Generic Literals

Florian Rabe

Jacobs University Bremen, Computer Science

Calculemus track at CICM 2015

Literal = atomic expression with fixed interpretation

Prevalent in formal systems:

- booleans: *true*, *false*
- natural numbers: $0, 1, \ldots$
- 32-bit integers: $-32767, \ldots, 32768$
- IEEE single precision floats: $1.234e2$, NaN, ...
- characters: 'a', 'b', ...
- strings: "abc", "def", ...
- physical units, regular expressions, URIs, colors, dates, ...

Formal system $\mathcal{F}$ = set of expressions e (and inference system)

Model M =

- set of values $|M|$
- interpretation function $e \mapsto [\![e]\!]^M \in |M|$

Literal = expression v such that for all M

$$[\![v]\!]^M := v$$

Values v come from background universe of $\mathcal{F}$

- logical foundation
- underlying programming language

Canonical options

- none (not as dumb as it may sound)
  - can use inductive types instead
  - optionally, e.g., parse 3 as $s(s(s(0)))$
- all (there aren't that many useful ones)
  - e.g., start with *nat*, *int*, *float*
  - extend implementation if necessary
- extensible by user this talk
  - just like types, operators, axioms/theorems, notations
  - elegant but has overhead

Hard-wired choice

Both MathML and OpenMath

- integers (unlimited precision)
- IEEE floats (double precision)
- strings
- byte arrays

Only MathML

- real numbers (unspecified text encoding) bug?

Frameworks Need Extensible Literals

Consider languages in which others are represented

logical frameworks, MathML, MMT, etc.

Reasonably

- allow any choice of literals — any language representable
- disallow literals in certain contexts — empty theory should have no literals

Ideally

- modular language definitions — reusable, orthogonal language features
- each set of literals separate feature — literals available if explicitly imported

New MMT Feature: modular, extensible Literals

- Vision: Universal framework for the formal representation of knowledge and its semantics
- Maturity:
  - developed since 2006
  - >300 pages of publications
  - $>30,000$ lines of Scala code
- Key features:
  - systematically abstract from foundational logics
  - maximize reusability of concepts, results, implementation

So far

- Theories — logics, theories, models, . . . ,
- Morphisms — imports, language translations,. . .
- Declarations — symbols, defintions, axioms/theotems, rules, . . .
- Objects — formulas, types, terms, proofs, . . .
- Typing relation — typing, provability, . . .

Now also: literals

Originally same as OpenMath objects:

$$O \quad ::= \quad s \mid x \mid Apply(O, O^*) \mid Bind(O, (x : O)^*, O)$$

$$\mid int \mid float \mid string \mid bytearray$$

awkward

Originally same as OpenMath objects:

$$O \quad ::= \quad c \mid x \mid Apply(O, O^*) \mid Bind(O, (x : O)^*, O)$$

$$\mid int \mid float \mid string \mid bytearray$$

$$\mid v^s$$

Now: single constructor v^s for literals

- v: the extra-linguistic value
- s: the symbol defining the semantics of v

3^{int}, $1.0^{IEEEDouble}$, ...

What v are allowed?

- any extra-linguistic value v
- in line with MathML philosophy:
  syntax allows anything that might make sense

Symbol s determines semantics of v^s in 3 ways:

declared extensibly in theories

1. informal documentation
2. practical implementation
3. theoretical definition

details on next slides

- Symbol s is declared in MMT theory ≈ content dictionary
- Documentation of s defines
  - legal values v
  - string encoding $E(v)$
- MMT concrete syntax of v^s uses string encoding

<literal type="s" value="$E(v)$"/>

<literal type="nat" value="3"/>

- MMT type checker parametric in set of rules
- MMT relegates to rules for all language-specific aspects
- Rules provided as Scala snippets

  e.g., $\sim$ 10 rules for LF, 10 loc each
- New abstract rule for s-literals
  - to check v^s, MMT looks for rule R_s for s-literals
  - R_s implements string encoding, validity check for s-literals
  - if valid, type of v^s is s

Natural number literals

```
val nat = "http://example.org?Literals?Nat"

object StandardNat extends LiteralRule(nat) {
  def fromString(s: String) = {
    val i = BigInt(s)
    if (i >= 0) Some(i)
    else None
  }
  def toString = ...
}
```

All OpenMath literals definable accordingly

- Type s declared in MMT theory T
- T-models M treated as theory extensions $T \hookrightarrow D_M$
- Typing rule (essentially)

$$\frac{v \in [\![s]\!]^M}{D_M \vdash v^s : s}$$

1. Define MMT theory T

MMT	Scala

theory Int {
u : type
zero: u
plus: u → u → u
}	

1. Define MMT theory T
2. MMT generates abstract Scala class S_T

MMT	Scala

theory Int {
 u : type
 zero: u
 plus: u → u → u
}
	abstract class Int {
	 type u
	 val zero: u
	 def plus(x1: u, x2: u): u
	}

1. Define MMT theory T
2. MMT generates abstract Scala class S_T
3. User provides T-model M by implementing S_T

MMT	Scala

`theory Int {`
` u    : type`
` zero: u`
` plus: u → u → u`
`}`
	`abstract class Int {`
	` type u`
	` val zero: u`
	` def plus(x1: u, x2: u): u`
	`}`

	`class StandardInt extends Int {`
	` type u = BigInt`
	` val zero = BigInt(0)`
	` def plus(x1:BigInt, x2: BigInt) =`
	`   x1 + x2`
	`}`

1. Define MMT theory T
2. MMT generates abstract Scala class S_T
3. User provides T-model M by implementing S_T
4. User imports theory D_M to use M-literals

MMT	Scala

theory Int {
 u : type
 zero: u
 plus: u → u → u
}
	abstract class Int {
	 type u
	 val zero: u
	 def plus(x1: u, x2: u): u
	}

theory Test {
 include Int
 include StandardInt
 test : u = plus(1,1)
}
	class StandardInt extends Int {
	 type u = BigInt
	 val zero = BigInt(0)
	 def plus(x1:BigInt, x2: BigInt) =
	   x1 + x2
	}

Function literals

Function Literals

- Do we need literals of non-atomic types?
- Only useful case: literals of function type
  - represent built-in operators
  - only way to compute with literals
- In MMT: function literals = infinite set of axioms

- Assume T-model M $(\mathbb{Z}, 0, +)$
- Diagram theory $T \hookrightarrow D_M$ defined by
  - one nullary constant v^s for each $v \in [\![s]\!]^M$ $0^{int}, 1^{int}, \ldots$
  - one axiom for each true instance of an atomic formula $\vdash 1^{int} + 1^{int} = 2^{int}, \ldots$
- Standard result:

$$D_M \vdash F \quad \text{iff} \quad M \models F$$

- Assume T-model M $(\mathbb{Z}, 0, +)$
- Diagram theory $T \hookrightarrow D_M$ defined by
  - one nullary constant v^s for each $v \in [\![s]\!]^M$ $0^{int}, 1^{int}, \ldots$
  - one axiom for each true instance of an atomic formula
    $\vdash 1^{int} + 1^{int} = 2^{int}, \ldots$
- Standard result:

$$D_M \vdash F \quad \text{iff} \quad M \models F$$

Side remark

- Is there a theory morphism $d_m : D_M \to D_{M'}$ for each model morphism $m : M \to M'$?
- Easy part: $d_m : v^s \mapsto v'^s$ whenever $m : v \mapsto v'$
- But
  - theory morphisms preserve all true sentences
  - model morphisms preserve all true atomic sentences

- Diagram D_M yields infinite set of atomic axioms
- In particular, function symbols defined by axioms of the form

$$\vdash f(v_1^{c_1}, \ldots, v_n^{c_n}) = v^c$$

- Reflected into MMT as rewrite rules

MMT	Scala
<code>theory Int {</code> <code> u : type</code> <code> zero: u</code> <code> plus: u → u \to u</code> <code>}</code>	<code>abstract class Int {</code> <code> type u</code> <code> val zero: u</code> <code> def plus(x1: u, x2: u): u</code> <code>}</code>
<code>theory Test {</code> <code> include Int</code> <code> include StandardInt</code> <code> test : u = plus(1,1)</code> <code>}</code>	<code>class StandardInt extends Int {</code> <code> type u = BigInt</code> <code> val zero = BigInt(0)</code> <code> def plus(x1:BigInt, x2: BigInt) =</code> <code> x1 + x2</code> <code>}</code>

$$\mathtt{Test} \vdash \mathtt{plus}(1^u, 1^u) \rightsquigarrow 2^u$$

Relationship to Biform Theories

Farmer and von Mohrenschildt, 2003

- Biform theory = axioms + syntax transformers
- syntax transformer: externally given algorithm that perform certain equality conversion
- allows combining logic with algorithms

This paper

- Biform theory = theories + models
- Two kinds of models: semantic or computational treated uniformly
- Models combined with axiomatic theories via diagrams D_M
- Diagrams of computational models yield
  - literals for all values
  - rewrite rules for all true atomic formulas

Future work: mixing computation and deduction is hard

not surprising

- Pure deduction: axiomatic theories typical for proof assistants
- Pure computation: computational models

  typical for computer algebra
- Reality: nice to mix both

Lots of difficulties

Example: find X such that

$$plus(1^{int}, X) = 3^{int}$$

comes up all the time during type checking, proof search

Partial solution in MMT: models may supply inversion rules

Inductive family of vectors dependently-typed, implicit arguments

$$
\begin{array}{ll}
\texttt{include}\ \texttt{StdNat} & \\
c & : a \\
a & : \texttt{type} \\
\texttt{vec} & : \texttt{nat} \to \texttt{type} \\
\texttt{nil} & : \texttt{vec}\,0 \\
\texttt{cons} & : \{n : \texttt{nat}\}\, a \to \texttt{vec}\, n \to \texttt{vec}\,(\texttt{succ}\, n) \\
\texttt{head} & : \{n : \texttt{nat}\}\texttt{vec}\,(\texttt{succ}\, n) \to a
\end{array}
$$

$$
\begin{array}{l}
\mathit{test}0 : \texttt{vec}\,2 = \texttt{cons}\, c\,(\texttt{cons}\, c\, \texttt{nil}) \\
\mathit{test}1 : a = \mathit{head}\ \mathit{test}0
\end{array}
$$

Checking $\mathit{test}0$ requires $\texttt{vec}(\texttt{succ}(\texttt{succ}\,0)) = \texttt{vec}2$
Checking $\mathit{test}1$ requires solving $\texttt{vec}(\texttt{succ}\, n) = \texttt{vec}\,1$

- Literals new feature in MMT
  - foundation-independent
    any choice of literals combinable with any logic
  - user-extensible like symbols, theorems, notations, ...
  - integrated with MMT type system
    dependent types, type reconstruction, module system, ...
- Library of literals as part of LATIN logic library
  import literals as needed
- Computation integrated with axiomatic logic
  - computation rules provided by models
  - computation called seamlessly during checking, proving
    computation also inverted if needed