Jacobs University Bremen

School of Engineering and Science

Modular Encoding of Type Theory

BSc Thesis in Computer Science

Iulia Ignatov

Supervisors:

Michael Kohlhase

Florian Rabe

May 31, 2012

Acknowledgements

Many thanks to Prof. Michael Kohlhase and Dr. Florian Rabe, for being my mentors and offering their guidance and support in research, but also in life prospects.

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.

Contents

1 Introduction **1**

2 Preliminaries **1**
2.1 Type Theory 1
2.2 LF and Twelf 2
2.3 LATIN 4

3 Modular formalization **5**
3.1 Base theories 5
3.1.1 Theories with no terms or untyped terms 5
3.1.2 Overview of basic type theory base modules. Classification 6
3.1.3 Church and Curry typing 10
3.1.4 Church typed terms 11
3.1.5 Curry typed terms 14
3.2 Modular Type Theory 15

4 Composing and relating features **17**

5 Conclusions and Further work **18**

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 document 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 formalization of type theories, starting with 3.1, the construction of the base diagram of type theory, continuing with 3.2, which builds the type theory features on top of the already described base signatures. The document then continues with an example of modular composition in 4. The document ends with a description of further work in and the description of our work with the translation of type theories into the foundation based on Zermelo-Fraenkel theory 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.                          %prefix 0 ded.
     true    : prop.
     false   : prop.
     not     : prop → prop.                          %prefix 75 not.
     imp     : prop → prop → prop.                   %infix left 50 imp.
     equiv   : prop → prop → prop
             = [a][b] (a imp b) and (b imp a).       %infix right 50 equiv.
     and     : prop → prop → prop.                   %infix left 50 and.
     or      : prop → prop → prop.                   %infix left 50 or.
     forall  : (i → prop) → prop.
     exists  : (i → prop) → prop.
     eq      : i → i → prop.                         %infix left 25 eq.
}.
```

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 : type.
}.

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

view σ : S → T = {
     a := a'.
     b := b'.
     c := [x : a'] b' x.
}.
```

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

3.1.1 Theories with no terms or untyped terms

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. Within the formalization of a mathematical library, it is desirable to break the formalized concepts into simplest ones, and then construct the more complicated ones from those which stand at their basis. As well, the mathematical structure of a library must preserve soundness in the first place, consistency, and agreement of notation. These are facilitated by reusing the already existent, instead of redeclaring theories and making views between them to specify the morphisms. Take equality as an example: the equality of types has the same properties as the equality of terms, among which symmetry. Hence, it will be (possible and) convenient to recover the type equality from the term equality, because in this manner, the symmetry property will not have to be redeclared: the type system will make the pushout and thus, automatically recover it.

In the formalization of mathematical foundations, we leave thus from the most simple theories, which, from the point of view of terms hierarchy, are the ones that do not comprise the concept of term. These theories are best exemplified by propositional logic, which only deals with propositions (with no bound variables):

```
%sig Prop = {
     o   : type.
     ded : o → type.   %prefix 0 ded.
}.
```

In the signature above, the type o is the type of propositions, and ded is the judgment type, which reasons over propositions. The theories containing untyped terms follow immediately, the most basic one being the one that only contains the declaration of the term type (in the signature, i).

```
%sig Univ = {
     i : type.
}.
```

Following the previous example, the corresponding signature is the one which formalizes the logic of universal type, and it is only an inclusion of the signatures Prop and Univ. On this signature, the formalization of the first order logic is built.

```
%sig UnivLogic = {
     %include Prop.
     %include Univ.
}.
```

3.1.2 Overview of basic type theory base modules. Classification

Finally to our interest, the next group of theories involve types and typed terms, formalizing therefore the type theory. These theories conceptually bifurcate into those which are based on the Church style, and the ones which illustrate the Curry style.

Hierarchy of Terms Dimension	
No terms	The theories will not use the concept of terms. The propositional logic is such a theory.
Untyped terms	Only the universal type is involved, i. The first order logic incorporates untyped terms.
Church style typed terms	Every term has a type, and its typing is done intrinsically, in the typed lambda calculus style. It does not allow for subtyping.
Curry style typed terms	The terms are typed extrinsically. The typing connective reasons about one type having a certain type. In some theorems, the terms have to be equipped with the proof that it belongs to a certain type. Subtyping can easily be formalized.

We have seen that from the point of view of terms hierarchy, the basic modules split into 4 categories: no terms, untyped terms, Church style typed terms and Curry style typed terms. Following the line of identifying the most basic constructs and formalizing them separately, looking from the point of view of how the propositions are formalized and how the judgment over them is made, the extrinsic logic and intrinsic logic are distinguished, to which the classification of the theorem which require no logic adds, as well as those which require both kinds of logic. The internal logic invokes the concept of typed terms and it can be formalized by declaring the type of propositions (in our case, this type will be called bool), and the propositions as terms of this type. On the other hand, the external logic does not depend on rest of the structure of the signatures it dwells in: the formalization of this logic only needs the declarations contained in the signature Prop, as it is previously shown.

Logic Dimension	
No Logic	No logic involved.
External Logic	The base declarations for the external logic are the proposition type and the connective which reasons about sentences.
Internal Logic	Requires the concept of type: the proposions are represented as terms of the boolean type.
External + Internal Logic	Inclusion of both kinds of logics. As well, it incorporates the proper conversions between external and internal logic.

The equality is also a reference point in dividing the modules: the equality between two terms can be, first of all, a proposition and thus, it can be reasoned about or it can be applied logical connectives. This is the extensional equality, meaning that the equality is determined by the pairs of points which are equal. The signatures which require extensional equality must have a sufficient structure before being equipped with the external logic, that is, the basic logical constructs (proposition type and the judgment connective). Following Martin-Löf type theory, the equality can also be seen as a type (also called identity type), which has the main advantage that it is decidable, unlike the extensional one. This kind of equality is called intensional. As for the logics classification, the modules can involve extensional equality, intensional equality, neither or can encompass both.

Equality Dimension	
No Equality	No equality involved.
Intensional Equality	The equality between terms or between types is a proof that the respective terms or types are identical.
Extensional Equality	The equality is a proposition (and hence, it requires the logic constructs).
Extensional + Intensional Equality	Comprises both externsional and intensional equality, together with the conversions between the them.

The table below depicts the described structure of the base modules; the nomenclature of a modules mainly represents the concatenation of what the represented theory encompasses. For facilitating the understanding of the 3D table (4 x 4 x 4), the dimension of equality is projected and depicted by colors (the brighter, the earlier the entry in table), as well as the used order in each cell: no equality, intensional equality, extensional equality and extensional+intensional equality.

The nomenclature of the signatures uses the CamelCase standard: Inteq is inensional equality; Exteq and Equality represent the extensional equality; Intlog is the internal logic; Extlog and Logic represent the external logic. Some of the cells in the table are empty because they pertain to the coordinates in the three dimension which have clashing requirements. For example, the signature of internal logic needs the declaration of types. Hence, it cannot be formalized for the theories which only have untyped terms.

	No Universe	Untyped terms	Church types	Curry types
no logic		Univ	Church	Curry
		Inteq	ChurchInteq	CurryInteq
logic external	Prop	UnivLogic	ChurchLogic	CurryLogic
		LogicInteq	ChurchLogicInteq	CurryLogicInteq
		Equality	ChurchEquality	CurryEquality
		LogicInteqExteq	ChurchInteqExteq	CurryInteqExteq
internal logic			ChurchIntlog	CurryIntlog
			ChurchIntlogInteq	CurryIntlogInteq
			ChurchIntlogEquality	CurryIntlogEquality
			ChurchIntlogInteqExteq	CurryIntlogInteqExteq
external + internal logic			ChurchIntlogExtlog	CurryIntlogExtlog
			ChurchIntlogExtlogInteq	CurryIntlogExtlogInteq
			ChurchIntlogExtlogEquality	CurryIntlogExtlogEquality
			ChurchIntlogExtlogInteqExteq	CurryIntlogExtlogInteqExteq

As a rule for the order in which the classifications are set in the table, the lower or more to the right the theory stays in the table, the more complex the theory is. Hence, whenever a feature which has already been formalized is needed for another theory, the uppermost, leftmost and lighter theory which includes the needed feature will be used.

Since in the Church/Curry type theory modules, we will want to recover features from theories based on untyped terms, as well as the ones without terms, it is highly important to formalize and depict these signatures adequately. First of all, the signatures for Prop, Univ, UnivLogic are previously shown. Inteq includes the signature Univ and adds the equality type, together with the rules which define the semantics of the equality. Equality on the other hand, formalizes the extensional equality for untyped terms. The specification of Zermello-Fraenkel set theory is built on top of this theory. Inteq is the pushout of Inteq and UnivLogic. Finally, ExteqInteq is the pushout of Equality and Inteq.

```
%sig Inteq = {
     %include Univ.
     ==           : i → i → type.                    %infix left 50 ==.
     reflexivity  : {X} X == X.
     symmetry     : X == Y → Y == X.
     transitivity : X == Y → Y == Z → X == Z.
}.
```

3.1.3 Church and Curry typing

Crossing towards type theory, the main signatures, which formalize the types and typed terms (and the relation between them), are Church and Curry:

```
%sig Church = {
     %struct tps : Univ = {} %open i %as tp.
     tm : tp → type.                                %prefix 10 tm.
}.

%sig Curry = {
     %struct tps : Univ = {} %open i %as tp.
     %struct tps : Univ = {} %open i %as tm.
     #  : tm → tp → type.                           %infix left 25 #.

     <  : tp → tp → type.                           %infix left 25 <.
     <I : ({x} x # A → x # B) → A < B.
     <E : A < B → {x} x # A → x # B.
}.
```

As depicted by the signatures, the Church and Curry signatures reuse the untyped term by declaring the type tp (representing types in type theory) as the corresponding of the untyped term in the Univ signature. This is possible since the two concepts follow the same traits (their types coincide) and have same properties (in this case, reflexivity, symmetry, transitivity and congP). Furthermore, within the Curry signature, it is possible to recover the type tm (representing terms in type theory) from the untyped term, since the typing is done extrinsically, by the typing connective, #, which is the proof that a certain term belongs to a certain type: $x\#A$ means that the term x belongs to type A. The Curry signature also contains the subtyping connective, $<$ ($A < B$ formalizes the fact that A is a subtype of B), while in the Church signature, it is ommitted.

This explains the fact that although they both depict typing styles, the Church style is placed intentionally in the table before the Curry style. In the Twelf specification of Church, the formalization of subtyping would be awkward and would even loose adequacy, and is thus skipped. The reason why this is happening is because the terms are intrinsically typed, thus a term can have only one type. On the other side, in the Curry style, the typing is extrinsic, and therefore the terms can have multiple types, condition necessary for the existence of subtypes: if the term x is of type A, and A is a subtype

of B, x will also be of type B. Therefore, the Church signature will only contain the declarations for types and for terms, whereas the Curry signature will comprise, along with the declarations for types, terms, and typing, the subtyping combinator.

Even if the Curry based theory is more powerful than the Church based one, there is no view from Church to Curry yet. There is a possibility to specify it in the current type system, using the following described trick. The signature of Curry would be extended with declarations which describe the Church style, and with rules and declarations which make the conversion between extrinsic and intrinsic typing for a term. The signature of Curry would then look in the following manner:

```
%sig Curry = {
     %struct tps : Univ = {} %open i %as tp.
     %struct tps : Univ = {} %open i %as tm.

     #  : tm → tp → type.                    %infix left 25 #.

     <  : tp → tp → type.                    %infix left 25 <.
     <I : ({x} x # A → x # B) → A < B.
     <E : A < B → {x} x # A → x # B.

     ttm   : tp → type.
     !     : {t:tm} ded t # A → ttm A.
     which : ttm A → tm.
     why   : {t: ttm A} ded which t # A.
}.
```

The declarations for Church style within the Curry signature would be equipped with rules which would adequately identify the Church types. ! would take as argument a term together with the proof that it belongs to a certain type, and it would return the Church typed term; in the inverse direction, *which* would only discard the type and return the term taken as argument without its type. *why* would represent the proofs that the conversions made with ! and *which* maintain the same term. For other features and axioms of the Church terms, such as equality (== : tm A -¿ tm A -¿type), a corresponding feature or axiom, respectively, would have to be declared in the Curry signature (=ttm= : ttm A -¿ ttm A -¿ type). The view from Church to Curry would become thus clear:

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

However, this trick would defy the purpose of the library, and would reduce the principle of only including the necessary constructs into the formalization of a theory, since it would add the (unnecessary and non-belonging) Church typing style to the Curry style theory. This view remains thus to in the realm of future work, namely, when patterns will be introduced within our type theory - the declarations within the signatures will remain the same, and the patterns will be added; the mapping of the patterns will complete the view between the Church and the Curry style.

3.1.4 Church typed terms

Since the type declaration is recovered in Church from the untyped term from Univ, the equality of types will also be recovered from the equality of untyped terms. Hence, the equality of types, =tp= in the signature ChurchInteq shown below, will be recovered from a structure of type Inteq (remember that the signature Inteq includes Univ):

```
%sig  ChurchInteq = {
```

```
    %include Church.
    %struct tps : Inteq = {
        %include tps.
    } %open == %as =tp=.
    == : tm A → tm A → type.
}.
```

This signature results in the digram from Figure 1.

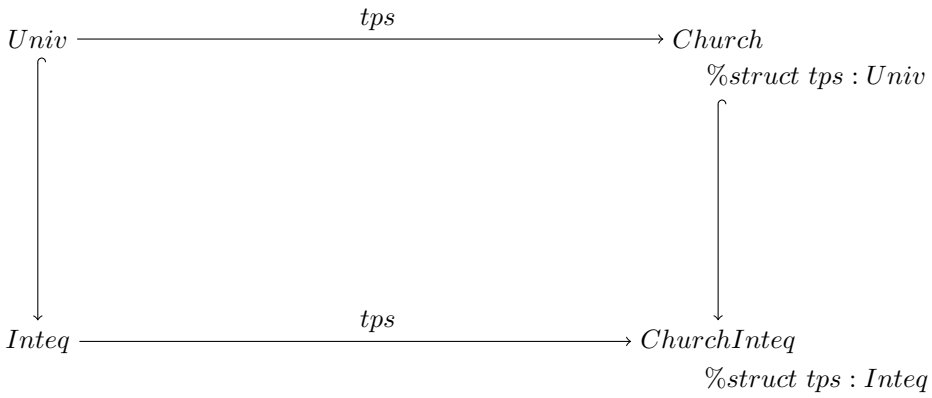

Figure 1: ChurchInteq resulting diagram

Moving to the external logic part, the signature ChurchLogic is formed by adding the base declarations of propositional logic to the Church signature:

```
%sig ChurchLogic = {
    %include ChurchTypes.
    %include Prop.
}.
```

The same as intensional equality between types was recovered from intensional equality between untyped terms, the extensional equality between types will be recovered from the signature Equality:

```
%sig ChurchEquality = {
    %include ChurchLogic.
    %struct stp : Equality = {
        %include stp.
    } %open == %as =tp=.
    == : tm A → tm A → prop.
}.
```

The signature ChurchLogicInteq is again, a simple inclusion of the signatures ChurchInteq and ChurchLogic (the Twelf system will merge the common parts, in this case, Church).

ChurchInteqExteq will comprise both sorts of equality. However, the inclusion of the respective signatures has to be made in such way that the declarations == and = tp = of internal logic do not overlap with the declarations == and = tp = of external logic. This can be easily achieved by renaming:

```
%sig ChurchInteqExteq = {
    %include ChurchEquality.
    %include ChurchInteq %open =tp= as =tp='
                                 ==    as  ==='.
}.
```

The internal logic is formalized by declaring the type bool and the judgment connective, needed in order to be able to reason about propositions, which are in this case of type bool.

```
%sig ChurchIntlog = {
     %include Church.
     bool : tp.
     ded' : tm bool → type.
}.
```

The signature of the internal logic differs from the one of external logic by the fact that in internal logic, propositions are represented as a type within the formalized type theory. As expected, the other 3 signatures based on internal logic, ChurchIntlogEquality, ChurchIntlogInteq and ChurchIntlogExtlog, follow the pattern of external logic. However, the extensional equality for types will not be recovered from untyped terms, because 1. extensional equality is a propostion and 2. internal logic is not formalized for untyped terms.

```
%sig ChurchIntlogEquality = {
     %include ChurchIntlog.
     =tp= : tp → tp → tm bool.
     ==   : tm A → tm A → tm bool.
}.

%sig ChurchIntlogInteq = {
     %include ChurchIntlog.
     %include ChurchInteq.
}.

%sig ChurchIntlogInteqExteq = {
     %include ChurchIntlogEquality.
     %include ChurchIntlogInteq %open =tp= as =tp='
                                     ==    as  ==='.
}.
```

In the case of signatures which contain both internal and external logic, they firstly include the corresponding logics from Church with internal logic and Church with external logics, respectively. We formalize the translation between internal and external logic by adding the reflect and lift connectives, together with the axioms which offers them meaning. In order to reuse these conversions instead of redeclaring them, the other theories with internal and external logic will include ChurchIntlogExtlog.

```
%sig ChurchIntlogExtlog = {
     %include ChurchIntlog.
     %include ChurchLogic.

     reflect      : tm bool → prop.
     conv_reflect : {x : tm bool} ded' x → ded (reflect x).

     lift         : prop → tm bool.
     conv_lift    : {x : prop} ded x → ded' (lift x).
}.

%sig ChurchIntlogExtlogEq = {
     %include ChurchIntlogEquality.
     %include ChurchEquality.
     %include ChurchIntlogExtlog.
}.

%sig ChurchIntlogExtlogInteq = {
     %include ChurchIntlogInteq.
     %include ChurchLogicInteq.
     %include ChurchIntlogExtlog.
}.
```

```
%sig ChurchIntlogExtlogInteqExteq = {
     %include ChurchIntlogInteqExteq.
     %include ChurchInteqExteq.
     %include ChurchIntlogExtlog.
}.
```

In our classification of signatures, we have distingueshed between internal and external logic. However, they represent the same concept. This fact is formalized by a view between the two types of formalization. Since the internal logic requires more structure than the external logic (they require the notion of type), only the view from external to internal logic.

```
%view ChurchLogicView : ChurchLogic → ChurchIntlog = {
     prop := tm bool.
     ded  := ded.
}.
```

The same consideration applies to the classification of equality: a view from intensional to extensional equality is provided. The inverse view is not formalized because extensional equality requires, in addition to what the intentional one engages, the base logic declarations.

```
%view ChurchEqualityView : ChurchInteq → ChurchEquality = {
     =tp= := [A : tp][B : tp] ded A =tp= B.
     ==   := [A : tp][a : tm A][b : tm A] ded a == b.
}.
```

3.1.5 Curry typed terms

The signatures based on the Curry style typing follow the same structure as the ones based on Church typing. For the Curry base modules, we will only show the signature of CurryInteq and CurryIntlog. The explanations for the other signatures can be easily determined following the explanations for the corresponding Church signatures.

CurryInteq follows in principle the same structure as ChurchInteq, only that in this case, the signature will contain 2 structures instead of one. The further declarations are the axioms which identify the behavior of the typing connective under equality of types and terms, and the behavior of the subtyping under the equality of types.

```
%sig CurryInteq  = {
     %include Curry.
     %struct tps : Inteq = {
          %include tps.
     } %open == %as =tp=.
     %struct tps : Inteq = {
          %include tps.
     } %open ==.

     #tpEq : X # A → A =tp= B → X # B.
     #tmEq : X # A → X == Y → Y # A.

     <eql  : A < B → A =tp= C → C < B.
     <eqr  : A < B → B =tp= C → A < C.
}.
```

This signature results in the diagram shown in Figure 2.

The main disadvantage of the Curry typed terms is the fact that in most of the theorems or even in the judgments over propositions, the terms have to be equipped with the proof that they belong to a certain type. For instance, within the internal logic, the judgement connective takes as argument, apart from the propositional term, the proof that the term actually is of the propostions (bool) type.

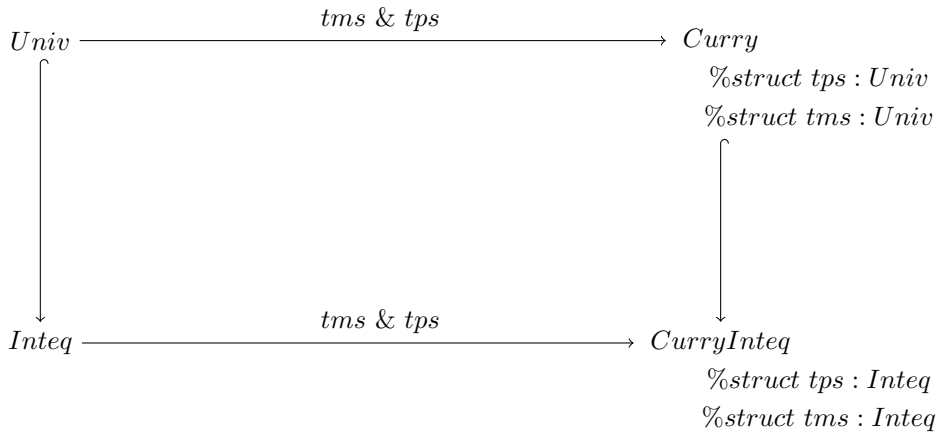

Figure 2: CurryInteq resulting diagram

```
%sig CurryIntlog = {
     %include Curry.
     bool : type.
     ded' : {p : tm} p # bool → type.
}.
```

The resulting diagram of the base modules (the final diagram will also contain the views between Church and Curry based signatures) is shown in Figure 3. The figure only depicts the structure for one of the Church typing and Curry typing, and hence, instead of writing Church or Curry, we named with C in the diagram. A complete diagram is simply the duplicate of the shown one.

3.2 Modular Type Theory

There are numerous type theories, but all of them are based on the same concept (the hierarchical structure of types and terms) and they all have similar features, and only minor differences. Each one of these features can be formalized separately, and thus, many of these theories are comprised by our formalization - a theory will merely be a union of features. For example, Martin-Löf's type theory will be composed of binary product types, unit type, dependent product types, function types, binary sum, empty, dependent sum, finite types, propositions as types, equality, finite product, unit, dependent product, function, finite sum, dependent sum. It is thus desirable to have the type theoretical side of the library as expanded (with as many features formalized) as possible; a great advantage of the modularity is that whenever needed, a new feature can be added fairly easily.

The table 1. shown in section 3.1. serves as the base structure on which the entire type theory is be built upon. As previously mentioned, the features will be based on the Church signatures wherever possible, since we are keeping the theories in a minimal state of declaration inclusion, and Curry is more powerful than Church by the fact that it incorporates subtyping. Therefore, only the features which need subtyping will have as the base inclusion the Curry signature, and all of the others, the Church signature. The views from Church-based signatures to Curry-based ones defy the purpose of this document (their specification will be done once the use of patterns will be introduced for our type theories).

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. To exemplify the formalization of the features, we have provided the signatures of dependent functions (based on Church) and dependent

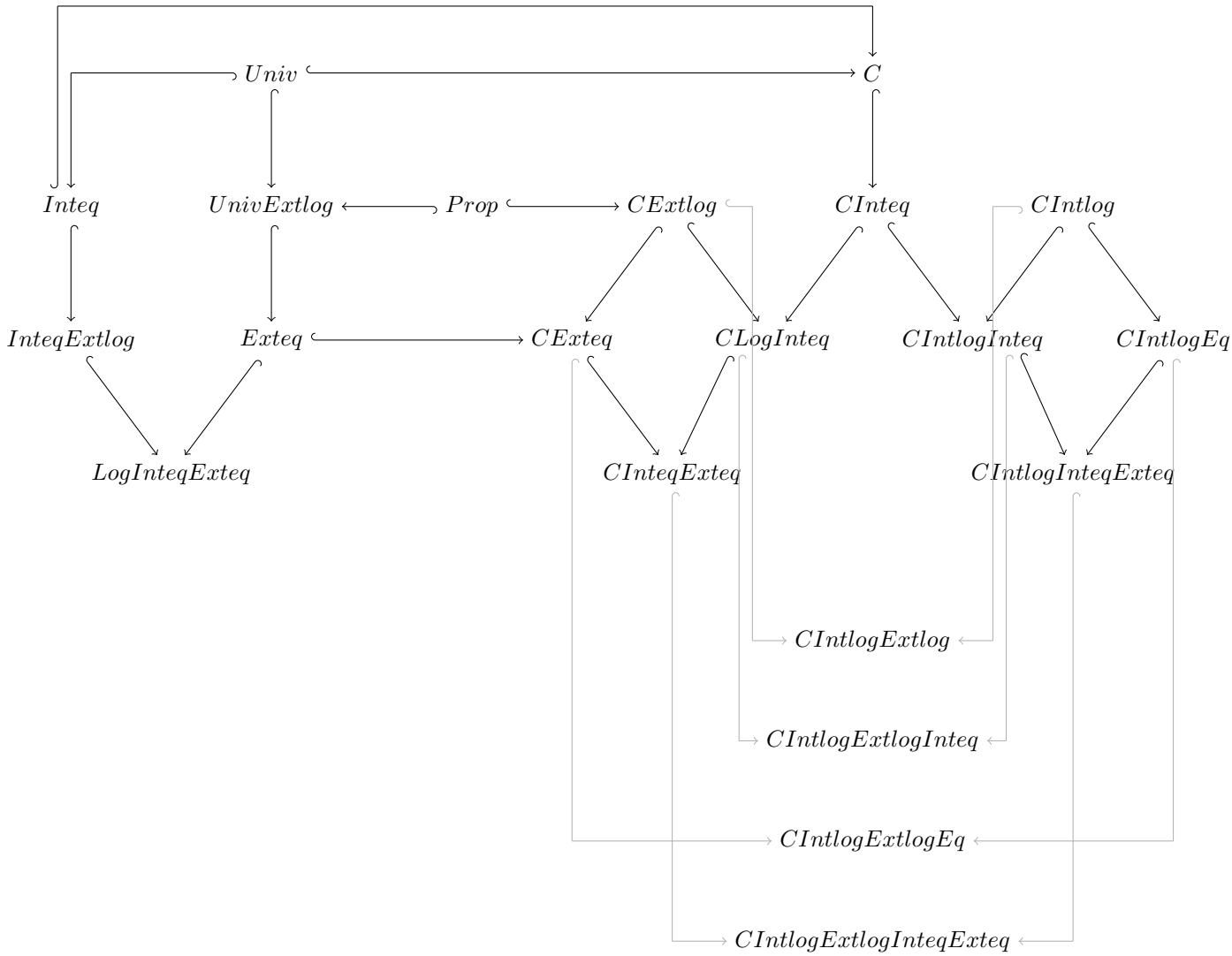

Figure 3: Diagram of type theoretical base modules

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)).
}.
```

4 Composing and relating features

Another great advantage of modularity lays in the fact that the features can be easily composed to recover other theories: a signature for one such theory will only represent an inclusion of the features it comprises. Since many theories are actually based on type theories, in this manner, a great amount of space for written code and hence, a great amount of time, will be saved by retrieving the features needed.

As an example, we took the case of the Isabelle foundation. The signature for the Pure sintax, as provided in [**?**], is shown below.

```
%sig Isabelle = {
    tp       : type.
    tm       : tp → type.
    func     : tp → tp → tp.                                    %infix right 5
        func.
    lambda   : (tm A → tm B) → tm (A func B).
    @        : tm (A func B) → tm A → tm B.                     %infix left 100
         @.
    prop     : tp.
    ded      : tm prop → type.                                  %prefix 0 ded.
    forall   : (tm A → tm prop) → tm prop.
    imp      : tm prop → tm prop → tm prop.                     %infix right 5
        imp.
    ==       : tm A → tm A → tm prop.                           %infix none 6 =
        =.
    forallI  : ({x: tm A} ded (B x)) → ded forall ([x] B x).
    forallE  : ded forall ([x] B x) → {x: tm A} ded (B x).
    impI     : (ded A → ded B) → ded A imp B.
    impE     : ded A imp B → ded A → ded B.
    refl     : ded X == X.
    subs     : {F: tm A → tm B} ded X == Y → ded F X == F Y.
    exten    : ({x : tm A} ded (F x) == (G x)) → ded lambda F == lambda G.
    beta     : ded (lambda [x: tm A] F x) @ X == F X.
    eta      : ded lambda ([x: tm A] F @ x) == F.
}.
```

All the symbols in Pure have, thus, a corresponding symbol in the union of the signatures Functions, UniversalQuantifiers, ChurchIntlogEquality, and Implication. (Implication is the signature formalized on top of Base, which includes the *imp* declaration, $imp : prop- > prop- > prop$, and its introduction and elimination rules). Moreover, the Pure syntax covers the union of these signatures. This is happening because every formalization of a certain feature only contains the declarations needed for that feature. For example, in the above case, Pure includes the constructs for function types, that is, func, lambda, @, exten, beta and eta. These are declared in the formalization of the function types feature in type theory the Function signature striclty contains only these declarations, together with the ones they depend on (for example, beta needs equality in order to be formalized). If Function would also contain, for example, the Curry typing, Pure will no longer be the union of the mentioned signatures, since in Pure, there is no manner to write extrinsically typed terms.

The conclusion is therefore that Isabelle's Pure syntax can be soundly and adequately formalized in the following fashion:

```
%sig Isabelle = {
      %include ChurchIntlogEquality.
      %include UniversalQuantifiers.
      %include Function.
      %include Implication.
}.
```

The formalization of the semantics is yet another gain: rather than specifying the model theory for every formalized theory, by constructing the theories in this manner, we can give meaning to them by simply including the meaning of all the corresponding features, included in the theory. Therefore, in this case, the model theory of Isabelle will be, rather than long, hard to implement and to descipher specification, the following view:

```
%view Isabelle-Model : Isabelle → ZFC = {
      %include ChurchIntlogEquality-Model.
      %include UniversalQuantifiers-Model.
      %include Function-Model.
      %include Implication-Model.
}.
```

5 Conclusions and Further work

Our specification of type theories in a modular fashion comprises all the necessary structure for adequately encoding type theoretical features. We have seen that on top of our basis formalization, depicted by the table, we can build any type theory desired, by treating each of its construct or type separately, as a stand-alone theory. Moving to the higer level of abstraction, the encoded features of type theories stay at the basis of entire formalized systems, which will be easily encoded, by merely including the features needed. This entire structure leads to a maximal amount of code reuse, and it brings efficiency with regards used space; as well, the time of the encoder is saved by reducing the time allocated for hundreds of lines of specification to only writing the component parts of the desired theory; the time spent to understand a system will also be broken down to only understandin each component of it.

The ability to express comples systems more and more easily comes with the library increase, with specifying increasingly many type theoretical features, based on the existent structure. Therefore, the work in progress is to expand the library with more type theoretical features and provide semantics for them (in the current case, ZFC serves as a foundational platform).

Not only the signatures are in our main focus, but also the manner of how the theories relate to each other: we have seen how the two types of logics and the two types of equality relate, but for typing styles, the translation between the two has not yet been formalized. As future work, the view from Church to Curry, as explained in detail in 3.1.3, will be formalized with the help of patterns.

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.