Research Proposal: A practical extension of dependent type theory with nominal features

Jacek Karwowski

Abstract

We describe backgrounds of dependent type theory and nominal set theory, and introduce the question of combining them. Even though there were a number of approaches to date, for the general case of languages with name-swapping terms and locally-scoped names, none of them proved satisfactory. We outline two strategies to be investigated: making the *freshness* # operator propositional, and, treating the name abstraction as modal operator, borrowing the strong normalization and type-checking results from computational modal type theories.

Contents

1 Introduction

There are two main criteria for why a particular area in mathematics or computer science is worth pursuing: breath of connections to other theories - and practical usefulness. What new insights does it provide, and what new tools does it enable - these are the crucial questions that have to be asked.[1]

In this document, a central role is played by dependent type theory and nominal theories, so it makes sense to introduce them separately first. Keeping

[1]There is also a third criterion: of the subject innate aesthetics. To many - as the famous A Mathematician's Apology is arguing - the most important one, but also the most subjective and informal. As what we cannot speak about we must pass over in silence, so should this subject, however significant, remain hidden for the rest of this work.

the above two criteria in mind, I will describe what these theories are, what is the intuition behind them, and why they are important. After that, I will lay out a vision for combining them, presenting prior work and addressing the question of why and how new proposed approaches might work out. As a conclusion, I will sketch a brief execution plan of the intended research.

2 Dependent types

There are three ways a "simple" type system[2] can be extended: by allowing the terms to depend on types (polymorphism), the types to depend on types (type operators) and types to depend on terms (dependent types). Although the first two extensions are already widely implemented in the mainstream programming languages, the third one is still considered somewhat novel. However, the situation is starting to change.

The above description, even though technically correct, does not convey a very clear picture. To see the actual power of dependent types, one has to understand that they allow expressing arbitrary type constraints on terms. So, just by using the type system, we can express statements like "this program is correct" - and if the program type checks, it actually *will be correct*. Without any tests and without any doubt or uncertainty. But there is more to it. One can not only express the correctness of an actual result, but also any aspect of the computation itself. For example, by formalizing the notion of the run-time complexity, it is possible to obtain a type of sorting functions that will guarantee that such a function runs in $O(n \log n)$ time, having strong real-time guarantees proved formally.

Programming languages theory is not the only place that dependent types shine. They do have much more to offer - and, deservedly, they are seeing increasing interest from an unusually broad circle of both theoreticians and practitioners.

In the mathematics community: as a logic to express foundations of mathematics, as an internal language of locally cartesian closed categories in category theory, allowing for extremely general constructions without having to worry about higher coherences, recently - under the name of Homotopy Type Theory[13] - resulting in new proofs in algebraic topology and synthetic homotopy theory;

In the formalization and mechanization of mathematics community: using the Curry-Howard-Lambek correspondence, they form the basis for multiple proof assistants, including Agda, Coq, Lean and F*, which are in turn used to aid working mathematicians and ensure correctness of particularly challenging technical proofs (most notably, the proof of the four color theorem);

Outside academia, dependent types are used commercially in the setting of formal specification, secure implementations of cryptographic primitives (Project Everest[1]), languages formalization (RustBelt[8]), assuring critical systems correctness, development of financial markets software and low-level systems verification (CompCert C compiler[9]).

[2] "simple" as in "simply typed lambda calculus"

3 Nominal sets

Nominal sets make up another big theory at the intersection of mathematics and computer science. Very often in these disciplines, one has to deal with some kind of "structure" involving names - names of variables, labelling of vertices in a graph, names of symbols on a tape or in automata. From the point of view of the theory, two structures having the same structure and different labelling appear the same. This means that there is some mechanism needed to consider not such structures themselves, but their equivalence classes closed under names automorphisms.

At the same time, even though the set of names is actually infinite, "from the inside" of the theory the user only has access to a finite fragment of it. This gives the subject a unique flavour, expressed elegantly in its tongue-in-cheek characterisation as the study of *"Slightly infinite sets"*[3].

In mathematics, structures described above are usually called "sets with atoms" or "sets with urelements", and form the basis of ZFA theory (with the Fraenkel-Mostowski model) - an alternative foundations of mathematics framework.

In computer science, the name "nominal sets" is more widely used, and there is a wealth of applications using them. (To give an interesting example - in automata theory, we can get results for register automata for free, just by changing the foundations from the usual ZF to ZFA). However, our main focus will be on one particular situation - that of the bound names in programming languages. Most often, a presentation of a programming language or a type theory uses the convenient notation of named variables, introducing the notion of "α-equivalence" and saying that two terms with the same bound variables are considered the same. The problem that is being pushed under the rug is that, even though α-conversion (and so, capture-avoiding substitution) is easy to work with pen and paper, when it comes to implementation, does not do as well. The most common choice is to use particular representatives for these equivalence classes - formed by de Bruijn indices. Since the end user doesn't deal with operations such as substitution by himself, the coding scheme is fully opaque.

However, things begin to change when one starts working on a meta-level, such as when working with interpreters, compilers, DSLs, proof assistants, or with languages that have more powerful reflection capabilities. Then, implementing de Bruijn arithmetic becomes a cumbersome and error-prone process.

The problem is important in the whole of formal reasoning/programming languages area, and multiple solutions, such as [2], are being proposed. Our preferred solution is to use nominal type theory. By having a first-class notion of a "name" and operating on them on the level of types, we can get things like α-equivalence for free. Some work in this direction was already done by the FreshML Project[12]. What needs to be done now is to extend this work to the more general setting of dependently typed language.

4 How to combine them?

So, the questions we are dealing with, is: how to get both the power of dependent types and the ease-of-use of nominal types in one system. One could also ask, if

all the paeans above are true, how is it possible that no one had already solved the problem?

The most important work in that area is [10], published 4 years ago. The authors successfully extended the Martin-Löf type theory (a prototypical example of a theory with dependent types) with FreshML-style nominal types. But, there is a problem in there, and not an easy one. Foreshadowing the work that is referenced later, *"Realizing all these goals in a way that preserves the crucial (and fragile!) syntactic properties of dependent type theory is extremely subtle"*.

What is lacking is the the actual type-checker algorithm. Working in the context of powerful type systems, one is constantly balancing on the edge of undecidability - having too many rules destroys it, having too few hurts the system usability. This is what the "practical" bit in the title is all about - developing the actual algorithm for type-checking is the primary focus here.

Some other perspective is offered by [4]. It has the advantage of having strong normalization and decidable type-checking, but sacrifices locally-scoped names and the overall power of the system to do that ([10], Example 2.2).

We identify two promising directions that can be explored here.

4.1 Martin-Löf type theory

When Per Martin-Löf developed his type theory, he had a very unique insight into the nature of equality relation. What he recognized, is that there are actually two distinct, but heavily inter-weaved concepts hidden in there.

On the one hand, there is judgemental equality - an "equality by definition". We may say that $2 = 1 + 1$, but this is because the symbol 2 is *defined* to be equal to $1 + 1$. There is nothing to prove or disprove in there, just a fact to be stated. Judgemental equality can be thought as a "algorithmic equality", that the type-checker knows about, and which it can use in its reduction rules.

On the other hand, there is a more powerful notion of propositional equality. This is an actual relation having to do with the inhabitation of equality types, which can be proved or disproved, argued for or against - and this is where we put the undecidability of the system.

Currently in FreshMLTT, when dealing with the *freshness* relation - expressing the fact that a name is fresh in a context, we can only use the first - judgemental - approach. This, of course, severely limits the power of the system - but maybe if we had the second notion of "propositional freshness", we could, borrowing the intuition from MLTT, push the undecidability to the "user space".

4.2 Modal type theory

The second inspiration stems from recent advances in modal type theories. I do not intend to delve into the details of the topic, since they are not of the primary focus, but the important bit is that there was a successful line of work done this year, culminating in developing a practical implementation of modal types in the setting of Martin-Löf type theory[7] [3]

[3]Where "practical", as before, means that not only the theory was proposed, but the algorithm for type-checking was developed as well.

An investigation on how exactly modal and nominal theories are related was conducted in [6], where authors, working in the setting of categories with families, find that various modal operators can be defined as a dependent right-adjoint functor on the context category. This builds on the previous work on such formalization in FreshMLTT[10] and modal lambda calculus [5].

From the perspective of this research project, the value here is in the possible translation from modal to nominal on a more algorithmic level - that is, one that would allow us to transpose the actual type-checking algorithm (using normalization by evaluation) from one to the other.

5 Conclusion

I covered the reasons of why it would be good to have a unification of the two described theories, described where the problem is with the current approaches, and what are the possible sources of inspiration. Concretely, the work I am proposing to do, is to:

First, familiarize myself with the existing body of the work. Understand how does the *lock* operator from the modal type theory work, read up on the categorical semantics of modality, see how singular and bunched context approaches differ.

Then, try out the aforementioned ideas: express name abstraction as a modal operator syntactically. Investigate what needs to be changed for the reduction rules from [7] to work. The core difficulty is with substitutions: the authors of [7] acknowledge that, and, in the case of FreshMLTT, in general, term denotation is not invariant with respect to names substitution.

Next, try out the other avenue of introducing the freshness propositional relation instead of treating it as a statement about the term being invariant under the permutation of a name with a meta-theoretical fresh variable. See what should be the rules for dealing with such a type and how does it impact the usability.

Along with the theoretical ventures implement the experiments in Agda, so that, if any of these succeeded, there will be a verified proof-of-concept implementation ready.

After that, look in depth at the use cases and find a good syntax, most probably relying on FreshML implementation. Investigate the relation to (inductively defined) data types, as there are known issues strongly limiting usefulness ([11], Section 7.1.2)

Finally, implement the feature as a library to Agda, or, if the use-cases are mostly centered at some other ecosystem, re-implement the library there (in Coq, LEAN or F*).

References

[1] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay Lorch, et al. Everest: Towards a verified, drop-in replacement of https. In *2nd Summit on Advances in Program-*

ming Languages (SNAPL 2017). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[2] Jasmin Christian Blanchette, Lorenzo Gheri, Andrei Popescu, and Dmitriy Traytel. Bindings as bounded natural functors. *Proc. ACM Program. Lang.*, 3(POPL):22:1–22:34, January 2019.

[3] Mikołaj Bojanczyk. Slightly infinite sets. *A draft of a book available at https://www. mimuw. edu. pl/˜ bojan/paper/atom-book*, 2016.

[4] J. Cheney. A dependent nominal type theory. *Logical Methods in Computer Science*, 8:(1:08), 2012.

[5] Ranald Clouston. Fitch-style modal lambda calculi. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures*, pages 258–275, Cham, 2018. Springer International Publishing.

[6] Ranald Clouston, Bassel Mannaa, Rasmus Møgelberg, Andrew Pitts, and Bas Spitters. Modal dependent type theory and dependent right adjoints, 04 2018.

[7] Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. Implementing a modal dependent type theory. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–29, Jul 2019.

[8] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):66, 2017.

[9] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363, 2009.

[10] Andrew M. Pitts, Justus Matthiesen, and Jasper Derikx. A dependent type theory with abstractable names. *Electronic Notes in Theoretical Computer Science*, 312:19–50, Apr 2015.

[11] Mark R Shinwell. The fresh approach: functional programming with names and binders. Technical report, University of Cambridge, Computer Laboratory, 2005.

[12] Mark R. Shinwell, Andrew M. Pitts, and Murdoch J. Gabbay. Freshml. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming - ICFP '03*. ACM Press, 2003.

[13] The Univalent Foundatiosn Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. http://homotopytypetheory.org/book/, 2013.