PhD Exposé – Soft Type Systems For Interactive Theorem Proving

Kevin Kappelmann

Abstract

Most interactive theorem provers are based on a rigid kind of type theory. In such systems, typing rules are fixed once and for all and users are bound to formlise their work subject to these rules. An alternative is to use soft type systems. In such systems, user can specify new typing rules at any time without threatening soundness of the prover. Soft types have the potential to make proof assistants more flexible and expressive. Yet, they are heavily underexplored. In my PhD project, I want to assess the practicality of soft type systems and lay the foundations for a modern soft type framework for interactive theorem provers.

1 Introduction and Motivation

The establishment of mathematical truths rests on the discovery of proofs. In simple terms, one may think of a proof as a coherent chain of logical arguments leading from a collection of premises to a conclusion. This, however, leads to many questions: When is an argument "logical"? What rules and premises are there to begin with? Some of the most distinguished mathematicians intensively debated these questions in the beginning of the 20th century, and the rigorous, axiomatic foundations of modern mathematics are the result of these debates [20].

However, on a daily basis, contemporary scientists do not publish complete formal proofs with respect to some axiomatic foundation, which often requires tedious low-level reasoning, but rather use reasoning principles of a higher level. A proof written this way is accepted if the community agrees that the reasoning at hand can, in theory, be translated to an accepted axiomatic basis. In practice, however, this translation is not spelled out, and the proof remains susceptible to human error.

Interactive theorem provers (also known as proof assistants), such as Isabelle [14], Coq [17], and Lean [13], try to close this gap by providing a computer-checked form of mathematical reasoning. At their core, they all rely on the axiomatic method of mathematics. As such, they fix a set of axioms on top of which users can axiomatise and define new concepts and prove theorems.

Most interactive theorem provers are based on some form of *type theory*. Simply speaking, types act as categorisations of mathematical values and structures. The natural numbers ($\mathbb{N}$), functions on integers ($\mathbb{Z} \Rightarrow \mathbb{Z}$), and the collection of $m \times n$ matrices in a given domain ($D^{m \times n}$) are examples of types.

Systems based on type theory use a set of *typing rules* and a *type checker*. The typing rules are the axioms of the type system, specifying which expressions may belong to each type. They are usually stated in a declarative form. The type checker is the algorithmic counterpart of these declarative rules. It takes a term t and a type T and checks, according to the typing rules, whether t is of type T, which we denote by $t : T$.

Type systems are a very helpful mechanism, preventing users from common mistakes, such as applying a function to a value outside its domain. However, as mentioned by Krauss [11], they deviate from the de facto standard in mathematics, where a set-theoretic

foundation is taken for granted. Moreover, virtually all major proof assistants take their set of typing rules and their type checker as a fixed primitive. When a mathematical concept is not easily expressible using these primitives, types can become a serious limitation (see Krauss [11] for some examples).

While many of these limitations could be addressed by appropriately modifying the type system at hand, this rarely happens in practice: First, the type system builds the axiomatic basis of said proof assistants. Changing it hence raises meta-theoretical soundness concerns. Second, modifying the core of such systems requires significant implementation work. Third, no set of typing rules is perfect and users thus may wish to switch from one set of rules to another one without introducing incompatible foundations.

I propose to address these limitations by developing a *soft type*[1] framework for proof assistants. Unlike traditional "hard" type systems, soft type systems do not take the notion of a type as primitive. Instead, they reduce it to existing concepts of the underlying logic. In their most simple form, soft types can be defined as predicates on terms: a term t is of soft type T if and only if $T(t)$. Moreover, a typing rule simply becomes a proven inference rule in the underlying logic. This way, types and typing rules become first-class objects, and type checking nothing more than theorem proving inside the given logic.

As explained by Krauss [11], soft type systems provide an extremely flexible and powerful framework: user can add and remove types and typing rules anytime, and they can do so without changing the foundation of the system. Treating types as predicates also more closely resembles the way mathematicians think about types in set-theoretic frameworks, as explicated by Harrison [9]. For example, given the set of natural numbers $\mathbb{N}$, one can define the type of natural numbers as the predicate $\texttt{Nat}\ n \coloneqq n \in \mathbb{N}$. Moreover, soft types allow for an inclusive interpretation of subtyping (in which $t : A$ and $A <: B$ implies $t : B$) rather than an coercive one (where terms of a subtype need to be lifted with coercion functions). As an example, $n : \mathbb{N}$ does not imply $n : \mathbb{Z}$ in usual hard type systems but the implication does hold in a suitable soft type system based on set theory.

2 State of the Art

Interactive theorem provers can broadly be classified into

1. the higher-order logic provers based on simply-typed lambda calculus (e.g. HOL-Light, Isabelle/HOL),
2. the dependent type provers (e.g. Agda, Coq, Lean), and
3. the provers based on traditional set theory (e.g. Isabelle/ZF, Metamath, Mizar).

By design, proof assistants of the first two categories are based on hard type systems of differing strengths and weaknesses. Though they proved quite successful and received more attention than their set-theoretic counterparts, they all suffer from the limitations sketched in Section 1. Proof assistants of the third category, in contrast, lend themselves particularly well to soft type systems. However, most such provers stay in a completely untyped or at best very restricted soft type framework.

A notable exception is Mizar [1]. Mizar is based on an untyped set theory but nevertheless supports the notion of types. Mizar users can define new types by defining predicates on sets. Its typing rules are not fixed axiomatically but rather are specified and proven to be sound with the help of the user in a controlled manner. Mizar pioneered the realm of soft types and has a remarkable library of formalised mathematics [2]. However, Mizar falls short in important aspects:

[1] The term originates from the programming language community [4], where it has a different meaning: it is the principle of statically type-checking as much as possible while inserting run-time checks for what cannot be statically checked.

1. It uses a fixed type checker which is not proof-producing but part of the trusted kernel [19]. As such, it cannot be modified without raising meta-theoretical soundness concerns. Moreover, Mizar is closed-source, and as such, users have no possibility to modify or replace the type checker.
2. It only accepts typing rules adhering to a strict format (so-called "registrations").
3. It is neither interactive nor programmable; as such, it is more akin to being "just" a proof verifier rather than an interactive theorem prover.
4. It uses idiosyncratic type-theoretic concepts and terminology, rendering its usage difficult to both computer scientists and mathematicians.

To the best of my knowledge, there is no other significant work towards a rich soft type framework for proof assistants.

3 Objectives and Research Questions

My objective is to lay the foundations for a modern soft type framework for interactive theorem provers. First, this requires a deep understanding of state-of-the-art hard type systems and formulating appropriate generalisations thereof that can deal with the more flexible soft type approach. Second, this requires proving practicality of the approach and contrasting it with existing frameworks employed by proof assistants by means of a workable implementation and appropriate case studies.

As part of this endeavour, a number of different research questions arise, such as:

- Which requirements should a modern soft type system meet? Which concepts known from hard type systems should be integrated? Which ones have to be generalised? Which additional concepts have to be introduced?
- Can we build a soft type system which is flexible, predictable, and sufficiently efficient?
- Which structures (e.g. sets, lambda terms) lend themselves well for soft type systems? Which soft typing rules form a practical basis for these structures?
- Some concepts, such as function spaces, may be expressible in both the underlying logical foundation (e.g. simply-typed lambda calculus) and the structures on top of which the soft type system is employed (e.g. sets). Can we efficiently move between both worlds?
- How can we integrate soft types in existing proof assistants in a user-friendly manner?
- Can soft types effectively help us expressing concepts that usually are impossible, or at least difficult, to formulate using just the underlying logic of a given proof assistant (e.g. using dependent soft types in HOL provers)?

Given the complexity and extent of modern research in type theory and proof assistants, I acknowledge that it is not realistic to answer all of these questions in my PhD project, but I aim to take major steps in this direction.

4 Methodology and Project Plan

Krauss and Chen [5] started a first prototype of a soft type system in Isabelle/HOL, which I joined at the beginning of my PhD. We built a simple soft type framework in which soft types are identified by predicates on terms. To test the framework, we first axiomatised a Tarski–Grothendieck set theory inside Isabelle/HOL. Relative consistency of such a theory has been shown by Brown et al. [3]. Second, we built a basic untyped library for this set theory, inspired by the work in Isabelle/ZF [15]. Third, we combined the untyped set theory with our soft type framework and built a small mathematical library that uses the idea

of soft types on sets. This library is called Isabelle/Set and is continuously updated on GitHub [5].

Let me explain our choices.

- We use Isabelle because:
  1. It supports multiple object logics. This allows us to test the flexibility and practicality of the soft type framework with respect to different underlying foundations (e.g. higher-order and first-order logic).
  2. It is one of the most prominent proof assistants, offering large proof libraries and good automation and programming documentation.
  3. It is co-developed at the Technical University of Munich and both my supervisor Prof. Nipkow and I have experience using it.
- We built a library in Isabelle/HOL rather than Isabelle/ZF because:
  1. The first-order nature of Isabelle/ZF prevents us from expressing soft types that require higher-order quantification (such as the second-order function $(\mathbb{N} \Rightarrow \alpha) \Rightarrow \alpha$). While in theory, such functions can be internalised in the set theory, this is not done in Isabelle/ZF and rewriting it to do so requires substantial work.
  2. Isabelle/HOL provides mechanisms which can prove valuable to the development of a soft type system. For example, a generalisation of the transfer and lifting packages [10] may be used to move between related softly-typed structures within the set theory.
- The choice of a Tarski-Grothendieck set theory is not crucial and could be replaced by other sufficiently strong theories.

With this prototype at hand, we were able test the feasibility of the approach and discovered new challenges. At this point, Krauss and soon after also Chen stepped back from the project and I became the sole researcher and developer of the project.

Following further analysis and an extensive phase of literature research, I identified the following problems:

- The current type checker is not strong enough: It is based on a simplification of the algorithm used in Mizar. However, the typing rules we use in Isabelle/Set are not restricted in the same way as Mizar's typing rules are. As such, many type checking problems that user's expect to be trivial cannot be discharged and are left for manual proof.
- The current type checker and synthesiser are two separate algorithms. This approach differs from modern programming languages and proof assistants, which use *bidirectional typing* instead. Bidirectional typing supports features for which pure type synthesis is undecidable [7]. This reduces the type annotation burden put on users.
- The current type checker and synthesiser are not flexible enough: the latter expects a certain form of function types and both can barely be extended or guided by users without touching the Isabelle/ML code.
- We developed a *set extension* mechanism that, for example, allows us to construct a set-theoretic number system with the wanted inclusions $\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$. As part of this process, a set extension takes a set A and creates a new set A' together with a set-isomorphism $f : A \to A'$. One then wishes to lift existing definitions and theorems on A to A' using f. However, the current lifting package of Isabelle/HOL does not support this use case since f is not right-total and right-unique on the underlying Isabelle/Pure type of A'.
- In certain cases, users must[2] – or may wish to – internalise Isabelle/Pure functions to set-theoretic functions. At the moment, users have to do this by hand. Automation would greatly improve the user experience.

[2]for example, when encoding type classes as sets

In the next steps of my PhD, I propose to address these problems as follows:

- The type checker and synthesiser are unified into a bidirectional typing framework. Inspired by state-of-the-art work on hard type systems [6, 8, 12, 16, 18] and based on my experience with Isabelle/Set, I propose to build the framework on a constraint logic programming basis. The framework will be extensible – users can register new constraint solvers at any time – and controllable – users can add, remove, and adjust the priorities of typing rules. To my understanding, most hard type systems can be encoded as a constraint logic program. I plan to stress this by providing examples of such encodings using my framework.
- The lifting package of Isabelle/HOL can be generalised to support the cases arising as part of the set extension mechanism of Isabelle/Set. Together with Juli Gottfriedsen, I already generalised the required theory and proved its correctness in unpublished work. As a next step, a generalisation of the lifting algorithm will be implemented and its usability tested in a case study. As part of this study, I expect to categorise classes of subproblems that are generated when applying the generalised lifting algorithm. Finding appropriate automation to solve these subproblems follows as a next step. I moreover plan to investigate whether the lifting package can be useful when dealing with the internalisation problem mentioned above.
- Having both the new soft type framework and lifting package at hand, I plan to assess the practicality of the resulting system by conducting a formalisation case study that requires advanced type-theoretical concepts (dependent and intersection types, type classes, etc.). The formalisation will be contrasted with those of corresponding developments in other proof assistants. The results of this study will indicate future directions and possible improvements of the framework.

Let me stress that, while the practical implementation of my work will be done in the Isabelle proof assistant, the theoretical knowledge gained during the project will be applicable to interactive theorem provers and the type theory community in general.

Finally, the project schedule of my PhD project can be found in Figure 1. I want to note, reflect, and acknowledge that the initial phase (Q4 2019 – Q2 2021) of my project did not progress as quickly as one would likely expect from a PhD project. The reasons are as follows:

- From Q4 2019 – Q2 2021, I was one of the main supervisors for various mandatory courses of the Informatics Bachelors degree with ≈ 1100 students/course. Supervision of these courses has been particularly time-consuming due to the Corona pandemic.
- From Q1 2021 – Q2 2021, I had to complete courses worth 15 ECTS due to formal requirements to get accepted into the TUM graduate school.
- While it might have been possible to publish a first paper about the initial prototype of Isabelle/Set, I decided against it and instead doubled down on analysing the problems of the prototype and reviewing the literature. I followed this path because I was not satisfied with the results at hand and believed that publishing a first weak report on Isabelle/Set would have diminished interest of the research community in follow-up papers and hence the overall research impact of the project.

Project schedule	2019	2020				2021				2022				2023				2024		
	Q4	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4	Q1	Q2	Q3
Study theoretical background																				
Prototype of Isabelle/Set																				
Analysis of prototype and literature review																				
Theoretical development of lifting generalisation																				
Implementation of bidirectional typing framework																				
Evaluation of typing framework and first paper																				
Implementation of lifting generalisation																				
Case study of updated Isabelle/Set																				
Evaluation of Isabelle/Set and second paper																				
Thesis writing and presentation																				

Figure 1: Project schedule

References

[1] Grzegorz Bancerek, Czesław Byliński, Adam Grabowski, Artur Korniłowicz, Roman Matuszewski, Adam Naumowicz, Karol Pak, and Josef Urban. Mizar: State-of-the-art and beyond. In *International Conference on Intelligent Computer Mathematics*, pages 261–279. Springer, 2015.

[2] Grzegorz Bancerek, Czesław Byliński, Adam Grabowski, Artur Korniłowicz, Roman Matuszewski, Adam Naumowicz, and Karol Pąk. The Role of the Mizar Mathematical Library for Interactive Proof Development in Mizar. *Journal of Automated Reasoning*, 61(1):9–32, Jun 2018. ISSN 1573-0670. doi: 10.1007/s10817-017-9440-6. URL https://doi.org/10.1007/s10817-017-9440-6.

[3] Chad E Brown, Cezary Kaliszyk, and Karol Pak. Higher-order Tarski Grothendieck as a foundation for formal proof. In *10th International Conference on Interactive Theorem Proving (ITP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[4] Robert Cartwright and Mike Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 278–292, 1991.

[5] Joshua Chen, Kevin Kappelmann, and Alexander Krauss. Isabelle/Set. https://github.com/kappelmann/Isabelle-Set, 2022.

[6] Leonardo De Moura, Jeremy Avigad, Soonho Kong, and Cody Roux. Elaboration in dependent type theory. *arXiv preprint arXiv:1505.04324*, 2015.

[7] Jana Dunfield and Neel Krishnaswami. Bidirectional typing. *ACM Computing Surveys (CSUR)*, 54(5):1–38, 2021.

[8] Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. Implementing type theory in higher order constraint logic programming. *Mathematical Structures in Computer Science*, 29(8):1125–1150, 2019.

[9] John Harrison. Let's make set theory great again. Talk at the Artificial Intelligence and Theorem Proving Conference, 2018.

[10] Brian Huffman and Ondřej Kunčar. Lifting and Transfer: A modular design for quotients in Isabelle/HOL. In *International Conference on Certified Programs and Proofs*, pages 131–146. Springer, 2013.

[11] Alexander Krauss. Adding Soft Types to Isabelle, 2010. URL https://www21.in.tum.de/~krauss/papers/soft-types-notes.pdf.

[12] Francesco Mazzoli and Andreas Abel. Type checking through unification. *arXiv preprint arXiv:1609.09709*, 2016.

[13] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.

[14] Lawrence Paulson. Isabelle - A Generic Theorem Prover. In *Lecture Notes in Computer Science*, 1994.

[15] Lawrence C Paulson and Krzysztof Grabczewski. Mechanizing set theory. *Journal of Automated Reasoning*, 17(3):291–323, 1996.

[16] Enrico Tassi, Claudio Sacerdoti Coen, Wilmer Ricciotti, and Andrea Asperti. A bi-directional refinement algorithm for the calculus of (co) inductive constructions. *Logical Methods in Computer Science*, 8, 2012.

[17] The Coq Development Team. The Coq Proof Assistant. Jan 2022. doi: 10.5281/zenodo.5846982. URL https://doi.org/10.5281/zenodo.5846982.

[18] Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. OutsideIn(X): Modular type inference with local assumptions. *Journal of Functional Programming*, 21:333–412, September 2011. URL https://www.microsoft.com/en-us/research/publication/outsideinx-modular-type-inference-with-local-assumptions/.

[19] Freek Wiedijk. Mizar's soft type system. In *International Conference on Theorem Proving in Higher Order Logics*, pages 383–399. Springer, 2007.

[20] Richard Zach. Hilbert's Program. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Fall 2019 edition, 2019.