Modular Formalization of Type Theory

Iulia Ignatov
Supervisors: Florian Rabe, Michael Kohlhase
Jacobs Universtiy Bremen

February 14, 2012

Abstract

Type theory is an important area of Mathematics, foundation of many programming languages and it has better computational behavior than the same mathematics based on theory of sets. Towards exploiting the practical computational capabilities of type theories, they are formalized using logical frameworks. However, these formalizations are often ad-hoc or do not relate different type theories to each other - in particular, many type theories can be seen as a combination of certain atomic features; this leads to duplicate formalizations and precludes the reuse of meta theorems. In the LATIN project, a similar problem has been solved in the realm of logics and languages for formulas and mathematics, but this method has not yet been applied systematically to type theories. We thus propose to apply the method of modular formalizations to type theories. We expect that this way, we will formalize individual atomic features separately and will be able to recover specific type theories as combinations of features. In particular, meta results can be established under per-feature basis, and then compose them into making results about individual type theories. As well, we can create interfaces between type theory and other formalized mathematics by providing only the views emergent from the base features.

1 Introduction

The last century saw an increasing expansion of mathematical knowledge, which lead to its formalization using diverse logical frameworks. With the development of computer science several systems were developed with the purpose of allowing the representation of mathematical knowledge and mathematical proofs in a computer environment so that a program can check their correctness or find proofs automatically. Some of these systems (examples are [Pau94], [BC04], [CAB$^+$86]) were quite successful and there is a large amount of theorems that were formalized in such a way.

Twelf is one such system, within which several mathematical foundations have ben specified, as well as an important fragment of logics. The purpose of this thesis is to expand the existing library with the theories of types formulated by Church and Curry respectively.

This proposal is organized as follows. Subection 2.1 gives a background overview on the theory of types, including a short history of its appearance. Further on, section 2.2 treats the subject of logical frameworks and intorduces Twelf, the type system of our focus. Finally for the introductory chapter, section 2.3 offers an outline of the LATIN project. Section 3 describes the intended work, starting with 3.1, the construction of the base diagram of type theory, continuing with 3.2 and 3.3, which builds the type theories on top of the already described base signatures, and ending with the translation of type theories into the foundation based on Zermelo-Fraenkel theory. The proposal ends with a description of the proposed deliverables in 4 and with further motivation, present in 5.

2 Preliminaries

2.1 Type Theory

The development of set theories of the 20th century has lead to the discovery of the paradox enunciated by Russel in 1901 in [Rus01]. Starting from the ambiguity of a predicate is predicated of itself, he saw the necessity of the theory of types, based on what he called extensional hierarchy, which stands for the differentiation between objects, predicates, predicates of predicates etc. In type theory, the objects are situated on either of 2 levels, at the top standing the types and at the bottom, the terms, the connection between the 2 layers being that each term has a certain type; reversely, a type is defined as a collection of objects exhibiting a common structure. This concept stands as the bolster of type systems, within which the statements have the form a : A. From a logical point of view, a is the proof of the formula A; in the computational world, a is an algorithm in a certain programming language, and A is the specification of a.

There are 2 main styles of typing within type theory ([Sel08]): Church, in which each abstraction indicates the type of the term (this is often called domain-full style, or intrinsic type theory), and Curry, in which no type is given within the declaration of the term (also called domain-free or extrinsic type theory). The latter style uses typing rules to make the assignment of terms to types and make judgements over typing.

The Church approach, as described in [Chu40], assembles typed objects from typed components; i.e., the objects are constructed together with their types. We use type annotations to indicate the types of basic objects: if N denotes the type of natural numbers, then $\lambda x : N.x$ denotes the identity function on N, having type $N \to N$. Each typed term has exactly one type.

In the Curry approach, types are assigned to existing untyped objects, using typing rules that refer to the structure of the objects in question. In this system, it is possible that an object has no type or it has more than one type. For example, the identity function $\lambda x.x$, has type $N \to N$, but also type $(N \to N) \to (N \to N)$.

Some of the basic features which construct the type theory are further described.

Disjoint Union. A term of type $A \uplus B$ is either of type A or of type B, together with an indication whether it belongs to type A or B.

Products. The type $A \times B$ returns the type containing all possible pairs of elements, the first element having the first type, A, and the second element, the second type, B.

Option Types. The option type is a polymorphic type which encapsulates an optional value. More specifically, it offers the possibility to use a term of the original type or the empty constructor, called $None$ or $Nothing$.

Partial Functions. As the name suggests, the type $A \mapsto B$ is the type of partial functions from A to B. In a similar fashion, the type $A \to B$ of total functions consints of functional terms which applied to a term of type A returns a term of type B.

Image types. From an intuitive point of view, the image types converts the image of a function into a type: given a function F, the type formed by the image of F is a subtype of the codomain of F.

Predicate Types. As well as for the function image types, given a predicate, the predicate type is a subset of the predicate domain, consisting of terms which satisfy the predicate.

Big Union. From a type theoretical perspective, the type $\bigcup F$ reasons about functions which take as argument a term and return a type. More precisely, let F be a function that takes as argument a term of type A and returns the type B; if a term x has the type B, it subsequently has type $\bigcup F$.

Big Intersection. Big intersection follows the point of view described above for big union: consider F to be a function that takes as argument a term of type A and returns the type B; if a term x has the type $\bigcap F$, it follows that x has type B.

Dependent Products. A term x of type $A \times B$ consists of a pair (a, b), where a is in type A and b is of type $B(a)$.

Dependent Functions. A term x of type $A \to B$ is a function $\lambda x.b$, where for all a of type A, b has type $B(a)$.

These and other properties can be combined to express other features and theorems within type theory; for simplicity however, we will remain on the description of this small fragment of features.

2.2 LF and Twelf

LF [HHP93] has been designed as a meta-logical framework to represent logics, and has become a standard tool for studying properties of logics. It is based on first order dependent type theory corresponding to the corner of the lambda cube which extends simple typed theory with dependent types.

Following the Curry-Howard correspondence [How80], LF represents all judgments as types and proofs as terms. To represent a proof theory in LF, appropriate type constructors have to be declared for the desired judgments. For example, for FOL, we need a judgment for the truth of propositions; for this, we first need the type for sentences, and then a judgement type:

```
prop : type.
ded : prop → type.
```

ded is called a type family: it is not a type itself, only after applying it true to a formula, it returns a type. For example, if a has type *prop*, *ded* a is a type. Thus, types may depend on terms, hence we speak of dependent types. All proofs are represented as terms. If a proof p proves the judgement J, then LF represents this as $p : J$ ([SP96]). With this intuition, we can already represent theories in LF and reason about them: we add constants to represent axioms.

The entities of LF are ordered on 3 levels:

- the top one, the level of kinds, are used to classify types; in particular, the kind *type* classifies the types
- the level of types and family of types represent syntactic classes, judgement forms or assertion forms
- the base level consists of objects, standing for syntactic entities, proofs, or inference rules.

Therefore, the objects are categorized by type families, classified on their turn by kinds.

Twelf [PS99] is an implementation of the LF framework and its module system [RS09b], based on signatures and signature morphisms (called views) [RS09a]. Within the signatures, all the declaration and definitions are made. In the Twelf module system, the views translate the symbols declared in the first signature into expressions of the second signature, preserving the typing; the axioms and

inference rules from the first signature are mapped, via the view, to proofs or derived inference rules, respectively. Their specification is done in the following manner:

```
%sig S = {Σ}
%view v : S → S' = {σ}
```

A specific syntax for Twelf is also held by the usual binding operators lambda and Pi, used in the following form:

- $\Pi x : A\ B(x)$ is declared as $x\ :\ A\ B\ x$
- $\lambda x : A\ t(x)$ is declared as $[x]\ t\ x$

Following the Twelf module system based on theory morphisms and LF entity hierarchy, we can specify the used grammar of LF (the following grammar only represents the fragment of Twelf which will serve to later use):

Signatures	Σ	::=	. | Σ, sig T = σ | Σ, view v : S → T = σ | Σ, include S | Σ, struct s : S = σ | Σ, c : A[= t] | Σ, a : K[=A]
Instantiations	σ	::=	. | σ, c := t | σ, a : A
Kinds	K	::=	type | A → K
Type families	A	::=	A t | x : A A
Terms	t	::=	x | [x : A] t | tt

The method for representing the syntax of a language is inspired by Church and Martin-Löf. The general approach is to associate an LF type to each syntactic category, and to declare a constant corresponding to each expression-forming construct of the object language, in such a way that a bijective correspondence between expressions of the object language and canonical forms of a suitable type is established. As an example, we demonstrate below the syntax of FOL in Twelf:

```
%sig FOL = {
    i : type.
    prop : type.
    ded : prop → type.
    true : prop.
    false : prop.
    not : prop → prop.
    imp : prop → prop → prop.
    equiv : prop → prop → prop
          = [a][b] (a imp b) and (b imp a).
    and : prop → prop → prop.
    or : prop → prop → prop.
    forall : (i → prop) → prop.
    exists : (i → prop) → prop.
    eq : i → i → prop.
}.
```

Intuitively, in the above signature, i represents the type of individuals, *prop* is the type of propositions (*true* and $false$, declared within the signature) and *ded* A stand for proofs of A; the logical connectives and quantifiers have the usual meaning. For equivalence, the definition in provided in terms of implication and conjunction; in the shown definition, $[a][b]$ $(a\ imp\ b)\ and\ (b\ imp\ a)$, the arguments, a and b will have the type *prop*.

A view from S to T instantiates all the symbols from S with T-expressions, with the obligation to preserve the typing. As an example, we show the specification of two simple signatures, S and T, and provide the view σ from one to enother:

```
%sig S = {
    a : type.
    b : a → type.
```

```
    c : b.
}.

%sig T = {
    a' : type.
    b' : a' → type.
}.

view \sigma : S \rightarrow T = {
    a := a'.
    b := b'.
}.
```

2.3 LATIN

The Logic Atlas and Integrator [KMR09] is intended to provide the tools for interoperability between multiple logics, languages and proof systems. From a foundational perspective, the LATIN library is already equipped with formalizations of logics, including both proof and model theory [Rab09], and mathematical foundations, which hold as the bolster of the logics model theory.

The atlas of logics can serve as bolster for automated reasoning, mathematics and software engineering, and its growth brings along more expressivity and more power. The actual encodings count over 1000 theories and morphisms [CHK+11b]. The folder structure of the atlas spans logic encodings, including first order logic, modal logic, description logic; translations between the existent logics; foundations, which contains the formalization of Zermelo-Fraenkel set theory, as well as HOL and other mathematical foundations; type theory, which comprises the formalization of the lambda cube [CHK+11a]. Extensibility is a constituent of main focus within the LATIN project - new logics can be added easily, including the reuse of already formalized logic features.

Organizing the content of the library is of crucial importance, as well as a challenge in itself. For insuring a systematic approach and development, the adopted conceptual model for representing the library is that of a graph of mathematical relations, in which the theories are the nodes and the translations between them are the edges. More precisely, the nodes consist of the specified signatures and the edges are the existing views between the signatures. In particular, the formalized logics are represented as theories and the translations between logics, as theory morphisms. A representation of a logic consists of specifying its syntax, and both its proof and model theory [Rab10]. The semantics of a logic is given by providing views into a specified foundation. The most used foundation consists of set theory.

The current library contains a formalization of ZF [**?**] which, starting from the Zermelo-Fraenkel axioms and FOL, defines all notions of set theories, even the natural numbers. Further on, the encoding reconstructs the typing from sets by defining the type family $Elem : set \to type$, which raises a set to the level of types. Thus, declaring an object X which contains an element x of type A will be encoded as $X : Elem\ A$. Although this encoding formalizes the notion of types, it has as a primitive the type set of sets and the connective $\in$: $set \to set$. What we intend to have is a formalization of the type theory which follow the two main theories concerning types, the one of Church and the one proposed by Curry. Furthermore, by constructing morphisms between the exisent signatures of set theories and our specification of type theories, we will create an interface between typed and untyped reasoning.

3 Modular Formalization

3.1 Base theories

Our intention is to develop a mathematical foundation on the base of the two theories which define the Church and the Curry logic [Sel08], respectively. Our formalization of the type theory has at its base the signatures of Church and Curry respectively, which are additionally equipped with the type *prop* for formulas and the type family *ded* for the truth judgement over the formulas, in order to represent the type theoretical logic. The equality of terms and types have to be defined separately, as they represent entities of distinct levels. Universal and existential quantifiers, as well as logical connectives, are defined in separate signatures which respectively include the signatures shown below.

The signatures of the Church and Curry type theories are provided below, as well as the signature of universal quantifier, which serves as example:

```
%sig Church = {
   tp   : type.
   ttm  : tp → type.

   prop : type.
   ded  : prop → type.

   ==   : ttm A → ttm A → prop.
   =tp= : tp → tp → prop.
}.
```

In this signature, the types are represented by *tp* and the terms, by *ttm*, which take types as arguments: a term of type A would be written then *ttmA*. *prop* and *ded* are the logical constructors described above. == stands for equality between two terms and $= tp =$, for equality between types.

```
%sig Curry = {
   tp : type.
   tm : type.

   prop : type.
   ded  : prop      type.

   #   : tm → tp → prop.
   =tm= : tm → tm → prop.
   =tp= : tp → tp → prop.

   <   : tp → tp → prop.
   <I  : ({x} ded x # A → ded x # B) → ded A < B.
   <E  : ded A < B → {x} ded x # A → ded x # B.
}.
```

Again, *tp* stands for types; however, the terms, represented by *tm*, are well formed by themselves, without depending on types. For typing, we introduce the family #, which applied to term a and type A, stands for a has type A. We add to the base signature of Curry theory the operator for subtyping, $<$, and the introductory and elimination rule for it, $< I$ and $< E$ respectively. *porp*, *ded*, $= tm =$ (equality over terms) and $= tp =$ have the same purpose as for the Church signature.

```
%sig UniversalQuantifier = {
   %include Church.
   tforall  : (ttm A → prop) → prop.
   tforallI : ({a : ttm A} ded F a) → ded tforall F.
   tforallE : ded tforall F → {a : ttm A} ded F a.
}.
```

The universal quantifier is specified by extending the signature Church. The quantifier is specified with its introductory and elimination rules. The introduction rule, $tforallI$ bounds the free variable a in $(a : ttm\ Aded\ F\ a)$, and the eliminaton rule takes as argument a proof that for the predicate F, all terms make it true, and a term x of appropriate type, returning the proof that x satisfies F.

An important part of the development graph is the translation between Church and Curry theories. The relation between the two styles of typing is well known [**?**]; however, our encodings will only comprise a mapping from Churhc to Curry. The reason for this has at its base the formalization of subtyping. As shown, the Curry signature contains rules for subtyping. However, the formalization of subtyping in Church is not desirable because the encoding would loose the adequacy. Therefore, the Curry logic becomes more expressive than the Church logic - an aftermath is that only Church will be translated into Curry. For the other direction, the translation of Church into Curry, the signature of the extrinsic type theory is extended with declarations which describe the Church style, and therefore we can map the Church assertion into the respective declarations within Curry:

```
%sig Curry = {
   ...
   ttm       : tp → type.
   !         : {t:tm} ded t # A → ttm A.
   which     : ttm A → tm.
   why       : {t: ttm A} ded which t # A.
   eq_which  : ded which (X ! P) =tm= X.
}.
```

The declarations for Church style within the Curry signature are equipped with rules which adequately identify the Church types. ! takes as argument a term together with the proof that it belongs to a certain type, and returns the Church typed term; in the inverse direction, *which* only discards the type and returns the term taken as argument without its type. *why* and *eq_which* represent the proofs that the conversions made with ! and *which* maintain the same term. All declarations in the Church signature have their correspondent in the Curry signature with the same nomenclature. For example, *ttm* in Church signature has the same semantics as *ttm* in Curry signature. The view from Church to Curry becomes thus clear:

```
%view μ : Church → Curry = {
   tp  := tp.
   ttm := ttm.

   prop := prop.
   ded  := ded.

   ==   := ==.
   =tp= := =tp=.
}.
```

We are also interested in the translations of the emergent theories into the boolean logic, within which boolean is considered a type and the two truth values are terms of type boolean. The signature of the booleans have to include the notions of type and term, and therefore has to include the signature of a type theory:

```
%sig Bool = {
   %include Church.
   bool : tp.
   1 : ttm bool.
   0 : ttm bool.
}.
```

bool is the type of booleans, and the terms it comprises are 1 and 0. By translating the type theoretical logic into the boolean logic, we will obtain the interface between internal and external propositions.

```
%view ω : Church → Bool = {
    tp  := tp.
    ttm := ttm.

    prop := ttm bool.
    ded  := [x] ded x == 1.

    ==   := ==.
    =tp= := =tp=.
}.
```

Having these theories as primitives, as well as the views from Church to Curry type theory, and form Church to boolean logic, we also want to formalize the pushouts of the diagram created by the present signatures and the currently described views between them. The figure 1 presents the diagram which stands at the base of our encoding of type theory.

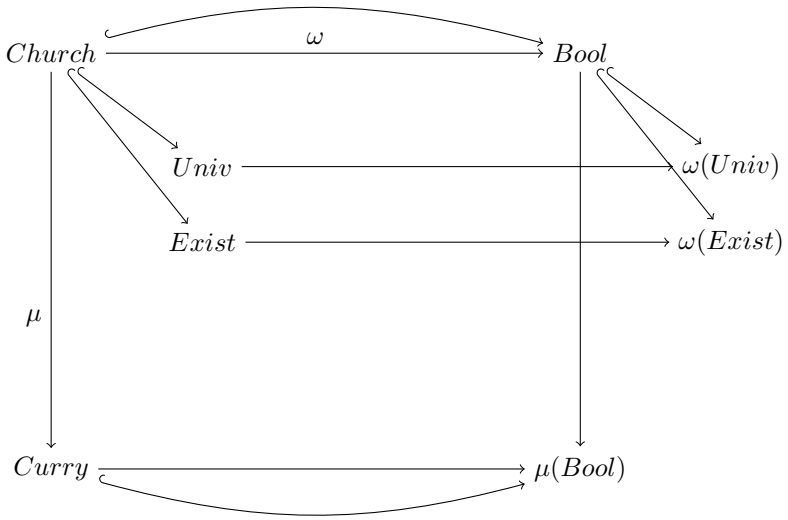

Figure 1: Base Type Theory Diagram

The arrow style as the one from *Church* to *Univ* portrays inclusion, whereas the one from *Church* to *Curry* is a morphism betewwn the two nodes.

3.2 Modular Type Theory

The above diagram will serve as the base structure on which the entire type theory will be built upon. Wherever possible, every signature based on type theory will make use of the signature of Church. This precept will be applied in the formalization of type theory features: wherever possible, we will declare the feature and its rules using the Church style, and only when more expressiveness is needed (for example, subtyping, which was not formalized for Church), base the feature upon Curry type theory. The reason for this is that within Church style, the terms are intrisically types, and therefore type checking is reduced to the one performed by LF; the Curry style however, reasons about typing using rules of inference, reducing the type checking to proofs, which renders the process slower. Thus, formalizing the type theory features described in Sect. 2.1, the diagram from figure 2 emerges.

In the specification of each feature, usually the operators are declared, as well as their introduction and elimination rules, which will adequately encode the feature. As an example, we provide the

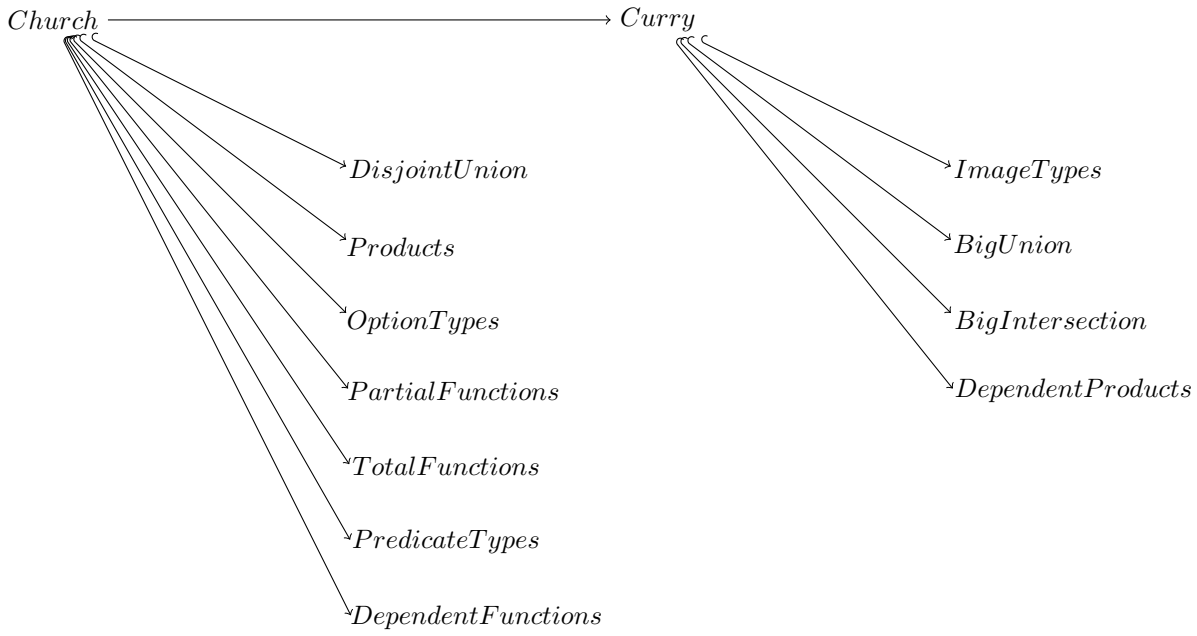

Figure 2: Type Theoretical Features

specifications of 2 of the features, one for each typing style, dependent functions (based on Church) and dependent products (based on Curry).

```
%sig DependentFunctions = {
   %include Church.
   pi : (ttm A → tp) → tp.
   Pi : (ttm A → tp) → type = [A][B] ttm (pi [x: ttm A] B x).
   lambda : ({a: ttm A} ttm (B a)) → pi [x: ttm A] B x..
   apply  : ttm pi [x: ttm A] B x → {a : ttm A} ttm (B a).
}.

%sig DependentProducts = {
   %include Curry.
   sig  : (ttm A → tp) → tp.
   Sig  : (ttm A → tp) → type = [A][B] ttm (depSum A B).

   pair : {a: ttm A} ttm (B a) → sig [x: ttm A] B x.
   pi1  : sig [x: ttm A] B x → ttm A.
   pi2  : {u: sig [x: ttm A] B x} ttm (B (pi1 u)).
}.
```

3.3 Composing and relating features

One of the purposes of having the modular encodings is being able to combine the individual features and obtain other (possibly more complicated) theorems and features.

A research question is the manner of constructing and correlating the content such that in the end, as many dependencies as possible are established and the resulting views between the signatures are established.

Therefore, the final graph of the specifications will additionally contain functors which will translate features between them. For example, the signature for partial functions can be expressed in terms of total functions and option types, which will induce a view from the signature PartialFunction to the signatures Functions and OptionTypes. However, the ability to construct the described views and obtain the desired graph also pertains to the implemented features of the Twelf system; analyzing the necessity of implementing other features within the type system will be subject of further work - taking the example above, we want to know if it is necessary for the system to support views to or from multiple signatures:

%view v : PartialFunctions → Functions + OptionTypes = { . . . }.

where the operator + composes the signatures Functions and OptionTypes, instead of declaring an additional signature to include the codomains of the view, which is the only fashion at the moment for declaring the desired view.

3.4 Modular semantics

A main advantage in the modular specification of type theory will be importing theorems from other mathematical foundations, by only formalizing the mapping of the basic features into the respective foundation. Particularly, it is desirable to have an interface between typed and untyped reasoning, as well as specifying the semantics of the type theoretical modules in terms of set theory.

Towards obtainig the mentioned interface, we provide the mapping from the type theories to the foundation based on Zermelo-Fraenkel theory, already formalized within the LATIN project. Furthermore, as bescribed by [**?**], the formalization comprises typed set theories, which will serve as the basis of our mapping. The diagram in figure 3 depicts a small sample of the mapping between the proposed type theoretical encodings and the typed set theoretical encodings - for the simplicity of the diagram, we will only show the morphim from one feature based on Church logic and one feature based on Curry logic, each having its correspondent in the ZF formalization.

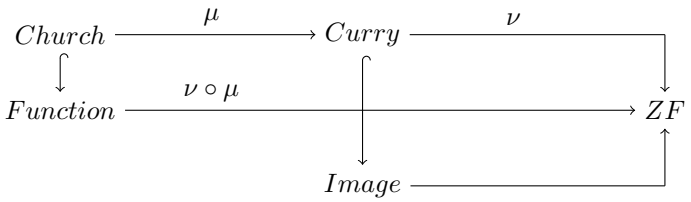

Figure 3: Mapping between Type Theory and Set Theory

4 Proposed work

The proposed work can be split into two deliverables.

For the first package, which consists of formalized type theories, the first research question which arises is what are the features that need to be specified - these will represent the nodes of the theory graph we will obtain in the end. Once the signatures are established, it remains to research how each of them can be specified with a minimum number of dependencies. As well, composing features can lead to other interesing results from a type theoretical perspective.

After specifying the nodes of the graph, the question of how can they relate follows. The second deliverable consists thus of establishing and formalizing views between the specified theories. This part can, however, pose questions about the first part of the proposed work: researching the question of views can bring the necessity of new signatures or change of the old ones. This second deliverable also includes the functor between type theories and the existing ZFC foundation.

5 Conclusion

Following the goal described above, we expect to formalize the type theories of Church and Curry, together with their features, making use of the modular system of Twelf. Additionally, beyond having a formalization of type theory, the main result consists of the graph of interlinked theories and the possibility to transfer theories and theorems from one foundation to another by only having morphisms between base features.

References

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

[CAB+86] R. Constable, S. Allen, H. Bromley, W. Cleaveland, J. Cremer, R. Harper, D. Howe, T. Knoblock, N. Mendler, P. Panangaden, J. Sasaki, and S. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, 1986.

[CHK+11a] 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.

[CHK+11b] M. Codescu, F. Horozal, M. Kohlhase, T. Mossakowski, F. Rabe, and K. Sojakova. Towards Logical Frameworks in the Heterogeneous Tool Set Hets. In H. Kreowski and T. Mossakowski, editors, *Recent Trends in Algebraic Development Techniques*, volume 7137 of *Lecture Notes in Computer Science*. Springer, 2011.

[Chu40] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5(1):56–68, 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.

[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.

[KMR09] M. Kohlhase, T. Mossakowski, and F. Rabe. The LATIN Project, 2009. see `https://trac.omdoc.org/LATIN/`.

[Pau94] L. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

[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.

[Rab09] F. Rabe. Representing Logics and Logic Translations. In D. Wagner et al., editor, *Ausgezeichnete Informatikdissertationen 2008*, volume D-9 of *Lecture Notes in Informatics*, pages 201–210. Gesellschaft für Informatik e.V. (GI), 2009. English title: Outstanding Dissertations in Computer Science 2008.

[Rab10] F. Rabe. A Logical Framework Combining Model and Proof Theory. see `http://kwarc.info/frabe/Research/rabe_combining_09.pdf`, 2010.

[RS09a] F. Rabe and C. Schürmann. A Module System for Twelf, 2009. see `https://cvs.concert.cs.cmu.edu/twelf/branches/twelf-mod`.

[RS09b] 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)*, volume LFMTP'09 of *ACM International Conference Proceeding Series*, pages 40–48. ACM Press, 2009.

[Rus01] B. Russll. Recent work in the philosophy of mathematics. *International Monthly*, 1901.

[Sel08] J. P. Seldin. The logic of church and curry. 2008.

[SP96] C. Schürmann and F. Pfenning. Automated theorem proving in a simple meta-logic for LF. In C. Kirchner and H. Kirchner, editors, *Proceedings of the 15th International Conference on Automated Deduction*, pages 286–300. Springer, 1996.