

Integrating a Logical Framework and a Knowledge Management Framework

Maria Alecu,
Supervisors: Florian Rabe, Michael Kohlhase
Jacobs University, Bremen, Germany

May 12, 2012

Abstract

The study of mathematical knowledge and theoretical computer science has been concerned with the development of two different types of frameworks, one related to representation techniques for mathematical concepts and mechanizations of theorems, and the other related to communication of the knowledge on a large scale and in an interactive way. For this purpose, logical frameworks such as LF and Knowledge Management systems such as OMDoc have been developed. The purpose of this thesis is to combine the benefits offered by both kinds of frameworks into a single framework by integrating LF into the MKM system offered by the MMT platform. In this way, we will have available in a single framework logic tools such as unification and type-checking but also the advantage of scaling to large knowledge bases.

1 Introduction

Mathematics has been a central activity of the intellectual efforts to understand better the world we live in and continues to play a crucial role also nowadays. Mathematical knowledge continues to grow - even exponentially as it has been estimated - with the total number of papers being published doubling every ten - fifteen years [A.M95]. Given this situation, it is crucial to organize this knowledge in order to be accessible in an efficient way to as much as possible of the mathematical community. In the meantime, starting with Hilbert's program in 1920 [Hil26] and with the "Principia Mathematica" [WR13], there has been a constant preoccupation for the formalization of mathematics. This enables the construction of tools that provide assistance to the mathematician in his efforts to solve problems and verify the correctness of his proofs. We are thus confronted with two different, even independent research areas, which are united by a common goal, that is to contribute to the progress of mathematics.

The process of assisting the mathematician in his attempt to solve problems or verify proofs starts with one of the oldest activities of human knowledge

- logic. Formal logics provide a precise language and a strict set of rules in which properties and information can be formulated and then exploited via the set of inference rules. Different logics are suited for different purposes, depending on the things one needs to represent and explore, thus the next effort was in providing a uniform way in which one can talk about all logics. This is where logical frameworks come into play as a general meta-language for specifying and implementing deductive systems given by axioms and rules of inference. There are two essential properties of logical frameworks: first of all, they are independent of the deductive system one might want to represent and secondly, the system implementing logical frameworks are focused on in-memory and batch-processing. The pipeline of processing for such systems usually follows the following pattern: input file written in concrete syntax - lexing - parsing - in-memory abstract representation. The abstract syntax enables the use of logic tools such as automated theorem proving, type-checking, type-reconstruction etc. Logical frameworks come in two main flavors: hereditary Harrop formulas, which are implemented by systems such as Isabelle [Pau89] and dependently typed λ -calculi such as LF [HHP93], with the most notable implementation in the Twelf system [PS99]. These two implementations serve different purposes: Isabelle focuses more on the tool support such as semi-automated reasoning, while Twelf focuses more on the breadth of the logics that can be encoded with it.

The other part of the mathematical activity with which we are concerned is providing a uniform way to organize all the new mathematical knowledge as well as providing tools for querying and updating the base knowledge. This is the focus of the field of Mathematical Knowledge Management. MKM systems use large knowledge bases, usually written in a content markup language which makes them easy to read for a machine without knowledge of the type system; thus, we can say that MKM systems are independent of the underlying deductive system, in the same way as logical frameworks are. However, MKM systems are very different in the sense that they allow large-scale [KRZ10] and interactive processing [GLR09]. OpenMath [BCC⁺04] and OMDoc [Koh00] are two such markup, content-oriented languages used as standards for representing mathematical knowledge, developed by the research in the MKM area. The fact that both MKM systems and logical frameworks are independent of the underlying deductive system stands as a motivation and foundation of the MMT language [RK11] which basically provides a new abstraction layer in which logical frameworks such as LF function as plugins and MKM standards such as OMDoc offer the scaling to a wide class of documents. MMT is a generic, formal module system for mathematical knowledge that is designed to be applicable to a large scale of declarative formal base languages and all of the MMT notions are fully abstract in the choice of the base language.

The purpose of our research is to combine the advantages offered by these two fields into a single platform, the MMT platform. Our aim is to study logical frameworks and LF in particular as foundations for MMT. We are interested to see how much of the logic tools provided by LF can be generalized up to the level of abstraction provided by MMT. The rest of the work that cannot be managed

by MMT will be handled by the LF plugin in MMT. The research is focused on two important algorithms, the type-checking algorithm and the unification algorithm and how they scale up to MMT. Type-checking in LF is an important algorithm since LF allows us to internalize proofs as objects inside the language, so type-checking means correctness of the proof. The unification algorithm can be integrated with other tool support that provide valuable information, such as automated theorem proving, but it also gives interesting applications by itself.

Since MMT does not have types, the type-checking algorithm is particular for the LF foundation. We have already implemented the type-checking algorithm for LF; we will further test it and develop applications for it. Our next goal is the unification algorithm. We will study unification in the general setting of MMT and then we will particularize to LF for the processes that cannot be supported by MMT.

The paper is organized as follows: section 2 describes the theoretical pillars of our research, i.e we will present LF, MMT and the subject of unification. Section 3 sketches the development of the implementation of LF in MMT and provides a general discussion about foundations in MMT. Section 4 presents some interesting applications of our work. We conclude with section 5 that gives a final overview and the work schedule.

2 Preliminaries

2.1 LF

LF [HHP93] is a meta-language that allows the representation of deductive systems given by axioms and rules of inference. It achieves this goal by providing a minimal type systems based on dependent types which provides many attractive tools such as higher-order abstract syntax [PE88] and the Curry-Howard isomorphism [How80]. LF occupies the bottom right corner of Barendregt’s λ -cube of abstractions [Bar92], responsible for the type-term abstraction characteristic to dependent types.

LF contains entities on three levels: terms, types and kinds. Terms are classified by types and types and type families are classified by kinds. We use K to range over kinds, A to range over types, M to range over terms, Σ to refer signatures and Γ to refer contexts. The grammar of the system is the following:

$$\begin{aligned}
K &::= \mathbf{type} \mid \Pi \\
A &::= a \mid A_1 \rightarrow A_2 \mid AM \mid \Pi x : A_1. A_2 \\
M &::= c \mid x \mid M_1 M_2 \mid \lambda x : A. M \\
\Sigma &::= . \mid c : A, \Sigma \\
\Gamma &::= . \mid x : A, \Gamma
\end{aligned}$$

One of the principal judgements in LF are the typing and kinding judgements, $\Gamma \vdash_{\Sigma} t : A$ and $\Gamma \vdash_{\Sigma} A : K$. They are parametrized by a signature containing constants and a context containing the free variables within a term.

We exhibit here two of the typing rules: (see [B.P05] for the complete system)

$$\frac{\Gamma \vdash S : \mathbf{type} \quad \Gamma, x : S \vdash t : T}{\Gamma \vdash \lambda x : S.t : \Pi x : S.T}$$

$$\frac{\Gamma \vdash t_1 : \Pi x : S_1.T \quad \Gamma \vdash t_2 : S_2 \quad \Gamma \vdash S_1 \equiv S_2}{\Gamma \vdash t_1 t_2 : [t_2/x]T}$$

As it can be noticed, the typing judgement uses an additional judgement, namely the equality judgement. The notion of equality we are using is that of the definitional equivalence, namely two terms are equal when they are β - η reducible to one another. We show here two of the equality rules:

$$\frac{\Gamma \vdash \mathit{whnf}(s) \equiv_{wh} \mathit{whnf}(t)}{\Gamma \vdash s \equiv t}$$

$$\frac{\Gamma, x : S \vdash s \equiv tx \quad t \quad \mathit{not} \quad a \quad \lambda}{\Gamma \vdash \lambda x : S.s \equiv_{wh} t}$$

The equality judgment uses the reduction to the weak-head normal form, which is defined by the following rules:

$$\frac{t_1 \rightarrow t'_1}{t_1 t_2 \rightarrow t'_1 t_2}$$

$$\overline{(\lambda x : A.t_1)t_2 \rightarrow [t_2/x]t_1}$$

The function $\mathit{whnf}(t)$ applies these rules to the term t , until no rule is applicable. The judgement $t_1 \equiv_{wh} t_2$ tests two terms in weak head normal form for equivalence. It is necessary to introduce this additional judgement as testing for equivalence at weak head normal form is much easier than the usual equivalence. In the case of whnf , we are basically testing the components of the term for equivalence.

2.2 Unification

Unification has a long history of research [AA01] and it was first started on first-order logic. The study was extended to higher-order logic, but this led immediately to complications.

While unification for first-order logic is decidable and for every solution, there is a most general unifier, as shown by Robinson with the resolution principle [J.A68], the problem is undecidable in higher-order logic even for simply typed lambda calculus. However it is semi-decidable in the sense that if the problem has a solution, it can be found. A first simple algorithm is a generate-and-test one, in which the free variables are substituted with long normal forms. Starting from this idea, Huet [G.H75] designed an algorithm in which this idea

is refined to choose only closed long normal forms for the free variables. Next, the unification equations are reduced until we have only the so called flexible-flexible equations in which the head of both of the two terms is a free variable. It is easily provable that this equations always have a solution, provided that every type is non-empty. Since we are only interested in a yes/no answer, we don't need to resolve these equations but only identify them. The idea was improved by Miller's mixed prefix strategy [D.M92] which gives rise to a more natural formulation of the problem of constraints in the unification process by formulating unification problems as propositions that need to be proven in an unification logic.

The problem of higher-order unification in the dependent lambda calculus was firstly studied by Conan Elliott [C.E89], following the idea of Huet. The $\lambda\Pi$ calculus poses two additional complications: firstly substitutions for free variables in terms may also lead to substitutions in types; thus not only terms need to be unified but also types. Furthermore, the flexible-flexible equations don't always have a solution since in the $\lambda\Pi$ calculus there are types that may not be inhabited. Thus, in this case, we also need to solve the flexible-flexible equations which leads to an exponential explosion in the cases that need to be analyzed.

Other attempts at solving this problem were identifying decidable fragments of this problem. The most important one is the pattern fragment which basically restricts the language to that where symbols of higher order function type are only applied to distinct bound variables. Starting from this observation, Miller designed a logic programming language in which all the unification problems are decidable [D.M91]. However, this language prevents a number of important representation techniques for LF, as discussed in the paper [F.P91]. One solution is to use the same algorithm as the one for the L_λ language and to postpone the unresolved problems as constraints, as it is done in the Twelf system [PS99].

2.3 MMT

MMT [RK11] is a generic, formal language for modular systems used for mathematical knowledge representation. The novelty of MMT resides in the fact that it provides the facilities of both an MKM system by making mathematical knowledge easily accessible, retrievable and transferable across the Internet and those of a meta-language in which other deductive systems may be represented, in a way similar to that of logic frameworks, but one abstraction layer above.

In MMT, mathematical knowledge is defined in terms of four important units: documents, modules, symbols and objects. The biggest unit of mathematical knowledge, the document is represented in MMT as a theory graph, which is a directed acyclic graph where nodes are theories and edges are links. A theory or a module is basically a declaration of symbols, while a link can be thought as a morphism from a theory S to another theory T . The symbol level is characterized by constant and structure declarations. The object level

provides the structure of the symbols inside a theory; objects are basically terms produced by a grammar in which the basic units are constants, variables, applications and bindings.

In order to scale to large implementation and be web-compliant, MMT uses canonical identifiers such that every knowledge item, including those that are induced have a unique global URI. The URI is transformed into a URL, thus relating to the physical location of the items. The grammar of URIs distinguishes between the four levels mentioned above and provides a set of rules such that the URIs formed are unambiguous and provide access to items at each level.

A crucial property of MMT is its foundational independence. Mathematical knowledge can be produced in different reference systems such as set theory or type theory. Each of these major directions come into multiple flavors, i.e. for set theory we have the Zermelo-Fraenkel set theory or the Godel-Bernays theory, or set theories with or without the axiom of choice. Thus, scalability across knowledge requires a representation language that is not committed to any foundation. MMT achieves this goal by encoding a foundation language as a theory; for instance, the two major foundations, ZFC and LF are represented as theories inside MMT. Then, the semantics of a specific theory T in terms of either ZFC or LF is given by a theory morphism between T and the ZFC/LF theory.

The semantics of the foundation is given externally by the typing and equality judgements, which are fundamental for any mathematical theory. Each particular foundation implements these judgements according to its own principles.

3 Foundations in MMT

We will study foundations and both their basic and advanced properties. In particular, we will relate our discussion to the LF logical framework since type theory is one of the main foundations of mathematics, along with set theory and LF is one of the most important type theory. Moreover, we have a considerable number of logics implemented in LF via the Twelf system in the Latin project [CHK⁺11], an advantage which will be used for testing purposes.

3.1 Foundations and LF in MMT

At the language level, a foundation is encoded as a theory which acts as a meta-theory for the other theories. The semantics of an arbitrary theory Γ with meta-theory F is given by F through the typing and equality judgements.

At the implementation level, a foundation is encoded as follows:

```
class Foundation {
  def typing (tm : Option[Term], tp : Option[Term], G :
    Context = Context()) {...} ;
  def equality (tm1 : Term, tm2 : Term) {...} ;
```

}

LF is a particular instance of the Foundation class.

The typing and equality functions are well-established properties of foundations since they give the meaning of the foundation. We will investigate what other properties fit the characterization of a foundation. Unification is one possibility. However, unification has a different status than the other two functions: the semantics of the foundation is given already by the typing and equality judgements while the unification "inherits" from this semantics. But, it is natural to put the unification next to these two functions since the unification algorithm is specific to every foundation.

The unification function will be implemented with the following parameters:

```
def unify(s : Term, T : Term, G : Context)(implicit lib :  
  Lookup) : Boolean = {...}
```

3.2 Basic properties of foundations

The typing and equality judgements are particular for each foundation. For LF, we have already implemented these two functions. The algorithms are basically a translation of the algorithmic version of the typing and equality judgements described in section 2.1. An important characteristic of type-checking is that it is decidable, thus our program always terminates.

MMT does not raise new problems to type-checking. On the contrary, it provides all the necessary "ingredients", i.e the constructs of LF, namely λ , Π , Σ , Γ and other necessary functions such as substitutions.

The functions are defined with the following parameters:

```
def equality(tm1 : Term, tm2 : Term)(implicit lib : Lookup) :  
  Boolean = { equal(tm1, tm2, Context()) }
```

```
def check(s : Term, T : Term, G: Context)(implicit lib :  
  Lookup) : Boolean = { ... }
```

To illustrate what we mean by "translation" of the rules, consider the following typing rule:

$$\frac{\Gamma \vdash T_1 : \mathbf{type} \quad \Gamma, x : T_1 \vdash T_2 : \mathbf{type}}{\Gamma \vdash \Pi x : T_1. T_2 : \mathbf{type}}$$

In code, this appears as:

```

case Pi(x, a, b) =>
  val G2 = G ++ OMV(x) % a
  (T == Univ(1) || T == Univ(2)) &&
  check(a, Univ(1), G) && check(b, T, G2)

```

We still need to implement expansion of definitions for the typing and equality algorithms. An algorithm for expanding definitions of constants is non-trivial since every constant has associated with it a "tower" of definitions and the algorithm is non-deterministic in the following sense: if we need to check that $c == c'$, we need to decide how to explore the tower of definitions for c and c' such that we obtain the right answer with the minimal number of steps.

3.3 Advanced properties of foundations: Unification

We will implement unification in the general setting of MMT; the unification steps that cannot be handled by MMT will be handled by the LF-specific unification algorithm. The general unification algorithm will enable us to unify expressions, independent of the semantics provided by foundation, thus avoiding repetitive work that is handled in the same way by any foundation. However, this approach has limitations in the sense that reduction operations on terms are specific to every foundation. For example, the terms c and $(x : A.x)c$, with c - constant, are unifiable in LF, but would not be unifiable with the algorithm described for MMT. We are thus interested in providing a coherent system that will exploit foundation-independance as much as possible and then hand the unsolvable unification tasks to the LF specific unification algorithm.

3.3.1 General unification for MMT

We will implement an algorithm that will unify generic expressions written in the MMT language. The unification algorithm will unify two expressions based on their forms. This will result in a simple algorithm: it will not take into account any form of reduction on the terms since reduction rules are dependent on a foundation.

The grammar of MMT terms is the following:

$$\begin{aligned}
E &::= c \mid x \mid @(E, E^*) \mid \beta(E, \Gamma, E) \\
\Gamma &::= . \mid \Gamma, x[: E]
\end{aligned}$$

The most important transformations implemented by the algorithm are the following:

- $\Gamma \vdash @(E_1, E_2, \dots, E_n) = @(E'_1, E'_2, \dots, E'_n) \Rightarrow \Gamma \vdash E_1 = E'_1 \wedge E_2 = E'_2 \wedge \dots \wedge E_n = E'_n$
- $\Gamma \vdash \beta(E, \delta, S) = \beta(E', \delta', S') \Rightarrow \Gamma \vdash E = E' \wedge \Gamma \vdash \Delta = \Delta' \wedge \Gamma, \Delta \vdash S = S'$

The resulting algorithm is sound, but not complete. However, there is a special case when the algorithm is complete, namely when $E_1 = E'_1 = c$ and c

is injective. This case corresponds to the case of strict definitions, implemented in the Twelf system [FC98]. This is a concrete example when a concept that was tailored for the LF framework can be extended to the full power of MMT.

3.3.2 LF-specific unification

As it was mentioned, the problem of unification for higher-order logic is undecidable; however, it is decidable for a fragment, called the pattern fragment, which arises in most of the unification problems. The idea is to reduce the higher-order pattern problems into first-order pattern problems. This is possible with the aid of meta-variables and explicit substitutions [FCTG91]. Meta-variables enable to encode unification goals as variables in an unification logic, while substitutions permit to collect the substitutions that are applicable for variables in a term through β -reduction and process them until they become pattern substitutions which have the desirable property of being invertable. In the end, we obtain equations of the form $X[\sigma]$, where X is a meta-variable and σ is invertible. Thus, from here, we find X , by inverting the substitution σ . The problems that don't fall in the pattern fragment are postponed as constraints.

To implement this approach, we need the following steps, as described in [FCTG91]:

1. Transform the terms into a representation with de Bruijn indices.

The de Bruijn representation of a term allows an easier manipulation and computation with terms since it deals with the problem of α -renaming of bound variables. In essence, the de Bruijn notation captures the form of a term, without needing to account for the names of bound variables. The representation of the term $\lambda x.\lambda y.yx$ is $\lambda\lambda 12$. The deBruijn function will be implemented around the following stub:

```
def deBruijn( s : Term , G : Context ) ( implicit
  lib : Lookup ) : Boolean = {...}
```

2. The calculus of explicit substitutions.

We will need to define a theory that will enable us to operate with the concepts of substitutions and meta-variables. This will be the signature of the following grammar:

$$\begin{aligned} A &::= K \mid A \rightarrow B \\ \Gamma &::= nil \mid A.\Gamma \\ a &::= 1 \mid X \mid (ab) \mid \lambda_A.a \mid a[s] \\ s &::= id \mid \uparrow \mid a : A.s \mid s \circ t \end{aligned}$$

3. "Pre-cook" the terms. The idea of pre-cooking is described in the paper. It basically raises the free variables to their proper context.

```
def lower(s : Term, G : Context)(implicit lib :  
  Lookup) : Boolean = {...}
```

4. Solve the unification equations.

```
def unifyEquations(list of unification equations,  
  list of goals, list of constraints): Boolean =  
  {...}
```

The unification problems are pairs of the form $\{s, t\}$, where s needs to be unified with t ; the goals are variables that need to be solved. Whenever we find a solution to an unsolved variable, we immediately replace it in the set of unification problems and continue with the new problem. Unification problems that cannot be solved are postponed as constraints; whenever we solve a new variable, we also replace it in the constraints and try to solve again the constraints. The transformations applied to the equations are those presented in [\[FCTG91\]](#)

4 Applications

4.1 Type inference in the web browser

An application of the type-checking algorithm has already been developed, namely type inference in the web browser. Since MMT can also be run as a HTTP server, this allows the Content Dictionaries of the Latin archive to be displayed in the browser. The application offers the possibility to select an expression and displays for each a menu of functions that can be used with those expressions, among which there is also the type inference function. This sends a query to the server which ultimately invokes the `infer` function of the foundation.

```

document derived.omdoc
  remote module FalsityExt
  remote module NEGExt
  theory IMPExt meta lf
    include IMP
    imp2I : ((ded A → ded B → ded C) → ded A imp (B imp C))
           = [f:ded A → ded B → ded C]impI ([p:ded A]impI ([q:ded B]f p q))
    imp2E : (ded A imp (B imp C) → ded A → ded B → ded C)
           = [p:ded A imp (B imp C)][q:ded A][r:ded B]impE (impE p q) r
  remote module CONJExt
  remote module DISJExt
  remote module Equiv

```

type ×

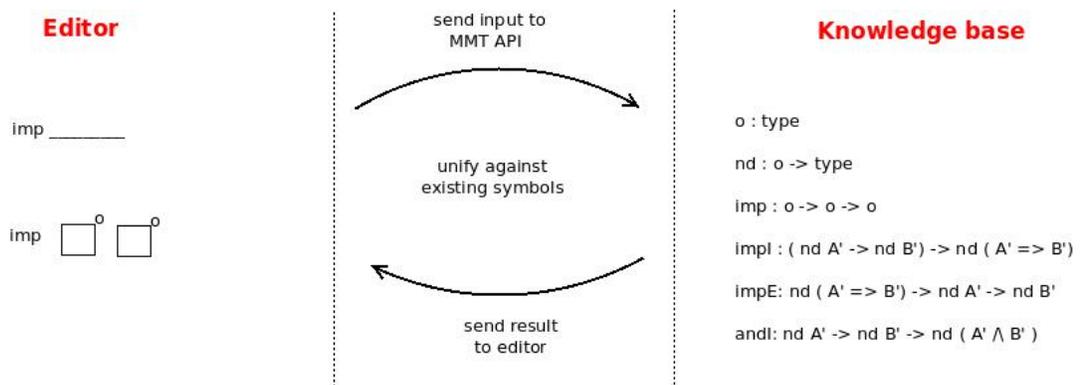
ded A imp (B imp C)

- infer type
- reconstructed types
- implicit arguments
- implicit binders
- redundant brackets
- Fold

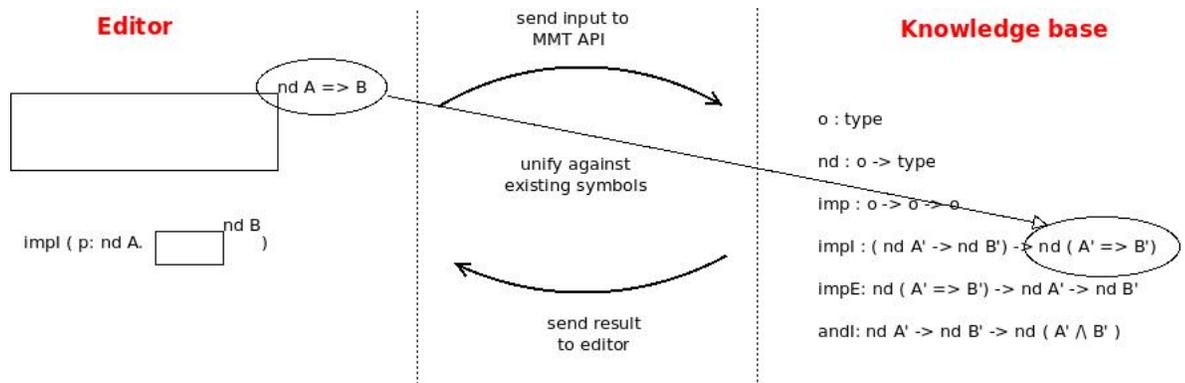
This is very convenient since it provides an interactive way to browse through the documents in the archive and to quickly identify the type of the desired expression, a process that for long expression could take the reader a very long time.

4.2 Auto-completion in a text editor

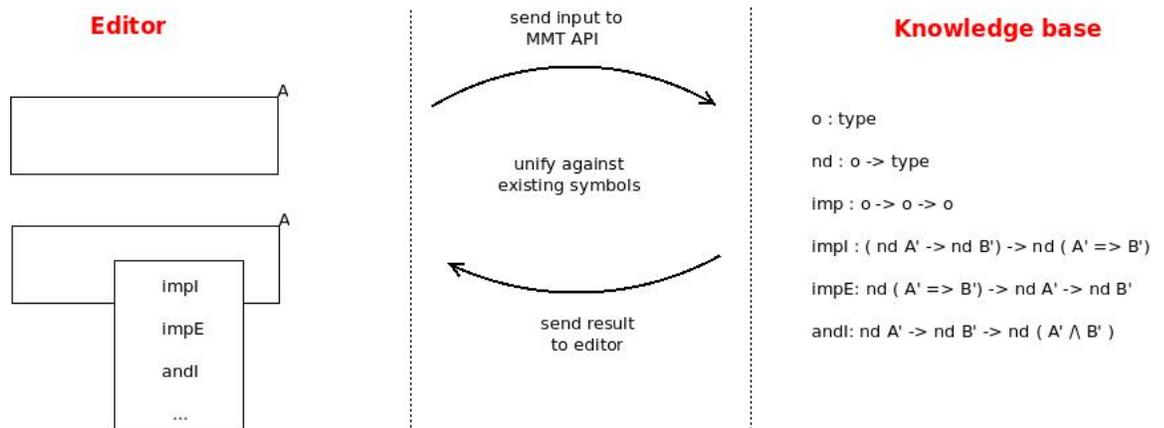
An interesting application for the unification algorithm will be auto-completion in a text editor. We illustrate this by three use cases. Note that the notation \square , superscripted by a symbol stands for the typed arguments of a function that need to be completed.



The above image illustrates the case when the user wants to write a simple connective symbol. The editor will indicate her how many arguments the symbol needs and the type of the symbol.



Suppose that we have already obtained an information about the type of an expression and now we want to complete it with the actual expression. In our case, suppose that we want to write the rule *implI*. This will unify against the rule already represented in the knowledge base and will provide the fields that need to be completed in order to write the complete rule.



In this last case, multiple expressions unify with the type of the expression that the user intends to write. All of these expressions will be returned and displayed in a menu.

5 Conclusion and timeline

By the end of this project, the type-checking algorithm for LF will be implemented and fully functional. Moreover, we will implement a general unification algorithm for the MMT language; this will be a simple algorithm that unifies expressions based on their syntactic form. We will continue with the

unification algorithm for the LF foundations, using the steps described above. The project will be built on top of the already existing implementation of the MMT platform, written in the Scala language.

Our research effort entails future work in two directions. First of all, the two algorithms stand as basis for other logic tools such as type reconstruction; furthermore, they can be used as a basis for a logic programming language, thus allowing proof search and automated theorem proving. Secondly, the algorithms by themselves provide interesting applications, as we have shown above.

The expected timeline for this project is the following:

- Implementing the general unification algorithm for MMT → February 29th
- Testing the type-checking algorithm and finishing the implementation → February 29th
- Implementation of the unification algorithm for LF → May 1st
- Improving readability for the code and documentation → May 1st
- Final report → May 15th

References

- [AA01] A. Robinson and A. Voronkov. *Handbook of automated Reasoning*. The MIT Press, North Holland, 2001.
- [A.M95] A.M. Odlyzko. Tragic loss or good riddance? The impending demise of traditional scholarly journals. *International Journal of Human-Computer Studies*, pages 42–71, 1995.
- [Bar92] H. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1992.
- [BCC⁺04] S. Buswell, O. Caprotti, D. Carlisle, M. Dewar, M. Gaetano, and M. Kohlhase. The Open Math Standard, Version 2.0. Technical report, The Open Math Society, 2004. See <http://www.openmath.org/standard/om20>.
- [B.P05] B. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, Cambridge, Massachusetts, 2005.
- [C.E89] C. Elliott. Higher-order unification with dependant types. *International Conference on Rewriting Techniques and Applications*, 355:121–136, 1989.
- [CHK⁺11] M. Codescu, F. Horozal, M. Kohlhase, T. Mossakowski, and F. Rabe. Project Abstract: Logic Atlas and Integrator (LATIN). In J. Davenport, W. Farmer, F. Rabe, and J. Urban, editors, *Intelligent Computer Mathematics*, volume 6824 of *Lecture Notes in Computer Science*, pages 287–289. Springer, 2011.

- [D.M91] D.Miller. A logic programming language with lambda-abstractions, function variables and simple unification. *Journal of Symbolic Computation*, pages 497–536, 1991.
- [D.M92] D.Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, pages 321–358, 1992.
- [FC98] F.Pfenning and C.Schrmann. Algorithms for equality and unification in the presence of notational definitions. In *Types for Proofs and Programs*, page 1657. Springer-Verlag LNCS, 1998.
- [FCTG91] F.Pfenning, C.Kirchner, T.Hardin, and G.Dowek. Unification via Explicit Substitutions: The case of Higher-Order Patterns. 1991.
- [F.P91] F.Pfenning. Logic Programming in the LF Logical Framework. 1991.
- [G.H75] G.Huet. A unification algorithm for typed λ calculus. *Theoretical Computer Science*, pages 27–57, 1975.
- [GLR09] J. Gičeva, C. Lange, and F. Rabe. Integrating Web Services into Active Mathematical Documents. In J. Carette, L. Dixon, C. Sacerdoti Coen, and S. Watt, editors, *Intelligent Computer Mathematics*, volume 5625 of *Lecture Notes in Computer Science*, pages 279–293. Springer, 2009.
- [HHP93] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.
- [Hil26] D. Hilbert. Über das Unendliche. *Mathematische Annalen*, 95:161–90, 1926.
- [How80] W. Howard. The formulas-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press, 1980.
- [J.A68] J.A.Robinson. New directions in mechanical theorem proving. *International Federation for Information Processing Congress*, pages 63–67, 1968.
- [Koh00] M. Kohlhase. OMDoc: An Infrastructure for OpenMath Content Dictionary Information. *Bulletin of the ACM Special Interest Group on Symbolic and Automated Mathematics*, 34:43–48, 2000.
- [KRZ10] M. Kohlhase, F. Rabe, and V. Zholudev. Towards MKM in the Large: Modular Representation and Scalable Software Architecture. In S. Autexier, J. Calmet, D. Delahaye, P. Ion, L. Rideau, R. Rioboo, and A. Sexton, editors, *Intelligent Computer Mathematics*, volume 6167 of *Lecture Notes in Computer Science*, pages 370–384. Springer, 2010.

- [Pau89] L. Paulson. The Foundation of a Generic Theorem Prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.
- [PE88] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Programming Language Design and Implementation*, pages 199–208, 1988.
- [PS99] F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. *Lecture Notes in Computer Science*, 1632:202–206, 1999.
- [RK11] F. Rabe and M. Kohlhase. A Scalable Module System. see <http://arxiv.org/abs/1105.0548>, 2011.
- [WR13] A. Whitehead and B. Russell. *Principia Mathematica*. Cambridge University Press, 1913.