Towards a Comprehensive and Efficient Theory of Programming Languages

Ambroise Lafont (Research project)

In the theory of programming languages, it is common for researchers to present proof methods that are demonstrated through specific examples. These proof methods may then be adapted to other settings. For instance, Pitts [1] presents Howe's method[1] by specifically focusing on untyped call-by-value λ-calculus and sketches how it applies for some other cases. To improve the situation, Turi and Plotkin initiated a search for a solid and stable mathematical definition of programming languages [3]. The motivation is that various results, proof methods, and notions can then be developed once and for all, and used out-of-the-box by language designers. In the same vein, my research project essentially focuses on tackling crucial questions about the foundational aspects of programming languages, as detailed in Section 1. More specifically, I plan to contribute to the transition to using certified proof assistants for programming languages, compilers, and tools. This means that these environments will be formally verified, providing strong guarantees about their correctness and reliability. Accordingly, a long-term goal consists in providing a mechanised library that can automatically generate a programming language with its expected properties from elementary data. Moreover, efforts will be more specifically devoted to the study of dependent type theories underlying proof assistants, which are constantly evolving as new features and capabilities are continually requested. My goal is to tackle these challenges by proposing mathematical abstractions that are suited for formalisation and allow for the reuse of proofs. This involves investigating modularisation of type systems, implementing reusable libraries for their various components, with a focus on unification, which is crucial to make the programmer's life easier, as it is essentially involved in inferring parts of a program left implicit by the user. Logic programming looks like a promising paradigm for that purpose, as it allows to define algorithms by inference rules, thus matching closely their mathematical description. On the theoretical side, I plan in particular to study the adequacy between the implementation of a type theory and its semantics, which has been worked out in details for some variant of Martin-Löf type theory [4]. Considering the involved technicalities, it is crucial to design a generic proof that could apply for different systems. Related issues will be investigated regarding inductive-inductive types, an advanced species of inductive types that can be used to formalise type theory in type theory, building on my previous experience with them [Laf1].

My research program involves a back-and-forth interaction between theory and practice in two ways. First, I prove theorems that generalise more concrete results from the literature, motivated by the applications themselves, and making practice easier as a result. Second, I am also involved in the transition I am trying to facilitate by contributing to the adaptation of proof assistants to my theoretical questions (see Section 2). An essential objective consists in leveraging diagrammatic techniques for designing and working with formal proofs.

1 Semantics of programming languages

1.1 Modularising type systems

Partitioning type systems into smaller, more manageable pieces crucially improves the readability and understandability of large, complex systems. This modularity also facilitates reuse and evolution of the type system, enabling developers to reuse certain parts in different contexts or to modify and extend them without affecting the rest of the system. Additionally, modularity improves the reliability and correctness of the type system by making it easier to reason about the correctness of each module independently. Accordingly, one

[1]See the report on previous work for a quick introduction to Howe's method [2].

long-term goal of my research project consists in implementing safe and reusable libraries for the various components of a type system.

I plan to start with unification, building upon my categorical analysis of generic pattern unification [Laf2], in collaboration with Neel Krishnaswami. Unification is involved in elaboration, a crucial component of proof assistants that consists in converting the "surface" syntax that is visible to the user into a proof term that is readable by the kernel of the proof assistant. This process involves inferring parts of the proof term that the surface syntax leaves implicit, which heavily relies on unification.

Structurally recursive pattern unification (short-term). Implementing a verified higher-order pattern unification library in Agda or Coq requires first to make the termination argument more explicit in the structure of the algorithm so that it is accepted by the proof assistant's termination checker. To this end, I plan to extend the work of McBride [5] which provides a structurally recursive first-order unification algorithm.

Generic efficient unification (mid-term). The generic pattern unification algorithm I worked out has exponential space complexity in the worst-case scenario, following Miller's original presentation of the algorithm [6]. However, Qian [7] devised an algorithm for simply-typed normal λ-terms, which is linear both in time and space, based on representing terms as directed acyclic graphs rather than trees. I plan to investigate how to describe and justify this algorithm in the generic setting I introduced.

Unification modulo reduction (long-term). Unification cannot always be cleanly separated from the rest of the type system (see, e.g., Agda's type system [8]). Accordingly, I would like to find a generic account of unification in syntax modulo equations, or in presence of reductions, so as to take into account situations where unification is inherently mixed with normalisation. This project would benefit from my previous work on the specification of operational semantics [Laf3, Laf4, Laf5].

Anti-unification (short-term). Anti-unification consists in finding the most specific generaliser of two given terms. Anti-unification is relevant in many contexts: inductive logic, program synthesis [9], program verification [10]. It is also used in Agda's type checker[2]. There is little work in the literature studying pattern anti-unification from an algebraic point of view. To bridge this gap, the plan is to start with first-order anti-unification in order to find a categorical generalisation that would also accommodate the higher-order case, in particular using the notion of signature I introduced for generic unification.

Logic programming for type systems (mid-term). Bidirectional type systems, which are typically used for dependent type theories, are standard ways to present concrete implementation: each involved judgement is explicitly split into the input and the output of the typing subroutine it accounts for. However, in practice, the implemented type system typically includes additional boiler-plate code that is absent from the mathematical description. Because inference rules are ubiquitous in the abstract presentation, one way of bridging the gap between the theory and the practice would consist in basing the implementation on the logic programming paradigm, in which programs are also defined by inference rules. Current logic programming languages such as ELPI [11] do not allow arbitrary inference rules, and thus it is unclear whether this is actually possible in the current state of affairs. As a first step, I plan to try and implement my generic higher-order pattern unification algorithm [Laf2], which is also presented with inference rules, in a logic programming language. In the longer run, I will investigate implementing type systems based on unification such as Agda as presented in Norell's PhD thesis [8]. I plan to collaborate with Enrico Tassi who is implementing a Coq plugin for the ELPI programming language and shares similar interests.

1.2 Dependent type theories

Dependent type theories are formal systems used as the foundation of proof assistants, such as Coq, Agda, or Lean. These proof assistants are software tools that help users construct and verify mathematical proofs. Dependent type theories are notable for their highly sophisticated type systems, which are used to represent

[2]See `https://github.com/UlfNorell/agda/commit/7c8bc3b40503b7efba55f96225ad8ea71942aa85`.

and reason about the various kinds of mathematical objects that can appear in a proof. In a dependent type theory, types can depend on other types and terms in a way that is not possible in most other programming languages. This allows users to specify precise constraints on the structure and behaviour of their mathematical arguments, and to ensure that their proofs are correct and consistent.

I plan to address crucial questions in the metatheory of dependent type theories that are still not satisfactory handled.

Towards a generic initiality conjecture (long-term). The proof of adequacy between the implementation of a type theory and its mathematical semantics can be very technical. Vladimir Voevodsky stressed its importance, referring to it as the initiality conjecture [12]. A mechanised proof has recently been worked out for an extended version of Martin-Löf type theory [4], but a generic proof is still lacking to avoid reproving it for each system: this is a long-term goal of my research project. To this end, I will build upon the work on finitary type theories [13] describing the implementation side, and Uemura's semantics of type theories [14] or Gratzer and Sterling's approach [15] based on locally cartesian closed categories. I will also investigate alternative semantics which are closer to the context-free implementation of type theories [13, 16], as is the case in the proof assistant Andromeda 2. One attractive feature of this viewpoint is that the core mathematical structure is quite simple: it is just a family of terms indexed over types, with an embedding of De Bruijn variables for each type. This will build upon my previous work on a sophisticated species of inductive types called inductive-inductive types [Laf1]. Indeed, those sophisticated inductive types allow the introduction of a family of types by mutual definition of the indexing type and the indexed types. Overall, the plan is to look for a notion of signature for type theories, exploiting the signatures for inductive-inductive types I previously introduced [Laf1], then define their models, and finally prove that the syntax gives an initial model, in the spirit of Initial Algebra Semantics.

Initiality conjecture for inductive-inductive types (mid-term). As a starting point, I would like to investigate a related question regarding inductive-inductive types, which will help me gain some confidence and expertise on this problem while building upon my previous experience with inductive-inductive types [Laf1]. Because the Coq proof assistant does not natively support them, it is a common strategy to encode a given inductive-inductive type by introducing an over-approximation using regular inductive types, which is then restricted using inductive well-formedness predicates. I plan to implement a tool that can perform this encoding automatically. There have been previous attempts [17, 18] without any decisive success so far. The main challenge is to recover the full recursion principle of the desired inductive-inductive type. As my report on previous work shows (see Section 3.2), I am already experienced with this kind of difficulty, which makes me particularly suitable to tackle this issue. A first approach I will adopt consists in examining whether my previous work on the construction of inductive-inductive types [Laf1] could be exploited for this purpose. A preliminary investigation suggests that it could indeed provide an automatic translation of a given inductive-inductive type. However, it would probably not match and justify the usual encoding. Thus, I also plan to tackle this issue from a more abstract viewpoint, in particular adopting a semantical notion of signature for inductive-inductive types as suggested in Forsberg's PhD thesis [19].

1.3 Auxiliary operations

A long-term goal of this research project consists in building a mechanised library that can automatically generate a programming language from its specification. My previous work on reduction/transition monads [Laf3, Laf5] is a first step in this direction. However, they are not really suitable for the situations where program execution involves auxiliary operations defined by recursion on the syntax, except from substitution which is always built-in. For example, the operational semantics of the differential λ-calculus [20] relies on such an operation called partial derivation. There is little work in the literature on the initial algebra semantics of operations which are auxiliary, in the sense that they are defined a posteriori by recursion on the syntax. In recent work [Laf6], I make a first step towards bridging this gap in the literature, which opens new research directions towards a satisfactory notion of specification for programming languages involving auxiliary operations.

Generalising Howe's method (short-term). Howe's method [2] is a proof method used to show that bisimilarity is an adequate notion of program equivalence in higher-order programming languages. Originally, it did not support operational semantics involving auxiliary operations. It was thus extended on a case by case basis to cover the λ-calculus with delimited continuations [21] or higher-order process calculi [22]. Indeed, these languages feature a notion of evaluation context, roughly a program with a hole, and an auxiliary operation called *context application*, which amounts to filling the hole with a program. With Tom Hirschowitz, I plan to combine our work on Howe's method [Laf7] and auxiliary operations [Laf6] to suggest a general setting that covers those new examples. in the longer run, I plan to to address program equivalences other than applicative bisimilarity, typically normal-form bisimilarity [23], which also often rely on context application.

Syntax with equations (short-term). I plan to extend my work to account for syntax with equations. One objective is indeed to cover the differential lambda calculus, which includes a commutative associative binary operation in the syntax. The crucial point consists in finding a general setting that ensures that auxiliary operations are compatible with the quotient performed on the syntax.

Programming with binders (mid-term). Substitution is a major syntactic auxiliary operation whose definition can be tricky when working in a syntax with binders. The FreshML programming language [24] has been specifically designed to mitigate those difficulties. However, the denotational semantics is not as tight as one would expect *a priori*. As an example, although it is possible to define the inductive type of λ-terms that can indeed be used to represent any closed λ-term, there are closed FreshML programs that actually denote open λ-terms in the denotational semantics [24, Equation (30)]. Therefore, I would like to explore an other approach, looking for a language with clean semantics inspired by Menni's notion of object of names [25]. It seems that a cartesian closed category with an atomic object V is already a good setting for dealing with binders. First, since simply-typed λ-calculus is the programming language for cartesian closed categories, it can be interpreted in such a category. Second, thinking of V as the object of names, given an algebraically free monad T on this category with some additional structure, it is possible to define unary substitution on $T(V)$ as a morphism $T(V)^V \times T(V) \to T(V)$. This situation specialises as expected when taking for T the monad for λ-calculus in Fiore, Plotkin, and Turi's presheaf setting [26]. My starting point thus consists in working out a programming language for cartesian closed categories equipped with an atomic object. I will test its viability by re-implementing specific instances of Fiore and Szamozvancev's mechanised library [27] of second-order syntax in this setting.

Semantics of inductive-recursive types (mid-term). I am interested in investigating the semantics of inductive-recursive types in light of my work on auxiliary operations [Laf6]. These are advanced inductive types available in the Agda proof assistant that enable the mutual definition of a set and an auxiliary operation on it, making type theory a suitable metalanguage for intuitionistic meta-mathematics [28].

2 Making mechanisation easier

This section is partly motivated by my recent work on the semantics of auxiliary operations [Laf6], which turned out to be technically more involved than expected. I am interested in mechanising those results, not only to solidify their correctness, but also to provide a reusable library. Preliminary work in this direction has raised some further research directions that could potentially provide more efficient proof strategies, geared towards mechanisation.

2.1 Diagrammatic reasoning

When working in a categorical setting, a recurrent task consists in proving commutation of diagrams, which can sometimes be pretty large. This motivated me to implement a diagram editor[3] with user-driven interaction capabilities with the Coq proof assistant to speed up my research. It turned out that enhancing Coq with diagrammatic reasoning capabilities is one of the main goals of a project led by Nicholas Behr (IRIF).

[3]`https://amblafont.github.io/graph-editor/`

The members of this project showed great interest in my diagram editor, and invited me to participate in the elaboration of a French ANR grant application on this topic. This application has been accepted, and, being employed abroad, I am mentioned as an external consultant. One long-term goal of the resulting ANR project CoREACT consists in developing an interactive editor, initially based on my software, that enables diagrammatic reasoning in a variety of contexts, starting with plain category theory.

A domain-specific type theory for diagrams (short-term). To guide the design of this tool, it is crucial to get an accurate mathematical representation of diagrams and reasoning about them. In the short term, I plan to contribute to the design of a domain-specific type theory keeping track of the geometric information, that will provide the foundations for a generic front-end for diagrammatic reasoning. In this respect, I will build upon my previous experience in mechanising domain specific type theories such as Brunerie's type theory [29] or the theory of signatures for inductive-inductive types [Laf1].

Formalising category theory (mid-term). Within this project, I also plan to get involved in the formalisation of category theory in Coq, exploiting my experience in formalising various categorical results in the UniMath Coq library. One aspect I particularly care about is the effectiveness of the definitions, so that algorithms relying on them indeed compute. I could then develop the mechanised libraries related to my research project (such as a generic unification algorithm) upon this framework.

2.2 Local coherence (mid-term)

To automate the proof of numerous intermediary lemmas, my collaborator Tom Hirschowitz and myself used a Coq tactic I implemented[4] that automatically proves equalities of morphisms involving monadic structure. It relies on the definition of monads as extension systems [30]: the tactic chooses an orientation of the monadic equations to compute a normal form from a given composition of morphisms. Testing equality of morphisms then amounts to comparing their normal forms. I have extended and used this tactic in the setting of a category with coproducts, equipped with a distributive law of monads. I plan to investigate this rewriting strategy to understand if it can be made complete. Indeed, in our specific situation, it seems that a local coherence result holds: given a monadic distributive law $TS \to ST$ and a T-algebra $TX \to X$, then two parallel morphisms targeting SX and composed of structural morphisms should always be equal. Solving this particular case would already entail significant simplifications in our proofs. Coherence results have been well studied in the literature (see, e.g., Mac Lane's theorem for monoidal categories [31, Chapter 7]). What makes this problem stands out, however, is that it is "local", in the sense that the morphisms that are expected to be equal all target the same object. As a consequence, general coherence theorems [32] do not seem to readily apply to this situation.

Personal publications and preprints

[Laf1] Ambrus Kaposi, András Kovács, and Ambroise Lafont. "For Finitary Induction-Induction, Induction Is Enough". In: *25th International Conference on Types for Proofs and Programs (TYPES 2019)*. Ed. by Marc Bezem and Assia Mahboubi. Vol. 175. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, 6:1–6:30. ISBN: 978-3-95977-158-0. DOI: `10.4230/LIPIcs.TYPES.2019.6`. URL: `https://drops.dagstuhl.de/opus/volltexte/2020/13070`.

[Laf2] Ambroise Lafont and Neel Krishnaswami. "Generic pattern unification: a categorical approach". Preprint. Oct. 2022. URL: `https://raw.githubusercontent.com/amblafont/unification/master/lncs-long.pdf`.

[Laf3] Benedikt Ahrens, André Hirschowitz, Ambroise Lafont, and Marco Maggesi. "Reduction monads and their signatures". In: *Proc. ACM Program. Lang.* 4.POPL (2020), 31:1–31:29. DOI: `10.1145/3371099`. URL: `https://doi.org/10.1145/3371099`.

[4] `https://amblafont.github.io/stuff/eqsolver.v`

[Laf4] André Hirschowitz, Tom Hirschowitz, and Ambroise Lafont. “Modules over Monads and Operational Semantics”. In: *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference)*. Ed. by Zena M. Ariola. Vol. 167. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 12:1–12:23. DOI: 10.4230/LIPIcs.FSCD.2020.12. URL: https://doi.org/10.4230/LIPIcs.FSCD.2020.12.

[Laf5] André Hirschowitz, Tom Hirschowitz, and Ambroise Lafont. “Modules over monads and operational semantics (expanded version)”. In: *Log. Methods Comput. Sci.* 18.3 (2022). DOI: 10.46298/lmcs-18(3:3)2022. URL: https://doi.org/10.46298/lmcs-18(3:3)2022.

[Laf6] Tom Hirschowitz and Ambroise Lafont. “A unified treatment of structural definitions on syntax for capture-avoiding substitution, context application, named substitution, partial differentiation, and so on”. Preprint. Apr. 2022. URL: https://hal.archives-ouvertes.fr/hal-03633933.

[Laf7] Tom Hirschowitz and Ambroise Lafont. “A categorical framework for congruence of applicative bisimilarity in higher-order languages”. In: *Log. Methods Comput. Sci.* 18.3 (2022). DOI: 10.46298/lmcs-18(3:37)2022. URL: https://doi.org/10.46298/lmcs-18(3:37)2022.

Other references

[1] Andrew M. Pitts. “Howe’s method for higher-order languages”. In: *Advanced Topics in Bisimulation and Coinduction*. Ed. by Davide Sangiorgi and Jan J. M. M. Rutten. Vol. 52. Cambridge tracts in theoretical computer science. Cambridge University Press, 2012, pp. 197–232.

[2] Douglas J. Howe. “Proving Congruence of Bisimulation in Functional Programming Languages”. In: *Inf. Comput.* 124.2 (1996), pp. 103–112. DOI: 10.1006/inco.1996.0008. URL: https://doi.org/10.1006/inco.1996.0008.

[3] Daniele Turi and Gordon D. Plotkin. “Towards a Mathematical Operational Semantics”. In: *Proceedings, 12th Annual IEEE Symposium on Logic in Computer Science, Warsaw, Poland, June 29 - July 2, 1997*. IEEE Computer Society, 1997, pp. 280–291. DOI: 10.1109/LICS.1997.614955. URL: https://doi.org/10.1109/LICS.1997.614955.

[4] Menno de Boer. *A Proof and Formalization of the Initiality Conjecture of Dependent Type Theory*. Licentiate defense over Zoom. 2020.

[5] Conor McBride. “First-order unification by structural recursion”. In: *J. Funct. Program.* 13.6 (2003), pp. 1061–1075. DOI: 10.1017/S0956796803004957. URL: https://doi.org/10.1017/S0956796803004957.

[6] Dale Miller. “A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification”. In: *J. Log. Comput.* 1.4 (1991), pp. 497–536. DOI: 10.1093/logcom/1.4.497. URL: https://doi.org/10.1093/logcom/1.4.497.

[7] Zhenyu Qian. “Unification of Higher-Order Patterns in Linear Time and Space”. In: *J. Log. Comput.* 6.3 (1996), pp. 315–341. DOI: 10.1093/logcom/6.3.315. URL: https://doi.org/10.1093/logcom/6.3.315.

[8] Ulf Norrell. “Towards a practical programming language based on dependent type theory”. PhD thesis. Chalmers University of Technology, 2007. URL: https://research.chalmers.se/en/publication/46311.

[9] Ute Schmid. *Inductive Synthesis of Functional Programs, Universal Planning, Folding of Finite Programs, and Schema Abstraction by Analogical Reasoning*. Vol. 2654. Lecture Notes in Computer Science. Springer, 2003. ISBN: 3-540-40174-1. DOI: 10.1007/b12055. URL: https://doi.org/10.1007/b12055.

[10] Jianguo Lu, John Mylopoulos, Masateru Harao, and Masami Hagiya. “Higher order generalization and its application in program verification”. In: *Ann. Math. Artif. Intell.* 28.1-4 (2000), pp. 107–126. DOI: 10.1023/A:1018952121991. URL: https://doi.org/10.1023/A:1018952121991.

[11] Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. "ELPI: Fast, Embeddable, \lambda Prolog Interpreter". In: *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*. Ed. by Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov. Vol. 9450. Lecture Notes in Computer Science. Springer, 2015, pp. 460–468. DOI: 10.1007/978-3-662-48899-7_32. URL: https://doi.org/10.1007/978-3-662-48899-7%5C_32.

[12] Vladimir Voevodsky. *Mathematical theory of type theories and the initiality conjecture*. Research proposal to the Templeton Foundation. 2016. URL: https://www.math.ias.edu/Voevodsky/voevodsky-publications_abstracts.html#TempletonProposal.

[13] Philipp G. Haselwarter and Andrej Bauer. "Finitary type theories with and without contexts". In: *CoRR* abs/2112.00539 (2021). arXiv: 2112.00539. URL: https://arxiv.org/abs/2112.00539.

[14] Taichi Uemura. "A General Framework for the Semantics of Type Theory". In: *CoRR* abs/1904.04097 (2019). arXiv: 1904.04097. URL: http://arxiv.org/abs/1904.04097.

[15] Daniel Gratzer and Jonathan Sterling. "Syntactic categories for dependent type theory: sketching and adequacy". In: *CoRR* abs/2012.10783 (2020). arXiv: 2012.10783. URL: https://arxiv.org/abs/2012.10783.

[16] Herman Geuvers, Robbert Krebbers, James McKinna, and Freek Wiedijk. "Pure Type Systems without Explicit Contexts". In: *Proceedings 5th International Workshop on Logical Frameworks and Meta-languages: Theory and Practice, LFMTP 2010, Edinburgh, UK, 14th July 2010*. Ed. by Karl Crary and Marino Miculan. Vol. 34. EPTCS. 2010, pp. 53–67. DOI: 10.4204/EPTCS.34.6. URL: https://doi.org/10.4204/EPTCS.34.6.

[17] Thorsten Altenkirch, Ambrus Kaposi, András Kovács, and Jakob von Raumer. "Reducing Inductive-Inductive Types to Indexed Inductive Types". 24th International Conference on Types for Proofs and Programs (TYPES). 2018.

[18] Thorsten Altenkirch, Ambrus Kaposi, András Kovács, and Jakob von Raumer. "Constructing Inductive-Inductive Types via Type Erasure". 25th International Conference on Types for Proofs and Programs (TYPES). 2019.

[19] Fredrik Nordvall Forsberg. "Inductive-inductive definitions". PhD thesis. Swansea University, UK, 2013. URL: https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.752308.

[20] Thomas Ehrhard and Laurent Regnier. "The differential lambda-calculus". In: *Theoretical Computer Science* 309.1 (2003), pp. 1–41. ISSN: 0304-3975. DOI: https://doi.org/10.1016/S0304-3975(03)00392-X. URL: https://www.sciencedirect.com/science/article/pii/S030439750300392X.

[21] Dariusz Biernacki and Sergueï Lenglet. "Applicative Bisimulations for Delimited-Control Operators". In: 2012, pp. 119–134. DOI: 10.1007/978-3-642-28729-9_8. URL: https://doi.org/10.1007/978-3-642-28729-9%5C_8.

[22] Sergueï Lenglet and Alan Schmitt. "Howe's Method for Contextual Semantics". In: 2015, pp. 212–225. DOI: 10.4230/LIPIcs.CONCUR.2015.212.

[23] S. Lassen. "Bisimulation in Untyped Lambda Calculus: Böhm Trees and Bisimulation up to Context". In: *Proc. 15th Conference on Mathematical Foundations of Progamming Semantics*. 1999, pp. 346–374. DOI: 10.1016/S1571-0661(04)80083-5.

[24] Mark R. Shinwell, Andrew M. Pitts, and Murdoch Gabbay. "FreshML: programming with binders made simple". In: *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*. Ed. by Colin Runciman and Olin Shivers. ACM, 2003, pp. 263–274. DOI: 10.1145/944705.944729. URL: https://doi.org/10.1145/944705.944729.

[25] Matías Menni. "About N-quantifiers". In: *Appl. Categorical Struct.* 11.5 (2003), pp. 421–445. DOI: 10.1023/A:1025750816098. URL: https://doi.org/10.1023/A:1025750816098.

[26] Marcelo P. Fiore, Gordon D. Plotkin, and Daniele Turi. "Abstract Syntax and Variable Binding". In: *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*. IEEE Computer Society, 1999, pp. 193–202. DOI: 10.1109/LICS.1999.782615. URL: https://doi.org/10.1109/LICS.1999.782615.

[27] Marcelo Fiore and Dmitrij Szamozvancev. "Formal metatheory of second-order abstract syntax". In: *Proc. ACM Program. Lang.* 6.POPL (2022), pp. 1–29. DOI: 10.1145/3498715. URL: https://doi.org/10.1145/3498715.

[28] Peter Dybjer and Anton Setzer. "Induction-recursion and initial algebras". In: *Ann. Pure Appl. Log.* 124.1-3 (2003), pp. 1–47. DOI: 10.1016/S0168-0072(02)00096-9. URL: https://doi.org/10.1016/S0168-0072(02)00096-9.

[29] G. Brunerie. "On the homotopy groups of spheres in homotopy type theory". PhD. Université Nice Sophia Antipolis, June 2016. URL: https://hal.archives-ouvertes.fr/tel-01333601.

[30] F Marmolejo and R. Wood. "Monads as extension systems - No iteration is necessary". In: *Theory and Applications of Categories* 24 (Jan. 2010), pp. 84–113.

[31] Saunders Mac Lane. *Categories for the Working Mathematician*. 2nd. Graduate Texts in Mathematics 5. 1998. DOI: 10.1007/978-1-4757-4721-8.

[32] Stephen Lack. "Towards Higher Categories". In: vol. 152. The IMA Volumes in Mathematics and its Applications. 2010. Chap. A 2-categories companion. DOI: 10.1007/978-1-4419-1524-5.