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 *$\lambda\!\!\!\backslash$-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 $\lambda\!\!\!\backslash$. The idea is that an $\lambda\!\!\!\backslash x$ 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 $\mathbb{N}$.