Flexary Operators for Formalized Mathematics

Fulya Horozal Florian Rabe Michael Kohlhase

Jacobs University, Bremen, Germany

Mathematical Knowledge Management
Conferences on Intelligent Computer Mathematics
July 07, 2014
Coimbra, Portugal

Flexary Operators

- **flexary** = flexible arity compare unary, binary, etc.
- Many operators naturally flexary pervade mathematics
  - associative operators

$$a_1 + \ldots + a_n \ = \ +(a_1, \ldots, a_n)$$

  - collection constructors

$$\{a_1, \ldots, a_n\} \ = \ set(a_1, \ldots, a_n)$$

  - vector, matrix, polynomial constructors
- Not commonly supported in content representation languages surprising

Flexary Binders

- Binder = operator that binds some variables in a scope
- Arity = number of bound variables
- Flexary binders very common

  actually, hard to find a fixed-arity binder
- Define: binder B is **associative** if

$$Bx, y.E \;=\; Bx.By.E$$

- Most binders not only flexary but also associative
  - associative: $\forall$, $\exists$
  - associative up to currying: $\int$, λ
  - not associative (but still naturally flexary): $\exists^1$
- Support for flexary binders equally desirable

Standard Solution (1)

Use (some incarnation of) lists

$$a_1 + \ldots + a_n \;=\; +(list(a_1, \ldots, a_n))$$

But

- awkward — even more so for a mathematician
- introduces foundational dependency — what if there are no lists in my language?

Standard Solution (2)

Use notations

- only fixed arity in content
- parser and printer adapted to mimic flexible arity

$a_1 + \ldots + a_n$ parsed as $+(a_1, \ldots, +(a_{n-1}, a_n) \ldots)$

But

- flexary representation often more natural
- requires choice between right- and left-associative notations
  no-canonical choice for non-associative flexary operators
- requires domain=codomain
  cannot make the $\{\ldots\}$ operator right-associative
- no flexary reasoning
  would be nice to quantify over the number of arguments

Ellipses

- Flexary operators naturally lead to ellipses
- **Sequential ellipsis**

$$\text{define} \quad [a_i]_{i=1}^{n} \quad \text{as} \quad a_1, \ldots, a_n$$

  example:

$$+[a_i]_{i=1}^{n} = +(a_1, \ldots, a_n)$$

- No standardized formalization

  dot-dot-dot notation fine on paper

Ellipses

- Flexary operators naturally lead to ellipses
- **Sequential ellipsis**

$$\text{define} \quad [a_i]_{i=1}^{n} \quad \text{as} \quad a_1, \ldots, a_n$$

  example:

$$+[a_i]_{i=1}^{n} = +(a_1, \ldots, a_n)$$

- No standardized formalization

  dot-dot-dot notation fine on paper

- **Nested ellipsis**

$$f_1(\ldots f_n(x) \ldots)$$

  special case of sequential ellipsis via flexary function composition:

$$f_1(\ldots f_n(x) \ldots) = \circ[f_i]_{i=1}^{n}(x)$$

Where to Formalize Flexible Arities?

- Theory level: not good
  - amounts to creating a theory of lists
  - must be imported into any theory with flexary operators

    e.g., monoids
- Logic level: better but
  - logics becomes more complicated
  - flexible arities logic-independent feature
- Logical framework level — our approach
  - once-and-for-all formalization
  - corresponds to mathematical practice

    flexible arity and ellipses are assumed at the meta-level

Overview

1. Define logical framework LFS
   extends Edinburgh Logical Framework (LF) with
   - sequences
   - ellipses
   - flexible arities
2. Use LFS to define flexary logics
   - flexary connectives
   - flexary quantifiers
   - with corresponding flexary inference rules

   concretely: flexary FOL, flexary λ-calculus
3. Use flexary logics to formalize mathematical examples

LF with Sequences (LFS)

- LF $=$ dependently-typed λ-calculus
- LF primitives
  - terms, types, and kinds
  - Π-types, λ, and application
  - typing judgment $\vdash E : E'$
- Very simple, but just right as logical framework
- New primitives in LFS
  - term, type, and kind **sequences**
  - natural numbers — needed for indices in ellipses
  - sequence ellipsis $[E(i)]_{i=1}^{n}$
  - flexary function composition — needed for nested ellipses

LFS Syntax

LF Grammar

$$E \quad ::= \quad \mathtt{type} \mid \Pi x : E.\, E \mid \lambda x : E.\, E \mid E\, E$$

olive

LFS Syntax

LFS Grammar

$$\begin{array}{rcl} E, n & ::= & \mathtt{type}^{n} \mid \Pi x : E.\, E \mid \lambda x : E.\, E \mid E\, E \\ & & \cdot \mid E, E \mid E_n \end{array}$$

- Empty sequence $\cdot$
- Concatenation E, E'
- Index E_n — n-th element of E

olive

LFS Syntax

LFS Grammar

$$E, n \quad ::= \quad \mathtt{type}^n \mid \Pi x : E.\, E \mid \lambda x : E.\, E \mid E\, E$$
$$\cdot \mid E, E \mid E_n \mid [E]_{x=1}^n \mid \circ\, E$$

- Empty sequence $\cdot$
- Concatenation E, E'
- Index E_n — n-th element of E
- Sequence ellipses $[E(x)]_{x=1}^n$ — reduces to $E(1), \ldots, E(n)$
- Flexary function composition $\circ f$

  $\circ\,(f_1, \ldots, f_n)\, s$ reduces to $f_1(\ldots(f_n\, s)\ldots)$

olive

Flexary Interpretation of LF Primitives

- LF primitives retained but now flexary
- Flexary application = sequence arguments

$$f\ \cdot\ =\ f \qquad f\ (E, E')\ =\ (f\ E)\ E'$$

- Flexary binding = sequence variables

$$\lambda x : \cdot . E \quad = \quad E[x/\cdot]$$

$$\lambda x : (E, E').F \quad = \quad \lambda x^1 : E.\ \lambda x^2 : E'.\ F[x/(x^1, x^2)]$$

accordingly for Π

- LF typing rules can be reused without change

Type System: Natural Numbers

Axiomatized as LF declarations

$$
\begin{array}{lcl}
\mathtt{nat} & : & \mathtt{type} \\
\leq & : & \mathtt{nat} \to \mathtt{nat} \to \mathtt{type} \\
0 & : & \mathtt{nat} \\
1 & : & \mathtt{nat} \\
+ & : & \mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}
\end{array}
$$

with appropriate axioms

Type System: Introduction of Sequences

Kind sequences

$$\frac{\Sigma \vdash n : \mathtt{nat} : \mathtt{type}}{\Sigma \vdash \mathtt{type}^n \; Kind}$$

Type sequences

$$\frac{\vdash \Sigma \; Sig}{\Sigma \vdash \cdot : \mathtt{type}^0} \qquad \frac{\Sigma \vdash U : \mathtt{type}^m \quad \Sigma \vdash V : \mathtt{type}^n}{\Sigma \vdash U, V : \mathtt{type}^{m+n}}$$

Term sequences

$$\frac{\vdash \Sigma \; Sig}{\Sigma \vdash \cdot : \cdot : \mathtt{type}^0} \qquad \frac{\Sigma \vdash S : U : \mathtt{type}^m \quad \Sigma \vdash T : V : \mathtt{type}^n}{\Sigma \vdash S, T : U, V : \mathtt{type}^{m+n}}$$

Type System: Elimination of Sequences

Term sequences

$$\frac{\Sigma \vdash S : U : \mathtt{type}^n \quad \Sigma \vdash x^* : 1 \leq x : \mathtt{type} \quad \Sigma \vdash x_* : x \leq n : \mathtt{type}}{\Sigma \vdash S_x : U_x : \mathtt{type}}$$

accordingly for type sequences

Static bound checking:

- Only valid indices within bounds well-typed
- 2 implicit arguments for $1 \leq x$ and $x \leq n$

Type System: Ellipses

Ellipsis for sequence of terms

$$\frac{\Sigma \vdash n : \mathtt{nat} : \mathtt{type} \qquad \Sigma,\ x{:}\mathtt{nat},\ x^*{:}1 \leq x,\ x_*{:}x \leq n \vdash S : U : \mathtt{type}}{\Sigma \vdash [S]_{x=1}^{n} : [U]_{x=1}^{n} : \mathtt{type}^{n}}$$

accordingly for sequence of types

Static bound checking:

- Actually binds 3 variables
- Bounds $1 \leq x$ and $x \leq n$ passed as assumptions

Flexary Function Composition

$$\frac{\Sigma \vdash U : \mathtt{type}^{n+1} \quad \Sigma \vdash F : [U_{i+1} \to U_i]_{i=1}^{n}}{\Sigma \vdash \circ F : U_{n+1} \to U_1}$$

Example: Folding

If

$$S : A, \ldots, A : type^n \quad \text{and} \quad f : A \to A \to A \quad \text{and} \quad a : A$$

then

$$i : nat \vdash \lambda x : A.\, f\ x\ S_i : A \to A$$

and we define

$$\mathtt{foldl}\ S\ f\ a \ = \ (\circ [\lambda x : A.\, f\ x\ S_i]_{i=1}^{n})\, a$$

(here $U_i = A$ for $1 \leq i \leq n+1$)

Flexary Connectives

LFS type *form* : `type` of FOL formulas

Notation: Write $form^n$ for $[form]_{i=1}^n$

Binary conjunction

$$\wedge : form \to form \to form$$

Flexary conjunction

$$\wedge^* \; : \; \Pi n : nat.\, form^n \to form \quad = \quad \lambda n : nat.\, \lambda F : form^n.\, \texttt{foldl}\, F \wedge true$$

Thus,

$$\wedge^* n\ F_1, \ldots, F_n = (\ldots(true \wedge F_1)\ldots) \wedge F_n$$

$$\wedge^*\ 0 \cdot = true$$

- Flexary proof rules also definable in terms of rules for binary conjunction
- Other flexary connectives defined accordingly

Flexary Quantifiers

LFS type *term* : `type` of FOL terms

Unary universal quantifier

$$\forall : (term \to form) \to form$$

Flexary universal quantifier

$$\forall^* : \Pi n : nat.\, (term^n \to form) \to form$$

$$= \lambda n : nat.\, \lambda F : term^n \to form.$$

$$\circ\, [\underbrace{\lambda f : term^i \to form.\, \lambda y : term^{i-1}.\, \forall \lambda x : term.\, f\,(y, x)}_{(term^i \to form) \to (term^{i-1} \to form)}]_{i=1}^{n}\, F$$

- Flexary proof rules definable accordingly
- Other flexary quantifiers definable accordingly

Powers

- Signature of monoids in FOL

$$a : type$$

$$\bullet : a \to a \to a$$

$$e : a$$

- Power operator routinely used in informal mathematics

  often introduced in same paragraph

  but not definable in FOL
- Now: flexary monoid operator definable in flexary FOL

$$\bullet^* \;:\; \Pi n : \mathtt{nat}.\, a^n \to a \;=\; \lambda n.\, \lambda x : a^n.\, \mathtt{foldl} \bullet x\, e$$

  and thus power operator definable

$$power \;:\; a \to \mathtt{nat} \to a \;=\; \lambda x.\, \lambda n.\, \bullet^*\; x^n$$

Multirelations

- Multi-relations routinely used in informal mathematics

  e.g., $a \in b \subseteq c$
- But cannot be defined as single operators within a fixary logic
- In flexary FOL:

$$\mathit{multirel} : \Pi n : \mathit{nat}.\ \mathit{term}^{n+1} \to (\mathit{term} \to \mathit{term} \to \mathit{form})^{n} \to \mathit{form}$$

$$= \lambda n.\lambda x.\,\lambda r.\wedge^{*}\,[r_i\,x_i\,x_{i+1}]_{i=1}^{n}$$

- Example:

$$a \in b \subseteq c \;=\; \mathit{multirel}\,(a, b, c)\,(\in, \subseteq)$$

Conclusion

- Sequences and ellipses meta-level operators of informal mathematics
- But a challenge for formalized mathematics
- Logical framework approach permits clean solution
  - LFS = LF with sequences and ellipses
  - flexary logics defined in LFS
  - natural formalizations in flexary logics
- Key properties
  - flexary operators take natural number argument
    arity polymorphism
  - LFS retains semantics of LF primitives
    no new type constructors, no change to typing rules
  - length of sequences known to type system static bounds check