Proof Theory and Dependent Type Theory: Distinct Foundations for Designing Proof Assistants

Dale Miller
Inria Saclay and LIX, Institut Polytechnique de Paris

This paper examines the foundational distinctions between proof theory and dependent type theory (DTT) in the design of interactive theorem provers. While several implemented systems are designed using the dependently typed λ-calculus to represent proofs, no major proof assistant is designed using modern structural proof theory, even though, as I will argue here, the sequent calculus offers a compelling alternative framework. Six specific topics are proposed where the proof-theoretic perspective is arguably superior to the DTT perspective. These topics include the separation of logic from proof structure, the strategic use of non-determinism in proof reconstruction, and the avoidance of complex typing-discipline issues such as universe levels and proof irrelevance. The final topic—the treatment of bindings—is further developed to demonstrate how a natural, intensional approach is achieved through the mobility of binders. This methodology is illustrated via the Abella theorem prover, which leverages λ-tree syntax and the ∇-quantifier to provide an elegant environment for reasoning about the meta-theory of languages and logics involving complex binding.

1 Introduction

In 2006, Wiedijk published a small volume [57] that showed the formalization of the proof of the irrationality of $\sqrt{2}$ in the following seventeen interactive theorem provers:

> ACL2, Alfa/Agda, B Method, Coq, HOL, IMPS, Isabelle/Isar, Lego, Metamath, Mizar, Minlog, Nuprl, Ωmega, Otter/Ivy, PVS, PhoX, and Theorema.

These systems vary significantly in their underlying logical foundations. Some are based on first-order logic, while others accept higher-order quantification. Employing a range of proof structures, some of these systems use Frege-style proofs (involving axioms and a small set of inference rules) because these can support simple proof-checking kernels. Others take a more sophisticated approach to proof structures and employ either a natural deduction style à la Gentzen and Prawitz, or the dependently typed λ-calculus directly. The role of *types* in these systems varies greatly. Most of these systems allow at least multi-sorted, first-order quantification. Others implement Church's Simple Theory of Types and allow quantification over function and predicate types. In addition, certain predicates that have well-understood reasoning principles can often be recast as types, with the usual consideration of whether type checking and/or type inference are decidable for those predicates cum types. Some of these systems (and the more recent Lean system [45]), are based on *dependent type theory* (DTT), whose origin is often traced back to De Bruijn [8] and Martin-Löf [37].

In this paper, I argue that modern aspects of proof theory can offer an alternative perspective to the design narrative behind the use of dependently typed λ-calculus in proof assistants. I will assume here that the reader is familiar with the latter formalism but is not familiar with the many advances that have occurred within the general topic of structural proof theory. I will survey that topic in the next section.

To appear in EPTCS.

2 Structural proof theory

Structural proof theory was started by Gentzen [25] with his invention of natural deduction and the sequent calculus. In particular, Gentzen described his intentions about natural deduction:

> I intended first to set up a formal system which comes as close as possible to actual reasoning. The result is a *'calculus of natural deduction'*.

Thus, it is no surprise that this calculus of natural deduction has had important influences on various proof assistants.

Early in the introduction of [25], Gentzen justifies the introduction of the sequent calculus as a technical vehicle for stating and proving the *Hauptsatz* for *both* intuitionistic and classical logic. In the general area of computational logic, sequent calculus is often seen as a technical device for tracing the search for an actual proof. In fact, there is the old chestnut: "Natural deductions are proofs while sequent calculus proofs are the computation of a proof" (see, for example, [28, Section 5.4]).

As we shall see in this paper, the sequent calculus can be viewed, not only as a useful technical device, but as a central framework for developing and justifying possible proof assistant designs. It is also possible to list several successful deployments of the sequent calculus in domains other than proof system design.

- Gentzen used the sequent calculus to prove the consistency of Peano arithmetic [26].
- Various researchers, for example [32, 46], have used the sequent calculus to provide independence proofs in certain axioms with respect to other axioms within mathematical theories.
- Often, the most successful proof system for a modern logic, such as modal and linear logics, is given using the sequent calculus.
- Various logic programming languages have been developed using the sequent calculus in classical, intuitionistic, and linear logics [40].
- The design of at least one model checker has been built on a sequent calculus proof theory that accounts for fixed points [7, 30].

Of the many recent developments in the theory of the sequent calculus that might not be familiar to those working on designing and implementing proof systems, we can list the notions of *polarity* and *focusing* (both inspired by the sequent calculus of linear logic) and the notion of the *mobility of binders*. These topics will be expanded on in the following section.

3 A proof theorist's view of dependent type theory

In this section, I identify six topics where a proof-theoretic perspective offers distinct advantages over that of DTT. Naturally, one could conversely argue that DTT provides a superior framework in other areas; its strengths include mature implementations, widespread adoption, robust support for automated inference, and the ability to provide concise specifications in a variety of domains. My objective here is to articulate the proof-theoretic perspective, as it is seldom represented in the literature. Furthermore, by focusing on foundational issues, I relegate the current state of software implementation to a secondary concern.

3.1 Regarding the structure of proof

A given DTT usually settles two questions simultaneously:

1. Which logic is being formalized? This is typically intuitionistic logic.
2. What is a proof? These are typically natural deduction proofs encoded as dependently typed λ-terms.

A proof theorist separates these questions, thereby allowing for a possibly large variety of proof systems for a fixed logic. For example, when fixing the logic to be intuitionistic, proofs can be formalized using natural deduction, sequent calculus, and tableaux [21], and inference rules can be applied in a forward chaining as well as a backward chaining fashion. When fixing the logic to be classical logic, many more proof structures are possible, additionally including Herbrand disjunctions, resolution refutations, natural deduction with restart [22], etc. If the logic is fixed to be linear, then all the previous proof structures are possible, including various forms of proof nets.

Of course, once one has a solid implementation of, say, natural deduction proofs as dependently typed λ-terms, one has a highly expressive computational setting capable of encoding a rich collection of other proof systems, as witnessed by the many proof systems that have been encoded into Dedukti [4]. Still, there is something to be said for treating these other proof systems as structures in their own right rather than merely as encodings into some other proof structure.

3.2 Problems with the proof-as-λ-term approach

Another frequently claim is that the λ-calculus is a canonical computational model for sequential computation. While I am not arguing against that particular sentiment, it is worth noting that many things about the (untyped) λ-calculus are far from canonical. For example, while λ-reduction is a confluent operation on λ-terms, the paths to (weak) normal forms are highly non-deterministic, with different computational platforms adopting different reduction strategies (call-by-value, call-by-name, call-by-need, etc), and these can affect correctness (e.g., call-by-value may not terminate with a normal form while call-by-name does) as well as efficiency.

Dependently typed λ-terms also introduce many complications. For example, there are issues around proof irrelevance (too many subproofs kept), implicit arguments (too inconvenient to supply all arguments), and universe levels (needed to organize rich typing structures). Approaches to encoding logic that omit such a rich and complex typing discipline do not need to address these issues, at least not in a foundational setting. Also, proofs-as-λ-terms can be highly redundant (leading to, for example, explicitly and implicitly typed versions of LF [50, 54]) and they do not include any explicit structure-sharing mechanisms. The sequent calculus, on the other hand, provides a simple and natural explicit mechanism [43].

3.3 The mutual development of dependent type theory and proof theory

Many central topics in the study of proof theory and DTT were investigated around the same time and influenced each other.

1. The notion of normalizing proofs was first developed in the context of sequent calculus (via cut-elimination [25]) and natural deduction (via normalization [52, 53]). These normalization processes (including the elimination of non-atomic initial rules) yield the familiar notions of β and η-rules. Of course, these normalization processes were also studied in the untyped and simply

typed λ-calculus by Church also in the 1930s and 1940s [14, 15]. The introduction of these concepts into dependently typed λ-calculus waited until de Bruijn [8] and Martin-Löf [36] introduced their typed calculi.

2. Linear logic [27] appeared first as a development in proof theory and later moved to dependently typed λ-calculus shortly after (see, for example, [9, 31]).

3. The notion of dependently typed λ-terms in *canonical form* (as defined for LF in [29]) is closely related to the proof theoretic concept of *uniform proofs* (as shown in [19, 20]). Eventually, the linear logic notions of *polarization* and *focusing* [3] were generalized within the proof theory setting to classical and intuitionistic logics (see, for example, [16, 17, 35]). Focused proofs are now generally understood to yield useful notions of canonical normal forms of proofs [13, 39].

3.4 The dependent type theory approach to classical logic

Gentzen [25] abandoned natural deduction since he could not see how that style of proof could be used uniformly to yield the Hauptsatz for both classical and intuitionistic logic. His solution for a more uniform approach to the proof theory of both logics was a multiple-conclusion sequent calculus. Unfortunately, Gentzen's better approach is generally ruled out in DTT, at least in the type theories in use in proof assistants today. The gap between treating classical logic proofs as dependently typed λ-terms plus axioms, such as the excluded middle, and viewing classical logic proofs, such as say Herbrand disjunctions (expansion trees) or resolution refutation, seems huge. Of course, elaborate techniques have been developed to bridge this gap, at least, in principle: see, for example, [18, 33].

3.5 Non-determinism provides a valuable resource

The trusted proof-checking kernels of proof assistants based on dependently typed λ-calculus are generally functional programs that expect the proof structures they check to contain enough information to make checking completely deterministic. It is well known that non-determinism can be a valuable resource in designing and verifying certificates: just consider the P and NP complexity classes. Allowing non-determinism in the design of both proof certificates and proof checkers allows for important trade-offs between proof certificate size and checking time.

Having a kernel that allows non-determinism (via backtracking search) is certainly possible (as in logic-programming-based kernels). Example: Consider a proof certificate for the claim that the sequent $\Gamma \vdash A$ is an instant of the *initial* rule, which can only be the case when A is a member of Γ. On one hand, the certificate could be constant in size while the proof checking kernel would then need to search for A in Γ: something that could be accomplished with a non-deterministic search (made deterministic in actual deterministic kernel using search). On the other hand, the certificate could provide the exact name/label of the assumption in Γ that is also labeled with A. In this case, the certificate is no longer constant-sized, but its checking would be a more direct, deterministic process. Having this trade-off between proof certificate size and non-deterministic proof reconstruction seems desirable. A kernel for such checking is a rather direct combination of some form of (higher-order) unification and backtracking search. While incorporating such features into the trusted code base of a kernel will increase the complexity of such kernels, these features have been well-developed and exploited in proof systems written in the λProlog programming language [41] and the Isabelle prover [47] since the late 1980s.

3.6 Treatment of bindings

In most modern proof assistant based on DTT or first-order logic, no canonical treatment for bindings has appeared. Instead, a wide range of technical approaches to encoding bindings have been deployed and studied. In general, the specifications of bindings in syntax are still considered problematic in most conventional systems, given the existence of challenge problems (POPLMark [5], POPLMark reloaded [1]), case studies, benchmarks, and surveys. The Twelf [49] and Beluga [51] systems are exceptions since they offer a computing environment where computed values can be dependently typed λ-terms. However, of the proof assistants lists in Section 1, only Isabelle and Minlog offer direct support for bindings in data structures and the associated notion of (higher-order) unification on such structures.

The sequent calculus provides an elegant and powerful setting to specify and compute with syntactic objects containing bindings, something that, to date, has not been matched in the setting of dependently typed λ-calculi. The approach available in the sequent calculus is outlined in more detail in the following section.

4 Sequents and binders

By making a slight embellishment of Gentzen's formulation of sequents, we write sequents here as $\Sigma : \Gamma \vdash C$, where the collection of variables in Σ is interpreted as being bound over $\Gamma \vdash C$. The variables in Σ correspond to Gentzen's notion of *eigenvariables*.

To illustrate how binder can be used with sequent calculus proof rules, consider specifying the predicate *typeof*, which relates an encoding of untyped λ-terms (given by the function $\lceil\cdot\rceil$) with an encoding of simple types (using $\rightarrow$ for the function type constructor). Here, we assume as is customary in many encodings of λ-terms (see, [41], for example) that binders in the untyped λ-terms are mapped to bindings in the result of the encoding.

When reading inference rules from conclusion to premises (as one does to understand the impact of an inference rule during the search for a proof), binders can be *instantiated*, as in the following instance of the left-introduction for $\forall$.

$$\frac{\Sigma : \Gamma, \mathit{typeof}\ c\ (\mathit{int} \rightarrow \mathit{int}) \vdash C}{\Sigma : \Gamma, \forall\alpha(\mathit{typeof}\ c\ (\alpha \rightarrow \alpha)) \vdash C}\ \forall L$$

Here, the bound variable α is instantiated with the term *int*, a term denoting the integer type. Binders can also *move*, as illustrated by the following inference rules.

$$\frac{\dfrac{\Sigma, \mathbf{x} : \Gamma, \mathit{typeof}\ x\ \alpha \vdash \mathit{typeof}\ \lceil B\rceil\ \beta}{\Sigma : \Gamma \vdash \forall\mathbf{x}(\mathit{typeof}\ x\ \alpha \supset \mathit{typeof}\ \lceil B\rceil\ \beta)}\ \forall R, \supset R}{\Sigma : \Gamma \vdash \mathit{typeof}\ \lceil\lambda\mathbf{x}.B\rceil\ (\alpha \rightarrow \beta)}$$

Here, the lower inference rule is justified by some external definition of the predicate *typeof* (a familiar specification technique for defining typing dating back to the early days of λProlog and LF [20, 29]). Here, binders move from *term-level* (λx) to *formula-level* ($\forall x$) to *proof-level* (eigenvariable): that is, this bound variable never becomes free. Thus, *binder mobility* is one way to formally realize A. Perlis's Epigram 47: "There is no such thing as a free variable." [48]. A consequence of this approach to binders is that the technique for implementing binders (e.g., named variables, nameless dummies, etc) does not need to be revealed to the person writing the logical specification of, in this case, *typeof*: instead, binders never stop being bound.

Another aspect of binders in the proof-theory setting is that they do not need to represent function spaces: they can be viewed, in the appropriate logical setting, as abstractions over syntax. Consider, for example, the formula

$$\forall w.\ \lambda x.x \neq \lambda x.w,$$

where w and x are given the same primitive type. If one views the λ-binding as specifying a function (i.e., extensionally), then this formula is not a theorem since the identity and a constant-valued function can coincide on singleton domains. If one views the λ-binder as simply part of the syntax (up to α-conversion, of course), then this formula should be a theorem since no (capture avoiding substitution) instance of $\lambda x.w$ can be equal to $\lambda x.x$.

This approach to binders in which syntactic binders are movable is called the *λ-tree syntax* approach [38], and it is available in the λProlog programming language [41], the LF logical framework [29], and the Rocq plugin for Elpi [55].

The sequent calculus can support at least one additional binding site other than the prefixed Σ binder. In particular, we add a binding context to each occurrence of a formula in a sequent. Thus, instead of the sequent $\Sigma\colon B_1,\ldots,B_n \vdash B_0$, we have the more general

$$\Sigma\colon \sigma_1 \triangleright B_1,\ldots,\sigma_n \triangleright B_n \vdash \sigma_0 \triangleright B_0.$$

Here, σ_i is a list of distinct variables scoped over B_i. The expression $\sigma_i \triangleright B_i$ is called a *generic judgment*. In order to exploit these proof-level binding sites, a new formula-level binder is needed: this is the role of the ∇-quantifier, for which the left and right introduction rules are given as

$$\frac{\Sigma : (\sigma, x : \tau) \triangleright B, \Gamma \longrightarrow \mathscr{C}}{\Sigma : \sigma \triangleright \nabla_\tau x.B, \Gamma \longrightarrow \mathscr{C}} \qquad \text{and} \qquad \frac{\Sigma : \Gamma \longrightarrow (\sigma, x : \tau) \triangleright B}{\Sigma : \Gamma \longrightarrow \sigma \triangleright \nabla_\tau x.B}\ .$$

Because of the similarity of the left and right introduction rules, it follows easily that ∇ is self-dual: that is, both the sequents $\neg\nabla x.Bx \vdash \nabla x.\neg Bx$ and $\nabla x.\neg Bx \vdash \neg\nabla x.Bx$ are provable. Furthermore, ∇ around an equality judgment can simply reduce to an equality judgment around λ-terms: that is, $\nabla x.(t = s)$ is provable exactly when $\lambda x.t = \lambda x.s$ is provable. Thus, the formula $\forall w.\ \lambda x.x \neq \lambda x.w$, turns out to be logically equivalent to $\forall w.\neg(\lambda x.x = \lambda x.w)$, which can be proved in a suitable system (such as Abella) since the (higher-order) unification problem $(\lambda x.x = \lambda x.w)$ has no unifiers. For details on how ∇-quantification interacts with the propositional constants, the other quantifiers, and induction and coinduction, see [24, 42].

The Abella proof assistant [6, 23] was developed on proof theoretic principles and includes support for λ-tree syntax and the ∇-quantifier.

5 The Abella proof assistant

During the search for a proof in Abella, one works with collections of *goals*, which are displayed as

```
Variables: x1 ... xm
H1 : A1
...
Hn : An
============================
 C
```

where $x_1,\ldots,x_m$ are universally quantified variables, `H1`, ..., `Hn` are *hypothesis labels* that are each associated with a unique *hypothesis formula* drawn from $A_1,\ldots,A_n$ and C is a formula called the *conclusion*

of the goal. The variables and hypotheses comprise the *context* of the goal. Not only does Abella view such a goal as a sequent, in this case being,

$$x_1 : \tau_1, \ldots, x_m : \tau_m :: A_1, \ldots, A_n \vdash C,$$

but its tactics for advancing the search for a proof are understood using sequent calculus inference rules [11, 12]. Abella implements the ∇-quantifier and the structural principles behind the logical equivalences $(\nabla x \nabla y.B) \equiv (\nabla y \nabla x.B)$ and $(\nabla x.C) \equiv C$, provided x is not free in C. As a result of these equivalences, it is possible to treat the local binding signatures using an equivalent mechanism that relies on nominal constants [24] (as a result, the local signatures are not displayed explicitly in the Abella goal structure above).

Abella is well-suited for reasoning about the meta-theory of languages and logics, especially those involving binding. For example, various aspects of the meta-theory of the λ-calculus have had successful Abella specifications: see, for example, a formulation of the λ-cube [2], a mechanical formulation of higher-ranked polymorphic type inference [58], and a formalization of parts of Barendregt's theory of the lambda-calculus [34]. Specifications of the operational semantics of the π-calculus and its meta-theory can make significant use of Abella's support for binder mobility. For example, the difference between open and closed bisimulation for the π-calculus is a simple matter of switching between using intuitionistic logic and classical logic [42]. Similarly, the presence of the three quantifiers $\forall$, $\exists$, and ∇ provides an easy and elegant way to capture the large collection of modal operators that have been proposed to capture an array of properties of π-calculus [44, 56]. Additionally, rather direct treatment of bisimulation-up-to techniques for the π-calculus can also be captured [10]. The website for Abella, `https://abella-prover.org/`, contains several other example formulations using Abella.

6 Conclusion

In this paper, I have argued that modern structural proof theory, rooted in the sequent calculus, offers a robust and compelling foundational alternative to basing interactive theorem provers on the "proofs-as-terms" paradigm of DTT. I have offered six explicit topics for which the proof theory approach can be argued to provide a possibly deeper and more flexible design than that typically offered by dependently typed λ-calculus. The last of these topics—the treatment of binding in syntactic expressions—is perhaps most significant. In this setting, the proof-theoretic approach provides a natural treatment of bindings via binder mobility. By treating binders as movable abstractions over syntax rather than function spaces, we achieve an elegant framework for reasoning about the meta-theory of complex languages. Many of these proof-theoretic notions have been built into the Abella theorem prover: in particular, this prover makes available the λ-tree syntax approach to bindings and the ∇-quantifier.

Acknowledgments I thank the reviewers for their helpful comments and perspectives on an earlier draft of this paper.

References

[1] Andreas Abel, Guillame Allais, Aliya Hameer, Alberto Momigliano, Brigitte Pientka, Steven Schaefer & Kathrin Stark (2019): *POPLMark reloaded: Mechanizing proofs by logical relations*. Journal of Functional Programming 29, doi:10.1017/S0956796819000170.

[2] Beniamino Accattoli (2012): *Proof pearl: Abella formalization of lambda calculus cube property*. In Chris Hawblitzel & Dale Miller, editors: *Second International Conference on Certified Programs and Proofs*, *LNCS* 7679, Springer, pp. 173–187.

[3] Jean-Marc Andreoli (1992): *Logic Programming with Focusing Proofs in Linear Logic*. *J. of Logic and Computation* 2(3), pp. 297–347, doi:10.1093/logcom/2.3.297.

[4] Ali Assaf, Guillaume Burel, Raphaël Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant & Ronan Saillard (2023): *Dedukti: a Logical Framework based on the λΠ-Calculus Modulo Theory*. Technical Report, arXiv, doi:10.48550/ARXIV.2311.07185.

[5] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich & Steve Zdancewic (2005): *Mechanized Metatheory for the Masses: The POPLmark Challenge*. In: *Theorem Proving in Higher Order Logics: 18th International Conference*, *LNCS* 3603, Springer, pp. 50–65, doi:10.1007/11541868_4.

[6] David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu & Yuting Wang (2014): *Abella: A System for Reasoning about Relational Specifications*. *J. of Formalized Reasoning* 7(2), pp. 1–89, doi:10.6092/issn.1972-5787/4650.

[7] David Baelde, Andrew Gacek, Dale Miller, Gopalan Nadathur & Alwen Tiu (2007): *The Bedwyr system for model checking over syntactic expressions*. In F. Pfenning, editor: *21th Conf. on Automated Deduction (CADE)*, *LNAI* 4603, Springer, New York, pp. 391–397, doi:10.1007/978-3-540-73595-3_28.

[8] N. G. de Bruijn (1980): *A Survey of the Project AUTOMATH*. In J. P. Seldin & R. Hindley, editors: *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, Academic Press, New York, pp. 589–606.

[9] Iliano Cervesato & Frank Pfenning (1996): *A Linear Logic Framework*. In: *11th Symp. on Logic in Computer Science*, IEEE Computer Society Press, New Brunswick, New Jersey, pp. 264–275.

[10] Kaustuv Chaudhuri, Matteo Cimini & Dale Miller (2015): *A Lightweight Formalization of the Metatheory of Bisimulation-Up-To*. In Xavier Leroy & Alwen Tiu, editors: *Proceedings of the 4th ACM-SIGPLAN Conference on Certified Programs and Proofs*, ACM, Mumbai, India, pp. 157–166, doi:10.1145/2676724.2693170. Available at `https://hal.inria.fr/hal-01091524/document`.

[11] Kaustuv Chaudhuri, Arunava Gantait & Dale Miller (2025): *Designing a Safe Forward Chaining Tactic Using Productive Proofs*. In G. L. Pozzato & T. Uustalu, editors: *Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX)*, *LNAI* 15980, Springer, pp. 299–317, doi:10.1007/978-3-032-06085-3_16.

[12] Kaustuv Chaudhuri, Ulysse Gérard & Dale Miller (2018): *Computation-as-deduction in Abella: work in progress*. In: *13th international Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, Oxford, United Kingdom. Available at `https://hal.inria.fr/hal-01806154`.

[13] Kaustuv Chaudhuri, Dale Miller & Alexis Saurin (2008): *Canonical Sequent Proofs via Multi-Focusing*. In G. Ausiello, J. Karhumäki, G. Mauri & L. Ong, editors: *Fifth International Conference on Theoretical Computer Science*, *IFIP* 273, Springer, pp. 383–396, doi:10.1007/978-0-387-09680-3_26.

[14] Alonzo Church (1932): *A Set of Postulates for the Foundation of Logic*. *Annals of Mathematics* 33(2), pp. 346–366.

[15] Alonzo Church (1941): *The Calculi of Lambda-Conversion*. Princeton University Press.

[16] V. Danos, J.-B. Joinet & H. Schellinx (1995): *LKT and LKQ: sequent calculi for second order logic based upon dual linear decompositions of classical implication*. In J.-Y. Girard, Y. Lafont & L. Regnier, editors: *Advances in Linear Logic*, *London Mathematical Society Lecture Note Series* 222, Cambridge University Press, pp. 211–224, doi:10.1017/CBO9780511629150.

[17] R. Dyckhoff & S. Lengrand (2006): *LJQ: a strongly focused calculus for intuitionistic logic*. In A. Beckmann & *et al.*, editors: *Computability in Europe 2006*, *LNCS* 3988, Springer, pp. 173–185.

[18] Gabriel Ebner & Matthias Schlaipfer (2018): *Efficient Translation of Sequent Calculus Proofs Into Natural Deduction Proofs*. In Boris Konev, Josef Urban & Philipp Rümmer, editors: *Proceedings of the 6th Workshop on Practical Aspects of Automated Reasoning co-located with Federated Logic Conference 2018*

(FLoC 2018), Oxford, UK, July 19th, 2018, *CEUR Workshop Proceedings* 2162, CEUR-WS.org, pp. 17–33. Available at `https://ceur-ws.org/Vol-2162/paper-03.pdf`.

[19] Amy Felty (1991): *Encoding dependent types in an intuitionistic logic*. In Gérard Huet & Gordon D. Plotkin, editors: *Logical Frameworks*, Cambridge University Press, pp. 215–251.

[20] Amy Felty & Dale Miller (1990): *Encoding a Dependent-Type λ-Calculus in a Logic Programming Language*. In Mark Stickel, editor: *Proceedings of the 1990 Conference on Automated Deduction*, *LNAI* 449, Springer, pp. 221–235.

[21] Melvin Fitting (1983): *Proof Methods for Modal and Intuitionistic Logics*. D. Reidel, Dordrecht, The Netherlands.

[22] D. M. Gabbay & U. Reyle (1984): *N-Prolog: An Extension of Prolog with Hypothetical Implications. I*. *Journal of Logic Programming* 1, pp. 319–355.

[23] Andrew Gacek (2008): *The Abella Interactive Theorem Prover (System Description)*. In A. Armando, P. Baumgartner & G. Dowek, editors: *Fourth International Joint Conference on Automated Reasoning*, *LNCS* 5195, Springer, pp. 154–161. Available at `https://arxiv.org/abs/0803.2305`.

[24] Andrew Gacek, Dale Miller & Gopalan Nadathur (2011): *Nominal abstraction*. *Information and Computation* 209(1), pp. 48–73, doi:10.1016/j.ic.2010.09.004.

[25] Gerhard Gentzen (1935): *Investigations into Logical Deduction*. In M. E. Szabo, editor: *The Collected Papers of Gerhard Gentzen*, North-Holland, Amsterdam, pp. 68–131, doi:10.1007/BF01201353. Translation of articles that appeared in 1934-35. Collected papers appeared in 1969.

[26] Gerhard Gentzen (1969): *The Consistency of Elementary Number Theory*. In M. E. Szabo, editor: *The Collected Papers of Gerhard Gentzen*, North-Holland, Amsterdam, pp. 132–213, doi:10.1016/S0049-237X(08)70823-1. Translated from the 1936 German original.

[27] Jean-Yves Girard (1987): *Linear Logic*. *Theoretical Computer Science* 50(1), pp. 1–102, doi:10.1016/0304-3975(87)90045-4.

[28] Jean-Yves Girard, Paul Taylor & Yves Lafont (1989): *Proofs and Types*. Cambridge University Press.

[29] Robert Harper, Furio Honsell & Gordon Plotkin (1993): *A Framework for Defining Logics*. *Journal of the ACM* 40(1), pp. 143–184, doi:10.1145/138027.138060.

[30] Quentin Heath & Dale Miller (2019): *A proof theory for model checking*. *J. of Automated Reasoning* 63(4), pp. 857–885, doi:10.1007/s10817-018-9475-3.

[31] Martin Hofmann (2003): *Linear types and non-size increasing polynomial time computation*. *Information and Computation* 183(1), pp. 57–85.

[32] Oiva Ketonen (2022): *Investigations into the Predicate Calculus*. College Publications. Ed. by S. Negri and J. von Plato.

[33] Anja Petković Komel, Michael Rawson & Martin Suda (2025): *Case Study: Verified Vampire Proofs in the LambdaPi-calculus Modulo*. *arXiv*, doi:10.48550/arxiv.2503.15541.

[34] Adrienne Lancelot, Beniamino Accattoli & Maxime Vemclefs (2025): *Barendregt's Theory of the lambda-Calculus, Refreshed and Formalized*. In: *16th International Conference on Interactive Theorem Proving (ITP 2025)*, *LIPIcs* 352, pp. 13:1–13:22, doi:10.4230/LIPIcs.ITP.2025.13.

[35] Chuck Liang & Dale Miller (2009): *Focusing and Polarization in Linear, Intuitionistic, and Classical Logics*. *Theoretical Computer Science* 410(46), pp. 4747–4768, doi:10.1016/j.tcs.2009.07.041. Abstract Interpretation and Logic Programming: A Special Issue in Honor of Professor Giorgio Levi.

[36] Per Martin-Löf (1982): *Constructive Mathematics and Computer Programming*. In: *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, North-Holland, Amsterdam, pp. 153–175.

[37] Per Martin-Löf (1984): *Intuitionistic Type Theory*. Studies in Proof Theory Lecture Notes, Bibliopolis, Napoli.

[38] Dale Miller (2019): *Mechanized Metatheory Revisited*. *Journal of Automated Reasoning* 63(3), pp. 625–665, doi:10.1007/s10817-018-9483-3.

[39] Dale Miller (2025): *Linear logic using negative connectives*. In Maribel Fernández, editor: *10th International Conference on Formal Structures for Computation and Deduction (FSCD 2025)*, LIPIcs, pp. 29:1–29:22, doi:10.4230/LIPIcs.FSCD.2025.29.

[40] Dale Miller (2025): *Proof Theory and Logic Programming: Computation as Proof Search*. Cambridge University Press, doi:10.1017/9781009561280. Preprint available at `https://www.lix.polytechnique.fr/Labo/Dale.Miller/ptlp/`.

[41] Dale Miller & Gopalan Nadathur (2012): *Programming with Higher-Order Logic*. Cambridge University Press, doi:10.1017/CBO9781139021326.

[42] Dale Miller & Alwen Tiu (2005): *A proof theory for generic judgments*. *ACM Trans. on Computational Logic* 6(4), pp. 749–783, doi:10.1145/1094622.1094628.

[43] Dale Miller & Jui-Hsuan Wu (2023): *A positive perspective on term representations*. In Bartek Klin & Elaine Pimentel, editors: *31st EACSL Annual Conference on Computer Science Logic (CSL 2023)*, *LIPIcs* 252, Dagstuhl, Germany, pp. 3:1–3:21, doi:10.4230/LIPIcs.CSL.2023.3.

[44] Robin Milner, Joachim Parrow & David Walker (1993): *Modal Logics for Mobile Processes*. *Theoretical Computer Science* 114(1), pp. 149–171.

[45] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn & Jakob von Raumer (2015): *The Lean Theorem Prover (System Description)*. In Amy P. Felty & Aart Middeldorp, editors: *25th International Conference on Automated Deduction (CADE-25)*, *Lecture Notes in Computer Science* 9195, Springer, pp. 378–388, doi:10.1007/978-3-319-21401-6_26.

[46] Sara Negri & Jan von Plato (2011): *Proof Analysis: A Contribution to Hilbert's Last Problem*. Cambridge University Press, Cambridge, doi:10.1017/CBO9780511921629.

[47] Lawrence C. Paulson (1986): *Natural Deduction as Higher-Order Resolution*. *Journal of Logic Programming* 3, pp. 237–258, doi:10.17863/CAM.23642.

[48] Alan J. Perlis (1982): *Epigrams on Programming*. *ACM SIGPLAN Notices* 17(9), pp. 7–13, doi:10.1145/947955.1083808.

[49] Frank Pfenning & Carsten Schürmann (1999): *System Description: Twelf — A Meta-Logical Framework for Deductive Systems*. In H. Ganzinger, editor: *16th Conf. on Automated Deduction (CADE)*, *LNAI* 1632, Springer, Trento, pp. 202–206, doi:10.1007/3-540-48660-7_14.

[50] Brigitte Pientka (2013): *An insider's look at LF type reconstruction: everything you (n)ever wanted to know*. *Journal of Functional Programming* 23(1), pp. 1–37, doi:10.1017/S0956796812000408.

[51] Brigitte Pientka & Joshua Dunfield (2010): *Beluga: A Framework for Programming and Reasoning with Deductive Systems (System Description)*. In J. Giesl & R. Hähnle, editors: *Fifth International Joint Conference on Automated Reasoning*, *LNCS* 6173, pp. 15–21.

[52] Jan von Plato (2008): *Gentzen's proof of normalization for natural deduction*. *Bulletin of Symbolic Logic* 14(2), pp. 240–257, doi:10.2178/bsl/1208442829.

[53] Dag Prawitz (1965): *Natural Deduction*. Almqvist & Wiksell, Uppsala.

[54] Jason Reed (2008): *Redundancy Elimination for LF*. *Electronic Notes in Theoretical Computer Science* 199, pp. 89–106, doi:10.1016/j.entcs.2007.11.014.

[55] Enrico Tassi (2026): *Elpi: rule-based extension language*. Habilitation à diriger des recherches, Université Côte d'Azur, Valbonne, France. Available at `http://www-sop.inria.fr/members/Enrico.Tassi/hdr/hdr.pdf`.

[56] Alwen Tiu & Dale Miller (2010): *Proof Search Specifications of Bisimulation and Modal Logics for the π-calculus*. *ACM Trans. on Computational Logic* 11(2), pp. 13:1–13:35, doi:10.1145/1656242.1656248.

[57] Freek Wiedijk, editor (2006): *The Seventeen Provers of the World*. *LNCS* 3600, Springer.

[58] Jinxu Zhao, Bruno C. d. S. Oliveira & Tom Schrijvers (2018): *Formalization of a Polymorphic Subtyping Algorithm*. In: *International Conference on Interactive Theorem Proving (ITP 2018)*, *LNCS* 10895, pp. 604–622, doi:10.1007/978-3-319-94821-8_36.