Master's Thesis

Translating of the IMPS library

Author:
Jonas Betzendahl

Program:
Intelligent Systems

Supervisors:
Prof. Dr. Michael Kohlhase
Prof. Peter Ladkin Ph.D.

April 16, 2018

Contents

Abstract

The IMPS system, by Farmer, Guttman and Thayer, was an influential automated reasoning system, pioneering mechanisations of features like theory morphisms, partial functions with subsorts and the little theories approach to the axiomatic method. It comes with a large library of formalised mathematical knowledge covering a broad spectrum of different fields.

Since IMPS is no longer under development, this library is in danger of being lost. In its present state, it is also not compatible for use with any other mathematical system.

To remedy that, I formalise the logic of IMPS (LUTINS), and draw on both the original theory library source files as well as the internal data structures of the system to generate a representation in a modern knowledge management format.

Using this approach, I translate the `foundation` section (a crucial subset of the IMPS theory library) to OMDoc and verify the result using type-checking in the MMT system against my implementation of LUTINS.

These results can serve as a solid foundation to making the whole IMPS theory library available in the future.

1 Introduction

The technologies of the internet and inexpensive computing have had a profound impact on how mathematics is performed in the 21st century. Computer systems are not only being used to store and retrieve mathematical knowledge, but have also taken their rightful place in the process of creating new math itself. They help to guide, verify, and – in a few specific cases – even generate new proofs and theorems.

Today, there are many libraries of formal knowledge for many different mathematical software systems. However, this wealth of information can also impose an undue penalty on anyone venturing to work with more than one of them, since the knowledge is often represented in a format that is only compatible with the system it was originally written for.
This barrier of non-interoperability severely limits possible uses of any one mathematical software system as well as progress on that system itself. Helpful tooling that was implemented with one system in mind often can not be used for developing another.

Some translations between different systems exist but they are often ad-hoc, only one-directional (although both systems are in frequent use) or overly restricted by the logical frameworks or foundations.
Most theorem proving systems today tend to fix (in what we will refer to as the *big theories approach*) one particular logical foundation along with its primitives (i.e. types, axioms, rules, ...) and only use conservative extensions to model domain knowledge (i.e. theorems, definitions, ...).
This goes against the way that modern mathematics is usually done on chalkboard or paper, where the foundation is often hidden and almost never directly referred to. It also makes it harder for two systems with different logical frameworks to successfully interact.

Approaching the problem with only direct system-to-system translations without common ground also means inviting threats of scale, since there would be $\mathcal{O}(n^2)$ different translations necessary to fully connect n systems with each other.

The second danger to effective use of mathematical knowledge libraries across software systems is that if one of these systems eventually falls out of use, the library is in danger of being lost to bitrot as fewer and fewer machines are actually capable of running the required software to interpret it.

In this thesis, we will discuss an attempt to "rescue" one library in particular from this fate. `IMPS` is an *I*nteractive *M*athematical *P*roof *S*ystem, originally developed at The MITRE Corporation by William M. Farmer, Joshua Guttman, and

Javier Thayer. Its library is home to a large amount of well-developed formalised mathematics with over 180 different theories and over 1200 distinct theorems and their proofs.
The system itself has not been in active development or regular use for well over 20 years now[1] and thus the library is in acute danger of being lost.

The `IMPS` system (and its library) is especially interesting because it was the first theorem proving assistant (to my knowledge) to make heavy use of *theory morphisms* and with emphasis on the *little theories* approach to mathematics.

Concretely, we want a translation of the `IMPS` library into OMDoc– the Open Mathematical Document format [Koh16]. OMDoc is a content markup scheme for (collections of) mathematical documents (including articles, textbooks, ...) with support for multiple degrees of formality. It can also serve as a medium / content language for communication between mathematical software services. It is being actively developed and used in research, which makes it a suitable candidate for our purposes.

MMT (short for *M*eta-*M*eta-*T*ool) is a foundation independent framework / language based on OMDoc for developing theories and meta-theories for knowledge representation. It is already being used to translate libraries of other theorem proving systems such as Mizar and PVS.
It also shares a few key design choices with the `IMPS` system, such as the focus on theory morphisms and adherence to the little theories approach, which makes it especially useful for this task and especially interesting to have the `IMPS` library available in the future.

This thesis builds on and advances the efforts by [Li02] of translating the `IMPS` library of mathematical knowledge to OMDoc. I extend and adapt their export mechanism to create a comprehensive `JSON` representation of the internal `IMPS` data structures. Both this and the original source files of the mathematical library are then parsed by my importer extension to MMT to create a structured and typed representation of almost all mathematical objects in the source files, with the only notable exception being proof scripts and macetes[2].
This representation can then easily be translated into the OMDoc/MMT language, with a formalisation of the foundational logic of `IMPS` (called LUTINS, see Section 2.1) serving as a formal basis for the translation.

The implementation of this process successfully translates the `foundation` section of the `IMPS` library, a small but crucial subset, mapping mathematical expressions in the source to usable mathematical expressions in the target language.

[1]The relevant entry on `theoremprover-museum.github.io` lists the active development phase as 1990 to 1993 [Far15]

[2]Macetes are tactic-like primitives for proofs, see Section 2.2.9

The generated output is verified (i.e. type-checked) by the MMT system against the implementation of the underlying logic LUTINS to ensure correctness of the translation progress.
This implementation has the benefit of future-proofing the OMDoc export against potential changes in the format. The results also provide a solid foundation for efforts of translating the entire IMPS library in the future.

The OMDoc output of the translation does not (only) need to be archived for the inherent value in not losing a rich database of formalised knowledge. The generated output could also be used in various ways by mathematical knowledge management systems or function as a reference point for other knowledge in a (partially) shared meaning space.

1.1 Related Work

There have been multiple attempts at translating libraries from one theorem proving system to another in an ad-hoc manner. For example, there exist translations from HOL Light to Coq [KW10], from Isabelle/HOL to Isabelle/ZF [KS10] (benefiting from the shared logical framework) and from HOL to Isabelle/HOL [OS06].
The translation approach using OMDoc/MMT has been previously used (and shown to be successful) in a number of importers for the MMT system for different mathematical systems, such as PVS [Koh+17b], Mizar [IKR11] and HOL Light [KR14], as part of the OAF (**O**pen **A**rchive of **F**ormalisations) project [OAF].

In all of these, the underlying logical foundation of the system has first been formalised natively in OMDoc/MMT as part of the LATIN library – an OMDoc-based atlas of formal logics, type theories, foundations and various translations between them ([Cod+11], [Rab14], available online at [LATIN]).
The resulting theory is then used as a meta-theory for importing the corresponding libraries. As in the case of this thesis, these imports tend to focus on translating the *statements* of theorems only, and pay less attention to the *proofs*, since proofs are often highly system-specific and difficult to translate without also reproducing all of the machinery of the system in question.
Li previously made an attempt to translate the IMPS math library to OMDoc in [Li02] to a reasonable degree of success. However, there have been a number of substantial changes to the OMDoc format since then (including multiple version bumps from 1.2 to 1.6). Furthermore, Li's approach was limited in a number of ways that are also addressed.

For one, this work differs from Li's previous attempt in that it doesn't try to translate from *only* the internal IMPS data structures. The implementation parses the source files those structures are generated from in a separate line of attack, allowing to compare and corroborate data from one direction of inquiry with data from the other.

The output of the translation is also verified by type-checking it in MMT on the basis of **LF** (short for *L*ogical *F*ramework, see [HHP93b] and [Rabb] and Section 2.3.3), whereas Li only checks the syntactic validity of the generated `XML` by an external tool.

1.2 Structure of This Thesis

In Section 2, I will introduce the necessary theoretical preliminaries for all involved systems, including MMT, `IMPS` and OMDoc. After that, I outline the general idea and some theoretical specifics of the translation process in Section 3. Section 4 elaborates on the specific implementation in MMT, Lisp and Scala. Finally, Section 5 concludes the thesis with a discussion of possible future work and a recap.

2 Preliminaries

2.1 Preliminaries: LUTINS

LUTINS (pronounced /ly.tɛ̃/, as in French, short for "**L**ogic of **U**ndefined **T**erms for **I**nference in a **N**atural **S**tyle", also the French translation of the English word "imps") is the underlying logic of the `IMPS` system.

LUTINS is a variant of Church's simple theory of types [Chu40]. It was developed to allow computerised mathematical reasoning that closely follows mathematical practise as performed by mathematicians "in the wild". And since standard mathematical reasoning often focuses on functions, their properties and operators on them, LUTINS allows for partial functions (meaning they are not necessarily defined on all arguments), and features a definite description operator as well as a system of subtypes.

LUTINS is a classical logic in the sense that it allows non-constructive reasoning, but non-classical in the sense that terms in LUTINS can be non-denoting. It also supports the following mechanisms:

- λ-Notation for functions;
- An infinite hierarchy of function types for higher-order functions;
- Full quantification (existential and universal) over all function types.

In this section, I will give a basic overview of LUTINS. A more thorough presentation of the logic can be found in [Gut91], focus is given to subtypes in [Far93a] and to partial functions in [Far90]. An overview of the semantics can also be found in [FGT98].

2.1.1 Languages

The notion of languages is central to LUTINS. Languages are 4-tuples $\langle \mathcal{A}, \xi, \mathcal{V}, \mathcal{C} \rangle$ where $(\mathcal{A}, \xi)$ is a sort system, $\mathcal{V}$ is a countable set of *variables* and $\mathcal{C}$ is a countable set of *constants*. For a detailed explanation, see [Far93b].

On a more abstract level, LUTINS languages contain two classes of objects: sorts (Section 2.1.2) and expressions (Section 2.1.3). Sorts denote (non-empty) domains of mathematical objects and expressions denote members of these domains. Expressions can be used to directly reference mathematical objects and to make statements about them.

2.1.2 Sorts, Types and Kinds

In the following, let $\mathcal{L}$ be the name of a hypothetical language. The set $\mathcal{S}$ of sorts of $\mathcal{L}$ is generated inductively from a finite set $\mathcal{A}$ of *atomic* sorts (that are explicitly stated by $\mathcal{L}$):

- Every $\alpha \in \mathcal{A}$ is in $\mathcal{S}$.
- If $\alpha_1, \ldots, \alpha_n$ $(n \geq 1)$ all are in $\mathcal{S}$, then $[\alpha_1, \ldots, \alpha_{n+1}]$ is in $\mathcal{S}$.

The latter class of sorts is called *compound sort*, denoting the domain of n-ary functions from $\alpha_1, \ldots, \alpha_n$ into α_{n+1}. Compound sorts can have arbitrary arity, so currying is not required (though it later became necessary in the implementation, see Section 4.4.2). Taking functions is, however, the only type-forming operation in LUTINS. Sorts may overlap, but they cannot be empty.

The *range* of a given sort α is defined as follows: if α is a function sort of the form $[\alpha_1, \ldots, \alpha_n, \alpha_{n+1}]$, then $\text{ran}(\alpha) = \alpha_{n+1}$. Otherwise, $\text{ran}(\alpha) = \text{ran}(\beta)$, where β is the enclosing sort of α. The notation for the range of α is $\text{ran}(\alpha)$.

Every language includes the base type $\star$ (sometimes also defined as $*$), denoting the set $\{\mathtt{T}, \mathtt{F}\}$ of standard truth values.

Sorts are divided into two *kinds*[3], $\star$ (read: *star* or *prop*) and ι (read: *ind*). A given sort α is of kind $\star$ if either $\alpha = \star$ or α is a compound sort *into* $\star$ (i.e. a compound sort of the form $[\alpha_1, \ldots, \alpha_n, \star]$, sometimes also called a *predicate*). In *all* other cases α is of kind ι. This includes all atomic sorts except $\star$ itself.

Note:
Although it is possible to encode mathematical objects to be of kind $\star$, most mathematics encoded in `IMPS` is of kind ι, and the deductive machinery reflects that. Predicates and formulas tend to be the only *useful* objects of kind $\star$. In particular, any term of kind $\star$ with an undefined component denotes `F`.

Subsorts

LUTINS prominently features subsorts in its type theory. Sorts can be defined to be subsorts of other sorts in multiple ways that will be discussed further in Section 2.2. For instance, the natural numbers $\mathbb{N}$ form a subsort of the real numbers $\mathbb{R}$ and the continuous (real) functions a subsort of the functions from $\mathbb{R}$ to $\mathbb{R}$.

Each atomic sort is also assigned a unique *enclosing sort* by the language $\mathcal{L}$ that defines it. This gives rise to a particular partial order on $\mathcal{S}$, which we will call $\preceq$, that has the following properties:

[3]not to be confused with **LF** kinds

Definition 1 *Properties of* LUTINS *subsort relation*

1. *If $\alpha \in \mathcal{A}$ and β is the enclosing sort of α, then $\alpha \preceq \beta$.*
2. *$\alpha_1 \preceq \beta_1, \ldots, \alpha_n \preceq \beta_n$ if and only if $[\alpha_1, \ldots, \alpha_n] \preceq [\beta_1, \ldots, \beta_n]$*
3. *If $\alpha, \beta \in \mathcal{S}$ with $\alpha \preceq \beta$ and α is compound, then β is also compound and has the same length as α.*
4. *For every $\alpha \in \mathcal{S}$, there exists a unique (!) $\beta \in \mathcal{S}$ s.t. $\alpha \preceq \beta$ and β is maximal with respect to $\preceq$.*

The relation $\preceq$ (also sometimes called "the subsort relation") is intended to denote set inclusion. A sort that is maximal in relation to $\preceq$ is called a *type*. The type of a given sort α has the notation $\tau(\alpha)$. A type is called a *base type* if it is atomic (i.e. non-compound).

$\beta \sqcup \gamma$ is notation for the *least upper bound* of the sorts β and γ with respect to the partial order $\preceq$. The least upper bound of two sorts with the same type is always defined.

Subsorting also applies to compound sorts, which integrates nicely since functions in LUTINS can be partial.
In particular, if $\sigma_0 \preceq \tau_0$ and $\sigma_1 \preceq \tau_1$, then $(\sigma_0 \rightarrow \sigma_1) \preceq (\tau_0 \rightarrow \tau_1)$. More precisely, the members of $(\sigma_0 \rightarrow \sigma_1)$ are exactly those functions that are never defined outside of σ_0 and never yield results outside of σ_1.

All of this is helpful for mechanised deduction because the subsorting relation can give important information about the value of an expression, should it be defined. Furthermore, many theorems have constraints that can easily be expressed in terms of a subtype and the prover can be programmed to handle these with special algorithms.

2.1.3 Expressions

Every language $\mathcal{L}$ also defines a set of *constants*, called $\mathcal{C}$. Each constant is assigned a sort in $\mathcal{S}$. The set of *expressions* of $\mathcal{L}$ (of sort α) is then also inductively defined via $\mathcal{C}$ and application of the constructors to the elements of $\mathcal{C}$.

Constructors in LUTINS are logical constants and can be used to form compound expressions. They are not part of any special language or scope, but rather are available for every theory and language.

While users of `IMPS`/LUTINS can not add additional constructors in this exact sense, it *is* possible to add so-called *quasi-constructors* (more closely examined in Section 2.2.7).

Constructor	**Mathematical Syntax**
`the-true`	T
`the-false`	F
`not`	$\neg\varphi$
`and`	$\varphi_1 \wedge \cdots \wedge \varphi_n$
`or`	$\varphi_1 \vee \cdots \vee \varphi_n$
`implies`	$\varphi \supset \psi$
`iff`	$\varphi \equiv \psi$
`if`	$\text{if}_\iota(\varphi, t_1, t_2)$
`if-form`	$\text{if}_\star(\varphi_1, \varphi_2, \varphi_3)$
`forall`	$\forall v_1 : \alpha_1, \ldots, v_n : \alpha_n, \varphi$
`forsome`	$\exists v_1 : \alpha_1, \ldots, v_n : \alpha_n, \varphi$
`lambda`	$\lambda v_1 : \alpha_1, \ldots, v_n : \alpha_n, t$
`equals`	$t_1 = t_2$
`apply`	$f(t_1, \ldots, t_2)$
`iota`	$\iota v : \alpha, \varphi$
`iota-p`	$\iota_\star v : \alpha, \varphi$
`is-defined`	$t\downarrow$
`defined-in`	$t\downarrow \alpha$
`undefined`	$\perp_\alpha$

Figure 1: The logical constructors in LUTINS

For example: one of the quasi-constructors that `IMPS` supplies natively is *quasi-equality.* Two terms are quasi-equal if they are either both undefined or both defined and equal.

All the fixed logical constructors available in LUTINS can be found in Figure 1. The precise rules for expression construction can be found in Figure 2 on page 10.

2.1.4 Partial Functions, Undefined and Non-Denoting Values

The stated goal of `IMPS` (and therefore LUTINS) is to allow for reasoning that is very close to mathematical practice. This means that there needs to be a way to deal with partial functions and undefined values since these make frequent appearances in chalk-and-whiteboard mathematics.

For example, all of the following terms are undefined in the standard theory of arithmetic over the real numbers:

$$\frac{5}{0} \qquad \sqrt{-3} \qquad \ln(-4) \qquad \tan\left(\frac{\pi}{2}\right)$$

Perspectives on undefined values

Historically, since at least [Rus05], there has been a plethora of opinions by logicians on how to approach the problem of non-denoting terms and non-denoting

Definition 2 *Rules for* LUTINS *expressions.*

1. *If x is a name and $\alpha \in \mathcal{S}$, then $x : \alpha$ is an expression of sort α.*

2. *If $A \in \mathcal{C}$ has been assigned sort α, the A is an expression of sort α.*

3. *If $A, B, C, A_1, \ldots, A_n$ are expressions of sort $\star$, then T, F, $\neg(A)$, $(A \supset B)$, $(A \equiv B)$, if-form(A, B, C), $(A_1 \wedge \cdots \wedge A_n)$, and $(A_1 \vee \cdots \vee A_n)$ are also expressions of sort $\star$.*

4. *If $x_1, \ldots, x_n$ are distinct names; $\alpha_1, \ldots, \alpha_n \in \mathcal{S}$; A is an expression of sort β; and $n \geq 1$, then $(\forall\, x_1 : \alpha_1, \ldots, x_n : \alpha_n, A)$ as well as $(\exists\, x_1 : \alpha_1, \ldots, x_n : \alpha_n, A)$ are expressions of sort $\star$.*

5. *If A and B are expressions of sort α and β respectively, with $\tau(\alpha) = \tau(\beta)$, then $(A = B)$ is an expression of sort $\star$.*

6. *If F is an expression of sort α and type $[\alpha_1, \ldots, \alpha_n, \alpha_{n+1}]$; $A_1, \ldots, A_n$ are expressions of sort $\alpha'_1, \ldots, \alpha'_n$; and $\alpha_1 = \tau(\alpha'_n), \ldots, \alpha_n = \tau(\alpha'_n)$, then $F(A_1, \ldots, A_n)$ is an expression of sort ran(α).*

7. *If $x_1, \ldots, x_n$ are distinct names; $\alpha_1, \ldots, \alpha_n \in \mathcal{S}$; A is an expression of sort β; and $n \geq 1$, then $(\lambda\, x_1 : \alpha_1, \ldots, x_n : \alpha_n, A)$ in an expression of sort $[\alpha_1, \ldots, \alpha_n, \beta]$*

8. *If x and y are names; α and β are sorts of kind ι and $\star$ respectively; and A is an expression of sort $\star$, then $(\iota\, x : \alpha, A)$ and $(\iota_\star\, y : \beta, A)$ are expressions of sort α and β respectively.*

9. *If A, B, C are expressions of sorts $\star, \beta, \gamma$ respectively, with $\tau(\beta) = \tau(\gamma)$, then if$(A, B, C)$ is an expression of sort $\beta \sqcup \gamma$.*

10. *If A is an expression and $\alpha \in \mathcal{S}$, then $(A\!\downarrow)$ and $(A\!\downarrow\!\alpha)$ are expressions of sort $\star$.*

11. *If α is a sort of kind ι, then $\perp_\alpha$ is an expression of sort α.*

Figure 2: LUTINS rules for creating expressions

definite description (e.g. "the present king of France") in particular. This is discussed more extensively in [Far90], where three perspectives are presented.

The perspective of the (philosophical) logician is different from that of the working mathematician. Often, the mathematician is less concerned with the meaning of non-denoting terms than the philosopher.
For example: while a mathematician usually takes terms like $\frac{1}{0}$ to be legitimate expressions that can be manipulated and reasoned with, but with absolutely no denotation. The logician, on the other hand, may think that an expression like "the present king of France" should absolutely denote something that at least "is a king", even if it doesn't exist.
Farmer attributes a third perspective to the computer scientist, for whom the

denotation of a term like $\frac{1}{0}$ would more likely be an "error" value, to be treated much like any other value (which – in turn – would be unpleasant for the mathematician). Even though functions are usually *strict* in regard to error values (i.e. a function applied to an error value also evaluates to an error value), some are also *non-strict*. Consider, for example, the following function:

$$f(x,y,z) = \begin{cases} y, & \text{if } x = 1 \\ z, & \text{if } x \neq 1 \end{cases}$$

The function f is non-strict, because $f(1,b,c) = b$, even if c is an error value. Non-strict functions are common in programming since they can lead to great increases in efficiency, especially if evaluating one or more of the branches is computationally expensive or would incur side effects.

The perspective that IMPS takes is – not too surprisingly – the mathematician's perspective. To live up to this, Farmer discusses several different approaches and their pros and cons, eventually settling on *"Partial valuation for terms, total valuation for formulas"*. This is represented in LUTINS in the dichotomy between kind $\star$ (terms of which evaluate to F if they have undefined subterms) and kind ι (terms of which can be *genuinely* non-denoting).

Note that there is a subtle difference between a term that is "undefined" and one that is "non-denoting". According to Farmer, a term is undefined if it is not assigned a "natural" meaning (e.g. "top of an empty stack") and non-denoting if it is not to be assigned any meaning at all.

Often, an undefined term is also non-denoting, but it *can* still have a denotation. In particular, a term of type $\star$ always has a denotation (if one of its constituents is undefined, that denotation is F).

Indicator Functions

An *indicator function* (sometimes just called an *indicator*) is a function used to represents a set over a given sort α. Normally, one could expect such a function to have a type of ($\alpha \to$ Boolean). In IMPS, however, since functions of kind ι can be partial, they instead map elements of the sort that are supposed to be included in the set into `unit%sort`, a sort from IMPS's `the-kernel-theory` (see Section 2.2.10) that only has one element (called `an%individual`).

This largely reduces the problem of membership in a set that is defined by an indicator function to definedness, for which the simplifier of the IMPS system is especially well-equipped.

Indicator functions over a sort α are used extensively in the library (for example to talk about ideals, compare Figure 3), with their own sort usually presented not

```
(def-constant ideal
  "lambda(x:sets[zz],
       nonempty_indic_q{x} and
       forall(a,b,c:zz, a in x and b in x
         implies
       (a+b) in x and c*a in x))"
  (theory h-o-real-arithmetic))

(def-theorem divisibility-preserves-ideal-membership
  "forall(x:sets[zz], k,m:zz ,
      ideal(x) and k in x and k divides m
        implies
      m in x)"
  (theory h-o-real-arithmetic)
  (proof ...))
```

Figure 3: `IMPS` expressions (from “primes.t”) using indicator functions

as ($\alpha \rightarrow$ `unit%sort`), but just as `sets[`α`]`. This is not a true type-constructor, however, but merely syntactic sugar.

2.1.5 Definite Description

One of the more prominent features of LUTINS is the possibility of reasoning with definite description via the ι (or *iota*) constructor.
Given a variable v of sort α of *kind* ι (not to be confused with the constructor itself) and an unary predicate φ over α, the expression

$$\iota\ v : \alpha \,.\, \varphi(v)$$

denotes the *unique* v, such that $\varphi(v)$, if there exists such a v. If there is no or more than one v that fulfils the predicate, the ι-expression is undefined.
For example, the following expression denotes $\sqrt{2} \in \mathbb{R}$:

$$\iota\ x : \mathbb{R}.(0 \leq x) \wedge (x \cdot x = 2)$$

While *this* expression is undefined:

$$\iota\ x : \mathbb{R}.x \cdot x = 2$$

Definite description can be very useful for dealing with functions, especially partial functions. For example: division over the reals of sort $[\mathbb{R}, \mathbb{R}, \mathbb{R}]$ (undefined whenever the second argument equals 0) can easily be defined over its relation to multiplication with the ι constructor:

$$\texttt{div} = \lambda\ x, y : \mathbb{R}.\ \iota\ z : \mathbb{R}.\ x \cdot z = y$$

The same applies to the square root or even the n^{th} root. The analogue to the following expression defines the `sqrt`-constant in the `reals.t` source file:

$$\texttt{sqrt} = \lambda\, x : \mathbb{R}.\ \iota\, y : \mathbb{R}.\ (0 \leq y) \wedge (y \cdot y = x)$$

Definite description for kind $\star$

There is a second definite description operator in LUTINS, called $\iota_\star$ (or *iota-star*). It has fundamentally the same semantics as ι, with one key difference: since kind $\star$ does not allow for non-denoting expressions, if the predicate can not be uniquely fulfilled, instead of being non-denoting, the expression here is actually false.
However, this operator is of little practical use and is not well supported by the `IMPS` deductive machinery. Hence, it is also not used in the library.

2.2 Preliminaries: `IMPS`

`IMPS` (short for "Interactive Mathematical Proof System") is an interactive theorem prover developed by William Farmer, Joshua Guttmann and Javier Thayer from 1990 to 1993 [Far15]. It was one of the influential systems in the era of automated reasoning.

It was originally developed in the `T` programming language (see [Kra+86] and [RAM88]) and later ported to Common Lisp [FGT98].

One of the goals in developing `IMPS` was to create a mathematical system that gave computational support to mathematical techniques common among actual mathematicians.
The development of the `IMPS` system has been heavily influenced (see [FGT98]) by three insights into real-life mathematics:

- Mathematics emphasizes the *axiomatic method.* The characteristics of mathematical structures are captured in axioms. Theorems are then derived from these axioms for *all* structures that satisfy the axioms.
  Often, what is needed for a proof is a clever change of perspective to see that one structure is indeed an instance of another theory, bringing additional theorems to bear.

- Many branches of mathematics emphasise *functions*, including partial functions. Moreover, the classes of objects studied may be nested, as are the integers and the real numbers; or overlapping, as are the bounded functions and the continuous functions.

- Mathematical proofs usually employ a mixture of both *formal inference* and *computation.*

Special attention is directed at the interplay of *computation* and *proof.* Farmer, Guttman and Thayer emphasise that, for example, a mathematician might devote considerable effort into proving lemmas that justify computational procedures[4] but are ultimately uninterested in the part of the derivation that is the "implementation" of these procedures.
Therefore, `IMPS` also allows for inferences based on sound computation and not merely formal inference. These are treated as atomic inferences, although a full formalisation in – for example – a Gentzen-style system might require hundreds or thousands of inference steps.

[4][FGT98] gives the example of the algorithm for differentiating polynomials for this.

```
(def-atomic-sort nn                  ;;; Name
  "lambda(x:zz, 0<=x)"               ;;; Defining Expression
  (theory h-o-real-arithmetic)       ;;; Home Theory
  (witness "0"))                     ;;; Witness to show the sort non-empty
```

Figure 4: IMPS Source Code (from "reals.t"): Def-Form defining the atomic sort $\mathbb{N}$

2.2.1 Little Theories

When following the axiomatic method to do mathematics – that is, logically reasoning from a given set of sentences in a formal language – there are two prominent approaches to chose from, which we will refer to as the "little theories" and "big theories" approach.

In the "big theories" version of the axiomatic method, all reasoning is carried out in one highly expressive axiomatic theory. The set of axioms selected is powerful enough, such that any model of them will contain all the mathematical objects that are of interest to us, and deduction from these powerful axioms will be enough to prove the relevant theorems in the theory.
Popular examples for a "big" axiomatic theory would be **ZFC** (Zermelo–Fraenkel set theory plus the axiom of choice), often cited as the "standard axiom set for set theory"([Bag17]), but also the Calculus of Inductive Constructions, the logic behind the Coq Theorem Prover [BC04].

Contrasted with that, the "little theories" approach uses a number of different theories with smaller, less powerful sets of axioms, to develop mathematics in. For example, one theorem could be true for all semi-rings, while another is only true in the theories of *commutative* rings.
Theorems are proved by logical derivation from the axioms of whatever theory supplies the necessary structure for the proof.

Both IMPS and MMT subscribe to the "little theories" approach to formal mathematics, a design choice that was informed by the face that the little theories approach lends itself well to the mechanism of theory interpretations [FGT92].

2.2.2 Def-Forms

IMPS source files contain information in so-called "def-forms" (short for *"definition forms"*). Each def-form is essentially the specification of one IMPS entity, from constants, theories, languages to translations.

For an example, see Figure 4.

All def-forms have a required argument of either `(theory ...)`, `(home-theory ...)` or `(language ...)` indicating in which language or theory the resulting mathematical object is to be installed.

For a complete list of all def-forms and their precise meaning within `IMPS`, refer to chapter 15 of [FGT98]. In section 3.3.2, we will discuss some of the more important def-forms and how they are translated.

2.2.3 Theories

In `IMPS`, a *theory* is a language $\mathcal{L}$ (i.e. an instance of a LUTINS language) coupled with a set of sentences in $\mathcal{L}$, called "axioms". Theories are the basic unit of representing mathematical knowledge in `IMPS`. In fact, Farmer (in [FGT98]) calls `IMPS` "a system for developing, exploring, and relating theories".

Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two `IMPS` theories. We call $\mathcal{T}_2$ a *supertheory* of $\mathcal{T}_1$ (or, analogously, $\mathcal{T}_1$ a *subtheory* of $\mathcal{T}_2$), if the language of $\mathcal{T}_1$ is a sublanguage of the language of $\mathcal{T}_2$ and every axiom of $\mathcal{T}_1$ is a theorem (not necessarily an axiom) of $\mathcal{T}_2$.

On definition, each theory $\mathcal{T}$ is assigned a (non-empty, at least **the-kernel-theory** is always included, see Section 2.2.10) set of *component theories*. Every component theory of $\mathcal{T}$ is also a subtheory of $\mathcal{T}$.

$\mathcal{T}_2$ is a *structural supertheory* of $\mathcal{T}_1$ (or, analogously, $\mathcal{T}_1$ is a *structural subtheory* of $\mathcal{T}_2$) if $\mathcal{T}_1$ is either a component theory of $\mathcal{T}_2$ or a structural subtheory of a component theory of $\mathcal{T}_2$.

2.2.4 Theorems

A *theorem* of a given theory $\mathcal{T}$ in `IMPS` is defined as a sentence of that theory that is true in all of $\mathcal{T}$s models. Note that this is a *semantic* definition of theorem, and does not directly depend on the `IMPS` proof system or any proof calculus for LUTINS.
Naturally though, the user has to rely on said proof system to *verify* that a theorem is actually a theorem and to *install* it in the theory so it can be used and built upon.

When a theorem is installed in a theory $\mathcal{T}$, it is also automatically installed in every structural supertheory of $\mathcal{T}$.

A theorem can be defined with a list of *usages* that tell the `IMPS` system how this theorem can be employed down the line. For example, the theorem could be used to create a rewrite rule, or a macete (see Section 2.2.9). For a precise listing of all options, refer to Section 6.4 in [FGT98].

2.2.5 Definitions

`IMPS` allows for four kinds of definition within a theory:

1. **Atomic Sorts**
   Examples: sort `nn` for $\mathbb{N}$, sort `zz_mod` for $\mathbb{Z}/n\mathbb{Z}$ for some n.

2. **Constants**
   Examples: constants like `true%val`, but also functions like `floor` or `divides`

3. **Recursive Functions**
   Examples: functions like `sum` (Σ), `product` (Π)

4. **Recursive Predicates**
   Examples: Predicates like `even` and `odd`

2.2.6 Theory Morphisms

A *theory morphism* (sometimes also called a *theory translation* or a *theory interpretation*) is a translation between two theories that maps expression from the one theory to expressions in the other, with the additional property that theorems are always mapped to theorems ([Far93b] and [FGT98]).

This is an integral part of the "little theories" approach (see Section 2.2.1), as theory morphisms are the tool to use to make results of one theory available in the other.

It is also close to mathematical practice, since seeing one structure as an instance of another (and therefore bringing all theorems of the other structure into play) is often the critical insight in non-trivial mathematical proofs.

The approach `IMPS` takes to theory morphisms ([Far93b]) is closely modelled after the standard approach to theory interpretations in first-order logic (see, for example, [End72], [Sho67] or [Mon76]).

Translations

Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be theories. We then call a *translation* from $\mathcal{T}_1$ to $\mathcal{T}_2$ any pair $\Phi = (\mu, \nu)$ where

- μ is a mapping from the atomic sorts of $\mathcal{T}_1$ to the sorts, unary predicates and indicators of $\mathcal{T}_2$ and

- ν is a mapping from the constants of $\mathcal{T}_1$ to expressions of $\mathcal{T}_2$.

We also in this context call $\mathcal{T}_1$ and $\mathcal{T}_2$ the *source theory* and the *target theory*, respectively.
The translation of any expression e of $\mathcal{T}_1$ is also an expression in $\mathcal{T}_2$ and we write it as $\Phi(e)$. It is generated from μ and ν in a manner that preserves expression structure. In particular, if e is a closed formula in $\mathcal{T}_1$, $\Phi(e)$ is a closed formula in $\mathcal{T}_2$.
We call a translation *normal* when μ maps all sorts of $\mathcal{T}_1$ to sorts in $\mathcal{T}_2$. If a translation is not normal, then some constructors that bind variables – λ, $\exists$, $\forall$, and ι – need to be *relativised.* Say – for example – the μ-part of Φ maps the sort α to the unary predicate P. Then all constructors that bind variables of sort α must be relativised.
Consider the following translation:

$$\Phi\left(\forall x : \alpha.\psi\right) = \forall x : \beta.P(x) \supset \Phi(\psi)$$

Something similar happens when μ maps a sort to an indicator instead of a predicate.
A translation that maps theorems to theorems is called a *theorem interpretation* or a *theory morphism.*

2.2.7 Quasi-Constructors

In addition to the LUTINS core logical constructors (see Figure 1 on page 9) it is also possible for a user of `IMPS` to define additional constructor-like forms called “quasi-constructors”. These are implemented as “macros” or “abbreviations”.

For example, in `IMPS`, there exists the notion of quasi-equality: two expressions are quasi-equal if and only if they are either both undefined or are both defined with the same value. In mathematical notation, this would be captured by the following biconditional:

$$E_1 \simeq E_2 \equiv (E_1{\downarrow} \vee E_2{\downarrow}) \supset E_1 = E_2$$

If the `IMPS` system now encounters an expression of the form $P \simeq Q$, it builds an internal expression of the form $(P\downarrow \vee Q\downarrow) \supset P = Q$. If it later finds an expression of this form that it is supposed to print, it will print it in the syntax $P \simeq Q$. These special syntax elements for quasi-constructors can also be defined by the user via def-form.

More precisely, a quasi-constructor consists of

- a name,
- a list of variables and
- a schema

In our example of quasi-equality, the name would be something like `quasi-equals`, the variables would be E_1 and E_2 and the schema would be $(E_1{\downarrow} \vee E_2{\downarrow}) \supset E_1 = E_2$.

User-defined Quasi-constructors

Users can define their own quasi-constructors with the `def-quasi-constructor` def-form. It takes a name, a mathematical expression and a designated language as arguments. The quasi-constructor is then available to use like one would any other constructor.

A possible implementation of quasi-equality in this def-form could look like the following:

```
(def-quasi-constructor QEQUALS
  "lambda(e1,e2:ind, #(e1) or #(e2) implies e1 = e2)"
  (language the-kernel-theory))
```

Figure 5: A possible implementation of quasi-equality as a quasi-constructor

System Quasi-Constructors

While Figure 5 shows one way that quasi-equality *could* be implemented in IMPS, this does not actually reflect the state of the system.
In addition to the user-defined quasi-constructors, the IMPS system also has a small number of so-called "system quasi-constructors" that are hard-wired into the deductive machinery. Quasi-equality is one of them. For a full list, see Figure 6.
Like logical constructors from LUTINS, these five system quasi-constructors are available in every theory, and not bound to any one language.

Quasi-Constructor	**Schema**
`quasi-equals`(E_1, E_2)	$(E_1{\downarrow} \vee E_2{\downarrow}) \supset E_1 = E_2$
`falselike`$([\alpha_1, \ldots, \alpha_{n+1}])$	$\lambda\, x_1 : \alpha_1, \ldots, x_n : \alpha_n \,.\,$ `falselike`(α_{n+1})
`domain`(f)	$\lambda\, x_1 : \alpha_1, \ldots, x_n : \alpha_n \,.\, f(x_1, \ldots, x_n){\downarrow}$
`total?`$(f, [\beta_1, \ldots, \beta_{n+1}])$	$\forall\, x_1 : \beta_1, \ldots, x_n : \beta_n \,.\, f(x_1, \ldots, x_n){\downarrow}$
`nonvacuous`(p)	$\exists\, x_1 : \alpha_1, \ldots, x_n : \alpha_n \,.\, f(x_1, \ldots, x_n){\downarrow}$

A couple of specifics to note:

- E_1 and E_2 need to have the same type.
  - The sort of f is $[\alpha_1, \ldots, \alpha_{n+1}]$
  - The sort of p is $[\alpha_1, \ldots, \alpha_n, \star]$
- $\tau([\alpha_1, \ldots, \alpha_{n+1}]) = \tau([\beta_1, \ldots, \beta_{n+1}])$

Figure 6: The five pre-defined quasi-constructors in `IMPS`

The reason these particular quasi-constructors need to be treated separately is that their schema can not accurately be represented using a LUTINS lambda expression. Usually, this is because they contain variables that range over function expressions without a fixed arity or over expressions of both kind $\star$ and kind ι.

For example, quasi-equality is supposed to be well-defined as long as both expressions have the same type, no matter the kind. But if we were to go by the definition for `QEQUALS` above, this would not hold in the case that both arguments are of kind $\star$.

Polymorphism

Another thing to note about quasi-constructors is that they are polymorphic in their schema variables, even if this polymorphism is not made explicit in the notation. Consider the user-defined quasi-constructor `GROUP` in Figure 18 (on page 35). It takes a subset of a sort `gg` (in this context specifically, `gg` is the base type of `GROUP-LANGUAGE`), two functions (one binary, one unary) and an element of the sort to a conjunction of the ordinary group axioms.

While the defining lambda-expression for this quasi-constructor is written using the language of the theory it is being defined in (i.e. `groups`), this is done solely because using the language of the home theory circumvents the need to introduce a separate schema language.
The resulting quasi-constructor itself is actually polymorphic in the type `gg` and can be used in theories that are entirely unrelated to the group theory itself.

```
(def-algebraic-processor FIELD-ALGEBRAIC-PROCESSOR
    (language fields)
    (base ((operations
           (+ +_kk)
           (* *_kk)
           (- -_kk)
           (zero o_kk)
           (unit i_kk))
    commutes)))
...
(def-theory-processors FIELDS
    (algebraic-simplifier
    (field-algebraic-processor *_kk +_kk -_kk))
    (algebraic-term-comparator field-algebraic-processor))
```

Figure 7: Def-Forms for declaring and installing an algebraic processor

2.2.8 Processors

`IMPS` allows for so-called *processors* to be defined and installed within theories. These have the purpose to give the simplifier context about what reductions can be taken within a given theory, if the user has more information than the simplifier.

For example, the expression $2 + 1$ should reduce to 3, the expression $x + x - x$ should reduce to just x and the expression $x < x + 1$ should reduce to truth. Under some circumstances, the simplifier can already infer these, however, in some theories, it needs to be given extra information (e.g. "This field element behaves like a 0, this operation behaves like +, ...").

For an example of source code that defines and installs a processor, see Figure 7.

2.2.9 Proofs and Macetes

Macetes and proof commands are topics of significant importance for the `IMPS` system itself. However, we will only take a short look at them here, because neither play a big role in the translation (see Section 4.4.4).

The `IMPS` proof system is unusual in that it employs a deduction *graph* to represent proofs. A deduction graph consists of *sequent nodes* (i.e. an assertion and a context) and *inference nodes* (i.e. a conclusion node, a (potentially empty) set of hypothesis nodes and a justification[5] for asserting the given relationship between these).

[5]Also called an *inference rule*.

A user does not directly interact with the deduction graph, but only manipulates it indirectly via so-called *proof commands*, convenient procedures that add appropriate nodes by calling primitive inferences. The sequence of proof commands that proves a theorem via deduction graph is called a *proof script*.

For a closer inspection of the proof system, see Chapters 10 and 16 of [FGT98].

A macete (pronounced /mɐ.'se.tʃi/, from the Portuguese word for "chisel" or "clever trick") are a second way to manipulate expressions by applying theorems. The other way to do this is the IMPS simplifier. However, macetes offer a more direct and fine-grained control.

The general idea is that a macete M takes a context Γ and an expression e called the *source expression* and returns another expression $M(\Gamma, e)$, called the *replacement expression*. Using a combination language, simple macetes can also be formed into more complex ones.
The resulting functions can be used on the one hand to apply a (collection of) theorems to a node in a deduction graph, but also, on the other hand, to compute with theorems.

For a more in-depth look at macetes and their classification, see Chapter 12 of [FGT98].

You can find a theorem including its proof in Figure 13 on page 33. An example macete can be found in Figure 8.

```
(def-compound-macete GROUP-CANCELLATION-LAWS
  (series
    (repeat mul-associativity)
    (repeat left-cancellation-law)
    left-trivial-cancellation-law-left
    left-trivial-cancellation-law-right
    (repeat reverse-mul-associativity)
    (repeat right-cancellation-law)
    right-trivial-cancellation-law-left
    right-trivial-cancellation-law-right))
```

Figure 8: IMPS Source Code (from "groups.t"): Example macete for group cancellation laws

2.2.10 The Kernel Theory

`IMPS` has a default theory (named **the-kernel-theory**) that is a component of every other theory.
It serves as the representation of LUTINS within the `IMPS` system in that it introduces the base types `prop`, `ind` (for $\star$ and ι, respectively) and `unit%sort`. It also defines the constant `an%individual` to be of sort `unit%sort`.

It contains only one axiom:

$$\forall\ x : \texttt{unit\%sort}\ .\ x = \texttt{an\%individual}$$

And one theorem:

$$\forall\ x, y : \texttt{unit\%sort}\ .\ (x = y)\ \ \texttt{iff}\ \ TT$$

The kernel theory also defines an identity-translation that maps all sorts and constants mentioned above to themselves.

2.3 Preliminaries: OMDoc/MMT

OMDoc (short for **O**pen **M**athematical **D**ocuments) is a semantics-oriented markup format for STEM-related documents extending OpenMath developed by the KWARC work group (see [Koh16]). OMDoc/MMT ([Rabb]) extends a fragment of OMDoc by additional language features.
OMDoc/MMT brings with it three distinct levels for expression of mathematical knowledge:

- **Object Level**
  Expressions (e.g. terms and formulae) expressed in OpenMath.
- **Declaration Level**
  Constants (functions, types, judgements) with an optional (object-level) type and/or definition.
- **Module Level**
  Theories and Views; sets of declarations that inhabit a common namespace and context.
  (e.g. the theory of the lambda calculus)

Both *formal* and *informal* knowledge is supported at all three levels. This gives OMDoc/MMT a higher flexibility and allows for use in more diverse contexts such as data exchange between mathematical software systems (like theorem provers or web services) and natural language text books for e-learning.

Theories in OMDoc/MMT are structurally similar to theories in `IMPS` and can include other theories. Hence MMT-theories allow for library development in concordance to the little theories paradigm. Views in MMT behave (for all purposes relevant in this thesis) analogously to theory morphisms in `IMPS`. An example[6] is given in Figure 9.

[6]Taken from the theory `test1` in `https://gl.mathhub.info/Test/General/blob/master/source/general.mmt`

```
<omdoc>
  <theory name="test1" base="http://test.kwarc.info">
    <constant name="R">
      <type>
        <OMS base="http://cds.omdoc.org/urtheories" module="Typed"
            name="type"></OMS>
      </type>
    </constant>
    <constant name="O">
      <type>
        <OMS base="http://cds.omdoc.org/urtheories" module="Typed"
            name="type"></OMS>
      </type>
    </constant>
    <constant name="Ftp">
      <type>
        <OMS base="http://cds.omdoc.org/urtheories" module="Typed"
            name="type"></OMS>
      </type>
      <definition>
        <OMA>
          <OMS base="http://cds.omdoc.org/urtheories" module="LambdaPi"
              name="arrow"></OMS>
          <OMS base="http://test.kwarc.info" module="test1"
              name="R"></OMS>
          <OMS base="http://test.kwarc.info" module="test1"
              name="O"></OMS>
        </OMA>
      </definition>
    </constant>
  </theory>
</omdoc>
```

Figure 9: A simplified OMDoc/MMT example

2.3.1 The MMT System

The OMDoc/MMT language is used by the MMT system, which provides an API to handle OMDoc/MMT content and services such as type checking, rewriting of expressions and computation, as well as notation-based presentation of OMDoc/MMT content and a general infrastructure for inspecting and browsing libraries.
Since OMDoc/MMT avoids committing to a specific semantics or logical foundation, foundation-dependent services and features (e.g. type checking, presentation) are implemented using (foundation-independent) generic algorithms extensible by foundation-dependent calculus rules via plugins (e.g. for handling content imported from external systems such as `IMPS`).

2.3.2 Theory Graphs

Theories and theory morphisms naturally lead to *theory graphs*, with theories as vertices and morphisms as edges. In fact, OMDoc/MMT-theories and morphisms form a category, which is exploited by the MMT-system to induce and translate knowledge in/between theories analogously to `IMPS` (see Section 2.2.6).
The possible arrows in OMDoc/MMT are:

- **Includes**, which import all declarations from the domain to the co-domain. As morphisms, they behave like the identity morphism,
- **Views**, which are judgement-preserving maps from the declarations in the domain to expressions over the co-domain,
- **Structures**, which are omitted for this thesis, and
- the **Meta-theory**-relation, which behaves like an include for most purposes.

The meta-theory-relation connects theories that live on different meta-levels; e.g. domain knowledge to its logical foundation and conversely the logical foundation to the logical framework it is formalised in.

An example graph is given in Figure 10. Dotted lines represent the meta-theory-relation (e.g. `FOL` and `HOL` are formalised in `LF`), hooked arrows are includes (e.g. the theory of commutative groups `CGroup` includes `Monoid`), squiggly arrows represent views (e.g. `mod` maps `Monoid` to `ZFC`, as sets with union and the empty set form a monoid), the normal (labelled) arrows represent structures.
The MMT system also provides a theory graph viewer (see [RKM17]), an example for which is given in Figure 34.

$$\begin{tikzcd}
 & & \mathsf{LF} \arrow[rr, hook] \arrow[dl, dotted] \arrow[dr, dotted] & & \mathsf{LF}+\mathsf{X} \\
\mathsf{ZFC} & \mathsf{FOL} \arrow[l, squiggly, "\mathsf{folsem}"'] \arrow[rr, squiggly, "\mathsf{f2h}"] \arrow[dl, dotted] \arrow[d, dotted] & & \mathsf{HOL} \arrow[d, dotted] & \\
\mathsf{Monoid} \arrow[u, squiggly, "\mathsf{mod}"'] \arrow[r, hook] \arrow[rrr, bend left, "\mathsf{mult}"] & \mathsf{CGroup} \arrow[rr, bend right, "\mathsf{add}"] & & \mathsf{Ring} &
\end{tikzcd}$$

Figure 10: An examplary theory graph with different meta-levels

2.3.3 The Logical Framework **LF**

For our purposes, we fix as a foundation the logical framework **LF** (see [HHP93a]), since it is particularly well supported by the MMT system.
LF is based on a dependently-typed lambda calculus, intended as a framework to formalise logical systems (such as type theories, lambda calculi or classical logics) themselves. As such, it lends itself easily to standard formalisation practices such as judgements-as-types and higher-order abstract syntax and is used to formalise LUTINS in MMT (for details, see Section 3.2 and Appendix C).
Correspondingly, **LF** serves as a meta-theory for the LUTINS-theory, which in turn serves as a meta-theory for all theories imported from the `IMPS`-library.
For a slightly more elaborate discussion of **LF** and its interplay with MMT, see [Rabb].

3 The IMPS importing process

The IMPS theory library, developed in parallel to the IMPS system itself, is a collection of theories, theory morphisms and theory constituents (e.g. theorems or definitions)[7] that serves as a database of mathematics.

The theory library provides a large variety of basic mathematics (such as cardinalities, basic algebra and group theory, number theory, metric spaces, sequences, calculus, ...).
It formalises roughly 100 theories with over 110 theory morphisms and over 1900 distinct theorems. Its purpose was to provide the user with a solid starting point to develop their own theory library suited to their personal needs and interests. It also served as a collection of examples to demonstrate the different ways in which mathematics could be formalised using the IMPS system.

For a detailed introduction to the IMPS theory library and details on the sections contained therein, see also Chapter 18 of [FGT98].

3.1 The IMPS Math Library

IMPS theory libraries are typically (though not necessarily) organised in *sections*. A section is defined via the def-form `(def-section ...)` and includes a set of files and potentially also other sections.

For the purposes of this work, I decided to consider the `imps-math-library` section, with special focus on its `foundation` subsection.

This section works as a sort of "standard library" for developing mathematics in IMPS and defines the most essential and foundational theories, such as basic number theory, multiple generic theories and the critical `h-o-real-arithmetic`, the theory of the real numbers as a complete ordered field.
It also includes a few generic theories with no non-logical axioms, designed to reason about such objects as sets, unary functions, and sequences.

The `foundation` section is relatively simple to translate as it uses only a fraction of all def-forms available in IMPS (see Section 3.3.1). The wider `imps-math-library` covers a much broader array of mathematics and makes use of almost all def-forms. However, most of the *interesting* def-forms are already used in the `foundation` section.

A complete listing of the section definition and therein included files can be found in Appendix D.

[7]All of which are implemented in IMPS source files in the T programming language and interpreted by the IMPS system.

3.2 The LUTINS Theory in MMT

To formalise LUTINS in MMT, I use the logical framework **LF**, which provides a dependently typed lambda calculus with the following features:

- Two universes `type` and `kind` with `type:kind`
- Dependent function types

$$\prod_{x:A} T(x)$$

  (in **LF**-syntax: `{x:A}T(x)`). If T does not contain the variable x, this is the same as the simple function type $A \to T$.

  Dependent function types are inhabited by lambda expressions $\lambda x : A.t(x)$ (in **LF**-syntax: `[x:A]t(x)`). The usual rules in a lambda calculus (extensionality, beta-reduction, ...) hold.

The full formalisation of LUTINS is given in Appendix C. We will discuss some of the core principles used therein:

Fundamentally, I declare:

1. a new **LF**-type `tp:type`, which serves as the universe of maximal `IMPS`-sorts,
2. a function `sort : tp → type`, and
3. a function `exp : {A : tp} sort A → type`.

Given some maximal `IMPS`-sort `A`, the type `sort A` then serves as the **LF**-type of all `IMPS` sorts, and given a sort `a : sort A`, the type `exp A a` corresponds to the **LF**-type of all `IMPS`-expressions of sort `a`.

I use the principles of *higher-order abstract syntax* to specify *binders* in `IMPS`. For example, consider an `IMPS` expression $\lambda x : A.t$, where the λ-constructor binds a new variable $x : A$. I formalise this behaviour by declaring the `IMPS` lambda to be an **LF** function `lambda`, that takes an **LF** lambda expression as argument which binds the variable x. As a result we get the **LF** expression `lambda ([x:A] t)` being the application of the function `lambda` to the **LF** function `[x:A]t`, effectively "embedding" an **LF** function on `IMPS` expressions as an `IMPS` function. Quantifiers and other binders are treated analogously.

For propositional judgements (i.e. axioms and theorems) in `IMPS`, I use the *judgements-as-types* paradigm by introducing an operator `thm : exp bool → type`, assigning to each proposition a type which we can think of as the "type of proofs" for that proposition. Correspondingly, we consider a proposition A to be "true" if the type `thm` A is inhabited. Axioms correspond to undefined constants of type `thm A`, whereas theorems correspond to defined constants of that type, their definition being a proof (although proofs are omitted in this thesis).

3.3 Translation to OMDoc

3.3.1 Dead and Delayed Def-Forms

To avoid unnecessary work in implementation, I conducted a survey on the `T` source code via shell script to determine which def-forms were actually used and which were factually "dead". The script goes through a list of all def-forms (taken from [FGT98]) and counts how often they are actually being used in the library.

A def-form is considered "dead" if it is not used in the source code we consider (i.e. the section `imps-math-library`), even if it *is* used somewhere else in the code base. We distinguish this from a def-form that is merely "delayed", i.e. it appears in the `imps-math-library`, but not in the `foundation`. The MMT importer for `IMPS` does not support the semantics of dead or delayed def-forms (but can be extended to).

The results of the survey can be examined in Figure 11. The first column denotes the amount of def-forms of that particular sort in `imps-math-library`, the second in `foundation`.
It determined fifteen def-forms to be dead or delayed, five of which are dead and ten of which are delayed.
The def-forms that did not appear anywhere (dead) are the following:

- `def-bnf`
- `def-cartesian-product`
- `def-primitive-recursive-constant`
- `def-record-theory`
- `def-sublanguage`

And the def-forms that appeared in the `imps-math-library` but not in `foundation` (delayed) are as follows:

- `def-imported-rewrite-rule`
- `def-section`
- `def-theory-ensemble`
- `def-theory-ensemble-instances`
- `def-theory-ensemble-multiple`
- `def-theory-ensemble-overloadings`

	6	1	def-algebraic-processor
	10	1	def-atomic-sort
DEAD	0	0	def-bnf
DEAD	0	0	def-cartesian-product
	43	10	def-compound-macete
	158	19	def-constant
DELAYED	1	0	def-imported-rewrite-rule
	7	1	def-inductor
	35	11	def-language
	2	1	def-order-processor
DEAD	0	0	def-primitive-recursive-constant
	56	33	def-quasi-constructor
DEAD	0	0	def-record-theory
	14	3	def-recursive-constant
DELAYED	19	0	def-renamer
	12	4	def-schematic-macete
	14	1	def-script
DELAYED	1	0	def-section
DEAD	0	0	def-sublanguage
	1239	207	def-theorem
	91	18	def-theory
DELAYED	31	0	def-theory-ensemble
DELAYED	13	0	def-theory-ensemble-instances
DELAYED	3	0	def-theory-ensemble-multiple
DELAYED	12	0	def-theory-ensemble-overloadings
DELAYED	2	0	def-theory-instance
	6	1	def-theory-processors
	46	1	def-translation
DELAYED	20	0	def-transported-symbols
DELAYED	17	0	def-overloading
	62	36	def-parse-syntax
	117	66	def-print-syntax

Figure 11: Survey results for usage of each def-form.

- `def-theory-instance`
- `def-transported-symbols`
- `def-overloading`
- `def-renamer`

All of which were not considered in the implementation for this thesis, but might be available in a future translation.

```
(def-language GROUP-LANGUAGE
  (base-types gg)
  (constants
   (e "gg")
   (mul "[gg,gg,gg]")
   (inv "[gg,gg]")))

(def-theory GROUPS
  (language group-language)
  (component-theories h-o-real-arithmetic)
  (axioms
   (left-mul-id       "forall(x:gg, e mul x = x)" rewrite)
   (right-mul-id      "forall(x:gg, x mul e = x)" rewrite)
   (left-mul-inv      "forall(x:gg, inv(x) mul x = e)" rewrite)
   (right-mul-inv     "forall(x:gg, x mul inv(x) = e)" rewrite)
   (mul-associativity "forall(x,y,z:gg, (x mul y) mul z = x mul (y mul z))"
                                                                   rewrite)))
```

Figure 12: IMPS Source Code (from "groups.t"): theory and language for groups

3.3.2 Def-Form Specifics

In this section, we will examine some of the most important and most frequent def-forms in the `foundation` section and discuss their translation to **LF**.

def-theory and def-language

Since IMPS and MMT share so many design choices and philosophical approaches, most parts of the theory structure easy to translate. Theories in IMPS, for example, directly match theories in MMT.

MMT does not directly differentiate between *theories* and *languages* like IMPS does. Instead I translate IMPS theories as MMT theories and include the constructs from the corresponding language as *constants* in the MMT theory.

Example source code for a theory and its language in the IMPS system can be found in Figure 12.

def-theorem

To translate an IMPS theorem, one adds a constant of the name of the theorem to the correct theory. The mathematical expression of the theorem (which has the IMPS type `prop`) is combined with the `thm` operator to give it the **LF**-type of proofs of the mathematical expression in question (*judgements-as-types*).

Usually, undefined constants of this type would correspond to axioms and defined constants of this type would correspond to theorems. However, since proofs are not considered for this translation, effectively all theorems from IMPS are treated

```
(def-theorem INV-OF-INV
  "forall(x:gg, inv(inv(x))=x)"
  (theory groups)
  (usages rewrite transportable-macete)
  (proof
   ((instantiate-theorem mul-associativity ("inv(inv(x))" "inv(x)" "x"))
    (contrapose "with(p:prop,p)")
    (force-substitution "inv(inv(x)) mul inv(x)" "e" (0))
    (force-substitution "inv(x) mul x" "e" (0))
    simplify
    simplify
    simplify)))
```

Figure 13: `IMPS` Source Code (from "groups.t"): Theorem proving $(x^{-1})^{-1} = x$

```
(def-constant sqrt
  "lambda(x:rr, iota([[[y],rr]], 0<=y and y*y=x))"
  (theory h-o-real-arithmetic))
```

Figure 14: `IMPS` Source Code (from "reals.t"): Constant defining `sqrt`

as axioms (the assumption being that they do indeed hold). See also section 3.2 for details.
The proof scripts from `IMPS` can not be translated into **LF** in a sensible way; see Section 4.4.4 for details.

An example of an `IMPS`-theorem, along with its proof script, can be found in Figure 13.

def-constant and def-recursive-constant

Constants in `IMPS`-theories (both of the normal and the recursive variety) are translated as constants in MMT-theories. When a constant def-form is to be translated, a new MMT-constant is created and added to the correct theory. This MMT-constant has the name of the `IMPS`-constant and the defining mathematical expression as the definition.

For ordinary constants, we just use the normal mathematical expression as the definition. For recursive constants, however, we use the ι-representation that is provided by `IMPS` instead of the least-fixed-point-representation (also called "functional representation", compare Appendix B).

Examples of one normal and one recursive constant are in Figures 14 and 15. Figure 16 shows the ι-representation of the recursive constant `sum`.

```
(def-recursive-constant sum
  "lambda(sigma:[zz,zz,[zz,rr],rr],
     lambda(m,n:zz,f:[zz,rr], if(m<=n,sigma(m,n-1,f)+f(n),0)))"
  (theory h-o-real-arithmetic)
  (definition-name sum))
```

Figure 15: IMPS Source Code (from "reals.t"): Recursive Constant defining sum

```
iota(f_0:[zz,zz,[zz,rr],rr],
  forsome(g_0:[zz,zz,[zz,rr],rr],
    g_0
    =lambda(m,n:zz,f:[zz,rr],
       if(m<=n, g_0(m,n-1,f)+f(n), 0))
     and
     forall(h_0:[zz,zz,[zz,rr],rr],
       h_0
       =lambda(m,n:zz,f:[zz,rr],
          if(m<=n, h_0(m,n-1,f)+f(n), 0))
        implies
        forall(u_0,u_1:zz,u_2:[zz,rr],
         #(g_0(u_0,u_1,u_2))
          implies
          g_0(u_0,u_1,u_2)=h_0(u_0,u_1,u_2)))
    and
     f_0=g_0))
```

Figure 16: Defining ι-expression for sum, string representation

def-atomic-sort

Atomic sorts in IMPS are also added to MMT-theories as constants of the **LF**-type `sort k` (where `k` is the `tp` corresponding to the sorts kind). We also add a second statement that declares this sort a subsort of its enclosing sort.

If a `witness` is defined, we also add a theorem to the theory that the witness is of that sort.

An example sort definition for the sort of natural numbers (as enclosed by the integers) can be seen in Figure 17.

def-quasi-constructor

As we will discuss in more detail in Section 4.4.5, quasi-constructors form a special case of being a prominent feature of IMPS and absolutely crucial for a fine-grained, type-checkable translation of any sort, but also being uniquely difficult

```
(def-atomic-sort nn
  "lambda(x:zz, 0<=x)"
  (theory h-o-real-arithmetic)
  (witness "0"))
```

Figure 17: IMPS Source Code (from "reals.t"): Definition of the sort ℕ

```
(def-quasi-constructor GROUP
  "lambda(gg%:sets[gg], mul%:[gg,gg,gg], e%:gg, inv%:[gg,gg],
     forall(x,y:gg, x in gg% and y in gg% implies mul%(x,y) in gg%) and
     e% in gg% and
     forall(x:gg, x in gg% implies inv%(x) in gg%) and
     forall(x:gg, x in gg% implies mul%(e%,x)=x) and
     forall(x:gg, x in gg% implies mul%(x,e%)=x) and
     forall(x:gg, x in gg% implies mul%(inv%(x),x)=e%) and
     forall(x:gg, x in gg% implies mul%(x,inv%(x))=e%) and
     forall(x,y,z:gg, ((x in gg%) and (y in gg%) and (z in gg%)) implies
       mul%(mul%(x,y),z) = mul%(x,mul%(y,z))))"
  (language groups))
```

Figure 18: IMPS Source Code (from "groups.t"): Group quasi-constructor

to translate automatically.
This is why for now, quasi-constructors and their semantics are implemented by hand in the math parser, but when they are encountered in the T source files, they are just translated as opaque data.

def-translation

IMPS translations (which are all *interpretations* in the focused part of the library, i.e. all the obligations of the translation are theorems in the target theory) are translated as MMT theory morphisms, or *views*.
For more on MMT views and theory graphs, refer to [Iac09].

There are some special cases that can not be captured *directly* by MMT views, which we will look at more closely in the following paragraphs.

```
(def-translation GROUPS->SUBGROUP
  (source groups)
  (target groups)
  (assumptions
   "with(a:sets[gg], nonempty_indic_q{a})"
   "with(a:sets[gg], forall(g,h:gg, (g in a) and (h in a)
                                         implies (g mul h) in a))"
   "with(a:sets[gg], forall(g:gg, (g in a) implies (inv(g) in a)))")
  (fixed-theories h-o-real-arithmetic)
  (sort-pairs
   (gg (indic "with(a:sets[gg], a)")))
  (constant-pairs
   (mul "with(a:sets[gg], lambda(x,y:gg, if((x in a) and (y in a),
                                                 x mul y, ?gg)))")
   (inv "with(a:sets[gg], lambda(x:gg, if(x in a, inv(x), ?gg)))"))
  force-under-quick-load
  (theory-interpretation-check using-simplification))
```

Figure 19: IMPS Source Code (from “subgroups.t”): Subgroup Translation

Theory-Translations from a theory to itself

Some theory morphisms in IMPS map a theory $\mathcal{T}$ to itself. An example would be the identity translation of the-kernel-theory that maps everything to itself.

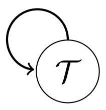

Figure 20: Theory morphism from $\mathcal{T}$ to itself. (IMPS)

MMT views, however, make the assumption that source and target theory are different, so to model a view from a theory $\mathcal{T}$ to itself in MMT, we create a copy of $\mathcal{T}$ called $\mathcal{T}'$ that plainly includes the original (and nothing else) and create a view from $\mathcal{T}$ to that.

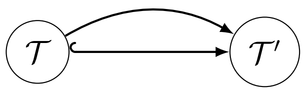

Figure 21: Theory morphism from $\mathcal{T}$ to a copy of itself. (MMT)

Theory-Morphisms with assumptions

A few theory morphisms in IMPS also have a collection of assumptions that need to be fulfilled (see Figure 19). These assumptions can be used to state that certain conditions must be met (e.g. when translating a group to a subgroup, the target set (indicator) must not be empty).

$$\begin{tikzcd}\mathcal{T}_1 \arrow[r, "a_1, \dots, a_n"] & \mathcal{T}_2\end{tikzcd}$$

Figure 22: Theory morphisms with assumptions. (IMPS)

Views in MMT, however, are not designed to have assumptions. What we do to circumvent this is to create a copy of the target theory $\mathcal{T}_2$, called $\mathcal{T}_2'$ that includes $\mathcal{T}_2$, but has the assumptions as additional axioms.

$$\begin{tikzcd} & \mathcal{T}_2' \\ \mathcal{T}_1 \arrow[ur, "a_1, \dots, a_n"] & \mathcal{T}_2 \arrow[u, hook] \end{tikzcd}$$

Figure 23: Theory morphisms with axioms. (MMT)

In case the translation has *both*, assumptions and identical source and target theories (as is the case with the example GROUPS->SUBGROUP in Figure 19), it is treated as if it were merely an instance of the former, since a second copy of the copy is not necessary any more.

3.4 Validation of Output

Once the content is translated against the implementation of LUTINS, it is also type-checked by MMT to measure / verify correctness of the translation process. This can help spot potential mistakes or mistranslations.

The goal for this mechanism is that everything that type-checks within the IMPS system also type-checks in the **LF**-implementation. See section 5.1 for the results.

4 Implementation

4.1 Overview

The transformation process we will be discussing in this section starts with IMPS library files[8] and uses several software systems over a number of different steps, which are outlined below.

The individual steps of the translation process are as follows:

- **Generate JSON from IMPS sources**
  The first step is to generate usable JSON representation of the internal data structures of the IMPS system. For this, I modified Li's exporter to export these data structures directly to JSON.

  This has the advantage of being easy to read (for both human and machine) and gives us direct access to the data in the internal data structures, instead of an outdated OMDoc *translation* of those structures.

  This part of the translation will be the topic of Section 4.2.

- **Import and combine JSON and IMPS sources**
  Next, both the generated JSON and the original IMPS library source files need to be parsed and interpreted to create a comprehensive data structure containing all relevant information.

  Parsing from both IMPS library source files and JSON generated from internal data structures, gives the possibility of including more data in the translation, even data that is not represented on a symbolic level within IMPS.

  This will be discussed in Section 4.3.

- **Translate combined structures to MMT/OMDoc**
  The last step is to translate the combined Scala data structures into MMT/OMDoc format using the **LF**-implementation of LUTINS.

  In this form, they can also be type-checked by MMT to verify their correctness. The final OMDoc output is also generated by the MMT system, which always produces OMDoc in the *current* standard of the format.

  For details, see Section 4.4.

Figure 24 gives a high-level schematic view of all involved systems and processes.

[8]Which – like the original IMPS system – are written in the T programming language and are hence often referred to simply as "T-files" in the following sections.

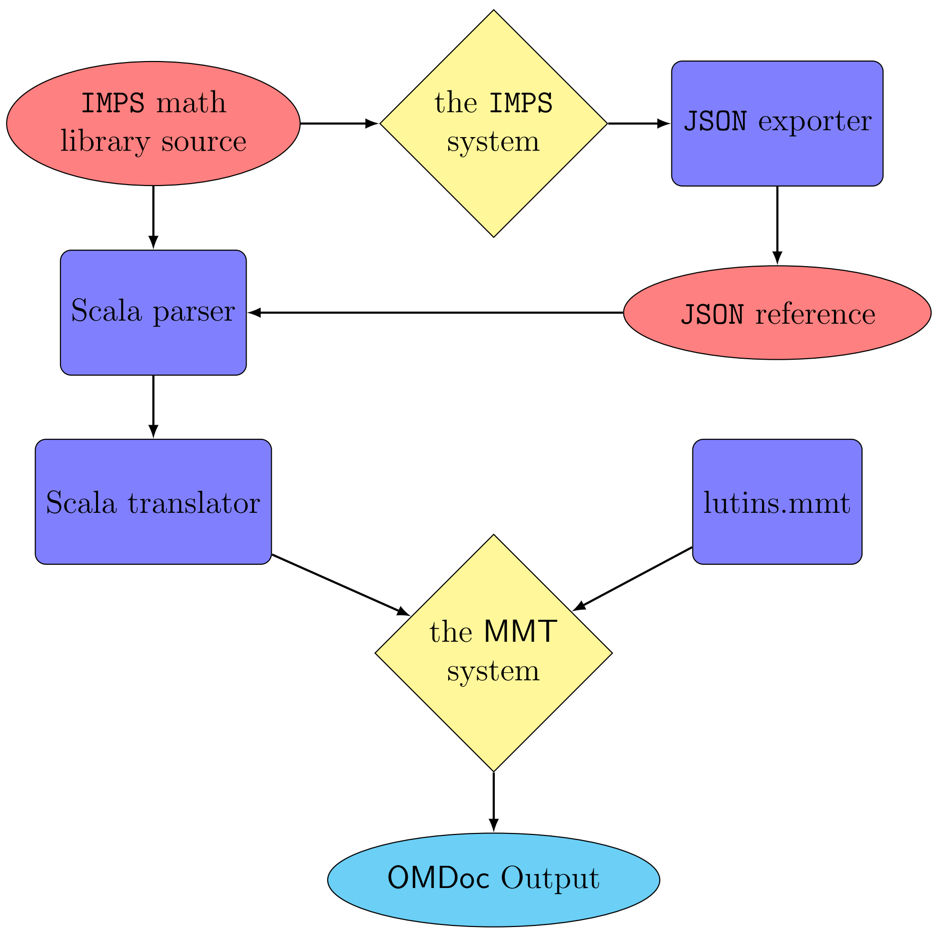

Figure 24: Overview of the architecture.
Red: Source Files, Blue: My contributions,
Yellow: Independent software systems
Cyan: Resulting OMDoc

4.2 IMPS to JSON Export

The first stage in the translation process is to create an easily parseable representation of the relevant internal IMPS data structures. Figure 25 shows the relevant part of the overall architecture.

In [Li02], Li presents a LISP-based exporter (called "imps2omdoc") from IMPS directly to the then-current version of OMDoc, using information directly represented in symbol form in the IMPS system.

I modified this exporter (now called "imps2json") so that it exports to JSON instead, which are then easily readable from my MMT importer extension. This made it a lot easier to import the data that IMPS provides since MMT already has the capability to parse and load JSON.

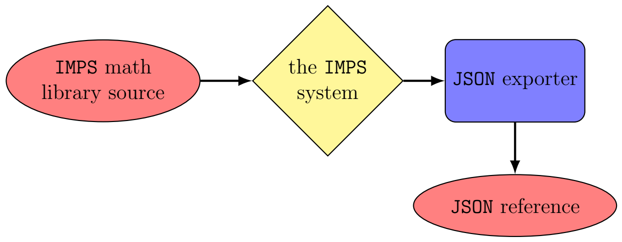

Figure 25: Overview: export from source to JSON

It is also more practical because keeping the old target format would have entailed writing an additional importer for an outdated version of OMDoc, a work-intensive and potentially confusing endeavour.
Additionally, for such an importer to work, one would have to reverse some transformations that Li applied to the data from the internal data structures. Simply exporting the content of the data structures themselves to be read and used later was a lot simpler.

The export into JSON has two major steps:

- **Translation from IMPS to intermediate structures**
  These intermediate data structures are not internal to IMPS, but were introduced in [Li02]. They are constructed to contain all relevant information IMPS has on the mathematical object they represent.

  For this to work, the IMPS system needs to run during the process of translation with all the def-forms loaded that are to be translated. The program then has access to both the internal representation of data in IMPS as well as functions on this data.

- **Translation from intermediate structures to JSON**
  Once the data structures are assembled, they need to be transformed into an easily parseable format. For this task, I chose the *Javascript Object Notation* (JSON) format, since it is easy to read for both humans and machines. In particular, MMT already has the capability to parse and interpret JSON, so no additional work was required.

I use largely the same intermediate structures as [Li02] (with some additions and modifications, see also Appendix B), but completely re-wrote the export into JSON format, since we are only interested the *content* of the internal data structures, in as easy a format to parse as possible, not a translation to something else.

4.2.1 Intermediate Structures

There are five intermediate structures presented in [Li02]. They are implemented as Common LISP record structures with fields storing the relevant information.

- **theorystruc**
  Structure for `IMPS` theories, including all associated data
  Examples: h-o-real-arithmethic, groups, vectorspaces. . .

- **languagestruc**
  Structure for `IMPS` languages,
  Examples: pre-numerical-structures, monoid-language. . .

- **objstruc**
  Structure for axiom, theorem, defined constant and defined atomic sort objects
  Examples: various theorems, functions like `sqrt` and `factorial`, etc.

- **recursivestruc**
  Structure for recursive constants.
  Examples: sum, prod, nn%quotient, . . .

- **translationstruc**
  Structure for translations between `IMPS` theories.
  Examples: complete-partial-order-to-h-o-real-arithmetic, . . .

These intermediate structures were left mostly as they were, with a few additions, where necessary. For example, since we are interested in the precise types of things[9], **objstruc** was expanded by a field for the sort of a given object.

A precise listing of all used intermediate structures and what parts were modified or added can be found in Figures 35 to 39 in Appendix B.

4.2.2 S-Expressions

We talked before about the different ways of representation in which `IMPS` displays mathematical objects to the user (string syntax, etc.). One of the most important features of the `JSON` export mechanism is the export of mathematical expressions in *s-expression syntax* instead of *string syntax.*

For example, consider the axiom `commutative-law-for-addition` of the theory `h-o-real-arithmetic`. In string presentation, it is printed like this:

```
forall(y,x:rr,x+y=y+x)
```

In s-expression syntax, however, this axiom is printed as follows:

```
(forall ((rr y x)) (= (apply-operator + x y)
                      (apply-operator + y x)))
```

[9]For example, the enclosing sorts of atomic sorts defined via def-form. This information is not explicitly given by the user, but automatically inferred by `IMPS`, so it's not available in the `T` source code.

While the string representation might be more familiar to the human eye, s-expressions are considerably easier to parse mechanically and make dealing with binding strength and operator precedence unnecessary.

They also use quasi-constructors by name in the style of any other constructor instead of by notation. This is advantageous because the *notation* of a quasi-constructor is not defined in the same def-form as the quasi-constructor itself. Going by string syntax would require a lot of additional effort.

An additional advantage of the s-expression syntax is that *function application* is made explicit (see above). This greatly simplifies the correct parsing of mathematical expressions.

4.3 Importing and Combining Sources

The next part of the translation process involves parsing and interpreting both the original `IMPS` `T` source files as well as the `JSON` created by `imps2json`. Then, the information from both sources needs to be combined into a new form of intermediate structure that can then be translated into MMT/OMDoc.

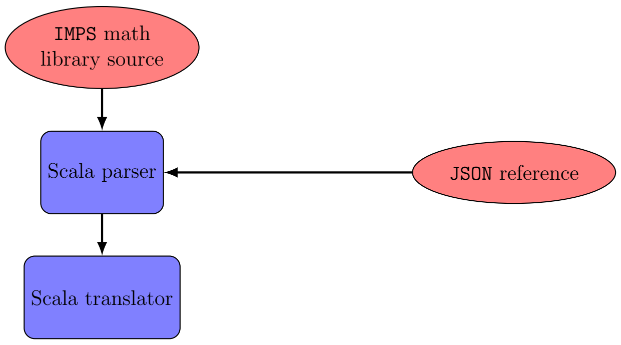

Figure 26: Overview: Integration of `T` and `JSON` sources

This is maybe the most significant difference to [Li02]. Instead of working *only* with the internal `IMPS` data structures, the implementation of a translation from `IMPS` to OMDoc works with *both* the internal `IMPS` data structures and the original `IMPS` source files those data structures are generated from.

This gives us significantly more information to work with, especially some of the information that is not represented on a symbol level in `IMPS`, like proof scripts or defining expressions for quasi-constructors.

4.3.1 Internal Representation

`IMPS` structures

I created an abstract data structure called `TExp` (for `T`-Expression) and added one instance for each kind of mathematical object in `IMPS` (minus those that

```
/* def-theorem
 * Documentation: IMPS manual pgs. 184 ff. */
case class Theorem(name    : String,                         /* Positional argument. Required. */
                   formula : IMPSMathExp,                    /* Positional argument. Required. */
                   lemma   : Boolean,                        /* Modifier argument. Optional. */
                   reverse : Boolean,                        /* Modifier argument. Optional. */
                   thy     : ArgumentTheory,                 /* Keyword Argument, Required */
                   usages  : Option[ArgumentUsages],         /* Keyword Argument, Optional */
                   trans   : Option[ArgumentTranslation],    /* Keyword Argument, Optional */
                   macete  : Option[Macete],                 /* Keyword Argument, Optional */
                   hmthy   : Option[HomeTheory],             /* Keyword Argument, Optional */
                   prf     : Option[Proof],                  /* Keyword Argument, Optional */
                   src     : Option[SourceRef])              /* SourceRef for MMT */
  extends TExp
{
  override def toString: String =
  {
    var str : String = "(def-theorem " + name + "\n  " + formula.toString
    if (lemma) { str = str + "\n  lemma"}
    if (reverse) { str = str + "\n  reverse"}
    str = str + "\n  " + thy.toString
    if (usages.isDefined) { str = str + "\n  " + usages.get.toString }
    if (macete.isDefined) { str = str + "\n  " + macete.get.toString }
    if (hmthy.isDefined)  { str = str + "\n  " + hmthy.get.toString }
    if (prf.isDefined)    { str = str + "\n  " + prf.get.toString }
    str = str + ")"
    str
  }
}
```

Figure 27: Scala data type representing a theorem

were dead or delayed). These data structures collect all relevant information in one place. They are created while parsing the T source code, but also draw on the parsed JSON for things like sorts or mathematical formulae in *s-expression* form.

The MMT system offers a data type called SourceRef (for *Source Reference*). Every structure representing an IMPS object carries one of these source references that carries data about where (that is, file name and line numbers) the original *source* (in our case: the original def-form) of the object is.
If one of these TExps carries another TExp as an argument (for example, theorems carry proofs, both of which have a TExp representation), then the argument TExp will also have a source reference that refines the source reference of its "parent".

Figure 27 shows an example of such a structure (for theorems) along with its .toString method, which can be used to create valid IMPS source code again[10].

IMPS Mathematical Expressions

Mathematical expressions have a different abstract data type called MathExp. The instances of this data type correspond to logical constructors, quasi-constructors

[10]Parsing a def-form followed by printing it via .toString is not guaranteed to give the identity. Some arguments to def-forms can come in any order, and that information is not saved, only what those arguments describe.

```
abstract class IMPSMathExp

case class IMPSMathSymbol(s : String) extends IMPSMathExp {
  override def toString: String = s
}

case class IMPSVar(v : String) extends IMPSMathExp {
  override def toString: String = v
}
...
case class IMPSNegation(p : IMPSMathExp) extends IMPSMathExp {
  override def toString: String = "not(" + p + ")"
}
...
case class IMPSIota(v1 : IMPSVar, s1 : IMPSSort, p : IMPSMathExp) extends IMPSMathExp {
  override def toString: String = "iota(" + v1 + ":" + s1 + "," + p + ")"
}
...
case class IMPSUndefined(s : IMPSSort) extends IMPSMathExp {
  override def toString: String = "?" + s.toString
}
...
case class IMPSLambda(vs : List[(IMPSVar, IMPSSort)], t : IMPSMathExp) extends IMPSMathExp
{
  override def toString: String =
  {
    var str : String = "lambda("
    for ((v, s) <- vs)
    {
      str = str + v.toString
      str = str + ":" + s.toString
      str = str + ","
    }
    str = str + t.toString + ")"
    str
  }
}
...
```

Figure 28: Part of Scala data type(s) that represent IMPS math expressions.

(more on this in Section 4.4.5) and generally follow the rules for expression-building laid out in Figure 2 (on page 10).

There are no source references for MathExps directly, though usually they are wrapped in a TExp that would have one.

As an example, see Figure 28 for a selection of instances of the data type for typical mathematical constructs.
Sorts are implemented in a similar data type, called IMPSSort. This data type does not extend MathExp (although one could make a reasonable case for that) as to make type-errors (e.g. putting a sort where an expression needs to go) easier to detect.

In cases where one needs an expression only as a stand-in for a sort, the implementation falls back on passing an IMPSUndefined of the correct sort, as is common practice in IMPS as well.

An example for the implementation of types of IMPS sorts can be seen in Figure 29.

```
abstract class IMPSSort

case class IMPSAtomSort(s : String) extends IMPSSort
{
  override def toString: String = s
}

...

case class IMPSBinaryFunSort(s1 : IMPSSort, s2 : IMPSSort) extends IMPSSort
{
  override def toString : String = "[" + s1 + "," + s2 + "]"
}
...
```

Figure 29: Part of Scala data type(s) that represent IMPS sorts

4.4 Translation Specifics

The last part of the translation process is taking the intermediate Scala structures and translating them into MMT/OMDoc format. We will discuss the general approach now and explain a few important special cases in the following sections.

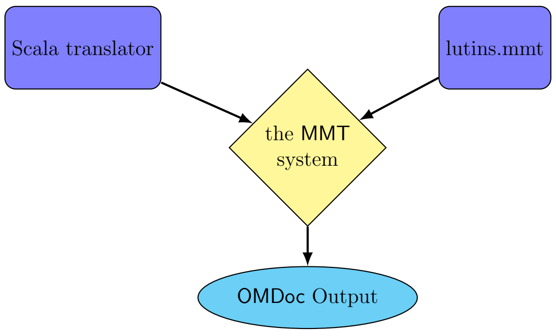

Figure 30: Translation to MMT/OMDoc

Translating against a reference implementation of the underlying logic gives us the possibility of *type-checking* for correctness of the translation, as well as the opportunity to benefit from MMT's simplifier and solver.
It also makes the translation process robust against future changes of the OMDoc format, since the actual export to OMDoc is now done by the MMT system itself.

The birds-eye view on the translation process is that mathematical expressions are translated from their representation in `MathExp` form to an equivalent expression in **LF** using the constructors and constants the implementation of LUTINS (see Appendix C) provides. Compare also Section 3.2.

Constants (like theorems, but also like functions) are added along with their name (if they have one) and their type (which in the case of theorems is best understood as the type of proofs of the theorem) and can be referred to by other elements. Every element is **LF**-type-checked by MMT.

Since the **LF**-constructors often take more arguments than the `IMPS` expressions explicitly provide in some places (e.g. correct maximal sorts), it is sometimes necessary to add unknowns to the translated expression. These are then later solved by the MMT system through inference from the sum of information it has available.

4.4.1 Subsorting

The implementation of LUTINS for MMT (see Appendix C) includes a construct to declare one sort as a subsort of another.

In case of declared subsorts of theories, the enclosing sort is always indicated directly in the same def-form and hence readily available. In case of atomic sorts declared via their own def-form, however, the enclosing sort is never directly mentioned (neither could it be, the def-form does not support that kind of argument). Here, the implementation falls back to the information in the `JSON` output. Since `IMPS` still infers the enclosing sort of the defining lambda expression, from which the enclosing sort of the sort being defined can be recovered with minimal effort[11].

Note that we do not use subtypes of **LF** types, but merely apply an **LF** term *indicating* that one sort is a subsort of another.
This can still be type-checked to ensure correctness, though the information will only be available on the logical level, not on the meta-logical level.

4.4.2 Rewriting Quantifiers to Fixed Arities

Many constructors in LUTINS do not have a fixed arity. Quantifiers like `forall` and `forsome`, connectors like `and` and `or`, the function sort constructor and multiple of the system quasi-constructors all accept lists of arguments of varying length.

MMT, however, does not support flexary operators at time of writing. So, to make it possible to represent these constructors correctly in OMDoc, they are rewritten to fixed arity during translation.

The operators in MMT's version of LUTINS are of a fixed arity (usually connectors are binary and quantifiers can only bind a single variable), but can be combined with each other to reconstruct the original input term.

So, for example, an expression like the following (from `SURJECTIVE-ON-LEMMA`)

[11]The sort of the defining lambda-expression is of the form $[\alpha, \texttt{prop}]$ since `IMPS` uses predicate subtyping for sorts defined via def-form. From this, we learn that the enclosing sort we are looking for is α.

$$\forall a : sets[ind_1], b : sets[ind_2], f : (ind_1 \to ind_2), x : ind_1.$$
$$((surjective\_on\_q(f, a, b) \wedge (x \in a)) \Rightarrow (f(x) \in b))$$

would be rewritten into

$$\forall a : sets[ind_1].\forall b : sets[ind_2].\forall f : (ind_1 \to ind_2).\forall x : ind_1.$$
$$((surjective\_on\_q(f, a, b) \wedge (x \in a)) \Rightarrow (f(x) \in b))$$

during translation to OMDoc.

Naturally, to make the types fit these adapted constructors[12], function sorts of `IMPS` function terms need to be rewritten in a similar manner as well.

4.4.3 Literals

`IMPS` languages allow for three distinct types of literals, that can be added via the `extensible` keyword along with a so-called "numeric type". These languages may contain an infinite amount of constants in one-to-one correspondence with a sort of the theory that is also specified as an argument.

The three categories of literals / numeric types present in the library are:

- Integers
  Keyword: `*integer-type*`
  Examples: `0, -1, 2, ...`
- Rationals
  Keyword: `*rational-type*`
  Examples: `[1/2], [-1/3], [2/3], ...`
- Octets
  Keyword: `*octet-type*`
  Examples: `32#8, 64#8, 128#8, ...`

MMT supports arbitrary literals that do not depend on a fixed set of constants that the framework needs to pre-commit to (see [Raba]).
The implementation of LUTINS includes one sort (of kind ι) each for the three types of literals, along with appropriate rules for parsing and typing them in the importer system.

If a language is translated that has the `extensible` keyword and assigns a numeric type n to a given sort α (compare [FGT98], pgs. 172 to 174), a subsort judgement

[12]Also because the constructor for function sort creation in the **LF** implementation is of fixed arity as well.

```
<opaque format="text">Opaque proof of theorem
        DEFINEDNESS-OF-DANGLING-CONDITIONALS
(proof
  (simplify))</opaque>
```

Figure 31: Generated opaque proof data for example theorem

$n \preceq \alpha$ is added to the translation. That way, all literals are always of the correct sort in a given theory.

4.4.4 Opaque Data

Every bit of information gathered from the `T` source code is represented in the generated OMDoc in one way or another. If one cannot represent it in a meaningful and formal way directly (e.g. processors, proofs or macetes, but also arguments to certain def-forms like `usages`), it is instead translated as so-called *opaque data*. This means that the information is still added to the resulting OMDoc file, but without typing or mathematical structure. In Figure 31, you can see the OMDoc representation of a simple theorem's proof.

Proofs were not considered at all in [Li02] because they don't have an internal representation in `IMPS`. The additional import of the `T` source files allowed us to at least include the scripts themselves as opaque data, even if not as formal mathematical properties.

4.4.5 Quasi-Constructors

The translation of quasi-constructors turned out to be quite challenging. As Li states in [Li02], the corresponding lambda expressions for quasi-constructors are not represented as symbols in `IMPS` and can therefore not be translated into `JSON` directly, like other expressions.
However, user-defined quasi-constructors are used *extensively* throughout the source. A survey of the `t` source files and the `JSON` output identified 33 quasi-constructors used hundreds of times within the `imps-math-library`.
This shows clearly that any serious effort to translate this code base would be incomplete without a rigorous treatment of quasi-constructors.

The solution I decided on was to represent each quasi-constructor that appears in the section of the library we focus on as an instance of an abstract Scala data type with an appropriate number of `MathExp`s as arguments and also added a parsing rule for each one.

```
(def-quasi-constructor I-IN
  "lambda(x:uu,a:sets[uu], #(a(x)))"
  (language indicators)
  (fixed-theories the-kernel-theory))
```

Figure 32: T code defining the “i-in” quasi-constructor

```scala
def removeQCs(input : IMPSMathExp, addCs : List[IMPSMathExp]) : IMPSMathExp =
{
    input match
    {
      ...
      case IMPSQCIn(e1_u,e2_u) =>
      {
        // "lambda(x:uu,a:sets[uu], #(a(x)))"
        val e1    : IMPSMathExp = removeQCs(e1_u,addCs)
        val e2    : IMPSMathExp = removeQCs(e2_u,List(e1) :::addCs)

        val x_var = (freshVar("x",List(e1,e2)          :::addCs), IMPSAtomSort("uu"))
        val a_var = (freshVar("a",List(e1,e2,x_var._1) :::addCs), IMPSSetSort(IMPSAtomSort("uu")))

        val inner  : IMPSMathExp = IMPSIsDefined(IMPSApply(a_var._1,List(x_var._1)))
        val lambda : IMPSMathExp = IMPSLambda(List(x_var,a_var), inner)

        IMPSApply(lambda,List(e1,e2))
      }
      ...
    }
}
```

Figure 33: Example Scala code that eliminates quasi-constructors

During translation of the Scala structures to OMDoc, all quasi-constructors are resolved when they are encountered. See figures 32 and 33 for the definition of a quasi-constructor and the Scala code that resolves it during translation.

I also experimented with resolving the quasi-constructors directly when parsed. However, this lead to unreasonably large and complicated expressions, especially for expressions that used multiple quasi-constructors or expressions using quasi-constructors that use other quasi-constructors.
The solution I ended up settling on had a much clearer representation of the intermediate Scala structures, which turned out helpful for debugging purposes.

This act of manual translation and cherry-picking might suffer from a certain lack of elegance and universality, but was unavoidable to be able to properly translate the actual mathematical statements in the IMPS library at all. Since the defining expressions for quasi-constructors are not represented in the internal IMPS data structures, imps2json can not export any s-expressions for them.

5 Discussion

5.1 Results

`JSON` export

The `imps2json` exporter that I built on top of Li's original `imps2omdoc` exporter works reliably and produces valid `JSON` output, easily interpretable by MMT's `JSON`-importer and the human eye, for all relevant information in the `imps-math-library`.

Parsing Mathematical Expressions

All mathematical expressions in the `imps-math-library` can be parsed and interpreted, including (hand-implemented) quasi-constructors. Hence, the math-parser is feature complete.

Parsing `T` Files

The implementation can parse all def-forms that are not either *dead* or *delayed* (compare section 3.3.1), i.e. all that occur in the `IMPS` `foundation`-section.

Translating to MMT/OMDoc

All theories from the `foundation` section and their components can be translated into OMDoc. Most of them can be translated meaningfully, with the exceptions of proof scripts, macetes, processors and a few minor arguments to `IMPS` def-forms, that need to be translated as opaque data.

All translated expressions successfully type-check in **LF** with two categories of exceptions:

- ι-representations of recursive constants
- Expressions involving quasi-constructors

Theory Graph

I also generated an MMT theory graph from the produced OMDoc, that can be viewed with **tgview**, an interactive theory graph viewer ([RKM17]) developed by the KWARC group.
The graph resulting the translation of the `foundation` section can be seen in Figure 34.

Software Sources

All software that is mentioned in this thesis is available online:

- **imps2json**
  IMPS to JSON exporter, including original IMPS source files, is available at:
  https://gl.mathhub.info/IMPS/theories
- **MMT extension**
  The development of the MMT extension for can be found on this branch:
  https://github.com/UniFormal/MMT/tree/imps
  Some helpful tutorials can be found at the following address:
  https://uniformal.github.io/doc/tutorials/
- **MMT archive**
  An MMT archive with the IMPS files for `foundation` (no JSON) is at:
  https://gl.mathhub.info/IMPS/imps
- **Generated OMDoc**
  Generated OMDoc files for the `foundation` section can be inspected at:
  https://gl.mathhub.info/IMPS/foundation

Interpretation

The translation process presented in this thesis is not complete and the generated output is not yet 100% correct. However, the most crucial problems have been solved:

Note that, although the `foundation` section is only a small part of the wider `imps-math-library` section, it already includes 16 out of 26 (61.5%) kinds of def-forms. Out of the 2037 individual def-forms that appear in the entire library, the implementation supports 1918 (94.2%) already (compare Figure 11).

Note also, that the parser for mathematical expressions in s-expression form is already complete. Adding support for a new kind of def-forms now "merely" requires implementing a parser and finding a suitable MMT equivalence of the term.

I also expect the effort to correct the parts of the translation, that currently produce non-type-checking constants, to be minimal.

Given these results, one can be optimistic for a complete import of the IMPS theory library in the near future.

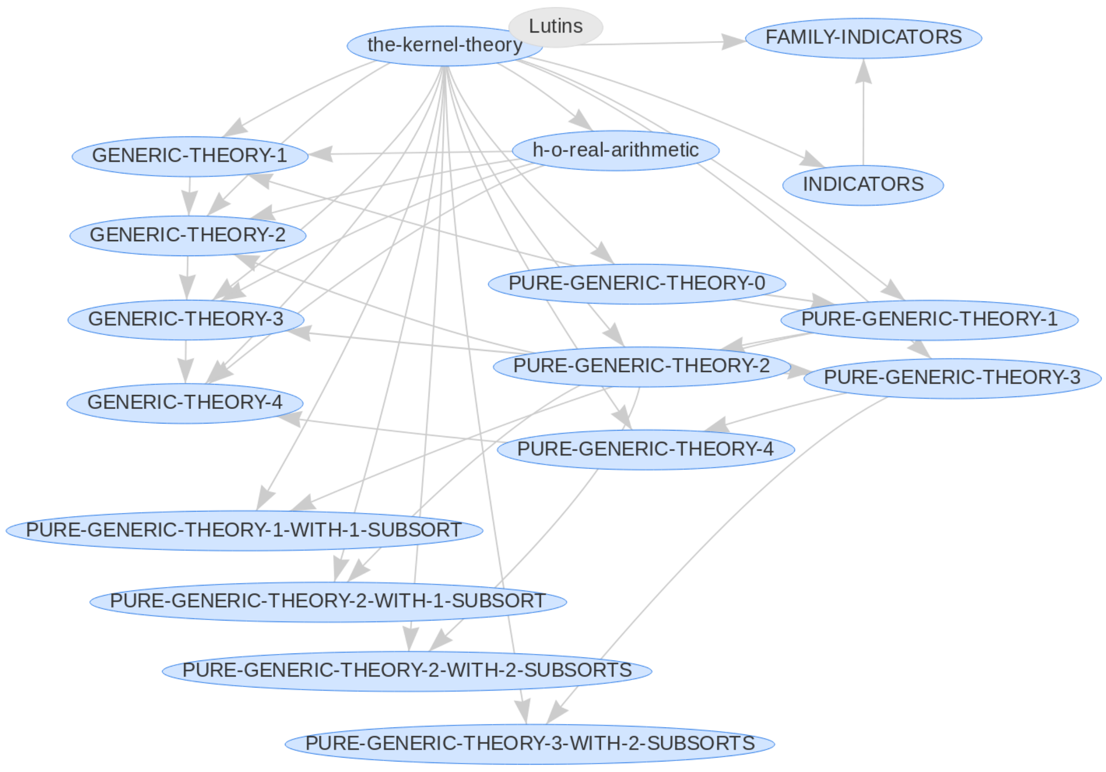

Figure 34: Theory Graph of the `foundation` section

5.2 Future Work

While the results for this thesis are promising, a lot of necessary and possible future work remains open:

The first goal would be to refine the implementation so that *all* the structures in the `foundation` section type-check successfully. This crucial step seems achievable in the light of the results, even in short to medium time frames.

After that, the next obvious step is to extend the translation to include all of the `imps-math-library`, instead of only the `foundation` section.
This will require a significant extension of the `T`-parser as well as the translation process. It might also require to make use of some of MMT's more advanced features like parametric theories or **LF** subtyping (see also below).

A "deeper" structural representation of the `IMPS` subtyping information would also be beneficial. **LF** does not support subtyping, but an extension to **LF** might[Koh+17a]. The translation to OMDoc could benefit from this feature if we would fix that extension instead of **LF** itself.

It also would be very beneficial to have a more structured interpretation of proofs and macetes.

For this to be possible, the **LF**-implementation of LUTINS would probably need to be adapted to support the proof commands and their combinators, as it currently has no support for these primitives.
There is also a significant body of supporting machinery that would need to be added, a process that promises to be non-trivial and time-consuming.

An additional avenue of further research appears obvious. Like other importers into MMT (such as the one from [Koh+17b]), the `IMPS` importer could be extended by an `IMPS` *exporter*, translation from MMT/OMDoc back to `IMPS` source files. Even though the `IMPS` system itself is not in use or development any more, it could bring some insights about the translation process itself, to test if an import followed by an export, would produce identical theories in `IMPS`.

5.2.1 Possible Additions to MMT

In the course of implementation, I came across some features that would have been useful, but that MMT does presently not have:

1. **MetaData for opaque elements**
   The implementation includes a lot of information (processors, macetes, proofs, ...) as *opaque* data in the generated OMDoc files. Opaque elements in MMT currently can not be associated with metadata, such as source references.

   This means the source references and other available metadata for the categories of data mentioned above are forgotten during translation, making the translation less complete.

   Extending MMT by allowing opaque elements to carry metadata would remedy this situation.

2. **Structures across multiple files**
   Currently, MMT assumes that all relevant definitions for a given theory are available in the same file as the theory definition itself. Not so, for example, for `IMPS`, where theorems and definitions of theories like `h-o-real-arithmetic` are spread out over a multitude of files.

   It's not too difficult to work around this, but it would be convenient if MMT supported structures that are split over multiple files natively to reduce doubled effort.

5.3 Conclusion

In this thesis, I endeavoured to make progress on "rescuing" the theory library of the IMPS system, i.e. to make the formalised knowledge contained therein available to mathematical knowledge management tools and saving it from the dangers of inevitable bitrot.

For this, I built a pipeline of software systems that can translate the most important elements of the library into OMDoc format and can also be extended in the future to translate the whole library without too much effort.

I used *both*, a JSON representation of the internal IMPS data structures, generated by adapting Li's `imps2omdoc` exporter, but also the original source files of the theory library as input.
The data from both information sources was then corroborated and integrated, before being translated against my **LF** implementation of the underlying logic LUTINS.

The resulting translation is then type-checked by MMT to verify correctness and used to generate valid and up-to-date OMDoc, which is available for use by external knowledge management systems.

This translation process provides an effective way to generate useful and usable data from the mathematical library of the IMPS system. It serves as a solid proof-of-concept as well as a first step towards future efforts.

We have seen the success of the two-pronged approach of using both information sources mentioned above, allowing a more complete and more accurate representation in the resulting OMDoc, as well as supplying useful metadata, such as source references.

Interfacing with the MMT system for generating OMDoc instead of generating it directly turned out to be an important part of the process. Type-checking against an implementation of LUTINS in **LF** gives an additional level of certainty about the correctness of the translation.

Making the IMPS library available in an accessible and machine-friendly format opens up new possibilities for possible uses. For example, one could use the generated OMDoc to compare similar mathematical objects in different prover environments ore use the objects IMPS supplies as reference implementations.

Structured mathematical documents could also play a role in education, be it about the field of provers themselves or just the mathematics they talk about.

Or maybe parts of the knowledge this saved can be useful to a bigger project in the future (be it a proof effort or a new system entirely). Having this knowledge available has many possible uses.

References

[Bag17] Joan Bagaria. “Set Theory”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Winter 2017. Metaphysics Research Lab, Stanford University, 2017.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. CoqArt: The Calculus of Inductive Constructions*. Springer, 2004. URL: `http://www.labri.fr/perso/casteran/CoqArt/`.

[Chu40] Alonzo Church. “A Formulation of the Simple Theory of Types”. In: *Journal of Symbolic Logic* 5 (1940), pp. 56–68.

[Cod+11] Mihai Codescu et al. “Project Abstract: Logic Atlas and Integrator (LATIN)”. In: *Intelligent Computer Mathematics*. Ed. by James Davenport et al. LNAI 6824. Springer Verlag, 2011, pp. 289–291. ISBN: 978-3-642-22672-4. URL: `https://kwarc.info/people/frabe/Research/CHKMR_latinabs_11.pdf`.

[End72] H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.

[Far15] William Farmer. *IMPS System Description*. 2015. URL: `https://github.com/theoremprover-museum/imps` (visited on 03/25/2018).

[Far90] William M. Farmer. “A partial-function version of Church’s simple theory of types”. In: *Journal of Symbolic Logic* 55 (1990), pp. 1269–1291.

[Far93a] William M. Farmer. “A simple type theory with partial functions and subtypes”. In: *Annals of Pure and Applied Logic* 64 (1993), pp. 211–240.

[Far93b] William M. Farmer. “Theory Interpretation in Simple Type Theory”. In: *HOA’93, an International Workshop on Higher-order Algebra, Logic and Term Rewriting*. LNCS 816. Amsterdam, The Netherlands: Springer Verlag, 1993.

[FGT92] William M. Farmer, Joshua Guttman, and Javier Thayer. “Little Theories”. In: *Proceedings of the 11th Conference on Automated Deduction*. Ed. by D. Kapur. LNCS 607. Saratoga Springs, NY, USA: Springer Verlag, 1992, pp. 467–581.

[FGT98] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. *The IMPS 2.0 User’s Manual*. 1st ed. The MITRE Corporation. Bedford, MA 01730 USA, Jan. 1998.

[Gut91] J.D. Guttman. *A proposed interface logic for verification environments*. Tech. rep. The MITRE Corporation, 1991.

[HHP93a] R. Harper, F. Honsell, and G. Plotkin. “A framework for defining logics”. In: *Journal of the Association for Computing Machinery* 40.1 (1993), pp. 143–184.

[HHP93b] Robert Harper, Furio Honsell, and Gordon Plotkin. “A framework for defining logics”. In: *Journal of the Association for Computing Machinery* 40.1 (1993), pp. 143–184.

[Iac09] Alin Iacob. *Reasoning about Theory Morphisms*. Bachelor’s Thesis. 2009. URL: `https://svn.eecs.jacobs-university.de/svn/eecs/archive/bsc-2009/aiacob.pdf`.

[IKR11] Mihnea Iancu, Michael Kohlhase, and Florian Rabe. *Translating the Mizar Mathematical Library into OMDoc format*. KWARC Report. Jacobs University Bremen, 2011. URL: `http://uniformal.github.io/doc/applications/LATIN/docs/Mizar2OMDoc-Report.pdf`.

[Koh+17a] Michael Kohlhase et al. “Knowledge-Based Interoperability for Mathematical Software Systems”. In: *MACIS 2017: Seventh International Conference on Mathematical Aspects of Computer and Information Sciences*. Ed. by Johannes Blömer, Temur Kutsia, and Dimitris Simos. LNCS 10693. Springer Verlag, 2017, pp. 195–210. URL: `https://github.com/OpenDreamKit/OpenDreamKit/blob/master/WP6/MACIS17-interop/crc.pdf`.

[Koh+17b] Michael Kohlhase et al. “Making PVS Accessible to Generic Services by Interpretation in a Universal Format”. In: *Interactive Theorem Proving*. Ed. by Mauricio Ayala-Rincón and César A. Muñoz. Vol. 10499. LNCS. Springer, 2017. ISBN: 978-3-319-66107-0. URL: `http://kwarc.info/kohlhase/submit/itp17-pvs.pdf`.

[Koh16] Michael Kohlhase. “An Open Markup Format for Mathematical Documents OMDoc [Version 1.6 (pre-2.0)]”. Draft Specification. 2016. URL: `https://github.com/OMDoc/OMDoc/blob/master/doc/spec/main.pdf`.

[KR14] Cezary Kaliszyk and Florian Rabe. “Towards Knowledge Management for HOL Light”. In: *Intelligent Computer Mathematics 2014*. Conferences on Intelligent Computer Mathematics. (Coimbra, Portugal, July 7–11, 2014). Ed. by Stephan Watt et al. LNCS 8543. Springer, 2014, pp. 357–372. ISBN: 978-3-319-08433-6. URL: `http://kwarc.info/frabe/Research/KR_hollight_14.pdf`.

[Kra+86] D. Kranz et al. “ORBIT An optimizing compiler for scheme”. In: *Proceedings of the* SIGPLAN *’86 Symposium on Compiler Construction.* 1986.

[KS10] Alexander Krauss and Andreas Schropp. “A Mechanized Translation from Higher-Order Logic to Set Theory”. In: *Interactive Theorem Proving: First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings.* Ed. by Matt Kaufmann and Lawrence C. Paulson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 323–338. ISBN: 978-3-642-14052-5. DOI: 10.1007/978-3-642-14052-5_23. URL: http://dx.doi.org/10.1007/978-3-642-14052-5_23.

[KW10] Chantal Keller and Benjamin Werner. “Importing HOL Light into Coq”. In: *ITP.* Vol. 6172. Lecture Notes in Computer Science. Springer, 2010, pp. 307–322.

[LATIN] *The LATIN Logic Atlas.* URL: https://gl.mathhub.info/MMT/LATIN (visited on 06/02/2017).

[Li02] Yan Li. “IMPS to OMDoc Translation”. Bachelor’s Thesis. McMaster University, Aug. 2002.

[Mon76] J.D. Monk. *Mathematical Logic.* Springer-Verlag, 1976.

[OAF] *The OAF Project & System.* URL: http://oaf.mathhub.info (visited on 04/23/2015).

[OS06] Steven Obua and Sebastian Skalberg. “Importing HOL into Isabelle/HOL”. In: *Automated Reasoning: Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006. Proceedings.* Ed. by Ulrich Furbach and Natarajan Shankar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 298–302. ISBN: 978-3-540-37188-5. DOI: 10.1007/11814771_27. URL: http://dx.doi.org/10.1007/11814771_27.

[Raba] Florian Rabe. “Generic Literals”. URL: http://kwarc.info/frabe/Research/rabe_literals_14.pdf.

[Rabb] Florian Rabe. “MMT: A Foundation-Independent Logical Framework”. URL: https://kwarc.info/people/frabe/Research/rabe_mmtsys_18.pdf.

[Rab14] Florian Rabe. “How to Identify, Translate, and Combine Logics?” In: *Journal of Logic and Computation* (2014). DOI: 10.1093/logcom/exu079.

[RAM88] J. A. Rees, N. I. Adams, and J. R. Meehan. *The T Manual.* Computer Science Department, Yale University. 1988.

[RKM17] Marcel Rupprecht, Michael Kohlhase, and Dennis Müller. “A Flexible, Interactive Theory-Graph Viewer”. In: *MathUI 2017: The 12th Workshop on Mathematical User Interfaces*. Ed. by Andrea Kohlhase and Marco Pollanen. 2017. URL: `http://kwarc.info/kohlhase/papers/mathui17-tgview.pdf`.

[Rus05] Bertrand Russell. “On Denoting”. In: *Mind (New Series)* 14 (1905), pp. 479–493.

[Sho67] J. R. Shoenfield. *Mathematical Logic*. Addison-Wesley, 1967.

A List of Figures

B Details on Intermediate Structures

These are the precise types and descriptions of the intermediate structures used in the `JSON` export step, see Section 4.2.1.

Additions to these structures by us are shaded. Compare also Section 4.4.1 in [Li02].

Note: *The data type* ***event*** *is internal to* `IMPS`*. Events are translated according to whatever data the event describes (events, axioms, constants, . . .).*

Field	**Type**	**Description**
`theoryname`	String	The name of the theory
`cotheory`	List[**theorystruc**]	A list of intermediate structures of all component theories of this theory.
`language`	**languagestruc**	The intermediate structure of this theory's language.
`axioms`	List[**objstruc**]	A list of intermediate structures of all axioms of this theory.
`defconsts`	List[**objstruc**]	A list of intermediate structures of this theory's defined constants
`def-recursive-consts`	List[**recursivestruc**]	A list of intermediate structures of this theory's recursive constants.
`defsorts`	List[**objstruc**]	A list of intermediate structures of this theory's defined sorts.
`theorems`	List[**objstruc**]	A list of intermediate structures of this theory's theorems.
`events`	List[**event**]	A list of all the events of this theory.
`translation`	List[**translationstruc**]	A list of intermediate structures of translations where this theory is the target.

Figure 35: **theorystruc**: Data type to represent `IMPS` theories

Field	**Type**	**Description**
`langname`	String	The name of the language.
`embedlang`	List[**languagestruc**]	A list of intermediate structures of the language's embedded languages.
`sorts`	List[String]	A list of the sorts of the language.
`const`	List[String]	A list of the constants of the language.
`primitive-sorts`	List[String]	A list of primitive sorts of the language.
`primitive-consts`	List[String]	A list of primitive constants of the language.
`languages`	List[String]	A list of the names of the language's embedded languages.

Figure 36: **langstruc**: Data type to represent `IMPS` languages

Field	**Type**	**Description**
`objname`	String	The name of the object.
`sorting`	String	The sort of the object.
`formula`	String	The string representation of the object.
`f-sexp`	String	The s-expression representation of the object.
`theoryname`	String	Name of the home theory of the object.
`usagelist`	List[String]	List of usages for the object.

Figure 37: **objstruc**: Data type to represent `IMPS` objects

Field	**Type**	**Description**
`objname`	List[String]	The names of the objects.
`functional-list`	List[String]	A list of string presentations of the objects' functionals
`defining-expr-list`	List[String]	A list of string representations of the objects' defining expressions (Least Fixed Point)
`defining-sexp-list`	List[String]	A list of s-expressions of the objects' defining expressions (Least Fixed Point)
`sortings`	List[String]	A list of sorts of the object.
`theoryname`	String	The home theory of the objects.
`usagelist`	List[String]	A list of usages for the objects

Figure 38: **recursivestruc**: Data type for recursively defined constants

Field	**Type**	**Description**
`name`	String	The name of the translation.
`source-theory`	String	The source theory of the translation.
`target-theory`	String	The target theory of the translation.
`fixed-theories`	List[String]	A list of the names of the fixed theories of the translation.
`assumptions`	List[String]	A list of string representations of the assumptions of the translation.
`assumptions-sexp`	List[String]	A list of s-expressions of the assumptions of the translation.
`sort-pairs`	List[String]	A list of string representations of the sorts paired by the translation.
`sort-pairs-sexp`	List[String]	A list of s-expressions of the sorts paired by the translation.
`constant-pairs`	List[String]	A list of string representations of the constants paired by the translation.
`constant-pairs-sexp`	List[String]	A list of s-expressions of the constants paired by the translation.
`interpretation?`	String	Denotes whether the translation is an interpretation or not.

Figure 39: **translationstruc**: Data type to represent `IMPS` translations

C lutins.mmt, LUTINS implemented in LF

Note:
This representation of MMT syntax foregoes the special unicode delimiters for printability.

```
namespace http://latin.omdoc.org/foundations/lutins

import rules scala://imps.mmt.kwarc.info

/T   The underlying logic for IMPS is called LUTINS (Logic of Undefined
Terms for Inference in a Natural Style). This Higher-Order-Logic
incorporates partial functions, undefinedness and subtyping.

More details can be found at http://imps.mcmaster.ca/manual/node12.html

theory Lutins : http://cds.omdoc.org/urtheories?LF =

  rule rules?IntLiterals
  rule rules?RatLiterals

    # Type System

    /T Lutins types (which are technically just maximal sorts)
       are represented as LF terms of type "tp".
    tp : type

    /T Sorts refine types. They cannot be empty, but they may
       overlap. A sort that refines type A is represented as
       an LF term of the type "sort A"
    sort : tp → type

    /T Every expression can have many sorts, but only one type.
       That type (i.e. maximal sort) is the unique type b such that the expression's
       sort a is ⪯ b. with respect to the enclosing sort relation ⪯.
   exp : {A : tp} sort A → type # exp 2 prec -1

   /T Function types are only needed to keep track of the type of function sorts.
   funType: tp → tp → tp

   /T Sorts are formed from the base types and by using function sorts.
      All functions are partial.
      The function sort constructor is actually flexary.
   top: {A : tp} sort A # ' 1
   fun: {A, B} sort A → sort B → sort (funType A B) # 3 ⇒ 4 prec 50

   /T LUTINS has primitive booleans, often called ⋆ in documentation.
   boolType : tp
   bool = ' boolType

   /T λ-abstractions
   lambda : {A, B, α : sort A, β : sort B}
      (exp α → exp β) → exp (α  ⇒  β) # λ 5

   /T You can apply a function to any argument that refines the given type.
      This will always be well-sorted but may still be undefined.
   apply : {A, B, α : sort A, γ : sort A, β : sort B}
        exp (α  ⇒  β) → exp γ → exp β # 6 @ 7 prec 100

   /T The description operator ι returns the _unique_ object (of kind ι)
      if such an object exists. Otherwise, it is undefined.
   description_i : {A, α : sort A} (exp α → exp bool) → exp α # ι 3

   /T There's a second unique description operator for kind ⋆,
      though little supported and hardly ever used. The difference being,
      when this predicate isn't uniquely witnessed, the expression
      is not undefined, but actually false.
   description_p : {α : sort boolType} (exp α → exp bool) → exp α # ι⋆ 2

   /T An ordinary if-then-else operator. Branches can be undefined.
   ifthenelse : {A, α : sort A} exp bool → exp α → exp α → exp α # if 3 then 4 else 5

   /T A generic undefined Term, independent of sort.
      Equal to false in ⋆, since booleans must always be defined.
    Undefined functions mapping into ⋆ are treated as if the returned false.
 undefined : {A, α : sort A} exp α# ? 2

  # Logic
  /T Standard stuff, no surprises here.

 thetrue  : exp bool # TT
 thefalse : exp bool # FF
 not      : exp bool → exp bool # ¬ 6 prec 40
 and      : exp bool → exp bool → exp bool # 1 ∧ 2 prec 30
 or       : exp bool → exp bool → exp bool # 1 ∨ 2 prec 30
 implies  : exp bool → exp bool → exp bool # 1 ⇒ 2 prec 30
 iff      : exp bool → exp bool → exp bool # 1 ⇔ 2 prec 30

 if_form  : exp bool → exp bool → exp bool → exp bool # ifform 1 then 2 else 3
          = [p,t,e] if p then t else e

 forall   : {A, α : sort A} (exp α → exp bool) → exp bool # ∀ 3 prec 20
 forsome  : {A, α : sort A} (exp α → exp bool) → exp bool # ∃ 3 prec 20

 /T A theorem is a (boolean) expression T for which there exists a proof
    (that is, an LF-term of type ⊢ T).
 thm : exp bool → type # ⊢ 1 prec -1

 /T Equality takes two expressions of the same type, though
    not necessarily the same sort. Evaluates to false if either
    argument is undefined. If this is inconvenient, consider quasi-equality.
 equals   : {A, α : sort A, β : sort A} exp α → exp β → exp bool # 4 = 5 prec 50

 isdefinedin : {A, α : sort A} exp α → sort A → exp bool # 3 ↓ 4 prec 95

 isdefined   : {A, α : sort A} exp α → exp bool # 3 ↓ prec 100
             = [A,a,x] x↓a

 /T α being a subsort of β means everyting in α is also defined in β.
 subsort     : {A} sort A → sort A → exp bool # 2 ⪯ 3 prec 50
             = [A,a,b] ∀ [x : exp a] x↓b

 # Quasi-Constructors

 /T Quasi-Equality also holds, even if either (!) term is undefined.
 quasiequals : {A, α : sort A, β : sort A} exp α → exp β → exp bool # 4 ≃ 5 prec 50
             = [A,a,b,x,y] (x↓ ∨ y↓) ⊃ x = y

 /T Predicate for checking a function for totality (actually flexary).
 total       : {A, B, α : sort A, β : sort B} exp (α ⇒ β) → exp bool # total 5
             = [A,B,a,b,f] ∀[x: exp a] ((f@x)↓)

 /T A meta-predicate that checks if a predicate is empty (actually flexary)
 nonvacuous  : {A, α : sort A} exp (α ⇒ bool) → exp bool # nonvacuous 3
             = [A,a,p] ∃[x : exp a]p@x

 /T A predicate for testing if a function is defined at an argument (actually flexary)
 domain      : {A, B, α : sort A, β : sort B} exp (α ⇒ β) → exp (α ⇒ bool) # domain 5
             = [A,B,a,b,f] λ [x] (f@x)↓

# Individuals

/T Like booleans, the type of individuals (and also a unit sort) are built-in.
indType : tp
ind : sort indType = ' indType

anIndividual : exp ind

unitsort : sort indType = ' indType
unitsortElem : ⊢ ∀ [x : exp unitsort] x = anIndividual

/T Subsets of a sort are partial functions into the unit sort.
sets : {A, α : sort A} sort (funType A indType) # sets [ 2 ]
     = [A, a] (a ⇒ unitsort)

# Numeric Types
integerType  : sort indType # ℤ
rationalType : sort indType # ℚ
octetType    : sort indType # 𝕆
```

D The IMPS Math Library

```
(def-section imps-math-library
    (component-sections
        pre-reals
        foundation
        foundation-supplements
        number-theory
        machine-arithmetic
        calculus-over-the-reals
        pairs
        sequences
        binary-relations
        iterate
        advanced-cardinality
        schroeder-bernstein-theorem-1
        counting-theorems-for-groups
        group-interpretations
        real-arithmetic-exponentiation
        auxiliary-monoids
        groups-as-monoids
        metric-space-subspaces
        metric-space-continuity
        abstract-calculus
        binomial-theorem
        schroeder-bernstein-theorem-2)
    (files
        (imps theories/algebra/quotient-structures)
        (imps theories/metric-spaces/ptwise-continuous-mapping-spaces)
        (imps theories/normed-spaces/normed-groups)
        (imps theories/normed-spaces/real-derivatives)
        (imps theories/partial-orders/intermediate-value-thm)
        (imps theories/partial-orders/more-convergence-and-order)
        (imps theories/partial-orders/linear-order)
        (imps theories/reals/more-applications)
        (imps theories/reals/additional-arithmetic-macetes)))
```

Figure 40: The section for the imps-math-library

```
== BUILDING DEPENDENCY TREE ==

Target section: foundation

> foundation
  | pure-generic-theories-with-subsorts.t
  | reals-supplements.t
  | some-obligations.t
  > generic-theories
    | generic-theories.t
    > basic-real-arithmetic
      | arithmetic-macetes.t
      | some-lemmas.t
      | number-theory.t
      > reals
        | reals.t
        | some-elementary-macetes.t
        | arithmetic-strategies.t
    > pure-generic-theories
      | pure-generic-theories.t
      | iota.t
  > mappings
    | mappings.t
    | mapping-lemmas.t
    | inverse-lemmas.t
    > indicators
      | indicators.t
      | indicator-lemmas.t
    > pure-generic-theories
      | pure-generic-theories.t
      | iota.t
    > generic-theories
      | generic-theories.t
      > basic-real-arithmetic
        | arithmetic-macetes.t
        | some-lemmas.t
        | number-theory.t
        > reals
          | reals.t
          | some-elementary-macetes.t
          | arithmetic-strategies.t
      > pure-generic-theories
        | pure-generic-theories.t
        | iota.t
```

Figure 41: The dependency tree for the `foundation` section.