Translating the IMPS theory library to MMT/ OMDoc

Jonas Betzendahl
jonas.betzendahl@fau.de

2018 – 08 – 14

Motivation

Why would we want to “rescue” an old library?

Motivation

Why would we want to “rescue” an old library?

- Useful mathematical knowledge (historical)
- Interoperability (practical)

A closer look at LUTINS

The basics of LUTINS

LUTINS[1] is the underlying logic of the IMPS theorem prover, both of which were developed in the 1980s and 1990s at the MITRE Corporation.

The name is shorthand for **L**ogic of **U**ndefined **T**erms for **I**nference in a **N**atural **S**tyle.

[1] pronounced as in French

The basics of LUTINS

LUTINS[1] is the underlying logic of the `IMPS` theorem prover, both of which were developed in the 1980s and 1990s at the MITRE Corporation.

The name is shorthand for **L**ogic of **U**ndefined **T**erms for **I**nference in a **N**atural **S**tyle.

LUTINS is a version of simple type theory. More specifically it is a modified version of the **H-A** system by Henkin and Andrews which in turn is a variant on Church's simple theory of types.
As the name implies, it supports:

- Non-denoting terms & partial functions.
- Subtyping
- n-th order reasoning $\forall n \in \mathbb{N}^+$

[1] pronounced as in French

Sorts, Types and Kinds (1)

A language in LUTINS contains two kinds of objects: *sorts* and *expressions*. Let the set of sorts of a language $\mathcal{L}$ be called $\mathcal{S}$. It is generated inductively from a finite set $\mathcal{A}$ of *atomic* sorts like so:

- 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}. Currying is not required. Taking functions is the only type-forming operation in LUTINS.

Sorts may overlap, but they cannot be empty.

Sorts, Types and Kinds (2)

Each sort is also assigned a unique *enclosing sort* by $\mathcal{L}$. This gives rise to a partial order $\preceq$ on $\mathcal{S}$ with the following properties:

Sorts, Types and Kinds (2)

Each sort is also assigned a unique *enclosing sort* by $\mathcal{L}$. This gives rise to a partial order $\preceq$ on $\mathcal{S}$ with the following properties:

- If $\alpha \in \mathcal{A}$ and β is the enclosing sort of α, then $\alpha \preceq \beta$.
- $\alpha_1 \preceq \beta_1, \dots, \alpha_n \preceq \beta_n$ iff $[\alpha_1, \dots, \alpha_n] \preceq [\beta_1, \dots, \beta_n]$
- 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.
- 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 $\preceq$.

A sort that is maximal in relation to $\preceq$ is called a *type*.

Sorts, Types and Kinds (2)

Each sort is also assigned a unique *enclosing sort* by $\mathcal{L}$. This gives rise to a partial order $\preceq$ on $\mathcal{S}$ with the following properties:

- If $\alpha \in \mathcal{A}$ and β is the enclosing sort of α, then $\alpha \preceq \beta$.
- $\alpha_1 \preceq \beta_1, \ldots, \alpha_n \preceq \beta_n$ iff $[\alpha_1, \ldots, \alpha_n] \preceq [\beta_1, \ldots, \beta_n]$
- 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.
- 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 $\preceq$.

A sort that is maximal in relation to $\preceq$ is called a *type*.

Sorts are divided into two *kinds*, $*$ and ι. A given sort α is of kind $*$ if either $\alpha = *$ or α is a function sort *into* $*$. In all other cases α is of kind ι. This includes all atomic sorts except $*$ itself.

Sorts, Types and Kinds (3)

An elaboration on subtyping:

If $\sigma_0 \preceq \tau_0$, while $\sigma_1 \preceq \tau_1$, then $[\sigma_0, \sigma_1] \preceq [\tau_0, \tau_1]$.

In particular, it contains just those partial (and total) functions that are never defined for arguments outside σ_0, and which never yield values outside σ_1. Note that this makes subtyping in `IMPS` *co*variant, not *contra*variant as in settings with total functions.

A common subtyping hierarchy would be, for example:

$$\mathbb{N} \preceq \mathbb{Z} \preceq \mathbb{Q} \preceq \mathbb{R} \preceq \iota$$

Constructors (1)

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 everywhere. However, it is not possible for a user, to extend the logic with new constructors themselves.

Constructors (2)

The constructors in LUTINS are:

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-form`	if-form$(\varphi_1, \varphi_2, \varphi_3)$
`if`	if(φ, t_1, t_2)

Constructors (3)

The constructors in LUTINS are (cont.):

Constructor	Mathematical Syntax
`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`	$\iota_p v : \alpha, \varphi$
`is-defined`	$t \downarrow$
`defined-in`	$t \downarrow \alpha$
`undefined`	$\perp_\alpha$

Quasi-Constructors (1)

LUTINS also sports *quasi-constructors* that function almost like regular constructors but are definable by the user. This allows uncomplicated extension of the logic.

Once a quasi-constructor is defined, it is available in every theory whose language contains the quasi-constructor's home language. They are implemented as "macros" that are automatically expanded by the system.

Quasi-Constructors (2)

Examples:

Quasi-Constructor	**Schema**
`quasi-equals`	$(E_1 \downarrow \vee E_2 \downarrow) \supset E_1 = E_2$
`total?`	$\forall x_1 : \alpha_1, \ldots, x_n : \alpha_n . f(x_1, \ldots, x_n) \downarrow$
`domain`	$\lambda x_1 : \alpha_1, \ldots, x_n : \alpha_n . f(x_1, \ldots, x_n) \downarrow$
`nonvacuous?`	$\exists x_1 : \alpha_1, \ldots, x_n : \alpha_n . p(x_1, \ldots, x_n)$
...	...

Quasi-Constructors (3)

An actual user-defined example:

```
(def-quasi-constructor I-SUBSETEQ
  "lambda(a,b:sets[uu], forall(x:uu,
                              (x in a) implies (x in b)))"
  (language indicators)
  (fixed-theories the-kernel-theory))
```

Quasi-Constructors (3)

An actual user-defined example:

```
(def-quasi-constructor I-SUBSETEQ
  "lambda(a,b:sets[uu], forall(x:uu,
                                 (x in a) implies (x in b)))"
  (language indicators)
  (fixed-theories the-kernel-theory))
```

QCs are actually *polymorphic*. The sort uu above is not fixed, but could be replaced by any sort.

The ι operator (1)

LUTINS features a *definite description operator*, also called ι. Terms of the form

$$\iota\, x : \alpha \,.\, \varphi$$

denote the *unique* x of sort α that satisfies the predicate φ, if there is one. If there isn't, the term is undefined.

The ι operator (1)

LUTINS features a *definite description operator*, also called ι. Terms of the form

$$\iota\, x : \alpha \,.\, \varphi$$

denote the *unique* x of sort α that satisfies the predicate φ, if there is one. If there isn't, the term is undefined.

Examples:

$$\begin{aligned}
(\iota\, x : \mathbb{R} \,.\, x \cdot x = 2) &\quad \texttt{// undefined!} \\
(\iota\, x : \mathbb{R} \,.\, 0 \leq x \wedge x \cdot x = 2) &\quad \texttt{// sqrt(2)} \\
(\lambda\, x, y : \mathbb{R} \,.\, (\iota\, z : \mathbb{R} \,.\, x = z \cdot y)) &\quad \texttt{// real division} \\
&\quad \texttt{// from multiplication}
\end{aligned}$$

Undefined Values & Partial Functions

The stated goal of IMPS and LUTINS is to allow for reasoning that is very close to mathematical praxis. This means that there needs to be a way to deal with partial functions and undefined values since these are popular in chalk-and-whiteboard mathematics.

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

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

Undefined vs. Non-denoting

There's a subtle difference between a term that is "undefined" and one that is "non-denoting".

According to Farmer, a term in undefined if is not assigned a "natural" meaning (e.g. $\frac{5}{0}$) 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, an undefined term of type $*$ still has a denotation (namely False).

Dealing with undefined values

There's multiple ways of dealing with undefined terms / partial functions, each with their own advantages and difficulties:

- Error Values
- Non-existent values
- Many-sorted Logic
- Total functions with unspecified values
- Partial valuation for terms and formulas
- Partial valuation for terms but total valuation for formulas (this is used in LUTINS)

Partial valuation for terms but total valuation for formulas (1)

In LUTINS, there's two rules that are added to the standard valuation rules:

- A term denotes a value only if all subterms denote values.
- An atomic formula is false if any term occurring in it is non-denoting.

Partial valuation for terms but total valuation for formulas (1)

In LUTINS, there's two rules that are added to the standard valuation rules:

- A term denotes a value only if all subterms denote values.
- An atomic formula is false if any term occurring in it is non-denoting.

This ensures the logic stays two-valued and corresponds most closely to mathematical practice.

Drawbacks: Not as flexible as nonexistent values, not suitable for nonstrict functions without another approach in conjunction.

Partial valuation for terms but total valuation for formulas (2)

<u>Examples:</u>

- $x + \frac{1}{x}$ and $\left(\frac{1}{x}\right)^2$ do not have values if the value of x is 0.
- If $\varphi\prime$ is the result of replacing $\left(\frac{1}{x}\right)^2$ with $\frac{1}{x^2}$ in φ, then φ and $\varphi\prime$ are equivalent regardless of the value of x.
- The equation $\sqrt{x} = 2$ over $\mathbb{R}$ is true for $x = 4$ and false everywhere else, even for $x < 0$.
- $x = \frac{z}{y} \supset x \cdot y = z$ is valid without restrictions on y.

Partial valuation for terms but total valuation for formulas (2)

Examples:

- $x + \frac{1}{x}$ and $\left(\frac{1}{x}\right)^2$ do not have values if the value of x is 0.
- If $\varphi\prime$ is the result of replacing $\left(\frac{1}{x}\right)^2$ with $\frac{1}{x^2}$ in φ, then φ and $\varphi\prime$ are equivalent regardless of the value of x.
- The equation $\sqrt{x} = 2$ over $\mathbb{R}$ is true for $x = 4$ and false everywhere else, even for $x < 0$.
- $x = \frac{z}{y} \supset x \cdot y = z$ is valid without restrictions on y.

. . . all of which seems sensible.

The IMPS system: LUTINS in context

IMPS (1)

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. It was one of the influential systems in the era of automated reasoning.

IMPS (1)

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. It was one of the influential systems in the era of automated reasoning.

One of the goals in developing IMPS was to create a mathematical system that gave computational support to mathematical techniques common among actual mathematicians.

IMPS (2)

The development of the IMPS system has been heavily influenced by three insights into real-life mathematics:

IMPS (2)

The development of the IMPS system has been heavily influenced by three insights into real-life mathematics:

- Mathematics emphasizes the *axiomatic method*.

IMPS (2)

The development of the IMPS system has been heavily influenced by three insights into real-life mathematics:

- Mathematics emphasizes the *axiomatic method*.
- Many branches of mathematics emphasise *functions*, including partial functions. Moreover, the classes of objects studied may be *nested*.

IMPS (2)

The development of the IMPS system has been heavily influenced by three insights into real-life mathematics:

- Mathematics emphasizes the *axiomatic method*.
- Many branches of mathematics emphasise *functions*, including partial functions. Moreover, the classes of objects studied may be *nested*.
- Mathematical proofs usually employ a mixture of both *formal inference* and *computation*.

Theories (1)

“The theory is the basic unit of mathematical knowledge in `IMPS`.”

A *theory* $\mathcal{T}$ in LUTINS is based off a language $\mathcal{L}$, but also incorporated additional sentences (in $\mathcal{L}$). These sentences are collectively called “axioms”, but need not be without proof or witness.
Theorems, for example, are also counted under this umbrella, as are a variety of other constants that can be added to a theory.

The handy catch phrase here is:

$$\text{Theory} = \text{Language} + \text{Axioms}$$

Theories (2)

```
(def-language MONOID-LANGUAGE
  (embedded-languages h-o-real-arithmetic)
  (base-types uu)
  (constants
  (e "uu")
  (** "[uu,uu,uu]")))

(def-theory MONOID-THEORY
  (component-theories h-o-real-arithmetic)
  (language monoid-language)
  (axioms
    (associative-law-for-multiplication-for-monoids
      "forall(z,y,x:uu, x**(y**z)=(x**y)**z)" rewrite)
    (right-multiplicative-identity-for-monoids
      "forall(x:uu,x**e=x)" rewrite)
    (left-multiplicative-identity-for-monoids
      "forall(x:uu,e**x=x)" rewrite)
    ("total_q(**,[uu,uu,uu])" d-r-convergence)))
```

Theory Interpretations

IMPS also supports the concept of a *theory interpretation*. A theory interpretation is a mapping from one theory into another theory with the additional property that theorems are mapped to theorems.

Example:

```
(def-translation MONOID-THEORY-TO-ADDITIVE-RR
  (source monoid-theory)
  (target h-o-real-arithmetic)
  (fixed-theories h-o-real-arithmetic)
  (sort-pairs
    (uu rr))
  (constant-pairs
    (e 0)
    (** +)
  (theory-interpretation-check using-simplification))
```

OMDoc

OMDoc (short for **O**pen **M**athematical **D**ocuments) is a semantics-oriented markup format for STEM-related documents extending OpenMath.

OMDoc

OMDoc (short for **O**pen **M**athematical **D**ocuments) is a semantics-oriented markup format for STEM-related documents extending OpenMath.

OMDoc/MMT brings with it three distinct levels for expression of (both formal and informal) mathematical knowledge, structurally similar to `IMPS`:

- **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.

MMT

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.

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

Formalisation of LUTINS in MMT

LUTINS in MMT (1)

To formalise LUTINS in MMT, we 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

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

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.

LUTINS in MMT (2)

To capture the primitives of LUTINS, we declare:

- a new **LF**-type `tp:type`, which serves as the universe of maximal `IMPS`-sorts,
- a function `sort : tp` $\rightarrow$ `type`, and
- a function `exp :` $\{$`A : tp`$\}$ `sort A` $\rightarrow$ `type`.

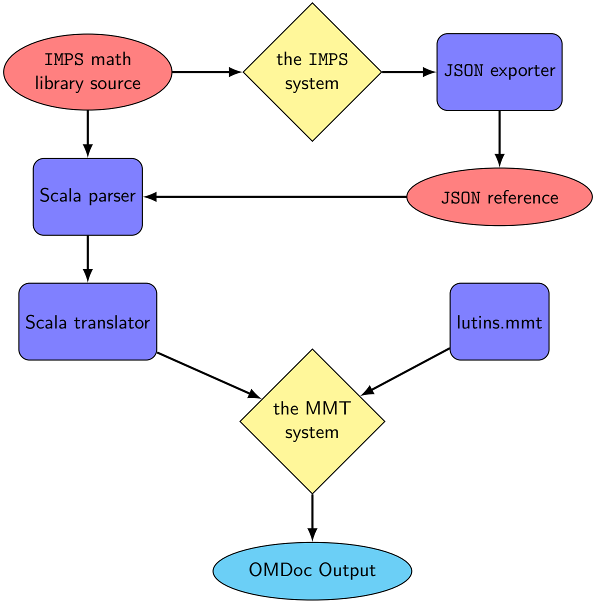
Given some maximal `IMPS`-sort `A`, the type `sort A` then serves as the **LF**-type of all `IMPS` sorts, and given a sort `a : sort A`, the type `exp A a` corresponds to the **LF**-type of all `IMPS`-expressions of sort `a`.

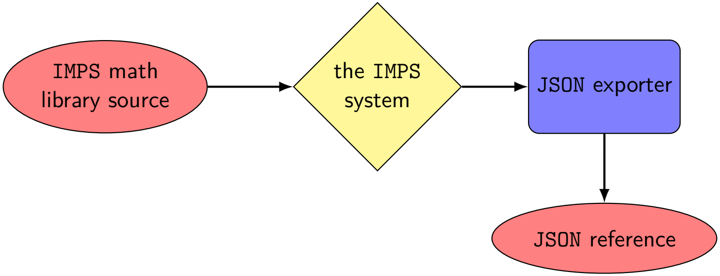
LUTINS in MMT (3)

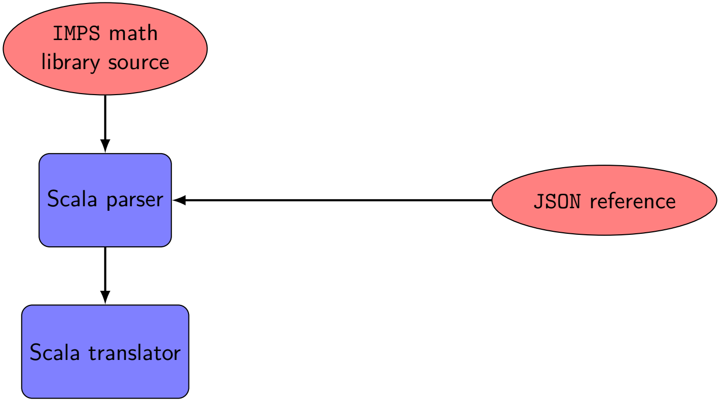
For propositional judgements (i.e. axioms and theorems) in `IMPS`, we use the *judgements-as-types* paradigm by introducing an operator `thm : exp bool` $\rightarrow$ `type`, assigning to each proposition a type which we can think of as the "type of proofs" for that proposition.

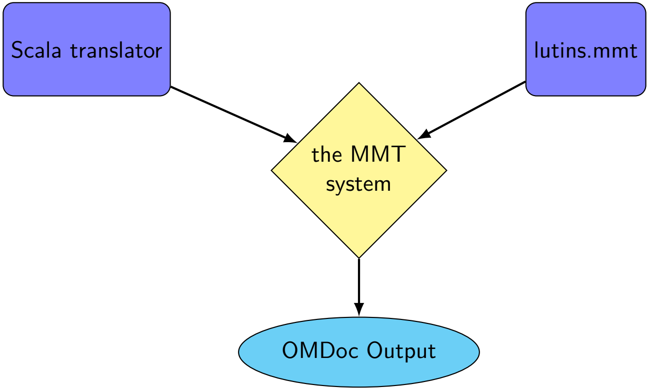
Correspondingly, we consider a proposition A to be "true" if the type `thm` A is inhabited. Axioms, for example, correspond to undefined constants of type `thm A`.

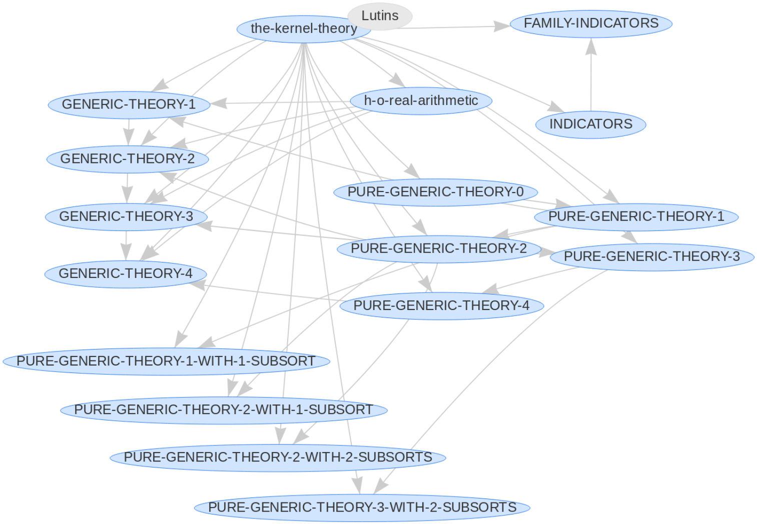
The translation process from MMT to OMDoc

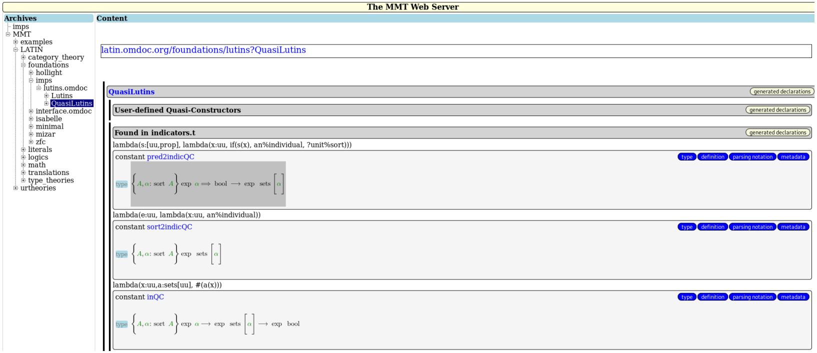

IMPS math library source
the IMPS system
JSON exporter
Scala parser
JSON reference
Scala translator
lutins.mmt
the MMT system
OMDoc Output


IMPS math library source
the IMPS system
JSON exporter
JSON reference


IMPS math library source
Scala parser
JSON reference
Scala translator


Scala translator
lutins.mmt
the MMT system
OMDoc Output


Lutins
the-kernel-theory
FAMILY-INDICATORS
h-o-real-arithmetic
INDICATORS
GENERIC-THEORY-1
GENERIC-THEORY-2
GENERIC-THEORY-3
GENERIC-THEORY-4
PURE-GENERIC-THEORY-0
PURE-GENERIC-THEORY-1
PURE-GENERIC-THEORY-2
PURE-GENERIC-THEORY-3
PURE-GENERIC-THEORY-4
PURE-GENERIC-THEORY-1-WITH-1-SUBSORT
PURE-GENERIC-THEORY-2-WITH-1-SUBSORT
PURE-GENERIC-THEORY-2-WITH-2-SUBSORTS
PURE-GENERIC-THEORY-3-WITH-2-SUBSORTS



The MMT Web Server

Archives

- imps
- MMT
  - examples
  - LATIN
    - category_theory
    - foundations
      - hollight
      - imps
        - lutins.omdoc
          - Lutins
          - QuasiLutins
      - interface.omdoc
      - isabelle
      - minimal
      - mizar
      - zfc
    - literals
    - logics
    - math
    - translations
    - type_theories
  - urtheories

Content

latin.omdoc.org/foundations/lutins?QuasiLutins

QuasiLutins generated declarations

User-defined Quasi-Constructors generated declarations

Found in indicators.t generated declarations

lambda(s:[uu,prop], lambda(x:uu, if(s(x), an%individual, ?unit%sort)))

constant pred2indicQC

type $\left\{A, \alpha\colon \mathrm{sort}\ A\right\} \mathrm{exp}\ \alpha \Longrightarrow \mathrm{bool} \longrightarrow \mathrm{exp}\ \ \mathrm{sets}\left[\alpha\right]$

lambda(e:uu, lambda(x:uu, an%individual))

constant sort2indicQC

type $\left\{A, \alpha\colon \mathrm{sort}\ A\right\} \mathrm{exp}\ \ \mathrm{sets}\left[\alpha\right]$

lambda(x:uu,a:sets[uu], #(a(x)))

constant inQC

type $\left\{A, \alpha\colon \mathrm{sort}\ A\right\} \mathrm{exp}\ \alpha \longrightarrow \mathrm{exp}\ \ \mathrm{sets}\left[\alpha\right] \longrightarrow \mathrm{exp}\ \ \mathrm{bool}$

Future Work and Intranslatables

Hand-rolled: QuasiConstructors

Quasi-constructors in `IMPS` are polymorphic, but their internal representation relies on the language of an underlying theory (in lieu of having an extra language just for this). There is no internal representation of a polymorphic expression that could be exported.

Based on that and the fact that there are < 60 quasi-constructors total (and additions to that list are unlikely), we decided it would be easier to write the **LF**-implementations by hand.
We also experimented with immediate expansion, but that led to huge, unhelpful terms.

Missing: Flexary functions (1)

Many (quasi-)constructors like `and` or `total` are actually flexary in LUTINS, but since **LF** does not support flexary terms, we have to approximate with taking one or two arguments at a time.

Example:

`p1 and p2 and ... and pn` // $\varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_n$

`and : exp bool` $\rightarrow$ `exp bool` $\rightarrow$ `exp bool` $\#$ `1` $\wedge$ `2 prec 30`

Missing: Flexary functions (2)

Example (cont.):

$$\texttt{total?}\quad (f, [\beta_1, \ldots, \beta_{n+1}])$$

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

Missing: Flexary functions (2)

Example (cont.):

$$\texttt{total?}\quad (f, [\beta_1, \ldots, \beta_{n+1}])$$

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

There *is* a version of **LF** that would support these, called **LFS** (short for **LF** + Sequences), which could support flexible arities. We hope to transport the implementation of LUTINS to **LFS** in the future.

Missing: Proofs

`IMPS` has a proof system that relies on a *deduction graph* (with *inference* and *sequent* nodes.). The user manipulates this graph via proof commands.

However, `IMPS` does not save the proof scripts internally in a structured way. Hence, we currently only “translate” proofs as opaque data from the sources.

In the future, we hope to formalise (parts of?) the `IMPS` proof system in **LF** also and gain at least partial verification of the proofs from the source code.

Thank you for your attention!