Undefinedness and Soft Typing in Formal Mathematics

PhD Research Proposal
by Jonas Betzendahl

January 7, 2020

Supervisors:
Prof. Dr. Michael Kohlhase
Dr. Florian Rabe

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

Abstract

During my PhD, I want to contribute towards more natural reasoning in formal systems for mathematics (i.e. closer resembling that of actual mathematicians), with special focus on two topics on the precipice between type systems and logics in formal systems: undefinedness and soft types.

Undefined terms are common in mathematics as practised on paper or blackboard by mathematicians, yet few of the many systems for formalising mathematics that exist today deal with it in a principled way or allow explicit reasoning with undefined terms because allowing for this often results in undecidability.

Soft types are a way of allowing the user of a formal system to also incorporate information about mathematical objects into the type system that had to be proven or computed after their creation. This approach is equally a closer match for mathematics as performed by mathematicians and has had promising results in the past.

The MMT system constitutes a natural fit for this endeavour due to its rapid prototyping capabilities and existing infrastructure. However, both of the aspects above necessitate a stronger support for automated reasoning than currently available. Hence, a further goal of mine is to extend the MMT framework with additional capabilities for automated and interactive theorem proving, both for reasoning in and outside the domains of undefinedness and soft types.

Contents

1 Introduction & Motivation **4**

2 State of the Art **5**
2.1 Undefinedness in Mathematics . . . 5
2.1.1 Common Sources of Undefinedness . . . 6
2.1.2 Non-consensus by Example . . . 8
2.1.3 Existing Approaches to Undefinedness in Formal Systems . . . 8
2.2 Type Systems for Mathematics . . . 11
2.2.1 Soft Types . . . 12
2.3 Logical Frameworks and MMT . . . 14

3 Research Objectives **14**
3.1 Undefinedness . . . 14
3.2 Soft Type Systems . . . 15
3.3 Reasoning . . . 16

4 Methodology **16**
4.1 Extending MMT with **LFU*** . . . 16
4.1.1 Farmer's Undefinedness . . . 16
4.1.2 Other Undefinedness Approaches . . . 16
4.1.3 Test: Reimplementing Logics with Undefinedness . . . 17
4.2 Soft Type Systems for MMT . . . 17
4.2.1 Subtyping and Soft Type Systems . . . 17
4.2.2 Test: Improving Interaction with Soft-Typed Systems . . . 18
4.3 Automatic Dismissal of Proof Obligations . . . 18
4.3.1 LambdaProlog & the ELPI System . . . 18
4.3.2 Test: Tableaux Prover . . . 19
4.3.3 Test: Softly Typed Set Theory . . . 19
4.3.4 Test: IMPS Proof Obligations . . . 20
4.4 Work-plan and Timeline . . . 20

1 Introduction & Motivation

Formal systems are on the rise! From the design of safety-critical hardware chips [FLH18] to operating system kernels [Kle+09] to musical improvisation [Don+14], more and more disciplines and researchers in industry and academia are relying on formal methods to detect errors and verify results.

This trend is also present in mathematics, including both the development of software systems to formalise mathematics in (such as `Coq`, Isabelle/HOL and numerous others), and the construction and verification of proofs. Notable successful ventures in the latter category include the proof of the Kepler conjecture [Hal+17] and the classification theorem of finite groups of odd order [Gon+13].

One aspect of mathematics that has so far not been widely incorporated into formal systems is undefinedness, the notion of terms or expressions that are syntactically well-formed but cannot be assigned a value. This typically arises when applying a *partial* function (or predicate, or operator, . . .) to a value that lies outside of its domain.

There are many reasons to take undefinedness seriously, however. The problem has been discussed academically for more than a century (see, for example, [Rus05]) and it is clear that the assumption that all expressions can be treated as defined (even though it would be terribly convenient) is ultimately indefensible both for mathematical terms (consider expressions like $\sqrt{(1 \div x^2) - 5}$ or $\ln -x^3$) and natural language (consider expressions like “The present king of France is bald.” while France is still a republic or “Zimbabwe lies to the east of the equator.”).

Indeed, mathematicians routinely manipulate and reason about expressions like the following, long before finding out (if they find out at all) whether or not they are, in fact, defined.

$$\int_a^b f(x)\,dx \qquad \lim_{x\to\infty} f(x) \qquad \sum_{n=0}^{\infty} a_n x^n \qquad \dots$$

This is less problematic in mathematics as practised with chalk on a blackboard, since the question can usually be postponed and if the expression actually does turn out to be undefined, one can address the fact in prose or find a different way around it. In formal systems however, assumptions about definedness and violations of these assumptions hit harder and are not as forgiving. In the worst case, they could lead to the system becoming inconsistent due to exploits using undefined terms.

Given all of the above, it becomes clear that undefinedness is mathematically interesting and modern formalisation efforts should include it as a concern and allow the user to reason about it, yet only very few systems actually do.

Type systems have long been and remain a popular choice for mathematical formalisms. We know that, via the Curry-Howard-Isomorphism [CHI; SU98], types correspond directly to mathematical propositions (with values inhabiting a type corresponding to proofs of the proposition). Having a sound type system for your formalisation effort then opens the door to various desirable properties, such as mechanised checking of correctness and many other properties of the mathematical objects in question as well as helpful documentation for the human users and designers.

When studying different formal systems for mathematics, one notices that almost all of them opt for a *decidable* type system. This is not a surprising choice, since it comes with additional beneficial guarantees. However, this means that many desirable features (e.g. behavioural subtyping, see [LW94]) can either not be implemented at all in these type systems or require awkward type-level encoding which takes the formalisation away from the source being formalised.

Another trend in type systems for mathematical formalisms is that many of them make use of *hard* types, as opposed to *soft*[1] ones. We will discuss the precise definition in Section 2.2; suffice it to say at this point that while hard types perhaps come more naturally to many, and with certain benefits, soft ones are also desirable especially when formalising mathematics. In particular, any system that wants to offer great flexibility with for designing formalisms should offer a choice of either, or even the ability to mix and match as the situation requires.

2 State of the Art

2.1 Undefinedness in Mathematics

When discussing undefinedness[2], especially not just in chalk-and-blackboard mathematics but also in formal, computerised systems, it is helpful to have a clear distinctions between *terms* and their *denotation*. In essence, a term is an expression in some language and its denotation is the meaning of the expression, i.e. the object it refers to.

Informally, a term is undefined, if it cannot be assigned a "natural" meaning. However, formally, it of course still can be assigned *a* value. For example, some formal systems give an arbitrary value (e.g. 0) to divisions by zero (see below). If the term is not assigned any value at all, however, it is called *non-denoting*.

The precise definition of these notions has been debated by many and for well over a century. Gottlob Frege, in [Fre92], distinguishes between the *sense* and the *denotation* of a phrase. The sense here being a way of conceiving the denotation, a mental map. Even if two expressions have the same denotation (i.e. refer to the same thing in the world), their senses might be different. For example, the names 'Samuel Clemens' and 'Mark Twain' have different senses, but the same denotation. The names 'Bruce Wayne' and 'Batman' have different senses, but no denotation.

Bertrand Russel gives a different theory of denotation in [Rus05] which (among other ideas and criticisms of Frege) puts forth the notion that expressions that contain denoting phrases without a denotation (e.g. "the greatest prime number" or "the present King of France") are not nonsensical, but indeed false, a perspective that we will encounter again later. This also avoids the problem of "truth-value gaps", allowing to stay true to a traditional two-valued logic.

Russel's theory has been criticised in it's own right (see, for example, [Str73]), and the criticism has in turn be criticised, a debate that continues even to this day. The interested reader is referred to contributions such as those by Marco Ruffino [Ruf17] or Francis Pelletier and Bernard Linsky [PL08]. It is not a goal of this proposal to judge which of these interpretations is the most correct or the most useful. However, the abundance of different perspectives on the subject does support the point that a given formal system should be flexible enough to adapt to the situation at hand and not just pick a favourite and commit solely to that.

There are different approaches to dealing with undefinedness that have been studied in philosophy, mathematics, logic (often under the name of "free logics", i.e. logics free from existence assumptions) and formal systems. According to Solomon Fefermann [Fef95], in philosophical logic, there are two distinct, yet similar perspectives on the precise "phrasing" of undefined

[1] It bears mentioning that the terminology of *soft types* is overloaded, which can lead to confusion. In this proposal, we will stick to the mathematical perspective of types for mathematical objects à la `Mizar`, and not to the programming perspective, where soft typing is sometimes used synonymously with weak typing, i.e. making little or no use of typing rules at compile time.

[2] What does or does not count as an "undefined" expression in mathematics is sometimes a matter of consensus and can shift over time. For example, in European mathematics, negative numbers were still considered absurd or nonsensical even until the 18th century [Mar06].

terms. Assuming a free variable x ranges over the domain of discourse D, then the first perspective understands asserts that not all elements of D can be expected to exist. A separate definedness predicate, called $E(x)$, is introduced that holds for x only if x actually exists (sometimes, the notation $x\downarrow$ is used instead, which also allows for the symmetric notation of $x\uparrow$ for "x is undefined"). The other perspective posits that all elements of D are assumed to exist by virtue of being in D alone. As a consequence $E(x)$ always holds. For any term or description t, however, $E(t)$ only holds if and only if t has a referent in D.

With both of these perspectives, it becomes necessary to introduce some changes to axiom systems. For example, the axiom of universal instantiation is adapted to the following by introducing an existence restriction:

$$\forall x A(x) \wedge E(t) \rightarrow A(t)$$

These changes in the axiom systems can lead to problems further down the line in the use of the system that makes the choice, as we will see when we will return to this discussion later.

Given the plethora of different approaches and opinions on undefinedness, it is no surprise that systems for doing mathematics reflect a similar diversity of thought. There have been multiple efforts, both in theory and in implementation, to come up with reasoning systems that support some sense of mathematical undefinedness or another. Bill Farmer also gives an overview of several potential approaches in [Far90]. We will also discuss existing systems and their way of dealing (or their way of avoiding to deal) with undefinedness in Section 2.1.3.

2.1.1 Common Sources of Undefinedness

"Partial functions arise naturally"
– CASL User Manual [BM04]

Partial Functions When we talk about partial functions (as opposed to *total* functions), we mean functions that are not defined on all values inside their domain. If they are applied to such a value, the resulting term is undefined. Ultimately, all undefinedness in mathematical practice arises because a partial function (or a partial predicate, or a partial operator, ...) is being applied to a value outside of its domain. Hence, the sections following this one can be understood as special cases of partial function applications that are prominent enough to be considered separately.

The formalisation of partial functions can take multiple shapes. The straightforward way would be to make the total functions a proper subtype of partial functions. However, partial functions can also be recovered from total functions via the use of option types (a partial function $A \rightarrow B$ is a total function $A \rightarrow \texttt{Option}[B]$). There is also the possibility to have to disjoint function type constructors (say, $\rightarrow$ for total and $\rightharpoonup$ or $\rightarrow ?$ for partial functions). This is indeed something that many actual mathematicians also do in their work as well, but would introduce the need for a lot of extra bookkeeping and also requires the user to know upfront whether the function they are defining is partial or total, a requirement that runs contrary to soft types, which we will talk about later.

Another possibility would be a system-wide automatic and invisible wrapping and unwrapping of option types. It would be possible for a formal system to represent all functions $f : A \rightarrow B$ as functions $f' : \texttt{Option[}A\texttt{]} \rightarrow \texttt{Option[}B\texttt{]}$ internally (both domain and co-domain need to be option types so that the functions remain composable) and automatically translate between the two representations. However, this is not only a departure from mathematical practice, the implicit conversions between types can also introduce problems with the formalisation itself and make it impossible to talk about this undefinedness from within the system.

Arithmetic Operations Maybe the most common and most intuitive source of undefinedness in mathematics are undefined arithmetic operations such as division by zero.

In `Mathematica`, the result of $\frac{1}{0}$ is `ComplexInfinity` (denoted $\tilde{\infty}$), which is a special value added to the complex plane $\mathbb{C}$ to yield the *extended* complex plane $\mathbb{C}^* = \mathbb{C} \cup \{\tilde{\infty}\}$. This approach, while completely mathematically sound, seems rather ad-hoc for this particular problem. Many users might also not expect to find themselves working in the extended complex plane without having specified it.

Some theorem proving systems such as `Coq` and LEAN opt to define division by zero as an "arbitrary" number, which in turn can be either known (a common choice in this case is 0) or unknown. This does not introduce inconsistencies into the logic, since it does not give the number a multiplicative inverse, it just touches the definition of division at that one point. The field axioms are not being violated [Way18]. However, while not introducing outright inconsistencies, it remains possible to introduce *surprises*[3] in certain systems that take this approach, as Cliff B. Jones warns in [Jon95].

Non-existent Limits Another important source of undefinedness in mathematics and therefore in formal systems for mathematics are limits that are not actually defined. Consider the following functions, that do not have limits for $x = 0$ and anywhere, respectively:

$$f(x) = \begin{cases} \sin \frac{5}{x-1} & \text{for } x < 1 \\ 0 & \text{for } x = 1 \\ \frac{0.1}{x-1} & \text{for } x > 1 \end{cases} \qquad g(x) = \begin{cases} 1 & \text{for } x \text{ rational} \\ 0 & \text{for } x \text{ irrational} \end{cases}$$

This source of undefinedness is so prominent, that both Bill Farmer with [Far04] as well as Sacerdoti Coen with [SCZ07] have published case studies on formalising undefinedness with special focus on calculus, where limits appear very frequently and their existence or non-existence is often a topic of mathematical discussion, more so than in other areas.

Non-Terminating Computation A term that describes a diverging computation (i.e. one that does not produce a result in finite time, e.g. a recursive search on an infinite list) is also often considered to be undefined.

This source of undefinedness introduces some unique challenges. Since the halting-problem in its general form is undecidable, it is not possible to make a clear-cut decision on whether any given computation will ever finish. Systems may use a "time-out" mechanism for convenience, which however might introduce unwelcome side-effects. A computation that times out on one computer might not on a more powerful system. Such inconsistencies might be rare, but could present significant challenges to a given formalisation effort.

Other Sources of Undefinedness There are, of course, many more potential sources of undefined expressions (such as definite description operators called with a predicate that does not name a unique member of the domain (e.g. $\iota\ n : \mathbb{N}\ .\ n < 0$)). We will not venture to give an exhaustive list of all mathematical contexts in which undefinedness can arise in this proposal, especially in light of the fact that ultimately, all undefinedness comes from some sort of language construct (be it a partial function, a predicate, ...) applied outside of its actual domain.

[3] If partial functions give an indeterminate element of their range if applied outside their domain and the system also allows to limit the range of a function to a range with only one unique element, the application can become determinate, even if the function is applied outside its domain.

2.1.2 Non-consensus by Example

We will now take a closer look at a few *specific* instances of potentially undefined expressions that will reappear throughout the rest of the proposal to illustrate certain points.

Equation (1) is especially interesting because it clearly demonstrates the split opinions about how to deal with undefinedness.

$$\forall x, y, z : \mathbb{R} \,.\, x = \frac{z}{y} \Rightarrow x \cdot y = z \tag{1}$$

For example, [Far90] and [KK94] both reference this example, but with contradictory connotations. The former paper posits that (1) should be considered true, even without the restriction that $y \neq 0$, the latter emphasises that such a restriction must be present. Both pieces refer to mathematical consensus in one fashion or another to justify their respective approach. This demonstrates not only that there is no mathematical consensus about when exactly equations like the one above should be treated as defined, or that such a consensus is not as obvious as previously thought, but that the situation is actually even worse than that. There is no consensus, yet people believe there is. This is another reason in favour of a flexible system that allows for different approaches instead of one unified one that captures whatever "apparent consensus".

A similar lesson can be learned from the following equation, introduced by Darvas, Metha and Rudich in their paper [DMR08]:

$$\frac{x}{y} = c_1 \ \wedge \ factorial(y) = c_2 \ \wedge \ y > 5 \tag{2}$$

where x and y are variables and c_1 and c_2 are constants. The authors introduce this formula as "well-defined" and state that it "always evaluates to either true or false". They justify this by case analysis of the last conjunct of the formula, arguing that if y is 0 or negative[4], the formula is already known to evaluate to false by the condition $y > 5$.

This may come as a surprise, since many definitions of definedness (such as that used in [Far90]) conclude a compound expression to be undefined if *any* of its subexpressions are undefined and do not consider the "short-circuiting" of logical connectives. Andrzej Blikle also calls attention to this in [Bli88] (in the context of software verification) as "strict" and "lazy" treatment of undefinedness respectively and argues that lazy undefinedness is often more convenient.

2.1.3 Existing Approaches to Undefinedness in Formal Systems

We will now discuss how currently existing, popular formal systems (logical frameworks, computational systems and theorem provers) deal with undefinedness, if at all.

Explicit Reasoning with Undefinedness `IMPS` is not the only formal system that allows for reasoning with undefinedness, but it certainly is one of the most prominent and most advanced ones. Other notable systems are Robin Millner et al.'s LCF (*Logic for Computable Functions* [GMW79], based on Dana Scott's *Logic of Computable functions* [Sco93]) and the `mural` System by Jones et al. [Jon+91] which is based on the three-valued LPF [BCJ84]. This latter system takes the approach of a logic with more than just two truth values, which we also will discuss in a bit more detail further below. Dana Scott also published an article together with Christoph Benzmüller on integrating free logic with the Isabelle/HOL system and applications in category theory [BS16].

[4] In this example, the *factorial* function is only defined on non-negative integers.

The specification language CASL [Cas] also allows explicit reasoning about partial functions. As an interesting similarity to `IMPS`, CASL is based on a two-valued logic with predicates on undefined terms defaulting to false.

Ad-hoc Approaches and Circumnavigations One way of avoiding having to deal with undefinedness altogether is crashing with errors as soon as an undefined value is encountered. This is simple to implement, yet does not allow reasoning close to real-life mathematics.

A slightly more elegant path is forbidding the definition of functions without also supplying a proof of their totality. This often means that common partial functions have to be wrapped in option types so they can return a `Nothing`-value when applied to values outside of their domain.

Many systems take the approach to introduce *definedness conditions*, additional obligations that need to be proved to ensure that no potential sources of undefinedness actually produce undefined expressions in the concrete cases in the library. The complexity of these conditions can grow quickly with the size of the input formula in naive implementations but this growth can be kept linear when done correctly (see [DMR08]).

John Harrison proposes an especially simple approach in his recent defence of set theory (rather than type theory) as a framework for formalising mathematics [Har18]. Any (potentially partial) function $f : A \to B$ comes with explicit domain A and co-domain B. Harrison proposes that whenever f is applied to anything outside of A, it should simply return B, the entire co-domain. This has the nice property of $x \in A \iff f(x) \in B$ (since, for set theories like ZF $B \notin B$).[5]

As answer to Harrison's "call to arms", Steffen Frerix and Peter Koepke introduced the Naproche-SAD system in [FK19], which is based on first-order logic and set theory. It is interesting to us both for its treatment of undefinedness and its approach to *notions* which basically establishes a soft type system. More on the soft typing aspects in section 2.2. Naproche-SAD builds on both the Naproche system [Cra+10] and SAD (*System for Automated Deduction*, see [LVP]). Naproche-SAD performs an *ontological check* at "runtime" (which is usually substantially more simple than the actual main proof task) to make sure any presuppositions are fulfilled. This will lead to any applications of partial functions to values outside of their domain (such as division by zero) to be rejected, if the domain of the partial function has been sufficiently restricted by the user. If it has not, the system still accepts division by zero or similar undefined expressions, which can lead to a potentially inconsistent system.

Many-valued Logics Another approach to undefinedness is allowing for more than just the two traditional truth values of truth and falsehood, from one additional value (the most common choice) up to uncountably many in some logics, sometimes especially aimed at representing probabilities. In his work [Kle52], Stephen Cole Kleene advocates adding an "undefined individual" (often simply called $\bot$ or *bottom*) to the domain of discourse and, correspondingly, a third truth value to the logic. This truth value is often called u, for *unknown*.

The usual logical connectives can be extended "generously" by giving a result whenever sufficient information is available, resulting in the truth tables seen below.

[5] Though this raises an interesting question about *total* functions in settings like this. It would seem that these would need to be proper classes instead of mere sets if they are supposed to pair the co-domain with all values inside *or outside* of the function's domain (which could be any set). Total functions being proper classes while partial functions are sets would certainly impede set-theoretic analysis of undefinedness. If we are to assume that even in set theoretic foundations, (partial) functions are always equipped with information on their domain and co-domain, and only need to pair up sets in their domain, this problem can be avoided.

$\wedge$	t	f	u
t	t	f	u
f	f	f	f
u	u	f	u

$\vee$	t	f	u
t	t	t	t
f	t	f	u
u	t	u	u

$\neg$	
t	f
f	t
u	u

Quantifiers were usually only thought to quantify over *defined* individuals, i.e. not over $\bot$.

Note that under these circumstances, some controversies can arise. For example, the law of the excluded middle $p \vee \neg p$ is no longer a tautology. Nor is $p \Rightarrow p$ (at least if the implication $p \Rightarrow q$ is still defined as $\neg p \vee q$).

The literature also contains numerous examples of logics with more than three values. The logic proposed in [Her73] has four (following a "two-dimensional" approach where each truth value comes in two flavours of "security"), the system introduced in [KK97] has five, which adds an additional truth value for undefinedness in order to distinguish between sentences that are secure (but may be true or false, e.g. "Amelia Earhart is alive."), insecure (but may be true or false, e.g. "Atlantis has sunken under the sea.") or nonsensical (e.g. "Zimbabwe lies east of the equator.").

Naturally, there are many more systems with more than the traditional two truth values, some systems even use infinitely many. Yet an exhaustive list or classification is beyond the scope of this proposal. The curious reader may refer for [Got17] for further discussion.

To be able to prototype and implement such logics easily and conveniently, one would hope to be able to merely write down the truth tables (or, in the case of a large, possibly infinite amount of truth values, functions to the same effect) for the connectives, since that is the way these logics are often defined. Any implementation effort for MMT should keep this in mind.

Sorted Logics Many systems cope with naturally occurring undefinedness by introducing sorted logics where previously partial functions become total. One standard example is reducing the domain on the second argument to the division function in real arithmetic to exclude 0, ensuring totality, which – by itself – seems straightforward enough.

$$f(x,y) = \frac{x}{y} : \mathbb{R} \to \mathbb{R} \setminus \{0\} \to \mathbb{R}$$

Yet even with slightly more complex terms, the types in these logics can quickly become disastrously complicated or even outright impossible to formulate in terms of common type systems, because the exceptions depend on variables not typically available in types. Consider the following two examples for an illustration:

$$g(x) = \tan(x) - \ln(x) : \mathbb{R}^+ \setminus \left\{ n \cdot \frac{\pi}{2}, n \in \mathbb{Z} \right\}$$

$$h(x,y) = \frac{1}{x+y} : \ldots ???$$

Also, even with sorted logics, some undefinedness might remain. Expressions like $\frac{1}{2-x}$ might not be well-sorted because the subtraction operation would be typed $- : \mathbb{Q} \to \mathbb{Q} \to \mathbb{Q}$, where you would need a $\mathbb{Q} \setminus \{0\}$ in the example above. The necessary "busywork" to weaken/strengthen certain sorts wherever required promises to be a tedious task indeed.

Another downside of this approach would be that it makes the type system undecidable. This might not be a fatal flaw in all circumstances (since sometimes we want to work in undecidable frameworks anyway), but it would be unfortunate to have no approach to undefinedness, that leaves decidable systems decidable.

Farmer's "Traditional Approach" In [Far04] Bill Farmer describes a so-called "traditional approach" of how mathematicians deal with undefinedness which calls back to the approach Russel took in [Rus05]. We will refer to this approach as "Farmer Undefinedness" in the remainder of this document as to not exacerbate the problems discussed in Section 2.1.2. Farmer Undefinedness forms the foundation of undefinedness reasoning in `IMPS` and is characterised primarily by three assumptions:

- Atomic terms (i.e. variables and constants) are always defined and always denoting.
- Compound terms may be undefined. Applications of functions to values outside of their domain are undefined as are definite descriptions that do not reference a *unique* value (i.e. there's either no value fulfilling the predicate or more than one).
- Formulas are always true or false, and hence, always defined. Should a subterm of the equation be undefined, the formula always denotes false.

Taking this particular approach to undefinedness comes with both benefits and drawbacks. For example, formulas like (1) hold without reservation or necessary restrictions, which can appear both beneficial and intuitive. However, formulas like (3), which one also might expect to hold on similar grounds, would not.

$$\forall a, b : \mathbb{R} \; . \; \neg \left(a \leq \frac{1}{b}\right) \equiv \left(a > \frac{1}{b}\right) \tag{3}$$

For the case of $b = 0$, since formulas (i.e. expressions of Boolean type) with undefined subterms evaluate to false, both inequalities would evaluate to false, breaking the equivalence and with that the universal quantification.

That (1) should work because the equation is "morally true" and yet, at the same time, (3) should not, is a serious problem for an approach that emphasises intuitiveness.

This approach also comes with the necessity to work in two different worlds: the world of formulas and the world of individuals. If formulas always denote a value and other expressions do not, it introduces the necessity to keep track of which world you are currently working in and shift mental gears accordingly. This may not be a challenge to mathematicians working on paper or blackboards who can chose the correct world whenever needed and not worry about it for the majority of the time. However, in formal systems, and hence also by people creating formalisations, it needs to be tracked basically everywhere.

2.2 Type Systems for Mathematics

Type systems, especially what we will refer to as *hard* type systems (i.e. formal systems that require assignment of a unique type to every object), are ubiquitous in mathematical systems, since they are both a popular foundation for mathematics themselves as well as a powerful tool for multiple tasks that are critical in formalisations, from ensuring proof correctness to automated knowledge management services to documentation.

There are numerous type systems in use today (see [CH88], [ML75], or [Uni13] for just a few of the more popular ones, an exhaustive discussion is out of scope for this proposal). They serve as foundations for formalisations, programming languages, proof assistants and automated theorem provers [BG01]. Many type systems are based on Alonzo Church's λ-calculus [Chu41]. Haskell Curry later discovered that the types from typed λ-calculus followed the same patterns as axioms in propositional logic [Cur34], which was later extended to predicate logic by William Alvin Howard and Nicolaas Govert de Brujin with the introduction of dependent types [SU98].

Henk Barendregt submitted a closer analysis of some of the more popular systems of typed λ-calculi (those on the way from the Simply Typed λ-calculus to the Calculus of Constructions) in [Bar92], introducing his famous λ-cube that highlights the roles of *polymorphism*, *type operators*, and *dependent types*.

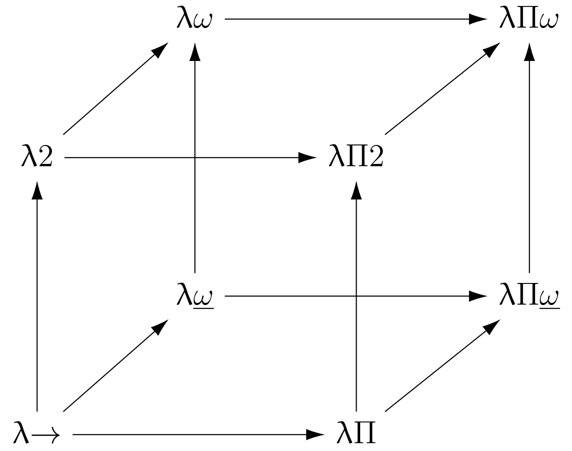

Figure 1: Barendregt's λ-cube.

2.2.1 Soft Types

Most type systems in formal systems (such as those of `Agda`, `Coq`, `LEAN`, ...) are what is commonly referred to as "hard" type systems, where every mathematical object in the domain of discourse needs to be assigned a unique type and the type needs to be known from the moment it is introduced. It is sometimes possible to prove after the fact that the object also inhabits a subtype of the original type (in which case the system can be referred to as *semi-soft*, like `IMPS` and PVS), but not an entirely different type.

The dual to these *hard* type systems are *soft* type systems, a class of type systems where not all typing information needs to be present from the start, but can also be supplied later (when accompanied by a proof that the object in question conforms to the definition of the type). Usually, these type systems also start out entirely *unityped* (i.e. with one catch-all type that all objects inhabit) and all further typing judgements are introduced (and proven) later, even if these types are very different and may require different "interpretations" of the same mathematical object.

Hard type systems dominate the current landscape of formal systems since they lend themselves the easiest to automation (since e.g. type checking is not reduced to undecidable theorem proving, necessitating complex and potentially hard to predict heuristics).

However, in hard and even in semi-soft type systems, the user is usually forced to make the somewhat artificial choice of whether to encode a given piece of information in the type system or the logical system (compare $\forall x : \mathbb{N} \,.\, P(x)$ and $\forall x \,.\, x \in \mathbb{N} \Rightarrow P(x)$). In untyped systems that mimic types through predicates, the type system becomes a feature of the logical system.

Softer type systems are furthermore interesting because they naturally mirror the way many mathematicians work in their everyday efforts. Most of the time, not all information about any given object is known from the start and it is a mathematical *discovery* (subject to proof) that a certain object also fits the definition of another class of objects.

Another advantage of soft type systems over hard type systems along these lines is that since they emphasise subtyping mechanisms they allow for a close correspondence between the natural language of mathematics (which often talks about sets and subsets) and the formalisation (of types and subtypes).

Type Classes Since introducing more typing information about objects after the fact is useful in many situations, multiple formal systems (such as Isabelle/HOL and `Coq`) that rely on hard types do still feature a way of doing so. One way of doing this is via type classes. Originally introduced to the `Haskell` programming language by Philipp Wadler and Stephen Blott [WB89], these offer many benefits (such as overloading, generic algorithms and the certainty that all types in a type class obey certain laws that can be relied on and used in proofs / program construction), as can be seen in implementation case studies ([HW07], [Haf19]).

```haskell
class Semigroup a
  where
    (<>) :: a -> a -> a
    ...
```

class *semigroup* =
  fixes $mult :: \alpha \Rightarrow \alpha \Rightarrow \alpha \quad (infixl\ \otimes\ 70)$
  assumes $assoc : (x \otimes y) \otimes z = x \otimes (y \otimes z)$

Figure 2: The semigroup type-class in `Haskell` and an equivalent in Isabelle/HOL

```haskell
instance Semigroup Int
  where
    n <> m = n + m
```

instantiation $int :: semigroup$
begin

definition
  $mult\text{-}int\text{-}def : i \otimes j = i + (j :: int)$

instance proof
  fix $i\ j\ k\ :: int$ **have** $(i + j) + k = i + (j + k)$ **by** *simp*
  then **show**$(i \otimes j) \otimes k = i \otimes (j \otimes k)$
    unfolding $mult\text{-}int\text{-}def$.
qed
end

Figure 3: The respective semigroup instantiations for integers with addition.

The general uses and effects of type classes are similar to those of soft types. The user generally introduces a new class (see Figure 2) that often comes with an operation or a function and may or may not introduce mathematical invariants (often referred to as the *laws* of that type class) that should hold in all circumstances for faithful instances of the type class.

After defining the type class, the user can then postulate that any given type is part of the type class by writing an *instance declaration* (see Figure 3). To do so, they need to give a way in which the type class operation(s) act on the new type either by referencing an existing piece of code or implementing it directly within the context of the instance declaration. The system also may or may not require the user to also supply a formal proof that the given laws do indeed hold for that type and that mapping.

After a type class has been both defined and instantiated, it then becomes possible to use the fact that a given type lies in the type class in question by using the operation of the type class with inhabitants of the type, whether in programming or in proofs.

Soft types in existing systems In the realm of formal systems for mathematics, soft type systems are not as abundant as hard ones. However, they do occur, even in prominent systems, which we will discuss now.

`Mizar`'s type system [Wie07] is of particular interest since it is maybe the most prominent example, has been in use since before the feature of type classes was introduced in other formal systems and also because there is a large library of `Mizar` formalisation in the `Mizar` mathematical library. In `Mizar`, types (e.g. `Homomorphism`) are introduced with so-called *modes* and can also be accompanied by *adjectives* (e.g. `non empty`, `X-dimensional`), which are introduced via so-called *attributes* [NU16].

The Naproche-SAD system's approach to types can also be interpreted as a soft type system [FK19]. More concretely, Naproche-SAD allows soft dependent types via *notions* and *adjectives*,

which are translated into first-order predicates and verified during the *ontological check* phase at “runtime”.

The type system of the computer algebra system GAP also has soft aspects that are interesting in this context. In general, the type system of GAP is object-oriented, not unlike that of the Java programming language, including inheritance as a form of subtyping. Inheritance alone is, however, not enough to achieve acceptable overloading resolution (also called *method selection*), because it does not provide a way of incorporating information that is proved or computed at a later point in time. There are many contexts where additional information is relevant in such a fashion, which can even influence the internal representation of an object. For example, a finite simple group G could be proved to be of odd order. Then the algorithm for computing the solvability of G (which can grow quite complex and resource-intensive in the general case) can be replaced with the much simpler constant value `true`.

To allow for this, GAP works with so-called *families* and *filters* [BL98]. Filters are Boolean qualities (such as `HasSize` or `IsAbelian`). Families are reminiscent of equivalence classes as in every object lies in a family and equal objects are in the same family. Families can also be created dynamically, while filters cannot. Together, families and filters make up GAP’s types. The system also has notions for filters that determine what operations are allowed on a given object called *categories* and sets of filters that represent how the object is actually represented (e.g. whether a matrix is stored densely or sparsely) called *representations*.

2.3 Logical Frameworks and MMT

A logical framework is a meta-formalism to specify both syntax and semantics of object-level logics or other formal systems, allowing the inheritance of certain aspects of the meta-framework, such as parsing, type reconstruction, variable substitution or a module system. A good overview of logical frameworks can be found in [Pfe01].

MMT is a modular framework for specifying formal languages (such as logics, type theories and set theories, but especially also other logical frameworks) developed by Florian Rabe and others at the KWARC research group [Rab18a]. Its prime selling point that differentiates it from (other) logical frameworks is, that it has a strong commitment to foundation-independence. Its kernel algorithms all are abstracted over a given set of rules that is collected from context, allowing multiple sets of rules to coexist within the same development pipeline.

In the development of MMT, one concerted effort has always been making the rapid development of logics for formalisations easy (taking implementation times down to person-days from person-years) and beneficial to the user, which is discussed in more detail in [MR19].

Creating imports from any source library into MMT/OMDoc is also encouraged as this opens up the existing MMT infrastructure of type checking, visualisation and other knowledge management services. This approach has been discussed at length in [Mül19].

In the KWARC research group, the predominant logical framework used for implementing logics in MMT is **LF** [HHP93] (which will be one focus point for improvements related to undefinedness, see below), but the underlying machinery does not force the user to use a specific logical framework.

3 Research Objectives

3.1 Undefinedness

The first prominent goal for my PhD research would be to allow for native (un)definedness reasoning within MMT, ideally both on the manual and the automated reasoning level.

Currently, neither MMT nor **LF** have a way of dealing with undefinedness, so a key research question here is to find out at which level this feature should be introduced. Can we give MMT a primitive, yet foundation-independent notion of undefinedness that all logics and systems that are implemented in MMT then could benefit from on an opt-in basis? If so, any system that was implemented without support for undefinedness should be able to simply "import" a suited notion of undefinedness instead of having to re-implement all the machinery again, then and there.

From this, one or multiple extensions to **LF** that also allow for undefinedness (working title: **LFU***, as a shorthand for $\mathbf{LFU}_1, \mathbf{LFU}_2, \ldots$) would be easy to create from the system-wide undefinedness apparatus. Alternatively, these extensions could take its place for approaches that cannot be implemented entirely foundation-independently on the MMT level. Similar extensions to **LF** have been implemented in the past (compare the extensions in [LFX] that allow for record types, subtyping, ...) and proved to be very helpful.

As we have seen in the previous section, there is more than one perspective on reasoning with and about (un)definedness in a formal system. So, to do justice to these different approaches, it would be imperative for these extensions to MMT and/or **LF** to not just cover one of them, but instead capture at least all of the most prominent approaches. It is not the goal to pick a new favourite, but to empower the user to chose and experiment after the context and circumstances of their own systems.

A closer study of the feasibility and/or usefulness of allowing *mixtures* of different approaches is also warranted.

3.2 Soft Type Systems

The other topic I hope to tackle that of a soft type system for mathematics. Unlike in traditional (hard) type systems, the type of mathematical objects does not need to be known or set from the start. At the core, the type system only has one type and all objects in the domain of discourse inhabit that type. Using definitions and proofs, they can later be shown to fulfil certain properties and hence inhabit certain types. For example, the set of naturals can be treated as both a subset of the real or complex numbers and as a measurable cardinal.

Like undefinedness, soft type systems have the potential to capture the realities of mathematics as performed by mathematicians. An object is shown to be an instance of one structure (e.g. a monoid) but can later be proven to also meet the criteria (axioms) of another structure (e.g. a group). Both of these angles (and any that follow) can be used in proofs and for further definitions. This is close to the way mathematicians often shift focus in their work, as a switch of perspective can sometimes be the deciding idea behind a proof. A soft type system also better connects to the notion of mathematics as a practise of discovery. There is probably no "perfect" type system anyone could pre-design that captures all relevant objects and inclusions and disjunctions between them. Starting from just one type and steadily moving outward seems like a promising idea in comparison.

This sort of flexibility has been one of the reasons behind the success of the remarkable `Mizar` library. Yet, in that system, the user is committed to one particular meta-logic (although the choice of axiom system is technically free, even if heavily influenced by the fact that the library uses Tarski-Grothendiek set theory axioms). It would be beneficial if they could switch between (meta-)logics easily without having to give up the benefits of a soft type system. As with undefinedness, I want to make it possible for soft types to be one of many building blocks a user of MMT can (easily and without much hassle) chose from to build exactly the right logic for any given effort.

As with undefinedness, it is also of crucial importance to find out, at what level in the feature would be best introduced and which advantages / drawbacks come with each possibility.

3.3 Reasoning

One major drawback to the two previous objectives is that I anticipate is that this might generate a plethora of "trivial" proof obligations (simple proof goals about objects having a certain type, terms being (un)defined, types being inhabited, objects being either equal or unequal, . . .).

This is why I also intend to increase the support for automated reasoning in MMT. Ideally, most or all of these additional "trivial" proof obligations should automatically proved by some component of the system as to not generate an additional burden for the user. Two choices to achieve this effect that come mind would be to let an external first-order ATP brute-force the proofs or implement all necessary infrastructure and algorithms in MMT, basically from scratch. These however, promise either weak performance or unreasonable implementation work. However, there is another alternative. Since these smaller proof obligations pose a problem that is natural to tackle with rule-based search algorithms, and these rules can likely be generated from MMT theories while preserving modularity, λProlog becomes an interesting candidate for external support that doesn't rely solely on brute-forcing. For more on this approach, see Section 4.3.

4 Methodology

4.1 Extending MMT with **LFU***

As discussed in Section 3.1, I hope to extend the MMT ecosystem by rigorous ways of dealing with undefinedness via a series of extensions to **LF**. However, we have seen in Section 2.1 that there are many different approaches to undefinedness in the literature and in practice, so it is not at all obvious which ones of these the extensions should take.

There are two prominent approaches to this goal that appear feasible for the proposed timespan. We will discuss both of them in the following sections.

4.1.1 Farmer's Undefinedness

Since MMT and `IMPS` share so many philosophies and design choices, one obvious candidate for introducing *any* kind of undefinedness to the MMT ecosystem would be undefinedness à la `IMPS`, allowing for non-denoting terms but not non-denoting formulas, having them default to `false` instead.

There are some details, in which the implementation might deviate from the on of the `IMPS` system. For example, this version of **LFU*** might allow for empty sorts and undefined constants or treat `falselike` (a quasi-constructor that deals with functions into the Booleans, see [FGT98]) differently or allow undefined constants. But the general idea will stay the same. Some of these changes were suggested by Farmer himself as convenience and quality-of-life improvements (deferred definedness proofs for constants could be beneficial, for example), some are expected to arise out of necessity.

4.1.2 Other Undefinedness Approaches

Once it is possible to import Farmer's undefinedness into a logic via import (or similar), it would then be good to create a range of others. These implementation efforts are expected to benefit from the learning experience of implementing the Farmer's undefinedness first.

Special attention should be paid to approaches like many-valued logics (both with finite and infinite amounts of truth values), that take a prominent role in the literature. This includes possible additions to the MMT surface syntax (or at least the notation mechanisms), since these

logics are usually most easily defined by giving the truth tables for the common connectives, something that would be cumbersome to do with only the current syntax.

Additionally, I would also like to spend a certain amount of time to research the possibility of *combining* two (or more) different approaches to undefinedness. Compositionality has proofed itself as a critical aspect of many a successful approach to difficult problems and I think that some time spent investigating the possibilities of making undefinedness features compositional would be time well spent.

4.1.3 Test: Reimplementing Logics with Undefinedness

As a measure of how well the extension from **LF** to **LFU*** works in practice, the possibility of re-implementing logics from the already present archive of formalisations that deal with undefinedness springs to mind, since rapid implementation of formal systems is one of MMT's explicit use cases. One hopes, that after making undefinedness features available to the logical framework, that reimplementing a logic that needs similar features should not require extraordinary amounts of work and yield compatible results.

There is at least one such logic: LUTINS, the underlying logic of the `IMPS` theorem prover, see [BK18]. Other logics for undefinedness (such as LCF or $\mathcal{SKL}$, depending on their compatibility to whatever the undefinedness available to MMT users ends up being) could also serve as valuable case studies on how adaptable and flexible the new undefinedness infrastructure is.

4.2 Soft Type Systems for MMT

Making soft type systems available as a "building block" of rapid prototyping of logics and formal systems means both making the necessary changes to the MMT type checker as well as designing and implementing a (ideally) foundation independent way of then proving a certain mathematical object in some logic that is abstracted over has a given soft type.

Both of these aspects still need considerable design work, as it is less clear to me as of now how exactly this can be done, and what pitfalls await on the way there. As a start, I intend to take a closer look at the implementation of soft types in the role model systems `Mizar` and Naproche-SAD.

It has previously been suggested by Florian Rabe [Rab18b] that a structural feature for type classes would be beneficial for MMT. We have already established the link between type classes and soft typing, so it would be natural and prudent to re-examine this issue and see if it would be feasible to implement, if it is made superfluous by another effort or indeed if it makes other efforts superfluous.

4.2.1 Subtyping and Soft Type Systems

The relationship between subtyping and soft type systems may appear obvious (since soft types can be implemented very naturally as predicate subtypes of an unityped foundation [NU16; Wie07]), yet it bears investigation if the exiting infrastructure in MMT for subtyping (see e.g. [MRK18]) is sufficient to supply enough power to construct a soft type system that can clear the bar set in Section 4.2 above. If this is possible, it could save a large amount of time and implementation energy to reuse the existing mechanisms, even if they need to be extended or reworked.

Also, I would like to spend some time to investigate the relationship between (soft type) subtyping and undefinedness, since having a rigorous treatment of undefinedness at hand can also offer new ways of defining subtypes "from the outside" rather than "from the inside".

4.2.2 Test: Improving Interaction with Soft-Typed Systems

To test the apparatus for soft types in MMT, it would be interesting to incorporate it in those places, where previous contributors had to work around the hardness of the existing MMT type system. An obvious choice for where to start would be the import/export mechanisms for systems that feature soft types themselves and that are already integrated with MMT to some extent, the most prominent examples being GAP and Mizar.

It seems plausible that the quality of the interaction or even the amount of coverage could be improved by a more faithful representation on the MMT side that also allows for soft types.

4.3 Automatic Dismissal of Proof Obligations

"This proof is trivial and left as an exercise to the compiler."
– Michael Burge in [Bur19]

Both prominent features discussed so far are likely to introduce a significant number of proof obligations that are not necessarily hard to prove, but do need to be proved nevertheless. Requiring the user to prove them herself would place her under an undue burden of "busywork", likely leading to justified frustration. So, ideally, these obligations would just be proved by the system.

All of the examples mentioned above are of course instances of larger problems that become intractable (or even unsolvable!) in larger contexts. Yet in many circumstances (prominently in undefinedness reasoning and soft typing, or so it stands to reason), the actual incarnations are solvable in a reasonable timespan often enough.

MMT already has a simple (yet still foundation-independent) iterative theorem proving system, yet it does not meet standards of usefulness or efficiency. I therefore hope to extend the MMT reasoning system by an automated component that significantly reduces the amount of busywork and proving of uninteresting lemmata that a user would have to do.

The MMT ecosystem currently allows for the user to annotate certain constants with a *role* that makes the constant available as an additional rule for the simplifier. Ultimately, it would also be prudent to extend this mechanism to definedness assumptions.

4.3.1 LambdaProlog & the ELPI System

In discussions between Michael Kohlhase, Florian Rabe, Claudio Sacerdoti Coen and myself, it was suggested to use λProlog (or, to be more precise, ELPI) as the programming language for the theorem prover that is to be implemented (that is, program it in ELPI and external to, though connected with MMT). This approach would make use of the fact that higher-order unification is already reliably implemented and could be used as a mechanism for proof search.

λProlog [Mil] is a programming language based on an intuitionistic fragment of Church's Simple Theory of Types that extends the logic-programming language Prolog by features such as abstract data types and higher-order programming, adding Harrop Formulas to Prolog's underlying foundation of Horn clauses. This allows to make better use of the logical structure and to use the first-order quantifiers. A good introduction to λProlog can be found in [NM88] or the more recent [MN12].

There have been multiple implementations of λProlog since its inception, such as the Teyjus system [Nad10]. The ELPI system [SCT15], written by Claudio Sacerdoti Coen and Enrico Tassi and introduced in [Dun+15], is another such implementation of λProlog with some additional features and a focus on performance, moving some previously intractable proof searches into reach. It is implemented in the OCaml programming language.

The proposed pipeline would look roughly like the following: For any given theory, one would generate an ELPI *kernel*, a file that introduces the declarations of a theory (in the case of propositional logic, these would include constants for truth and falsehood, the connectives, as well as types for propositional formulae and proof certificates) and ELPI rules that reflect the inference rules of the theory (e.g. $\wedge_I$ or $\neg_E$).

One would also need *experts*, i.e. ELPI modules that apply strategies and/or apply help predicates to reduce the enormous search space of proof terms that would otherwise be intractable. One way an expert module could help with this effort is specifying which of the relevant inference rules are "invertible", i.e. usable in both directions without incurring huge search space penalties, and which ones are only to be used under specific circumstances (and what those circumstances are). To further adhere to the principle of "no wild guesses" in the process of trying to prove an obligation, "forward" and "backward" use of inference rules could be alternated in their application, but obviously the precise implementation details are subject of future research.

Proof obligations that arise in MMT would be automatically translated and handed to ELPI (running in server mode). The fact that MMT would be calling an external system for proving introduces an issue of trust. How can it rely on the proof found in ELPI actually being correct? For this, it would be helpful to introduce *proof certificates* (on which Dale Miller also has done extensive work during the *ProofCert* project, see [Mil15]), that could be generated by ELPI and communicated back to MMT to be checked for correctness.

Claudio Sacerdoti Coen provided some preliminary examples for both kernels and experts for propositional and first-order logics[6]. These hand-crafted examples could be used as a reference for *generating* similar proof rules and help statements. It remains to be seen and subject to experimentation, just exactly how much of all necessary code could be generated and how much would need to be written by hand. It was our guess that the kernels should be generate-able but that the experts would largely still need a human programmer.

4.3.2 Test: Tableaux Prover

One possible collaboration would be with Jan-Frederic Schaefer, also working at the KWARC group, to create a Tableaux prover modelled after the ideas discussed in Section 4.3.1. It would serve as a basic reasoning component in his work on logic and the semantics of natural languages and the GLF project in particular [KS].

This tableaux prover would employ many the same strategies as the full prover and/or could work as a first prototype. Ideally, it would also already be as logic-independent as possible (see [AG09]).

4.3.3 Test: Softly Typed Set Theory

One possible case study to test whether or not the automatic dismissal of proof obligations works to a satisfying degree would be to implement a softly typed set theory after the extensions discussed above are already in place. This set theory would be fundamentally untyped (or rather, uni-typed), but still rely on types in its syntax. These types would be formalised as unary predicates on sets, generating a plethora of proof obligations for the system to solve.

Note that this is not the same as the approach discussed in Section 4.2. This would be a single, object-level logic which is expected to generate lots of simple proof obligations, while Section 4.2 discusses adding features to the underlying logical framework to make soft typing easily available in *arbitrary* logics via import.

[6]The code can be found on the MathHub GitLab at `https://gl.mathhub.info/elpi/elpi-playground`.

4.3.4 Test: IMPS Proof Obligations

Previous work on importing the IMPS theory library into MMT and translating it to OMDoc has treated all proofs of theorems merely as opaque data [BK18]. Another way of putting the improved automated reasoning mechanisms to the test would be an effort to include the proofs in the translation, to the maximal semantic extent possible. Since the specialised IMPS reasoning machine are not present in MMT, it is to be expected that the automated reasoning system would be needed to fill in context or take the role of the IMPS simplifier.

Complete coverage will likely remain out of reach, but it may well be possible to recover enough structure from the proof terms to learn interesting things about the proofs or compare them to proofs from other formal libraries.

4.4 Work-plan and Timeline

In summation, I identify the following work packages with their estimated associated duration, categorised into four main work areas:

- **Work Area 1: Undefinedness**

  A. **IMPS-Style Undefinedness**
  Research and develop this one specific type of undefinedness for MMT (See Section 4.1.1).
  This includes time to familiarise myself with the involved MMT internals.
  Estimated completion time: 9 person-months

  B. **Implement other Undefinedness Approaches**
  Research and implement support for other approaches and combinations (See Section 4.1.2).
  Estimated completion time: 3 person-months

- **Work Area 2: Soft Type System**

  A. **Subtyping and Type Systems**
  Investigate relationship of soft type sytems, subtyping and automation (See Section 4.2.1)
  Estimated completion time: 6 person-months

  B. **Implementing a soft type system extension for MMT**
  Foundation-independent soft type systems, as discussed in Section 4.2
  Estimated completion time: 9 person-months

- **Work Area 3: (Automated) Reasoning**

  A. **Foundation-independent definedness reasoning**
  Develop solver support for foundation-independent undefinedness reasoning (See Section 4.1).
  Estimated completion time: 6 person-months

  B. **Automatic Dismissal of Proof Obligations**
  Develop stronger automatic reasoning support for MMT, as discussed in Section 4.3
  Estimated completion time: 9 person-months

- **Work Area 4: Tests and Case Studies**

A. **Softly-Typed Set Theory**
Develop a softly typed set theory in MMT as discussed in Section 4.3.3
Estimated completion time: 3 person-months

B. **(Re-)Implementing Undefinedness Logics**
Finding different undefinedness logics and (re-)implement them as discussed in Section 4.1.3
Estimated completion time: 1.5 person-months

Based on all of the above, my current best estimate for a work timeline is the following:

Year 1	Year 2	Year 3	Year 4
WP1-A			WP1-B
	WP2-A	WP2-B	
WP3-A	WP3-B.1		WP3-B.2
	WP4-A		WP4-B

Figure 4: Work Plan Gantt Chart

References

[AG09] Pietro Abate and Rajeev Goré. "The Tableau Workbench". In: *Electronic Notes in Theoretical Computer Science* 231 (Mar. 2009), pp. 55–67. DOI: 10.1016/j.entcs.2009.02.029.

[Bar92] Henk P. Barendregt. "Lambda Calculi with Types". In: *Handbook of Logic in Computer Science*. Ed. by S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum. Vol. 2. Oxford University Press, 1992, pp. 117–309.

[BCJ84] H. Barringer, H. H. Cheng, and C. B. Jones. "A Logic Covering Undefinedness in Program Proofs". In: *Acta Informatica* 21 (1984), pp. 251–269.

[BG01] Henk Barendregt and Herman Geuvers. "Proof-Assistants Using Dependent Type Systems". In: *Handbook of Automated Reasoning*. Ed. by Alan Robinson and Andrei Voronkov. Vol. I and II. Elsevier Science and MIT Press, 2001, pp. 1149–1238.

[BK18] Jonas Betzendahl and Michael Kohlhase. "Translating the IMPS theory library to OMDoc/MMT". In: *Intelligent Computer Mathematics (CICM) 2018*. Conferences on Intelligent Computer Mathematics. Ed. by Florian Rabe et al. LNAI 11006. Springer, 2018. ISBN: 978-3-319-96811-7. DOI: 10.1007/978-3-319-96812-4. URL: http://kwarc.info/kohlhase/papers/cicm18-imps.pdf.

[BL98] Thomas Breuer and Steve Linton. "The GAP 4 Type System: Organising Algebraic Algorithms". In: *Proceedings of the 1998 International Symposium on Symbolic and Algebraic Computation*. ISSAC '98. Rostock, Germany, 1998, pp. 38–45. ISBN: 1-58113-002-3. DOI: 10.1145/281508.281540.

[Bli88] Andrzej Blikle. "Three-valued predicates for software specification and validation". In: *VDM '88 VDM — The Way Ahead*. Springer Berlin Heidelberg, 1988, pp. 243–266. ISBN: 978-3-540-45955-2.

[BM04] M. Bidoit and Peter D. Mosses. *CASL — the Common Algebraic Specification Language: User Manual*. LNCS 2900. Springer Verlag, 2004.

[BS16] Christoph Benzmüller and Dana Scott. "Automating Free Logic in Isabelle/HOL". In: *Mathematical Software – ICMS 2016*. Ed. by Gert-Martin Greuel et al. 2016. ISBN: 978-3-319-42432-3. URL: https://link.springer.com/chapter/10.1007/978-3-319-42432-3_6.

[Bur19] Michael Burge. *"This proof is trivial and left as an exercise to the compiler"*. https://twitter.com/TheMichaelBurge/status/1108867483287457793. Tweet. 2019.

[Cas] *CASL*. CoFI. Feb. 12, 2008. URL: http://www.informatik.uni-bremen.de/cofi/index.php/CASL (visited on 08/19/2019).

[CH88] Thierry Coquand and Gérard Huet. "The Calculus of Constructions". In: *Information and Computation* 76.2/3 (1988), pp. 95–120.

[CHI] Wikipedia. *Curry–Howard correspondence — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/w/index.php?title=Curry-Howard_correspondence (visited on 05/16/2017).

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion. (AM-6)*. Princeton University Press, 1941. ISBN: 9780691083940. URL: http://www.jstor.org/stable/j.ctt1b9x12d.

[Cra+10] Marcos Cramer et al. "The Naproche Project Controlled Natural Language Proof Checking of Mathematical Texts". In: *Controlled Natural Language, Workshop on Controlled Natural Language, CNL 2009. Revised Papers*. Ed. by Norbert E. Fuchs. LNCS 5972. Springer, 2010, pp. 170–186. ISBN: 978-3-642-14417-2. DOI: 10.1007/978-3-642-14418-9_11.

[Cur34] H. B. Curry. "Functionality in Combinatory Logic". In: *Proceedings of the National Academy of Sciences of the United States of America* 20.11 (1934), pp. 584–590. ISSN: 00278424. URL: http://www.jstor.org/stable/86796.

[DMR08] Ádám Darvas, Farhad Mehta, and Arsenii Rudich. "Efficient Well-Definedness Checking". In: *Automated Reasoning*. Ed. by Alessandro Armando, Peter Baumgartner, and Gilles Dowek. Springer Berlin Heidelberg, 2008, pp. 100–115. ISBN: 978-3-540-71070-7. URL: https://link.springer.com/content/pdf/10.1007%2F978-3-540-71070-7_8.pdf.

[Don+14] Alexandre Donzé et al. "Machine Improvisation with Formal Specifications". In: *Proceedings of the 40th International Computer Music Conference (ICMC)*. Sept. 2014, pp. 1277–1284. URL: http://hdl.handle.net/2027/spo.bbp2372.2014.196.

[Dun+15] Cvetan Dunchev et al. "ELPI: fast, Embeddable, λProlog Interpreter". In: *Proceedings of LPAR*. Suva, Fiji, 2015. URL: https://hal.inria.fr/hal-01176856.

[Far04] William M. Farmer. "Formalizing Undefinedness Arising in Calculus". In: *Automated Reasoning*. Vol. 3097. Lecture Notes in Computer Science. Springer, 2004, pp. 475–489. URL: http://imps.mcmaster.ca/doc/calculus.pdf.

[Far90] William M. Farmer. "A Partial Functions Version of Church's Simple Theory of Types". In: *Journal of Symbolic Logic* 55 (1990), pp. 1269–1291. URL: http://www.jstor.org/stable/2274487.

[Fef95] Solomon Feferman. "Definedness". In: *Erkenntnis* 43.3 (1995), pp. 295–320. URL: https://math.stanford.edu/~feferman/papers/definedness.pdf.

[FGT98] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. *The IMPS 2.0 User's Manual*. 1st ed. The MITRE Corporation. Bedford, MA 01730 USA, Jan. 1998.

[FK19] Steffen Frerix and Peter Koepke. *Making Set Theory Great Again*. Talk at the Artificial Intelligence and Theorem Proving Conference. 2019. URL: http://aitp-conference.org/2019/slides/PK.pdf.

[FLH18] Tomás Frimm, Djones Lettnin, and Michael Hübner. "A Survey on Formal Verification Techniques for Safety-Critical Systems-on-Chip". In: *Electronics* 7 (May 2018). DOI: 10.3390/electronics7060081.

[Fre92] G. Frege. "Über Sinn und Bedeutung". In: *Funktion, Begriff, Bedeutung. Fünf Logische Studien*. Ed. by G. Patzig. Göttingen: Vanden Hoeck, 1892.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*. LNCS 78. Springer Verlag, 1979.

[Gon+13] Georges Gonthier et al. "A Machine-Checked Proof of the Odd Order Theorem". In: *Interactive Theorem Proving*. Ed. by Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie. Vol. 7998. LNCS. Springer, 2013, pp. 163–179. ISBN: 978-3-642-39633-5. DOI: 10.1007/978-3-642-39634-2_14.

[Got17] Siegfried Gottwald. "Many-Valued Logic". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Winter 2017. Metaphysics Research Lab, Stanford University, 2017. URL: https://plato.stanford.edu/archives/win2017/entries/logic-manyvalued/.

[Haf19] Florian Haftmann. *Haskell-style type classes with Isabelle/Isar*. June 2019. URL: https://isabelle.in.tum.de/doc/classes.pdf.

[Hal+17] Thomas Hales et al. "A formal proof of the Kepler conjecture". In: *Forum of Mathematics, Pi* 5 (2017). DOI: 10.1017/fmp.2017.1.

[Har18] John Harrison. *Let's Make Set Theory Great Again*. Talk at the Artificial Intelligence and Theorem Proving Conference. 2018. URL: http://aitp-conference.org/2018/slides/JH.pdf.

[Her73] H. Herzberger. "Dimensions of truth." In: *Journal of Philosophical Logic* 2 (1973), pp. 335–356.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. "A framework for defining logics". In: *Journal of the Association for Computing Machinery* 40.1 (1993), pp. 143–184.

[HW07] Florian Haftmann and Makarius Wenzel. "Constructive Type Classes in Isabelle". In: *Types for Proofs and Programs, TYPES 2006*. Ed. by T. Altenkirch and C. McBride. Vol. 4502. LNCS. Springer, 2007.

[Jon+91] C. B. Jones et al. *mural: A Formal Development Support System*. Springer-Verlag, 1991. URL: http://homepages.cs.ncl.ac.uk/cliff.jones/publications/Books/mural.pdf.

[Jon95] C.B. Jones. "Partial functions and logics: A warning". In: *Information Processing Letters* 54.2 (1995), pp. 65–67. DOI: 10.1016/0020-0190(95)00042-B. URL: http://dx.doi.org/10.1016/0020-0190(95)00042-B.

[KK94] Manfred Kerber and Michael Kohlhase. "A Mechanization of Strong Kleene Logic for Partial Functions". In: *Proceedings of the 12th Conference on Automated Deduction*. Ed. by Alan Bundy. LNAI 814. Nancy, France: Springer Verlag, 1994, pp. 371–385. URL: http://kwarc.info/kohlhase/papers/KeKo94-CADE.pdf.

[KK97] Manfred Kerber and Michael Kohlhase. *Reasoning without Believing: On the Mechanization of Presuppositions and Partiality*. Tech. rep. CSRP-97-23. University of Birmingham, School of Computer Science, Sept. 1997. URL: ftp://ftp.cs.bham.ac.uk/pub/tech-reports/1997/CSRP-97-23.ps.gz.

[Kle+09] Gerwin Klein et al. "seL4: formal verification of an OS kernel". In: *SOSP*. 2009.

[Kle52] Stephen C. Kleene. *Introduction to Meta-Mathematics*. North Holland, 1952.

[KS] Michael Kohlhase and Jan Frederik Schaefer. "GF + MMT = GLF – From Language to Semantics through LF". In: *LFMTP 2019, Proceedings*. Electronic Proceedings in Theoretical Computer Science (EPTCS). URL: https://kwarc.info/kohlhase/submit/lfmtp-19.pdf.

[LFX] *MathHub MMT/LFX Git Repository*. URL: http://gl.mathhub.info/MMT/LFX (visited on 05/15/2015).

[LVP] Alexander Lyaletski, Konstantin Verchinine, and Andrei Paskevich. *The System for Automated Deduction (SAD)*. URL: http://nevidal.org/sad.en.html (visited on 09/16/2019).

[LW94] Barbara H. Liskov and Jeanette M. Wing. "A Behavioral Notion of Subtyping". In: *ACM Transactions on Programming Languages and Systems* 16 (1994), pp. 1811–1841.

[Mar06] Alberto A. Martínez. "History: Meaningful and Meaningless Expressions". In: *Negative Math: How Mathematical Rules Can Be Positively Bent*. Princeton University Press, 2006, pp. 43–79. URL: http://www.jstor.org/stable/j.ctv301hfp.7.

[Mil] Dale Miller. *λProlog*. URL: http://www.lix.polytechnique.fr/Labo/Dale.Miller/lProlog/.

[Mil15] Dale Miller. "Foundational Proof Certificates". In: *All about Proofs, Proofs for All*. Ed. by David Delahaye and Bruno Woltzenlogel Paleo. Vol. Mathematical Logic and Foundations. All about Proofs, Proofs for All 55. College Publications, Jan. 2015, pp. 150–163. URL: https://hal.inria.fr/hal-01239733.

[ML75] Per Martin-Löf. "An Intuitionistic Theory of Types: Predicative Part". In: *Proceedings of the '73 Logic Colloquium*. Ed. by H. E. Rose and J. C. Shepherdson. North-Holland, 1975, pp. 73–118.

[MN12] Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Camebridge University Press, Aug. 2012. DOI: https://doi.org/10.1017/CBO9781139021326.

[MR19] Dennis Müller and Florian Rabe. "Rapid Prototyping Formal Systems in MMT: 5 Case Studies". In: *LFMTP 2019*. Electronic Proceedings in Theoretical Computer Science (EPTCS), 2019. URL: https://kwarc.info/people/frabe/Research/MR_prototyping_19.pdf.

[MRK18] Dennis Müller, Florian Rabe, and Michael Kohlhase. "Theories as Types". In: ed. by Didier Galmiche, Stephan Schulz, and Roberto Sebastiani. Springer Verlag, 2018. URL: http://kwarc.info/kohlhase/papers/ijcar18-records.pdf.

[Mül19] Dennis Müller. "Mathematical Knowledge Management Across Formal Libraries". Doctoral Thesis. Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), 2019, p. 215. URL: https://opus4.kobv.de/opus4-fau/files/12359/thesis.pdf.

[Nad10] Gopalan Nadathur. *Teyjus: A λProlog Implementation*. 2010. URL: https://www.cs.nmsu.edu/ALP/wp-content/uploads/2010/02/article1.pdf.

[NM88] Gopalan Nadathur and Dale Miller. "An overview of λProlog". In: *Fifth International Logic Programming Conference*. MIT Press, Seattle, Washington, 1988, pp. 810–827.

[NU16] Adam Naumowicz and Josef Urban. "A Guide to the Mizar Soft Type System". In: *Types for Proofs and Programs, TYPES 2016*. 2016.

[Pfe01] Frank Pfenning. "Logical Frameworks". In: *Handbook of Automated Reasoning*. Ed. by Alan Robinson and Andrei Voronkov. Vol. I and II. Elsevier Science and MIT Press, 2001.

[PL08] Francis Jeffry Pelletier and Bernard Linsky. "One Hundred Years After On Denoting: Russell vs. Meinong". In: N. Griffin & D. Jacquette, 2008. Chap. Russell's Criticisms of Frege's Theory of Descriptions. URL: http://www.sfu.ca/~jeffpell/papers/Pelletier_Linsky_2nd.pdf.

[Rab18a] Florian Rabe. "MMT: A Foundation-Independent Logical Framework". Online Documentation. 2018. URL: https://kwarc.info/people/frabe/Research/rabe_mmtsys_18.pdf.

[Rab18b] Florian Rabe. *Structural feature for type classes*. July 2018. URL: https://github.com/UniFormal/MMT/issues/367.

[Ruf17] Marco Ruffino. "Definite Descriptions, Presuppositions and Reference". In: *Cadernos de Estudos Lingüísticos* 59 (Apr. 2017), p. 37. DOI: 10.20396/cel.v59i1.8648396.

[Rus05] Bertrand Russell. "On Denoting". In: *Mind (New Series)* 14 (1905), pp. 479–493.

[RV01] Alan Robinson and Andrei Voronkov, eds. *Handbook of Automated Reasoning*. Vol. I and II. Elsevier Science and MIT Press, 2001.

[Sco93] Dana S. Scott. "A type-theoretical alternative to ISWIM, CUCH, OWHY". In: *Theoretical Computer Science* 121 (1993), pp. 411–440. URL: http://www.cs.cmu.edu/~kw/scans/scott93tcs.pdf.

[SCT15] Claudio Sacerdoti Coen and Enrico Tassi. *The ELPI system*. 2015. URL: https://github.com/LPCIC/elpi.

[SCZ07] Claudio Sacerdoti Coen and Enrico Zoli. "A Note on Formalising Undefined Terms in Real Analysis". In: *PATE'07* (2007), p. 3. URL: https://www.cs.unibo.it/~sacerdot/PAPERS/pate07.pdf.

[Str73] P. F. Strawson. "On referring". In: *Presuppositions in Philosophy and Linguistics*. Ed. by János S. Petöfi and Dorothea Franck. Athenäum Verlag, 1973, pp. 193–220.

[SU98] Morten Heine B. Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. 1998. DOI: 10.1.1.17.7385. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.17.7385.

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: http://homotopytypetheory.org/book, 2013.

[Way18] Hillel Wayne. *1/0 = 0*. Aug. 2018. URL: https://www.hillelwayne.com/post/divide-by-zero/.

[WB89] Philip Wadler and Stephen Blott. "How to Make Ad-hoc Polymorphism Less Ad Hoc". In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '89. Austin, Texas, USA, 1989, pp. 60–76. ISBN: 0-89791-294-2. DOI: 10.1145/75277.75283. URL: http://doi.acm.org/10.1145/75277.75283.

[Wie07] Freek Wiedijk. "Mizar's Soft Type System". In: *Theorem Proving in Higher Order Logics*. Springer Berlin Heidelberg, 2007, pp. 383–399. ISBN: 978-3-540-74591-4.