recursively; it is pushed upon when passing an abstraction, and popped from when meeting an end-of-scope:

Definition 3.3.2 Renaming the outer x-box in $\lambda x.M$ into a y-box, $\lambda y.M[x{:=}y]$, *is defined using* $M[x{:=}y] = M[x{:=}y, \Box]$, *where* $M[x{:=}y, Z]$ *is defined by the following recursive equations:*

$$
\begin{array}{rcll}
z[x{:=}y, Z] & = & z, \textit{ if } z \in Z \textit{ or } z \neq x & (1) \\
z[x{:=}y, Z] & = & y, \textit{ if } z \notin Z \textit{ and } z = x & (2) \\
(\lambda z.M)[x{:=}y, Z] & = & \lambda z.M[x{:=}y, zZ] & (3) \\
(\reflectbox{\ensuremath{\lambda}} z.M)[x{:=}y, \Box] & = & \reflectbox{\ensuremath{\lambda}} y.M, \textit{ if } x = z & (4) \\
(\reflectbox{\ensuremath{\lambda}} z.M)[x{:=}y, \Box] & = & \reflectbox{\ensuremath{\lambda}} z.M, \textit{ if } x \neq z & (5) \\
(\reflectbox{\ensuremath{\lambda}} z.M)[x{:=}y, z'Z] & = & \reflectbox{\ensuremath{\lambda}} z.M[x{:=}y, Z], \textit{ if } z = z' & (6) \\
(\reflectbox{\ensuremath{\lambda}} z.M)[x{:=}y, z'Z] & = & (\reflectbox{\ensuremath{\lambda}} z.M)[x{:=}y, Z], \textit{ if } z \neq z' & (7) \\
(M_1 M_2)[x{:=}y, Z] & = & M_1[x{:=}y, Z] M_2[x{:=}y, Z] & (8)
\end{array}
$$

In clause (5) $\reflectbox{\ensuremath{\lambda}} z$ implicitly closes the scope of x; therefore, in the result, we can think of $\reflectbox{\ensuremath{\lambda}} z$ as implicitly closing the scope of y.

Example 3.3.5 $(x\reflectbox{\ensuremath{\lambda}} x.x)(\lambda x.(x\reflectbox{\ensuremath{\lambda}} x.x))[x{:=}y] = (y\reflectbox{\ensuremath{\lambda}} y.x)\lambda x.(x\reflectbox{\ensuremath{\lambda}} x.y)$

In case of ordinary λ-terms the stack Z only grows during renaming, rendering the variable to be renamed inaccessible once it is abstracted from again:

Lemma 3.3.1 *If M is a λ-term, then* $(\lambda x.M)[x{:=}y] = \lambda x.M$.

Proof. Note that $(\lambda x.M)[x{:=}y] = \lambda x.M[x{:=}y, x]$. One proves by induction on the λ-term M, that if x occurs in X, then $M[x{:=}y, X] = M$.

The following two properties of renaming will be needed in the proof of commutation of β-reduction and α-equivalence. The first states commutativity of renaming:

Lemma 3.3.2 $M[x{:=}z, XyY][y{:=}z', X] = M[y{:=}z', X][x{:=}z, Xz'Y]$, *if z fresh for y, M, X and z' fresh for M, X.*

Secondly, renaming x into z, followed by renaming z into z', amounts to the same as directly renaming x into z':

Lemma 3.3.3 $M[x{:=}z, X][z{:=}z', X] = M[x{:=}z', X]$, *if* $z \notin M, X$.

When we project β-reductions in the $\reflectbox{\ensuremath{\lambda}}$-calculus to β-reductions in the λ-calculus (Section 3.4.4), we need to reason about the set of free variables of a term:

Definition 3.3.3 *The set of free variables of a term M, $\mathsf{FV}(M)$, is defined as $\mathsf{FV}(M, \Box)$, where $\mathsf{FV}(M, X)$ is defined by the following recursive equations.*

$$
\begin{array}{rcll}
\mathsf{FV}(x, X) & = & \{x\} - X & (1) \\
\mathsf{FV}(\lambda x.M, X) & = & \mathsf{FV}(M, xX) & (2) \\
\mathsf{FV}(\reflectbox{λ} x.M, \Box) & = & \mathsf{FV}(M, \Box) & (3) \\
\mathsf{FV}(\reflectbox{λ} x.M, x'X) & = & \mathsf{FV}(M, X), \textit{ if } x = x' & (4) \\
\mathsf{FV}(\reflectbox{λ} x.M, x'X) & = & \mathsf{FV}(\reflectbox{λ} x.M, X), \textit{ if } x \neq x' & (5) \\
\mathsf{FV}(MN, X) & = & \mathsf{FV}(M, X) \cup \mathsf{FV}(N, X) & (6)
\end{array}
$$

Note that, in clause (5), $\reflectbox{$\lambda$} x$ implicitly closes the scope x', which is therefore popped from the stack of currently open scopes.

Remark 3.3.1 *Note that if M is balanced under X we have $\mathsf{FV}(M, X) = \emptyset$. This implies that terms balanced under the empty stack are closed.*

In the sequel it will sometimes be useful to forget about the binding structure of terms. To that end, we map terms to first-order terms by simply forgetting names (equivalently: mapping all names to a single one):

Definition 3.3.4 *First we define a set T of first-order terms:*

$$T ::= \odot \mid \reflectbox{λ} T \mid \lambda T \mid @TT$$

The skeleton $[M]_{\mathsf{skel}}$ *of an $\reflectbox{$\lambda$}$-term M is such a first-order term defined inductively by:*

$$
\begin{array}{rcl}
[x]_{\mathsf{skel}} & = & \odot \\
[\reflectbox{λ} x.M]_{\mathsf{skel}} & = & \reflectbox{λ}[M]_{\mathsf{skel}} \\
[\lambda x.M]_{\mathsf{skel}} & = & \lambda[M]_{\mathsf{skel}} \\
[M_1 M_2]_{\mathsf{skel}} & = & @[M_1]_{\mathsf{skel}}[M_2]_{\mathsf{skel}}
\end{array}
$$

For instance, the proof of Lemma 3.4.20 proceeds by induction over the skeleton of a term, using the fact that renaming preserves skeletons:

Lemma 3.3.4 $[M[x{:=}y]]_{\mathsf{skel}} = [M]_{\mathsf{skel}}$.

Unary contexts are used to express congruences.

Definition 3.3.5 Unary contexts *are typed $\Lambda \to \Lambda$ and built from*

$$[\,], C \circ C', \lambda x.[\,], \reflectbox{λ} x.[\,], M[\,], [\,]M$$

where C, C' are unary contexts. We write $C[M]$ to denote the result of filling the hole in C by term M; $(C \circ C')[M] = C[C'[M]]$.

Now that we have defined some technical preliminaries, we are ready to extend the three notions of α-equivalence given in Subsection 3.3.1 to the $\reflectbox{$\lambda$}$-calculus.

Church

The notion of α-conversion is extended to the $\reflectbox{$\lambda$}$-calculus.

Definition 3.3.6 *The α-rule is $\lambda x.M \to \lambda y.M[x{:=}y]$ if $y \notin M$. Single-step α-renaming, $\to_\alpha$, is defined as the compatible closure of the α-rule:*

$$\frac{(M,N) \in \alpha}{M \to_\alpha N} \qquad \frac{M \to_\alpha N}{\lambda x.M \to_\alpha \lambda x.N} \qquad \frac{M \to_\alpha N}{\reflectbox{λ} x.M \to_\alpha \reflectbox{λ} x.N}$$

$$\frac{M \to_\alpha M'}{MN \to_\alpha M'N} \qquad \frac{N \to_\alpha N'}{MN \to_\alpha MN'}$$

The relation $=^{\mathsf{c}}_\alpha$, which we define as $\leftrightarrow^*_\alpha$, i.e. the equivalence closure of $\to_\alpha$, is called α-conversion.

The clause dealing with $\reflectbox{$\lambda$}$ is just a compatibility clause, cf. 3.1.1. Definition of [4], since at the time one comes across an $\reflectbox{$\lambda$}$, all the (renaming) work has already been performed by its matching abstraction. Due to Lemma 3.3.1, our definitions of α-step and α-conversion coincide with that of [4] in the case of λ-terms.

Schroer

Our definition of α-equivalence à la Schroer makes use of an auxiliary stack Z which records the variables chosen thusfar for renaming.

Definition 3.3.7 α-equivalence à la Schroer*, $M =^{\mathsf{s}}_\alpha N$, is defined as $\exists Z.M =^Z_\alpha N$, where $M =^Z_\alpha N$ is defined by:*

$$\frac{M[x{:=}z] =^Z_\alpha N[y{:=}z] \quad z \notin M, N, Z}{\lambda x.M =^{zZ}_\alpha \lambda y.N}$$

$$\frac{}{x =^Z_\alpha x} \qquad \frac{M =^Z_\alpha N}{\reflectbox{λ} x.M =^Z_\alpha \reflectbox{λ} x.N} \qquad \frac{M =^Z_\alpha M' \quad N =^Z_\alpha N'}{MN =^Z_\alpha M'N'}$$

Again all the work is performed in the clause for abstraction. Compared to α-conversion $=^{\mathsf{c}}_\alpha$ above, the variable chosen for renaming is now much fresher: not only must it be fresh for M, but also for N and for the variables Z chosen thusfar. The clause dealing with $\reflectbox{$\lambda$}$ is just a compatibility clause, as above.

Kahrs

Our definition of α-equivalence à la Kahrs reads as follows. It makes use of two auxiliary stacks (both initially empty), to set up the correspondence between the variables in M and N mentioned above.

Definition 3.3.8 *We define $M =^{\mathsf{k}}_{\alpha} N$, if $\langle\Box\rangle M =^{\mathsf{k}}_{\alpha} \langle\Box\rangle N$, where for stacks of variables X and Y, $\langle X\rangle M =^{\mathsf{k}}_{\alpha} \langle Y\rangle N$ is inductively defined as follows:*

$$
\begin{array}{rcl}
\langle\Box\rangle x & =^{\mathsf{k}}_{\alpha} & \langle\Box\rangle x \\
\langle xX\rangle x & =^{\mathsf{k}}_{\alpha} & \langle yY\rangle y,\ \textit{if}\ |X| = |Y| \\
\langle x'X\rangle x & =^{\mathsf{k}}_{\alpha} & \langle y'Y\rangle y,\ \textit{if}\ x' \neq x,\ y' \neq y,\ \textit{and}\ \langle X\rangle x =^{\mathsf{k}}_{\alpha} \langle Y\rangle y \\
\langle X\rangle \lambda x.M & =^{\mathsf{k}}_{\alpha} & \langle Y\rangle \lambda y.N,\ \textit{if}\ \langle xX\rangle M =^{\mathsf{k}}_{\alpha} \langle yY\rangle N \\
\langle\Box\rangle ʎx.M & =^{\mathsf{k}}_{\alpha} & \langle\Box\rangle ʎx.N,\ \textit{if}\ \langle\Box\rangle M =^{\mathsf{k}}_{\alpha} \langle\Box\rangle N \\
\langle xX\rangle ʎx.M & =^{\mathsf{k}}_{\alpha} & \langle yY\rangle ʎy.N,\ \textit{if}\ \langle X\rangle M =^{\mathsf{k}}_{\alpha} \langle Y\rangle N \\
\langle x'X\rangle ʎx.M & =^{\mathsf{k}}_{\alpha} & \langle y'Y\rangle ʎy.N,\ \textit{if}\ x' \neq x,\ y' \neq y\ \textit{and}\ \langle X\rangle ʎx.M =^{\mathsf{k}}_{\alpha} \langle Y\rangle ʎy.N \\
\langle X\rangle M_1 M_2 & =^{\mathsf{k}}_{\alpha} & \langle Y\rangle N_1 N_2,\ \textit{if}\ \langle X\rangle M_1 =^{\mathsf{k}}_{\alpha} \langle Y\rangle N_1\ \textit{and}\ \langle X\rangle M_2 =^{\mathsf{k}}_{\alpha} \langle Y\rangle N_2
\end{array}
$$

where $|X|$ denotes the length of stack X.

The variable, abstraction and application clauses in the definition above can easily be seen to be corresponding to the clauses in Figure 3.2. The end-of-scope clauses are analogous to the clauses for variables. Unique reading holds up to α-equivalence:

Lemma 3.3.5 *If $\langle X\rangle M =^{\mathsf{k}}_{\alpha} \langle Y\rangle N$, then*

- *M and N have the same skeleton: $[M]_{\mathsf{skel}} = [N]_{\mathsf{skel}}$.*
- *X and Y have the same length: $|X| = |Y|$.*
- *M and N have the same set of free variables: $\mathsf{FV}(M, X) = \mathsf{FV}(N, Y)$.*

As a consequence we have conservativity of α-equivalence over the ʎ-calculus (Lemma 3.3.7). Note that the third end-of-scope clause of Definition 3.3.8 expresses that ending the scope of some variable x automatically ends the scope of the variables which were bound later than x. By conservativity of $=^{\mathsf{k}}_{\alpha}$, this clause can be omitted for scope-balanced terms. For balanced terms we can do with only four clauses:

$$
\begin{array}{rcl}
\langle x\rangle x & =^{\mathsf{k}}_{\alpha} & \langle y\rangle y \\
\langle xX\rangle ʎx.M & =^{\mathsf{k}}_{\alpha} & \langle yY\rangle ʎy.N \\
\langle X\rangle \lambda x.M & =^{\mathsf{k}}_{\alpha} & \langle Y\rangle \lambda y.N, \text{ if } \langle xX\rangle M =^{\mathsf{k}}_{\alpha} \langle yY\rangle N \\
\langle X\rangle M_1 M_2 & =^{\mathsf{k}}_{\alpha} & \langle Y\rangle N_1 N_2, \text{ if } \langle X\rangle M_1 =^{\mathsf{k}}_{\alpha} \langle Y\rangle N_1 \text{ and } \langle X\rangle M_2 =^{\mathsf{k}}_{\alpha} \langle Y\rangle N_2
\end{array}
$$

If $\langle X\rangle M =^{\mathsf{k}}_{\alpha} \langle Y\rangle N$ holds, then the pair of stacks (X,Y) can be seen as an update on the identity relation (which obviously is a bijection) on variable names, the result of which is a bijection between the 'free' variables of M and N. Indeed, as stated by the following lemma, inserting the same variable on the bottom of both stacks is irrelevant. This lemma is needed, e.g., to show that $=^{\mathsf{k}}_{\alpha}$ is a congruence (abstraction case).

Lemma 3.3.6 *$\langle X\rangle M =^{\mathsf{k}}_{\alpha} \langle Y\rangle N$ iff $\langle Xz\rangle M =^{\mathsf{k}}_{\alpha} \langle Yz\rangle N$*

Proof. The proof is by induction on the derivations. The only interesting cases are the variable and end-of-scope cases, which are similar. So, suppose $M = x$ and $N = y$.

($\Leftarrow$) By induction on stack X, and inversion[6] of the instances of $\langle Xz\rangle x =^{\mathsf{k}}_{\alpha} \langle Yz\rangle y$.

- If $X = \square$, then $Y = \square$ (by Lemma 3.3.5), and $\langle\square\rangle x =^{\mathsf{k}}_{\alpha} \langle\square\rangle y$ follows, since either $x = z = y$ or $x \neq z \neq y$.
- If $X = x'X'$, then $Y = y'Y'$ (by Lemma 3.3.5). Inverting $\langle x'X'z\rangle x =^{\mathsf{k}}_{\alpha} \langle y'Y'z\rangle y$ gives two possibilities. Either $x = x'$, $y = y'$ and $|X'z| = |Y'z|$, then $\langle x'X'\rangle x =^{\mathsf{k}}_{\alpha} \langle y'Y'\rangle y$ follows by application of the second clause; or $x \neq x'$ and $y \neq y'$, then our goal follows by application of the third clause of $=^{\mathsf{k}}_{\alpha}$ and the induction hypothesis.

($\Rightarrow$)

- Case $\langle\square\rangle x =^{\mathsf{k}}_{\alpha} \langle\square\rangle x$. If $x = z$, then we conclude by application of the second defining clause of $=^{\mathsf{k}}_{\alpha}$. If $x \neq z$, then $\langle z\rangle x =^{\mathsf{k}}_{\alpha} \langle z\rangle x$ follows from the third clause of $=^{\mathsf{k}}_{\alpha}$ and the assumption.
- Case $\langle xX\rangle x =^{\mathsf{k}}_{\alpha} \langle yY\rangle y$ with $|X| = |Y|$; so $|Xz| = |Yz|$ and $\langle xXz\rangle x =^{\mathsf{k}}_{\alpha} \langle yYz\rangle y$ follows from application of the second clause of $=^{\mathsf{k}}_{\alpha}$.
- Case $\langle x'X\rangle x =^{\mathsf{k}}_{\alpha} \langle y'Y\rangle y$, where $x \neq x'$ and $y \neq y'$; then $\langle x'Xz\rangle x =^{\mathsf{k}}_{\alpha} \langle y'Yz\rangle y$ follows from application of the third rule of $=^{\mathsf{k}}_{\alpha}$ and the induction hypothesis.

Results on α-equivalences

Theorem 3.3.1 *All three notions of α-equivalence are equivalent:*

$$=^{\mathsf{c}}_{\alpha} \; = \; =^{\mathsf{s}}_{\alpha} \; = \; =^{\mathsf{k}}_{\alpha}$$

Note that to prove that $\reflectbox{$\lambda$}$-terms which are α-equivalent à la Kahrs are α-equivalent according to the other two definitions, one essentially uses the Fresh variable axiom. (It is not needed in the other direction.)

Theorem 3.3.2 *α-equivalence is a congruent equivalence relation.*

Proof. Taking the inductive definition of Kahrs, the results are all proven by straightforward inductions on the definition, loading them appropriately with stacks.

Lemma 3.3.7 *α-equivalence preserves $\reflectbox{$\lambda$}$-terms, scope-balancedness, balancedness, and λ-terms.*

[6] *Inverting* a statement $P(t)$, where P is an inductive predicate, means to derive for each possible constructor $c_i : A_1 \to \cdots \to A_n \to P(t)$ all the necessary conditions $A_1, \ldots, A_n$ that should hold for the instance $P(t)$ to be proved by c_i.

Preservation of λ-terms implies that also for the ordinary λ-calculus, the three notions of α-equivalence are equivalent (in the way we have formalised them), yielding as far as we know the first formal such results, e.g. of transitivity and decidability (only assuming decidability of equality of names).

Remark 3.3.2 *Proving the three definitions of α-equivalence to be equivalent served mainly as sanity check for our extension of α-equivalence from λ- to ʎ-calculus. We do not (cl)aim to have covered all definitions of α-equivalence in the literature, see e.g. [68], but, based on the above, we strongly believe that the notion we have captured is the proper one. During proof development, (the generalisation of) Kahrs' definition was by far the easiest to work with, because of it being defined inductively. Note that his definition 'works' directly for the infinitary ʎ-calculus as well (defined, say, analogously to Chapter 12 of [67]).*

3.4 β

We extend β-reduction to ʎ-terms, and show it to be confluent without renaming. Confluence of β-reduction up to α-equivalence is obtained as a corollary, by defining suitable projections and liftings of their respective reductions.

3.4.1 $\lambda\beta$

In Chapter 3 of [4], the binary relation $\rightarrow_\beta$ on Λ is defined as the compatible closure of the notion of reduction $\beta = \{((\lambda x.M)N, M[x{:=}N]) | M, N \in \Lambda\}$. The substitution $M[x{:=}N]$ in the right-hand side of β is the naive one, i.e. up to α-congruence which is denoted by $\equiv_\alpha$. The naive approach is in turn justified by showing α-congruence to be a congruence for Curry's definition of substitution:

> Let M, $N \in \Lambda$. Then $M[x{:=}N]$ is defined inductively as follows (even if the variable convention is not observed).

M	$M[x{:=}N]$
x	N
$y \not\equiv x$	y
$M_1 M_2$	$M_1[x{:=}N]M_2[x{:=}N]$
$\lambda x.M_1$	$\lambda x.M_1$
$\lambda y.M_1$, $y \not\equiv x$	$\lambda z.M_1[y{:=}z][x{:=}N]$ where $z \equiv y$ if $x \notin \mathrm{FV}(M_1)$ or $y \notin \mathrm{FV}(N)$, else z is the first variable in the sequence $v_0, v_1, v_2, \ldots$ not in M_1 or N.

Our notion of substitution on Λ differs from Curry's in several ways.[7]

The first difference is 'under the hood'. Curry's definition is *not* a recursive one (to Coq) because of its final clause. Instead, we base our recursive definition on the skeleton $[M]_{\mathsf{skel}}$ (Definition 3.3.4).

The second difference is more important and serves to 'make α-congruence explicit'. The point is that the last clause in Curry's definition of substitution

[7] Apart from that we do not assume variables to be ordered, as mentioned above.

is neither perspicuous nor technically convenient. On the one hand, it encodes several cases at once relying on the 'coding trick' that $M[y{:=}y]$ equals M, in case $x \notin \mathrm{FV}(M_1)$ or $y \notin \mathrm{FV}(N)$. On the other hand, renaming of bound variables is not incorporated in a modular way. Our definition addresses both issues by performing renaming first on $\lambda y.M_1$ in case there is the threat of confusion of variables. The definition is recursive (to Coq) if one decrees 'threat of confusion of variables' larger than 'no confusion'.

Definition 3.4.1 *Substitution on λ-terms is defined as above, except for the clauses of λ-abstraction, which are to be replaced by:*

$\lambda y.M_1$	$\lambda y.M_1[x{:=}N]$ *if $x \neq y$ and $y \notin FV(N)$*
$\lambda y.M_1$	$(\lambda z.M_1[y{:=}z])[x{:=}N]$ *otherwise, with z such that $\lambda y.M_1 =_\alpha \lambda z.M_1[y{:=}z]$, $x \neq z$, and $z \notin FV(N)$.*

Despite the apparent differences, this definition is seen (proven) to be more liberal than Curry's (it does not need the variables to be linearly ordered).

Remark 3.4.1 *The proviso that names are ordered linearly only serves to make Curry's definition definite. However, the definiteness assumption doesn't cause β-reduction to be definite. That is, different β-reductions possibly result in α-different normal forms, as in the following example from [36]; abbreviate $M = (\lambda x.(\lambda y.\lambda x.xy)x)y$:*

$$M \to_\beta (\lambda x.\lambda z.zx)y \to_\beta \lambda z.zy$$
$$M \to_\beta (\lambda y.\lambda x.xy)y \to_\beta \lambda x.xy$$

Thus, the definiteness of Curry's definition still gives an arbitrary choice. Even stronger, as no definite α-renaming scheme would cause β-reduction to be definite, we don't assume definiteness. Of course, for implementation purposes often a choice function *is needed.*

3.4.2 ƛβ

We present the definition of β-reduction and the salient points of its proof of confluence. Compared to the ordinary λ-calculus, the β-rule must now take care of an arbitrary number of ƛs which are 'inbetween' the application and the abstraction. In such cases, the scopes of the ƛs are *extruded* in a minimal way so as to contain the scope of the abstraction, after which β-reduction proceeds as usual (see Figure 3.3, where it is irrelevant where scopes are in N). In order to perform all these operations in one go, our notion of substitution as employed by β-reduction has three arguments, of which the second corresponds to the usual one.

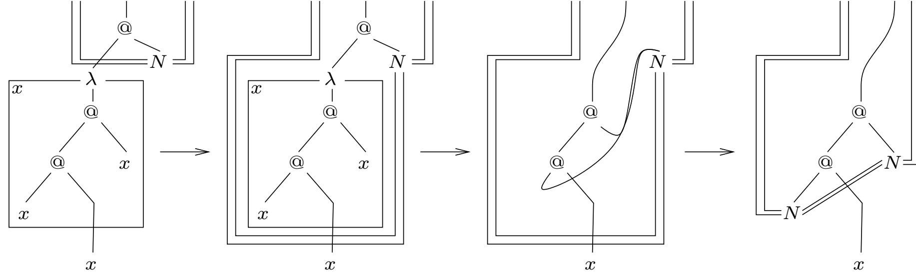

Figure 3.3: β-reduction: scope extrusion, rewiring and x-box removal, and replication.

Definition 3.4.2 *The β-rule is $(\reflectbox{\lambda}X.\lambda x.M)N \to M[X, x{:=}N, \Box]$. The relation $\to_\beta$ is the compatible closure of the β-rule:*

$$\frac{(M,N) \in \beta}{M \to_\beta N} \qquad \frac{M \to_\beta N}{\lambda x.M \to_\beta \lambda x.N} \qquad \frac{M \to_\beta N}{\reflectbox{\lambda} x.M \to_\beta \reflectbox{\lambda} x.N}$$

$$\frac{M \to_\beta M'}{MN \to_\beta M'N} \qquad \frac{N \to_\beta N'}{MN \to_\beta MN'}$$

The third argument of substitution, which initially is the empty stack, serves to determine whether an occurrence of x in M matches with the x to be substituted for. In particular, during substitution this stack is pushed upon when encountering an abstraction, and popped from when meeting an end-of-scope:

Definition 3.4.3 Substitution *$M[X, x{:=}N, Y]$ is defined by:*

$$\begin{array}{rcll}
y[X, x{:=}N, Y] & = & y, \text{ if } y \in Y & (1)\\
y[X, x{:=}N, Y] & = & \reflectbox{\lambda}Y.N, \text{ if } y \notin Y,\ x = y & (2)\\
y[X, x{:=}N, Y] & = & \reflectbox{\lambda}Y.\reflectbox{\lambda}X.y, \text{ if } y \notin Y,\ x \neq y & (3)\\
(\lambda y.M)[X, x{:=}N, Y] & = & \lambda y.M[X, x{:=}N, yY] & (4)\\
(\reflectbox{\lambda}y.M)[X, x{:=}N, JyY'] & = & \reflectbox{\lambda}y.M[X, x{:=}N, Y'], \text{ if } y \notin J & (5)\\
(\reflectbox{\lambda}y.M)[X, x{:=}N, Y] & = & \reflectbox{\lambda}Y.\reflectbox{\lambda}X.M, \text{ if } y \notin Y,\ x = y & (6)\\
(\reflectbox{\lambda}y.M)[X, x{:=}N, Y] & = & \reflectbox{\lambda}Y.\reflectbox{\lambda}X.\reflectbox{\lambda}y.M, \text{ if } y \notin Y,\ x \neq y & (7)\\
(M_1M_2)[X, x{:=}N, Y] & = & M_1[X, x{:=}N, Y]M_2[X, x{:=}N, Y] & (8)
\end{array}$$

Capture of free variables in the argument N is avoided by closing all open scopes Y, as expressed in clause (2). In case $\reflectbox{\lambda}X$ is to be put, the open scopes Y have to be closed first $(3, 6, 7)$, as will be explained below. Important clauses are (6) and (7), which explain the end-of-scope. Basically they say that if we have reached an end-of-scope, which matches (6) or jumps (7) the variable x to be subtituted for, then we can just throw the argument N away; this is safe since we know that x does not occur free in M.

To explain clauses $(5, 6, 7)$, consider an initial call:

$$(\reflectbox{\lambda}y.M)[X, x{:=}N, \Box]$$

Firstly, note that jumps *within* the body M remain untouched. To see this, imagine a recursive call:

$$(\reflectbox{\ensuremath{\lambda}}y.M')[X, x{:=}N, JyY']$$

on a subterm $\reflectbox{\ensuremath{\lambda}}y.M'$ of M, and suppose $y \notin J$. According to our jump semantics, $\reflectbox{\ensuremath{\lambda}}y$ implicitly ends the scopes J; this implicitness is kept in the resulting $\reflectbox{\ensuremath{\lambda}}y.M'[X, x{:=}N, Y']$ (clause (5)).

Secondly, in order for minimal scope extrusion of the X (originating from a β-redex) to be safe, the opened scopes J in $(\reflectbox{\ensuremath{\lambda}}y.M')[X, x{:=}N, J]$ have to be closed explicitly in case $y \notin J$, i.e. in case $\reflectbox{\ensuremath{\lambda}}y$ ends the scope of some λy *outside* the body M. In that case, jump semantics prescribes that the scope of the substitution variable x is closed either explicitly ($x = y$, (6)) or implicitly ($x \neq y$, (7)) by $\reflectbox{\ensuremath{\lambda}}y$ and we want to put the $\reflectbox{\ensuremath{\lambda}}X$. Now if we do not put the $\reflectbox{\ensuremath{\lambda}}J$ first, there is the threat that X *explicitly* closes J, which is unintended. This point is demonstrated by the following example:

Example 3.4.1 *Consider the term $P = (\reflectbox{\ensuremath{\lambda}}z.\lambda y.(\lambda x.\reflectbox{\ensuremath{\lambda}}y.M)N)L$ and note that both scopes of the abstractions λx and λy are closed in front of M, as $\reflectbox{\ensuremath{\lambda}}y$ implicitly closes λx. Therefore both N and L vanish when reducing P, as shown by the following two reduction paths from P. Substitutions are written out explicitly; the numbers stacked above '$=$' refer to the clauses in Definition 3.4.3. Assume $x \neq y$. The sequence starting with contraction of the inner redex of P runs as follows:*

$$\begin{array}{lll} P & \to_\beta & (\reflectbox{\ensuremath{\lambda}}z.\lambda y.(\reflectbox{\ensuremath{\lambda}}y.M)[\Box, x{:=}N, \Box])L \\ & \stackrel{7}{=} & (\reflectbox{\ensuremath{\lambda}}z.\lambda y.\reflectbox{\ensuremath{\lambda}}y.M)L \\ & \to_\beta & (\reflectbox{\ensuremath{\lambda}}y.M)[z, y{:=}L, \Box] \\ & \stackrel{6}{=} & \reflectbox{\ensuremath{\lambda}}z.M \end{array}$$

The sequence starting with contraction of the outer redex of P runs as follows:

$$\begin{array}{lll} P & \to_\beta & ((\lambda x.\reflectbox{\ensuremath{\lambda}}y.M)N)[z, y{:=}L, \Box] \\ & \stackrel{8,4}{=} & (\lambda x.\underline{(\reflectbox{\ensuremath{\lambda}}y.M)[z, y{:=}L, x]})N[z, y{:=}L, \Box] = P' \\ & \stackrel{6}{=} & (\lambda x.\underline{\reflectbox{\ensuremath{\lambda}}x.\reflectbox{\ensuremath{\lambda}}z.M})N[z, y{:=}L, \Box] \\ & \to_\beta & (\reflectbox{\ensuremath{\lambda}}x.\reflectbox{\ensuremath{\lambda}}z.M)[\Box, x{:=}N[z, y{:=}L, \Box], \Box] \\ & \stackrel{6}{=} & \reflectbox{\ensuremath{\lambda}}z.M \end{array}$$

Focus on the underlined substitution in the second sequence:

$$(\reflectbox{\ensuremath{\lambda}}y.M)[z, y{:=}L, x] = \reflectbox{\ensuremath{\lambda}}x.\reflectbox{\ensuremath{\lambda}}z.M$$

and note that it would be wrong to forget that $\reflectbox{\ensuremath{\lambda}}y$ implicitly closes the λx in front, and put $(\reflectbox{\ensuremath{\lambda}}y.M)[z, y{:=}L, x] \stackrel{\text{wrong!}}{=} \reflectbox{\ensuremath{\lambda}}z.M$. That this is wrong shows up if,

by coincidence, $x = z$. We would then get:

$$\begin{array}{rcl} P' & \overset{\text{wrong!}}{=} & (\lambda x.\reflectbox{λ} z.M)N[z, y{:=}L, \square] \\ & \to_\beta & (\reflectbox{λ} z.M)[\square, x{:=}N[z, y{:=}L, \square], \square] \\ & \overset{6}{=} & M \,(\textit{assuming } x = z) \end{array}$$

and confluence would be broken: $\reflectbox{$\lambda$} z.M \neq M$.

Remark 3.4.2 *The defining clauses $(5, 6, 7)$ for $(\reflectbox{$\lambda$} y.M)[X, x{:=}N, Y]$ are exhaustive, because if $y \in Y$, then $Y = JyY'$ for some J, Y' such that $y \notin J$. The implementation actually uses a nested recursion on Y with an auxiliary stack J (initially empty) recording the opening scopes in Y that are jumped (and closed) by $\reflectbox{$\lambda$} y$ (thus, the invariant is: $y \notin J$). Define $(\reflectbox{$\lambda$} y.M)[X, x{:=}N, Y] = (\reflectbox{λ} y.M)[X, x{:=}N, Y](\square)$, where $(\reflectbox{$\lambda$} y.M)[X, x{:=}N, Y](J)$ is defined as follows.*

$$\begin{array}{rcll} (\reflectbox{λ} y.M)[X, x{:=}N, \square](J) & = & \reflectbox{λ} J.\reflectbox{λ} X.M, \textit{ if } x = y & (J_1) \\ (\reflectbox{λ} y.M)[X, x{:=}N, \square](J) & = & \reflectbox{λ} J.\reflectbox{λ} X.\reflectbox{λ} y.M, \textit{ if } x \neq y & (J_2) \\ (\reflectbox{λ} y.M)[X, x{:=}N, zY](J) & = & \reflectbox{λ} y.M[X, x{:=}N, Y], \textit{ if } y = z & (J_3) \\ (\reflectbox{λ} y.M)[X, x{:=}N, zY](J) & = & (\reflectbox{λ} y.M)[X, x{:=}N, Y](Jz), \textit{ if } y \neq z & (J_4) \end{array}$$

Note that z is inserted at the bottom of J in clause (J_4), to maintain the original order of scopes. The clauses for $\reflectbox{$\lambda$} y.M$ in Definition 3.4.3 are derived from these J-clauses.

Confluence of $\reflectbox{$\lambda$}$-calculus

We discuss our formalised proof of confluence of $\to_\beta$. Our proof strategy is the usual Tait and Martin-Löf proof [4], hence is essentially based on the so-called substitution lemma on page 27 of [4]:

> 2.1.16. SUBSTITUTION LEMMA. If $x \not\equiv y$ and $x \notin \mathrm{FV}(L)$, then
>
> $$M[\,x{:=}N][\,y{:=}L] \equiv M[\,y{:=}L]\Big[\,x{:=}N[\,y{:=}L]\Big]$$

which arises when computing the critical pair for the λ-term $(\lambda y.(\lambda x.M)N)L$. Interestingly, in our case the substitution lemma splits into three lemmas: the *closed* substitution lemmas arise when the scope of y is ended (either explicitly or implicitly) in front of the λx; the *open* substitution lemma is the usual one, enriched with scoping information. We will comment on this below. Otherwise, the proof is entirely standard, (inductively) introducing multi-steps, proving that multi-steps have the diamond property and that β-reduction transits multi-steps.

What is interesting to note is that *no* α-conversion is needed. One might say that this is no surprise, since explicitly dealing with end-of-scopes constitutes a renaming mechanism in itself. Still, it *is* in our opinion surprising that the minimal scope-extrusion mechanism works nicely on non-balanced terms (cf. the discussion of confluence of MELL proof net reduction in [46]).

Adapting the substitution lemma to our calculus, we end up simplifying expressions of the shape $(\reflectbox{$\lambda$}X.M)[Z,x{:=}N,Y]$. First, to get an understanding of what is going on, consider the case of scope-balanced terms. Suppose that $\reflectbox{$\lambda$}X.M$ is scope-balanced under YxW. Then we know, since end-of-scopes X 'balance' (a part of) YxW, that X and Y are *overlapping*, that is, either:

- Z exceeds Y, $X = YxX'$, then $(\reflectbox{$\lambda$}X.M)[Z,x{:=}N,Y] = \reflectbox{λ}YZX'.M$; or:
- X is part of Y, $Y = XY'$, then $(\reflectbox{$\lambda$}X.M)[Z,x{:=}N,Y] = \reflectbox{λ}X.M[Z,x{:=}N,Y']$.

Jump terms demand extra care and we need a different (more general) notion of comparison between opening scopes Y and closing scopes X. Consider, once more, $(\reflectbox{$\lambda$}X.M)[Z,x{:=}N,Y]$. We distinguish three cases.

1. End-of-scopes X close more scopes than opened by Y, thus X includes the scope of the substitution variable x, which is closed either
   (a) explicitly; or
   (b) implicitly.
2. End-of-scopes X close some (possible all) of the open scopes Y.

To formalise this case distinction, we define *scope subtraction*. Subtraction $Y - X$ results in a pair of stacks, of which the first is either negative (item 1) or positive (the complementary case, item 2). The second stack of the pair computed by $Y - X$ is a stack J used only if the first stack is negative, say $-zX'$. In that case the substitution variable x is either matched (item 1a) or jumped (item 1b) by z, as will be shown below.

Definition 3.4.4 *Define $Y - X = Y -_{\Box} X$, where $X -_J Y$ is defined by the following clauses.*

$$
\begin{array}{rcll}
Y -_J \Box & = & (Y, J) & (1) \\
\Box -_J xX & = & (-xX, J) & (2) \\
yY -_J xX & = & Y -_{\Box} X, \ \textit{if } x = y & (3) \\
yY -_J xX & = & Y -_{Jy} xX, \ \textit{if } x \neq y & (4)
\end{array}
$$

The auxiliary stack J (initially empty) consists of the scopes in Y jumped by the current top of X. This can be inferred by clause (4), where y is inserted at the bottom of J, to maintain the original order of scopes. The argument J is reset to $\Box$ in case the top of X matches the top of Y (clause (3)). It's easy to see that if $Y - X$ is positive, then J is the empty stack.

Note that the idea of an auxiliary stack J is similar to the idea in Remark 3.4.2, only in a more general form to cope with $\reflectbox{$\lambda$}X$ instead of $\reflectbox{$\lambda$}y$. Indeed, clauses $(5, 6, 7)$ of Definition 3.4.3 can also be defined using scope-subtraction:

$$
(\reflectbox{λ}y.M)[Z,x{:=}N,Y] = \begin{cases} \reflectbox{λ}y.M[Z,x{:=}N,Y'], \text{ if } Y - y = (Y', \Box) \\ \reflectbox{λ}Y.\reflectbox{λ}Z.M, \text{ if } Y - y = (-y, Y),\ x = y \\ \reflectbox{λ}Y.\reflectbox{λ}Z.\reflectbox{λ}y.M, \text{ if } Y - y = (-y, Y),\ x \neq y \end{cases}
$$

To see the equivalence, note that:

- if $y \in Y$, then $Y = JyY'$ for some J, Y' such that $y \notin J$, and $JyY' - y = yY' -_J y = (Y', \Box)$;
- if $y \notin Y$, then $Y - y = (-y, Y)$.

If X and Y are overlapping, clause (4) in Definition 3.4.4 never applies (cf. the case distinction for scope-balanced terms on page 69):

Lemma 3.4.1

$$\begin{array}{rcll} Y_1Y_2 - Y_1 & = & (Y_2, \Box) & (Y \geq X, Y = Y_1Y_2, X = Y_1) \\ X_1 - X_1xX_2 & = & (-xX_2, \Box) & (Y < X, Y = X_1, X = X_1xX_2) \end{array}$$

For arbitrary stacks X, Y, we can decide whether the outcome of subtracting X from Y is positive or negative:

Lemma 3.4.2 *The result $Y - X$ of scope-subtracting X from Y is either*

$$\begin{array}{l} (Y_2, \Box), \text{ and then } Y = Y_1Y_2; \text{ or} \\ (-zX_2, J), \text{ and then } X = X_1zX_2. \end{array}$$

Proof. By appropriately loaded induction over Y. We refer to our Coq formalisation for more details.

Now we are ready to simplify expressions $(⅄X.M)[Z, x{:=}N, Y]$ based on the case distinction mentioned above.

Lemma 3.4.3 *If $Y - X_1zX_2 = (-zX_2, J)$ and $x = z$, then*

$$(⅄X_1zX_2.M)[Z, x{:=}N, Y] = ⅄X_1JZ.⅄X_2.M$$

Lemma 3.4.4 *If $Y - X_1zX_2 = (-zX_2, J)$ and $x \neq z$, then*

$$(⅄X_1zX_2.M)[Z, x{:=}N, Y] = ⅄X_1JZzX_2.M$$

Lemma 3.4.5 *If $Y_1Y_2 - X = (Y_2, \Box)$, then*

$$(⅄X.M)[Z, x{:=}N, Y_1Y_2] = ⅄X.M[Z, x{:=}N, Y_2]$$

Let us now present the substitution lemmas. We use the notation S^- to denote the reversal of a stack S, i.e. if $S = x_0 \ldots x_n$, then $S^- = x_n \ldots x_0$. In general, we want to compute the critical pair(s) from:

$$P = (⅄Z.\lambda y.\lambda Y^-.(⅄X.\lambda x.\lambda W^-.M)N)L$$

Inside-out reduction, $P \to_\beta^2 P_{\mathsf{in,out}}$, gives:

$$P_{\mathsf{in,out}} = \lambda Y^-.\lambda W^-.M[X, x{:=}N, W][Z, y{:=}L, WY]$$

First β-reducing the outer redex, gives:

$$P_{\mathsf{out}} = \lambda Y^-.\underbrace{(⅄X.\lambda x.\lambda W^-.M)[Z, y{:=}L, Y]}_{Q}N[Z, y{:=}L, Y]$$

We distinguish three cases for $P_{\mathsf{out}} \to_\beta P_{\mathsf{out,in}}$:

- $X = X_1 z X_2$, $Y - X = (-zX_2, J)$ and $y = z$, then, by Lemma 3.4.3:

  $$Q = \text{ʎ}X_1 JZX_2.\lambda x.\lambda W^-.M\text{; and we get:}$$
  $$P_{\mathsf{out,in}} = \lambda Y^-.\lambda W^-.M[X_1 JZX_2, x{:=}N[Z, y{:=}L, Y], W].$$

  Then, $P_{\mathsf{in,out}} = P_{\mathsf{out,in}}$ by Lemma 3.4.6.

- $X = X_1 z X_2$, $Y - X = (-zX_2, J)$ and $y \neq z$, then by Lemma 3.4.4:

  $$Q = \text{ʎ}X_1 JZzX_2.\lambda x.\lambda W^-.M\text{; and we get:}$$
  $$P_{\mathsf{out,in}} = \lambda Y^-.\lambda W^-.M[X_1 JZzX_2, x{:=}N[Z, y{:=}L, Y], W].$$

  Then, $P_{\mathsf{in,out}} = P_{\mathsf{out,in}}$ by Lemma 3.4.7.

- $Y = Y_1 Y_2$ and $Y - X = (Y_2, \Box)$; then, by Lemma 3.4.5:

  $$Q = \text{ʎ}X.\lambda x.\lambda W^-.M[Z, y{:=}L, WxY_2]\text{; and we get:}$$
  $$P_{\mathsf{out,in}} = \lambda Y^-.\lambda W^-.M[Z, y{:=}L, WxY_2][X, x{:=}N[Z, y{:=}L, Y_1 Y_2], W].$$

  Then, $P_{\mathsf{in,out}} = P_{\mathsf{out,in}}$ by Lemma 3.4.8.

The *closed* substitution lemmas arise when the scope of y *is* ended by some (possibly implicit) $\text{ʎ}y$ in front of the λx, e.g. in $(\lambda y.(\text{ʎ}y.\lambda x.M)N)L$.

Example 3.4.2 *As an illustration, we compute the critical pair arising from* $P = (\lambda y.(\text{ʎ}y.\lambda x.M)N)L$. *If we start with the inner redex, we get:*

$$\begin{array}{lll} P & \to_\beta & (\lambda y.M[y, x{:=}N, \Box])L \\ & \to_\beta & M[y, x{:=}N, \Box][\Box, y{:=}L, \Box] \end{array}$$

Performing the outer redex first:

$$\begin{array}{lll} P & \to_\beta & ((\text{ʎ}y.\lambda x.M)N)[\Box, y{:=}L, \Box] \\ & = & (\text{ʎ}y.\lambda x.M)[\Box, y{:=}L, \Box]N[\Box, y{:=}L, \Box] \\ & = & (\lambda x.M)N[\Box, y{:=}L, \Box] \\ & \to_\beta & M[\Box, x{:=}N[\Box, y{:=}L, \Box], \Box] \end{array}$$

Note that the substitution for y in M has disappeared from the right-hand side, corresponding to the erasing effect of the $\text{ʎ}y$ in front of it. Indeed,

$$M[y, x{:=}N, \Box][\Box, y{:=}L, \Box] = M[\Box, x{:=}N[\Box, y{:=}L, \Box], \Box]$$

follows from Lemma 3.4.6.

If the scope of the substitution variable y is ended *explicitly* by some $\text{ʎ}y$ in front of the λx, the following lemma springs up.

Lemma 3.4.6 (Closed substitution lemma (match))

$$\begin{array}{ll} M[X_1 zX_2, x{:=}N, W][Z, y{:=}L, WY] & \\ \quad = \; M[X_1 JZX_2, x{:=}N[Z, y{:=}L, Y], W], & \text{if } Y - X_1 zX_2 = (-zX_2, J),\ y = z \end{array}$$

Remark 3.4.3 *By the previous lemma and Lemma 3.4.1 we obtain:*

$$\begin{aligned} & M[X_1 y X_2, x{:=}N, W][Z, y{:=}L, W X_1] \\ = \quad & M[X_1 Z X_2, x{:=}N[Z, y{:=}L, X_1], W] \end{aligned}$$

which is applicable when proving the multi-step substitution lemma for scope-balanced terms.

If the scope of the substitution variable y is ended *implicitly* by some $\reflectbox{$\lambda$}z$ in front of the λx, the following lemma springs up.

Lemma 3.4.7 (Closed substitution lemma (jump))

$$\begin{aligned} & M[X_1 z X_2, x{:=}N, W][Z, y{:=}L, W Y] \\ = \quad & M[X_1 J Z z X_2, x{:=}N[Z, y{:=}L, Y], W], \textit{ if } Y - X_1 z X_2 = (-z X_2, J),\ y \neq z \end{aligned}$$

The *open* substitution lemma arises when the scope of y *is not* ended by some end-of-scope in front of the λx. Then we obtain the usual substitution lemma, appropriately enriched with scoping information.

Lemma 3.4.8 (Open substitution lemma) *If* $Y_1 Y_2 - X = (Y_2, \Box)$*, then*

$$\begin{aligned} & M[X, x{:=}N, W][Z, y{:=}L, W Y_1 Y_2] \\ = \quad & M[Z, y{:=}L, W x Y_2][X, x{:=}N[Z, y{:=}L, Y_1 Y_2], W] \end{aligned}$$

Remark 3.4.4 *By the previous lemma and Lemma 3.4.1 we obtain:*

$$\begin{aligned} & M[Y_1, x{:=}N, W][Z, y{:=}L, W Y_1 Y_2] \\ = \quad & M[Z, y{:=}L, W x Y_2][Y_1, x{:=}N[Z, y{:=}L, Y_1 Y_2], W] \end{aligned}$$

which is applicable when proving the multi-step substitution lemma for scope-balanced terms.

We introduce *multi-steps* that contract all β-redexes in a given term simultaneously.

Definition 3.4.5 Multi-steps $\mathrel{\circ\!\!\!\!\longrightarrow}$ *are defined by:*

$$\frac{M_1 \mathrel{\circ\!\!\!\!\longrightarrow} N_1 \quad M_2 \mathrel{\circ\!\!\!\!\longrightarrow} N_2}{(\reflectbox{λ}X.\lambda x.M_1)M_2 \mathrel{\circ\!\!\!\!\longrightarrow} N_1[X, x{:=}N_2, \Box]}$$

$$\frac{}{x \mathrel{\circ\!\!\!\!\longrightarrow} x} \qquad \frac{M_1 \mathrel{\circ\!\!\!\!\longrightarrow} N_1 \quad M_2 \mathrel{\circ\!\!\!\!\longrightarrow} N_2}{M_1 M_2 \mathrel{\circ\!\!\!\!\longrightarrow} N_1 N_2}$$

$$\frac{M \mathrel{\circ\!\!\!\!\longrightarrow} N}{\lambda x.M \mathrel{\circ\!\!\!\!\longrightarrow} \lambda x.N} \qquad \frac{M \mathrel{\circ\!\!\!\!\longrightarrow} N}{\reflectbox{λ}x.M \mathrel{\circ\!\!\!\!\longrightarrow} \reflectbox{λ}x.N}$$

In a multi-step from M, multiple β-redexes in M may be contracted. In particular, β-redexes occurring in the *parallel* branches M_1 and M_2 of any application subterm $M_1 M_2$ of M, may be contracted 'simultaneously' (cf. the single 'parallel' compatibility clause for application of $\mathrel{-\!\!\circ\!\!\rightarrow}$ to the two 'sequential' compatibility clauses for application of $\rightarrow_\beta$ of Definition 3.4.2). All β-redexes occurring *nested* inside the body M_1 or argument M_2 of a redex subterm M' of M, may be contracted 'at the same time' as M' itself. (cf. the 'nested' β-redex *inference rule* of $\mathrel{-\!\!\circ\!\!\rightarrow}$ to the β-redex *axiom* of $\rightarrow_\beta$.) Finally, note that the number of β-redexes may be zero, i.e. $M \mathrel{-\!\!\circ\!\!\rightarrow} M$ for any term. As it turns out, it is enough to assume this for variables only (cf. the clause $x \mathrel{-\!\!\circ\!\!\rightarrow} x$).

Remark 3.4.5 *Note that the relation $\mathrel{-\!\!\circ\!\!\rightarrow}$ is* not *transitive. Contraction might create redexes not yet present in the starting term. e.g. we have $I^3 \mathrel{-\!\!\circ\!\!\rightarrow} I^2 \mathrel{-\!\!\circ\!\!\rightarrow} I$, but not $I^3 \mathrel{-\!\!\circ\!\!\rightarrow} I$.*

The reason for the switch from single steps to multi-steps is that the former do not have the diamond property whereas the latter do. This is because contraction of a β-redex may replicate other redexes. Hence, for a notion of reduction extending β to possess the diamond property it must be 'closed under replication'. Multi-steps are just the least extension of single steps fitting the bill. At the technical level, closure under replication corresponds to the so-called substitution lemma:

Lemma 3.4.9 (Multi-step substitution lemma)

$$M \mathrel{-\!\!\circ\!\!\rightarrow} M' \wedge N \mathrel{-\!\!\circ\!\!\rightarrow} N' \Rightarrow M[Z, y{:=}N, Y] \mathrel{-\!\!\circ\!\!\rightarrow} M'[Z, y{:=}N', Y]$$

Proof. By induction on $M \mathrel{-\!\!\circ\!\!\rightarrow} M'$. In case $M = (\curlywedge X.\lambda x.M_1)M_2$, the proof obligation is:

$$(\curlywedge X.\lambda x.M_1)[Z, y{:=}N, Y]M_2[Z, y{:=}N, Y] \mathrel{-\!\!\circ\!\!\rightarrow} M_1'[X, x{:=}M_2', \Box][Z, y{:=}N', Y]$$

where $M_1 \mathrel{-\!\!\circ\!\!\rightarrow} M_1'$, $M_2 \mathrel{-\!\!\circ\!\!\rightarrow} M_2'$ and $N \mathrel{-\!\!\circ\!\!\rightarrow} N'$. The proof proceeds distinguishing cases in a similar way as on page 70, where we computed the general form of critical pairs and application of the substitution lemmas 3.4.6, 3.4.7 and 3.4.8.

Lemma 3.4.10 (Multi-step diamond property) *Multi-steps satisfy the diamond property.*

Proof. By induction on the diverging steps. All cases are trivial, except for the so-called coherence case when the starting term is a redex $(\curlywedge X.\lambda x.M)N$, and at least one step is a β-step.

- If both are β-steps, the result follows from the induction hypothesis, using the multi-step substitution lemma (Lemma 3.4.9).
- If only one of them is a β-step, then the results are $M_1[X, x{:=}N_1, \Box]$ and $(\curlywedge X.\lambda x.M_2)N_2$ respectively, for $M \mathrel{-\!\!\circ\!\!\rightarrow} M_i$ and $N \mathrel{-\!\!\circ\!\!\rightarrow} N_i$. By the

induction hypothesis $M_i \mathrel{\circ\!\!\!\twoheadrightarrow} M'$ and $N_i \mathrel{\circ\!\!\!\twoheadrightarrow} N'$ for some M' and N'. Hence the result follows since

$$M_1[X, x{:=}N_1, \Box] \mathrel{\circ\!\!\!\twoheadrightarrow} M'[X, x{:=}N', \Box]$$

by the multi-step substitution lemma (Lemma 3.4.9), and

$$(⅄X.\lambda x.M_2)N_2 \mathrel{\circ\!\!\!\twoheadrightarrow} M'[X, x{:=}N', \Box]$$

by definition of $\mathrel{\circ\!\!\!\twoheadrightarrow}$.

Lemma 3.4.11 *$\to_\beta$ transits $\mathrel{\circ\!\!\!\twoheadrightarrow}$.*

Proof. By induction on the definitions of $\to_\beta$ and $\mathrel{\circ\!\!\!\twoheadrightarrow}$, respectively. The $\mathrel{\circ\!\!\!\twoheadrightarrow} \subseteq \to_\beta^*$ part follows from simulating inside-out developments of $\mathrel{\circ\!\!\!\twoheadrightarrow}$. For this one needs congruence of $\to_\beta^*$: if $M \to_\beta^* N$, then $C[M] \to_\beta^* C[N]$, which is proved by induction on unary contexts C.

Theorem 3.4.1 (Confluence of $\to_\beta$) *$\to_\beta$ is confluent on Λ.*

Proof. From Lemmas 3.2.2, 3.4.11 and 3.4.10.

3.4.3 α and β

We prove that α and β commute on scope-balanced terms, which is enough for present purposes. We are confident that commutation of α and β holds for *all* ⅄-terms, but leave this for future work. In particular, this would require more general formulations (using scope subtraction) of the substitution/renaming lemmas (Lemmas 3.4.16 and 3.4.17).

During the proof development, we experimented with all three α-equivalences $=_\alpha^{\mathsf{c}}$, $=_\alpha^{\mathsf{s}}$, and $=_\alpha^{\mathsf{k}}$. For the commutation lemma of $\to_\beta$ and $=_\alpha$ (Lemma 3.4.20), we took $=_\alpha^{\mathsf{s}}$ for $=_\alpha$. We first present some lemmas used in the proof of that lemma. If we rename the free occurrences of a variable x (and the matching occurrences of $⅄x$) in α-equivalent terms M and N, the results are still α-equivalent:

Lemma 3.4.12 *$M =_\alpha^Z N \Rightarrow M[x{:=}z, Y] =_\alpha^Z N[x{:=}z, Y]$ if z fresh for M, N, Z.*

Proof. By induction on $M =_\alpha^Z N$, using Lemma 3.3.2 in the abstraction case.

In renaming expressions $M[x{:=}z]$ it is safe to replace z by a z' just as fresh:

Lemma 3.4.13 *$M[x{:=}z] =_\alpha^Z N[y{:=}z] \Rightarrow M[x{:=}z'] =_\alpha^Z N[y{:=}z']$, if z fresh for M, N and z' fresh for M, N, Z.*

Proof. Let $z \neq z'$ (the statement trivially holds if $z = z'$), then $z' \notin M[x{:=}z, Y]$ and $z' \notin N[y{:=}z, Y]$. By the previous lemma we obtain

$$M[x{:=}z, Y][z{:=}z', Y] =_\alpha^Z N[y{:=}z, Y][z{:=}z', Y]$$

Conclude by rewriting Lemma 3.3.3 twice.

The relation $=_\alpha^Z$ depends on the 'freshness' of Z and on Z being sufficiently long only:

Lemma 3.4.14 *If $|Z_2| \geq |Z_1|$, Z_2 fresh for M, N, Z_1, and all elements of Z_2 are distinct, then:* $M =_\alpha^{Z_1} N \Rightarrow M =_\alpha^{Z_2} N$.

Proof. By induction over $M =_\alpha^{Z_1} N$ and Lemma 3.4.13.

Lemma 3.4.14 solves the difficulty of proving transitivity of $=_\alpha^{\mathsf{s}}$ mentioned on page 57.

Lemma 3.4.15 *The relation $=_\alpha^{\mathsf{s}}$ is transitive.*

Proof. First prove that, for given Z, $=_\alpha^{Z}$ is transitive (*). Then, given $M =_\alpha^{Z_1} N =_\alpha^{Z_2} P$, choose Z_3 of length $\mathsf{max}(|Z_1|, |Z_2|)$ fresh for Z_1, Z_2, M, N, P. Then, by Lemma 3.4.14 we obtain $M =_\alpha^{Z_3} N =_\alpha^{Z_3} P$. Finally $M =_\alpha^{Z_3} P$ follows from (*).

Next, we present the substitution/renaming lemmas:

Lemma 3.4.16 (Open substitution/renaming lemma)

$$\begin{aligned} & M[X_1, y{:=}N, X_0][x{:=}z, X_0X_1X_2] \\ = \quad & M[x{:=}z, X_0yX_2][X_1, y{:=}N[x{:=}z, X_1X_2], X_0] \end{aligned}$$

if z fresh for y, M, X_0.

Lemma 3.4.17 (Closed substitution/renaming lemma)

$$\begin{aligned} & M[X_1xX_2, y{:=}N, X_0][x{:=}z, X_0X_1] \\ = \quad & M[X_1zX_2, y{:=}N[x{:=}z, X_1], X_0] \end{aligned}$$

Note that, for balanced terms, as we have that $M[x{:=}y, X] = M[y, x{:=}y, X]$, Lemmas 3.4.16 and 3.4.17 follow from Lemmas 3.4.6 and 3.4.8.

It is safe to rename in β-reductions:

Lemma 3.4.18 *If $z \notin M$ and M is scope-balanced under YxW, then $M \to_\beta N$ implies $M[x{:=}z, Y] \to_\beta N[x{:=}z, Y]$.*

Proof. Consider the case $(\curlywedge X.\lambda x.M_1)M_2 \to_\beta M_1[X, x{:=}M_2, \Box]$. Because of the assumption $\langle YxW\rangle(\curlywedge X.\lambda y.M_1)$, either X exceeds Y, and then the closed substitution/renaming lemma applies, or X is part of Y, and then the open substitution/renaming lemma applies.

If $\lambda x.M$ and $\lambda y.M'$ are α-equivalent (so also the outer (x- resp. y-)boxes have the same shape) and N and N' are α-equivalent, then the β-contractum of $(\curlywedge X.\lambda x.M)N$ is α-equivalent to the β-contractum of $(\curlywedge X.\lambda y.M')N'$:

Lemma 3.4.19

$$\begin{aligned} & M[x{:=}z, Y] =_\alpha^{Z_1} M'[y{:=}z, Y] \wedge N =_\alpha^{Z_2} N' \\ \Rightarrow \quad & \exists Z_3.M[X, x{:=}N, Y] =_\alpha^{Z_3} M'[X, y{:=}N', Y] \end{aligned}$$

if Z_1 fresh for X, Y, Z_2, M, M', N, N', and z fresh for M, M'.

On scope-balanced terms, β-reduction and α-equivalence commute (Schema E in Figure 3.4). Here, we take $=_\alpha^{\mathsf{s}}$ for $=_\alpha$.

Lemma 3.4.20 *If $\langle X\rangle M$, $M \to_\beta N$, and $M =_\alpha^Z M'$, then there exists a term N' and a stack Z' such that $M' \to_\beta N'$ and $N =_\alpha^{Z'} N'$.*

Proof. By induction on the skeleton of M. We show some interesting cases. Recall that renaming doesn't alter the skeleton of a term and that scope-balancedness is closed under $=_\alpha$.

- Case $\lambda x.M_0 \to_\beta \lambda x.N_0$. By inverting $\lambda x.M_0 =_\alpha^Z M'$, we obtain $M' = \lambda y.M_0'$ and $Z = zZ_0$ such that $M_0[x{:=}z] =_\alpha^{Z_0} M_0'[y{:=}z]$ for $z \notin M_0, M_0', Z_0$. By Lemma 3.4.18, we have $M_0[x{:=}z] \to_\beta N_0[x{:=}z]$. Then, by the induction hypothesis, there exist P and Z_0' such that $M_0'[y{:=}z] \to_\beta P$ and $N_0[x{:=}z] =_\alpha^{Z_0'} P$. We are able prove that, for some N_0', $P = N_0'[y{:=}z]$ and $M_0' \to_\beta N_0'$. Choose z' fresh for N_0, N_0', Z_0' (the Fresh variable axiom (Axiom 3.2) guarantees the existence of z'). From Lemma 3.4.13, we obtain $N_0[x{:=}z'] =_\alpha^{Z_0'} N_0'[x{:=}z']$ ($\to_\beta$ doesn't introduce new names, therefore $z \notin N_0, N_0'$ follows from $z \notin M_0, M_0'$). Then, $N' = \lambda y.N_0'$ and $Z' = z'Z_0'$ witness the goal $\exists N'.\exists Z'.\lambda y.M_0' \to_\beta N' \wedge \lambda x.N_0 =_\alpha^{Z'} N'$.

- Case $M_1M_2 \to_\beta N_1M_2$. So $M_1 \to_\beta N_1$, $M' = M_1'M_2'$, $M_1 =_\alpha^{Z_1} M_1'$, and $M_2 =_\alpha^{Z_1} M_2'$. By the induction hypothesis, there exist N_1' and Z_2 such that $M_1' \to_\beta N_1'$ and $N_1 =_\alpha^{Z_2} N_1'$. By Lemma 3.4.14, we obtain $\exists Z'.M_1'M_2' =_\alpha^{Z'} N_1'M_2'$.

- Case $(⋏X.\lambda x.M_1)M_2 \to_\beta M_1[X, x{:=}M_2, \Box]$. Derive $M' = (⋏X.\lambda y.M_1')M_2'$, $M_1[x{:=}z] =_\alpha^Z M_1'[y{:=}z]$, and $M_2 =_\alpha^{zZ} M_2'$ with $z \notin M_1, M_2, Z$. Choose Z_0 fresh for $M_1, M_2, M_1', M_2', X, Z$ and such that $|Z_0| = |Z|$ (apply the Fresh variable axiom (Axiom 3.2) $|Z_0|$ times). Then Z_0 is provably fresh for $M_1[x{:=}z]$ and $M_1'[y{:=}z]$ as well. By Lemma 3.4.14 we obtain $M_1[x{:=}z] =_\alpha^{Z_0} M_1'[y{:=}z]$ Take $N' = M_1'[X, y{:=}M_2', \Box]$ as witness for the existential statement we are proving. Finally, by Lemma 3.4.19, there exists Z' such that $M_1[X, x{:=}M_2, \Box] =_\alpha^{Z'} M_1'[X, y{:=}M_2', \Box]$.

3.4.4 Confluence of λ-calculus

As a corollary we obtain confluence of the ordinary λ-calculus (see Figure 3.4). The exposition of the proof proceeds in a top-down fashion, forward referring to lifting and projection lemmas. We use $\to_{\lambda\beta}$ and $\to_{⋏\beta}$ to distinguish β-reduction in the λ-calculus from β-reduction in the ⋏-calculus, respectively.

Theorem 3.4.2 *$\to_{\lambda\beta}$ is confluent up to $=_\alpha$.*

Proof.

1. Consider two diverging $\lambda\beta$-reductions $M \to_{\lambda\beta}^* N$ and $M \to_{\lambda\beta}^* P$.

2. Lift these stepwise to diverging $\reflectbox{$\lambda$}\beta$-reductions $M \to^*_{\reflectbox{$\lambda$}\beta} N'$ and $M \to^*_{\reflectbox{$\lambda$}\beta} P'$ (Lemma 3.4.27). (Note that M being a λ-term, it is a scope-balanced $\reflectbox{$\lambda$}$-term.)
3. By confluence of $\reflectbox{$\lambda$}\beta$-reduction, we can find some $\reflectbox{$\lambda$}$-term Q' such that $N' \to^*_{\reflectbox{$\lambda$}\beta} Q'$, $P' \to^*_{\reflectbox{$\lambda$}\beta} Q'$ (Theorem 3.4.1).
4. Projecting $N' \to^*_{\reflectbox{$\lambda$}\beta} Q'$ and $P' \to^*_{\reflectbox{$\lambda$}\beta} Q'$ back to $\lambda\beta$-reduction yields $N \to^*_{\lambda\beta} Q_1$ and $P \to^*_{\lambda\beta} Q_2$ (Lemma 3.4.28), for some α-equivalent λ-terms Q_1 and Q_2 (Corollary 3.4.1), establishing the desired confluence of $\lambda\beta$ up to α-equivalence.

Remark 3.4.6 *As far as we know the only formalised proof of confluence of β-reduction modulo α, in our setting, i.e. with a single variable space is [68]. However, their proof technique is entirely different, uniquely renaming all variables, before performing β-steps, whereas our schema, which works via the $\reflectbox{$\lambda$}$-calculus, only performs the necessary updates (in the sense of [25]).*

Lifting and Projection of β-reduction

Projection of $\reflectbox{$\lambda$}$-terms to λ-terms is the composition of first performing an α-equivalence step followed by a so-called ω-step removing all $\reflectbox{$\lambda$}$s in one go.[8] For instance, no ω-step is possible from $\lambda x.\reflectbox{$\lambda$}x.x$ since removing $\reflectbox{$\lambda$}x$ would turn the free variable x into a bound variable in $\lambda x.x$. Obviously, uniquely renaming all variables would guarantee that an ω-step could be performed. However, we rename only if necessary.

Definition 3.4.6 *We define $M \mathrel{-\!\!\bullet\!\!\!\to}_\omega N$, if $\langle\Box\rangle M \mathrel{-\!\!\bullet\!\!\!\to}_\omega N$, where $\langle X\rangle M \mathrel{-\!\!\bullet\!\!\!\to}_\omega N$ is defined by the following clauses.*

$$\frac{}{\langle X\rangle x \mathrel{-\!\!\bullet\!\!\!\to}_\omega x} \qquad \frac{\langle xX\rangle M \mathrel{-\!\!\bullet\!\!\!\to}_\omega M'}{\langle X\rangle \lambda x.M \mathrel{-\!\!\bullet\!\!\!\to}_\omega \lambda x.M'}$$

$$\frac{\langle X\rangle M_1 \mathrel{-\!\!\bullet\!\!\!\to}_\omega N_1 \quad \langle X\rangle M_2 \mathrel{-\!\!\bullet\!\!\!\to}_\omega N_2}{\langle X\rangle M_1 M_2 \mathrel{-\!\!\bullet\!\!\!\to}_\omega N_1 N_2} \qquad \frac{\langle X\rangle M \mathrel{-\!\!\bullet\!\!\!\to}_\omega M' \quad x \notin \mathsf{FV}(M)}{\langle xX\rangle \reflectbox{λ}x.M \mathrel{-\!\!\bullet\!\!\!\to}_\omega M'}$$

Thus, ω-steps are maximal, in the sense that one ω-step removes all $\reflectbox{$\lambda$}$s in one go. We write $\langle X\rangle M =^{\mathsf{k}}_\alpha \langle Y\rangle N \mathrel{-\!\!\bullet\!\!\!\to}_\omega P$ to abbreviate $\langle X\rangle M =^{\mathsf{k}}_\alpha \langle Y\rangle N \wedge \langle Y\rangle N \mathrel{-\!\!\bullet\!\!\!\to}_\omega P$, and, conversely, we write $\langle X\rangle M \mathrel{-\!\!\bullet\!\!\!\to}_\omega N =^{\mathsf{k}}_\alpha \langle Y\rangle P$ to abbreviate $\langle X\rangle M \mathrel{-\!\!\bullet\!\!\!\to}_\omega N \wedge \langle X\rangle N =^{\mathsf{k}}_\alpha \langle Y\rangle P$. Note that the source of the $\mathrel{-\!\!\bullet\!\!\!\to}_\omega$ is, by definition, forced to be scope-balanced:

$$\langle X\rangle M \mathrel{-\!\!\bullet\!\!\!\to}_\omega N \Rightarrow \langle X\rangle M$$

Also note that $\mathrel{-\!\!\bullet\!\!\!\to}_\omega$ doesn't alter the set of free variables:

$$\langle XY\rangle M \mathrel{-\!\!\bullet\!\!\!\to}_\omega M' \Rightarrow \mathsf{FV}(M, X) = \mathsf{FV}(M', X)$$

[8] ω could be decomposed itself by first pushing $\reflectbox{$\lambda$}$s to the variables, i.e. performing *maximal* scope extrusion before omitting $\reflectbox{$\lambda$}$s.

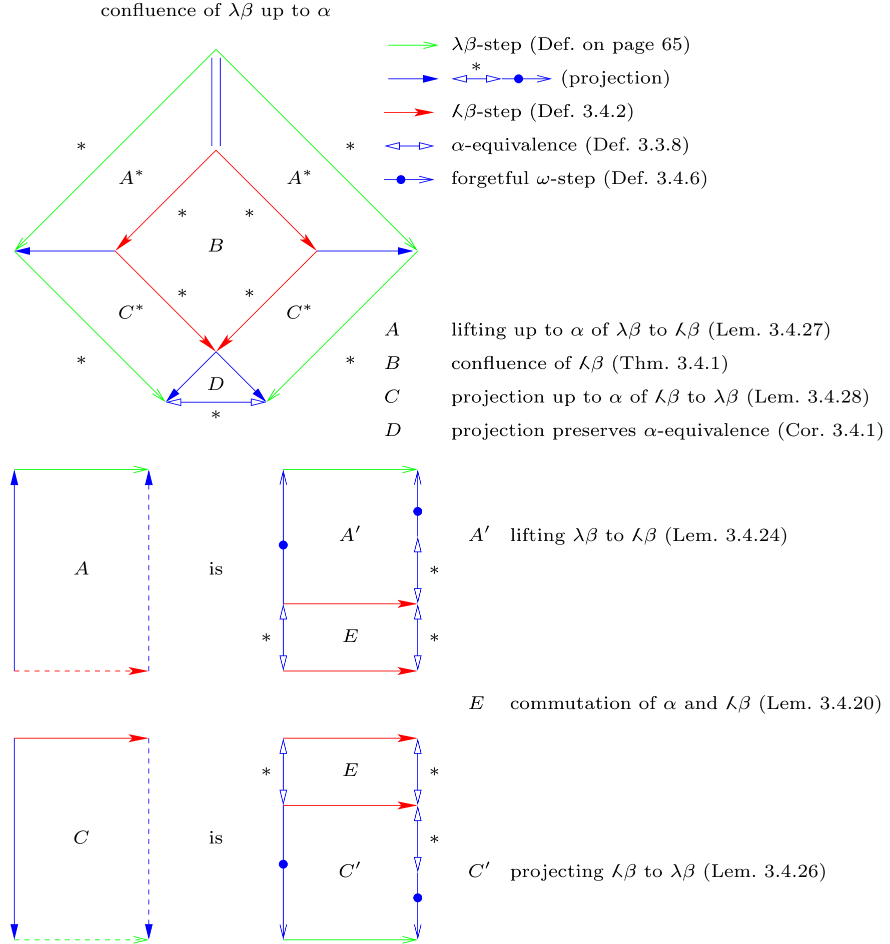

Figure 3.4: Confluence of ƛ-calculus implies confluence of λ-calculus.

Example 3.4.3 *If we first rename the bound xs in $\lambda x.\reflectbox{$\lambda$} x.x$, by some $x' \neq x$, then it is safe to forget the $\reflectbox{$\lambda$} x'$: $\langle x\rangle\lambda x.\reflectbox{λ} x.x =^{\mathsf{k}}_{\alpha} \langle x\rangle\lambda x'.\reflectbox{λ} x'.x \mathrel{-\!\!\bullet\!\!\to}_{\omega} \lambda x'.x$.*

Example 3.4.4 *It is incorrect to forget end-of-scopes in the jump calculus, as witnessed by the $\reflectbox{$\lambda$}$-term $\lambda x.\lambda y.\reflectbox{λ} x.y$. The variable y is free in this term since $\reflectbox{$\lambda$} x$ implicitly closes the scope of y. However, forgetting this end-of-scope would yield the λ-term $\lambda x.\lambda y.y$ where y is bound. The easiest way to proceed seems to be to insert as many $\reflectbox{$\lambda$}$s as are needed to make the $\reflectbox{$\lambda$}$-term scope-balanced:*

$$
\begin{array}{rcl}
\mathsf{scb}(x, X) & = & x \\
\mathsf{scb}(\lambda x.M, X) & = & \lambda x.\mathsf{scb}(M, xX) \\
\mathsf{scb}(\reflectbox{λ} x.M, \Box) & = & \mathsf{scb}(M, \Box) \\
\mathsf{scb}(\reflectbox{λ} x.M, yX) & = & \reflectbox{λ} x.\mathsf{scb}(M, X) \ \textit{if } x = y \\
\mathsf{scb}(\reflectbox{λ} x.M, yX) & = & \reflectbox{λ} y.\mathsf{scb}(\reflectbox{λ} x.M, X) \ \textit{if } x \neq y \\
\mathsf{scb}(M_1 M_2, X) & = & \mathsf{scb}(M_1, X)\mathsf{scb}(M_2, X)
\end{array}
$$

Indeed, for all terms M and stacks X, $\mathsf{scb}(M, X)$ is scope-balanced under X. Applied to the example, we first obtain $\mathsf{scb}(\lambda x.\lambda y.\reflectbox{λ} x.y, \Box) = \lambda x.\lambda y.\reflectbox{λ} y.\reflectbox{λ} x.y$. Now we see that in order to omit the $\reflectbox{$\lambda$} y$, we have to rename it first, say to z yielding $\lambda x.\lambda z.\reflectbox{λ} z.\reflectbox{λ} x.y$. Forgetting end-of-scopes now yields the (correct) λ-term $\lambda x.\lambda z.y$.

Remark 3.4.7 *In $\lambda\beta$-reduction renamings are performed, as soon as there is a confusion threat. However, such a threat may turn out to be innocuous, as in:*

$$(\lambda y.\lambda x.(\lambda z.I)yx)x \to \lambda x'.(\lambda z.I)xx' \to \lambda x'.Ix'$$

The renaming is caused by the substitution for the variable x which is erased later anyway. On the other hand, no renaming takes place during $\reflectbox{$\lambda$}\beta$-reduction:

$$(\lambda y.\lambda x.(\lambda z.I)yx)x \to \lambda x.(\lambda z.I)(\reflectbox{λ} x.x)x \to \lambda x.Ix$$

Observe that despite the final term of this $\reflectbox{$\lambda$}\beta$-reduction being an ordinary λ-term, α-conversion is *needed to project it (see Lemma 3.4.28).*

The relation $\mathrel{-\!\!\bullet\!\!\to}_{\omega}$ preserves α-equivalence:

Lemma 3.4.21 *$N \mathrel{{}_{\omega}\!\!\leftarrow\!\!\bullet\!\!-} \langle X\rangle M =^{\mathsf{k}}_{\alpha} \langle X'\rangle M' \mathrel{-\!\!\bullet\!\!\to}_{\omega} N'$ implies $\langle X\rangle N =^{\mathsf{k}}_{\alpha} \langle X'\rangle N'$.*

Proof. By induction on the proposition $\langle X\rangle M =^{\mathsf{k}}_{\alpha} \langle X'\rangle M'$. We show the case $\langle xX\rangle\reflectbox{λ} x.M =^{\mathsf{k}}_{\alpha} \langle yX'\rangle\reflectbox{λ} y.M'$. Then, $\langle X\rangle M \mathrel{-\!\!\bullet\!\!\to}_{\omega} N$, $x \notin \mathsf{FV}(M)$, $\langle X'\rangle M' \mathrel{-\!\!\bullet\!\!\to}_{\omega} N'$, and $y \notin \mathsf{FV}(M')$. Inversion gives $\langle X\rangle M =^{\mathsf{k}}_{\alpha} \langle X'\rangle M'$. By the induction hypothesis we have $\langle X\rangle N =^{\mathsf{k}}_{\alpha} \langle X'\rangle N'$, which, by the following lemma, implies the goal, $\langle xX\rangle N =^{\mathsf{k}}_{\alpha} \langle yX'\rangle N'$, because N, N' are free of $\reflectbox{$\lambda$}$s.

Corollary 3.4.1 *Schema D in Figure 3.4, stating that $Q' =_{\alpha} Q'_1 \mathrel{-\!\!\bullet\!\!\to}_{\omega} Q_1$ and $Q' =_{\alpha} Q'_2 \mathrel{-\!\!\bullet\!\!\to}_{\omega} Q_2$ imply $Q_1 =_{\alpha} Q_2$, now easily follows: first show that $Q'_1 =_{\alpha} Q'_2$ (by symmetry and transitivity of $=_{\alpha}$) and then apply the previous lemma.*

Lemma 3.4.22 *If M is a λ-term, that is, M contains no $\curlywedge$s, and $x \notin \mathsf{FV}(M)$, then*

$$\langle xX\rangle M =^{\mathsf{k}}_{\alpha} \langle yY\rangle N \textit{ implies } \langle X\rangle M =^{\mathsf{k}}_{\alpha} \langle Y\rangle N \textit{ and } y \notin \mathsf{FV}(N)$$

Conversely, if $x \notin \mathsf{FV}(M)$ and $y \notin \mathsf{FV}(N)$, then

$$\langle X\rangle M =^{\mathsf{k}}_{\alpha} \langle Y\rangle N \textit{ implies } \langle xX\rangle M =^{\mathsf{k}}_{\alpha} \langle yY\rangle N$$

Given a sequence of $\twoheadrightarrow_{\omega}$-steps and α-steps, the $\twoheadrightarrow_{\omega}$-steps can always be postponed until the α-steps are performed:

Lemma 3.4.23 $\langle X\rangle M \twoheadrightarrow_{\omega} P =^{\mathsf{k}}_{\alpha} \langle Y\rangle N \Rightarrow \exists Q.\langle X\rangle M =^{\mathsf{k}}_{\alpha} \langle Y\rangle Q \twoheadrightarrow_{\omega} N.$

Proof. By induction on the definition of $\twoheadrightarrow_{\omega}$. Consider case $\langle xX\rangle \curlywedge x.M \twoheadrightarrow_{\omega} P =^{\mathsf{k}}_{\alpha} \langle yY\rangle N$. Then $x \notin \mathsf{FV}(M)$ and $\langle X\rangle M \twoheadrightarrow_{\omega} P$. Because $\twoheadrightarrow_{\omega}$ doesn't change the set of free variables, we have that $x \notin \mathsf{FV}(P)$. By Lemma 3.4.22, we get $\langle X\rangle P =^{\mathsf{k}}_{\alpha} \langle Y\rangle N$ and $y \notin \mathsf{FV}(N)$. By the induction hypothesis, we have Q such that $\langle M\rangle X =^{\mathsf{k}}_{\alpha} \langle Y\rangle Q \twoheadrightarrow_{\omega} N$. $y \notin \mathsf{FV}(Q)$ follows and we obtain $\langle xX\rangle \curlywedge x.M =^{\mathsf{k}}_{\alpha} \langle yY\rangle \curlywedge y.Q \twoheadrightarrow_{\omega} N$.

Both projection and lifting of reductions are performed stepwise. That is, a single $\lambda\beta$-step lifts to a single $\curlywedge\beta$-step and vice versa (not to reduction sequences, as in calculi with explicit substitutions). Lifting of $\rightarrow_{\lambda\beta}$ to $\rightarrow_{\curlywedge\beta}$ (Schema A' in Figure 3.4) is stated as follows.

Lemma 3.4.24 *If $M \rightarrow_{\lambda\beta} N$ and $\langle X\rangle M' \twoheadrightarrow_{\omega} M$, then there are N_1, N_2 such that:*

$$\langle X\rangle N_1 =^{\mathsf{k}}_{\alpha} \langle X\rangle N_2 \twoheadrightarrow_{\omega} N \textit{ and } M' \rightarrow_{\curlywedge\beta} N_1$$

Proof. As an illustration, consider the case $M = (\lambda x.M_1)M_2$, and $M' = L_1 L_2$. We have $\langle X\rangle L_1 \twoheadrightarrow_{\omega} \lambda x.M_1$ and $\langle X\rangle L_2 \twoheadrightarrow_{\omega} M_2$. By inversion, we obtain $L_1 = \curlywedge X_1.\lambda x.L'_1$, $X = X_1 X_2$, $X_1 \cap \mathsf{FV}(\lambda x.L'_1) = \emptyset$ and $\langle xX_2\rangle L'_1 \twoheadrightarrow_{\omega} M_1$. The proof obligation is

$$\exists N_1, N_2.\langle X\rangle N_1 =^{\mathsf{k}}_{\alpha} \langle X\rangle N_2 \twoheadrightarrow_{\omega} M_1[x{:=}M_2] \wedge (\curlywedge X_1.\lambda x.L'_1)L_2 \rightarrow_{\curlywedge\beta} N_1$$

Take $N_1 = L'_1[X_1, x{:=}L_2, \Box]$. The following lemma (Lemma 3.4.25) guarantees the existence of an $\curlywedge$-term P such that

$$\langle X\rangle L'_1[X_1, x{:=}L_2, \Box] =^{\mathsf{k}}_{\alpha} \langle X\rangle P[X_1, x{:=}L_2, \Box] \twoheadrightarrow_{\omega} M_1[x{:=}M_2]$$

The witnessing $N_2 = P[X_1, x{:=}L_2, \Box]$ solves our goal.

The following lemma states projection of $\curlywedge$-substitution to λ-substitution.

Lemma 3.4.25

$$\begin{array}{ll} \multicolumn{2}{l}{\langle xZ\rangle M_1 \twoheadrightarrow_{\omega} M} \\ \wedge & \langle XZ\rangle N' \twoheadrightarrow_{\omega} N \\ \wedge & X \cap \mathsf{FV}(\lambda x.M_1) = \emptyset \\ \Rightarrow & \exists P{:}\Lambda.\langle XZ\rangle M_1[X, x{:=}N', \Box] =^{\mathsf{k}}_{\alpha} \langle XZ\rangle P[X, x{:=}N', \Box] \twoheadrightarrow_{\omega} M[x{:=}N] \end{array}$$

Proof. The difficult part was to find the right induction loading:

$$
\begin{array}{ll}
\langle Y_1 x Z\rangle M_1 =^{\mathsf{k}}_{\alpha} \langle Y_2 x Z\rangle M_2 \mathrel{-\!\!\bullet\!\!\to_{\omega}} M \\
\quad \wedge \quad \langle XZ\rangle N' \mathrel{-\!\!\bullet\!\!\to_{\omega}} N \\
\quad \wedge \quad X \cap \mathsf{FV}(M_1, xY_1) = \emptyset \\
\quad \wedge \quad Y_2 \cap (\{x\} \cup \mathsf{FV}(N')) = \emptyset \\
\quad \Rightarrow \quad \exists P : \Lambda . \langle Y_1 X Z\rangle M_1[X, x{:=}N', Y_1] =^{\mathsf{k}}_{\alpha} \langle Y_2 X Z\rangle P[X, x{:=}N', Y_2] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathrel{-\!\!\bullet\!\!\to_{\omega}} M[x{:=}N]
\end{array}
$$

Once appropriately loaded, the proof is a straightforward induction over M_1, the only interesting lemma used being Lemma 3.4.23.

Projecting $\to_{\not\lambda\beta}$ to $\to_{\lambda\beta}$ (Schema C' in Figure 3.4) is stated as follows.

Lemma 3.4.26 *If $M \to_{\not\lambda\beta} N$ and $\langle X\rangle M \mathrel{-\!\!\bullet\!\!\to_{\omega}} M'$, then there are N_1, N_2 such that:*

$$\langle X\rangle N =^{\mathsf{k}}_{\alpha} \langle X\rangle N_1 \mathrel{-\!\!\bullet\!\!\to_{\omega}} N_2 \text{ and } M' \to_{\lambda\beta} N_2$$

Proof. The β-rule case calls Lemma 3.4.25 again.

Lifting of $\lambda\beta$-reduction sequences to $\not\lambda\beta$-reduction sequences (Schema A^* in Figure 3.4) is stated by the following lemma.

Lemma 3.4.27

$$
\begin{array}{l}
M_1 \to^{*}_{\lambda\beta} M_2 \wedge P_1 =^{\mathsf{k}}_{\alpha} Q_1 \mathrel{-\!\!\bullet\!\!\to_{\omega}} M_1 \\
\quad \Rightarrow \quad \exists P_2, Q_2 : \Lambda . P_2 =^{\mathsf{k}}_{\alpha} Q_2 \mathrel{-\!\!\bullet\!\!\to_{\omega}} M_2 \wedge P_1 \to^{*}_{\not\lambda\beta} P_2
\end{array}
$$

Proof. By Lemmas 3.4.24 and 3.4.20 single $\lambda\beta$-steps can be lifted to single $\not\lambda\beta$-steps (Schema A in Figure 3.4). The result for sequences follows by reflexively, transitively closing the single step case.

Projection of $\not\lambda\beta$-reduction sequences to $\lambda\beta$-reduction sequences (Schema C^* in Figure 3.4) is stated by the following lemma.

Lemma 3.4.28

$$
\begin{array}{l}
P_1 \to^{*}_{\not\lambda\beta} P_2 \wedge P_1 =^{\mathsf{k}}_{\alpha} Q_1 \mathrel{-\!\!\bullet\!\!\to_{\omega}} M_1 \\
\quad \Rightarrow \quad \exists M_2 : \Lambda . \exists Q_2 : \Lambda . P_2 =^{\mathsf{k}}_{\alpha} Q_2 \mathrel{-\!\!\bullet\!\!\to_{\omega}} M_2 \wedge M_1 \to^{*}_{\lambda\beta} M_2
\end{array}
$$

Proof. By Lemmas 3.4.26 and 3.4.20 single $\not\lambda\beta$-steps can be lifted to single $\lambda\beta$-steps (Schema C in Figure 3.4). The result for sequences follows by reflexively, transitively closing the single step case.

3.5 Applications

We think that the $\not\lambda$-calculus provides an intuitive understanding of scoping in the λ-calculus. We claim it can provide solutions to problems which are known to be hard for the λ-calculus.

Expressing free variable conditions In the λ-calculus one often has use for free variable conditions. Not only are these necessary to *express* e.g. the η-rule:

$$\lambda x.Mx \to M, \text{ if } x \notin \mathsf{FV}(M),$$

but knowing that x does not occur in the free variables of M would also *speed up* reduction of the β-redex $(\lambda x.M)N$; in that case one may simply erase N. Rather than reifying the negative concept of *a variable not occurring free in a subterm*, cf. e.g. [28], our ⅄-operator makes the positive concept of *the ending of the scope of a variable* explicit. Using it, the free-variable condition of the η-rule can be expressed in the object language as:

$$\lambda x.(⅄x.M)x \to M,$$

and the β-redex becomes $(\lambda x.⅄x.M)N$, which indeed executes more efficiently. In [25] some statistical evidence is presented that this is a frequently occurring situation, i.e. that it is worthwhile to retain scoping information when evaluating ordinary λ-terms.

In the next section we present some further evidence to the usefulness of the ⅄-calculus.

3.6 Conclusion and Discussion

The reification of scopes provides a fundamental understanding of scoping mechanisms of *named*[9] terms. Our results can be summarised as follows.

- Confluence of the ⅄-calculus *without* α-conversion.
- Scope information is retained, possibly *speeding up* β-reduction.
- Free variable conditions are expressable in the object language.
- Unintended capture of free variables in the substituens by binders in the substitution body is avoided, not by renaming the binders, but by prefixing the substituens by ⅄s ending the scope of those binders. Because the ⅄s are not pushed to the variables (we don't perform *maximal* scope extrusion, that is), the transformation of arguments during substitution is avoided; thus, they can still be subject to sharing.
- α-conversion is a decidable congruent equivalence.

Restricting to a single name, the ⅄-calculus corresponds to the λ-calculus formalised using a generalisation of De Bruijn indices, with the *shift substitution* $[\uparrow]$ (cf. ⅄) as explicit term constructor (see the paragraph on related work in Section 3.1).

[9] Names are more pleasant for human beings (e.g. for debugging purposes), and we want our pen-and-paper proofs to be formalisable in a direct way. Moreover, to be user-friendly, implementations must use names, either internally or just for parsing and printing.

By lifting β-reduction of the λ-calculus to β-reduction of the ʎ-calculus, and projecting back, we can analyse more precisely renamings performed 'on the fly' by β-reduction in the λ-calculus. Confluence (up to α) of the λ-calculus is obtained as a derived result.

We conclude this chapter by discussing two potential applications of the ʎ-calculus we are currently investigating. In fact, it were these two applications which have led us to the discovery of the calculus.

Explicit substitution calculi The first part of this work arose from trying to understand Chapter 4 of [17] on perpetuality in David and Guillaume's calculus with explicit substitutions λ_{ws}, in a named setting, cf. [26], and in an atomic way. David and Guillaume introduce the λw-calculus as:

> We avoid the counter-example to the PSN property of the λs_e-calculus by adding to the usual syntax a new constructor that we call a *label* and which represents an updating information. The term t with label k (denoted by $\langle k \rangle t$) corresponds to the term t where all free indices have been increased by k (i.e. $\phi_0^k(t)$ in λs_e).
>
> In the terms we are finally interested in, two successive labels are not allowed. We first define preterms without this restriction.
>
> **Definition 3.1.** We define the set of λ_w-preterms by the following grammar:
>
> $$t ::= \underline{n} \mid \lambda t \mid (t\ t) \mid \langle k \rangle t \quad \text{with} \quad n, k \in \mathbb{N}$$

Observing that labels can be seen as repetitions of successors should make the relationship to the ʎ-calculus clear.[10] In [26] a named version of λ_w is presented having sets of names as labels. Our approach is more elementary as their labels are added only *after* α-conversion (not in their abstract syntax, but in their calculus), so they cannot get confluence of β-reduction without it.

Application: Preservation of Strong Normalisation The λ_{ws} calculus was introduced as a calculus having, among other desirable properties, the preservation of strong normalisation (PSN) property. From [25] we understand that λ_{ws} arose in a seemingly *ad hoc* way from barring counterexamples to PSN for existing calculi with explicit substitutions. We think the ʎ-calculus offers an easy insight as to *why* the calculus works as follows.

The problem with PSN arises when one tries to orient, as a reduction rule, the critical pair arising from (an explicit version of) the substitution lemma. (see page 68). The problem with orienting the ensuing critical pair from right to left is that the resulting rule is non-left-linear (L occurs twice in its left-hand side), causing non-confluence, which is undesirable. However, orienting the critical pair from left to right is also problematic since the resulting rule is non-terminating, just by itself, since the left-hand side can be embedded into

[10] The relationship is not entirely trivial since in their λ_w calculus, David and Guilaume make use of commutativity and associativity of addition of natural numbers, whereas we are only allowed to manipulate *stacks* of names. However, one may reformulate their rules such that commutativity and associativity are not needed.

the right-hand side. (Note that this orientation corresponds to transforming from inside-out to outside-in (standard) order of contraction of the β-redexes.)

The key insight is that in the ⅄-calculus, we can recognise the fact that we are already in outside-in order: consider the substitution lemma above oriented from left to right and enriched with end-of-scope information (but for the moment forgetting the first component of ⅄-substitutions which are empty in this example):

$$M[x{:=}N,\Box][y{:=}L,\Box] \to M[y{:=}L,\underline{x}][\underline{x}{:=}N[y{:=}L,\Box],\Box]$$

Now we *recognise* that the two underlined xs in the right-hand side match with one another, hence that these substitutions are already in standard order. Forbidding further applications of the rule in such situations, should break the infinite reduction and regain PSN (roughly speaking the maximal length of a reduction can be bounded by the length of a standard reduction, in the spirit of [44]). Applying this idea to the closed and open substitution lemmas (Lemmas 3.4.6 and 3.4.8) should give rise to named versions of the following two λ_{ws} rules, see e.g. page 65 of [17]:

$$\begin{array}{llll} M[k/N,l][\![i/P,j]\!] & \to_{lc1} & M[k/N[\![i-k/P,j]\!],j+l-1] & k \le i < k+l \\ M[k/N,l][\![i/P,j]\!] & \to_{lc2} & M[\![i-l+1/P,j]\!][k/N[\![i-k/P,j]\!],l] & k+l \le i \end{array}$$

which should yield a named version of λ_{ws} having PSN.

Localising scope extrusion The second part of this work arose from trying to understand Coppola's PhD thesis [24] on the (complexity of) an optimal implementation of the λ-calculus. The idea is to view the boxes featuring in that work as our boxes, i.e. as representing scoping information, and to view their optimal implementation as a *local* implementation of our (minimal) scope extrusion.

Application: optimal reduction Lamping provided in [47] the first implementation of the λ-calculus which was optimal in the sense of Lévy [48]. His implementation was based on a translation of λ-terms to graphs having nodes (fan-in and fan-out) for both explicit sharing and unsharing. In order for sharing and unsharing nodes to match up properly (the 'oracle'), he had to introduce two further types of nodes, the control nodes (square bracket and croissant). These control nodes had an *ad hoc* justification and their definitive understanding was considered to be the main open problem of this technique according to Chapter 9 of [3].

We claim that the oracle can be understood to arise from making β-reduction in the ⅄-calculus *local* in the sense of [45]. That is scope extrusion and x-box removal as in Figure 3.3 are to be made local (replication is dealt with by the sharing nodes). A way in which this can be implemented is shown on the left in Figure 3.5. In fact, a key insight (cf. the second step of Figure 3.5) is that x-box removal is superfluous as long as scopes can always be moved out of the way (of a β-redex). We have a working optimal implementation of the λ-calculus based on rules achieving just that, such as the zheh-rule in Figure 3.5 for

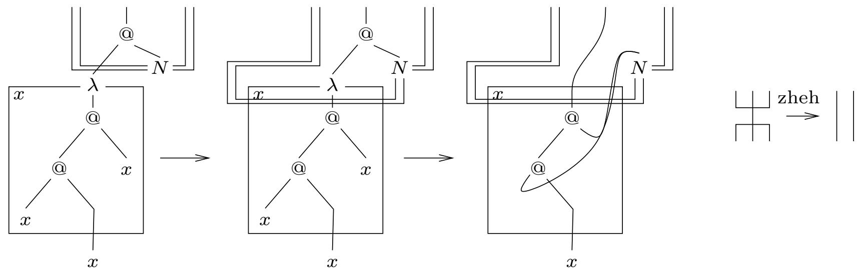

Figure 3.5: Left: β-reduction: local scope extrusion and rewiring. Right: scope fusion.

fusing two adjacent scopes. The implementation performs well on the examples in [3], without the need for either their safe nodes or heuristics (we have only one control node). E.g. computing their most complex example, (f ten) in Figure 9.23 of [3], takes us roughly 5 times as many interactions (compared to BOHM 1.1).[11]

[11]The difference might be explainable by that we do not employ compound nodes.

Bibliography

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[2] T. Altenkirch. *Constructions, inductive types and strong normalisation.* PhD thesis, Laboratory for the Foundations of Computer Science, University of Edinburgh, 1994.

[3] A. Asperti and S. Guerrini. *The Optimal Implementation of Functional Programming Languages.* Cambridge University Press, 1998.

[4] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics.* North-Holland, 1984.

[5] S. Baro and F. Maurel. The $q\nu$ and $q\nu k$ calculi : name capture and control. PPS prépublication 16, Université Denis Diderot, 2003.

[6] B. Barras. *Auto-validation d'un système de preuves avec familles inductives.* PhD thesis, Université Paris 7, November 1999.

[7] B. Barras and B. Werner. Coq in Coq. `http://pauillac.inria/~barras/coqincoq.ps.gz`, 1997.

[8] G. Barthe, M. Ruys, and H. Barendregt. A two-level approach towards lean proof-checking. In *Proceedings of Types '95*, volume 1128 of *Lecture Notes in Computer Science*, pages 16–35. Springer, 1995.

[9] Z. Benaissa, Briaud, P. D., Lescanne, and J. Rouyer-Degli. $\lambda\upsilon$, a calculus of explicit substitutions which preserves strong normalisation. *Journal of Functional Programming*, 6(5):699–722, 1996.

[10] L.S. van Benthem Jutting, J. McKinna, and R. Pollack. Checking algorithms for Pure Type Systems. In H. Barendregt and T. Nipkow, editors, *Proceedings of the International Workshop on Types for Proofs and Programs*, volume 806 of *Lecture Notes in Computer Science*, pages 19–61, Nijmegen, The Netherlands, 1994. Springer-Verlag.

[11] K.J. Berkling. A symmetric complement to the lambda-calculus. Interner Bericht ISF-76-7, GMD, D-5205, St. Augustin 1, West Germany, 1976.

[12] M. Bezem, D. Hendriks, and H. de Nivelle. Automated proof construction in type theory using resolution. `http://www.phil.uu.nl/~hendriks/coq/blinc`, 2002. Coq source, tools and sample applications.

[13] M. Bezem, D. Hendriks, and H. de Nivelle. Automated proof construction in type theory using resolution. In D. McAllester, editor, *Automated Deduction – CADE-17*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 148–163. Springer-Verlag, 2000.

[14] M. Bezem, D. Hendriks, and H. de Nivelle. Automated proof construction in type theory using resolution. *Journal of Automated Reasoning*, 29(3–4):253–275, 2002.

[15] R.S. Bird and R.A. Paterson. De Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–91, 1999.

[16] S.C.C. Blom. *Term Graph Rewriting, syntax and semantics*. PhD thesis, Vrije Universiteit Amsterdam, 2001.

[17] E. Bonelli. *Substitutions explicites et réécriture de termes*. PhD thesis, Université Paris XI, 2001.

[18] S. Boutin. Using reflection to build efficient and certified decision procedures. In M. Abadi and T. Ito, editors, *Theoretical Aspects of Computer Software*, volume 1281 of *Lecture Notes in Computer Science*, pages 515–529. Springer-Verlag, 1997.

[19] N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indagationes Mathematicae*, 34:381–392, 1972.

[20] L. Cardelli and A.D. Gordon. Mobile ambients. In M. Nivat, editor, *FOSSACS '98*, volume 1378 of *LNCS*, pages 140–155. Springer, 1998.

[21] C. Chen and H. Xi. Meta-programming through typeful code representation. `http://www.cs.bu.edu/~hwxi/`, 2002.

[22] A. Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33:346–366, 1932.

[23] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. `http://www.grappa.univ-lille3.fr/tata`, 1997. October, 1st 2002.

[24] P. Coppola. *On the Complexity of Optimal Reduction of Functional Programming Languages*. PhD thesis, Università degli Studi di Udine, 2002.

[25] R. David and B. Guillaume. A λ-calculus with explicit weakening and explicit substitution. *Mathematical Structures for Computer Science*, 11:169–206, 2001.

[26] R. Di Cosmo, D. Kesner, and E. Polonovski. Proof nets and explicit substitutions. In *FOSSACS) '00*, volume 1784 of *LNCS*, pages 63–81. Springer, 2000.

[27] M.J. Gabbay and A.M. Pitts. A new approach to abstract syntax involving binders. In *LICS '99*, pages 214–224. IEEE Computer Society, 1999.

[28] A.D. Gordon and T.F. Melham. Five axioms of alpha-conversion. In J. von Wright, J. Grundy, and J. Harrison, editors, *TPHOLs '96*, volume 1125 of *LNCS*, pages 173–190. Springer, 1996.

[29] The Omega Group. The OMEGA System. `http://www.ags.uni-sb.de/~omega`. Mathematical assistant.

[30] B. Günzel. Logik und das Auswahlaxiom. Diplomarbeit, Fakultät für Mathematik und Informatik der Ludwig-Maximilians-Universität München, 2000.

[31] D. Hendriks. Clausification of first-order formulae, representation & correctness in type theory. Master's thesis, Department of Philosophy, Utrecht University, 1998.

[32] D. Hendriks. Proof reflection in Coq. *Journal of Automated Reasoning*, 29(3):277–307, 2002.

[33] D. Hendriks. Proof reflection in Coq. `http://www.phil.uu.nl/~hendriks/coq/prfx`, 2002. Coq source.

[34] D. Hendriks and V. van Oostrom. ʎ. In F. Baader, editor, *Automated Deduction – CADE-19*, volume 2741 of *Lecture Notes in Artificial Intelligence*, pages 136–150. Springer-Verlag, 2003.

[35] D. Hendriks and V. van Oostrom. Adbmal-calculus. `http://preprints.phil.uu.nl/lgpr`, 2003. Coq source.

[36] J.R. Hindley. *The Church–Rosser property and a result in combinatory logic*. PhD thesis, University of Newcastle-upon-Tyne, 1964.

[37] M. Hollenberg and C.F.M Vermeulen. Counting variables in a dynamic setting. *Journal of Logic and Computation*, 6(5):725–744, 1996.

[38] X. Huang. Translating machine-generated resolution proofs into ND-proofs at the assertion level. In *Proceedings of PRICAI-96*, pages 399–410, 1996.

[39] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980.

[40] G. Huet. Residual theory in lambda-calculus: a formal development. *Journal of Functional Programming*, 4(3):371–394, July 1994.

[41] J. Hurd. Integrating gandalf and hol. In *Proceedings TPHOL's 99*, volume 1690 of *Lecture Notes in Computer Science*, pages 311–321, 1999.

[42] F. Joachimski and M. Matthes. Short proofs of normalization for the simply-typed λ-calculus, permutative conversions and gödel's T. *Archive for Mathematical Logic*, 42(1):59–87, 2003.

[43] S. Kahrs. Context rewriting. In M. Rusinowitch and J.-L. Rémy, editors, *CTRS '92*, volume 656 of *LNCS*, pages 21–35. Springer, 1993.

[44] Z. Khasidashvili, M. Ogawa, and V. van Oostrom. Uniform normalisation beyond orthogonality. In A. Middeldorp, editor, *RTA '01*, LNCS, pages 122–136. Springer, 2001.

[45] Y. Lafont. Interaction nets. In *POPL '90*, pages 95–108. ACM Press, 1990.

[46] Y. Lafont. From proof-nets to interaction nets. In J.-Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic*, volume 222 of *London Mathematical Society Lecture Note Series*, pages 225–248. Cambridge University Press, 1995.

[47] J. Lamping. An algorithm for optimal lambda calculus reduction. In *POPL '90*, pages 16–30. ACM Press, 1990.

[48] J.-J. Lévy. *Réductions correctes et optimales dans le λ-calcul.* Thèse de doctorat d'état, Université Paris VII, 1978.

[49] W. McCune and O. Shumsky. IVY: A preprocessor and proof checker for first-order logic. Technical Report Preprint ANL/MCS-P775-0899, Argonne National Laboratory, Argonne IL, 1999.

[50] J. McKinna and R. Pollack. Pure Type Systems formalized. In M. Bezem and J.F. Groote, editors, *Proceedings 1st International Conference on Typed Lambda Calculi and Applications, TLCA'93, Utrecht, The Netherlands, 16–18 March 1993*, volume 664, pages 289–305. Springer-Verlag, Berlin, 1993.

[51] J. McKinna and R. Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23(3–4):373–409, 1999.

[52] G. Nadathur and D. Miller. Higher-order logic programming. In D. Gabbay, C. Hogger, and J. Robinson, editors, *Handbook of Logic in Artificial Intelligence*, volume 5, pages 499–590, Oxford, 1998. Clarendon Press.

[53] Q.-H. Nguyen. Certifying term rewriting proofs in ELAN. *Electronic Notes in Theoretical Computer Science*, 59(4), 2001.

[54] H. de Nivelle. Bliksem, a first-order, resolution-based theorem prover. `http://www.mpi-sb.mpg.de/~nivelle/programs/bliksem/index.html`.

[55] J. Parrow. The fusion calculus: Expressiveness and symmetry in mobile processes. In *LICS '98*, pages 176–185. IEEE Computer Society, 1998.

[56] R.A. Paterson. Non-deterministic lambda-calculus: A core for integrated languages. In *Declarative Programming*, SassBachwalden, 1991. Springer.

[57] H. Persson. *Type Theory and the Integrated Logic of Programs*. PhD thesis, Chalmers University of Technology and University of Götenborg, 1996.

[58] F. Pfenning. Analytic and non-analytic proofs. In *Proceedings CADE 7*, volume 170 of *Lecture Notes in Computer Science*, pages 394–413. Springer-Verlag, 1984.

[59] J. van de Pol. *Termination of Higher-order Rewrite Systems*. PhD thesis, Utrecht University, Department of Philosophy, Utrecht, 1996.

[60] R. Pollack. How to believe a machine-checked proof. In G. Sambin and J. Smith, editors, *Twenty-Five Years of Constructive Type Theory*. Oxford University Press, 1998.

[61] D. Prawitz. Ideas and results in proof theory. In J.E. Fenstad, editor, *Proceedings of the Scandinavian Logic Symposium*, pages 235–307, Amsterdam, 1971. North-Holland.

[62] D.E. Schroer. *The Church–Rosser Theorem*. PhD thesis, Cornell University, 1965.

[63] H. Schwichtenberg. Logic and the axiom of choice. In M. Boffa, D. van Dalen, and K. McAloon, editors, *Logic Colloquium '78*, pages 351–356. North-Holland, 1979.

[64] J. Smith and T. Tammet. Optimized encodings of fragments of type theory in first-order logic. In *Proceedings Types '95*, volume 1158 of *Lecture Notes in Computer Science*, pages 265–287. Springer-Verlag, 1995.

[65] G. Sutcliffe. The CADE-16 ATP system competition. *Journal of Automated Reasoning*, 24:371–396, 2000.

[66] The Coq Development Team. *The Coq Proof Assistant Reference Manual, version 7.2*. INRIA-Rocquencourt, December 2001. `http://coq.inria.fr/doc-eng.html`.

[67] Terese. *Term Rewriting Systems*. Cambridge University Press, 2003.

[68] R. Vestergaard and J. Brotherston. A formalised first-order confluence proof for the λ-calculus using one-sorted variable names. In A. Middeldorp, editor, *RTA '01*, LNCS, pages 306–321. Springer, 2001.

[69] B. Werner. *Une Theorie des Constructiones Inductives*. PhD thesis, Université Paris, 1994.

Samenvatting

Hoofdstuk 1: *Automated Proof Construction in Type Theory using Resolution*

We incorporeren resolutielogica in typentheorie. Een vertaling van resolutiebewijzen naar bewijstermen in een typentheoretisch systeem levert een verificatieprocedure voor die bewijzen. Bovendien komt de kracht van automatische stellingbewijzers die zijn gebaseerd op resolutielogica beschikbaar in bewijsassistenten die zijn gebaseerd op typentheorie. Een en ander wordt geïllustreerd door de implementatie van een 'tool' die het gebruik van de stellingbewijzer Bliksem binnen de bewijsassistent Coq mogelijk maakt. Het clausificatie-algoritme is geformaliseerd en correct bewezen in Coq. Het checken van de bewijsobjecten die het resultaat zijn van de vertaling gebeurt per keer. De vertaling van resolutiebewijzen naar λ-termen is zodanig dat de representatie van resolutiestappen lineair is in de grootte van de premissen. Hiertoe definiëren we een nieuw formaat clauses in minimale logica.

Hoofdstuk 2: *Proof Reflection in* Coq

We formaliseren natuurlijke deductie voor intuïtionistische eerste-orde logica met expliciete bewijstermen in het systeem Coq. We laten zien dat ons afleidingssysteem correct is met betrekking tot de logica van Coq. Middels *reflectie* kunnen we zodoende redeneren over een deel van de meta-taal zelf. Als voorbeeld van de manipulatie van bewijzen als *objecten* definiëren we Prawitz' bewijsreductie met permutatieve conversies. De formalisatie van dit stuk theorie kan als basis dienen voor het bewijzen van meta-theoretische stellingen over de eerste-orde logica (denk bijvoorbeeld aan een syntactisch bewijs van de conservativiteit van het keuzeaxioma), alsook voor het automatiseren van stellingbewijzen.

Hoofdstuk 3: ʎ

We maken de notie van *scope* (bereik) in de λ-calculus expliciet. Daartoe breiden we de syntax van de λ-calculus uit met een operator ʎ die het afsluiten van een

scope representeert. Het idee is dat ƛx correspondeert met λx erboven (in de termboom). De noties van α-equivalentie en β-reductie worden overeenkomstig uitgebreid. We laten zien dat de resulterende ƛ-calculus confluent is zonder gebruik te maken van α-equivalentie. Confluentie van β-reductie in de λ-calculus wordt verkregen als een afgeleid resultaat, door het hernoemen van variabelen en het weglaten van ƛs. Alle bewijzen zijn geverifieerd in Coq.

Curriculum Vitae

- Geboren op 6 augustus 1973 te IJsselstein.
- 1985–1991: VWO aan het Montessori Lyceum Herman Jordan, te Zeist.
- 1993–1998: Cognitieve Kunstmatige Intelligentie, Universiteit Utrecht.
- 1998–2003: AIO aan de Faculteit der Wijsbegeerte, Universiteit Utrecht.

Sinds 1992 is Dimitri partner van Nelleke van Hienen. In 2002 werd hun zoon Ole Iwan Hendriks geboren.