Mechanized metatheory revisited

Dale Miller

Inria Saclay & LIX, Ecole polytechnique
Palaiseau, France

Abstract

Proof assistants and the programming languages that implement them need to deal with a range of expressions that involve bindings. Many mature proof assistants have been extended with various packages and libraries so that bindings are addressed using various techniques (e.g., de Bruijn numerals and nominal logic). I argue here, however, that bindings are such an intimate aspect of the structure of expressions that they should be accounted for directly in the underlying programming language support for proof assistants. High-level and semantically elegant programming language support can be found in rather old and familiar concepts. In particular, Church's Simple Theory of Types has long ago provided answers to how bindings interact with logical connectives and quantifiers. Similarly, the proof search interpretation of Gentzen's proof theory provides a rich model of computation that supports bindings. I outline several principles for dealing computationally with bindings that follow from their treatments in quantificational logic and sequent calculus. I also briefly describe some implementations of these principles that have helped to validate their effectiveness as computational principles.

1 Mechanization of meta theory

A decade ago, the POPLmark challenge suggested that the theorem proving community had tools that were close to being usable by programming language researchers to formally prove properties of their designs and implementations. The authors of the POPLmark challenge looked at existing practices and systems and urged the developers of proof assistants to make improvements to existing systems.

> Our conclusion from these experiments is that the relevant technology has developed almost to the point where it can be widely used by language researchers. We seek to push it over the threshold, making the use of proof tools common practice in programming language research—mechanized metatheory for the masses. [4]

In fact, a number of research teams have used proof assistants to formally proved significant properties of entire programming languages. Such properties include type preservation, determinancy of evaluation, and the correctness of an OS microkernel and of various compilers: see, for example, [17, 18, 20, 26].

As noted in [4], the poor support for binders in syntax was one problem that held back proof assistants from achieving even more widespread use by programming language researchers and practitioners. In recent years, a number of extensions to programming languages and to proof assistants have been developed for treating bindings. These go by names such as locally nameless [7, 33], nominal reasoning [3, 8, 32, 36], and parametric higher-order abstract syntax [9]. Some of these approaches involve extending underlying programming language implementations while the others do not extend the proof assistant or programming language but provide various packages, libraries, and/or abstract datatypes that attempt to hide and orchestrate various issues surrounding the syntax of bindings. In the end, nothing canonical seems to have appeared and we are left with a simple grid that rates different approaches on various attributes [31].

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Clearly, mature and extensible proof assistants, such as, say, Coq, HOL, and Isabelle/HOL can be extended to deal with syntactic challenges (such as bindings in syntax) that they were not originally designed to handle. At the same time, it seems plausible and desirable to pursue approaches to the problem of bindings in syntax and metatheory more generally. In the next section, I suggest an analogies that might help us take the courage to propose another approach to computation and reasoning with bindings.

2 An analogy: concurrency

Historically speaking, the first high-level, mature, and expressive programming languages to be developed were based on sequential computation. When those languages were forced to deal with concurrency, parallelism, and distributed computing, they were augmented with, say, thread packages and remote procedure calls. Earlier pioneers of computer programming languages and systems—e.g., Dijkstra, Hoare, Milner—saw concurrency and communications not as incremental improvements to existing imperative languages but as a new paradigm deserving a separate study. The concurrency paradigm required a fresh and direct examination and in this respect, we have seen a great number of concurrency frameworks appear: e.g., Petri nets, CSP, CCS, IO-automata, and π-calculus. Given the theoretical results and understanding that have flowed from work on these and related calculi, it has been possible to find ways for conventional programming languages to make accommodations within the concurrency and distributed computing settings. Such understanding and accommodations were not likely to flow from clever packages added to programming languages: new programming principles from the theory of concurrency and distributed computing were needed.

I will now present some foundational principles to the treatment of bindings that are important to accommodate directly and immediately, even if we cannot immediately see how those principles might fit into some existing, mature programming languages and proof assistants.

3 How abstract is your syntax?

Two of the earliest formal treatments of the syntax of logical expressions were given by Gödel [14] and Church [10] and, in both of these cases, their formalization involved viewing formulas as strings of characters. Clearly, such a view of logical expressions contains too much information that is not semantically meaningful (e.g., white space, infix/prefix distinctions, parenthesis) and does not contain explicitly semantically relevant information (e.g., the function-argument relationship). For this reason, those working with syntactic expressions generally parse such expressions into *parse trees*: such trees discard much that is meaningless (e.g., the infix/prefix distinction) and records directly more meaningful information (e.g., the child relation denotes the function-argument relation). One form of "concrete nonsense" generally remains in parse trees since they traditionally contain the *names* of bound variables.

One way to get rid of bound variable names is to use de Bruijn's nameless dummy technique [11] in which (non-binding) occurrences of variables are replaced by positive integers that count the number of bindings above the variable occurrence through which one must move in order to find the correct binding site for that variable. While such an encoding makes the check for α-conversion easy, it can greatly complicate other operations that one might want to do on syntax, such as substitution, matching, and unification. While all

such operations can be supported and implemented using the nameless dummy encoding [11, 19, 27], the complex operations on indexes that are needed to support those operations clearly suggests that they are best dealt within the implementation of a framework and not in the framework itself.

The following four principles about the encoding and treatment of syntax will guide our further discussions.

> **Principle 1:** The names of bound variables should be treated as the same kind of fiction as we treat white space: they are artifacts of how we write expressions and have no semantic content.

Of course, the name of variables are important for parsing and printing expressions but such names should not be part of the meaning of an expression. This first principle simply repeats what we stated earlier. The second principle is a bit more concrete.

> **Principle 2:** There is "one binder to ring them all."[1]

With this principle, we are adopting Church's approach [10] to binding in logic, namely, that one has only λ-abstraction and all other bindings are encoded using that binder.

> **Principle 3:** There is no such thing as a free variable.

This principle is taken from Alan Perlis's epigram 47 [28]. By accepting this principle, we recognize that bindings are never dropped to reveal a free variable: instead, we will ask for bindings to *move*. This possibility suggests the main novelty in this list of principles.

> **Principle 4:** Bindings have *mobility* and the equality theory of expressions must support such mobility.

Since the other principles are most likely familiar to the reader, I will now describe this last principle in more detail.

4 Mobility of bindings

Since typing rules are a common operation in metatheory, I illustrate the notion of binding mobility in that setting. In order to specify untyped λ-terms (to which one might attribute a simple type via an inference), we introduce a (syntactic) type tm and two constants

$$abs : (tm \to tm) \to tm \qquad \text{and} \qquad app : tm \to tm \to tm.$$

Untyped λ-terms are encoded as terms of type tm using the following translation:

$$\lceil x \rceil = x, \qquad \lceil \lambda x.t \rceil = (abs\ (\lambda x.\lceil t \rceil)), \qquad \text{and} \qquad \lceil (t\ s) \rceil = (app\ \lceil t \rceil\ \lceil s \rceil)$$

This translation has the property that it maps bijectively α-equivalence classes of untyped λ-terms to $\alpha\beta\eta$-equivalence classes of simply typed λ-terms of type tm.

In order to satisfy Principle 3 above, we shall describe a Gentzen-style sequent as a triple $\Sigma : \Delta \vdash B$ where B is the succedent (a formula), Δ is the antecedent (a multiset of formulas), and Σ a signature, that is, a list of variables that are formally bound over the scope of the sequent. Thus all free variables in the formulas in $\Delta \cup \{B\}$ are bound by Σ. Gentzen referred to the variables in Σ as *eigenvariables*.

[1] A scrambling of J. R. R. Tolkien's "One Ring to rule them all, ... and in the darkness bind them."

The following inference rule is a familiar rule.

$$\frac{\Sigma : \Delta, \mathit{typeof}\ x\ (\mathit{int} \to \mathit{int}) \vdash C}{\Sigma : \Delta, \forall\tau(\mathit{typeof}\ x\ (\tau \to \tau)) \vdash C}\ \forall L$$

This rule states that if the symbol x can be attributed the type $\tau \to \tau$ for all instances of τ, then it can be assumed to have the type $\mathit{int} \to \mathit{int}$. Thus, bindings can be instantiated (the $\forall\tau$ is removed by instantiation). On the other hand, consider the following rules.

$$\frac{\dfrac{\Sigma, x : \Delta, \mathit{typeof}\ \lceil x \rceil\ \tau \vdash \mathit{typeof}\ \lceil B \rceil\ \beta}{\Sigma : \Delta \vdash \forall x(\mathit{typeof}\ \lceil x \rceil\ \tau \supset \mathit{typeof}\ \lceil B \rceil\ \tau')}\ \forall R}{\Sigma : \Delta \vdash \mathit{typeof}\ \lceil \lambda x.B \rceil\ (\tau \to \tau')}\ \text{Definition of } \mathit{typeof}.$$

These rules illustrates how bindings can, instead, *move* during the construction of a proof. In this case, the term-level binding for x in the lower sequent can be seen as moving to the formula level binding for x in the middle sequent and then to the proof level binding (as an eigenvariable) for x in the upper sequent. Thus, a binding is not lost or converted to a "free variable": it simply moves.

This mobility of bindings needs supported from the equality theory of expressions. Clearly, equality already includes α-conversion by Property 1. We also need a small amount of β-conversion. If we rewrite this last inference rule using the definition of the $\lceil\cdot\rceil$ translation, we have the inference figures

$$\frac{\dfrac{\Sigma, x : \Delta, \mathit{typeof}\ x\ \tau \vdash \mathit{typeof}\ (Bx)\ \tau'}{\Sigma : \Delta \vdash \forall x(\mathit{typeof}\ x\ \tau \supset \mathit{typeof}\ (Bx)\ \tau')}\ \forall R}{\Sigma : \Delta \vdash \mathit{typeof}\ (\mathit{abs}\ B)\ (\tau \to \tau')}\ \text{Definition of } \mathit{typeof}.$$

Note that here B is a variable of arrow type $tm \to tm$ and that instances of these inference figures will create an instance of $(B\ x)$ that may be a β-redex. As I now argue, that β-redex has a limited form. First, observe that B is a schema variable that is implicitly universally quantified around this inference rule: if one formalizes this approach to type inference in, say, λProlog or Twelf, one would write a specification similar to the formula

$$\forall B \forall\tau \forall\tau' [\forall x(\mathit{typeof}\ x\ \tau \supset \mathit{typeof}\ (Bx)\ \tau') \supset \mathit{typeof}\ (\mathit{abs}\ B)\ (\tau \to \tau')].$$

Second, any closed instances of $(B\ x)$ that is a β-redex is such that the argument x is not free in the instance of B: this is enforced by the nature of (quantificational) logic since the scope of B is outside the scope of x. Thus, the only form of β-conversion that is needed to support this notion of binding mobility is the so-called β_0-conversion rule $(\lambda x.t)x = t$ or equivalently (in the presence of α-conversion) $(\lambda y.t)x = t[x/y]$, provided that x is not free in $\lambda y.t$.

Given that β_0-conversion is such a simple operation, it is not surprising that higher-order pattern unification, which simplifies higher-order unification to a setting only needing α, β_0, and η conversion, is decidable. For this reason, matching and unification can be used to help account for the mobility of binding. Note also that there is an elegant symmetry provided by binding and β_0-reduction: if t is a term over the signature $\Sigma \cup \{x\}$ then $\lambda x.T$ is a term over the signature Σ and, conversely, if $\lambda x.s$ is a term over the signature Σ then the β_0 reduction of $((\lambda x.s)\ y)$ is a term over the signature $\Sigma \cup \{y\}$.

5 Logic programming provides a framework

As is suggested by the discussion above, quantificational logic using the proof-search model of computation can be used to capture all four principles listed in the previous section. While

it might be possible to account for these principles also in, say, a functional programming language (a half-hearted attempt at such a design was made in [22]), the logic programming paradigm supplies an appropriate framework for satisfying all these properties. In particular, the higher-order hereditary Harrop [24] subset of an intuitionistic variant of Church's Simple Theory of Types [10] provides an expressive foundation for logic programming, much of which is built into λProlog [23].

The use of logic programming principles in proof assistants pushes against usual practice: since the first LCF prover [15], many (most?) proof assistants have not only been built using functional programming but also exploit functional programming principles in the constructive interpretation of proofs as well as the design of, say, tactics and tacticals.

Besides the obvious, there are several high-level differences between using functional programming and logic programming principles that are relevant to the mechanization of metatheory.

5.1 Expressions versus values

In logic programming, (closed) terms denote themselves and only themselves (in the sense of free algebra). It often surprises people that in Prolog, the goal `?- 3 = 1 + 2` fails, but the expression that is the numeral `3` and the expression `1 + 2` are, of course, different expression. The fact that they have the same value is a secondary calculation (performed in Prolog using the `is` predicate). Functional programming however fundamentally links expressions and values: the value of an expression is the result of applying some evaluation strategy (e.g., call-by-value) to an expression. Thus the value of both `3` and `1 + 2` is `3` and these two expressions are, in fact, equated. Of course, one can easily write datatypes in functional programming languages that denote only expressions: datatypes for parse trees are such an example. However, the global notion that expressions denote values is particularly problematic when expressions denote λ-abstractions. The value of such expressions in functional programming is trivial and immediate: such values simply denote a function (a closure). In the logic programming setting, however, an expression that is a λ-abstraction is just another expression: equality based on α (and η) conversion is non-trivial and, when the properties of Section 3 are respected, expressive. The λ-abstraction-as-expression aspect of logic programming is one of that paradigm's major advantages for the mechanization of metatheory.

5.2 Syntactic types

Given the central role of expressions (and not values), types in logic programming often simply denote *syntactic categories*. That is, such syntactic types are useful for distinguishing, say, encodings of types from terms from formula from proofs or program expressions from commands from evaluation contexts. For example, the *typeof* specification in Section 4 is a binary relation between the syntactic categories tm (for untyped λ-terms) and, say, ty (for simple type expression). The logical specification of the *typeof* predicate might attribute integer type or list type to different expressions via clauses such as

$$\forall T\colon tm\, \forall L\colon tm\, \forall \tau\colon ty\, [typeof\, T\, \tau \supset typeof\, L\, (list\, \tau) \supset typeof\, (T :: L)\, (list\, \tau)].$$

Given our discussion above, it seems natural to propose that if τ and τ' are both syntactic categories, then $\tau \to \tau'$ is a new syntactic category that describes objects of category τ' with a variable of category τ abstracted. For example, if o denotes the category of formula (a la [10]) and tm denotes the category of terms, then $tm \to o$ denotes the type of term-level

abstractions over formulas. As we have been taught by Church, the quantifiers $\forall$ and $\exists$ can then be seen as constructors that take expressions of syntactic category $tm \to o$ to formulas: that is, these quantifiers are given the syntactic category $(tm \to o) \to o$.

5.3 Substitution lemmas for free

Assume that there is only one rule for proving $(typeof\ (abs\ R)\ (\tau \to \tau'))$ (a typical assumption for the arrow type constructor). Since the introduction rules for $\forall$ and $\supset$ are invertible, the sequent $\Sigma : \Delta \vdash typeof\ (abs\ R)\ (\tau \to \tau')$ is provable if and only if $\Sigma, x : \Delta, typeof\ x\ \tau \vdash typeof\ (R\ x)\ \tau'$. Given that we are committed to using a proper logic (such as higher-order intuitionistic logic), it is the case that modus ponens is valid and that instantiating an eigenvariable in a provable sequent yields a provable sequent. In this case, the sequent $\Sigma : \Delta, typeof\ N\ \tau \vdash typeof\ (R\ N)\ \tau'$ must be provable. Thus, we have just shown, using nothing more than rather minimal assumptions about the specification of *typeof* (and formal properties of logic) that if $\Sigma : \Delta \vdash typeof\ (abs\ B)\ (\tau \to \tau')$ and $\Sigma : \Delta \vdash typeof\ N\ \tau$ then $\Sigma : \Delta \vdash typeof\ (B\ N)\ \tau'$. (Of course, instances of the term $(B\ N)$ are β-redexes and the reduction of such redexes result in the substitution of N into the bound variable of the term that instantiates B.) Such lemmas about substitutions are common and often difficult to prove [38]: in this setting, this lemma is essentially an immediate consequent of using logic and logic programming principles. In principle, Gentzen's cut-elimination theorem (the formal justification of modus ponens) is essentially the mother of all substitution lemmas. Even if one is not interested in using β-reduction as a black box to encode object-level substitution, this framework still provides simple means to specify substitution and to prove substitution lemmas [1].

5.4 Dominance of relational specifications

Another reason that logic programming can make a good choice for metatheoretic reasoning systems is that logic programming is based on relations (not functions) and that metatheoretic specifications are often dominated by relations. For example, the typing judgment describe in the Section 4 is a relation. Similarly, operational semantics, both small step (SOS) and big step (natural semantics) are relations. Occasionally, specified relations—typing or evaluation—describe a partial function but that is generally a result proved about the relation.

A few logic programming-based systems have been used to illustrate how typing and operational semantic specifications can be animated. The core engine of the Centaur project, called Typol, used Prolog to animate metatheoretic specifications and both λProlog and Twelf have been used to provide convincing and elegant specifications of typing and operational semantics for expressions involving bindings [2, 23].

6 λ-tree syntax

The term *higher-order abstract syntax* (HOAS) was originally defined as an approach to syntax that used "a simply typed λ-calculus enriched with products and polymorphism" [29]. A subsequent paper identified HOAS as a technique "whereby variables of an object language are mapped to variables in the meta-language" [30]. The term HOAS is problematic for a number of reasons. First, it seems that few, if any, researchers use this term in a setting that includes products and polymorphism (although simple and dependently typed λ-calculus are often used). Second, since the metalanguage (often the programming language) can vary

a great deal, the resulting notion of HOAS is can vary similarly, including the case where HOAS is a representation of syntax that incorporate function spaces on expressions [16]. Third, the adjective higher-order seems over used and inappropriate here: in particular, the equality (and unification) of terms discussed in Section 4 is completely valid without reference to typing. If there are no types, what exactly is "higher-order"? For these reasons, the term "λ-tree syntax", with its obvious parallel to the term "parse tree syntax" is now more commonly used for the style of encoding described in this paper [5, 23].

7 Reasoning with λ-tree syntax

There is more to mechanizing metatheory than performing unification and doing logic programming-style search. One should also deal with negations: for example, one wants to prove that certain terms do not have simple types (i.e., $\vdash \neg\exists\tau : ty.\ typeof\ (abs\ \lambda x\ (app\ x\ x))\ \tau$). Proving that a certain relation actually describes a (partial or total) function has proved to be an important kind of metatheorem to prove [30]. Additionally, one should also deal with induction and co-induction and be able to reason directly about, say, bisimulation of π-calculus expressions as well as confluence of λ-conversion. The Twelf system [30] is able to prove many of the simpler forms of such metatheorems.

In recent years, several researchers have developed two extensions to logic and proof theory that have made it possible to reason in rich and natural ways about expressions containing bindings. One of these extensions involved a proof theory for least and greatest fixed points: results from [21, 35] have made it possible to build automated and interactive theorem provers in a simple, relational setting. Another extension [13, 25, 25] introduced the ∇-quantifier which allows logic to reason in a rich and natural way with bindings: in terms of mobility of bindings, the ∇-quantifier provides an additional formula-level and proof-level binding, thereby enriching the expressiveness of quantificational logic.

Given these developments in proof theory, it has been possible to build both an interactive theorem prover, called Abella [5, 12], and an automatic theorem prover, called Bedwyr [6], that unfolds fixed points in a style similar to a model checker. These systems have successfully been able to prove a range of properties involving, for example, the λ-calculus and the π-calculus [1, 5, 34]. The directness and naturalness of the encoding for the π-calculus bisimulation is evident in the fact that simply changing the underlying logic from intuitionistic to classical changes the interpretation of that one definition from open bisimulation to late bisimulation [34].

8 Conclusions

The higher-order logic of Church and the sequent calculus of Gentzen provide a simple and high-level framework for specifying both computation and deduction involving syntax containing bindings. Given that the style of computation that is used here is based on the search for (cut-free) proofs, logic programming-style specifications seems a natural fit for mechanizing metatheory. Various computer systems, in particular, λProlog, Twelf, Bedwyr, and Abella are built on these principles and help to validate this approach to metatheory. A great deal more support for system building of these kind still need to be done in order to get this new breed of proof assistants accepted by the masses.

Acknowledgments. This work was funded by the ERC Advanced Grant ProofCert.

References

1 Beniamino Accattoli. Proof pearl: Abella formalization of lambda calculus cube property. In Chris Hawblitzel and Dale Miller, editors, *Second International Conference on Certified Programs and Proofs*, volume 7679 of *LNCS*, pages 173–187. Springer, 2012.

2 Andrew W. Appel and Amy P. Felty. Polymorphic lemmas and definitions in λProlog and Twelf. *Theory and Practice of Logic Programming*, 4(1-2):1–39, 2004.

3 Brian Aydemir, Aaron Bohannon, and Stephanie Weirich. Nominal reasoning techniques in Coq. In *International Workshop on Logical Frameworks and Meta-Languages:Theory and Practice (LFMTP)*, pages 69–77, Seattle, WA, USA, August 2006.

4 Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *Theorem Proving in Higher Order Logics: 18th International Conference*, number 3603 in LNCS, pages 50–65. Springer, 2005.

5 David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. Abella: A system for reasoning about relational specifications. *Journal of Formalized Reasoning*, 2015. To appear.

6 David Baelde, Andrew Gacek, Dale Miller, Gopalan Nadathur, and Alwen Tiu. The Bedwyr system for model checking over syntactic expressions. In F. Pfenning, editor, *21th Conf. on Automated Deduction (CADE)*, number 4603 in LNAI, pages 391–397, New York, 2007. Springer.

7 Arthur Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, pages 1–46, May 2011.

8 James Cheney and Christian Urban. Nominal logic programming. *ACM Trans. Program. Lang. Syst.*, 30(5):1–47, 2008.

9 Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In James Hook and Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 143–156. ACM, 2008.

10 Alonzo Church. A formulation of the Simple Theory of Types. *J. of Symbolic Logic*, 5:56–68, 1940.

11 Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with an application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34(5):381–392, 1972.

12 Andrew Gacek. The Abella interactive theorem prover (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Fourth International Joint Conference on Automated Reasoning*, volume 5195 of *LNCS*, pages 154–161. Springer, 2008.

13 Andrew Gacek, Dale Miller, and Gopalan Nadathur. Nominal abstraction. *Information and Computation*, 209(1):48–73, 2011.

14 Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte der Mathematischen Physik*, 38:173–198, 1931. English Version in [37].

15 Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *LNCS*. Springer, 1979.

16 M. Hofmann. Semantical analysis of higher-order abstract syntax. In *14th Symp. on Logic in Computer Science*, pages 204–213. IEEE Computer Society Press, 1999.

17 Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In

Proceedings of the 22nd Symposium on Operating Systems Principles (22nd SOSP'09), Operating Systems Review (OSR), pages 207–220, Big Sky, MT, October 2009. ACM SIGOPS.

18 Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

19 Chuck Liang, Gopalan Nadathur, and Xiaochu Qi. Choices in representing and reduction strategies for lambda terms in intensional contexts. *Journal of Automated Reasoning*, 33:89–132, 2005.

20 Donald MacKenzie. *Mechanizing Proof*. MIT Press, 2001.

21 Raymond McDowell and Dale Miller. Cut-elimination for a logic with definitions and induction. *Theoretical Computer Science*, 232:91–119, 2000.

22 Dale Miller. An extension to ML to handle bound variables in data structures: Preliminary report. In *Informal Proceedings of the Logical Frameworks BRA Workshop*, Nice, France, June 1990. Available as UPenn CIS technical report MS-CIS-90-59.

23 Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, June 2012.

24 Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

25 Dale Miller and Alwen Tiu. A proof theory for generic judgments. *ACM Trans. on Computational Logic*, 6(4):749–783, October 2005.

26 J. Strother Moore. A mechanically verified language implementation. *J. of Automated Reasoning*, 5(4):461–492, 1989.

27 Gopalan Nadathur and Debra Sue Wilson. A notation for lambda terms: A generalization of environments. *Theoretical Computer Science*, 198(1-2):49–98, 1998.

28 Alan J. Perlis. Epigrams on programming. *ACM SIGPLAN Notices*, pages 7–13, September 1982.

29 Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, June 1988.

30 Frank Pfenning and Carsten Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In H. Ganzinger, editor, *16th Conf. on Automated Deduction (CADE)*, number 1632 in LNAI, pages 202–206, Trento, 1999. Springer.

31 The POPLmark Challenge webpage. `http://www.seas.upenn.edu/~plclub/poplmark/`, 2015.

32 François Pottier. An overview of Cαml. In *ACM Workshop on ML*, ENTCS, pages 27–51, September 2005.

33 Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming*, 20(01):71–122, 2010.

34 Alwen Tiu and Dale Miller. Proof search specifications of bisimulation and modal logics for the π-calculus. *ACM Trans. on Computational Logic*, 11(2), 2010.

35 Alwen Tiu and Alberto Momigliano. Cut elimination for a logic with induction and coinduction. *Journal of Applied Logic*, 2012.

36 Christian Urban. Nominal reasoning techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40(4):327–356, 2008.

37 Jean van Heijenoort. *From Frege to Gödel: A Source Book in Mathematics, 1879-1931*. Source books in the history of the sciences series. Harvard Univ. Press, Cambridge, MA, 3rd printing, 1997 edition, 1967.

38 Myra VanInwegen. *The Machine-Assisted Proof of Programming Language Properties*. PhD thesis, University of Pennsylvania, May 1996.