Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

Mechanized metatheory revisited*

Dale Miller
Draft: November 16, 2016

Abstract Proof assistants and theorem provers implement the meta-theory of logical systems, which implies that they must deal with a range of linguistic expressions (e.g., types, formulas, and proofs) that involve variable bindings. Since most mature proof assistants do not have built-in methods to treat bindings, they have been extended with various packages and libraries that allow them to encode such syntax using, for example, de Bruijn numerals. We 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 via packages and libraries. We present an approach to designing programming languages and proof assistants that supports bindings directly in syntax by accounting for them within the pervasive notion of term equality. However, more than enhancing equality is needed: one must also support the *mobility* of binders between the term-level bindings, formula-level bindings (quantifiers), and proof-level bindings (eigenvariables). A coherent semantics for such an approach can be described using the proofs that result from a combination of Church's approach to terms and formulas (found in his Simple Theory of Types) and Gentzen's approach to sequent calculus proofs. We will illustrate how such a framework provides an effective and semantically clean treatment of both computation on and reasoning with syntax containing bindings. Some implemented systems that support this approach to binding will be briefly described.

Contents

* The material in this paper is compiled from talks given by the author at the following meetings: CL 2000: Computational Logic, TPHOLS 2003, CSL 2004, Workshop on Algebraic Process Calculi: The First Twenty Five Years and Beyond (2005), IJCAR 2006, SOS 2008, ACM-BCS Visions of Computer Science Conference 2010, APLAS 2010, and TYPES 2016.

Inria-Saclay and LIX, Ecole Polytechnique, Palaiseau, France

To Do:

- Should we state (the obvious) somewhere? A function can be made relational in rather direct ways. Of course, relations can be made into set-valued functions but many popular specification frameworks rely on relations.
- Do I really have a DEFINITION of lambda tree syntax. Is there something quotable here.
- Theme: push on *mobility of bindings*, lambda-tree syntax, and nabla. Make the claim that the treatment of binding is intimate, not something that should be added afterwards.
- Missing citations to OTT and
  - the two special issues in JAR about the poplmark/bound variable issue.
  - Explain the differences with Hybrid (a Coq/Isabelle library) and the work here. Consider citing capretta09lc and several other hybrid papers.
  - There are now two papers on ELPI (lpar 2015 and lfmtp 2016).
- Explain equality-left: The proof theory approach to reasoning moves from turning simple failures into successes by using the decision process that is (first-order) unification and flipping a failure (to unify) into a success for proof building. From this, much richer aspects of negation can be established.
- To keep the notion of indeterminates, a new quantifier is introduced. It coincides with forall when there are "no negations", etc. Should I try to link this discussion to Selinger's paper on indeterminates in the semantics of the lambda-calculus?
- Argument often argued: mathematicians treat syntax like this so we do too. But CS can teach mathematicians a thing or two.
- Declarative approaches can mean that the interaction of features is clear: abstract datatypes, modules, higher-order programming are all based on abstractions and these all fit together into one logic / framework with a uniform presentation.
- Great things can happen. Predict π_I. The differences between closed and open bisimulation is a difference between classical and intuitionistic logic (such as excluded middle on equality of names. Also, on page 19 of [132] with Alwen, we show that there are three ways to move from the box modal for "free input action" to modals for "bound input action.
- Modularity of logic: if bindings are not in syntax, then a first-order logic (without raised quantifiers and without nabla) arises from this story.
- Cite? "Encoding a dependent-type lambda-calculus in a logic programming language by A Felty, D Miller (CADE)
- Mention PHOAS [24].
- There was also a proposal to have two different arrow types in ML, one denoting the syntactic abstraction and one function abstraction [73].
- One also has papers such as [58] which provides a "semantic analysis of higher-order abstract syntax": that analysis is only concerned with the abstraction-as-function interpretation. Semantic analysis of programming with λ-tree syntax is likely to be given by Kripke-style models such as those found in [90, 75].]]
- Should I mention the presentation of Peter Selinger on indeterminates and the eta-rule. A proper treatment of the lambda-calculus needs a means for extending signatures. Or adding indeterminates (in Selinger's terminology). In the proof theory setting, these are eigenvariables. In proof theory, substitution for indeterminates follows from using cut-elimination: for him, it looks like its a universal property in the underlying category theory presentation.

- Make sure that I stress somewhere that: LP technology can implement, for example, evaluation and typing. For example, given the specification of typing, type checking and type inference is 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 [26].
- The following seems useful to say somewhere: The trap when thinking about induction and higher-order typing is to believe that natural deduction proofs are the only notion of proof: indeed, they make a poor choice of proof since hypothetical reasoning (implications in assumptions) are hard to make into inductive structures. When switching to sequent calculus, instead, inductive reasoning about the structure of provability in sequent calculus captures much more reasoning power. (papers by McDowell and Miller). Thus, we move from attempting to prove induction on $pv\ B$ to $\Gamma \vdash B$. Eigenvariables, however, are not captured in this inductive structure: they still have a hypothetical (contravariant) nature (not the parallels between implications and universal quantification). McDowell and Miller attempted an ad hoc approach that was later made into a logical approach with Tiu and Miller.

1 Metatheory and its mechanization

Mechanized theorem proving—in both its interactive and automatic forms—has been applied in a wide range of domains. A frequent use of theorem provers is to formally establish various properties of specific programs related to their correctness: e.g., prove that a given program correctly sorts a list and always terminates or prove that a given loop satisfies a given invariant.

A more niche domain to which theorem proving is being applied is that of the *metatheory* of programming languages. In this domain, one takes a formal definition of a particular programming language's static semantics (e.g., typing), dynamic semantics (e.g., evaluation), and translation semantics (e.g., compilation) and establishes properties about all programs in that programming language.

We list a few, typical examples of metatheorems that are commonly proved.

1. If evaluation attributes values U and V to program M, then U and V are equal. Thus, evaluation is a partial function.
2. If M is attributed the value V and it has the type A, then V has type A also. Thus, types are preserved when evaluating an expression.
3. Applicative bisimulation for the programming language is a congruence [1,60]. Thus, equational-style rewriting using applicative bisimulation preserves that relation.

A theorem prover that is used for proving such metatheorems must deal with structures that are linguistic in nature: that is, metatheorems often need to quantify over programs, program phrases, types, values, terms, and formulas. A particularly challenging aspect of linguistic expressions, one which separates them from other inductive data structures (such as lists and binary trees), is their incorporation of bindings.

In fact, a number of research teams have used proof assistants to formally prove 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, [64,65,67,92].

The authors of the POPLmark challenge [9] have pointed out that proving metatheorems about programming languages is often a difficult task given the proof assistants available at that time (in 2005). In particular, their experiments using various systems to work with on the metatheory of programming languages lead them to urged the developers of proof assistants to make improvements to their 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. [9]

These authors also acknowledge that 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 the decade following the POPLmark challenge, a number of approaches to representing syntax containing bindings have been proposed, analyzed, and applied to metatheory issues. These approaches go by names such as *locally nameless* [22], *nominal reasoning* [10,109,112,135], and *parametric higher-order abstract syntax* [24]. In the end, nothing canonical seems to have arisen: see [111,8] for detailed comparisons

between different representational approaches. On the other hand, most of these approaches have been used to take existing mature proof assistants, such as Coq or Isabelle, and extend them with new packages, new techniques, new features, and/or new front-ends.

The incremental extension of mature proof assistants is only one way to address this issue. In this paper, we propose another approach to mechanized metatheory and we use the following analogy to set the stage for that proposal.

Early implementations of operating systems and distributed systems forced programmers to deal with concurrency, a feature not present in early programming languages. Various treatments of concurrency and distribution were addressed by adding to mature programming languages thread packages, remote procedure calls, and/or tuple spaces. Such additions made important contributions to what computer systems could do in concurrent settings. None-the-less, early pioneers such as Dijkstra, Hoare, Milner, and Petri considered new ways to express and understand concurrency via formalisms such as CCS, CSP, Petri Nets, π-calculus, etc. These pioneers left the world of known and mature programs in an attempt to find natural and direct treatments of concurrent behavior. While the resulting process calculi did not provide a single, canonical approach to concurrency, their development and study have lead to much greater insight into computation, computation, and interaction.

In a similar spirit, we will examine here an approach to metatheory that is not based on extending mature theorem proving platforms. Instead, we look for means to compute and reason with bindings within syntax that arise directly from *logic* and *proof theory*, two topics that have a long tradition of allowing abstractions into the details of syntactic representations. There has been a large number of technical papers and a few implementations that provide an alternative approach to mechanized metatheory. The goal of this paper is not technical: instead, it is intended to provide an overview of this earlier work.

2 Dropping mathematics as an intermediate

Before directly addressing some of the computational principles behind bindings in syntax, it seems prudent to describe and challenge also the conventional design of a wide range of proof assistants.

The traditional approach to designing theorem provers to reason about computation employed in almost all ambitious proof assistants today follows the following two step approach [78].

Step 1: *Implement mathematics.* This step is achieved by picking a general, well understood formal system. Common choices are first-order logic, set theory [97], higher-order logic [25,50], or some foundation for constructive mathematics, such as Martin-Löf type theory [68,27,28].

Step 2: *Reduce reasoning about computation to mathematics.* Computational systems can be encoded via a model theoretic semantics (such as denotational semantics) or as an inductive definition over a proof system encoding, say, an operational semantics.

Placing (formalized) mathematics in the middle of this approach to reasoning about computational systems is problematic since traditional mathematical approaches assume *extensional* equality for sets and functions while computational settings may need

to distinquish such objects based on *intensional* properties. 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 of integers). In a purely extensional treatment, functions are represented directly and descriptions of algorithms 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 $\forall w.\ \lambda x.x \neq \lambda x.w$ is a theorem (assume that x and w are variables of some primitive type i). In a setting where λ-abstractions denote functions (the usual extensional treatment), this formula is equivalent to $\forall w_i.\neg\forall x.x = w$ 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 λ-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 $\lambda x.x$. Taking this latter step means, of course, separating λ-abstraction from the mathematical notion of function.

A key methodological element of this proposal is that we shall drop mathematics as an intermediate and attempt to find a direct and intimate connection between computation, reasoning, and logic.

Church's Simple Theory of Types [25] is one of the most significant and early steps taken in the design of a rich and expressive logic. In that paper, Church showed how it was possible to turn the tables on the usual presentation of terms and formulas in quantificational logic. Most presentations of quantification logic defined terms first and then formulas were defined to incorporate such terms (within atomic formulas). Church however defined the general notion of *simply typed λ-term* and defined formulas as a subset of such λ-terms, namely, those of type o. The resulting formal system provided an elegant way to reduce all formula-level bindings (e.g., the universal and existential quantifiers) to the term-level λ-binder. His approach also immediately captured the binders used in the definite description operators and Hilbert's ϵ-operator. Church's presentation of formulas and terms are used in many active computational logic systems such as the HOL provers [52], Isabelle [98], and λProlog [79].

Actually, Church's 1940 paper introduced what today is seen as *two* higher-order logics. Both of these logics are based on the same notion of term and formulas and use the same inference rules—namely, $\beta\eta$-conversion, substitution, modus ponens, and $\forall$-generalization—but use different sets of axioms.

The first of Church's logics is now often called *elementary type theory* (ETT) [5] and involves using only axioms 1-6 which involve the axioms for classical propositional logic as well as the basic rules for quantificational logic at higher-order (simple) types. The second of Church's logics is aforementioned *simple theory of types* (STT). This logic arises by adding to ETT axioms 7-11: these axioms guarantee the existence of a non-empty domain and of an infinite domain as well as contains the axioms of description and choice as well as extensionality for functions. Church's goal in strengthening ETT by adding these additional axioms was to position STT as a proper foundations for much of mathematics. Indeed, formal developments of significant parts of mathematics can be found in Andrews's textbook [6] and in systems such as HOL [50,57].

When we speak of dropping mathematics as an intermediate, it is at this point that we wish to rewind the steps taken by Church (and implementers of some proof assistants): for the task of mechanized metatheory, we wish to return to ETT and not accept all of the mathematics oriented axioms.

3 Elementary type theory

ETT is an appealing starting place for it parsimony in addressing both quantification and bindings in terms by mapping them both to binding in simply typed λ-calculus. Furthermore, the equality theory of ETT is that of α, β, and η-conversion is capable of providing both support for (higher-order) quantification as well as for terms containing bindings. Both alphabetic changes of bound variable names and Variable avoiding substitutions are all accounted for by the logical rules underlying ETT. The proof theory for ETT has been well developed for both both intuitionistic and classical variants of ETT (Church's original version was based on classical logic). Among the results know for ETT are cut-elimination [47, 116, 127], Herbrand's theorem and the soundness of Skolemization [72], completeness of resolution [4], and unification [62]. Subsets and variants of ETT have been implemented and employed in various computational logic systems. For example, the TPS theorem prover [84], the core of the Isabelle theorem prover [100], the logic programming language λProlog [79], and the proof system Minlog [124] are all based on various subsets of ETT. For more about the history of the automation of ETT and STT see the handbook article [17].

The simple types in ETT are best though of as *syntactic categories* and that the arrow type $\gamma \to \gamma'$ is the syntactic categories of abstractions of categories γ over γ'. Typing in this weak sense is essentially the same as Martin-Löf's notion of *arity types* [99]. This approach to syntactic categories is common within computational linguistics where, for example, a transitive verb is given the syntactic category $(S\backslash NP)/NP$ which is a sentence (S) that has two noun phrases (NP) abstracted and where the forward and backward slashes correspond to arrows indicating if the argument for and abstract should be found to the left or to the right of the transitive verb. In Church's logic, the type o (omicron) is the type of formulas: other primitive types provide for multisorted terms. For example, the universal quantifier $\forall_\gamma$ is not applied to a term of type γ and a formula (of type o) but rather to an abstraction of type $\gamma \to o$. Both $\forall_\gamma$ and $\exists_\gamma$ belong to the syntactic category $(\gamma \to o) \to o$. When using ETT to encode some object-level language, the terms and types of that language can be encoded as belonging to two different primitive types: the syntactic categories object-level term and object-level type.

Richer type systems, such as the dependently typed λ-calculi—known variously as LF, λP, and $\lambda\Pi$ [56, 16]—are also important in a number of computational logic systems, such as Coq [19], Agda [21], and Twelf [107], to name a few. Although we shall limit the type system of our meta-logic to be simple types, the intuitionistic variant of ETT is completely capable of faithfully encoding such dependently typed calculi [33, 126].

To be useful as the foundation of a mechanized metatheory, ETT needs extensions. For example, ETT does not directly offer induction and coinduction which are both clearly important for any logic hoping to prove metatheoretic results. Keeping close to a proof-theoretic presentation of ETT, Section 8 describe an extension to ETT in which term equality is treated as a *logical connective* (following the work by Schroeder-Heister [121] and Girard [48]) and inference rules for induction [70] and coinduction [12, 128, 133] are added. Section 9 presents an additional extension to the core of ETT with the addition of the ∇ quantifier [43, 83, 128].

In conclusion, we have explicitly ruled out Church's extension of ETT to STT as a proper foundation for our revisit to metatheory. Instead we shall illustrate that a separate extension to ETT—based on introducing inference rules for equality, fixed points,

and ∇-quantification—satisfy many of the needs for an expressible and implementable logic for mechanizing metatheory. It is important to note that while STT is equipped to deal with the mathematical notion of function (given the use of the definite description choice operator and extensionality), the extension to ETT we use here does not provide a rich notion of function. Instead, relations are used to directly encode computations and specifications. Of course, relations can encode functions—for example, the addition of two natural numbers would be a relation belonging to the syntactic category $nat \to nat \to nat \to o$—but the syntactic category $nat \to nat \to nat$ would not contain the usual functional notion of addition. Fortunately, meta-theory abounds with relations that may or may not be functional. For example, the relationship between a program and its type, between a process and its transitions, and between formula and a proof are all relations.

4 How abstract is your syntax?

Two of the earliest formal treatments of the syntax of logical expressions were given by Gödel [49] and Church [25] and, in both of these cases, their formalization involved viewing formulas as strings of characters. Even in the 1970's, one could find logicians using strings as representations of formulas: for example, in [4], an atomic formulas is defined as a formula-cum-string in which the leftmost primitive system which is not a bracket is a variable or parameter. Clearly, such a view of logical expressions contains too much information that is not semantically meaningful (e.g., white space, infix/prefix distinctions, brackets, 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). The *names* of bound variables is one form of "concrete nonsense" that generally remains in parse trees.

One way to get rid of bound variable names is to use de Bruijn's nameless dummy technique [30] 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 [30, 66, 95], 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 have no semantic context: they are artifact of how we write expressions and should not be present in abstracted syntax.

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: All term-level and formula-level bindings are encoded using a single binder.

With this principle, we are adopting Church's approach [25] to binding in logic, namely, that one has only λ-abstraction and all other bindings are encoded using that binder. For example, the universal quantified expression $(\forall x.\, B\, x)$ is actually broken into the expression $(\forall(\lambda x.\, B\, x))$, where $\forall$ is now treated as a constant of higher-type. Note that this latter expression is η-equivalent to $(\forall\ B)$ and universal instantiation of that quantified expression is simply the result of using λ-normalization on the expression $(B\ t)$. In this way, many details about quantifiers can be reduced to details about λ-terms.

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

This principle is taken from Alan Perlis's epigram 47 [103]. By accepting this principle, we recognize that bindings are never dropped to reveal a free variable. This principle also suggests the following, which is the main novelty in this list of principles.

Principle 4: Bindings have *mobility* and the equality theory of expressions must support such mobility [77, 79].

Since the other principles are most likely familiar to the reader, we describe in the next section this last principle in more detail.

An important cost in parsing the string representation of, say, formulas to an abstract syntax is the need to eventually produce a readable version of such abstracted syntax. Since white space is removed during parsing, printing can be a complex task of deciding on where new lines and indentations should be inserted. Similarly, when bound variable names are removed, deciding on how to name binding during printing is under constrained. We shall not consider here this problem with the printing of readable versions of abstract syntax.

Another consequence of the principles describe above is that we shall give up on the ability explicitly manipulate the names of bound variables. While some researchers feel that such manipulations are "of key paractical importance" [36], we have taken the opposite perspective: making it impossible to manipulate bound variable names explicitly forces our specifications to be more abstract and more close to the underlying semantics of computation systems. By analogy, when one programs in, say, modern functional programming languages, one gives up being able to explicitly manipulate the memory location associated to variables. While such abstracting variables from locations has allowed for a great many innovations in the execution of functional programming languages (such as optimizing compilers, parallel execution, etc), certain practical considerations made much harder (such as the implementation of garbage collection) Similarly, giving up on explicitly manipulating bound variable names causes some practical difficulties (such as those involved with printing abstract syntax), we shall argue here that the benefits of this particular abstraction are significant and interesting.

5 Mobility of bindings

Since the search for proofs is a key principle behind the proof search paradigm, we make use of Gentzen style sequents to encode the judgment that is the current attempt at being proved since such sequent explicitly maintain the "current set of assumptions and the current attempted consequence." For example, the sequent $\Delta \vdash B$ is the judgment

that states that B is a consequence of the assumptions in Δ. A literal translation of Gentzen's sequents makes us of free variables. In particular, when attempting to prove a sequent with a universal quantifier on the right, the corresponding right introduction rule employs an *eigenvariable*, that is a "new" or "fresh" variable. For example, in the inference figure

$$\frac{B_1, \ldots, B_n \vdash B_0[v/x]}{B_1, \ldots, B_n \vdash \forall x_\gamma . B_0} \ \forall\mathcal{R},$$

the variable v is not free in the lower sequent. Gentzen called such new variables *eigenvariables*. Unfortunately, written this way, this inference figure violates the Perlis principle (Principle 3 in Section 4). Instead, we augment sequents with a prefix Σ that collects eigenvariables and *binds* them over the sequent. The universal-right introduction rule now reads as

$$\frac{\Sigma, v\colon \gamma \ :: \ B_1, \ldots, B_n \vdash B_0[v/x]}{\Sigma \ :: \ B_1, \ldots, B_n \vdash \forall x_\gamma . B_0} \ \forall\mathcal{R},$$

where we assume that the eigenvariable signature contains always distinct variables (as is always possible given α-conversion for binding constructs). As a result, sequents contain both assumptions and eigenvariables as well as the target goal to be proved. We shall refer to eigenvariables as *sequent-level bindings*. (Ultimately, a second kind of sequent-level binding will be introduced in Section 9).

To illustrate the notion of binder mobility, consider specifying the typing relation that holds between untyped λ-terms and simple types. Since this problem deals with the two syntactic categories of expressions, we introduce two primitive types: tm is the type of terms encoding untyped λ-terms and ty is the type of terms encoding simple type expressions. Untyped λ-terms can be specified using two constants $abs\colon (tm \to tm) \to tm$ and $app\colon tm \to tm \to tm$ (note that there is no third constructor for treating variables). Untyped λ-terms are encoded as terms of type tm using the following translation function:

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

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. Simple type expressions can be encoded by introducing two constants, say $i\colon ty$ and $arrow\colon ty \to ty \to ty$. Let $of :: tm \to ty \to o$ be the type of the predicate encoding the type typing relation between untyped terms and simple types (following Church [25], we use the type o as the type of formulas).

The following inference rule is a familiar rule regarding typing.

$$\frac{\Sigma : \Delta, of\ t\ (arrow\ i\ i) \vdash C}{\Sigma : \Delta, \forall y (of\ t\ (arrow\ y\ y)) \vdash C} \ \forall L$$

This rule states (when reading it from premise to conclusion) that if the formula C follows from the assumption that t has type $(arrow\ i\ i)$ then C follows from strengthen that assumption to the assertion that t can be attributed the type $(arrow\ y\ y')$ for all instances of y. In this rule, the binding for y is instantiated: this inference rule is an example of Gentzen's rule for the introduction of the $\forall$ quantifier on the left.

On the other hand, consider the following inferences.

$$\dfrac{\dfrac{\Sigma, x : \Delta, of\ \lceil x \rceil\ y \vdash of\ \lceil B \rceil\ y'}{\Sigma : \Delta \vdash \forall x(of\ \lceil x \rceil\ y \supset of\ \lceil B \rceil\ y')}\ \forall R,\ \supset R}{\Sigma : \Delta \vdash of\ \lceil \lambda x.B \rceil\ (y \to y')}\ \text{backchaining}$$

These inferences 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 converted to a "free variable": it simply moves. The last inference rule is justified by backchaining with respect to a clause that we present below.

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 these last inference rules using the definition of the $\lceil \cdot \rceil$ translation, we have the following inference figures.

$$\dfrac{\dfrac{\Sigma, x : \Delta, of\ x\ y \vdash of\ (B\ x)\ y'}{\Sigma : \Delta \vdash \forall x(of\ x\ y \supset of\ (B\ x)\ y')}\ \forall R}{\Sigma : \Delta \vdash of\ (abs\ B)\ (arrow\ y\ y')}\ \text{backchaining}$$

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: that β-redex has, however, a greatly restricted form. In particular, 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 intuitionistic logic, then the following formula captures that quantification.

$$\forall B \forall y \forall y' [\forall x(of\ x\ y \supset of\ (Bx)\ y') \supset of\ (abs\ B)\ (arrow\ y\ y')]. \quad (*)$$

Also observe that the alternation of quantifiers implies that any instantiation of B leaves the β-redex $(B\ x)$ in the state where the argument x is not free in the instance of B: this is enforced by the fact that substitutions into formulas does not capture bound variables. 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, defined as as $(\lambda y.t)x = t[x/y]$, provided that x is not free in $\lambda y.t$. (Note that this conversion is equivalent to $(\lambda x.t)x = t$ in the presence of α-convergence.) The backchaining inference rule above is now justified was the act of backchaining [80] using this displayed formula (which is assumed to be present in the set of assumptions Δ).

Mobility of bindings is supported using β_0 since the internally bound variable y in the expression $(\lambda y.t)x$ is replaced by the externally bound variable x in the expression $t[x/y]$. Note that β_0 supports the following symmetric interpretation of λ-abstraction.

- If t is a term over the signature $\Sigma \cup \{x\}$ then λ-introduction yields the term $\lambda x.t$ which is a term over the signature Σ.
- If $\lambda x.s$ is a term over the signature Σ then the β_0 reduction of $((\lambda x.s)\ y)$ is a λ-elimination yielding $[x/y]t$, a term over the signature $\Sigma \cup \{y\}$.

Thus, β_0 reduction provides λ-abstraction with a rather weak form of functional interpretation: give a λ-abstraction and an increment to a signature, β_0 yields a term

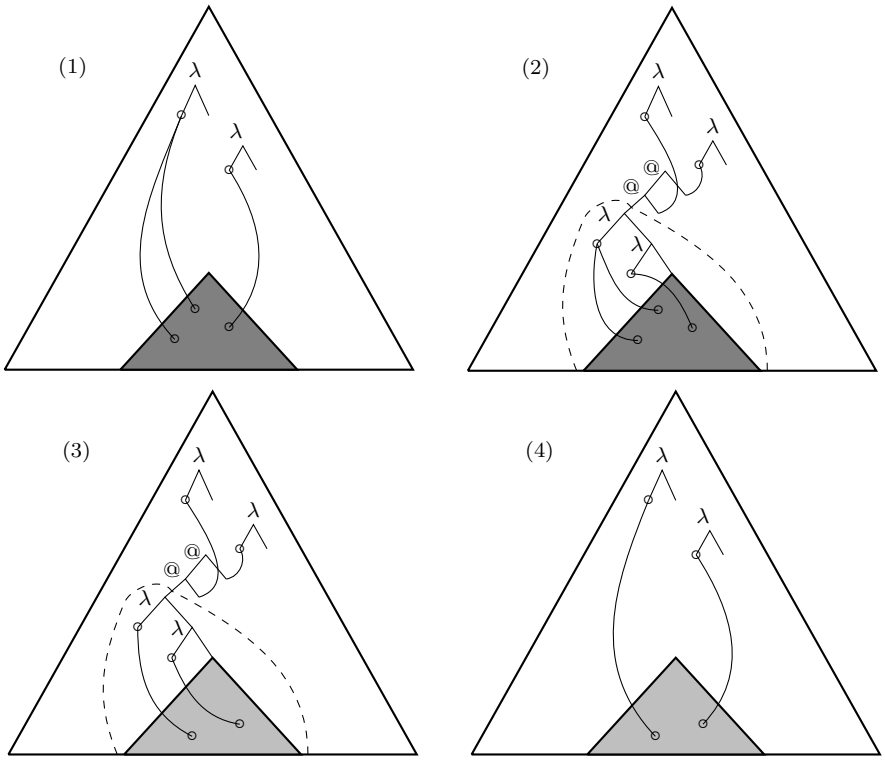

Fig. 1 Moving from (1) to (2) involves β_0 expansions; moving from (2) to (3) involves a rewriting of λ-abstracted terms; and moving from (3) to (4) involves β_0 contractions.

over the extended signature. The λ-abstraction has a dual interpretation since it takes a term over an incremented signature and hides that increment.

To further illustrate how β_0 conversion supports the mobility of binders, consider how one specifies the following rewriting rule: given a universal quantification of the conjunction of formulas, rewrite it to be the conjunction of universally quantified formulas. In this setting, we would write something like

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

where A and B are schema variables. To rewrite an expression such as $(\forall\lambda z(p\ z\ z \wedge q\ a\ z))$, we first need to use β_0 expansion to get the expression

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

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

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

Finally, a β_0 contraction yields the expected expression $(\forall(\lambda x.(p\ x\ x) \wedge (q\ a\ x)))$. Notice that at no time did a bound variable become unbound.

Figure 1 graphically illustrates this process of rewriting in the presence of bindings. Assume that we have a term (illustrated in (1) of Figure 1 as a large triangle) and that we wish to replace a subterm (the dark gray triangle) with another term (the light gray triangle in image (4)). Since the subterm in (1) contains occurrences of two bound variables, we write that subterm as $t(x, y)$ (where we assign the names x and y to those two bindings). When moving from image (1) to (2), we use β_0 expansion to replace replace $t(x, y)$ with $(\lambda u \lambda v.t(u, v))xy$. Notice now that the subterm $\lambda u \lambda v.t(u, v)$ is now closed and, as a result, it can be rewritten to, say $\lambda u \lambda v.s(u, v)$ (yielding (3)). Finally, β_0-reduction yields the term illustrated in (4). Thus, β_0 expansion and reduction allows a subterm be released from its dependency on bindings in its environment by making those dependencies into local bound variables. Of course, instead of rewriting simply t to s, we needed to rewrite the abstractions $\lambda u \lambda v.t(u, v)$ to $\lambda u \lambda v.s(u, v)$.

6 Proof search provides a framework

From a proof theoretic perspective, reasoning can be seen as a process that builds a (sequent calculus) proof. The cut rule (the use of both modus ponens and lemmas) is a dominate inference rule when reasoning is seen in this fashion [45]. The *proof search* approach to computation [80] is also such a process for building proofs but one that is governed by a *systematic* search for a *cut-free* proof. In general, cut-elimination is not part of these approaches to computation or reasoning. With the growing use of formal systems to encode aspects of mathematical reasoning, there are starting to appear some applications of cut-elimination within the reasoning process: consider, for example, *proof mining* where implicit but formal proofs can be manipulated to extract mathematically useful information [**?**]. In section 11, we shall provide a different set of examples where cut-elimination is used to formally reason about computations specified using the proof search paradigm.

One of the appealing aspects of using proof search to describe computation and reasoning is that it is possible to give a rich account of binder mobility (as illustrated in Section 5). Thus, this paradigm allows for specifying both recursive programming over data with bindings as well as reasoning inductively about such specifications. As such, proof search within ETT can accommodate all four principles dealing with abstract syntax that were listed in Section 4. To be clear, when we speak of the "logic programming" approach to specifications, we do not mean to include specific aspects of implementations of logic programming languages, such as Prolog or λProlog. For example, both of those languages make use of depth-first search: such a search regime is often inappropriate for automated reasoning.

The use of logic programming principles in proof assistants pushes against usual practice: since the first LCF prover [51], many (most?) proof assistants have had intimate ties to functional programming. For example, such theorem provers are often implemented using functional programming language: in fact, the notion of LCF tactics and tacticals was originally designed and illustrated using functional programming principles [51]. Also, such provers often view proofs constructively and can output the computational content of proofs as functional programs [18].

Most of the remainder of this paper provides an argument and some evidence that the proof search paradigm is an appropriate and appealing setting for mechanizing metatheory. Also, I shall focus on the *specification* of mechanized meta-theory 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, current implementations of Teyjus and Abella are written in OCaml [32,117]).

6.1 Expressions versus values

Keeping with the theme mentioned in Section 2 that types denote *syntactic types*, the terms of logic must then denote expressions. If we are representing terms without bindings, then terms denote themselves, in the sense of free algebras: for example, the equality $3 = 1 + 2$ fails to hold. While this is a standard expectation in the logic programming paradigm, the functional programming paradigm recognizes this equality as holding since, in that paradigm, expressions do not denote themselves but their *value*. That is, in the functional programming paradigm, if we wish to speak of expressions, we would need to introduce a datatype for abstract syntax (e.g., parse trees) and then one would have difference expressions for "three" and for "one plus two": in such a datatype, expressions and their value coincides.

The treatment of syntax with bindings within the functional programming paradigm is generally limited to two different but broad approaches. First, one can map binders in syntax to function abstractions: thus, abstract syntax may contain functions. We shall speak more about that approach to encoding in the functional programming setting in Section 7. Second, one can build a datatype denote syntax trees using different representations of bindings, such as names or de Bruijn's nameless dummies [30]. The implementer of such a datatype would also need to encode notions such as α-equality, free/bound distinctions, and capture avoiding substitution. Such an approach to encoding syntax with bindings is usually challenged when attempting to treat Principles 3 and 4 of Section 4. In order to support the notion that there are no free variables, contexts must be introduced and used as devices for encoding bindings: such bindings usually become additional data-structures and additional arguments and technical devices that must treated with care. With its formal treatment of contexts (based on Contextual Model Type Theory of [96]), the Beluga programming language [108] represents the state-of-the-art in this approach to syntax.

The logic programming paradigm with its emphasis on expressions instead of values provides another approach to treating syntax containing bindings that simply involves adopting an equality theory on expressions. In order to treat syntax with bindings at the proper level of abstraction, it is necessary to have the treatment of expressions In particular, by supporting both α-conversion and β_0-conversion it is possible for both Principle 1 and 4 to be supported. It has been know since the late 1980's that the logic programming paradigm can support the theory of α, η, and full β–conversions and, as such, it can support a suitably abstract approach to syntax with bindings: for example, the systems λProlog [79,93], Twelf [107], and Isabelle [101] provide such a proof search based approach to abstract syntax. While unification of simply typed λ-terms modulo $\alpha\beta\eta$ is undecidable in general [61], the systematic search for unifiers has been described [62]. It is also known that within the *higher-order pattern unification* restriction, unification modulo $\alpha\beta_0\eta$ is not only decidable and unary but it is also complete for unification modulo $\alpha\beta\eta$ [74]. This restricted form of unification is all that is needed to automatically support the kind of term processing illustrated in Figure 1.

6.2 Dominance of relational specifications

The focus of most efforts to mechanize meta-theory is to build tools to support programming language researchers and designer when they reason about the static and the dynamic semantic definitions of various specifications languages (such as the λ-calculus and the π-calculus) and programming languages. Static semantics is usually presented as a typing systems. Dynamic semantics will be given using either *small step* semantics, such as is used in structural operational semantics (SOS) [110], or as *big step* semantics, such as is used in natural semantic [63]. In all of these styles of semantic specifications, relations and not functions are the direct target of specifications. For example, the specification of proof systems and type systems use provability and typing judgments binary predicates such as $\Xi \vdash B$ or $T \colon \gamma$. A relation, such as $M \Downarrow V$, is also used when specifying the evaluation of, say, a functional program M to a value V. In case it holds that evaluation is a (partial) function, then it is a meta-theorem that

$$\forall M \forall V \forall V' \; [M \Downarrow V \wedge M \Downarrow V' \supset V = V']$$

Of course, if the programming language being considered involves communication with its environment, evaluation may not be functional in this simple sense. Relations and not functions are the usual specification vehicle for capturing a range of programming semantics.

For a concrete example, consider the specification of CCS where small step semantic specification are usually given by defining the ternary relation of labeled transition systems $P \xrightarrow{a} Q$ between two processes P and Q and an action a. For example, the usual SOS specification of labeled transitions for CCS contains the inference rules in Figure 2. The connection between those inference rules and logic programming clauses is transparent: in particular, the rules in Figure 2 can be written naturally as the the logic programming clauses in Figure 3. The close connection between such semantic specifications and logic programming allows for the immediate animation of such specifications using common logic programming interpreters. For example, both typing judgments and such operational semantic specifications have been animated via a Prolog interpreter in the Centaur project [26] and via a λProlog interpreter for computational systems employing binders [7, 53, 79].

The connection between semantic specifications and logic programs goes further than mere animation. Such logic programs can be taken as formal specifications about which it is possible to prove properties of the original. For example, logic programs have been systematically transformed in meaning perserving ways in order to prove that certain abstract machines implement certain simply functional programming languages [54, 55]. The Twelf system provided automated tools for reasoning about logic programs, thereby allowing direct proofs of, for example, progress theorems and type perservation [106, 122]. We shall illustrate a systematic approach to reasoning about logic programming specifications in Abella in Section 11.

6.3 Trading side conditions for more expressive logics

The inference rules used to specify both static semantics (e.g., typing) and dynamic semantics (e.g., small-step and big-step operational semantics) often contain an assortment of side conditions. Such side conditions can break and obscure the declarative

$$\frac{P \xrightarrow{A} P'}{P + Q \xrightarrow{A} P'} \qquad \frac{Q \xrightarrow{A} Q'}{P + Q \xrightarrow{A} Q'} \qquad \frac{P \xrightarrow{A} P'}{P \,|\, Q \xrightarrow{A} P' \,|\, Q} \qquad \frac{Q \xrightarrow{A} Q'}{P \,|\, Q \xrightarrow{A} P \,|\, Q'}$$

$$\frac{P \xrightarrow{A} P' \qquad Q \xrightarrow{\bar{A}} Q'}{P \,|\, Q \xrightarrow{\tau} P' \,|\, Q'}$$

Fig. 2 A few rules that can be a part of the formalization of labeled transitions for CCS. Tokens starting with a captial letter are schematic variables.

```
kind proc, act      type.

type tau            act.
type bar            act -> act.

type plus, par      proc -> proc -> proc.
type one            proc -> act  -> proc -> o.

one (plus P Q) A    P'           :- one P A P'.
one (plus P Q) A    Q'           :- one Q A Q'.
one (par  P Q) A    (par P' Q)   :- one P A P'.
one (par  P Q) A    (par P Q')   :- one Q A Q'.
one (par  P Q) tau  (par P' Q')  :- one P A P', one Q (bar A) Q'.
```

Fig. 3 The logic programming specification of SOS rules for CCS, written using the syntax of λProlog [79]. Here, the `kind` keyword declares `proc` and `act` as two syntactic categories denoting processes and actions, respectively. Tokens starting with a captial letter are variables that are universally quantified around the individual clauses.

nature of specifications: their presences can signal that a more expressive logical framework for specifications should be used.

The inference rules in Figure 2 for describing the transition system for CCS have no side conditions and their mapping into first-order Horn clauses (Figure 3) is unproblematic. Consider, however, some simple specifications regarding the untyped λ-calculus. The specification of call-by-value evaluation for untype λ-terms can be written as

$$\frac{M \Downarrow \lambda x.R \qquad N \Downarrow U \qquad S \Downarrow V}{(M\ N) \Downarrow V} \quad \text{provided } S = R[U/x].$$

There, the side condition requires that S is the result of substituting U for the free variable x in R. Similarly, when specifying a typing discipline on untyped λ-terms, we typically see specifications such as

$$\frac{\Gamma, x : \gamma \vdash t : \sigma}{\Gamma \vdash \lambda x.t : \gamma \to \sigma} \quad \text{provided } x \notin fn(\Gamma).$$

Here, the side condition specifies that the variable x is not free in the context Γ. In system such as π-calculus, which includes sophisticated uses of bindings, transition system come with numerous side conditions. Take, for example, the open inference rule [88]

$$\frac{P \xrightarrow{\bar{x}y} P'}{(y)P \xrightarrow{\bar{x}(w)} P'\{w/y\}} \quad \text{provided } y \neq x,\ w \notin fn((y)P').$$

Here the side condition has two conditions on variables name appearring in that inference rule.

As we shall see in Sections 8 and 12, the side conditions in the inference rules above can all be eliminated simply by encoding those rules in a logic richer than first-order Horn clauses. In particular, the logic underlying λProlog, the *hereditary Harrop formulas* [80] provide an immediate specification of these rule, in part, because the intuitionistic logic theory of hereditary Harrop formulas directly supports the λ-tree approach to syntax.

6.4 Substitution lemmas for free

One of the reason to use a logic to formalize static and dynamic semantic specifications is that that formalism—as an artifact—can have significant formal properties of its own. For example, proof search as a computation paradigm usually constructs *cut-free proofs* of the propositions on which it is asked to compute. A famous meta-theorem of intuitionistic logic is the *cut elimination* theorem of Gentzen [45]: if properly used, the cut-elimination theorem can be seen as the "mother of all substitution lemmas". An example of a *substitution lemma* is the following: if $\lambda x.B$ has type $\gamma \to \gamma'$ and N has type γ then the result of substituting N for x in B, i.e., $[N/x]B$, has type γ'. *** To illustrate this claim, we return to the specification of the *typeof* predicate given in Section 5. This binary relation relates 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

$$\forall T\colon tm\ \forall L\colon tm\ \forall y\colon ty\ [of\ T\ y \supset of\ L\ (list\ y) \supset of\ (T :: L)\ (list\ y)].$$

Consider an attempt to prove the sequent $\Sigma : \Delta \vdash of\ (abs\ R)\ (y \to y')$ where the assumptions (the theory) contains only one rule for proving such a statement (a typical assumption for the arrow type constructor) and that that assumption is 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

$$\Sigma, x : \Delta, of\ x\ y \vdash of\ (R\ x)\ y$$

is provable. Given that we are committed to using a proper logic (such as intuitionistic logic), it is the case that instantiating an eigenvariable in a provable sequent yields a provable sequent. Thus, the sequent

$$\Sigma : \Delta, of\ N\ y \vdash of\ (R\ N)\ y$$

must be provable. Thus, we have just shown, using nothing more than rather simple properties of logic that if

$$\Sigma : \Delta \vdash of\ (abs\ B)\ (y \to y') \quad \text{and} \quad \Sigma : \Delta \vdash of\ N\ y$$

then (using modus ponens) $\Sigma : \Delta \vdash of\ (B\ N)\ y'$. (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 [3,137]: in this setting, this lemma is essentially an immediate consequent of using logic and logic programming principles. In Section 11, we illustrate how the Abella theorem prover provides a general methodology that explicitly uses the cut-elimination theorem in this fashion.

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" [105]. It seems that few researchers currently use this term in a setting that includes products and polymorphism (although simple and dependently typed λ-calculus are often used). A subsequent paper identified HOAS as a technique "whereby variables of an object language are mapped to variables in the meta-language" [107]. While this definition of HOAS seems the dominate one in the literature, this term is problematic for at least two reasons.

First, this term is ambiguous since the metalanguage (often the programming language) can similarly vary a great deal. For example, if the metalanguage is a functional programming language or an intuitionistic type theory, the binding in syntax is usually mapped to the binding available for defining functions. In this setting, HOAS representation of syntax incorporates *function spaces* on expressions [31, 58] in order to define expressions. If the metalanguage is a logic programming language such as λProlog or Twelf, then the λ-abstraction available in those languages does not correspond to function spaces but to the weaker notion of hiding variables within a term, thereby producing a term of an abstracted syntactic type (see Section 2). Referring to these different approach to encoding syntax with the same term leads to misleading statements in the literature.

Second, the adjective "higher-order" is unnecessary. When applied in computational logic settings, the term "higher-order" is used in ambiguous ways [79]. Also underlying notion of equality and unification of terms discussed in Section 5 is completely valid without reference to typing and it is the order of a type that usually determines whether or not a variable is first-order or higher-order. When it comes to unification, in particular, it seems more appropriate to view pattern unification as a mild extension to first-order unification than it is to view it as an extreme restriction to "higher-order unification" (a.k.a. unification of simply typed λ-terms). For example, pattern unification can be applied to untyped λ-terms [74, Section 9.3]. Thus, if there are no types, and hence no types of "higher-order", why retain this adjective?

The ambiguity of the term HOAS causes confusion and misunderstanding. For example, the literature has statements such as the following.

- Referring to HOAS, the authors of [36] say that "[i]ts big drawback, in its original form at least, is that one looses the ability to define functions on syntax by structural recursion and to prove properties by structural induction—absolutely essential tools for our intended applications to operational semantics".
- In [118, p. 365], we find the statement that "higher-order abstract syntax used in a shallow embedding" when applied to "the π-calculus have been studied in Coq and λProlog. Unfortunately, higher-order datatypes are not recursive in a strict sense, due to the function in the continuations of binders. As a consequence, plain structural induction does not work, making syntax-analysis impossible. Even worse, in logical frameworks with object-level constructors, so-called *exotic* terms can be derived."

If not read carefully, these negative conclusions about HOAS can be interpreted as applying to all methods of encoding object-level bindings into a meta-level binding. Note that the use of the term "exotic" above applies to those encodings of syntax where rich function spaces are embedded within terms: while exotic terms can appear

in Coq encodings [31], they are not possible in λProlog since it contains *no function spaces*.

To avoid this ambiguous term, the term "λ-tree syntax" was introduced in [81]: with its obvious parallel to the term "parse tree syntax", this term seems to be a more appropriate to describe an approach to syntactic representation described in the previous sections and based on the notions of *syntactic categories*, β_0-conversion, and mobility of bindings. Thus, the term "λ-tree syntax" refers to the form of HOAS that is available in such logic programming settings as λProlog and Twelf. In those settings, it has long been known how to write relational specifications that compute by recursion over the syntax of expressions containing bindings. At the end of the 1990's, explicit reasoning about such relational specifications was part of the Twelf project [107] and was being developed for λProlog specifications following the "two-level logic approach" [69,71]. Still more sophisticated reasoning about relational specifications has been built into the Abella proof assistant where it is routine to prove inductive and coinductive theorems involving λ-tree syntax (see [13,38,44]). We describe Abella more in Section 10.

8 Computing and reasoning with λ-tree syntax

We illustrate here how the proof theory of intuitionistic logic with higher-order (but not predicate) quantification provides a rich computational setting for the direct manipulation of λ-tree syntax.

8.1 Relational specifications using λ-tree syntax

A common method to specify the call-by-value evaluation of untyped λ-terms is using *natural semantics* [63] (also known as big-step semantics). For example, the following two inference rules

$$\frac{}{\lambda x.R \Downarrow \lambda x.R} \qquad \frac{M \Downarrow \lambda x.R \quad N \Downarrow U \quad S \Downarrow V}{(M\ N) \Downarrow V} \text{ provided } S = R[N/x]$$

are commonly taken to be the definition of the binary relation $\cdot \Downarrow \cdot$ which relates two untyped λ-terms exactly when the second is the value (following the call-by-value strategy for reduction) of the first. Notice that rule for evaluating an application involves a side condition that refers to a (capture avoiding) substitution.

A typical formal representation of the untyped λ-calculus echos Scott's familiar way of encoding their semantics by introducing a suitable domain D and two retracts between $[D \to D]$ and D: in our more syntactic approach we introduce a new syntactic category, the type tm, and two constructors mimicking the two retracts $abs : (tm \to tm) \to tm$ and $app : tm \to (tm \to tm)$. This encoding is supported by the following correspondences: terms of type tm in $\beta\eta$-long normal form are in one-to-one correspondence with closed, untyped λ-terms modulo α-convergence.

Using such constructors, the inference rules displayed above can be written as

$$\frac{}{abs\ R \Downarrow abs\ R} \qquad \frac{M \Downarrow (abs\ R) \quad N \Downarrow U \quad (R\ U) \Downarrow V}{(app\ M\ N) \Downarrow V}$$

```
sig cbv.

kind  tm      type.
type  abs     (tm -> tm) -> tm.
type  app     tm -> tm -> tm.
type  eval    tm -> tm -> o.

module cbv.

eval (abs R) (abs R).
eval (app M N) V :- eval M (abs R), eval N U, eval (R U) V.
```

Fig. 4 A signature and a module file specifying call-by-value evaluation for the untyped λ-calculus.

In these inference rules, the schematic variables M, N, U, and V have type R while tm ranges over the syntactic category of $tm \to tm$. Notice that the formal substitution written as $S = R[N/x]$ above has been replaced by the application $(R\ N)$: since this term is a β-redex, the untyped λ-term that it corresponds to is the result of performing a β-reduction. Using quantification in ETT, it is natural to encode these two inference rules as the following Horn clauses in ETT.

$$\forall R\ (eval\ (abs\ R)\ (abs\ R))$$
$$\forall M \forall N \forall U \forall V \forall R\ (eval\ M\ (abs\ R) \wedge eval\ N\ U \wedge eval\ (R\ U)\ V \supset eval\ (app\ M\ N)\ V)$$

Here, the infix notation $\cdot \Downarrow \cdot$ is replaced by the prefixed symbol *eval* and the type of the quantified variables M, N, U, V, and R is the same as when they were used as schematic variables. Since the equality theory of ETT contains β-conversion, the term $(R\ U)$ might be a β-redex if R is an abstraction (that can always be assumed given the presence of η-conversion). In that case, the result of performing a β-reduction would result in the formal substitution of the argument U into the abstraction R, thereby correctly implementing the substitution proviso of the first pair of displayed inference rules above. The λProlog syntax for this specification is given in Figure 4: here kinds and types are explicitly declared in the signature part of specification and several usual logic programming conventions are used to displayed Horn clauses (upper case letters denote variables that are universally quantified around the full clause, conjunctions are written with a comma, and implication is written instead as `:-` denoting "implied-by".

8.2 A specification of object-level substitution and its correctness proof

The example above involving the specification of evaluation of untyped λ-terms illustrates how abstracted syntax can be used along with β-conversion in order to perform object-level substitution. As was illustrated above, the of full β-conversion present in ETT (and λProlog) makes it immediate to encode object-level substitution. We can, however, open up the blackbox that is β-conversion and write a simple specification of substitution using only relational specifications and the mobility of binders.

Figure 5 contains the specification of two predicates. In isolation, the `copy` predicate encodes equality in the following sense. Let $\mathcal{C}$ denote the set of clauses in Figure 5. The judgment $\mathcal{C} \vdash$ `copy` $M\ N$ is provable if and only if M and N are equal (that is, $\beta\eta$-convertible). The forward direction of this theorem can be proved by a simple

```
type tm                          tm -> o.
type copy                  tm -> tm -> o.
type subst (tm -> tm) -> tm -> tm -> o.

tm (app M N) :- tm M, tm N.
tm (abs R)   :- pi x\ tm x => tm (R x).

copy (app M N) (app P Q) :- copy M P, copy N Q.
copy (abs M)   (abs N)   :- pi x\ copy x x => copy (M x) (N x).

subst M T S :- pi x\ copy x T => copy (M x) S.
```

Fig. 5 A relational specification of object-level substitution.

```
kind ty      type.
type i, j    ty.
type arr     ty -> ty -> ty.
type of      tm -> ty -> o.

of (app M N) A          :- of M (arr B A), of N B.
of (abs R)   (arr A B) :- pi x\ of x A => of (R x) B.
```

Fig. 6 A relational specification of object-level typing.

induction on the uniform proof [80] of that judgment $\mathcal{C} \vdash$ `copy` M N. The converse is proved by induction on the structure of the $\beta\eta$-long normal form of terms of type tm. If the `copy` predicate is used hypothetically, as in the specification of the `subst` relation, then it can be used to specify substitution. The following is an immediate (and informal) proof of the following correctness statement for `subst`: $\mathcal{C} \vdash$ `subst` R M N is provable if and only if N is equal to the $\beta\eta$-long normal form of $(R\ M)$. The proof of the converse direction is, again, done by induction on the $\beta\eta$-long form of M (of type $tm \to tm$). The forward direction has an even more direct proof: since the only way one can prove $\mathcal{C} \vdash$ `subst` R M N is to prove $\mathcal{C}$, `copy` x $M \vdash$ `copy` $(R\ x)$ N, where x is a new (eigenvariable). Since instantiating an eigenvariable in a sequent with any term of the same type yields another provable sequent, then we know that $\mathcal{C}$, `copy` M $M \vdash$ `copy` $(R\ M)$ N is provable. By the previous theorem about `copy`, we also know that $\mathcal{C} \vdash$ `copy` M M holds and by the cut-rule of the sequent calculus (modus ponens), we know that $\mathcal{C} \vdash$ `copy` $(R\ M)$ N is provable which means (using again the theorem about `copy`) that N is equal to $(R\ M)$.

One of the keys to reasoning about relational specifications using logical specifications is the central use of sequent calculus judgments. For example, in the example above, we did not attempt to reason by induction on the provability of $\vdash$ `copy` M N but rather on the provability of $\Gamma \vdash$ `copy` M N for suitable context Γ.

8.3 The open-world and closed-world perspectives

As previous examples have illustrated, the specification of atomic formulas, such as `of M N` and `copy M N`, assume the *open world* assumption. For example, in order to prove `copy (abs R) (abs S)` from assumptions $\mathcal{C}$, the process of searching for a proof

generates a new member (an eigenvariable) of the type `tm`, say `c`, and add the formula `copy c c` to the set of assumptions $\mathcal{C}$. Thus, we view the type `tm` and the theory (the logic program) about members of that type as expandable. Such an *open world perspective* is common in relational specification languages that manipulate λ-tree syntax [56, 79, 80, 107].

The open-world perspective to specification has, however, a serious problem: in that setting, it is not generally possible to prove interesting negations. For example, one would certainly want to prove that self-application in the untyped λ-calculus does not have a simple typing: for example, our metalogic should be strong enough to prove

$$\Sigma : \mathcal{C} \vdash \neg\, \exists y : ty.\ of\,(abs\ \lambda x\ (app\ x\ x))\ y,$$

where Σ is the signature and $\mathcal{C}$ is the specification of the $(of\ \cdot\ \cdot)$ predicate in Figure 6. This not possible since intuitionistic logic satisfies the following monotonicity property: if $\mathcal{C}$ is a subset of $\mathcal{C}'$ and if $\Sigma : \mathcal{C} \vdash G$ then $\Sigma : \mathcal{C}' \vdash G$ (for any formula G). In particular, let $\mathcal{C}'$ extend $\mathcal{C}$ with the additional atomic formula $(of\,(abs\ \lambda x\ (app\ x\ x))\ c)$ for some constant c of type `ty`. If the negation above is provable then so too is

$$\Sigma : \mathcal{C}' \vdash \neg\, \exists y : ty.\ of\,(abs\ \lambda x\ (app\ x\ x))\ y,$$

which means that our logic specification for the predicate *of* is inconsistent, which is, in fact, not the case.

The contrast to the open-world perspective is the familiar closed-world perspective. Consider proving the theorem

$$\forall n[\ fib(n) = n^2\ \supset\ n \leq 20\ ],$$

where $fib(n)$ is the n^{th} Fibonacci number. Of course, we do not attempt a proof by assuming the existence of a new (non-standard) natural number n for which the n^{th} Fibonacci number is n^2. Instead, we prove that among the (standard) natural numbers, we find that there are only three values of n (0, 1, and 12) such that $fib(n) = n^2$ and that all three of those values are less than 20. The set of natural numbers is closed and induction allows us to prove such theorems about them.

Thus, it seems that in order to prove theorems about λ-tree syntax, we need both the open-world and the close-world perspectives: the trick is, of course, discovering how it is possible to accommodate these two contradictory perspectives at the same time.

8.4 Induction, coinduction, and λ-tree syntax

Since any discussion of mechanizing metatheory needs to have induction and coinduction reasoning principles, we shall assume that these are part of the logic we are using for reasoning. There are many ways to provide schemes for least and greatest fixed points within a proof theory setting. Gentzen's proof of the consistency of Peano arithmetic introduced natural number induction using a familiar notion invariant-based induction [46]. Both Schroeder-Heister [121] and Girard [48] considered approachs to adding fixed point unfolding to proof theory but neither of them considered the problem of *least* and *greatest* fixed points. A series of papers [12, 43, 70, 128, 133] has developed a proof-theoretic approach for both induction and coinduction within intuitionistic and linear logics. Based on such work, we assume that the metalogic used in the rest of this paper is an intuitionistic logic with both inference rules for induction and coinduction.

While we shall not describe the proof theory of that logic in detail here, we do mention the following.

- Inductive and coinductive definitions generally need to be stratified in some manner so that their use can be guaranteed to be consistent, which is usually shown by proving cut-elimination.
- Monotonicity of intuitionistic provability does not damage these approaches to the closed-world perspective since one views inductive definitions as outside of sequents by either making such definitions auxilary to the sequent calculus [121, 70] or by putting such definitions into the term structure of formulas via μ- and ν-expressions [12, 15]. In either case, augmenting assumptions (the left-hand side context of sequents) does not change the definitions of inductive predicates.

Given that we have adopted these strong principles in the logic, the closed-world perspective is enforced. How then can we recover the open-world perspective in this setting? We do this in two steps. First, we describe in Section 9 the ∇ (nabla) quantifier which reintroduces the notion of generic quantification critical for supporting the mobility of binder. Second, we describe in Section 11 the *two-level logic* approach to reasoning that allows us to embed within our reasoning logic an inductive data structure which encodes the sequent calculus of the logic that permits the open world perspective.

9 The ∇-quantifier

Consider the following problem about reasoning with an object-logic (taken from [82]). Let $\mathcal{H}$ be the set containing the following three (quantified) formulas.

$$\forall x \forall y[q\ x\ x\ y], \qquad \forall x \forall y[q\ x\ y\ x], \qquad \forall x \forall y[q\ y\ x\ x]$$

Here, q is a predicate constant of three arguments. The sequent

$$\mathcal{H} \longrightarrow \forall u \forall v[q\ \langle u, t_1\rangle\ \langle v, t_2\rangle\ \langle v, t_3\rangle]$$

is provable (in either Gentzen's LJ or LK sequent calculi [45]) from $\mathcal{H}$ only if terms t_2 and t_3 are *equal*. We can attempt to formalize this statement about object-level provability with the following kind of meta-level formula

$$\forall t_1 \forall t_2 \forall t_3 \big(\{\mathcal{H} \vdash \forall u \forall v[q\ \langle u, t_1\rangle\ \langle v, t_2\rangle\ \langle v, t_3\rangle]\} \supset t_2 = t_3\big).$$

We use the curly brackets here informally to denote the provability of object-level provability of the sequent it encodes. One can imagine trying to treat the object-level universal quantifiers as meta-level universal quantifiers, as in the following formulas.

$$\forall t_1 \forall t_2 \forall t_3 \big([\forall u \forall v\{\mathcal{H} \vdash (q\ \langle u, t_1\rangle\ \langle v, t_2\rangle\ \langle v, t_3\rangle)\}] \supset t_2 = t_3\big)$$

This second formula is only provable, however, if there are at least two different members of the underlying object-level type. That approach to proving this second formula is unfortunate since the original formula is provable without any assumptions on extensions of the object-level type. Thus, it seems to be a mistake to reduce these object-level universal quantifiers to the meta-level universal quantifier.

For a similar but simpler example, consider the ξ inference rule, often written as

$$\frac{t = s}{\lambda x.t = \lambda x.s}\ .$$

This inference rule violates the Perlis principle (Principle 3 in Section 4) since occurrences of x in the premise are free. If we fix this violation by inserting the universal quantifier into the rule

$$\frac{\forall x.t = s}{\lambda x.t = \lambda x.s}$$

then the equivalence $(\forall x.t = s) \equiv (\lambda x.t = \lambda x.s)$ holds. As we argued in Section 2, this equivalence is problematic for λ-tree syntax since we want $\forall w\ \neg(\lambda x.x = \lambda x.w)$ to be provable because it is impossible for there to be a (capture avoiding) substitution for w into $\lambda x.w$ that results in the term $\lambda x.x$. However, since this latter formula is equivalent to $\forall w\ \neg\forall x.x = w$ this (first-order) formula cannot be proved since it is false for a first-order model with a singleton domain.

The ∇-quantifier [82,83] provides an elegant logical treatment of these examples and it is the one logical feature that does not appear in conventional notions of computational logic. It is the case, however, that ∇-quantification is similar to the Gabbay and Pitt's freshness quantifier [37]: they are both self dual, i.e., $\nabla x \neg Bx \equiv \neg\nabla xBx$, and in weak settings (roughly Horn clauses), they coincide [40].

While this new quantifier can informally be described as providing a formalization of "newness" and "freshness" in a proof system, it is possible to describe it more formally using the theme of the mobility of binders. In particular, sequents are generalized from having one *global* signature (the familiar Σ) to also having several *local* signatures,

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

where σ_i is a list of variables, locally scoped over the formula B_i. The expression $\sigma_i \triangleright B_i$ is called a *generic judgment*. The ∇-introduction rule proof for binder mobility with these local signatures.

$$\frac{\Sigma : (\sigma, x_\gamma) \triangleright B, \Gamma \vdash \mathcal{C}}{\Sigma : \sigma \triangleright \nabla x_\gamma.B, \Gamma \vdash \mathcal{C}}\ \nabla\mathcal{L} \qquad \frac{\Sigma : \Gamma \vdash (\sigma, x_\gamma) \triangleright B}{\Sigma : \Gamma \vdash \sigma \triangleright \nabla x_\gamma.B}\ \nabla\mathcal{R}$$

In these rules, the variable x is assumed to not occur in the local signature to which it is added: such an assumption is always possible since α-conversion is available for all term, formula, and sequent-level bindings. Apparently, the generic judgment $(x_1, \ldots, x_n) \triangleright t = s$ can be identified, at least informally with the generic judgment $\triangleright \nabla x_1 \cdots \nabla x_n.t = s$ and with the formula $\nabla x_1 \cdots \nabla x_n.t = s$. Since these introduction rules are the same on the left and the right, one expects that this quantifier is *self-dual*. Instead of listing all the other inference rules for formulas using this extended sequent, we simply note that the following equivalences involving logical connectives hold as well.

$$\begin{array}{cc}
\nabla x \neg Bx \equiv \neg\nabla xBx & \nabla x(Bx \wedge Cx) \equiv \nabla xBx \wedge \nabla xCx \\
\nabla x(Bx \vee Cx) \equiv \nabla xBx \vee \nabla xCx & \nabla x(Bx \Rightarrow Cx) \equiv \nabla xBx \Rightarrow \nabla xCx \\
\nabla x \forall y Bxy \equiv \forall h \nabla xBx(hx) & \nabla x \exists y Bxy \equiv \exists h \nabla xBx(hx) \\
\nabla x \forall y Bxy \Rightarrow \forall y \nabla xBxy & \nabla x.\top \equiv \top, \quad \nabla x.\bot \equiv \bot
\end{array}$$

Note that the scope of a ∇ quantifier can be moved in over all propositional connectives. Moving it's scope below the universal and existential quantifier requires the familiar

notion of *raising* [76]: that is, the when ∇ moves inside a quantified expression, the type of the quantified variable must be raised by the type of the ∇-quantified variable.

The ∇-quantifier is the missing quantifier for formulating the ξ-rule: that is, the rule can now be written as

$$\frac{\nabla x.t = s}{\lambda x.t = \lambda x.s}.$$

Using this inference rule, the following three formulas are equivalence.

$$\forall w\ \neg(\lambda x.x = \lambda x.w) \qquad \forall w\ \neg\nabla x.x = w \qquad \forall w\ \nabla x.x \neq w.$$

All of these formulas are provable using the rules for ∇ presented above.

As mentioned in Section 8.4, the logic we are considering here does not contain non-logical (undefined) predicate symbols. Instead, all relations on which we wish to reason are defined as fixed point expressions. If the only thing that one does with fixed point expressions is to unfold them, then the initial rule, often written for Gentzen sequents as

$$\frac{}{\Sigma : \Gamma, A \vdash A}\ \mathit{initial}$$

is not, in fact, needed. Thus, it is the rules for equality (and the logical constants for true and false) that represent the leaves of a proof. As a result, it can be straightforward to extend sequent calculus proof search procedures to accommodate the ∇-quantifier. In particular, when the ∇ quantifier is encountered, the quantified variable is move to the corresponding local binding location; when existential and universal variables are encountered, these can be raised by the variables in the corresponding local context, and when an generic judgment involving only the equality is encountered, say, $(x_1, \ldots, x_n) \triangleright t = s$ then consider this a standard equality but of λ-terms, i.e., $\lambda x_1 \cdots \lambda x_n.t = \lambda x_1 \cdots \lambda x_n.s$. For example, the Bedwyr model checker [14] was extended to allow for the ∇-quantifier: the main challenge to such an implementation was the inclusion of (subsets of) simply typed λ-term unification [62, 74].

The story behind ∇ becomes a bit more complex when one strengthens the logic so that in addition to unfolding fixed points, induction and coinduction inference rules are used to provide for least and greatest fixed points. In those cases, the initial rule plays an important role that cannot be removed. The issue becomes: when is a sequent of the form $\Sigma : \Gamma, \sigma \triangleright A \vdash \sigma' \triangleright A'$ to be considered initial. There seems to be two natural approaches to defining the initial rule in the presence of generic judgments.

Minimal approach One approach declares $\Sigma : \Gamma, x_1, \ldots, x_n \triangleright A \vdash y_1, \ldots, y_m \triangleright A'$ to be initial exactly when $\lambda x_1 \cdots \lambda x_n.A$ and $\lambda y_1 \cdots \lambda y_m.A'$ are λ-convertible. Such a definition seems too strong, however, since the order of variables in two different local context does not seem important: in particular, it would seem natural that $\nabla x \nabla y.B$ should be logically equivalent to $\nabla y \nabla x.B$. This *minimal* approach was used and analyzed in [11]. In that setting, local signature contexts are allowed to exchange the order of their variables.

Nominal approach Besides exchange, it might also seem natural to allow a form of strengthening: that is, to allow the equivalence of $\nabla x.B$ with B whenever x is not free in B. A consequence of such an equivalence is that all types are non-empty. For example, the formula $\exists x_i.B$ is not provable if the type i does not contain any inhabitants. However, the formula $\nabla y_i \exists x_i.B$ might be provable: there is, at least, one inhabitant of

type i, namely, the nominal y. This kind of argument can easily be generalized to show that this strengthening equivalence implies that types for which one uses ∇ necessarily contain an infinite number of members. While Baelde argues [11] that certain adequacy issues can be complicated when strengthening is allow, the strengthening principle has been formally studied [41,39,43,130] and implemented into the Abella theorem prover [13]. The nominal approach also allows for a different way of writing local (generic) contexts within sequent. Via the strengthening rule, all local contexts can have the same number of variables (just add more to those that are shorter than the maximum length). Furthermore, all contexts can be variables with the same names (using α-conversion). In such a setting, then, instead of writing the many local signatures that are now all the same, we can write that local signature as if it is global (although acting locally). Such a conventions is taken, for example, in displaying sequents within the Abella prover.

One remaining feature which strengthens the integration of the ∇-quantifier with the rest of the (fixed point) logic is to allow ∇ *in the head* of fixed point definitions [41] or the roughly equivalent enrichment of term equality called *nominal abstractions* [43]. Instead of describing this extension here, we illustrate it in the next section.

10 The Abella theorem prover

Outline:

- illustrate some simple definitions and inductive reasoning
- Example: cbn or cbv definition. Theorem: cbv is functional. Theorem: the divergent operator has not cbv value.

Most of the proof theory principles and logic designs that we have motivated so far are implemented in the Abella interactive theorem prover. First implemented by Gacek in 2009 as part of his PhD [39], the system has attracted a number of additional developers and users. Abella is written in OCaml and the most recent versions of the system are available via GitHub and OPAM. A tutorial appears online as [13]. The logical foundation that is closest to that which is implemented is the logic $\mathcal{G}$ in [43]. The approach to induction and coinduction in Abella differs significantly with that based on proof theory: in particular, the proof theory of $\mathcal{G}$ requires entering invariants and co-invariants for induction and coinduction rules, which Abella leaves such invariants implement, opting for a natural and convenient kind of guarded circular reasoning.

11 The two-level logic approach

DM Mention that nabla does not always imply universal. But it is the case that this does hold for object-level proof system.

DM Mention that stratification in the specification language is not important: the seq judgment is always stratified. The seq predicate is a stratified inductive predicate in the reasoning logic that can be used to encode non-stratified inductive logic programs (theories) in the specification logic.

DM Explicitly define the seq predicate for a small logic. hH2 might be okay. Abella is more general.

Abella allows for the convenient and powerful reasoning about object-level reasoning using the so-called *two level logic* approach to reasoning about computation [44, 71]. In principle, Abella's least fixed point definitions are able to completely capture provability of hereditary Harrop formula within intuitionistic logic. Such a definition can capture provability of λProlog specifications, such as those given in Figures 4, 5, and 6. Furthermore, Abella can also have special tactics that allow for meta-theorems about the object-level logic to be supported directly: in particular, the cut-elimination theorem for the object-level logic is a powerful tactic, as we now illustrate.

Assume that the clauses in Figures 4 and 6 are gathered together into one logic program that is loaded into Abella. The formula `{eval M V}` denotes the Abella-level statement that the goal `eval M V` is provable from that one loaded logic program. More generally, the formula `{H1,...,Hn |- G}` denotes the fact that the object-logic sequent with formulas `H1,...,Hn` on the left and formula `G` on the right is provable in the object-logic. (When the left hand side of the sequent is empty, then the turnstile `|-` is not displayed.)

```
Theorem type_preserve :
  forall E V T, {eval E V} -> {of E T} -> {of V T}.

induction on 1. intros. case H1.
  search.
  case H2.
     apply IH to H3 H6. case H8. apply IH to H4 H7.
     inst H9 with n1 = U. cut H11 with H10.
     apply IH to H5 H12. search.
```

This is the complete proof script entered into Abella. While there is some similarities with Coq scripts, the `inst` and `cut` commands are specific to Abella and to its built-in support for a two-level logic reasoning. Just before the `inst` command is issued, the proof system of Abella appears as follows.

```
Variables: V T U R N M B
IH : forall E V T, {eval E V}* -> {of E T} -> {of V T}
H3 : {eval M (abs R)}*
H4 : {eval N U}*
H5 : {eval (R U) V}*
H6 : {of M (arr B T)}
H7 : {of N B}
H9 : {of n1 B |- of (R n1) T}
H10 : {of U B}
============================
 {of V T}
```

The list of variables are the eigenvariables that are bound in this sequent. The inductive hypothesis is labeled with `IH` and the asterisks one some of the assumptions are part

of Abella's approach to doing induction (about which we say no more here: see, [13, 39] for more).

Assumption `H9` captures the object-level provability judgment that for a fresh object-level eigenvariable `n1` (captured as a nominal variable), that the sequent with `of n1 B` on the left and `of (R n1) T` on the right is provable. The `inst H9 with n1 = U` is responsible for instantiating the nominal variable `n1` with the term `U` yielding the hypothesis

```
H11 : {of U B |- of (R U) T}
```

That is, since `H9` holds generically (that is, for a nominal constant `n1`) then it holds for every instant of that nominal constant. Similarly, the `cut` command applies that hypothetical with the assumption `H10` that *of* N U and this yields the following assumption (added to those above).

```
H12 : {of (R U) T}
```

Applying the inductive hypothesis `IH` to hypotheses `H5` and `H12` finally yields the desired goal.

The combination of the `inst` and `cut` commands provides an elegant and completely formal proof of the *substitute lemma* that states that if the type of `(abs R)` is the arrow type `(arr B T)` and if `U` has type `B` then the result of instantiating the abstract `(abs R)` with `U`, that is, `(R U)`, has type `T`. This substitution lemma employs the meta-theory of the object-level sequent calculus in order to make this kind of research now immediate.

The Abella system has been successfully used to prove a range of metatheoretic properties about well known formal systems. Complete formalizations for all the following topics can be found on the Abella web site or in cited papers.

- Untyped λ-calculus: Takahashi's proof of the Church-Rosser property using complete developments, a characterization of β-reduction via paths through terms; Loader's proof of standardization; type preservation of call-by-name and call-by-value for simple types and system F types; and Huet's proof of the cube property of λ-calculus residuals [2].
- Simply typed λ-calculus: Tait's logical relations argument for weak normalization and Girard's proof of strong normalization.
- Object-level proof systems: cut-elimination and the the completeness of a Hilbert-style proof system.
- Process calculi, particularly CCS and π-calculus.

The logic $\mathcal{G}$ [43] and the Abella theorem prover have been successfully at providing elegant and direct specifications of and formal proofs about many aspects of the π-calculus. In the next section, we focus on this formal treatment of this particular process calculus.

DM Accattoli's CPP paper argues that Abella can also give substitution-lemmas-for-cheap: they can be stated and proved directly and the lambda-tree syntax representation make it all simply.

12 A case study: the π-calculus

The π-calculus [87,88] is an interesting challenge for formalizations since its meta-theory must deal with not only bindings, substitution, and α-conversion but also with induction and coinduction. It also has a mature theory [89,120] which helps in developing and judging successful formalizations.

12.1 Encoding the syntax of the π-calculus

DM Put also here the OPEN inference rule: at the end of 6.3 (page 18) I put this inference rule (with nabla) and say that we will review it again. I should probably put the pi-calculus transition system in both lambda Prolog and Abella syntax: this should be a place where we can state that forall and nabla are interchangable (for may behavior). The lambda Prolog subsystem in Abella maps object-level forall to meta-level nabla.

In order to encode the π calculus processes, we introduce two primitive types denoting the syntactic categories for *processes* and *names* and we use the primitive types p and n for these. The syntax is the following:

$$P := 0 \mid \tau.P \mid x(y).P \mid \bar{x}y.P \mid (P \mid P) \mid (P + P) \mid (x)P \mid [x = y]P$$

There are two binding constructors here. The *restriction* operator $(x)P$ is encoded using a constant of type $(n \to p) \to p$. The *input* operator $x(y).P$ is encoded using a constant of type $n \to (n \to p) \to p$.

```
kind   n, p, a        type.   % Sorts for names, processes, actions

type   null           p.
type   bang, taup     p -> p.
type   match, out     n -> n -> p -> p.
type   plus, par      p -> p -> p.
type   nu             (n -> p) -> p.
type   in             n -> (n -> p) -> p.
```

In order to encode π-calculus transitions we introduce a new primitive type for the syntactic type of *action* expressions. There are three constructors for actions: τ : a for *silent* actions, $\downarrow$: $n \to n \to$ a for *input* actions, and $\uparrow$: $n \to n \to$ a for *output* actions.

$\downarrow xy$: a denotes the action of inputting y on channel x
$\uparrow xy$: a denotes the action of outputting y on channel x
$\uparrow x : n \to$ a denotes outputting of an abstracted name, and
$\downarrow x : n \to$ a denotes inputting of an abstracted variable.

```
type   tau            a.
type   dn, up         n -> n -> a.
```

```
oneb (in X M)     (dn X) M.                    % bound input
one  (out X Y P) (up X Y) P.                   % free output
one  (taup P) tau P.                           % tau
one  (match X X P) A Q :- one  P A Q.          % match prefix
oneb (match X X P) A M :- oneb P A M.
one  (plus P Q) A R :- one  P A R.             % sum
one  (plus P Q) A R :- one  Q A R.
oneb (plus P Q) A M :- oneb P A M.
oneb (plus P Q) A M :- oneb Q A M.
one  (par P Q) A (par P1 Q) :- one P A P1.   % par
one  (par P Q) A (par P Q1) :- one Q A Q1.
oneb (par P Q) A (x\par (M x) Q) :- oneb P A M.
oneb (par P Q) A (x\par P (N x)) :- oneb Q A N.
% restriction
one  (nu x\P x) A (nu x\Q x)       :- pi x\ one  (P x) A (Q x).
oneb (nu x\P x) A (y\ nu x\Q x y) :- pi x\ oneb (P x) A (y\ Q x y).
% open
oneb (nu x\M x) (up X) N :- pi y\ one (M y) (up X y) (N y).
% close
one (par P Q) tau (nu y\ par (M y) (N y)) :-
  oneb P (dn X) M , oneb Q (up X) N.
one (par P Q) tau (nu y\ par (M y) (N y)) :-
  oneb P (up X) M , oneb Q (dn X) N.
% comm
one (par P Q) tau (par (M Y) T) :- oneb P (dn X) M, one Q (up X Y) T.
one (par P Q) tau (par R (M Y)) :- oneb Q (dn X) M, one P (up X Y) R.
```

Fig. 7 Specifications for the syntax and one step (late) transitions for the finite π-calculus.

12.2 Encoding the labeled transition system

Mention different choices: early / late are easy to accommodate.

References: [81, 83, 128].

One-step transitions are encoded as two different predicates:

$$P \xrightarrow{A} Q \text{ free or silent action, } A : \mathrm{a}$$
$$P \xrightarrow{\downarrow x} M \text{ bound input action, } \downarrow x : n \to \mathrm{a},\ M : n \to p$$
$$P \xrightarrow{\uparrow x} M \text{ bound output action, } \uparrow x : n \to \mathrm{a},\ M : n \to p$$

```
type  one          p -> a -> p -> o.
type  oneb         p -> (n -> a) -> (n -> p) -> o.
```

Three example inference rules defining the semantics of π-calculus.

$$\frac{}{\bar{x}y.P \xrightarrow{\bar{x}y} P} \qquad \frac{P \xrightarrow{\alpha} P'}{[x = x]P \xrightarrow{\alpha} P'} \qquad \frac{P \xrightarrow{\alpha} P'}{(y)P \xrightarrow{\alpha} (y)P'}\ y \notin \mathrm{n}(\alpha)$$

$$\text{OUTPUT-ACT}: \quad \bar{x}y.P \xrightarrow{\bar{x}y} P \triangleq \top$$
$$\text{MATCH}: \quad [x = x]P \xrightarrow{\alpha} P' \triangleq P \xrightarrow{\alpha} P'$$
$$\text{RES}: \quad (x)Px \xrightarrow{\alpha} (x)P'x \triangleq \nabla x.(Px \xrightarrow{\alpha} P'x)$$

DM Finite pi-calculus versus full pi-calculus. Reference Tiu's paper.

Internal mobility The π_I-calculus (the π-calculus with internal mobility [119]) can also be seen as a setting where only β_0-conversion is needed [79]. In the π-calculus literature there is a notion of "internal mobility" captured by the π_I-calculus of Sangiorgi [1996]. In this fragment, β_0 is the only form of β that is needed to bind input variables to outputs. It is noted in [79] that if one takes a λ-tree specification for one-step transitions for the π-calculus and removes from it those clauses that may require β-conversion (as opposed to β_0-conversion), then one is left with a specification of the π_I-calculus for "internal mobility" [119]: that is, the notion of *binder mobility* described in Section 5 directly accounts for the internal mobility captured by this subset of the π-calculus.

Non-transition. Negation. Consider the process $(y)[x = y]\bar{x}z.0$. It cannot make any transition since y has to be "new"; that is, it cannot be x. The following statement is provable.

$$\forall x \forall Q \forall \alpha.[((y)[x = y](\bar{x}z.0) \xrightarrow{\alpha} Q) \supset \bot]$$

DM Explain why this is provable. There is an essential feature of nabla here.

12.3 Encoding simulation in the π-calculus

Show that a certain pi-calculus expression is bisim to 0. This is equivalent to showing that that expression does not make a transition. This is then motivates a certain formula that, if treated with a universal quantification, leads to something that is not provable. Instead, the nabla solves the problem.

Simulation for the π-calculus is defined simply as:

$$\begin{aligned} sim\ P\ Q \triangleq\ & \forall A, P'\,[P \xrightarrow{A} P' \Rightarrow \exists Q'.Q \xrightarrow{A} Q' \wedge sim\ P'\ Q'] \wedge \\ & \forall X, P'\,[P \xrightarrow{\downarrow X} P' \Rightarrow \exists Q'.Q \xrightarrow{\downarrow X} Q' \wedge \forall w.sim\ (P'w)\ (Q'w)] \wedge \\ & \forall X, P'\,[P \xrightarrow{\uparrow X} P' \Rightarrow \exists Q'.Q \xrightarrow{\uparrow X} Q' \wedge \nabla w.sim\ (P'w)\ (Q'w)] \end{aligned}$$

Bisimulation is easy to encode (just add additional cases).

Bisimulation corresponds to *open* bisimulation. If the meta-logic is made classical, then *late* bisimulation is captured. The difference can be reduced to the excluded middle $\forall x \forall y.\ x = y \vee x \neq y$ [132].

DM 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 [132].

```
CoDefine bisim : p -> p -> prop by
bisim P Q :=
  (forall A P1, {one P A P1} ->
    exists Q1, {one Q A Q1} /\ bisim P1 Q1) /\
  (forall X M, {oneb P (dn X) M} ->
    exists N, {oneb Q (dn X) N} /\ (forall w, bisim (M w) (N w)) ) /\
  (forall X M, {oneb P (up X) M} ->
    exists N, {oneb Q (up X) N} /\ nabla w, bisim (M w) (N w)) /\
  (forall A Q1, {one Q A Q1} ->
    exists P1, {one P A P1} /\ bisim Q1 P1) /\
  (forall X N, {oneb Q (dn X) N} ->
    exists M, {oneb P (dn X) M} /\ (forall w, bisim (N w) (M w))) /\
  (forall X N, {oneb Q (up X) N} ->
    exists M,  {oneb P (up X) M} /\ nabla w, bisim (N w) (M w)).
```

Honsell, Miculan, and Scagnetto in [59], theory of context. Coding in Coq. Actually some of there specifications come close to looking like those by Miller and Tiu. Different logical foundations but similar looking specifications.

(a) Propositional connectives and *free action* modalities:

$$\begin{array}{lll}
\text{(true :)} & P \models \text{true} & \triangleq\ \top. \\
\text{(and :)} & P \models A\&B & \triangleq\ P \models A \wedge P \models B. \\
\text{(or :)} & P \models A\hat{\vee}B & \triangleq\ P \models A \vee P \models B. \\
\text{(match :)} & P \models \langle X \doteq X\rangle A & \triangleq\ P \models A. \\
\text{(match :)} & P \models [X \doteq Y]A & \triangleq\ (X = Y) \supset P \models A. \\
\text{(free :)} & P \models \langle X\rangle A & \triangleq\ \exists P'(P \xrightarrow{X} P' \wedge P' \models A). \\
\text{(free :)} & P \models [X]A & \triangleq\ \forall P'(P \xrightarrow{X} P' \supset P' \models A).
\end{array}$$

(b) The obvious approach to bound action modalities.

$$\begin{array}{ll}
\text{(out :)} & P \models \langle \uparrow X\rangle A \ \triangleq\ \exists P'(P \xrightarrow{\uparrow X} P' \wedge \nabla y.P'y \models Ay). \\
\text{(out :)} & P \models [\uparrow X]A \ \triangleq\ \forall P'(P \xrightarrow{\uparrow X} P' \supset \nabla y.P'y \models Ay).
\end{array}$$

(c) The obvious approach to bound action modalities.

$$\begin{array}{ll}
\text{(in :)} & P \models \langle \downarrow X\rangle A \ \triangleq\ \exists P'(P \xrightarrow{\downarrow X} P' \wedge \exists y.P'y \models Ay). \\
\text{(in :)} & P \models [\downarrow X]A \ \triangleq\ \forall P'(P \xrightarrow{\downarrow X} P' \supset \forall y.P'y \models Ay).
\end{array}$$

(d) *Late* modality:

$$\begin{array}{l}
P \models \langle \downarrow X\rangle^l A \ \triangleq\ \exists P'(P \xrightarrow{\downarrow X} P' \wedge \forall y.P'y \models Ay). \\
P \models [\downarrow X]^l A \ \triangleq\ \forall P'(P \xrightarrow{\downarrow X} P' \supset \exists y.P'y \models Ay).
\end{array}$$

(e) *Early* modality:

$$\begin{array}{l}
P \models \langle \downarrow X\rangle^e A \ \triangleq\ \forall y \exists P'(P \xrightarrow{\downarrow X} P' \wedge P'y \models Ay). \\
P \models [\downarrow X]^e A \ \triangleq\ \exists y \forall P'(P \xrightarrow{\downarrow X} P' \supset P'y \models Ay).
\end{array}$$

Fig. 8 Modal logics for the π-calculus in λ-tree syntax

12.4 Modal logics for mobility

Modal logics for pi-calculus [134, 132].

12.5 Bisimulation up to

DM Drop this section. Leave on some citations at most.

Consider the problem of formalizing the meta-theory of bisimulation-up-to [85, 115] for the π-calculus [86]. Such a meta-theory 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 meta-theory it 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 meta-theorems surrounding bisimulation-up-to (including the subtle meta-theory surrounding weak bisimulation) [113, 114, 20]. Second, a package for the treatment of bindings and names that are used to describe the operational semantics of the π-calculus. Such packages exist (for example, see [10]) and, when combined with treatments of coinduction, may allow one to make progress on the meta-theory of the π-calculus. Recently, the Hybrid systems [34] 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 meta-theory task on top of an establish 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 enrich logic with least and greatest fixed points [12, 70, 91] and with a treatment of bindings [132, 41, 43]. Such a logic has been given a model-checking-style implementation [14] and is the basis of the Abella theorem prover [13, 42]. Using such implementations, the π-calculus has been implemented, formalized, and analyzed in some detail [131, 129] including some of the meta-theory of bisimulation-up-to for the π-calculus [23].

12.6 Performing proof search specifications

DM Is this subsection needed at all?
lambda Prolog for one-step
Bedwyr for model checking (finite) pi-calculus
Role of unification, backtracking, and focused proofs.
A Proof Search Specification of the π-Calculus [131, 129].

13 Related work

Besides the Abella, Bedwyr, and Twelf system mentioned above, there are a number of other implemented systems that support some or all aspects of λ-tree syntax: these include Beluga [108], Hybrid [34], Isabelle [102], Minlog [123], and Teyjus [94]. See [35] for a survey and comparison of several of these systems. NB There is also the recent paper "The Logic of Hereditary Harrop Formulas as a Specification Logic for Hybrid" by Chelsea Battell and Amy Felty. (dropbox for Amy)

There are many efforts for mechanizing metatheory. See *various JAR special issues*. Comparisons are difficult although many have been published: see ...

Here, I limit discussions about systems that are closer in spirit to what is proposed here. Twelf, Beluga, Agda?,

13.1 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 of 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 $t : tm$ ("t denotes a term") can be refined to $t : tm\ (int \to int)$ ("t denotes a term of type $int \to int$). Such richer types are supported (and generalized) by the *dependent type* paradigm [29,56] and given a logic programming implementation in, for example, Twelf [104,107].

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 [125,126] but it is not committed to only that notion of typing and proof.

In this paper, we keep our attention on logic instead of type theory since we do not wish to commit now to one notion of proof (such as dependently typed λ-terms) nor to just intuitionistic logic principles: embracing classical and linear logic proof principles also seem an essential part of specifying and reasoning about meta-theory.

DM Another problem with dependent typing is that we need to deal with two logics ultimately: one for specifying computations and one for reasoning about computation. It has been hard enough to pick the right logics (with associated connectives) for these logics: picking their proofs and forcing them to look like lambda-terms is a bit too hard. Maybe additional research will uncover new roles to play with dependently typed terms.

DM (Maybe this does not go under related work?) Ultimately, one should explore to what extent the mathematical concerns that Church expressed by moving from ETT to STT can be addressed as extensions to the logic described in this paper. To the extent that bindings in expressions is not a concern of a given formalization, then ∇ and its associated operations (nominal abstractions, higher-order unification) found in ETT are not used. There has been occasionally interest in having a formal language that contains two abstractions with two arrow types: one for abstractions over syntax and

one for function spaces. The substitution operation on syntax would map the former into the later [73]. Maybe references to Tim Sheard on Dali are appropriate here but they seem not to have been cited and they are old.

DM From [36, Page 1] there is a quote from the Honsell, et al paper. There are recent proposals to overcome this shortcoming

14 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 λ-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 α and β_0 (at least) and must allow for the mobility of binders from within terms to within formulas (i.e., quantifiers) or proof state (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. Thus, in contrast to the conclusions of the POPLmark challenge that increments to existing provers will solve the problems surrounding the mechanization of metatheory, I argue that we need to consider making a significant shift to the underlying paradigm that has been built into the most mature of today's proof assistants.

DM A problem here is that treating term equality and unification as a black box is likely to lead to serious problems. For first-order problems (including the pattern unification subset), mgus exist for unifiable pairs. But for richer subsets of logic, CSU are complex objects that may be infinite [62]. Clearly, such complexity is not a good idea is a primitive inference.

I have described an extension of ETT targeting metatheory and not mathematics. The resulting logic provides for λ-tree syntax in a direct fashion, via binder-mobility, ∇-quantification, and the unification of λ-terms. Induction over syntax containing bindings is available: in its richest setting, such induction is done over sequent calculus proofs of typing derivations. Operational semantics and typing judgments are often encoded directly. The Abella system has been used to successfully capture important aspects of the metatheory of the λ-calculus, π-calculus, programming languages, and object-logics.

DM Ultimately, we would like to merge into one logic these two branches: namely $\mathcal{G}$ and STT. Hybrid is an attempt to do this at the systems level.

Acknowledgments. Part of this work has been funded by the ERC Advanced Grant ProofCert.

References

1. S. Abramsky. The lazy lambda calculus. In D. A. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Welsey, Reading, MA, 1990.
2. 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.
3. Thorsten Altenkirch. A formalization of the strong normalization proof for system F in LEGO. In *Typed Lambda Calculi and Applications (TLCA)*, volume 664, pages 13–28, 1993.
4. Peter B. Andrews. Resolution in type theory. *J. of Symbolic Logic*, 36:414–432, 1971.
5. Peter B. Andrews. Provability in elementary type theory. *Zeitschrift fur Mathematische Logic und Grundlagen der Mathematik*, 20:411–418, 1974.
6. Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Kluwer Academic Publishers, second edition, 2002.
7. 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.
8. Brian Aydemir, Stephan A. Zdancewic, and Stephanie Weirich. Abstracting syntax. Technical Report MS-CIS-09-06, University of Pennsylvania, 2009.
9. 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.
10. 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.
11. David Baelde. On the expressivity of minimal generic quantification. In A. Abel and C. Urban, editors, *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2008)*, number 228 in ENTCS, pages 3–19, 2008.
12. David Baelde. Least and greatest fixed points in linear logic. *ACM Trans. on Computational Logic*, 13(1), April 2012.
13. 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.
14. 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.
15. David Baelde and Dale Miller. Least and greatest fixed points in linear logic. In N. Dershowitz and A. Voronkov, editors, *International Conference on Logic for Programming and Automated Reasoning (LPAR)*, volume 4790 of *LNCS*, pages 92–106, 2007.
16. Henk P. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, April 1991.
17. Christoph Benzmüller and Dale Miller. Automation of higher-order logic. In J. Siekmann, editor, *Logic and Computation*, volume 9 of *Handbook of the History of Logic*, pages 215–254. North Holland, 2014.
18. Ulrich Berger, Stefan Berghofer, Pierre Letouzey, and Helmut Schwichtenberg. Program extraction from normalization proofs. *Studia Logica*, 82(1):25–49, 2006.
19. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.
20. 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.
21. Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda-A functional language with dependent types. In *TPHOLs*, volume 5674, pages 73–78. Springer, 2009.
22. Arthur Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, pages 1–46, May 2011.

23. 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.
24. 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.
25. Alonzo Church. A formulation of the Simple Theory of Types. *J. of Symbolic Logic*, 5:56–68, 1940.
26. Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, Laurent Hascoët, and Gilles Kahn. Natural semantics on the computer. Research Report 416, INRIA, Rocquencourt, France, June 1985.
27. Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.
28. Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.
29. 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.
30. 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.
31. 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.
32. 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 *LNCS*, pages 460–468. Springer, 2015.
33. 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 *LNAI*, pages 221–235. Springer, 1990.
34. 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.
35. Amy P. Felty, Alberto Momigliano, and Brigitte Pientka. The next 700 challenge problems for reasoning with higher-order abstract syntax representations: Part 2A survey. *J. of Automated Reasoning*, 2015.
36. 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.
37. M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2001.
38. 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.
39. Andrew Gacek. *A Framework for Specifying, Prototyping, and Reasoning about Computational Systems*. PhD thesis, University of Minnesota, 2009.
40. Andrew Gacek. Relating nominal and higher-order abstract syntax specifications. In *Proceedings of the 2010 Symposium on Principles and Practice of Declarative Programming*, pages 177–186. ACM, July 2010.
41. Andrew Gacek, Dale Miller, and Gopalan Nadathur. Combining generic judgments with recursive definitions. In F. Pfenning, editor, *23th Symp. on Logic in Computer Science*, pages 33–44. IEEE Computer Society Press, 2008.
42. 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 ENTCS, pages 85–100, 2008.
43. Andrew Gacek, Dale Miller, and Gopalan Nadathur. Nominal abstraction. *Information and Computation*, 209(1):48–73, 2011.

44. 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.
45. Gerhard Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland, Amsterdam, 1935.
46. Gerhard Gentzen. New version of the consistency proof for elementary number theory. In M. E. Szabo, editor, *Collected Papers of Gerhard Gentzen*, pages 252–286. North-Holland, Amsterdam, 1938. Originally published 1938.
47. Jean-Yves Girard. Une extension de l'interpretation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *2nd Scandinavian Logic Symposium*, pages 63–92. North-Holland, Amsterdam, 1971.
48. Jean-Yves Girard. A fixpoint theorem in linear logic. An email posting to the mailing list linear@cs.stanford.edu, February 1992.
49. Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte der Mathematischen Physik*, 38:173–198, 1931. English Version in [136].
50. M. J. C. Gordon and T. F. Melham. *Introduction to HOL – A theorem proving environment for higher order logic*. Cambridge University Press, 1993.
51. Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *LNCS*. Springer, 1979.
52. Mike Gordon. From LCF to HOL: a short history. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 169–186. MIT Press, 2000.
53. John Hannan. Extended natural semantics. *J. of Functional Programming*, 3(2):123–152, April 1993.
54. John Hannan and Dale Miller. From operational semantics to abstract machines: Preliminary results. In M. Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 323–332. ACM, ACM Press, 1990.
55. John Hannan and Dale Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 2(4):415–459, 1992.
56. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.
57. John Harrison. HOL light: an overview. In *International Conference on Theorem Proving in Higher Order Logics*, pages 60–66. Springer, 2009.
58. 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.
59. Furio Honsell, Marino Miculan, and Ivan Scagnetto. π-calculus in (co)inductive type theories. *Theoretical Computer Science*, 2(253):239–285, 2001.
60. Douglas J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, 1996.
61. Gérard Huet. The undecidability of unification in third order logic. *Information and Control*, 22:257–267, 1973.
62. Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.
63. Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, volume 247 of *LNCS*, pages 22–39. Springer, March 1987.
64. 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.
65. Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.
66. 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.
67. Donald MacKenzie. *Mechanizing Proof*. MIT Press, 2001.
68. Per Martin-Löf. *Intuitionistic Type Theory*. Studies in Proof Theory Lecture Notes. Bibliopolis, Napoli, 1984.

69. 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.
70. Raymond McDowell and Dale Miller. Cut-elimination for a logic with definitions and induction. *Theoretical Computer Science*, 232:91–119, 2000.
71. 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.
72. Dale Miller. A compact representation of proofs. *Studia Logica*, 46(4):347–370, 1987.
73. 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.
74. Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. of Logic and Computation*, 1(4):497–536, 1991.
75. Dale Miller. Abstract syntax and logic programming. In *Logic Programming: Proceedings of the First and Second Russian Conferences on Logic Programming*, number 592 in LNAI, pages 322–337. Springer, 1992.
76. Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14(4):321–358, 1992.
77. Dale Miller. Bindings, mobility of bindings, and the ∇-quantifier. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *18th International Conference on Computer Science Logic (CSL) 2004*, volume 3210 of *LNCS*, page 24, 2004.
78. 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.
79. Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, June 2012.
80. 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.
81. Dale Miller and Catuscia Palamidessi. Foundational aspects of syntax. *ACM Computing Surveys*, 31, September 1999.
82. Dale Miller and Alwen Tiu. A proof theory for generic judgments: An extended abstract. In Phokion Kolaitis, editor, *18th Symp. on Logic in Computer Science*, pages 118–127. IEEE, June 2003.
83. Dale Miller and Alwen Tiu. A proof theory for generic judgments. *ACM Trans. on Computational Logic*, 6(4):749–783, October 2005.
84. Dale A. Miller, Eve Longini Cohen, and Peter B. Andrews. A look at TPS. In Donald W. Loveland, editor, *Sixth Conference on Automated Deduction*, volume 138 of *LNCS*, pages 50–69, New York, 1982. Springer.
85. Robin Milner. *Communication and Concurrency*. Prentice-Hall International, 1989.
86. Robin Milner. *Communicating and Mobile Systems : The π-Calculus*. Cambridge University Press, New York, NY, USA, 1999.
87. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Part I. *Information and Computation*, 100(1):1–40, September 1992.
88. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Part II. *Information and Computation*, 100(1):41–77, 1992.
89. Robin Milner, Joachim Parrow, and David Walker. Modal logics for mobile processes. *Theoretical Computer Science*, 114(1):149–171, 1993.
90. John C. Mitchell and Eugenio Moggi. Kripke-style models for typed lambda calculus. *Annals of Pure and Applied Logic*, 51(1-2):99–124, 1991.
91. 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 LNCS, pages 293–308, January 2003.
92. J. Strother Moore. A mechanically verified language implementation. *J. of Automated Reasoning*, 5(4):461–492, 1989.
93. Gopalan Nadathur and Dale Miller. An Overview of λProlog. In *Fifth International Logic Programming Conference*, pages 810–827, Seattle, August 1988. MIT Press.
94. Gopalan Nadathur and Dustin J. Mitchell. System description: Teyjus — A compiler and abstract machine based implementation of λProlog. In H. Ganzinger, editor, *16th Conf. on Automated Deduction (CADE)*, number 1632 in LNAI, pages 287–291, Trento, 1999. Springer.

95. Gopalan Nadathur and Debra Sue Wilson. A notation for lambda terms: A generalization of environments. *Theoretical Computer Science*, 198(1-2):49–98, 1998.
96. Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual model type theory. *ACM Trans. on Computational Logic*, 9(3):1–49, 2008.
97. Adam Naumowicz and Artur Korniłowicz. A brief overview of Mizar. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *LNCS*, pages 67–72, 2009.
98. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Number 2283 in LNCS. Springer, 2002.
99. Bengt Nordstrom, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's type theory : an introduction*. International Series of Monographs on Computer Science. Oxford: Clarendon, 1990.
100. Lawrence C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.
101. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Number 828 in LNCS. Springer Verlag, 1994.
102. Lawrence C. Paulson. A generic tableau prover and its integration with isabelle. *J. UCS*, 5(3):73–87, 1999.
103. Alan J. Perlis. Epigrams on programming. *ACM SIGPLAN Notices*, pages 7–13, September 1982.
104. 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.
105. 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.
106. Frank Pfenning and Ekkehard Rohwedder. Implementing the meta-theory of deductive systems. In *Proceedings of the 1992 Conference on Automated Deduction*, number 607 in LNCS, pages 537–551. Springer, June 1992.
107. 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.
108. 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 LNCS, pages 15–21, 2010.
109. Andrew M. Pitts. Alpha-structural recursion and induction. *J. ACM*, 53(3):459–506, 2006.
110. Gordon D. Plotkin. A structural approach to operational semantics. *J. of Logic and Algebraic Programming*, 60-61:17–139, 2004.
111. The POPLmark Challenge webpage. `http://www.seas.upenn.edu/~plclub/poplmark/`, 2015.
112. François Pottier. Static name control for freshML. In *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*, pages 356–365. IEEE, 2007.
113. Damien Pous. Weak bisimulation upto elaboration. In C. Baier and H. Hermanns, editors, *CONCUR*, volume 4137 of *LNCS*, pages 390–405. Springer, 2006.
114. Damien Pous. Complete lattices and upto techniques. In Zhong Shao, editor, *APLAS*, volume 4807 of *LNCS*, pages 351–366, Singapore, November 2007. Springer.
115. 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.
116. Dag Prawitz. Hauptsatz for higher order logic. *Journal of Symbolic Logic*, 33:452–457, 1968.
117. Xiaochu Qi, Andrew Gacek, Steven Holte, Gopalan Nadathur, and Zach Snow. The Teyjus system – version 2, 2015. http://teyjus.cs.umn.edu/.
118. C. Röckl, D. Hirschkoff, and S. Berghofer. Higher-order abstract syntax with induction in Isabelle/HOL: Formalizing the pi-calculus and mechanizing the theory of contexts. In F. Honsell and M. Miculan, editors, *Proc. FOSSACS'01*, volume 2030 of *LNCS*, pages 364–378. Springer, 2001.
119. Davide Sangiorgi. π-calculus, internal mobility and agent-passing calculi. *Theoretical Computer Science*, 167(2):235–274, 1996.

120. Davide Sangiorgi and David Walker. *π-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.
121. Peter Schroeder-Heister. Rules of definitional reflection. In M. Vardi, editor, *8th Symp. on Logic in Computer Science*, pages 222–232. IEEE Computer Society Press, IEEE, June 1993.
122. Carsten Schürmann and Frank Pfenning. Automated theorem proving in a simple meta-logic for LF. In Claude Kirchner and Hélène Kirchner, editors, *15th Conf. on Automated Deduction (CADE)*, volume 1421 of *Lecture Notes in Computer Science*, pages 286–300. Springer, 1998.
123. Helmut Schwichtenberg. Minlog. In Freek Wiedijk, editor, *The Seventeen Provers of the World*, volume 3600 of *LNCS*, pages 151–157. Springer, 2006.
124. Helmut Schwichtenberg. MINLOG reference manual. *LMU München, Mathematisches Institut, Theresienstraße*, 39, 2011.
125. 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.
126. 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.
127. Moto-o Takahashi. A proof of cut-elimination theorem in simple type theory. *Journal of the Mathematical Society of Japan*, 19:399–410, 1967.
128. Alwen Tiu. *A Logical Framework for Reasoning about Logical Specifications*. PhD thesis, Pennsylvania State University, May 2004.
129. Alwen Tiu. Model checking for π-calculus using proof search. In Martín Abadi and Luca de Alfaro, editors, *Proceedings of CONCUR'05*, volume 3653 of *LNCS*, pages 36–50. Springer, 2005.
130. Alwen Tiu. A logic for reasoning about generic judgments. In A. Momigliano and B. Pientka, editors, *Int. Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'06)*, volume 173 of *ENTCS*, pages 3–18, 2006.
131. Alwen Tiu and Dale Miller. A proof search specification of the π-calculus. In *3rd Workshop on the Foundations of Global Ubiquitous Computing*, volume 138 of *ENTCS*, pages 79–101, 2005.
132. 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.
133. Alwen Tiu and Alberto Momigliano. Cut elimination for a logic with induction and co-induction. *Journal of Applied Logic*, 10(4):330–367, 2012.
134. Alwen Tiu, Gopalan Nadathur, and Dale Miller. Mixing finite success and finite failure in an automated prover. In *Empirically Successful Automated Reasoning in Higher-Order Logics (ESHOL'05)*, pages 79–98, December 2005.
135. Christian Urban. Nominal reasoning techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40(4):327–356, 2008.
136. 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.
137. Myra VanInwegen. *The Machine-Assisted Proof of Programming Language Properties*. PhD thesis, University of Pennsylvania, May 1996.