Generic unification for type theories

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

May 30, 2012

Abstract

Logical and knowldege management operations for deductive systems are expensive to define and implement, thus the need to design frameworks that support the efficient implementation of these tools. We propose and implement two unification algorithms in a framework that supports knowledge management tools and offers a proper setting for the implementation of logical support. We overcome the challenge posed by the genericity of the language by proposing a new approach to the problem of unification, namely by composition of features, which is available in the case of type theories.

1 Introduction

Formal systems have had and continue to have an important role in the development of scientific domains and in related applications. A central concern related to formal systems is to have implementations that support the execution of useful operations within them. Typically, the operations one is interested in are of two types and until now, they have been offered in disjoint sets of systems. The first category is that of **logical operations**, such as unification, automated theorem proving, type-checking or type-reconstruction; these are the operations that produce useful results and new knowldege. The second category is that of **knowledge management operations** that deal with how the system interacts with the exterior and how the exterior affects the system. These operations refer to change management, interactive access to the system and knowledge transfer across the web.

Traditionally, the logical support needed by formal systems has been offered by ad-hoc implementations of the respective formal system. An improvement to this situation is brought by **logical frameworks**, which are general meta-languages for specifying and implementing deductive systems given by axioms and rules of inference. The key advantage of logical frameworks is that they are **independent** of the deductive system one might want to represent and that the operations done in the logical framework **induce** the respective

operations in the deductive system. For instance, unification at the level of the logical framework induces unification at the level of the deductive system. There are many varieties of logical frameworks, but the most notable ones are Isabelle [Pau89], which is based on hereditary Harrop formulas and LF [HHP93], which is a dependantly typed λ-calculi.

However, logical frameworks don't offer knowledge management support, which is supported by Mathematical Knowledge Management languages and tools. **MKM systems** are also 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 content-oriented, markup languages used as standards for representing mathematical knowledge, developed by the research in the MKM area.

MMT [RK11] is the first framework that offers the setting to **combine the advantages** of logical and knowledge management operations. MMT makes use of the fact that both MKM systems and logical frameworks are independent of the underlying deductive system to provide a generic, formal module system that is designed to be applicable to a large scale of formal base languages, given that its notions are fully abstract in the choice of the base language. In MMT, MKM tools are available through the OMDoc standard, with which MMT complies. Logical tools are available in MMT only if we encode a deductive system in MMT and the logical support as offered by the respective deductive system, the most general way of doing this being the encoding of a logical framework.

The purpose of our research is to scale one of the basic logical operations, **unification** to the level of genericity of MMT. The key challenge is that unification algorithms depend on **equational theories** which are specific to each deductive system. However, MMT lacks any equational theory, except α-conversion.

We propose a new approach to the problem of equational theory, inspired by the **modularity** of MMT: in MMT, a central concept is that of theory; we can compose theories from reusable components which are symbols. We apply the same principle to equational theories and unification algorithms and thus design and implement two unifcation algorithms, one that is generic for the MMT language and another one that is available in case of type theories and works by composition.

This paper is organized as follows: section 2 presents the MMT system and a history of the problem of unification; section 3 describes the unification algorithms in MMT and section 4 discusses the properties of the algorithms and relevant results obtained with the implementation of algorithms. We conclude in section 5 with the description of future work and applications.

2 Preliminaries

2.1 MMT

MMT [RK11] is a **generic, formal language for modular systems** used for knowledge representation. The novelty of MMT resides in the fact that it provides the facilities of both an MKM system 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, knowledge is defined in terms of four important units: documents, modules, symbols and objects. The biggest unit of knowledge, the **document** is represented in MMT as a theory graph, which is a directed acyclic graph where nodes are theories and edges are morphisms. A **theory** Thy is a list of declarations of constants. The **symbol** level is characterized by constant declarations, $[c : E = E]$. The object level provides the structure of the symbols inside a theory; **objects** ω are **constants** c, **variables** x, **applications** @ and **bindings** β.

The fragment of MMT relevant for our work is the following:

$$\begin{array}{lll} Con & ::= & c : \omega = \omega \mid c : \omega \mid c = \omega \mid c \\ Thy & ::= & Con^* \\ \omega & ::= & T?c \mid x \mid @(\omega, \omega^+) \mid \beta(b, \Gamma, \omega) \\ \Gamma & ::= & . \mid \Gamma, x[: \omega] \end{array}$$

A crucial property of MMT is its **foundation independence**. This is achieved through a representation language that is not committed to any foundation such as set theory or type theory, thus enabling scalability across knowledge. Foundation languages are encoded as theories; for instance, the two major foundations, ZFC [HJ84] and LF [HHP93] are represented as theories inside MMT.

Type theories are another example of MMT theories, which embody untyped constants. In MMT, both **terms and types** are represented as **MMT terms**; in other words, MMT does not distinguish between terms and types. For instance, the LF type theory is a theory declaring the constants Π and λ; the term $\lambda x : A.x$ is encoded as the MMT term $\beta(\lambda, [x : A], x)$ and the type $\Pi x : A.B$ is encoded as the MMT term $\beta(\Pi, [x : A], B)$ (A, B are constants of the theory).

Throughout the rest of the section, we will also need the operation of **head** on types; this gives the type constructor of a type term. For instance, $head(\beta(\Pi, [x : A], A)) = \Pi$, where A is a constant.

2.2 Unification

Unification is an essential operation in logical systems, the analogue of solving polynomial equations in algebra. In its essence, the unification problem can be expressed as follows: given two terms in a language containing some variables, find, if it exists, the substitution which makes the two terms equal.

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 λ 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 contraints, as it is done in the Twelf system [PS99].

3 Unification in MMT

In this section, we present two unification algorithms: the first treats the general problem of unification at the level of genericity of MMT. The second one treats a subset of the unification problems, namely unification problems in type theories, where the type at which we want to unify is known.

3.1 Problem formulation and notations

The generic problem of unification can be written formally in the following way: $\mathbf{\Delta}, \mathbf{\Gamma} \vdash \mathbf{t_1} = \mathbf{t_2}$. For the restricted version of unification for type theories, we use the following problem formulation: $\mathbf{\Delta}, \mathbf{\Gamma} \vdash \mathbf{t_1} = \mathbf{t_2} : \mathbf{T}$.

Throughout the rest of the section, we use the following concepts and notations:

- Δ : MMT context, encoding the **metacontext** of the problem; it contains the variables for which we want to find solutions; the variables will be named **metavariables** and denoted by capital letters, $X, Y, ...$
- Γ : MMT context, encoding the **context** of the problem; it contains the bound variables of the problem.
- t_1, t_2 : MMT terms, encoding the **terms** which we want to unify
- T : an MMT term, encoding the **type** at which we want to unify

A **solution** to the problem is a **substitution**, σ that provides values for all the metavariables in the metacontext such that $\sigma(t_1) = \sigma(t_2)$.

3.2 Generic unification

As it was previously described, MMT lacks any equational theory, except α-conversion, so the unification algorithm is basically a pattern matching algorithm.

The approach to the problem is straightforward: given two MMT terms, we analyze their shapes. If their shape is different, the terms cannot unify at the level of MMT. If they have the same shape, we recurse in the components. The algorithm can be expressed in the following way:

1. If one of the terms is a **metavariable**, X that does not occur in the other term, we assign the value of the second term to X and substitute it in all branches. We continue the unification process with the substituted terms.
2. If one of the terms is a **constant**, c, the algorithm succeeds when the other term is a constant. In case the constants have definitions, we continue with the unification process for the definiens. Otherwise, the algorithm succeeds only when the constants are equal.
3. If one of the terms is an **application**, $@(E_1, E_2)$, the other term must also be an application and we recurse into components. Otherwise, the algorithm fails.
4. If one of the terms is a **binding**, $\beta(b, \Gamma, t)$, the other term must also be a binding. The algorithm succeeds when the binders are the same, the contexts have the same length, the types of the respective variables in the context unify and also the bound expressions unify.

3.3 Foundation-specific Unification

The type theory unification algorithm makes use of the information given by the type at which we want to unify. In order to do this, we introduce the notion of **type theory feature** and represent type theories as specific sets of features. To understand what is the rationale behind the concept of features, we first give an overview on type theories in section 3.3.1 and then state the formal definition of features in section 3.3.2.

3.3.1 Types and type theories

A type theory or a type system is a syntactic system that classifies phrases according to the values they compute [B.P05b]. It is defined in terms of three components, a **formal language, typing judgements and equational theory**. We will describe what these concepts mean and exemplify them as they appear in one particular type theory, the *LF* [HHP93] type theory. Our discussion follows [B.P05a].

1. **Formal language**. The language of a type theory provides symbols and a grammar that shows how to combine the symbols in order to obtain the terms and types of the theory. At this level, there are two important concepts, namely the **type constructors** and the **introduction symbol** of a given type constructor. The type constructor is a symbol that allows us to construct types and the introduction symbol defines a symbol characteristic for terms of a given type.
   In the case of LF, Π is the type constructor for dependent types and λ is the introduction symbol for dependently-types functions.

   An important characteristic of the introduction symbol of a type constructor is the **injectivity** of the symbol. Injectivity allows us to transform the problem of equality of terms into the equality of their respective components. Formally, injectivity of a symbol s, in the unary case and without binding, means that if $sA = sB$, then $A = B$. We can generalize this to binding, in which case injectivity amounts to an extensionality rule. λ is an injective symbol and generally, the theories with which we deal have injective introduction symbols.

2. **Typing judgements**. Typing judgements are a means to attribute types to terms. Associated with the typing judgements are the **introduction rules** and **elimination rules**, that define how the introduction symbol of a type constructor creates new terms or is eliminated from terms. In LF, the typing judgement is written in the following way: $\Gamma \vdash t : A$. The introduction rule for the Π constructor is as follows:

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

and the elimination rule:

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

3. **Equational theory**. Equality is given by **reduction** and **expansion** rules by which terms that have different shapes are brought to the same form. The reduction rule is usually refered by the name β-rule and it enables the computations in a term; the expansion rule is usually called η-rule and it brings terms of a certain type to a standard form.
   In LF, the β-rule is:

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

   and the η-rule is:

$$\frac{t : \Pi x : A.B}{t \to \lambda x : A.tx}$$

3.3.2 Features

Features encode the information that is relevant for a type theory. A feature associates with a **type constructor**, an **introduction symbol**, **introduction** and **elimination rules** and **reduction** and **expansion** rules. We will use the notational conventions for symbols of the MMT grammar throughout this section.

Definition: A **typing judgement** is a tuple $\Theta = (\Gamma, \omega, \omega)$ of an MMT context and two MMT terms, the first denoting a term and the second denoting a type.

Definition: A **feature** is a tuple $\Phi = (\mathbf{typeConstr}, \mathbf{introSymbol}, \mathbf{intro}, \mathbf{elim}, \mathbf{reduction}, \mathbf{expansion})$, where:

1. **typeConstr** is an MMT constant, denoting a type constructor
2. **introSymbol** is an MMT constant, denoting the introduction symbol
3. **intro** is a function from the set of typing judgements to the set of lists of typing judgements, denoting the introduction rule
4. **elim** is a function from the set of typing judgements to the set of lists of typing judgements, denoting the elimination rule
5. **reduction** is a function from the set of terms to the set of terms, denoting the reduction rule
6. **expansion** is a function from the set of typing judgement to the set of terms, denoting the expansion rule

With the *introSymbol* component, we will also associate a boolean flag indicating whether the symbol is **injective** or not.

Definition: A **type theory** is a list of features.

An **example** of type theories as lists of features can be given in the case of the LF type theory. According to the principle of features, LF can be defined with a single feature, $DepType$ that has the following definition: $DepType = (\Pi, \lambda, introduction, elimination, \beta, \eta)$, with the notations as in section 3.3.1.

3.3.3 Algorithm description

We will describe an algorithm that unifies two MMT terms representing terms of a type theory at a given type. The algorithm has an additional **input**, which is a list of features, denoted by **TT**. All the other notations are as described in section 3.1.

The **idea** of the algorithm is to transform terms that are at a composed type into terms at a simple type, by applying the expansion rules and recursing into components; once the algorithm reaches terms with simple types, we apply, if possible, reduction rules. The idea is inspired by the approach on the problem of equivalence and canonical forms in LF, as described in [RF05].

The steps of the algorithm can be described in the following way:

1. If the head of the type **T** matches with the **typeConstr** component of any of the features in **TT**, we apply the **expansion** function of the respective feature to the terms. We then apply the **intro** component of the feature to find the components of the terms.
   (a) If the **introSymbol** is injective, we recurse into these components. Note that the **intro** rule returns typing judgements, so we have the information needed to call the unification function recursively. Also, note that we need to unify firstly the types of the components to be able to call the function recursively; we will use the generic MMT unification algorithm for that.
   (b) If the **introSymbol** is not injective, we apply the generic MMT unification algorithm.
2. In the contrary case, we iterate through the **reduction** rules of the features and check to see if any is applicable.
   (a) If there is one, the algorithm continues with the reduced terms.
   (b) Otherwise, we call the generic MMT unification algorithm for the two terms.

4 Implementation

We have implemented these two algorithms in Scala, using the existing MMT platform. The problem formulation, the typing judgements and the features directly translate into Scala classes. The rest of the implementation followed the steps as described above.

We have also provided a generic class for unification operation that contains operations such as *occurCheck* (verifying if a variables appears in a terms) and *replace*(replacing a variable with a value in a term),that appear in all the unification algorithms.

We have implemented the **LF** type theory as a list of features containing the feature *DepType* as exemplified in section 3.3.2 and we have tested the unification algorithms with this type theory.

1. ***Generic unification algorithm.*** The typical problem where the algorithm succeeds is the following: $\beta(\lambda, [x : A], @(c, x)) = \beta(\lambda, [z : A], @(Y, z))$, with $\Delta = [Y], \Gamma = []$ and c, A arbitrary constants of the LF theory. However, the algorithm fails for simple cases, because of the lack of equational theory. An example in this sense is the following: consider the problem $@(\beta(\lambda, x : A, x), Y) = c$, with $\Delta = [Y]$, $\Gamma = []$ and c and A being constants. The algorithm will fail to unify in MMT, but in LF this would unify since, by the β-rule of LF , $@(\beta(\lambda, x : A, x), Y) \to Y$, which offers the solution $\sigma = [c/Y]$ to the problem.

2. ***Foundation-specific unification algorithm.*** This algorithm succeeds in the case of the above example. Generally speaking, the algorithm will succeed if the terms of composed types that appear in application, @ are in their standard form at that type because, in this way, the algorithm can apply reduction rules once it has reached a simple type. The algorithm does not succeed for more complicated examples, such as the following problem: $\beta(\lambda, [x : \beta(\Pi, [y : A], A)], @(x, Z)) = \beta(\lambda, [x : \beta(\Pi, [y : A], A]], @(Y, x)) : \beta(\Pi, [x : A], \beta(\Pi, [y : A], A))$, with $\Delta = [Y, Z]$, $\Gamma = []$ and A a constant. The algorithm will simplify this problem to $@(@(x, Z), t) = @(@(Y, x), t)$, where t is a variable of dependent type, from where it cannot advance anymore, since no computation rules are applicable. A solution in this case is $\sigma = [\beta(\lambda, [x : A], @(x, Z))/Y]$ and Z arbitrary, but the algorithm cannot infer the shape of terms from the context; in this case, it doesn't infer that Y is of function type, so it has to be a λ term.

Soundness and Completeness. Both algorithms are sound; this is straightforward to prove since the solution obtained at the MMT level is a solution for the problem in the encoded theory, once we translate the solution back into the language of interest. However, neither algorithm is complete, as shown by the examples above, but the second algorithm represents an improvement of the first one.

5 Conclusions

Logical support for deductive systems can be found at the most advanced level in logical frameworks; however, these do not support MKM tools, which are of increasing value nowadays. We have made a first step to bridge the gap between these two by implementing two unification algorithms in the MMT framework, which supports MKM tools. MMT offers the possibility of logical support by encoding logical frameworks and their associated logical tools into the framework, but it did not provide logical tools at the level of the MMT language. The high genericity of the language posed a challenge, which was overcome by a new approach to unification algorithms, namely by composition of features.

5.1 Future Work

Future work for this thesis lies in two main directions, namely enriching the features with additional properties and improving the unification algorithm.

1. **Enriching features**. Unification algorithms depend not only on **equational theories**, but also on **structural theorems**. Structural theorems offer generic shapes for terms of a given type. For instance, in the case of the λ calculus, the structural theorem says that a term of type $\alpha^n \to \beta$, with the head of type $\alpha^m \to \beta$, has the form: $\lambda x^n.h(H_1x^n)...(H_mx^n)$, where the superscripts n, m indicate vectors of components and $H_1, ..., H_m$ are new heads. These structural theorems are exploited in unification algorithms; for instance, in Huet's algoritm [G.H75] for unification in case of λ-calculus, the structural theorem of λ calculus is the basis for the **imitation** and **projection** rules of the algorithm. Thus, structural theorems should also be part of the components of features.

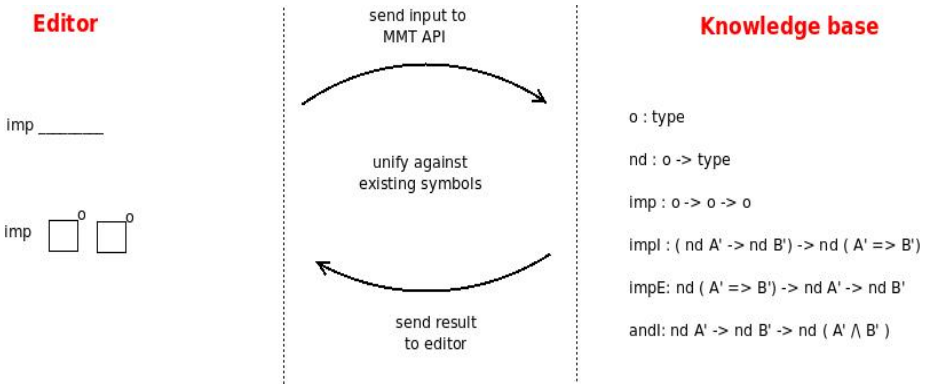
2. **Improving the algorithms**. The algorithms could be improved if at the moment when they stop at terms that represent eliminations where no reduction rule is applicable, the algorithm could do type inference for the type of the components and then use the information given by the type to provide a shape for the term. Type inference can be coded as a second-class component of features, since it is based on the **elim** component of features.

Of course, a major objective of future work is the implementation of the rest of logical tools in the MMT platform, the **type reconstruction** algorithm being the first step in this direction. Moreover, the benefits of unification by composition of types need to be further tested by implementing more type theories as lists of features and testing the performance of the algorithms.

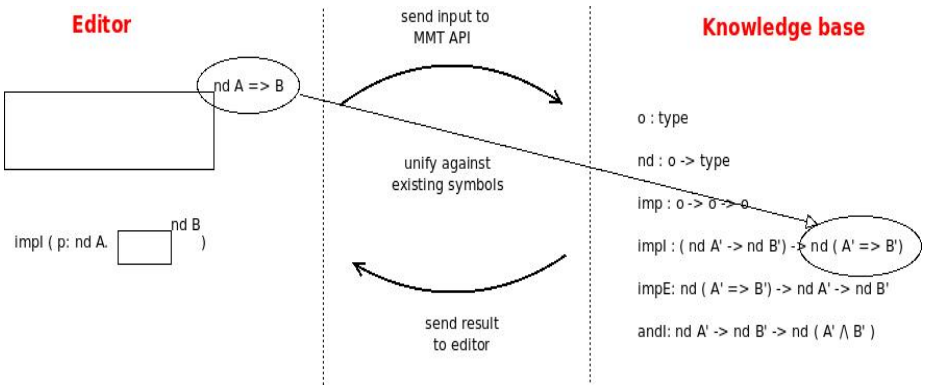
5.2 Future Applications

An interesting application for the unification algorithm will be **auto-completion in a text editor**. Auto-completion is based on unification between the expression currently written by the user and expressions already existing in a knowledge base. For the purpose of this example, we will use encodings of the LF[HHP93] type theory, as they appear in the Twelf system [PS99]. This will use the unification algorithm based on the $DepType$ feature, described in section 3.3.2.

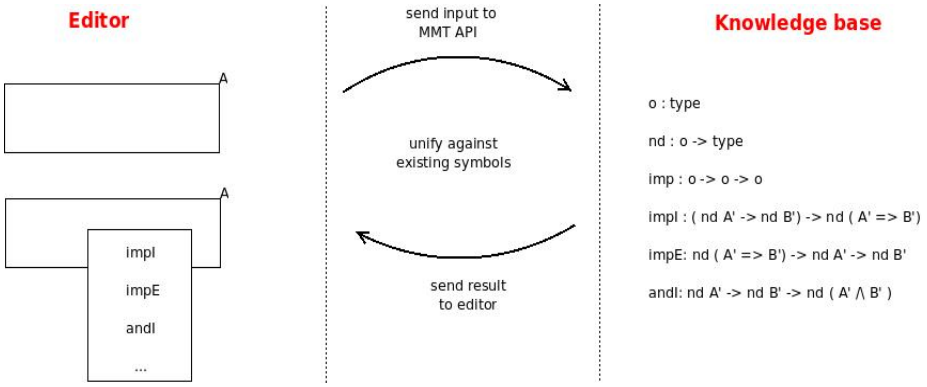
Note that the notation $\square$, superscripted by a symbol stands for the typed arguments of a function that need to be completed.



The above image ilustrates 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 $impI$. 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.



] ../proposal/dia3.jpg

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.

References

[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.P05a] B.Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, Cambridge, Massachusetts, 2005.

[B.P05b] B.Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, Massachusetts, 2005.

[C.E89] C.Elliott. Higher-order unification with dependant types. *International Conference on Rewritting Techniques and Applications*, 355:121–136, 1989.

[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.

[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.

[HJ84] K. Hrbacek and T. Jech. *Introduction to Set Theory*. Marcel Dekker Inc., New York, 1984.

[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.

[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.

[RF05] R.Harper and F.Pfenning. On equivalence and canonical forms in the LF type theory. *ACM Trans. Comput. Logic*, 6:61–101, 2005.

[RK11] F. Rabe and M. Kohlhase. A Scalable Module System. see `http://arxiv.org/abs/1105.0548`, 2011.