# Mechanized Metatheory Revisited: An Extended Abstract

**Dale Miller**
Inria and LIX, École Polytechnique, Palaiseau, France
0000-0003-0274-4954

**Abstract**

Proof assistants and the programming languages that implement them need to deal with a range of linguistic expressions that involve bindings. Since most mature proof assistants do not have built-in methods to treat this aspect of syntax, many of them have been extended with various packages and libraries that allow them to encode bindings using, for example, de Bruijn numerals and nominal logic features. I put forward the argument 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 and not added later using packages and libraries. One possible approach to designing programming languages and proof assistants that directly supports such an approach to bindings in syntax is presented. The roots of such an approach can be found in the *mobility* of binders between term-level bindings, formula-level bindings (quantifiers), and proof-level bindings (eigenvariables). In particular, by combining Church's approach to terms and formulas (found in his Simple Theory of Types) and Gentzen's approach to sequent calculus proofs, we can learn how bindings can declaratively interact with the full range of logical connectives and quantifiers. I will also illustrate how that framework provides an intimate and semantically clean treatment of computation and reasoning with syntax containing bindings. Some implemented systems, which support this intimate and built-in treatment of bindings, will be briefly described.

**2012 ACM Subject Classification** Theory of computation: proof theory, automated reasoning, higher order logic

**Keywords and phrases** mechanized metatheory, mobility of binders, lambda-tree syntax, higher-order abstract syntax

**Digital Object Identifier** 10.4230/LIPIcs.TYPES.2016.1

**Related Version** A full version of this paper, titled "Mechanized metatheory revisited," is available at `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/pubs.html`.

**Funding** This work was funded by the ERC Advanced Grant ProofCert.

## Foreword

This extended abstract is a non-technical look at the mechanization of formalized metatheory. While this paper may be provocative at times, I mainly intend to shine light on a slice of literature that is developing a coherent and maturing approach to mechanizing metatheory.

## 1 Mechanization of metatheory

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


22nd International Conference on Types for Proofs and Programs (TYPES 2016).
Editors: Silvia Ghilezan, Herman Geuvers, and Jelena Ivetić; Article No. 1; pp. 1:1–1:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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. [5]

In fact, a number of research teams have used proof assistants to formally prove significant properties of programming language related systems. Such properties include type preservation, determinancy of evaluation, and the correctness of an OS microkernel and of various compilers: see, for example, [41, 42, 44, 59].

As noted in [5], 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 enhancements to programming languages and to proof assistants have been developed for treating bindings. These go by names such as locally nameless [12, 76], nominal reasoning [3, 14, 69, 83], and parametric higher-order abstract syntax [15]. 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 orchestrate various issues surrounding the syntax of bindings. In the end, nothing canonical seems to have arisen: see [4, 68] for detailed comparisons.

## 2 An analogy: concurrency theory

While extending mature proof assistants (such as Coq, HOL, and Isabelle) with facilities to handle bindings is clearly possible, it seems desirable to consider directly the computational principles surrounding the treatment of binding in syntax independent of a given programming language. Developments in programming design has, of course, run into similar situations where there was a choice to be made between accounting for features by extending existing programming languages or by the development of new programming languages. Consider, for example, the following analogous (but more momentous) situation.

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 the $\pi$-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.

Before directly addressing some of the computational principles behind bindings in syntax, it seems prudent to critically examine the conventional design of a wide range of proof assistants. (The following section updates a similar argument found in [52].)

## 3 Dropping mathematics as an intermediate

Almost all ambitious theorem provers in use today follow the following two step approach to reasoning about computation.

**Step 1:** *Implement mathematics.* This step is achieved by picking a general, well understood formal system. Common choices are first-order logic, set theory, higher-order logic [16, 36], or some foundation for constructive mathematics, such as Martin-Löf type theory [18, 19, 45].

**Step 2:** *Reduce reasoning about computation to mathematics.* Computation is generally encoded via some model theoretic semantics (such as denotational semantics) or as an inductive definition over an operational semantics.

A key methodological element of this proposal is that we shall drop mathematics as an intermediate and attempt to find more direct and intimate connections between computation, reasoning, and logic. The main problem with having mathematics in the middle seems to be that many aspects of computation are rather "intensional" but a mathematical treatment requires an extensional encoding. The notion of *algorithm* is an example of this kind of distinction: there are many algorithms that can compute the same function (say, the function that sorts lists). In a purely extensional treatment, it is functions that are represented directly and algorithm descriptions that are secondary. If an intensional default can be managed instead, then function values are secondary (usually captured via the specification of evaluators or interpreters).

For a more explicit example, consider whether or not the formula $\forall w_i.\ \lambda x.x \neq \lambda x.w$ is a theorem. In a setting where $\lambda$-abstractions denote functions (the usual extensional treatment), we have not provided enough information to answer this question: in particular, this formula is true if and only if the domain type $i$ is not a singleton. If, however, we are in a setting where $\lambda$-abstractions denote syntactic expressions, then it is sensible for this formula to be provable since no (capture avoiding) substitution of an expression of type $i$ for the $w$ in $\lambda x.w$ can yield $\lambda x.x$.

For a more significant example, consider the problem of formalizing the metatheory of bisimulation-up-to [56, 72] for the $\pi$-calculus [57]. Such a metatheory can be used to allow people working in concurrent systems to write hopefully small certificates (actual bisimulations-up-to) in order to guarantee that bisimulation holds (usually witnessed directly by only infinite sets of pairs of processes). In order to employ the Coq theorem prover, for example, to attack such metatheory, Coq would probably need to be extended with packages in two directions. First, a package that provides flexible methods for doing coinduction following, say, the Knaster-Tarski fixed point theorems, would be necessary. Indeed, such a package has been implemented and used to prove various metatheorems surrounding bisimulation-up-to (including the subtle metatheory surrounding weak bisimulation) [11, 70, 71]. Second, a package for the treatment of bindings and names that are used to describe the operational semantics of the $\pi$-calculus would need to be added. Such packages exist (for example, see [6]) and, when combined with treatments of coinduction, may allow one to make progress on the metatheory of the $\pi$-calculus. Recently, the Hybrid systems [27] has shown a different way to incorporate both induction, coinduction, and binding into a Coq (and Isabelle) implementation. Such an approach could be seen as one way to implement this metatheory task on top of an established formalization of mathematics.

There is another approach that seeks to return to the most basic elements of logic by reconsidering the notion of terms (allowing them to have binders as primitive features) and the notion of logical inference rules so that coinduction can be seen as, say, the de Morgan (and proof theoretic) dual to induction. In that approach, proof theory principles can be identified in that enriched logic with least and greatest fixed points [7, 47, 58] and with a treatment of bindings [81, 32]. Such a logic has been given a model-checking-style implementation [9] and is the basis of the Abella theorem prover [8, 31]. Using such implementations, the $\pi$-calculus has been implemented, formalized, and analyzed in some detail [80, 79] including some of the metatheory of bisimulation-up-to for the $\pi$-calculus [13].

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

## 4 How abstract is your syntax?

Two of the earliest formal treatments of the syntax of logical expressions were given by Gödel [35] and Church [16] 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 [21] 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 $\alpha$-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 [21, 43, 61], 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 treatment of bindings in syntax will guide our further discussions.

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

Of course, the name of variables are important for parsing and printing expressions (just as is white space) 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]

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

With this principle, we are adopting Church's approach [16] to binding in logic, namely, that one has only $\lambda$-abstraction and all other bindings are encoded using that binder. For example, the universally quantified expression $(\forall x.\, B\, x)$ is actually broken into the expression $(\forall(\lambda x.\, B\, x))$, where $\forall$ is treated as a constant of higher-type. Note that this latter expression is $\eta$-equivalent to $(\forall\ B)$ and universal instantiation of that quantified expression is simply the result of using $\lambda$-normalization on the expression $(B\ t)$. In this way, many details about quantifiers can be reduced to details about $\lambda$-terms.

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

This principle is taken from Alan Perlis's epigram 47 [63]. 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 [51, 53].

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

## 5 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 $\lambda$-terms (to which one might attribute a simple type via an inference), we introduce a (syntactic) type *tm* and two constants

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

Untyped $\lambda$-terms are encoded as terms of type *tm* using the translation define as

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

The first clause here indicates that bound variables in untyped $\lambda$-terms are mapped to bound variables in the encoding. For example, the untyped $\lambda$-term $\lambda w.ww$ is encoded as $(abs\ \lambda w.\ app\ w\ w)$. This translation has the property that it maps bijectively $\alpha$-equivalence classes of untyped $\lambda$-terms to $\alpha\beta\eta$-equivalence classes of simply typed $\lambda$-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), $\Delta$ is the antecedent (a multiset of formulas), and $\Sigma$ 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 $\Sigma$. Gentzen referred to the variables in $\Sigma$ as *eigenvariables* (although he did not consider them as binders over sequents).

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 (when reading it from conclusions to premise) that if the symbol $x$ can be attributed the type $\tau \to \tau$ for all instances of $\tau$, 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 inferences.

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

These inferences illustrate 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.

The mobility of bindings needs to be supported by the equality theory of expressions. Clearly, equality already includes $\alpha$-conversion by Property 1. We also need a small amount of $\beta$-conversion. If we rewrite this last inference rule using the definition of the $\lceil\cdot\rceil$ translation, we have the inference figure.

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

Note that here $B$ is a variable of arrow type $\mathit{tm} \to \mathit{tm}$ and that instances of these inference figures will create an instance of $(B\ x)$ that may be a $\beta$-redex. As I now argue, that $\beta$-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, $\lambda$Prolog, 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 instance of $(B\ x)$ that is a $\beta$-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 $\beta$-conversion that is needed to support this notion of binding mobility is the so-called $\beta_0$-conversion rule [50]: $(\lambda x.t)x = t$ or equivalently (in the presence of $\alpha$-conversion) $(\lambda y.t)x = t[x/y]$, provided that $x$ is not free in $\lambda y.t$.

Given that $\beta_0$-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 $\alpha$, $\beta_0$, and $\eta$ conversion, is decidable and unitary [50]. 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 $\beta_0$-reduction: if $t$ is a term over the signature $\Sigma \cup \{x\}$ then $\lambda x.t$ is a term over the signature $\Sigma$ and, conversely, if $\lambda x.s$ is a term over the signature $\Sigma$ then the $\beta_0$-reduction of $((\lambda x.s)\ y)$ is a term over the signature $\Sigma \cup \{y\}$.

To illustrate how $\beta_0$-conversion supports the mobility of binders, consider how one specifies the following rewriting rule: given a conjunction of universally quantified formulas, rewrite it to be the universal quantification of the conjunction of formulas. In this setting, we would write something like:

$$(\forall(\lambda x.A\ x)) \wedge (\forall(\lambda x.B\ x)) \mapsto (\forall(\lambda x.(A\ x \wedge B\ x))).$$

To rewrite an expression such as $(\forall\lambda z(p\ z\ z)) \wedge (\forall\lambda z(q\ a\ z))$ (where $p$, $q$, and $a$ are constants) we first need to use $\beta_0$-expansion to get the expression

$$(\forall\lambda z((\lambda w.(p\ w\ w))z)) \wedge (\forall\lambda z((\lambda w.(q\ a\ w))z))$$

At this point, the pattern variables $A$ and $B$ in the rewriting rule can now be instantiated by the *closed* terms $\lambda w.(p\ w\ w)$ and $\lambda w.(q\ a\ w)$, respectively, which yields the expression

$$(\forall(\lambda x.((\lambda w.(p\ w\ w))\ x \wedge (\lambda w.(q\ a\ w))\ x))).$$

Finally, a $\beta_0$-contraction yields the expected expression $(\forall(\lambda x.(p\ x\ x) \wedge (q\ a\ x)))$. Note that at no time did a bound variable become unbound. Since pattern unification incorporates $\beta_0$-conversion, such rewriting can be accommodated simply by calls to such unification.

The analysis of these four principles above do not imply that full $\beta$-conversion is needed to support them. Clearly, full $\beta$-conversion will implement $\beta_0$-conversion and several systems (which we shall speak about more below) that support $\lambda$-tree syntax do, in fact, implement $\beta$-conversion. Systems that only implement $\beta_0$-conversion have only been described in print. For example, the $L_\lambda$ logic programming language of [50] was restricted so that proof search could be complete while only needing to do $\beta_0$-conversion. The $\pi_I$-calculus (the $\pi$-calculus with internal mobility [74]) can also be seen as a setting where only $\beta_0$-conversion is needed [53].

## 6 Logic programming provides a framework

As the discussion above suggests, quantificational logic using the proof-search model of computation can 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 [49]), the logic programming paradigm supplies an appropriate framework for satisfying all these properties. Such a framework is available using the higher-order hereditary Harrop [54] subset of an intuitionistic variant of Church's Simple Theory of Types [16]: $\lambda$Prolog [53] is a logic programming language based on that logic and implemented by the Teyjus compiler [73] and the ELPI interpreter [24].

The use of logic programming principles in proof assistants pushes against usual practice: since the first LCF prover [37], many (most?) proof assistants have had intimate ties to functional programming. For example, such theorem provers are often implemented using functional programming languages: in fact, the notion of LCF tactics and tacticals was originally designed and illustrated using functional programming principles [37]. Also, such provers frequently view proofs constructively and can output the computational content of proofs as functional programs [10].

I argue here that a framework based on logic programming principles might be more appropriate for mechanizing metatheory than one based on functional programming principles. Note that the arguments below do not lead to the conclusion that first-order logic programming languages, such as Prolog, are appropriate for metalevel reasoning: direct support for $\lambda$-abstractions and quantifiers (as well as hypothetical reasoning) are critical and are not supported in first-order logic programming languages. Also, I shall focus on the *specification* of mechanized metatheory tasks and not on their *implementation*: it is completely possible that logic programming principles are used in specifications while a functional programming language is used to implement that specification language (for example, Teyjus and Abella are both implemented in OCaml).

### 6.1 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 expressions. 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 $\lambda$-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 $\lambda$-abstraction is just another expression: following the principles stated in Section 4, equality of two such expressions needs to be based on the rather simple set of conversion rules $\alpha$, $\beta_0$, and $\eta$. The $\lambda$-abstraction-as-expression aspect of logic programming is one of that paradigm's major advantages for the mechanization of metatheory.

## 6.2 Syntactic types

Given the central role of expressions (and not values), types in logic programming are better thought of as denoting *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 5 is a binary relation between the syntactic categories *tm* (for untyped $\lambda$-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

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

Given our discussion above, it seems natural to propose that if $\tau$ and $\tau'$ are both syntactic categories, then $\tau \to \tau'$ is a new syntactic category that describes objects of category $\tau'$ with a variable of category $\tau$ abstracted. For example, if $o$ denotes the category of formulas (a la [16]) 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 $(tm \to o) \to o$.

## 6.3 Substitution lemmas for free

Consider an attempt to prove the sequent

$$\Sigma : \Delta \vdash \mathit{typeof}\ (\mathit{abs}\ R)\ (\tau \to \tau')$$

where the assumptions (the theory) contains only one rule for proving such a statement, such as the clause used in the discussion of Section 5. Since the introduction rules for $\forall$ and $\supset$ are invertible, the sequent above is provable if and only if the sequent

$$\Sigma, x : \Delta, \mathit{typeof}\ x\ \tau \vdash \mathit{typeof}\ (R\ x)\ \tau'$$

is provable. 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

$$\Sigma : \Delta, \mathit{typeof}\ N\ \tau \vdash \mathit{typeof}\ (R\ N)\ \tau'$$

must be provable (for $N$ a term of syntactic type *tm* all of whose free variables are in $\Sigma$). Thus, we have just shown, using nothing more than rather minimal assumptions about the specification of *typeof* (and formal properties of logic) that if $\Sigma : \Delta \vdash \mathit{typeof}\,(\mathit{abs}\ B)\ (\tau \rightarrow \tau')$ and $\Sigma : \Delta \vdash \mathit{typeof}\ N\ \tau$ then $\Sigma : \Delta \vdash \mathit{typeof}\ (B\ N)\ \tau'$. (Of course, instances of the term $(B\ N)$ are $\beta$-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 [85]: in this setting, this lemma is essentially an immediate consequent of using logic and logic programming principles [8, 46]. In this way, Gentzen's cut-elimination theorem (the formal justification of modus ponens) can be seen as the mother of all substitution lemmas. The Abella theorem prover's implementation of the *two-level logic* approach to reasoning about computation [33, 48] makes it possible to employ the cut-elimination theorem in exactly the style illustrated above.

## 6.4 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 5 is a relation. Similarly, both small step (SOS) and big step (natural semantics) approaches to operational semantics describe evaluation, for example, as a relation. 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 [17] and λProlog has been used to provide convincing and elegant specifications of typing and operational semantics for expressions involving bindings [2, 53].

## 6.5 Dependent typing

The typing that has been motivated above is rather simple: one takes the notions of syntactic types as syntactic category—e.g., programs, formulas, types, terms, etc—and adds the arrow type constructor to denote abstractions of one syntactic type over another one. Since typing is, of course, an open-ended concept, it is completely possible to consider any number of ways to refine types. For example, instead of saying that a given expression denotes a term (that is, the expression has the syntactic type for terms), one could instead say that such an expression denotes, for example, a function from integers to integers. For example, the typing judgment $t : \mathit{tm}$ ("$t$ denotes a term") can be refined to $t : \mathit{tm}\ (\mathit{int} \rightarrow \mathit{int})$ ("$t$ denotes a term of type $\mathit{int} \rightarrow \mathit{int}$). Such richer types are supported (and generalized) by the *dependent type* paradigm [20, 38] and given a logic programming implementation in, for example, Twelf [64, 66].

Most dependently typed λ-calculi come with a fixed notion of typing and with a fixed notion of proof (natural deduction proofs encoded as typed λ-terms). The reliance described here on logical connectives and relations is expressive enough to specify dependently typed frameworks [26, 77, 78] but it is not committed to only that notion of typing and proof.

# 7 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" [65]. A subsequent paper identified HOAS as a technique "whereby variables of an object language are mapped to variables in the meta-language" [66]. 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 $\lambda$-calculus are often used). Second, since the metalanguage (often the programming language) can vary a great deal, the resulting notion of HOAS can vary similarly, including the case where HOAS is a representation of syntax that incorporates *function spaces* on expressions [22, 39]. Third, the adjective higher-order seems inappropriate here: in particular, the equality (and unification) of terms discussed in Section 5 is completely valid without reference to typing. If there are no types, what exactly is "higher-order"? For these reasons, the term "$\lambda$-tree syntax" [8, 53], with its obvious parallel to the term "parse tree syntax," has been introduced as a more appropriate term for the approach to syntactic representation described here.

While $\lambda$-tree syntax can be seen as a kind of HOAS (using the broad definition of HOAS given in [66]), there is little connections between $\lambda$-tree syntax and the problematic aspects of HOAS that arise when the latter uses function spaces to encode abstractions. For example, there are frequent claims that structural induction and structural recursive definitions are either difficult, impossible, or semantically problematic for HOAS: see, for example, [29, 39, 40]. When we consider specifically $\lambda$-tree syntax, however, induction (and coinduction) and structural recursion in the $\lambda$-tree setting have been given proof theoretic treatments and implementations.

## 8 Reasoning with $\lambda$-tree syntax

Proof search (logic programming) style implementations of specifications can provide simple forms of metatheory reasoning. For example, given the specification of typing, both type checking and type inference are possible to automate using unification and backtracking search. Similarly, a specification of, say, big step evaluation can be used to provide a symbolic evaluator for at least simple expressions [17].

There is, however, much more to mechanizing metatheory than performing unification and doing logic programming-style search. One must also deal with negations (difficult for straightforward logic programming engines): for example, one wants to prove that certain terms do not have simple types: for example,

$$\vdash \neg\, \exists \tau : ty.\ \mathit{typeof}\ (\mathit{abs}\ \lambda x\ (\mathit{app}\ x\ x))\ \tau.$$

Proving that a certain relation actually describes a (partial or total) function has proved to be an important kind of metatheorem to prove: the Twelf system [66] is able to automatically prove many of the simpler forms of such metatheorems. Additionally, one should also deal with induction and coinduction and be able to reason directly about, say, bisimulation of $\pi$-calculus expressions as well as confluence of $\lambda$-conversion.

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 [47, 82] have made it possible to build automated and interactive inductive and coinductive theorem provers in a simple, relational setting. Another extension [32, 55] introduced the $\nabla$-quantifier which allows logic to reason in a rich and natural way with bindings: in terms of mobility of bindings, the $\nabla$-quantifier provides an additional formula-level and proof-level binder, 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 [8, 30], and an automatic theorem prover, called Bedwyr

[9], that unfolds fixed points in a style similar to a model checker. These systems have successfully been able to prove a range of metatheoretic properties about the $\lambda$-calculus and the $\pi$-calculus [1, 8, 81]. The directness and naturalness of the encoding for the $\pi$-calculus bisimulation is evident in the fact that simply adding the excluded middle on name equality changes the interpretation of that one definition from open bisimulation to late bisimulation [81].

Besides the Abella, Bedwyr, and Twelf system mentioned above, there are a number of other implemented systems that support some or all aspects of $\lambda$-tree syntax: these include Beluga [67], Hybrid [27], Isabelle [62], Minlog [75], and Teyjus [60]. See [28] for a survey and comparison of several of these systems.

The shift from conventional proof assistants based on functional programming principles to assistants based on logic programming principles does disrupt a number of aspects of proof assistants. For example, when computations are naturally considered as functional, it seems that there is a lost of expressiveness and effectiveness if one must write those specifications using relations. Recent work shows, however, that when a relation actually encodes a function, it is possible to use the proof search framework to actually compute that function [34]. A popular feature of many proof assistants is the use of tactics and tacticals, which have been implemented using functional programs since their introduction [37]. There are good arguments, however, that those operators can be given elegant and natural implementations using (higher-order) logic programs [23, 25, 53]. The disruptions that result from such a shift seem well worth exploring.

## 9 Conclusions

I have argued that parsing concrete syntax into parse trees does not yield a sufficiently abstract representation of expressions: the treatment of bindings should be made more abstract. I have also described and motivated the $\lambda$-tree syntax approach to such a more abstract framework. For a programming language or proof assistant to support this level of abstraction in syntax, equality of syntax must be based on $\alpha$ and $\beta_0$ (at least) and must allow for the mobility of binders from within terms to within formulas (i.e., quantifiers) to within proofs (i.e., eigenvariables). I have also argued that the logic programming paradigm—broadly interpreted—provides an elegant and high-level framework for specifying both computation and deduction involving syntax containing bindings. This framework is offered up as an alternative to the more conventional approaches to mechanizing metatheory using formalizations based on more conventional mathematical concepts. While the POPLmark challenge was based on the assumption that increments to existing provers will solve the problems surrounding the mechanization of metatheory, I have argued and illustrated here that we need to make a significant shift in the underlying paradigm that has been built into today's most mature proof assistants.

## 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 *Lecture Notes in Computer Science*, pages 173–187. Springer, 2012.

2 Andrew W. Appel and Amy P. Felty. Polymorphic lemmas and definitions in $\lambda$Prolog 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 Aydemir, Stephan A. Zdancewic, and Stephanie Weirich. Abstracting syntax. Technical Report MS-CIS-09-06, University of Pennsylvania, 2009.

**5** 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 Lecture Notes in Computer Science, pages 50–65. Springer, 2005.

**6** Brian E. Aydemir, Aaron Bohannon, and Stephanie Weirich. Nominal reasoning techniques in Coq: (extended abstract). *Electr. Notes Theor. Comput. Sci*, 174(5):69–77, 2007.

**7** David Baelde. Least and greatest fixed points in linear logic. *ACM Trans. on Computational Logic*, 13(1), April 2012.

**8** 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*, 7(2), 2014.

**9** 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 Lecture Notes in Artificial Intelligence, pages 391–397, New York, 2007. Springer.

**10** Ulrich Berger, Stefan Berghofer, Pierre Letouzey, and Helmut Schwichtenberg. Program extraction from normalization proofs. *Studia Logica*, 82(1):25–49, 2006.

**11** Filippo Bonchi and Damien Pous. Checking NFA equivalence with bisimulations up to congruence. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 457–468. ACM, 2013.

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

**13** Kaustuv Chaudhuri, Matteo Cimini, and Dale Miller. A lightweight formalization of the metatheory of bisimulation-up-to. In Xavier Leroy and Alwen Tiu, editors, *Proceedings of the 4th ACM-SIGPLAN Conference on Certified Programs and Proofs*, pages 157–166, Mumbai, India, January 2015. ACM.

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

**15** 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.

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

**17** Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, Laurent Hascoët, and Gilles Kahn. Natural semantics on the computer. Research Report 416, INRIA, Rocquencourt, France, June 1985.

**18** Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

**19** Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

**20** N. G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606. Academic Press, New York, 1980.

**21** 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.

**22** Joëlle Despeyroux, Amy Felty, and Andre Hirschowitz. Higher-order abstract syntax in Coq. In *Second International Conference on Typed Lambda Calculi and Applications*, pages 124–138, April 1995.

**23** Cvetan Dunchev, Claudio Sacerdoti Coen, and Enrico Tassi. Implementing HOL in an higher order logic programming language. In Gilles Dowek, Daniel R. Licata, and Sandra Alves, editors, *Proceedings of the Eleventh Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP 2016, Porto, Portugal, June 23, 2016*, pages 4:1–4:10. ACM, 2016.

**24** Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. ELPI: fast, embeddable, λProlog interpreter. In Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, volume 9450 of *Lecture Notes in Computer Science*, pages 460–468. Springer, 2015.

**25** Amy Felty and Dale Miller. Specifying theorem provers in a higher-order logic programming language. In *Ninth International Conference on Automated Deduction*, number 310 in Lecture Notes in Computer Science, pages 61–80, Argonne, IL, May 1988. Springer.

**26** Amy Felty and Dale Miller. Encoding a dependent-type λ-calculus in a logic programming language. In Mark Stickel, editor, *Proceedings of the 1990 Conference on Automated Deduction*, volume 449 of *Lecture Notes in Artificial Intelligence*, pages 221–235. Springer, 1990.

**27** Amy Felty and Alberto Momigliano. Hybrid: A definitional two-level approach to reasoning with higher-order abstract syntax. *J. of Automated Reasoning*, 48:43–105, 2012.

**28** Amy P. Felty, Alberto Momigliano, and Brigitte Pientka. The next 700 challenge problems for reasoning with higher-order abstract syntax representations: Part 2–A survey. *J. of Automated Reasoning*, 55(4):307–372, 2015.

**29** M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax involving binders. In *14th Symp. on Logic in Computer Science*, pages 214–224. IEEE Computer Society Press, 1999.

**30** 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 *Lecture Notes in Computer Science*, pages 154–161. Springer, 2008.

**31** Andrew Gacek, Dale Miller, and Gopalan Nadathur. Reasoning in Abella about structural operational semantics specifications. In A. Abel and C. Urban, editors, *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2008)*, number 228 in Electronic Notes in Theoretical Computer Science, pages 85–100, 2008.

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

**33** Andrew Gacek, Dale Miller, and Gopalan Nadathur. A two-level logic approach to reasoning about computations. *J. of Automated Reasoning*, 49(2):241–273, 2012.

**34** Ulysse Gérard and Dale Miller. Separating functional computation from relations. In Valentin Goranko and Mads Dam, editors, *26th EACSL Annual Conference on Computer Science Logic (CSL 2017)*, volume 82 of *LIPIcs*, pages 23:1–23:17, 2017.

**35** Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte der Mathematischen Physik*, 38:173–198, 1931. English Version in [84].

36 M. J. C. Gordon and T. F. Melham. *Introduction to HOL – A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

37 Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.

38 Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

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

40 Furio Honsell, Marino Miculan, and Ivan Scagnetto. $\pi$-calculus in (co)inductive type theories. *Theoretical Computer Science*, 2(253):239–285, 2001.

41 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.

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

43 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.

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

45 Per Martin-Löf. *Intuitionistic Type Theory*. Studies in Proof Theory Lecture Notes. Bibliopolis, Napoli, 1984.

46 Raymond McDowell and Dale Miller. A logic for reasoning with higher-order abstract syntax. In Glynn Winskel, editor, *12th Symp. on Logic in Computer Science*, pages 434–445, Warsaw, Poland, July 1997. IEEE Computer Society Press.

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

48 Raymond McDowell and Dale Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Trans. on Computational Logic*, 3(1):80–136, 2002.

49 Dale Miller. An extension to ML to handle bound variables in data structures: Preliminary report. In *Proceedings of the Logical Frameworks BRA Workshop*, pages 323–335, Antibes, France, June 1990. Available as UPenn CIS technical report MS-CIS-90-59.

50 Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. of Logic and Computation*, 1(4):497–536, 1991.

51 Dale Miller. Bindings, mobility of bindings, and the $\nabla$-quantifier. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *18th International Conference on Computer Science Logic (CSL) 2004*, volume 3210 of *Lecture Notes in Computer Science*, page 24, 2004.

52 Dale Miller. Finding unity in computational logic. In *Proceedings of the 2010 ACM-BCS Visions of Computer Science Conference*, ACM-BCS '10, pages 3:1–3:13. British Computer Society, April 2010.

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

54 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.

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

56 Robin Milner. *Communication and Concurrency*. Prentice-Hall International, 1989.

**57** Robin Milner. *Communicating and Mobile Systems: The $\pi$-Calculus*. Cambridge University Press, New York, NY, USA, 1999.

**58** Alberto Momigliano and Alwen Tiu. Induction and co-induction in sequent calculus. In Mario Coppo, Stefano Berardi, and Ferruccio Damiani, editors, *Post-proceedings of TYPES 2003*, number 3085 in Lecture Notes in Computer Science, pages 293–308, January 2003.

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

**60** Gopalan Nadathur and Dustin J. Mitchell. System description: Teyjus — A compiler and abstract machine based implementation of $\lambda$Prolog. In H. Ganzinger, editor, *16th Conf. on Automated Deduction (CADE)*, number 1632 in Lecture Notes in Artificial Intelligence, pages 287–291, Trento, 1999. Springer.

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

**62** Lawrence C. Paulson. A generic tableau prover and its integration with isabelle. *J. UCS*, 5(3):73–87, 1999.

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

**64** Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon D. Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

**65** 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.

**66** 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 Lecture Notes in Artificial Intelligence, pages 202–206, Trento, 1999. Springer.

**67** Brigitte Pientka and Joshua Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In J. Giesl and R. Hähnle, editors, *Fifth International Joint Conference on Automated Reasoning*, number 6173 in Lecture Notes in Computer Science, pages 15–21, 2010.

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

**69** François Pottier. An overview of C$\alpha$ml. In *ACM Workshop on ML*, Electronic Notes in Theoretical Computer Science, pages 27–51, September 2005.

**70** Damien Pous. Weak bisimulation upto elaboration. In C. Baier and H. Hermanns, editors, *CONCUR*, volume 4137 of *Lecture Notes in Computer Science*, pages 390–405. Springer, 2006.

**71** Damien Pous. Complete lattices and upto techniques. In Zhong Shao, editor, *APLAS*, volume 4807 of *Lecture Notes in Computer Science*, pages 351–366, Singapore, November 2007. Springer.

**72** Damien Pous and Davide Sangiorgi. Enhancements of the bisimulation proof method. In Davide Sangiorgi and Jan Rutten, editors, *Advanced Topics in Bisimulation and Coinduction*, pages 233–289. Cambridge University Press, 2011.

**73** Xiaochu Qi, Andrew Gacek, Steven Holte, Gopalan Nadathur, and Zach Snow. The Teyjus system – version 2, 2015. `http://teyjus.cs.umn.edu/`.

**74** Davide Sangiorgi. $\pi$-calculus, internal mobility and agent-passing calculi. *Theoretical Computer Science*, 167(2):235–274, 1996.

**75** Helmut Schwichtenberg. Minlog. In Freek Wiedijk, editor, *The Seventeen Provers of the World*, volume 3600 of *Lecture Notes in Computer Science*, pages 151–157. Springer, 2006.

**76** 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.

**77** Zachary Snow, David Baelde, and Gopalan Nadathur. A meta-programming approach to realizing dependently typed logic programming. In Temur Kutsia, Wolfgang Schreiner, and Maribel Fernández, editors, *ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 187–198, 2010.

**78** Mary Southern and Kaustuv Chaudhuri. A two-level logic approach to reasoning about typed specification languages. In Venkatesh Raman and S. P. Suresh, editors, *34th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 29 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 557–569, New Delhi, India, December 2014. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

**79** Alwen Tiu. Model checking for $\pi$-calculus using proof search. In Martín Abadi and Luca de Alfaro, editors, *Proceedings of CONCUR'05*, volume 3653 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 2005.

**80** Alwen Tiu and Dale Miller. A proof search specification of the $\pi$-calculus. In *3rd Workshop on the Foundations of Global Ubiquitous Computing*, volume 138 of *ENTCS*, pages 79–101, 2005.

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

**82** Alwen Tiu and Alberto Momigliano. Cut elimination for a logic with induction and co-induction. *Journal of Applied Logic*, 10(4):330–367, 2012.

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

**84** 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.

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