Flexary Operators for Formalized Mathematics

Fulya Horozal, Florian Rabe, and Michael Kohlhase

Computer Science, Jacobs University Bremen, Germany http://kwarc.info

Abstract. We study representation formats that allow formally defining what we call *flexary* operators: functions that take arbitrarily many arguments, like $\sum_{k=1}^{n} a_k$ or binders that bind arbitrarily many variables, like $\forall x_1, \ldots x_n. F$. Concretely, we define a flexary logical framework based on LF, and use it as a meta-language to define flexary first-order logic and flexary simple type theory. We use these to formalize several flexary mathematical concepts including arithmetical and logical operators, matrices, and polynomials.

1 Introduction & Related Work

Ellipses $(\ldots)$ such as in $a_1, \ldots, a_n$ are commonly and indispensably used in mathematical texts. However, representation formats for formalized mathematics typically do not provide a structural analog for ellipses. This is problematic because many common operators are naturally defined using a primitive ellipsis operator, and formalizations have to work around the missing language infrastructure. We will now lay out the problem, survey and discuss the most commonly used workarounds and then present a solution which introduces sequences as a language feature at the meta-level.

1.1 Flexary Operators and Ellipses

We say that an operator is **of flexible arity** or **flexary** if it can take arbitrarily many arguments. Common examples are set-construction $\{a_1, \ldots, a_n\}$ or addition $a_1 + \ldots + a_n$. We speak of **fixary** operators if the number of arguments is fixed.

Ellipses Flexary operators are closely related to the ellipsis operator …. For presentation-oriented formats, ellipsis are no challenge. For example, LaTeX offers `\ldots`, and presentation MathML [Aus+03] marks up the corresponding Unicode character as an operator via the `mo` element. However, the content-oriented formats that we need for formalized mathematics have devoted much less attention to ellipses. This is surprising considering how ubiquitous they are in mathematical practice.

We can distinguish 4 kinds of ellipses. We speak of a **sequence** ellipsis if we give a sequence of arguments to a flexary operators as in $a_1 + \ldots + a_n$.

The **nesting** ellipsis uses a characteristic double-… pattern to compose a sequence of functions as in $f_1(\ldots f_n(x)\ldots)$. It corresponds to folding the function

f over the list of arguments. In the presence of a flexary operator for function composition, we can recover the nesting ellipsis as a special case of the sequence ellipsis via $f_1(\ldots f_n(x)\ldots) := (f_1 \circ \ldots \circ f_n)(x)$.

When working with matrices, we use **2-dimensional** ellipses as in the matrix on the right. If we define vectors using a flexary constructor $(a_1, \ldots, a_n)$ and matrices as vectors of vectors, we can recover the 2-dimensional ellipses by combining two sequence ellipses. Of course, that would still leave the problem of presenting vectors with ellipses.

$$E_n = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & 0 & 1 \end{pmatrix}$$

Finally, we have the **infinite** ellipsis used mainly for infinite series as in $a_1 + a_2 + \ldots$.

Flexary Binders We can generalize the above concepts to binding operators. We speak of a **flexary binder** if it can bind an arbitrary number of variables as in $\forall x_1, \ldots, x_n.F$. Most unary binders such as quantifiers and λ are usually assumed to be flexary in this sense.

1.2 Flexary Notations

A common approach is to use representation languages that are fixary at the content level but flexary at the presentation level. The connection between the two is performed by **flexary notations**. Typically, these use associativity constraints on binary infix operators.

For example, we can define flexary addition $+(a_1, a_2, a_3)$ as identical to $a_1 + a_2 + a_3$, which in turn is an abbreviation for $(a_1 + a_2) + a_3$. Here we use the logical property of associativity to justify a left-associative notation.

We can also use associative notations for logically non-associative operators. For example, we can define flexary implication $\Rightarrow (a_1, a_2, a_3)$ as $a_1 \Rightarrow (a_2 \Rightarrow a_3)$, i.e., by using a right-associative notation for binary implication.

While there is no established terminology, we can apply similar notations to binders. We call a flexary binder **associative** if $Qx_1.Qx_2.F = Qx_1, x_2.F$. In that case, we can define the flexary version of the binder from the unary version. This is very common because the important binders of universal and existential quantifier are associative and (up to currying) so are λ, integral (e.g. $\int\ dx\ dy\ dz$), sum (e.g. $\sum_{i,j\in\mathbb{N}}$), and product (e.g. $\prod_{n+m<k}$). A notable exception is the quantifier of unique existence.

Using flexary notations has the advantage that the content level remains simpler: Flexary operators are always implicitly reduced to fixary ones. This is important for language analysis and perfectly sufficient in informal mathematics. However, in formalized mathematics, the implicit conversions must be explicitly implemented. Usually, this is achieved by using the notation declarations to direct the parser and printer to convert between the seemingly-flexary human-facing syntax and the fixary official syntax.

This is unsatisfactory for several reasons. Firstly, for logically associative operators, there is no canonical choice between using a left- or a right-associative

notation. In a flexary content representation both $(a_1+a_2)+a_3$ and $a_1+(a_2+a_3)$ would normalize to the canonical $+(a_1, a_2, a_3)$.

Secondly, this trick only works well if the domains and codomain of the operator are equal. Consider the flexary set construction operator $\{a_1, a_2, a_3\}$. We can only approximate it using a left-associative notation for an adjoin operator $a\&b := a \cup \{b\}$ and then write $\varnothing\&a_1\&a_2\&a_3$.

Thirdly, the flexary representation is often the more natural one for implementation, e.g. for the left-associative application $f@t$ in simple type theory. In the usual fixary type theory, the n-ary functions $f(x, y)$ are represented in the curried form $f@x@y$, which internally expands to $@(@(f, x), y)$. Thus, the head f of the term is not available at the root of the syntax tree and has to be looked up by traversing the tree. In practice, this traversal is so awkward that many implementations of type theory, e.g., Twelf [PS99], internally use a flexary application operator after all so that $f(x, y)$ can be represented as $@(f, x, y)$. This leads to the strange situation that both the user and the developer effectively use flexary operators, and only the official language definition uses fixary ones.

Finally, this approach only works in general for the case where the number of arguments is constant: We cannot use it to represent a sequence ellipsis like $+(a_1, \ldots, a_n)$, where the number of arguments is a variable. For the special case of conjunction and disjunction, notations for ellipses were realized in Mizar [CICM1212]. For example, a special binary connective $\&\ldots\&$ is used for the conjunction of a sequence ellipsis, and the parser expands $F(m)\&\ldots\&F(n)$ into $\forall i.\, m \leq i \leq n \Rightarrow F(i)$.

1.3 Flexary Representation Languages

Instead of simulating flexary operators through notations, we can use a representation language that supports flexary operators at the content level. There are several content features that can be used.

Lists We can represent flexary operators as unary operators that take a list as an argument. In that case, we represent $a_1 + a_2 + a_3$ as $+(List(a_1, a_2, a_3))$. Actually, this tacitly assumes that we have at least a flexary list constructor. In a pure fixary language, we would have to represent it as $+(cons(a_1, cons(a_2, cons(a_3, nil))))$, which is quite different from the informal mathematical object.

This approach permits using variables that quantify over sequences, and – using map and fold – it is easy to represent ellipses. This is widely used in programming languages, where lists are an accepted foundational data type.

In mathematics however, it is artificial to use lists since any formal mathematical theory for flexary operators would depend on the theory of lists, which itself is rarely used in informal mathematics. Another drawback is that all arguments must have the same domain. To permit different argument domains, we must allow lists whose elements have different types (or use sufficiently imprecise types).

Sequences Sequence types use a monadic type constructor $Seq : type \to type$ like lists and enjoy the same advantages. The difference is that sequences are always flattened, i.e., the canonical functions $Seq(Seq(A)) \to SeqA$ and $A \to Seq(A)$ are inclusions. For example, $Seq(a, b, Seq(c, d), e, f) = Seq(a, b, c, d, e, f)$. This makes sequence types closer to informal mathematics because they need less representational artifacts. Variants of sequence types occur in some programming languages but are rare in typed languages for formalized mathematics.

Sequences are more common in untyped languages. In the absence of a type system and in the presence of flattening, there is no need to write $f(Seq(a, b, c))$ at all. Instead, we can simply write $f(a, b, c)$ (even if one of the arguments is another sequence).

This approach is used in Common Logic (CL [Com]), an untyped flexary variant of first-order logic. There, every non-logical symbol is flexary and variables may quantify over sequences. This substantially complicates the semantics because models must interpret every function symbol as a function that takes an arbitrary sequence of arguments; incidentally a proof theoretical semantics is not defined in CL.

[KB04] defines a flexary first-order logic and studies its semantics. The signature defines the arity of each non-logical symbol, and the arity can either be fixed or flexible. Similarly, variables are divided into individual and sequence variables.

Mathematica [Wol12] also uses untyped sequences, including sequence variables. Functions are fixary, but flexary functions can be defined by matching arguments against sequence patterns. Because Mathematica focuses on computation rather than logic, this is less problematic than for CL.

The untyped approaches to sequences usually cannot represent ellipses well because they tend to lack higher-order functions.

Indexed types Mixed-type lists can be represented concisely in Martin-Löf type theory [ML74], calculus of constructions [CH88] and related languages. Example implementations are Agda [Nor05] and Coq [The14]. If we write $[n]$ for the type containing $0, \dots, n-1$, we call objects of type $T : [n] \to type$ indexed types. Mixed-type lists can be defined as indexed terms $l : \Pi i : [n]. T(i)$. Then flexary functions can be declared concisely as binary functions that take a natural number n and term indexed by $[n]$.

Ellipses can be represented very elegantly now, e.g., $a_1, \dots, a_n$ is simply $\lambda i. a_i$. Moreover, contrary to all of the above, the length of a sequence is statically known, which permits static index-within-bounds checking when accessing an element of a sequence. Quantification only affects sequences of a certain length, e.g., $\forall x : [n] \to A. F$; to quantify over all sequences, we can use $\forall n. \forall x : [n] \to A. F$.

A disadvantage is the substantial commitment at the language level, which goes far beyond simply adding sequences: The language must be able to express the types $[n]$ and $[n] \to type$ (e.g. via inductive constructions and a universe hierarchy in Coq).

Type sequences We introduce a novel approach: we use term sequences $a_1, \ldots, a_n$ that are typed component-wise by a type sequence $A_1, \ldots, A_n$. Importantly, type sequences $A_1, \ldots, A_n$ are not types themselves – they are simply sequences of types.

Like sequences and contrary to indexed types and list types, this has the advantage that we do not change the underlying type theory. No representational artifacts are needed to flatten sequences or to apply a function to a sequence of arguments. And like for indexed types, the length of a sequence is statically known.

1.4 Flexary Meta-Languages

For content representations of flexary operators, the previous section discussed which representational primitives to use. An orthogonal question is at which language level they should be introduced.

Consider the first-order theory of monoids in which we want to define the flexary version of the composition operator in the obvious way. This should also include the power $a^n = a \circ \ldots \circ a$ for a natural number n as a special case. We might do that by importing the theory of sequences and then using some kind of induction. But this is awkward because the theory of monoids would become much more complex. We might even say that it becomes polluted by the imported operations.

We might try to move the definition to a special enriched theory, which includes both sequences and monoids. But that would contradict mathematical practice, where the definition of the flexary composition is likely to be found in the same paragraph where the binary one is.

Thus, we should add sequences to the logic as a fixed interpreted sort. However, now a similar argument applies: The logic is complicated. Moreover, it does not account for the fact that we would like to use sequences in any logic. Therefore, our goal is to add sequences at the level of the logical framework. This corresponds most closely to informal mathematics where sequences and ellipses are assumed to be given at the informal meta-level and not explicitly defined at the logical or set theoretical level.

The approach of type sequences is most suitable in this respect because it is already orthogonal to the type theoretical and logical foundations. Thus, it can be added easily as a framework feature. Once we go down that road, it also becomes very easy and natural to add constructors for sequence and nesting ellipses at the framework level.

1.5 Overview

Following the above analysis, we develop a logical framework with type sequences. We choose the logical framework LF [HHP93] as an example logical framework since it has been used to represent a large variety of formalisms. But our approach can be transfered to other frameworks (e.g. Isabelle [Pau94]) as well, because it is orthogonal to the underlying type theory.

We briefly summarize LF in Sect. 2 and then extend it to LFS (LF with sequences) in Sect. 3. Notably, our extension is minimally invasive, keeping the essence of the LF type theory unchanged (and reusing the existing rules). We use LFS in Sect. 4 to define flexary versions of first-order logic and simple type theory. In both cases, we declare flexary versions of all operators (where reasonable) and show that LFS can formally define the flexary versions in terms of the fixary ones. Finally, we use our two flexary logics to formalize a collection of common mathematical examples in Sect. 5.

2 The Edinburgh Logical Framework

In this section, we briefly revisit LF [HHP93], a dependently-typed λ-calculus that can be used well as a logical framework [Pfe01]. We give the LF grammar in Fig. 1. LF expressions are grouped into kinds K, kinded type-families $U : K$, and typed terms $S : U$. The kinds are the base kind `type` and the dependent function kinds $\Pi x : U. K$. The type families are the symbols a, the dependent function type $\Pi x : U. V$, abstractions $\lambda x : U. V$, applications $U\ S$; type families of kind `type` are called types. The terms are symbols x, abstractions $\lambda x : U. S$, and applications $S\ T$. Signatures Σ consist of typed or kinded symbols $x : U[= S]$ or $a : K[= U]$ with optional definiens.

As usual, we write $U \to E$ instead of $\Pi x : U. E$ if x does not occur free in E, and we omit the types of bound variables if they can be inferred. Free variables are implicitly bound on the outside of the expression (implicit arguments). Substitution of T for x in E is written $[T/x]E$.

Kinds	K	$::=$	`type` $\mid \Pi x : U. K$
Type Families	$U, V \ldots$	$::=$	$a \mid \Pi x : U. V \mid \lambda x : U. V \mid U\ S$
Terms	$S, T \ldots$	$::=$	$x \mid \lambda x : U. S \mid S\ T$
Signatures	Σ	$::=$	$\cdot \mid \Sigma, x : U[= S] \mid \Sigma, a : K[= U]$

Fig. 1. LF Grammar

We use the signatures given in Fig. 2 as running examples throughout this paper. The signature SFOL on the left defines the syntax and proof rules of sorted first-order logic. The signature STT on the right defines the syntax and β-conversion rule of simple type theory.

In order to emphasize the similarity between LF and LFS, we also give the judgments for well-formed LF expressions in Fig. 3 and the inference rules in Fig. 4. For brevity, we omit the equality judgement, whose rules consist of equivalence, congruence, and $\alpha\beta\eta$-conversion.

$$
\begin{array}{ll}
\mathit{sort} & : \mathtt{type} \\
\mathit{tm} & : \mathit{sort} \to \mathtt{type} \\
\mathit{form} & : \mathtt{type} \\
\mathit{ded} & : \mathit{form} \to \mathtt{type} \\
\\
\mathit{true} & : \mathit{form} \\
\wedge & : \mathit{form} \to \mathit{form} \to \mathit{form} \\
\forall & : \Pi S : \mathit{sort}.\, (\mathit{tm}\, S \to \mathit{form}) \to \mathit{form} \\
\\
\mathit{true}_I & : \mathit{ded}\, \mathit{true} \\
\wedge_I & : \mathit{ded}\, F \to \mathit{ded}\, G \to \mathit{ded}\, F \wedge G \\
\wedge_{El} & : \mathit{ded}\, F \wedge G \to \mathit{ded}\, F \\
\wedge_{Er} & : \mathit{ded}\, F \wedge G \to \mathit{ded}\, G \\
\forall_I & : (\Pi x : \mathit{tm}\, S.\, \mathit{ded}\, F\, x) \to \mathit{ded} \forall F \\
\forall_E & : \mathit{ded} \forall F \to \Pi x : \mathit{tm}\, S.\, \mathit{ded}\, F\, x
\end{array}
$$

$$
\begin{array}{ll}
\mathit{tp} & : \mathtt{type} \\
\mathit{tm} & : \mathit{tp} \to \mathtt{type} \\
\Longrightarrow & : \mathit{tp} \to \mathit{tp} \to \mathit{tp} \\
= & : \mathit{tm}\, A \to \mathit{tm}\, A \to \mathtt{type} \\
\mathit{lam} & : \mathit{tm}\, A \to \mathit{tm}\, B \to \mathit{tm}\, (A \Longrightarrow B) \\
\mathit{app} & : \mathit{tm}\, (A \Longrightarrow B) \to \mathit{tm}\, A \to \mathit{tm}\, B \\
\mathit{beta} & : \mathit{app}\, (\mathit{lam}(\lambda x : \mathit{tm}\, A.T))\, X = (T\, X)
\end{array}
$$

Fig. 2. LF Signatures for SFOL (left) and STT (right)

Judgment	Meaning
$\Sigma \vdash S : U$	S is a well-formed term of type U over Σ
$\Sigma \vdash U : K$	U is a well-formed type family of kind K over Σ
$\Sigma \vdash K\ \mathit{Kind}$	K is a well-formed kind over Σ.

Fig. 3. LF Judgments

$$\frac{}{\Sigma \vdash \mathtt{type}\ \mathit{Kind}}\ \mathit{baseKind}$$

$$\frac{\Sigma \vdash U : \mathtt{type} \quad \Sigma, x : U \vdash V : \mathtt{type}}{\Sigma \vdash \Pi x : U.\, V : \mathtt{type}}\ \mathit{depType} \qquad \frac{\Sigma \vdash U : \mathtt{type} \quad \Sigma, x : U \vdash K\ \mathit{Kind}}{\Sigma \vdash \Pi x : U.\, K\ \mathit{Kind}}\ \mathit{depKind}$$

$$\frac{\Sigma \vdash U : \mathtt{type} \quad \Sigma, x : U \vdash S : V}{\Sigma \vdash \lambda x : U.\, S : \Pi x : U.\, V}\ \mathit{termAbstr} \qquad \frac{\Sigma \vdash U : \mathtt{type} \quad \Sigma, x : U \vdash V : K}{\Sigma \vdash \lambda x : U.\, V : \Pi x : U.\, K}\ \mathit{typeAbstr}$$

$$\frac{\Sigma \vdash S : \Pi x : U.\, V \quad \Sigma \vdash T : U}{\Sigma \vdash S\, T : [T/x]V}\ \mathit{termAppl} \qquad \frac{\Sigma \vdash V : \Pi x : U.\, K \quad \Sigma \vdash T : U}{\Sigma \vdash V\, T : [T/x]K}\ \mathit{typeAppl}$$

Fig. 4. LF Inference Rules

3 A Flexary Logical Framework

3.1 Natural Numbers

In this section we extend LF to form our logical framework LFS (LF with Sequences). LFS adds type sequences and ellipsis constructors.

Therefore, we also need natural numbers as indices to access elements of sequences and to form ellipses. We do this by assuming that the LF signature on the right is always present. Moreover, we assume declarations that formalize the usual computation rules to normalize expressions of type `nat`.

$$
\begin{array}{ll}
\texttt{nat} & : \texttt{type} \\
\leq & : \texttt{nat} \to \texttt{nat} \to \texttt{type} \\
= & : \texttt{nat} \to \texttt{nat} \to \texttt{type} \\
0 & : \texttt{nat} \\
1 & : \texttt{nat} \\
+ & : \texttt{nat} \to \texttt{nat} \to \texttt{nat} \\
- & : \Pi n : \texttt{nat}.\, \Pi m : \texttt{nat}.\, m \leq n \to \texttt{nat}
\end{array}
$$

Note that $-$ is a partial subtraction operator: It takes as a third argument a proof that $n - m$ is defined. We will omit that argument whenever the needed proof is straightforward. This restriction guarantees that we work with natural numbers but not with negative integers. We additionally assume proof irrelevance, i.e., an axiom of type $-(m, n, P) = -(m, n, Q)$. Moreover, we will use the symbols $\leq$, $=$, $+$, and $-$ in infix notation.

Here, for simplicity, we do not formally restrict the use of natural numbers within LFS. However, we consider their status to be similar to that of types. In particular, we assume that all terms of type `nat` refer only to the above signature and free variables introduced in the toplevel context. Since we do also not use multiplication, this keeps our language of natural numbers decidable.

3.2 Syntax

Kinds	K	$::=$	$\texttt{type}^S \mid \Pi x : U.\, K$
Type Seq. Families	U, V	$::=$	$\cdot \mid U, V \mid \underline{U_S} \mid [U]_{x=1}^{S} \mid a \mid \Pi x : U.\, V \mid \lambda x : U.\, V \mid U\, S$
Term Sequences	S, T	$::=$	$\cdot \mid S, T \mid S_T \mid [S]_{x=1}^{T} \mid \circ S \mid x \mid \lambda x : U.\, S \mid S\, T$

Fig. 5. LFS Grammar

We can now give the grammar of LFS in Fig. 5 by extending the grammar of LF. All productions for LF are retained but generalized to sequences. All our extensions are underlined: The singly underlined productions add sequences, the doubly underlined ones add ellipses.

The **term sequences** S, T are formed by the **empty term sequence** $\cdot$ and **concatenation** S, T . If $n : \texttt{nat}$, then S_n accesses the **n-th element** of a sequence S. **Type sequences** are formed in the same way. We write E^n for the sequence $[E]_{x=1}^{n}$ if x does not occur free in E.

If a type family sequence has a length other than 1, all its elements will be types. Consequently, the only **kind sequences** we need are $\texttt{type}, \ldots, \texttt{type}$, which we write as $\texttt{type}^S$ for $S : \texttt{nat}$. We recover $\texttt{type}$ as an abbreviation for $\texttt{type}^1$.

The term **sequence ellipses** constructor is $[S(x)]_{x=1}^{n}$. It takes an argument $n : \texttt{nat}$ and binds the symbol $x : \texttt{nat}$ in S. Its intended meaning is that it reduces to the term sequence $S(1), \ldots, S(n')$ whenever n reduces to a natural number n'. We use an analogous constructor $[U(x)]_{x=1}^{n}$ for type sequence ellipses.

The constructor for **nesting ellipses** is more complicated. After several experiments, we opted for a **flexary function composition** operator as a primitive concept. The intended meaning of $\circ\, S$ is that it takes a sequence of functions and returns their composition. Thus, $\circ\,(f_1, \ldots, f_n)\, s$ reduces to $f_n\,(\ldots(f_1\, s)\ldots)$. Notably, this is more general than a fold operator because the type of each f_i may depend on i.

Finally, we have to clarify the intuitions of the now-flexary primitives of LF. **Flexary variable bindings** $x : U$ formalize variable sequences, i.e., binding $x : (U_1, \ldots, U_n)$ corresponds to binding $x_1 : U_1, \ldots, x_n : U_n$. Thus, the type sequence ellipses immediately induces a corresponding ellipses constructor for variable bindings. Accordingly, a **flexary application** $f\, T$ applies a function f to an argument sequence T.

$\lvert x\rvert$	$= \lvert U\rvert$ if $x : U$ in Σ
$\lvert a\rvert$	$= \lvert K\rvert$ if $a : K$ in Σ
$\lvert\texttt{type}^n\rvert$	$= n$
$\lvert \Pi x : E.\, F\rvert$	$= 1$
$\lvert \lambda x : E.\, F\rvert$	$= 1$
$\lvert E\, F\rvert$	$= 1$
$\lvert \cdot \rvert$	$= 0$
$\lvert\ E, F\ \rvert$	$= \lvert E\rvert + \lvert F\rvert$
$\lvert E_n\rvert$	$= 1$
$\lvert [E]_{x=1}^{n}\rvert$	$= \lvert n\rvert$
$\lvert \circ\ S\rvert$	$= 1$

Fig. 6. Length of a Sequence

Much of the intuition behind our grammar becomes clear from the function $|E|$ for the **length of a sequence** defined in Fig. 6. It maps LFS expressions to expressions of type $\texttt{nat}$ (where E and F range over any expression allowed by the grammar). We already mention that the type system given below will respect length, i.e., $S : U : \texttt{type}^n$ only if $|S|$, $|U|$, and n are provably equal.

Functions f and applications $f\, T$ always have length 1 and so have the bodies of the binders. This forbids computations that returns sequences. This restriction could be lifted, but we find it is more reasonable to introduce such computations in object languages defined within LFS.

Before giving the type system, we fortify our intuitions by defining a few useful abbreviations that we will use later on. The **reversal** of a sequence is defined by:

$$\texttt{revert}\, E = [E_{|E|+1-i}]_{i=1}^{|E|}.$$

The **generalized sequence ellipses** $a_n, \ldots, a_1$ and $a_m, \ldots, a_n$ (if $m \leq n$ for natural numbers m, n) are defined by

$$[E(x)]_1^{x=n} = \texttt{revert}\,[E(x)]_{x=1}^{n} \qquad [E(x)]_{x=m}^{n} = [E(m+x-1)]_{x=1}^{n-m+1}$$

We obtain the usual fold operator in terms of flexary composition:

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

Thus, $\texttt{foldl}\, f\, S\, a$ reduces to $(f \ldots (f\, (f\, a\, S_1)\, S_2) \ldots S_n)$ for a folding function $f : A \to B \to A$, a start element $a : A$, and a sequence $S : B^n$. `foldr` is defined analogously.

3.3 Type System

LFS uses the same judgments as LF, i.e. the ones from Fig. 3. However, we will write the typing judgment $\Sigma \vdash S : U$ (where $U : \mathit{type}$ is implied in LF) as $\Sigma \vdash S : U : \texttt{type}^n$ to keep track of the length of S and U. This is redundant because the length is statically known anyway, but it makes the notations much simpler.

Most importantly, term sequences are typed by type sequences of the same length, and type sequences are kinded by type^n, where n is their length.

The inference rules essentially reuse the rules of LF from Fig. 4. We only make two minor changes to the four rules for binders. Firstly, the four occurrences of `type` are replaced with $\texttt{type}^n$ for $\Sigma \vdash n : \texttt{nat}$. This permits binders to bind variables sequences $x : U_1, \ldots, U_n$. The LF rules are recovered as the special case $n = 1$. Secondly, we add a premise to each of the four rules that ensure that the body of a binder always has length 1.

Then we add the rules of Fig. 7 for sequences and ellipses. *kindSeq* makes type^n a valid kind. The rules for the empty sequences and the concatenation of sequences are obvious. The rules *termIndex* and *typeIndex* for taking an element of a sequence implement the index-within-bounds check: E_i is only valid if $1 \le i \le |E|$.

The rules *termSeqEll* and *typeSeqEll* handle the sequence ellipsis. Note that $[E]_{x=1}^{n}$ actually binds three variables in E: The index x and two assumptions x_* and x^* that guarantee that x is within 1 and n. These assumptions can be used later on to discharge the proof obligations posited by the rules *termIndex* and *typeIndex* and by the subtraction of natural numbers.

Finally, *nestEll* handles the nesting ellipsis $\circ S$ by checking that the function in S are actually composable. This is easiest if we restrict attention to the composition of simple functions.

We only sketch the **conversion rules** that we add to the ones of LF. Binders distribute over sequences:

$$\lambda x : U, V.\, E = \lambda x_1 : U.\, \lambda x_2 : V.\, [x_1, x_2/x]T \qquad \lambda x : \cdot.\, T = [\cdot/x]T$$

and similarly for Π. Sequence elements can be projected if the sequence is normal

$$(E^1, \ldots, E^n)_x = E_x \qquad \text{if } |E^1| = 1, \ldots, |E^n| = 1$$

Ellipses are expanded if enough information is available:

$$[E]_{x=1}^{n} = [1/x]E, \ldots, [n/x]E \qquad \text{if } n = 1 + \ldots + 1$$

Fig. 7. Inferece Rules for Sequences and Ellipses

$$\circ\, \cdot = \lambda x.\, x \qquad \circ\, (f, g) = \lambda x.\, (\circ\, g)\, ((\circ\, f)\, x) \qquad \circ\, f = f \text{ if } |f| = 1$$

These conversions have the effect that LFS is conservative over LF in the following sense: If $\Sigma \vdash S{:}U : \mathtt{type}^n$ and all terms of type $\mathtt{nat}$ reduce to $1 + \ldots + 1$, then n reduces to m, and S and U reduce to $S^1, \ldots S^m$ and $U^1, \ldots, U^m$, and $S^i : U^i$ for $i = 1, \ldots, m$ in the LF type theory. This means that if the involved natural number expressions normalize, then LFS-expressions reduce to sequences of LF-expressions, and LFS-judgments reduce to sequences of LF-judgments. Under this condition, the canonical LFS expressions are sequences of canonical LF expressions. Consequently, an adequate encoding of objects as LF-expressions, yields an adequate encoding of sequences of objects as LFS-expressions.

Reducing full LFS to LF would require a formalization of sequences in LF already, which is doable, but also very costly.

4 Flexary Logics

Now we use LFS to define the flexary analogues of two languages commonly used for formalized mathematics.

We define **flexary sorted first-order logic** SFOL* by extending the syntax of SFOL from Fig. 2 with flexary symbols $\wedge^*$, $\Rightarrow^*$ and $\forall^*$ along with their proof rules. All are defined in terms of their fixary counterparts.

$$\begin{aligned}
\wedge^* &: \mathit{form}^n \to \mathit{form} \\
&= \lambda F : \mathit{form}^n.\, \texttt{foldl}\ \wedge\ F\ \mathit{true} \\
\wedge^*_I &: \Pi F : \mathit{form}^n.\, [\mathit{ded}\ F_x]_{x=1}^n \to \mathit{ded} \wedge^* F \\
&= \lambda n.\, \lambda F.\, \lambda D : [\mathit{ded}\ F_x]_{x=1}^n. \\
&\qquad \circ [\lambda x : \mathit{ded}\ (\wedge^*\ [F_j]_{j=1}^{i-1}).\ \wedge_I\ \underline{(\wedge^*\ [F_j]_{j=1}^{i-1})}\ \underline{F_i}\ x\ D_i]_{i=1}^n\, \mathit{true}_I \\
\wedge^*_E &: \Pi F : \mathit{form}^n.\, \Pi i : \texttt{nat}.\, \Pi i_* : 1 \leq i.\, \Pi i^* : i \leq n.\, \mathit{ded} \wedge^* F \to \mathit{ded}\ F_i \\
&= \lambda F.\lambda i.\, \lambda i_*.\lambda i^*.\, \lambda D.\ \wedge_{Er}\ \underline{(\wedge^*\ [F_k]_{k=1}^{i-1})}\ \underline{F_i} \\
&\qquad \circ [\lambda x.\ \wedge_{El}\ \underline{(\wedge^*\ [F_k]_{k=1}^{n-j-1})}\ \underline{F_{n-j}}\ x]_{j=1}^{n-i}\, D
\end{aligned}$$

abbreviation: $S_a^b := [\mathit{tm}\ S_j]_{j=a}^b$

$$\begin{aligned}
\forall^* &: \Pi S : \mathit{sort}^n.\, (S_1^n \to \mathit{form}) \to \mathit{form} \\
&= \lambda S.\, \lambda F.\ \circ\ [\lambda f : S_1^i \to \mathit{form}.\, \lambda y : S_1^{i-1}.\, \forall \lambda x : \mathit{tm}\ S_i.\ f\ (y, x)]_1^{i=n}\, F \\
\forall^*_I &: (\Pi x : S_1^n.\, \mathit{ded}\ F\ x) \to \mathit{ded} \forall^*\ F \\
&= \lambda D.\ \circ\ [\lambda d : \big(\Pi y : S_1^i.\, \mathit{ded} \forall^* \lambda x : S_{i+1}^n.\, D\ (y, x)\big). \\
&\qquad\qquad \lambda y : S_1^{i-1}.\, \forall_I\, \lambda x : \mathit{tm}\ S_i.\ d\ (y, x) \\
&\qquad]_1^{i=n}\, D \\
\forall^*_E &: \mathit{ded} \forall^*\ F \to \Pi x : S_1^n.\, \mathit{ded}\ F\ x \\
&= \lambda D.\lambda x.\ \circ\ [\lambda d : \mathit{ded} \forall^* \lambda y : S_i^n.\, F([x_j]_{j=1}^{i-1}, y).\, \forall_E\, d\, x_i]_{i=1}^n\, D
\end{aligned}$$

The flexary conjunction $\wedge^*$ takes a natural number n and then a sequence of n conjuncts. We have $\wedge^* F_1\ \ldots, F_n = (\ldots(\mathit{true} \wedge F_1)\ldots) \wedge F_n$ and $\wedge^* \cdot = \mathit{true}$. The introduction rule uses essentially the same folding: Without implicit arguments, it would simply read $\wedge^*_I\, D_1\ \ldots, D_n = \wedge_I\, (\ldots(\wedge_I\, \mathit{true}_I\, D_1)\ldots)\, D_n$ and $\wedge^*_I \cdot = \mathit{true}I$. However, to demonstrate that it is in fact definable, we also give the implicit arguments in detail: They are underlined. That complicates the definition because each occurrence of $\wedge_I$ uses different implicit arguments, which themselves need ellipses to write down. The elimination rule proceeds along the same lines except that we have to take guards i_* and i^* to make sure the indices F_i are within bounds.

For disjunction, we would use $\vee^*\ F = \texttt{foldl}\ \vee\ F\ \mathit{false}$ accordingly. For implication, which is not associative, we define $\Rightarrow^*$: $\mathit{form}^n \to \mathit{form} \to \mathit{form}$ and $\Rightarrow^*\ F\, G = \texttt{foldr}\ \Rightarrow\ F\, G$.

The definition of flexary quantifiers is more involved. Intuitively, we want $\forall^*\, S\, F = \forall \lambda x^1 : S_1. \ldots \forall \lambda x^n : S_n.\, F(x^1, \ldots, x^n)$. Let $[\circ\, G(i)]_1^{i=n}$ be the ellipsis

in the definiens of $\forall^*$. Then the type of $G(i)$ is $(S_1^i \to \mathit{form}) \to (S_1^{i-1} \to \mathit{form})$, and when constructing $G(n)\,(\ldots(G(1)\,F)\ldots)$, each $G(i)$ introduces $\forall x^i$. Note that all variables are called x, the names x^i are introduced when LFS α-renames x during capture-avoiding substitution.

The definitions of the proof rules are conceptually straightforward but equally subtle. The flexary existential quantifier can be defined in the same way.

Next, we define **flexary simple type theory** STT* by extending the syntax of STT from Fig. 2. We define a flexary function type constructor, flexary λ-abstraction, and flexary application in terms of the fixary ones. We omit the proof of the flexary β-reduction.

$$\begin{array}{lll}
\Longrightarrow^* & : & \mathit{tp}^n \to \mathit{tp} \to \mathit{tp} \\
& = & \lambda A : \mathit{tp}^n.\, \lambda B : \mathit{tp}.\, \texttt{foldr} \implies A\, B \\
\mathit{lam}^* & : & ([\mathit{tm}\, A_i]_{i=1}^n \to \mathit{tm}\, B) \to \mathit{tm}\, (A \Longrightarrow^* B) \\
& = & \lambda F.\ \circ\, [\lambda f : [\mathit{tm}\, A_j]_{j=1}^i \to \mathit{tm}\, \big([A_j]_{j=i+1}^n \Longrightarrow^* B\big). \\
& & \qquad \lambda y : [\mathit{tm}\, A_j]_{j=1}^{i-1}.\, \mathit{lam}\, \lambda x : \mathit{tm}\, A_i.\ f\,(y,x) \\
& & \quad]_1^{i=n}\, F \\
\mathit{app}^* & : & \mathit{tm}\, (A \Longrightarrow^* B) \to [\mathit{tm}\, A_i]_{i=1}^n \to \mathit{tm}\, B \\
& = & \lambda F.\, \lambda a.\ \circ\, [\lambda f : A_i^n \implies B.\, \mathit{app}\, f\, x_i]_{i=1}^n\, F \\
\mathit{beta}^* & : & \mathit{app}^*\, (\mathit{lam}^*(\lambda x : [\mathit{tm}\, A_i]_{i=1}^n.T))\, X = (T\, X) = \ldots
\end{array}$$

5 Flexary Mathematics

Now we use SFOL* and STT* to formalize a collection of common mathematical examples.

Monoid Operations Like we did for conjunction, we can define the flexary version of any associative binary operator. Consider a monoid with carrier α : `type`, binary operation $\circ : \alpha \to \alpha \to \alpha$ and unit element $e : \alpha$. Then we define the flexary operation $\circ^*$ as follows:

$$\circ^* \ : \ \Pi n : \texttt{nat}.\, \alpha^n \to \alpha \ = \ \lambda n.\, \lambda x : \alpha^n.\, \texttt{foldl}\ \circ\, x\, e$$

This immediately yields the power operator:

$$\mathit{power} \ : \ \alpha \to \texttt{nat} \to \alpha \ = \ \lambda x.\, \lambda n.\ \circ^*\ x^n$$

By specializing to the monoid of endofunctions on a type, we obtain the iteration of functions as follows:

$$\mathit{iter} \ : \ (\alpha \to \alpha) \to \texttt{nat} \to (\alpha \to \alpha) \ = \ \lambda f.\, \lambda n.\ \circ\ f^n$$

Multi-relations Multi-relations like $a \in b \subseteq c$ are routinely used in informal mathematics but cannot be defined as single operators within a fixary logic. They can only be defined as notations within an implementation of the logic. Using SFOL*, we can define it as a flexary operator elegantly:

$$\begin{aligned} \mathit{multirel} :\ & \Pi A : \mathit{sort}^n.\, (\mathit{tm}\, A)^{n+1} \to (\mathit{tm}\, A \to \mathit{tm}\, A \to \mathit{form})^n \to \mathit{form} \\ & = \lambda x : (\mathit{tm}\, A)^{n+1}.\, \lambda r : (\mathit{tm}\, A \to \mathit{tm}\, A \to \mathit{form})^n.\, \wedge^* [r_i\, x_i\, x_{i+1}]_{i=1}^n \end{aligned}$$

For example, we can now write the above multi-relation as $\mathit{multirel}\,(a, b, c)\,(\in, \subseteq)$. More generally, we can define multi-relations for relations between different types:

$$\begin{aligned} \mathit{multirel}' :\ & \Pi A : \mathit{sort}^n.\, [\mathit{tm}\, A_i]_{i=1}^{n+1} \to [\mathit{tm}\, A_i \to \mathit{tm}\, A_{i+1} \to \mathit{form}]_{i=1}^n \to \mathit{form} \\ & = \lambda A.\, \lambda x.\, \lambda r.\, \wedge^* [r_i\, x_i\, x_{i+1}]_{i=1}^n \end{aligned}$$

Vectors and Matrices Now we formalize vectors in STT*. For simplicity, we assume a fixed base type a with the structure of a ring as given on the right

$$\begin{array}{ll} a' & : \mathit{tp} \\ a & : \mathtt{type} = \mathit{tm}\, a' \\ 0_a & : a \\ 1_a & : a \\ + & : a \to a \to a \\ \times & : a \to a \to a \end{array}$$

Then we axiomatize a type constructor of fixed-length vectors as follows, where n is an implicit argument everywhere:

$$\begin{array}{ll} \mathit{Vec}' & : \mathtt{nat} \to \mathit{tp} \\ \mathit{Vec} & : \mathtt{nat} \to \mathtt{type} = \lambda n.\, \mathit{tm}\,(\mathit{Vec}'\, n) \\ \mathit{vec} & : a^n \to \mathit{Vec}\, n \\ \mathit{index} & : \Pi m : \mathtt{nat}.\, \Pi m_* : 1 \le m.\, \Pi m^* : m \le n.\, \mathit{Vec}\, n \to a \\ \mathit{compute} & : \Pi m : \mathtt{nat}.\, \Pi m_* : 1 \le m.\, \Pi m^* : m \le n.\, \Pi x : a^n. \\ & \qquad \mathit{index}\, m\, p\, q\, (\mathit{vec}\, x) = x_m \\ \mathit{complete} & : \Pi v :\ \mathit{Vec}\, n.\, v\ =\ \mathit{vec}\, [\mathit{index}\, i\, i^*\, i_*\, v]_{i=1}^n \end{array}$$

It may appear strange that we call $\mathit{vec} : \mathtt{nat} \to \mathit{tp}$ an example in *simple* type theory. However, recall that `nat` is not a type of the logic STT* but a feature of the framework. Thus, there is no substantial structural difference between our *vec* and type operators like $\mathit{list} : \mathit{tp} \to \mathit{tp}$. Indeed, our treatment of implicit arguments of type `nat` is very similar to the treatment of free type variables in higher-order logics.

We can now define the addition and scalar multiplication elegantly:

$$\begin{aligned} \vec{+} :\ & \mathit{Vec}\, n \to \mathit{Vec}\, n \to \mathit{Vec}\, n \\ & = \lambda v.\, \lambda w.\, \mathit{vec}\, [(\mathit{index}\, i\, i_*\, i^*\, v) + (\mathit{index}\, i\, i_*\, i^*\, w)]_{i=1}^n \\ \vec{\times} :\ & a \to \mathit{Vec}\, n \to \mathit{Vec}\, n \\ & = \lambda x.\, \lambda v.\, \mathit{vec}\, [x \times (\mathit{index}\, i\, i_*\, i^*\, v)]_{i=1}^n \end{aligned}$$

In implementations, these definitions would resemble the ones in informal mathematics even more: We would use appropriate notations for *vec* and *index*, and it is straightforward for a theorem prover to find the guard arguments of *index* automatically.

More generally, we can formalize the type of vectors $\mathit{Vec} : \mathit{tp} \to \mathtt{nat} \to \mathit{tp}$ over an arbitrary base type and define matrices as vectors of vectors. Then matrix addition and multiplication can be defined accordingly.

Polynomials Finally, we axiomatize a type of polynomials (over the same fixed base type $a : \mathtt{type}$ as above) as follows:

$$
\begin{array}{ll}
\mathit{Poly}' & : \mathit{tp} \\
\mathit{Poly} & : \mathtt{type} = \mathit{tm}\ \mathit{Poly}' \\
\mathit{poly} & : a^{n+1} \to \mathit{Poly} \\
\mathit{deg} & : \mathit{Poly} \to \mathtt{nat} \\
\mathit{coeff} & : \mathit{Poly} \to \mathtt{nat} \to a \\
\mathit{comp\_deg} & : \Pi c : a^{n+1}.\ \mathit{deg}\,(\mathit{poly}\,c) = n \\
\mathit{comp\_coeff}_1 & : \Pi c : a^{n+1}.\ \Pi m : \mathtt{nat}. \\
 & \qquad 1 \le m \to m \le n+1 \to \mathit{coeff}\,(\mathit{poly}\,A)\,m = A_m \\
\mathit{comp\_coeff}_2 & : \Pi c : a^{n+1}.\ \Pi m : \mathtt{nat}.\ n+2 \le m \to \mathit{coeff}\,(\mathit{poly}\,A)\,m = 0_a \\
\mathit{complete} & : \Pi p : \mathit{Poly}.\ p = \mathit{poly}\,[\mathit{coeff}\ p\ i]_{i=1}^{1+deg\ p}
\end{array}
$$

Here *poly* constructs a polynomial from coefficients, *deg* returns an upper bound on the degree, and *coeff* returns a coeffiecient. *comp_deg*, $\mathit{comp\_coeff}_1$, and $\mathit{comp\_coeff}_2$ compute the degree and coefficients of an explicitly given polynomial. And *complete* makes every polynomial equal to the one formed from its coefficients.

We can now define the evaluation of a polynomial for a given x concisely:

$$\mathit{eval} : \mathit{poly} \to a \to a = \lambda p.\ \lambda x.\ +^*\ [(\mathit{coeff}\ p\ i) \times (\times^*\ x^{i-1})]_{i=1}^{1+\deg\ p}$$

where $+^*$ and $\times^*$ are the flexary versions of $+$ and $\times$, respectively.

The ring operations on polynomials can be defined accordingly.

6 Conclusion & Future Work

It is almost impossible to write about mathematical objects without using sequence ellipsis (...). This observation is independent of the language used, or the formal system employed: if we eliminate ellipses, then expressions get more complicated and communication can become quite awkward. This universality strongly suggests that sequences and sequence ellipsis are a meta-level feature of mathematical languages.

Guided by this realization, we present LFS, an extension of LF with a novel feature of type sequences. Using the extended framework for logic development enables us to specify flexary logics with flexary operators and calculi that deal with them in proofs. We exemplify this ability with flexary sorted first-order logic and flexary simple type theory. As they use LFS as their meta-language, both can define flexary operators in terms of their fixary counterparts. A central theme is that the type of a flexary operator depends on a natural number argument, which instantiates the flexible arity: We call this **arity polymorphism** because it is very similar to the well-known type polymorphism where the type of an operator depends on a type argument.

Numerous examples from everyday mathematics show that the flexary languages allow more adequate formalizations of complex objects like vectors, matrices, and polynomials.

In the future, we want to extend/complete MKM support for LFS. In particular, we want to *i*) look at additional examples, e.g. the complex matrix representations in [SS06], *ii*) investigate type reconstruction of LFS and implement

it based on MMT [RK13; Rab13], which will in particular infer omitted natural number arguments, *iii*) extend the support for sequences and elisions to flexiformal mathematics markup systems like LaTeX and sTeX, *iv*) integrate native support for argument sequences into OpenMath and content MathML completing the work started in [HKR11], and finally *v*) develop semantics reconstruction techniques that transform $1, 2, \ldots, n$ into $[i]_{i=1}^{n}$ or $1, 2, 4, 8, \ldots$ into $[2^i]_{i=1}^{\infty}$.

Acknowledgements Work on the concepts presented here has been partially supported by the German Research Foundation (DFG) under grant KO 2428/13-1.

References

[Aus+03] R. Ausbrooks et al. *Mathematical Markup Language (MathML) Version 2.0 (second edition)*. See http://www.w3.org/TR/MathML2. 2003.

[CH88] T. Coquand and G. Huet. "The Calculus of Constructions". In: *Information and Computation* 76.2/3 (1988), pp. 95–120.

[CICM1212] Artur Kornilowicz. "Tentative Experiments with Ellipsis in Mizar". In: *Intelligent Computer Mathematics*. Conferences on Intelligent Computer Mathematics (CICM). (Bremen, Germany, July 9–14, 2012). Ed. by Johan Jeuring et al. LNAI 7362. Berlin and Heidelberg: Springer Verlag, 2012, pp. 453–457. ISBN: 978-3-642-31373-8.

[Com] *Information technology — Common Logic (CL): a framework for a family of logic-based languages*. Tech. rep. 24707:2007. ISO/IEC, 2007.

[HHP93] 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.

[HKR11] F. Horozal, M. Kohlhase, and F. Rabe. "Extending OpenMath with Sequences". In: *Intelligent Computer Mathematics, Work-in-Progress Proceedings*. Ed. by A. Asperti et al. University of Bologna, 2011, pp. 58–72.

[KB04] Temur Kutsia and Bruno Buchberger. "Predicate Logic with Sequence Variables and Sequence Function Symbols". In: *Mathematical Knowledge Management, MKM'04*. Ed. by Andrea Asperti, Grzegorz Bancerek, and Andrej Trybulec. LNAI 3119. Springer Verlag, 2004, pp. 205–219.

[ML74] P. Martin-Löf. "An Intuitionistic Theory of Types: Predicative Part". In: *Proceedings of the '73 Logic Colloquium*. North-Holland, 1974, pp. 73–118.

[Nor05] U. Norell. *The Agda WiKi*. http://wiki.portal.chalmers.se/agda. 2005.

[Pau94] L. Paulson. *Isabelle: A Generic Theorem Prover*. Vol. 828. Lecture Notes in Computer Science. Springer, 1994.

[Pfe01] F. Pfenning. “Logical frameworks”. In: *Handbook of automated reasoning*. Elsevier, 2001, pp. 1063–1147.

[PS99] F. Pfenning and C. Schürmann. “System Description: Twelf - A Meta-Logical Framework for Deductive Systems”. In: *Lecture Notes in Computer Science* 1632 (1999), pp. 202–206.

[Rab13] F. Rabe. “The MMT API: A Generic MKM System”. In: *Intelligent Computer Mathematics*. Ed. by J. Carette et al. Springer, 2013, pp. 339–343.

[RK13] F. Rabe and M. Kohlhase. “A Scalable Module System”. In: *Information and Computation* 230.1 (2013), pp. 1–54.

[SS06] Alan Sexton and Volker Sorge. “Processing Textbook-Style Matrices”. In: *Mathematical Knowledge Management, MKM'05*. Ed. by Michael Kohlhase. LNAI 3863. Springer Verlag, 2006, pp. 111–125.

[The14] The Coq Development Team. *The Coq Proof Assistant: Reference Manual*. Tech. rep. INRIA, 2014.

[Wol12] Wolfram Research, Inc. *Mathematica 9.0*. 2012.