JACOBS UNIVERSITY BREMEN

100392 - GUIDED RESEARCH MATHEMATICS II

Formalizing Syntactical Objects within Formalized Set Theory

Author: Vladimir ZAMDZHIEV
First Advisor: Prof. Dr. Michael KOHLHASE
Second Advisor: Dr. Florian RABE
Date: JULY 7, 2011

Abstract

In formalized mathematics, the distinction between the notions of syntax and semantics is very important. However, this distinction in traditional mathematics is not so paramount. Aided by their intuition, mathematicians often mix the two notions by employing a mathematical tool called quotation. Quotation is a mechanism to construct a semantic object from the syntactical structure of some expression. It is used by mathematicians all the time in many areas of mathematics.

Our long term goal is to gain a better understanding of quotation from a formal perspective. For that, however, we first need to select a formal language in which to reason about it. As our language of choice we use OpenMath, which is a generic formal language, designed to be foundationally unconstrained. We express the OpenMath objects in ZFC set theory, which has already been formalized using the Edinburgh Logical Framework. Finally, we formalize the OpenMath objects using the proposed representations into the formalied ZFC set theory, thereby setting the foundations for further formal reasoning.

Contents

1 Introduction

1.1 Formalized Mathematics

Formalized mathematics consists of mathematical theorems and proofs stated in a formal language, with enough detail that a computer program (called a proof assistant) can mechanically verify all of the steps, thereby certifying correctness. Contrary to traditional mathematics, formalized mathematics entails a strict distinction between the notions of *syntax* and *semantics*. The syntax describes how one can create and manipulate expressions in the formal language. The semantics define the meaning of each expression. An interpretation function is used which maps each expression into some mathematical object outside of the formal language. This distinction is not so fundamental in traditional mathematics. In fact, mathematicians often mix the two. An example of this is the definition of polynomials which involves both notions. There is another completely semantic definition which involves categories that is equivalent to the previous one. However, the difference between syntax and semantics is of great importance to formalized mathematics as computers can only work with syntax and not with semantics.

1.2 Quotation and Evaluation

Quotation is a mechanism that allows the mathematician to construct a semantic object purely from the syntactical structure of some expression. Then, for every expression e we can get a new expression which is 'e' - its quotation. The semantic interpretation of the quotation of the expression should be a mathematical object which describes the syntactic structure of e. Thus, for each expression we can acquire two meanings - one of them is the meaning of the expression itself as defined by the interpretation function and the other to be the mathematical object which describes its quotation and thus gives us information about its syntactical structure. This allows for the formal language (or logic) to talk about its own expressions. This process is called reflection.

Evaluation is a mechanism that allows the mathematician to obtain the semantic value of an expression that represents the syntactical structure of another expression. If a is an expression which denotes the syntactical structure of another expression, say b, then $\underline{a}$ denotes the value of b.

1.3 Examples of Quotation

Quotation is used by mathematicians all the time. An example of quotation is the computation of derivatives for polynomials. Consider the function

$$f(x) = 3x^2$$

The derivative of the function f at a is defined to be

$$f'(a) = \lim_{h \to 0} \frac{f(a+h) - f(a)}{h}$$

However, instead of using the definition of derivative to compute $f'(x)$ we can do that in a purely syntactical way - namely

$$f'(x) = (3x^2)' = 2 \times 3x^{2-1} = 6x$$

For that, however, we need to first consider the syntactical structure of the expression $3x^2$ and in doing so we are using quotation. By performing this expression based differentiation we compute an expression which denotes the syntactical structure of the result we need. By using evaluation on the said expression we get the expression which represents the derivative of the function.

Another example is the construction of the free group. A group G is called free if there is a subset S of G such that any element of G can be written in one and only one way as a product of finitely many elements of S and their inverses disregarding trivial variations such as $st^{-1} = su^{-1}ut^{-1}$. However, in order to construct the free group F_S given some symbol set S we need to perform syntactical operations on words formed by the elements of the set S. This is an example of quotation as the syntactical structure of S is used to create the free group F_S.

2 Motivation

We are trying to gain a better undestanding of what quotation is formally. For that matter, we decide to use OpenMath[BCC$^+$04] as our language of choice. OpenMath is a generic language for mathematical expressions, designed to be foundationally unconstrained. Experimental evidence has shown that one can describe all formal languages using OpenMath. For that, however, we need to formalize foundations of mathematics themselves. Set theory has already been formalized using LF [IR10, HHP93]. As a first step of this guided research we express OpenMath objects in ZFC set theory. Next, using the set-theoretical representations, we formalize the OpenMath objects using the already existing formalization of ZFC set theory. The correctness of the proofs is certified by the Twelf system [PS99]. Finally, individual formal languages can be uncovered from this by imposing well-formedness constraints.

3 Preliminaries

3.1 OpenMath

OpenMath[BCC$^+$04] is a standard for representing mathematical data in as unambiguous a way as possible. It can be used to exchange mathematical objects between software packages or via email, or as a persistent data format in a database. It is tightly focussed on representing semantic information. Formally, an OpenMath object is a labelled tree whose leaves are the basic OpenMath objects integers, IEEE double precision floats, unicode strings, byte arrays, variables or symbols. OpenMath objects can be built up recursively in a number of ways. One of them is function application, another is attribution, which can be used to add additional information to an object without altering its foundational meaning. Another example is binding objects which are used to represent an expression containing bound variables.

For our purposes we will use a subset of the OpenMath expressions. Precise definitions are given below.

OpenMath represents mathematical objects as terms or as labelled trees that are called OpenMath objects or OpenMath expressions. The definition of an abstract OpenMath object is then the following :

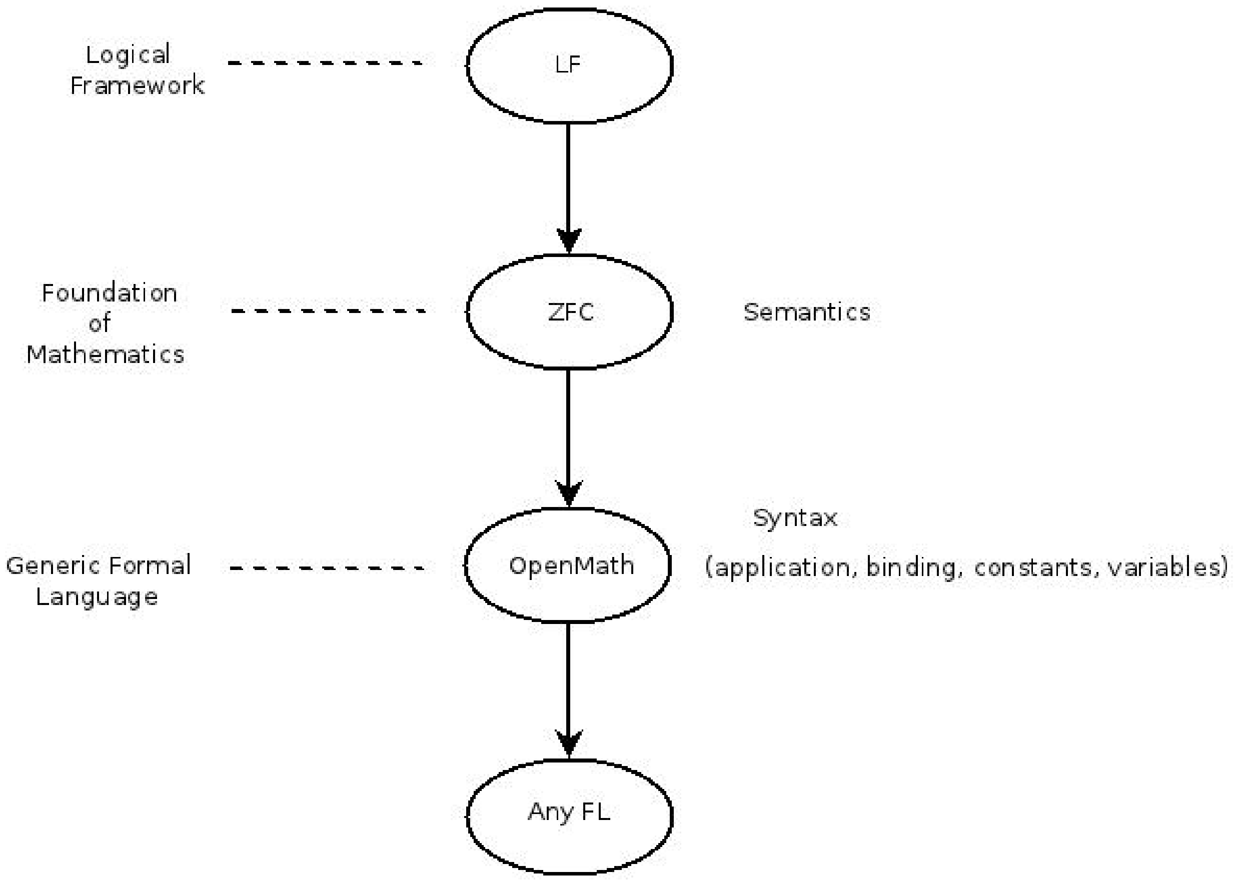

Figure 1: Illustartion of the relevant levels of abstraction

3.1.1 Basic OpenMath Objects

The Basic OpenMath Objects form the leaves of the OpenMath Object tree. A Basic OpenMath Object is one of the following.

- Integer - Integers in the mathematical sense, with no predefined range. They are "infinite precision" integers
- Variable - A variable should be of the form x_i, where i is a natural number.

3.1.2 OpenMath Objects

OpenMath objects are built recursively as follows.

- Basic OpenMath objects are OpenMath objects.
- If $A_1, A_2, ..., A_n (n > 0)$ are OpenMath objects then $@(A_1, A_2, ..., A_n)$ is an OpenMath **application** object.
- If B and C are OpenMath objects and $v_1, v_2, ..., v_n$ $(n > 0)$ are OpenMath variables, then $\beta(B, v_1, v_2, ..., v_n, C)$ is an OpenMath **binding** object

The OpenMath objects are therefore given by the following grammar

$$o := \quad c \quad | \quad v \quad | \quad @(o, o^+) \quad | \quad \beta(o, v^+, o)$$

Denote with L_o the set of all OpenMath objects.

3.2 LF

In this subsection we will present a brief introduction into the relevant parts of LF [HHP93] for this work. The next subsection describes the LF encodings of ZFC set theory and can be used as an example to illustrate the concepts introduced here.

The Edinburgh Logical Framework (LF) provides a means to define (or present) logics. It is based on a general treatment of syntax, rules and proofs by means of a dependently typed lambda calculus. Syntax is treated in a style similar to, but more general than Per Martin-Loef's system of arities. We will work with the Twelf [PS99] implementation of LF and its module system [RS09].

One of the most important concepts of the LF type theory is the *signature.* A *signature* is a list Σ of *kinded type family* symbols $a : K$ and *typed constant* symbols $c : A$. The Twelf module system allows inclusions between signatures. If a signature Σ includes another signature Σ', then all symbols which are declared or included into Σ' are also available in Σ. In this work, we avoided introducing any naming clashes, so symbols in different signatures have unique names and they can be reffered to without specifying the name of the signature.

Another important concept is that of an LF *context.* An LF context Γ is a list of typed variables $x : A$. Given a signature and Σ and a context Γ, the possible LF expressions are *kinds* K, *kinded type families* $A : K$ and *typed terms* $t : A$. `type` is a special kind in LF and type families of kind `type` are called *types.*

We will use the Twelf notation for the rest of the work when describing expressions.

- The dependent fucntion type $\Pi_{x:A}B(x)$ is written $\{x : A\}B\ x$. Similiarly, dependent function kinds are written as $\{x : A\}K\ x$.
- The λ-abstraction $\lambda_{x:A}t(x)$ is written $[x : A]t\ x$. Similiarly, type families are denoted as $[x : A](B\ x)$
- Application is written juxtaposition.

Twelf allows the user to omit the type of bound variables if the type can be inferred from the surrounding expression. We will also omit some types when it is clear what they are in order to improve readability. The reader can use the next section as an example of a Twelf encoding.

3.3 Formalized Set Theory

In this subsection we briefly introduce the formalization of ZFC set theory we will be using later in this work. This has been described in greater detail in [IR10].

3.3.1 Axioms

As a first step, the axioms of ZFC set theory have to be formalized. For that matter, the Twelf signature for First-Order Logic can be used. The names of the components in the Twelf files are composed from ASCII characters, but we will use the traditional symbols to improve readability. In addition to the *FOL* signature, we will also need to define the elementhood predicate before we can formalize the axioms. A new signature called *ZFC* is created and we make the declaration $\in$: $set \longrightarrow set \longrightarrow prop$. *set* and *prop* are types declared in the signature *FOL* used to denote the sets and propositions respectively. We use an infix notation for the elementhood predicate. It takes two sets as arguments and the result is a proposition.

Now, the axioms of ZFC set theory can be written into the signature. The axioms used are extensionality, set existence, unordered pairing, union, power set, specifiaction, replacement, regualarity and choice and infinity. The formalization of the axioms can be made easily from their mathematical definitions. For instance, the axiom of unordered pairing states

$$\forall x \forall y \exists a (\forall z (z = x \vee z = y) \Longrightarrow z \in a)$$

and the corresponding formalization is

$$ax\_pairing : ded\ \forall\ [x]\ \forall\ [y]\ \exists\ [a]\ (\forall\ [z]\ (z = x \vee z = y) \Longrightarrow z \in a)$$

ded is a type family indexed by propositions declared in the signature *FOL*. Terms of type *ded* P represent proofs of the proposition P. In the case of the axioms, a definition is not required as we assume them to be correct. All other terms need to have a definition, so that Twelf can certify the correctness of the proof or definition by performing type evaluation and checking.

3.3.2 Operations

Using the axioms, common operations can be defined. Some relevant examples :

- $uopair : set \longrightarrow set \longrightarrow set = ...$
- $bigunion : set \longrightarrow set = ...$
- $filter : set \longrightarrow (set \longrightarrow prop) \longrightarrow set = ...$

Given two sets X and Y, *uopair* X Y encodes $\{X, Y\}$. Given a set X, *bigunion* X represents the set $\bigcup X$. Given a set X and a unary predicate P, $filter\ X\ P$ represents the set $\{x \in X | P(x)\}$. We have omitted the definitions of the symbols.

From the basic operations we can define others that are more complex or just more convenient to work with. One such is the binary union $X \cup Y$, which by definition is $X \cup Y := \bigcup \{X, Y\}$. It is formalized as

$$union : set \longrightarrow set \longrightarrow set = [x][y]\ bigunion\ (uopair\ x\ y)$$

3.3.3 Typed Set Theory

The following symbols are of interest to us as they allow the creation of new specific types out of existing sets, thereby making reasoning about elementhood and set equality easier.

- $Elem : set \longrightarrow type$
- $elem : \{a : set\}\ ded\ a \in set \longrightarrow elem\ A$
- $which : elem\ A \longrightarrow set$
- $why : \{a : elem\ A\}\ ded\ (which\ a) \in A$
- *eq_which : ded which (elem a P) = a*

Again, definitions are omitted. *Elem A* encodes the class $\{a|a \in A\}$. Given a set A, *Elem A* returns a new type, such that any term of type $a : Elem\ A$ is a member of the set A. *elem a P* returns an element of the class. The second argument P is a proof that $a \in A$. *which* and *why* can be thought of as projections acting on *elem* that return the set and the proof respectively.

Using these basic typed set constructors, more complicated typed operation symbols can be introduced, such as typed equality ($=^*$), typed quantifiers ($\forall^*, \exists^*$) and other operations similiar to their untyped counterparts.

3.3.4 Lists and Trees

The OpenMath syntax allows the binding and application objects to accept any number of arguments. We address this by using lists. A list over a set X is a sequence of elements of X which has a finite length. We make use of the signature *Lists* and we briefly introduce some of the more important symbols.

- *list : set* $\longrightarrow$ *set*
- *List : set* $\longrightarrow$ *type* = $[a]$ *Elem (list a)*
- *nil : List A*
- *cons : Elem A* $\longrightarrow$ *List A* $\longrightarrow$ *List A*

list X produces a list of any length with elements from the set X. A term of type *List* X will be a list of elements of X. *nil* and *cons* have the same meaning as in the traditional functional programming constrcuts. *nil* is used to denote the empty list over any set of elements. *cons* a L takes an already existing list L and produces a new list out of L by appending the element a to the left of L (a becomes the first element in the new list and the rest are ordered as in L).

In section 3.1 OpenMath objects are defined in terms of trees. In our formalization we also naturally make use of trees when defining the objects. For that matter, we use the signature $Trees$ and describe it below.

- *treelike : set* $\longrightarrow$ *set*
- *leaf : Elem A* $\longrightarrow$ *Elem (treelike A)*
- *compose : Elem A* $\longrightarrow$ *List (treelike A)* $\longrightarrow$ *Elem (treelike A)*

treelike A is a set which contains all finite trees with nodes labelled with elements of A. However, not all of its elements are trees. Nevertheless this is sufficient for our purposes. $leaf\ a$ produces a tree with a single node labelled a. Given an element a and a forest $A_1, A_2, ..., A_n$ we can create a new tree using *compose* a $[A_1\ A_2\ ...\ A_n]$. The root of the tree is labelled a and it has subtrees $A_1, A_2, ..., A_n$

4 Expressing OpenMath Objects in ZFC Set Theory

4.1 ZFC Representations

In this section we will express the set of OpenMath objects using ZFC Set Theory by providing a bijection from L_o to the set U which we will define below.

Define $C := \{(1, n)|n \in \mathbb{N}\}$. We will use C to represent OpenMath constants.

Define $V := \{(2, n)|n \in \mathbb{N}\}$. We will use V to represent OpenMath variables.

Define $T := C \cup V$

If X is some set, then let $F(X)$ be the set defined by

$F(X) := X \cup \{(3, f, arg) | f \in X, arg \in X^+\} \cup \{(4, b, vars, t) | b, t \in X, vars \in V^+\}$

Define $G(0) := T$ and $G(n+1) := F(G(n))$ $(n \geq 0)$

Finally, let

$$U = \bigcup_{n \in \mathbb{N}} G(n)$$

4.2 Bijection

We will present a bijection $\phi : L_o \to U$ by providing an inductive definition over the OpenMath grammar. Let ϕ be defined by :

If c is an OpenMath constant, then

$$\phi(c) = (1, c)$$

If x_i is an OpenMath variable, then

$$\phi(x_i) = (2, i)$$

If $@(A_1, A_2, ..., A_n)$ is an OpenMath application object, then

$$\phi(@(A_1, A_2, ..., A_n)) = (3, \phi(A_1), \phi(A_2), ..., \phi(A_n))$$

If $\beta(B, v_1, ..., v_n, C)$ is an OpenMath binding object, then

$$\phi(\beta(B, v_1, ..., v_n, C)) = (4, \phi(B), \phi(v_1), ..., \phi(v_n), \phi(C))$$

Proof that ϕ is a bijection can be found in [Zam11b].

4.3 Properties

We will prove the following theorems in our formalization. The theorems are stated informally here, for the precise formulation see 5.3.

Theorem 4.1. *Let P be a unary predicate on the OpenMath objects. Then, if*

1. *P is true for any OpenMath constant*
2. *P is true for any OpenMath variable*
3. *If P is true for f, $a_1, a_2, a_3, ...a_n$, then P is true for $@(f, a_1, a_2, ..., a_n)$*
4. *If P is true for B, C, $v_1, v_2, ..., v_n$ then P is true for $\beta(B, v_1, v_2, ..., v_n, C)$*

all hold, then P holds for any OpenMath object.

Theorem 4.2. *Any OpenMath object is equal to some OpenMath constant, variable, application or binding object.*

Theorem 4.3. *Each OpenMath constructor function is injective.*

Theorem 4.4. *The different OpenMath constructor functions produce different OpenMath objects.*

5 Encoding OpenMath Objects in Formalized Set Theory

5.1 Untyped OpenMath Objects

The full encodings span more than 300 lines and can be seen in [Zam11a]. Here we provide an overview of the formalization and list only some of the shortest definitions.

We begin by introducing the object constructors

- *oms : Nat $\longrightarrow$ Elem (treelike nat) $=$ $[n]$ compose Zero (cons (leaf n) nil).*
- *omv : Nat $\longrightarrow$ Elem (treelike nat) $=$ $[n]$ compose One (cons (leaf n) nil).*
- *oma : Elem (treelike nat) $\longrightarrow$ List (treelike nat) $\longrightarrow$ Elem (treelike nat)$=$ $[fun][args]$ compose Two (cons fun args)*

We omit the binding objects here to improve readability as they are very similiar to the application objects, but they can be seen in the actual encodings. *nat* is the set of natural numbers, while Nat is a type and terms of type Nat are natural numbers. The *oms* constructs an OpenMath constant (or symbol) and it is encoded as a tree with root labelled 0 and one child labelled n. The *omv* constructs an OpenMath variable and is encoded in the same way as *oms* except that the root is labelled 1. The application object *oma* is a tree with root 2. The leftmost child is the root of another tree that represents the object to be applied to the arguments. The rest of the children are the roots of trees that represent other OpenMath objects.

Next we introduce some of the important subsets of the OpenMath objects.

- *omsymbols : set = Image* $[n]$ *(which (oms n)).*
- *omvars : set = Image* $[n]$ *(which (omv n)).*
- *ombaseterms : set = omsymbols* $\cup$ *omvars*

These sets correspond to the sets C, V and T in section 4.1. Finally, we can create the set of OpenMath objects using the symbols below.

- *Iterate : Elem (powerset (treelike nat))* $\longrightarrow$ *Elem (powerset (treelike nat))*
- *G : Nat* $\longrightarrow$ *Elem (powerset (treelike nat)) = ind_fun Ombaseterms Iterate*
- *g : Nat* $\longrightarrow$ *set =* $[n]$ *which (G n)*
- *omobjects : set = Bigunion g*

The symbols *Iterate*, G, *omobjects* correspond to the functions F and G and the set U in section 4.1 respectively.

5.2 Typed OpenMath Objects

The encodings in the previous section are straightforward and easy to implement in Twelf. The vast majority of the work required for the formalization was into defining the proper typed equivalents of the OpenMath constructs and proving the theorems listed in section 4.1.

The typed constructors are

- *Oms : Nat* $\longrightarrow$ *Elem omobjects*
- *Omv : Nat* $\longrightarrow$ *Elem omobjects*
- *Oma : Elem omobjects* $\longrightarrow$ *List omobjects* $\longrightarrow$ *Elem omobjects*

These constructors use *elem* in order to construct the same tree object as its untyped equivalent which is in addition paired with a proof that the tree object is in the set of OpenMath objects - *omobjects*.

5.3 Formalized OpenMath Object Properties

The properties in section 4.3 have been proved. Their proofs are involved and we only describe the type of the theorems and properties. Again, we omit the cases for the binding objects.

T1 : {P : Elem omobjects $\longrightarrow$ *o}*
ded ($\forall^*$ $[n]$ *P (Oms n))* $\longrightarrow$
ded ($\forall^*$ $[n]$ *P (Omv n))* $\longrightarrow$
ded ($\forall^*$ $[f]$ $\forall^*$ $[args]$ *((P f* $\wedge$ *(list_for_ all P args))* $\Longrightarrow$ *P (Oma f args)))* $\longrightarrow$
ded ($\forall^*$ $[o]$ *P o)*

T2 : ded $\forall^*$ $[o]$ *(((*$\exists^*$ $[n]$ *o Eq (Oms n))* $\vee$ *(*$\exists^*$ $[n]$ *o Eq (Omv n)))* $\vee$ *(*$\exists^*$ $[f]$ $\exists^*$ $[a]$ *o Eq (Oma f a)))*

The injective properties are written below

Oms_ inj : ded (Oms N) Eq (Oms N') -> ded N Eq N'

Omv_ inj : ded (Omv N) Eq (Omv N') -> ded N Eq N'

Oma_ injl : ded (Oma A B) Eq (Oma A' B') -> ded A Eq A'

Oma_ injr: ded (Oma A B) Eq (Oma A' B') -> ded B Eq B'

And finally, the properties from the last theorem in 4.3

Oms_ne_Omv : ded not (Oms N) Eq (Omv N')

Oms_ne_Oma : ded not (Oms N) Eq (Oma A B)

Omv_ne_Oma : ded not (Omv N) Eq (Oma A B)

6 Future Work

Part of the future work will involve finding a way to use quotation and evaluation within the formalization of OpenMath objects. This can be done by extending the OpenMath syntax to support quotation and evaluation and defining proper semantics. Another way would be to use special symbols for quotation and evaluation which are possibly defined within the framework.

7 Related Work

Chiron[Far07] is a multi-paradigm logic, which is intended to be a practical, general-purpose logic for mechanizing mathematics. It is a derivative of von-Neumann-Bernays-Goedel (NBG)[vN25, Ber37, Göd40] set theory and as such, Chiron has the same theoretical expressivity as the ZF(Zermelo-Fraenkel) [Zer08, Fra22] and NBG set theories. However, Chiron has a much higher level of practical expressivity than traditional logics. One of the most notable features of Chiron is its mechanism for *quotation*. This mechanism in Chiron is inspired by the *quote* operator in the LISP programming language. Using quotation in Chiron, one can formalize the law of beta reduction *inside* Chiron. Beta reduction is very important for all logics, but however, reasoning about beta reduction is usually done in the meta-logic of the logic, instead of the logic itself.

Isabelle/HOL[NPW02]and Coq[BC04] are interactive theorem provers. They both use reflection.

Maude[CELM96] is a high-performance reflective language and system supporting both equational and rewriting logic specification and programming for a wide range of applications. It supports in a systematic and efficient way logical reflection. This makes Maude remarkably extensible and powerful, supports an extensible algebra of module composition operations, and allows many advanced metaprogramming and metalanguage applications.

References

[BC04] Y. Bertot and P. Castéran. *Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[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`.

[Ber37] P. Bernays, 1937. Seven papers between 1937 and 1954 in the Journal of Symbolic Logic.

[CELM96] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic*, volume 4, pages 65–89, 1996.

[Far07] W. Farmer. Biform Theories in Chiron. In M. Kauers, M. Kerber, R. Miner, and W. Windsteiger, editors, *Towards Mechanized Mathematical Assistants*, volume 4573 of *Lecture Notes in Computer Science*, pages 66–79. Springer, 2007.

[Fra22] A. Fraenkel. The notion of 'definite' and the independence of the axiom of choice. 1922.

[Göd40] K. Gödel. The Consistency of Continuum Hypothesis. *Annals of Mathematics Studies*, 3:33–101, 1940.

[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.

[IR10] M. Iancu and F. Rabe. Formalizing Foundations of Mathematics. Under review, see `http://kwarc.info/frabe/Research/IR_foundations_10.pdf`, 2010.

[NPW02] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, 2002.

[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.

[RS09] F. Rabe and C. Schürmann. A Practical Module System for LF. In J. Cheney and A. Felty, editors, *Proceedings of the Workshop on Logical Frameworks: Meta-Theory and Practice (LFMTP)*, pages 40–48. ACM Press, 2009.

[vN25] J. von Neumann. Eine Axiomatisierung der Mengenlehre. *Journal für die reine und angewandte Mathematik*, 154:219–240, 1925.

[Zam11a] Zamdzhiev. OpenMath Twelf Encodings, 2011. `https://svn.kwarc.info/repos/twelf/set_theories/zfc/openmath.elf`.

[Zam11b] V. Zamdzhiev. Quotation in Formalized Mathematics. Guided Research Mathematics I Final Report, 2011.

[Zer08] E. Zermelo. Untersuchungen ÃŒber die Grundlagen der Mengenlehre I. *Mathematische Annalen*, 65:261–281, 1908. English title: Investigations in the foundations of set theory I.