Analyzing individual proofs as the basis of interoperability between proof systems

Gilles Dowek*

Abstract. We describe the first results of a project to analyze in which theories formal proofs can be expressed and use this analysis as the basis of interoperability between proof systems.

1 Introduction

Sciences study both individual objects and generic ones. For example, Astronomy studies both the individual planets of the Solar system: Mercury, Venus, etc. determining their radius, mass, composition, etc., but also the motion of generic planets: Kepler's laws, that do not just apply to the six planets known at the time of Kepler, but also to those that have been discovered after, and those that may be discovered in the future. Computer science studies both algorithms that apply to generic data, but also specific pieces of data. Mathematics mostly studies generic objects, but sometimes also specific ones, such as the number π or the function ζ.

Proof theory mostly studies generic proofs. For example, Gentzen's cut elimination theorem for Predicate logic applies to any proof expressed in Predicate logic. Those that were known at the time of Gentzen, those that have been constructed after, and those that will be constructed in the future. Much less effort is dedicated to studying the individual mathematical proofs, with a few exceptions, for example [29].

Considering the proofs that we have, instead of all those that we may build in some logic, sometimes changes the perspective. For example, consider a cut elimination theorem for a logic $\mathcal{L}$. The stronger the logic $\mathcal{L}$, the stronger the theorem. In contrast, consider a specific proof π, say a proof of Fermat's little theorem, and consider a theorem of the form: the proof π can be expressed in the logic $\mathcal{L}$. In this case, the weaker the logic, the stronger the theorem. So, studying generic proofs leads to focus on stronger and stronger logics, while studying individual proofs on weaker and weaker ones.

In this paper, we discuss the program of analyzing the formal proofs that have been developed in computerized proof systems such as Coq[1], Matita[2], HOL

* Inria and École normale supérieure de Paris-Saclay, 61, avenue du Président Wilson, 94235 Cachan Cedex, France, gilles.dowek@ens-paris-saclay.fr

[1] coq.inria.fr

[2] matita.cs.unibo.it

LIGHT[3], ISABELLE/HOL[4], PVS[5], FOCALIZE[6], etc. In particular, we want to be able to analyze in which logics, each of these proofs can be expressed.

Such a project in largely inspired by the reverse mathematics project [18, 34], but has some differences. First, we do not propose to classify theorems according to the logics in which they can be proved, but to classify the proofs in function of the logic in which they can be expressed. Some theorems, for example, the prime number theorem, have very different proofs, some very elementary and some very complex. Second, we focus on formal proofs, that is proofs expressed and checked in computerized proof systems. Third, our project is in some sense less ambitious, as it focuses on stronger theories than reverse mathematics. We typically, address the question of the possibility to express a proof in the Calculus of constructions [12], or in Simple type theory [11], while reverse mathematics focuses on fragments of Second-order arithmetic.

Knowing in which logic, which proof can be expressed is a fundamental question, as it is part of our understanding of these proofs.

It is also a practical one, as it opens the way to interoperability between computerized proof systems. In the domain of formal proofs, we currently have COQ proofs, HOL LIGHT proofs, etc. of various theorems. And, when we have a COQ proof of a theorem, we do not necessarily have a HOL LIGHT proof of this theorem, and vice versa. The problem to be addressed here is not just that of the translation of proofs from one logic to another, as these logics cannot express the same proofs, but also that of the analysis of the logics in which these proofs can or cannot be expressed.

2 From logics to theories

2.1 Logical Frameworks

When we have a proof expressed in a logic $\mathcal{L}$, analyzing the logics in which this proof can be expressed amounts to analyze the ingredients of the logic $\mathcal{L}$ that it uses. This requires to analyze this logic into a number of ingredients, for example into a number of axioms. Set theory, for example, is naturally analyzed into a number of axioms and axiom schemes, including, for example, the axiom of choice, and this analysis permits to classify the proofs according to the axioms they use: the proof that every vector has a unique decomposition in a given basis does not use the axiom of choice, the proof that every vector space has a basis does. These ingredients of set theory are expressed in a logical framework: Predicate logic. So, analyzing a logic amounts to express it as a theory in a logical framework.

To express set theory, it is also possible to choose another logical framework: Constructive predicate logic and add the excluded middle as an axiom scheme.

[3] `www.cl.cam.ac.uk/~jrh13/hol-light`
[4] `isabelle.in.tum.de`
[5] `pvs.csl.sri.com`
[6] `focalize.inria.fr`

Then, we can express that the proof that every vector has a unique decomposition in a given basis does not use the excluded middle, while the proof of the Bolzano-Weierstrass theorem does.

The examples of the axiom of choice and of the excluded middle show that, even in everyday mathematics, we do care about the analysis of proofs, even if this analysis is often restricted to the very specific cases of the axiom of choice and of the excluded middle.

2.2 Axioms and rewrite rules

Predicate logic is certainly the most widely used logical framework. But it also has some limitations. For example, many theories are expressed with an infinite number of axioms and eliminating axiom schemes, like in Von Neumann-Bernays-Gödel set theory, is often cumbersome [28]. Moreover, if Simple type theory can easily be expressed in Predicate logic, expressing the Calculus of constructions, the Calculus of inductive constructions, etc. is more cumbersome.

Another logical framework is that of Pure type systems [6], where it is possible to express Simple type theory, the $\lambda\Pi$-calculus, the system F, the Calculus of constructions, etc. and analyze these logics into sorts, axioms, and rules. This permits to classify the proofs expressed, for example, in the Calculus of constructions, into those that use the polymorphic rules and those that do not. But this logical framework also has limitations as the Calculus of inductive constructions cannot be expressed in it.

Another logical framework is the $\lambda\Pi$-calculus, that is the λ-calculus with dependent types [23]. Like in Predicate logic, the ingredients of a logic expressed in this framework are axioms and it is difficult to express in the $\lambda\Pi$-calculus the logics that include a computational, or definitional, equality that identifies some terms.

So, we shall use another logical framework that is a synthesis of Predicate logic, Pure type systems and the $\lambda\Pi$-calculus: the *$\lambda\Pi$-calculus modulo theory* [13], also called the *Martin-Löf logical framework* [30], where the ingredients of a logic are axioms and rewrite rules, like in Deduction modulo theory [16, 17]. Simple type theory can easily be expressed as a theory in the $\lambda\Pi$-calculus modulo theory with three rewrite rules, and the Calculus of constructions as a theory with just four rules. Moreover, the $\lambda\Pi$-calculus modulo theory has an efficient implementation: the system Dedukti [5], whose first implementation has been presented in [7] and its most recent in [33].

3 Translating proofs expressed in the Calculus of constructions into proofs in Simple type theory

We show, in this section, how to express Constructive simple type theory and the Calculus of constructions, in the $\lambda\Pi$-calculus modulo theory. We then discuss how this can be used to analyze if a proof, expressed in the Calculus of constructions, can be reformulated in Simple type theory or not, according to these expressions of these theories in the $\lambda\Pi$-calculus modulo theory.

3.1 Simple type theory as a theory in the $\lambda\Pi$-calculus modulo theory

Simple type theory can be expressed in Deduction modulo theory [15] and hence in the $\lambda\Pi$-calculus modulo theory [3, 4].

The types of Simple type theory are expressed as terms of type *type*, with three constants o, *nat*, and *arrow*. The type o is that of propositions, the type *nat* that of individuals, often written ι, and those built with the constant *arrow* are the functional types. For example, the type of Simple type theory $nat \to nat$ is expressed as the term $(arrow\ nat\ nat)$ of type *type*. Then, to each term t of type *type*, is associated a type $(\eta\ t)$ of the $\lambda\Pi$-calculus modulo theory, using a constant η of type $type \to Type$ and the rewrite rule

$$(\eta\ (arrow\ x\ y)) \longrightarrow (\eta\ x) \to (\eta\ y)$$

For example, to the term $(arrow\ nat\ nat)$, is associated the type $(\eta\ (arrow\ nat\ nat))$ that reduces to $(\eta\ nat) \to (\eta\ nat)$.

The terms of Simple type theory of type t are then expressed as terms of type $(\eta\ t)$. For example, the term $\lambda x : nat\ x$ of Simple type theory is expressed as the term $\lambda x : (\eta\ nat)\ x$ of type $(\eta\ nat) \to (\eta\ nat)$.

In particular, the propositions are expressed as terms of type $(\eta\ o)$, using two constants $\Rightarrow$ and $\forall$. For example, the proposition $\forall X : o\ (X \Rightarrow X)$ is expressed as the term $\forall\ o\ \lambda X : (\eta\ o)\ (\Rightarrow\ X\ X)$. Note that, in this expression of Simple type theory in the $\lambda\Pi$-calculus modulo theory, we do not have a quantifier $\forall_A$ for each type A, but a single quantifier $\forall$, that is applied to a term A of type *type*. Then, to each term p of type $(\eta\ o)$, is associated a type $(\varepsilon\ p)$ of the $\lambda\Pi$-calculus modulo theory, using a constant ε of type $(\eta\ o) \to Type$ and the rewrite rules

$$(\varepsilon\ (\Rightarrow\ x\ y)) \longrightarrow (\varepsilon\ x) \to (\varepsilon\ y)$$

$$(\varepsilon\ (\forall\ x\ y)) \longrightarrow \Pi z : (\eta\ x)\ (\varepsilon\ (y\ z))$$

For example, to the term $\forall\ o\ \lambda X : (\eta\ o)\ (\Rightarrow\ X\ X)$, is associated the type $(\varepsilon\ (\forall\ o\ \lambda X : (\eta\ o)\ (\Rightarrow\ X\ X)))$ that reduces to $\Pi X : (\eta\ o)\ ((\varepsilon\ X) \to (\varepsilon\ X))$.

Finally, the proofs of a proposition p in Simple type theory are expressed as terms of type $(\varepsilon\ p)$. For example, the usual proof of the proposition $\forall X : o\ (X \Rightarrow X)$ is expressed as the term $\lambda X : (\eta\ o)\ \lambda\alpha : (\varepsilon\ X)\ \alpha$.

This leads to the theory presented in Figure 1.

3.2 The Calculus of constructions as a theory in the $\lambda\Pi$-calculus modulo theory

We consider a slight extension of the Calculus of constructions with a symbol *nat* of type $Kind$. Such an extension can be obtained just by adding a constant *nat* and a rule assigning it the type $Kind$, a rule allowing to declare a variable of type $Kind$, or an extra sort allowing to declare a variable of type $Kind$ [19, 20].

$$
\begin{array}{rcl}
type & : & Type \\
\eta & : & type \to Type \\
o & : & type \\
nat & : & type \\
arrow & : & type \to type \to type \\
\varepsilon & : & (\eta\ o) \to Type \\
\Rightarrow & : & (\eta\ o) \to (\eta\ o) \to (\eta\ o) \\
\forall & : & \Pi a : type\ (((\eta\ a) \to (\eta\ o)) \to (\eta\ o))
\end{array}
$$

$$
\begin{array}{rcl}
(\eta\ (arrow\ x\ y)) & \longrightarrow & (\eta\ x) \to (\eta\ y) \\
(\varepsilon\ (\Rightarrow\ x\ y)) & \longrightarrow & (\varepsilon\ x) \to (\varepsilon\ y) \\
(\varepsilon\ (\forall\ x\ y)) & \longrightarrow & \Pi z : (\eta\ x)\ (\varepsilon\ (y\ z))
\end{array}
$$

Fig. 1. Simple type theory

This logic can be expressed, in the $\lambda\Pi$-calculus modulo theory [13], as the theory presented in Figure 2. Note that this presentation slightly differs from that of [13]: the symbol U_{Type} has been replaced everywhere by the term $\varepsilon_{Kind}(\dot{Type})$ allowing to drop the rule

$$\varepsilon_{Kind}(\dot{Type}) \longrightarrow U_{Type}$$

Then, to keep the notations similar to those of Simple type theory, the constant U_{Kind} is written *type*, the constant $\dot{Type}$, *o*, the constant ε_{Kind}, η, the constant ε_{Type}, ε, the constant $\dot{\Pi}_{\langle Kind,Kind,Kind\rangle}$, *arrow*, the constant $\dot{\Pi}_{\langle Type,Type,Type\rangle}$, $\Rightarrow$, the constant $\dot{\Pi}_{\langle Kind,Type,Type\rangle}$, $\forall$, and the constant $\dot{\Pi}_{\langle Type,Kind,Kind\rangle}$, π. Finally, a symbol *nat* is added, as we consider the extension of the Calculus of constructions with such a symbol.

3.3 Comparing Simple type theory and the Calculus of constructions

Now that the theories have been formulated in the same logical framework, we can compare their expressions.

A first difference is that the symbol *arrow* has type

$$type \to type \to type$$

that is

$$\Pi x : type\ (type \to type)$$

in Simple type theory and

$$\Pi x : type\ (((\eta\ x) \to type) \to type)$$

in the Calculus of constructions. This reflects the fact that this symbol is non-dependent in Simple type theory and dependent in the Calculus of constructions,

$$
\begin{array}{rcl}
type & : & Type \\
\eta & : & type \to Type \\
o & : & type \\
nat & : & type \\
arrow & : & \Pi x : type\ (((\eta\ x) \to type) \to type) \\
\varepsilon & : & (\eta\ o) \to Type \\
\Rightarrow & : & \Pi x : (\eta\ o)\ (((\varepsilon\ x) \to (\eta\ o)) \to (\eta\ o)) \\
\forall & : & \Pi x : type\ (((\eta\ x) \to (\eta\ o)) \to (\eta\ o)) \\
\pi & : & \Pi x : (\eta\ o)\ (((\varepsilon\ x) \to type) \to type)
\end{array}
$$

$$
\begin{array}{rcl}
(\eta\ (arrow\ x\ y)) & \longrightarrow & \Pi z : (\eta\ x)\ (\eta\ (y\ z)) \\
(\varepsilon\ (\Rightarrow\ x\ y)) & \longrightarrow & \Pi z : (\varepsilon\ x)\ (\varepsilon\ (y\ z)) \\
(\varepsilon\ (\forall\ x\ y)) & \longrightarrow & \Pi z : (\eta\ x)\ (\varepsilon\ (y\ z)) \\
(\eta\ (\pi\ x\ y)) & \longrightarrow & \Pi z : (\varepsilon\ x)\ (\eta\ (y\ z))
\end{array}
$$

Fig. 2. The Calculus of constructions

where, in the type $A \to B$, written $\Pi x : A\ B$, the type B may contain a variable x of type A.

In the same way, the symbol $\Rightarrow$ is non-dependent in Simple type theory, but it is dependent in the Calculus of constructions, where, in the proposition $A \Rightarrow B$, also written $\Pi x : A\ B$, the proposition B may contain a variable x, that is a proof of A.

In contrast, the symbol $\forall$ is dependent is both theories: in the expression $\forall x : A\ B$, the proposition B may always contain the variable x of type A.

Finally, there is an extra constant π in the Calculus of constructions, with its associated rewrite rule. This symbol permits to type functions mapping proofs to terms, for example a function mapping a proof of $\exists x\ P(x)$ to a term t verifying the predicate P.

So, the Calculus of constructions is an extension of Simple type theory, because the symbols *arrow* and $\Rightarrow$ are dependent and because it includes a symbol π.

3.4 Analyzing proofs expressed in the Calculus of constructions

We can define a subset S of the proofs expressed in the Calculus of constructions that do not use the dependency of the symbol *arrow*, do not use the dependency of the symbol $\Rightarrow$, and do not use the symbol π, where we say that a proof *does not use the dependency of the symbol arrow* if each time the symbol *arrow* is used, it is applied to two terms, the second one being a λ-abstraction where the abstracted variable does not occur in the body of the abstraction: $(arrow\ A\ \lambda x : (\eta\ A)\ B)$ with x not free in B, and that it *does not use the dependency of the symbol* $\Rightarrow$ if each time the symbol $\Rightarrow$ is used, it is applied to two terms, the second being a λ-abstraction where the abstracted variable does not occur in the body of the abstraction: $(\Rightarrow\ A\ \lambda x : (\varepsilon\ A)\ B)$ with x not free in B.

As we shall see, many proofs expressed in the Calculus of constructions are in this subset.

3.5 Translating proofs to Simple type theory

When a proof expressed in the Calculus of constructions is in the subset S, it can easily be translated to Simple type theory. All that needs to be done is to replace the terms of the form $(arrow\ A\ \lambda x : (\eta\ A)\ B)$ with $(arrow\ A\ B)$ and the terms of the form $(\Rightarrow\ A\ \lambda x : (\varepsilon\ A)\ B)$ with $(\Rightarrow\ A\ B)$.

If, in contrast a proof is not in the set S, then it genuinely uses a feature of the Calculus of constructions that does not exist in Simple type theory and it cannot be expressed in Simple type theory. In the same way a proof expressed in ZFC, that genuinely uses the axiom of choice, cannot be expressed in ZF.

When a proof, expressed in the Calculus of constructions, is an element of the set S and is translated to Simple type theory, we say that the proof and its translation are the *same* mathematical proof, expressed in different theories, although they are different linguistic objects. Generalizing this notion of identity of proofs across theories remains to be done.

4 An arithmetic library

The example of the translation of proofs from the Calculus of constructions to Simple type theory is a toy example, because there is no implementation of the Calculus of constructions *per se*. The systems Coq and Matita, for example, implement extensions of the Calculus of constructions with various features, at least inductive types and universes.

M. Boespflug and G. Burel [8] have shown how to extend the theory presented in Figure 2 to inductive types and A. Assaf [3, 4] has shown how to extend it to universes. This has permitted to express a large library of Matita proofs in Dedukti, including a proof of Fermat's little theorem.

F. Thiré [35] has then shown that the symbol π, and the dependency of the symbols *arrow* and $\Rightarrow$ could be eliminated from this library, but also that universes could be eliminated and that inductive type could be replaced by a very simple principle: an induction principle on natural numbers, that permits to prove propositions by induction and to define functions by induction.

So, we now have an evidence that Fermat's little theorem not only has a proof in the Calculus of inductive constructions with universes, but also in constructive Simple type theory. Such a result was expected, but note that neither proving Fermat's little theorem in Matita, that contains dependent products, inductive types, universes, etc. nor proving it in HOL Light, that contains the excluded middle, extensionality, choice etc., provides a proof in such a weak theory.

On the more practical side, this has permited to export this library from Dedukti to HOL Light, Isabelle/HOL, HOL4, Coq, etc. [36], HOL Light, Isabelle/HOL, and HOL4 sharing a common input language: OpenTheory [27].

The size of this library is 1.5 Mo. It contains around 340 lemmas. It is checked in Dedukti in a few milliseconds.

5 Abstracting enough

5.1 Natural numbers

Both in the Calculus of inductive constructions and in Simple type theory, it is possible to prove propositions by induction and to define functions by induction. But these principles are justified in different ways.

In the Calculus of inductive constructions, the declaration of the inductive type *nat* comes with a recursion operator that permits to define functions by induction and, as proofs are functions, this operator permits also to build proofs by induction.

In Simple type theory, in contrast, the set of natural numbers is impredicatively defined as the intersection of all sets containing zero and closed by successor. From this definition, the induction principle can be proved. Then, this induction principle and other properties of natural numbers permit to prove the existence of functions defined by induction [24].

These details should be ignored by the arithmetic library, that should be exported to any system that contains a notion of natural number, an induction principle and a way to define functions by induction, regardless the way this induction principle is proved and this induction operator is defined there [36]. Using such an abstract definition of the natural numbers, R. Cauderlier and C. Dubois [10] have built a proof of the correctness of Eratosthenes' sieve in the expression of FoCaLiZe in Dedukti, using definitions coming from Coq and lemmas coming from HOL Light.

5.2 Connectives and quantifiers

The same holds for the connectives and quantifiers, that are primitive neither in the Calculus of inductive constructions nor in Simple type theory. They are defined as inductive types in the Calculus of inductive constructions. They are defined from equality in Simple type theory [25, 1, 2].

But these details should be ignored by the arithmetic library, that only needs to specify that A should be provable when $A \wedge B$ is, etc. regardless the way this connective $\wedge$ is defined.

So, developing a library of proofs that can be exported to different proof systems gives a formal counterpart to the slogan that defining real numbers with Cauchy's construction or with Dedekind's is immaterial, or that defining complex numbers as ordered pairs of real numbers, as similarities, or as classes of polynomials is immaterial.

Eventually, this should lead to define algorithms to transform proofs about objects in a given structure to proofs about objects in an isomorphic structure [26, 37].

6 Classical and constructive logics

An important difference between logics, and proof systems implementing these logics, is that some of them are classical and others are constructive, that is the excluded middle $A \vee \neg A$ is provable in some but not in others. For example, Simple type theory, and HOL Light, are classical and the Calculus of inductive constructions, and Matita, are constructive.

A logical framework, such as the $\lambda\Pi$-calculus modulo theory, and its implementation Dedukti, should not make any choice on the excluded middle, but should be able to express both classical and constructive logics as theories. A possibility is to not assume the excluded middle in the framework and include it as an axiom in the definition of some theories. Then, the proofs of Simple type theory, for instance those developed in the system HOL Light, can be expressed in Dedukti using the excluded middle and just like in Section 3, we can analyze which of these proofs use the excluded middle and which do not, and translate these to another theory: constructive Simple type theory.

An alternative is to use the idea, defended for example in [22, 32, 14, 31], that the excluded middle is not a question of theory, but a question of meaning of the connectives and quantifiers. This leads to introduce two existential quantifiers: the constructive one $\exists$ and the classical one $\exists_c$, two disjunctions, etc. and deduction rules defining the meaning of these connectives and quantifiers, in such a way that $A \vee \neg A$ is not provable, but $A \vee_c \neg_c A$ is.

This permits to define connectives and quantifiers once for all in the framework and to use various quantifiers in various theories, as well as translating proofs using one set of quantifiers into proofs using another [21, 9], changing, in this case, the statement of the theorem.

7 Future work

The arithmetic library, described in Section 4, is, of course, only the beginning of a library of proofs, that could be shared by various proof systems. Each proof in this library should be labeled with the ingredients it uses, hence the systems to which it may be exported.

Also, we now have a formal proof of Fermat's little theorem, in constructive Simple type theory, but we should continue to transform it, to express it in weaker theories, such as Heyting arithmetic and beyond.

Acknowledgments

Many thanks to Catherine Dubois, Stéphane Graham-Lengrand, and François Thiré for very useful remarks on a first draft of this paper.

References

1. P.B. Andrews. A reduction of the axioms for the theory of propositional types. *Fundam. Math.*, 52:345–350, 1963.

2. P.B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Academic Press, 1986.
3. A. Assaf. A Calculus of Constructions with explicit subtyping. In H. Herbelin, P. Letouzey, and M. Sozeau, editors, *Types*, volume 39 of *LIPICS*, 2014.
4. A. Assaf. *A framework for defining computational higher-order logics*. PhD thesis, École polytechnique, 2015.
5. A. Assaf, G. Burel, R. Cauderlier, D. Delahaye, G. Dowek, C. Dubois, F. Gilbert, P. Halmagrand, O. Hermant, and R. Saillard. Dedukti: a logical framework based on the lambda-Pi-calculus modulo theory. Manuscript, 2016.
6. H. Barendregt. Lambda calculi with types. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 117–309. Oxford University Press, 1992.
7. M. Boespflug. *Conception d'un noyau de vérification de preuves pour le lambda-Pi-calcul modulo*. PhD thesis, École polytechnique, 2011.
8. M. Boespflug and G. Burel. CoqInE : Translating the Calculus of Inductive Constructions into the lambda-Pi-calculus modulo. In *Second International Workshop on Proof Exchange for Theorem Proving*, 2012.
9. R. Cauderlier. A rewrite system for proof constructivization. `https://who.rocq.inria.fr/Raphael.Cauderlier/constructivization.pdf`, 2017.
10. R. Cauderlier and C. Dubois. Focalize and Dedukti to the rescue for proof interoperability. In M. Ayala-Rincón and C.A. Muñoz, editors, *Interactive Theorem Proving*, volume 10499 of *Lecture Notes in Computer Science*, pages 131–147. Springer, 2017.
11. A. Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940.
12. T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, pages 95–120, 1988.
13. D. Cousineau and G. Dowek. Embedding Pure Type Systems in the lambda-Pi-calculus modulo. In S. Ronchi Della Rocca, editor, *Typed lambda calculi and applications*, volume 4583 of *Lecture Notes in Computer Science*, pages 102–117. Springer, 2007.
14. G. Dowek. On the definition of the classical connectives and quantifiers. In E.H. Haeusler, W. de Campos Sanz, and B. Lopes, editors, *Why is this a Proof?, Festschrift for Luiz Carlos Pereira*. College Publications, 2015.
15. G. Dowek, Th. Hardin, and C. Kirchner. HOL-lambda-sigma: an intentional first-order expression of higher-order logic. *Mathematical Structures in Computer Science*, 11:1–25, 2001.
16. G. Dowek, Th. Hardin, and C. Kirchner. Theorem proving modulo. *Journal of Automated Reasoning*, 31:33–72, 2003.
17. G. Dowek and B. Werner. Proof normalization modulo. *The Journal of Symbolic Logic*, 68(4):1289–1316, 2003.
18. H. Friedman. Systems of second-order arithmetic with restricted induction, I, II. *The Journal of Symbolic Logic*, 41(2):557–559, 1976.
19. H. Geuvers. *Logics and Type systems*. PhD thesis, Nijmegen, 1993.
20. H. Geuvers. The Calculus of Constructions and Higher Order Logic. In Ph. de Groote, editor, *The Curry-Howard isomorphism*, volume 8 of *Cahiers du Centre de logique*, pages 139–191. Université catholique de Louvain, 1995.
21. F. Gilbert. Automated constructivization of proofs. In J. Esparza and A.S. Murawski, editors, *Foundations of Software Science and Computation Structures*, volume 10203 of *Lecture notes in computer science*, pages 480–495. Springer, 2017.

22. J.-Y. Girard. On the unity of logic. *Annals of Pure and Applied Logic*, 59(3):201–217, 1993.
23. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.
24. L. Henkin. On mathematical induction. *The American Mathematical Monthly*, 67(4):323–338, 1960.
25. L. Henkin. A theory of propositional types. *Fundam. Math.*, 52:323–344, 1963. Errata Ibid., 53, 119, 1964.
26. B. Huffman and Kunçar. Lifting and transfer: A modular design for quotients in Isabelle/HOL. In G. Gonthier and M. Norrish, editors, *Certified Programs and Proofs*, pages 131–146, 2013.
27. J. Hurd. The OpenTheory standard theory library. In M. Bobaru, K. Havelund, G.J. Holzmann, and R. Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 2011.
28. F. Kirchner. A finite first-order theory of classes. In Th. Altenkirch and C. McBride, editors, *Types for Proofs and Programs*, Lecture Notes in Computer Science. Springer, 2006.
29. A. Leitsch. On proof mining by cut-elimination. In D. Delahaye and B. Woltzenlogel Paleo, editors, *All about Proofs, Proofs for All*, volume 55 of *Mathematical Logic and Foundations*, pages 173–200. College Publications, 2015.
30. B. Nordström, K. Petersson, and J.M. Smith. Martin-Löf's type theory. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 1–37. Clarendon Press, 2000.
31. L.C. Pereira. Personal communication, 2017.
32. D. Prawitz. Classical versus intuitionistic logic. In E.H. Haeusler, W. de Campos Sanz, and B. Lopes, editors, *Why is this a Proof?, Festschrift for Luiz Carlos Pereira*. College Publications, 2015.
33. R. Saillard. *Type Checking in the Lambda-Pi-Calculus Modulo: Theory and Practice*. PhD thesis, École des Mines, 2015.
34. S.G. Simpson. *Subsystems of second-order arithmetic*. Cambridge University Press, 2009.
35. F. Thiré. Personal communication, 2017.
36. F. Thiré. Sharing arithmetic proofs from Dedukti to HOL. Manuscript, 2017.
37. T. Zimmermann and H. Herbelin. Automatic and transparent transfer of theorems along isomorphisms in the Coq proof assistant. Work in progress session, Conference on Intelligent Computer Mathematics, `https://hal.archives-ouvertes.fr/hal-01152588/file/paper.pdf`, 2015.