Syntax for the Working Mathematician

Alexander Kuklev, JetBrains Research

To design a syntax suitable for doing formalized maths, we carefully examined several math-oriented languages (Fortress, Agda, Coq, Lean, Isabelle, Julia and Python) and combined their strengths while leaving out many nuisances. We also introduce two main novelities:

- treating text and code on par by making use of keyword sigils and siginificant indentation;
- sigils for fresh variables: backslash is the keyword sigil in TEX and Arend, but we propose it as freshness sigil to retain the notation $\backslash x \mapsto f(x)$ for closures, and use hash # for keywords.

Mathematicians love the richness of mathematical notation in Fortress, yet prefer monospaced fonts (albeit with ligatures and unicode math characters, greek, etc) classical to programming whenever nested blocks appear. Most mathematicians appreciate some not entirely linear pretty-printing directly in the editor, namely displaying

`x~n~`	as	x_n (Markdown sub-),
`x^n^`	as	x^n (and superscript),
`\x`	as	$\underline{x}$ (underlined fresh variables),
`#Theorem`	as	**`Theorem`** (boldface for keywords),

as well as using italics for single-letter identifiers and removing whitespaces between them whenever multiplication is meant (`2b x + c` as $2bx + c$). They also tend to appreciate the block structure in Coq and Lean and prefer it to **`where`**-blocks of Agda, and enjoy extensive use of Agda-esque mixfix operators with extensive use of unicode, while disliking too restrictive whitespace handling.

1 Treating Text and Code on par

Utilizing GitHub even for one-man projects has become a major trend in the mathematical community over the past few years. Sources became public, rising an incentive to keep them nice and clean. Amenability to diffing and merging also required reducing clutter. Thus, while most quality articles still rely on (La)TEX for rendering, their sources are increasingly often written using unicode-rich Markdown [RFC7763], in particular dialects like Pandoc well suited both for web-based documentation and academic papers utilizing the full strength of TEX. If we want mathematicians to provide formal statements and proofs, those have to be seamlessly embeddable.

In most if not all established programming languages, accompanying text is a second-class citizen in source files; it has to be painstakingly fenced and usually has to adhere various rather burdensome limitations. When writing math papers with formalized definitions, statements and proofs and in literate programming things are turned upside down: one writes a text with code snippets comprising a valid source of program or a library when extracted, i. e. the code is second-class.

But if the code is only contained in the lines beginning with a keyword like "**`fun`**" or "**`class`**", and indented blocks immediatelly following such lines (ignoring empty lines), while the keywords have to begin with a sigil (we propose the hash #),[1] code can be interleaved with text without any fencing:

```
In 1637 on the margin of a copy of Diophantus' Arithmetica a French
mathematician Pierre de Fermat stated the following famous theorem:

Theorem ('Fermat's Tast Theorem') Wiles95
 : ∀(n : ℕ{_> 2}, a b c : ℤ) a^n + b^n ≠ c^n
  ...
```

[1] The section sign § is semantically superior, but doesn't belong to the default keyboard layout.

2 Freshness sigil

Pattern matching as implemented in most languages has a major problem: patterns can contain both variables with already defined values (which are interpolated) and placeholder-variables to be filled with values when matching the pattern, distinguished by freshness. Hence a new constant definition in a distant library far far away can mess any pattern up if it happens to define a constant with the same name as used in the pattern. All of this can be prevented by use of a freshness sigil \ for variables extracted by pattern matching (arguments of named and anonymous functions being a special case). The choice of sigil is to retain the very common notation $\backslash x \mapsto f(x)$ for lambda-expressions/anonymous functions.

Here are some examples to provide an impressions:

```
val (a, b): pair₁

def add: (a, b) ↦ a + b

def incr(n : ℕ): n + 1

def muladd(a b c: ℕ): a·b + c

def _!: ℕ → ℕ
  0 ↦ 1
  (n + 1) ↦ (n + 1) · n!

def id{T}(value : T): value
```

The alternative to freshness sigils widely used in some languages is to distinguish fresh variables by mandatory type annotations — a burden for typing with naming conventions (see `http://agda.readthedocs.io/en/v2.6.0.1/language/generalization-of-declared-variables.html`) and gradual typing/partial type inference, all of which are excellent tools to improve code readability.

3 Renamable Fields

As in Fortress, we allow structures to have renamable fields. Here is the motivating example:

```
Structure Monoid{@on T : *}(compose : T × T → T):
  unit : T
  l-canceler(x : T) : unit ‹compose› x = x
  r-canceler(x : T) : x ‹compose› unit = x
  associator(x y z : T) : (x ‹compose› y) ‹compose› z = x ‹compose› (y ‹compose› z)

 Possible usage:
  def reduce{T : Monoid(∘)}(x y : T):
    x ∘ y

  def muladd{m1 m2 : Monoid{T}}(x y z : T):
    (x ‹m1.compose› y) ‹m2.compose› z

  Structure Group{@on T : *}(\_∘_) extends Monoid{T}(∘):
    ...

  Structure CMonoid{@on T : *}(\_+_) extends Monoid{T}(+):
    ...

  Structure AbGroup{@on T : *} iss Group{T}(+) ∧ CMonoid{T}(+).
```

4 Indentation, Nesting and Line-Breaks

Off-side rule (also known as significant indentation) used as in Python, Haskell, Scala 3 and many other languages allows structuring code blocks with minimal visual clutter. It is often overlooked that off-side rule also allows to embed multiline literals and foreign code without any fencing. We propose to use a variant of off-side rule that (1) explicitly allows such possibility, (2) prohibits visually misguiding use cases, and (3) does not prevent fancy allignment in multiline formulas, etc.

Syntax proposals, Level II govern indentation and line breaks:

- The indentation base of a block (sequence of lines) is the longest prefix consisting entirely of whitespaces and tab characters common to all nonemtpy lines of the block. Base b_2 is said to be strictly longer than b_1 iff $b_2 = b_1 s$, where s contains at least two whitespaces or a tab character. (Initial tabstop size of at least two whitespaces is required.)
- A block can contain nested blocks. Nested block is a sequence of lines with indentation base strictly longer than that of the parent block, such that the lines immedeately above and below the nested block have an equal and shorter indentation base (nested blocks are thus visually well-enclosed).
  Empty lines above a block belong to the block, empty lines below don't.

```
Error-prone due to visual misalignment:

while (a):
  unless (b):
    repeat (c):
      ...
  beep

until (e): ...
```

```
Visually well-enclosed:

while (a):
  unless (b):
    repeat (c):
      ...
    .
  beep

until (e): ...
```

- When the tokenizer encounters a nested block, it should strip its content of its indentation base and box it as a single token: a "block literal". Further processing of the "block literal" should be performed in the parsing phase, when the parser gets to know what kind of entity is expected there (a normal code block, or a multi-line literal/foreign code block) and passes the "block literal" to subordinate handler accordingly.
  In the case of multiline literal, inner indents of the nested block are the concern of the respective handler, but nested blocks containing Level II-compliant code have to keep their inner blocks well-enclosed.

```
def greet(name : String):
  Console.print
    Hello [name]!
  .
```

- In a block of code, lines indented exactly one whitespace beyond the indentation base of the block should be treated as continuations of the previous line, i. e. the linebreak between the lines should be stripped by tokenizer:

```
Technically,
 this is one and
 the same line.
```

- Proportional (non-monospaced) fonts make fancy alignment and Haskell-style indentation with whitespaces impossible. However, IDEs and typesetting software can implement elastic tabstop algorithm to enable complex (in particular, Haskell-style) indentation and fancy alignment with tabs only.

5 Lexical Structure

Lexical structure of a language defines how to split a stream of characters into a stream of tokens to be parsed. Tokenization can be seen as a part of the parsing process, yet it has to be isolated for languages that support user-defined operators and notations, i. e. extend and override their parsing rules on the fly. Our approach can be largely described as "Agda with better (Fortress-inspired) whitespace handling and fancy literals".

Syntax proposals, Level III (Lexical Structure):

- Level II proposals already mandate the approach to indented blocks: they have to be boxed as self-contained tokens to be processed later in the parsing phase by respective handlers.
- Save for comments, quoted tokens and literals, brackets must be balanced.
- With the same exceptions, the stream of characters is split into tokens by whitespaces (one or more characters with unicode property `WSpace=Y`) and token boundaries (see below).
- There is a list of suffix characters: all brackets except '`(`', `!:;,.•`, etc. If such character follows a closing bracket or an alphanumeric character, a token boundary is introduced immediately before this character, and the token is tagged *suffix*, unless the character is irrelevantly suffix (`;,.•`). This rule makes the correct tokenizaion of `m, n!` possible.
- There is a list of prefix characters: opening brackets, `-!@#&$;,.•`, etc. If such character belongs to a non-suffix token and is immediatelly followed by an alphanumeric character or an opening bracket, a token boundary is introduced after it and the token is tagged *prefix*, unless the character is irrelevantly prefix (`;,•`). This facilitates tokenization of `Math.sin -2 •π`, while keepeng prefix and infix minus distinguished by token tags.
- To allow kebab-case and double-dashed identifiers, dashes are considered prefix only after a whitespace: `if (n is-even?) !(p --is-even)`.
- '`\`' `<space>` immediately after a non-prefix token treats the rest of the line as a nested block. It can be used for end-of-line literals: `Text\ Sample text`.
- `<type>` '`\`' begin custom literals spanning to the first whitespace or closing bracket after inner brackets are closed: `List\(d`$_1$`, Date\2014-01-24)`.
- `<tab>` (`'--'` | `<em-dash>`) `<space>` begin end-of-line comments.
  Tab character and a whitespace around the (double) dash are mandatory!
- '`(`' `<backtick>`$^+$ begins an inline comment spanning to `<backtick>`$^+$ '`)`' with the same number of backticks. Inline comments immediatelly preceeding definitions/declarations are considered to be alt-names, e.g.

```
def (‘Factorial‘) _!: ℕ → ℕ
  0 ↦ 1
  (n + 1) ↦ (n + 1) · n!
```

- The single (`'`), double (`"`) or repeated double quotes introduce a legacy string literal spanning until the matching quote or quote sequence appears. Single quote starts a literal only if it appears after a whitespace, newline or a prefix token. Otherwise it can be a part of the token (`state ← state'`).
- '`«`' ... '`»`' are used for "quoted" tokens containing whitespaces, dots, etc.

6 Syntactic Structure

A common syntax framework should strive for maximal flexibility, so we'll take guidance from Agda. Like many languages, besides closed identifiers, Agda supports user-defined prefix (`-n`), postfix (`n!`) and infix (`n + m`) operators with flexible precedence relations (arbitrary DAGs). The novelty of Agda is to allow them to have inner argument "places": one can define the floor function `⌊_⌋`, the "prefix" operator `if_then_else_`, the "postfix" sequence subscripting operator: `_⌊_⌋` and the infix typing relation `_⊢_:_`. This approach can still be extended by allowing "infix" operators to have distinct left and right precedence, and by allowing user-specified composition schemes for same-precedence infix operator chain like $(a \leqslant b < c) \rightsquigarrow ((a \leqslant b)\ \text{‹and›}\ (b < c))$. This approach easily manages even the notoriously complex METAFONT-like notation for curves:

```
a -- b                  — Defines a line
a -- b -- c --@         — Defines a triangle
a ~~ b ~~ c ~~@         — Defines a circle through a, b and c
```

With this extensions and inherent support for multiline, end-of-line and inline literals, the Agdaesque approach to parsing (see "Parsing Mixfix Operators" by Nils Anders Danielsson and Ulf Norell) seems to cover the needs of most sophisticated linear mathematical notations and all computer-processed languages known to the autors, even the most obscure ones[2].

Note that such flexible approach to syntax means that an expression can be often parsed ambigously despite operators having specified precedences. In order to deal with ambiguity, the operators are defined together with their signatures: this is a desirable effect allowing operator overloading. All possible interpretations of the expression undergo a typechecking attempt, and only those that do survive. If there are still multiple nonequivalent interpretations, parsing fails providing the user with information about ambiguities which the user has to eliminate. It is still possible to have a dynamically typed language though: a dynamically typed language is essentially just a single-typed language, one where all terms have the same type.

In order to make notations predictable, composable and extendible, we need to fix some standard and naming conventions.

- Context of a scope is the list of defined operators together with their signatures and their precedence graph.
- Every context contains the operator `(_)` (standalone parentheses) acting like identity. Note that the opening parenthesis is not a suffix token, so `foo(_)` and `foo (_)` do not have to be equivalent. Many existing languages and common mathematical notation already make this distinction: standalone parentheses are used for grouping while "postfix parentheses" are used for evaluation-at-a-point.
- The most simple kind of operators are nullary closed operators, they are also called simple identifiers. A space separated sequence of identifiers is called application chain and gathers to the left. If f, x and y are simple identifiers, `f x y ≡ (f x) y`.

  Note that operators may bind tighter than application, in particular the (right-gathering) logarithm **`log log`** `n ≡` **`log`** `(`**`log`**` n)` can be defined.
- Naming conventions should provide visual distinctions between such cases and simple identifiers. We propose using `a ‹max› b` for word-like binary infix operators, `n --is-even` for word-like unary postfix operators and `#if b #then a #else b` for all other word-like operators. (Should be rendered boldface without the hash sigil).
- The underscore is a special "placeholder" character, which can be used to turn every operator into identifier, allowing to write any expression as application chain (S-expression): **`if`**`(n --is-even?) a + b` **`else`** `0 ≡ (`**`if`**`(_)_`**`else`**`_ (_--is-even n) (_+_ a b) 0)`. Consider our

[2]Say, derivative `(_)'`, sum $\sum_{[\underline{n} < \underline{m} < 5]}$`expr` and **`select`** `(:name, :age)` **`from`** `tbl` **`where`**...

endeavour to be a pursuit for M-expressions that were never really defined in LISP.

- The comma operator `_,_` should be work as follows: Whenever the type of the resulting expression can be infered and is an inductive type with single constructor of n arguments, a_1, a_1, …, a_n should be interpreted as application of the constructor without naming it. If the list of comma-separated values is longer than n, right associativity is applied. In particular, if the resulting type is $A \times B \times C \times D$, the expression `a, b, c, d` is parsed as `( MkPair (MkPair (MkPair a b) c) d)`. Empty parentheses `()` should work analoguously for types with a single nullary constructor.

- We propose the following solution to the currying syntax problem: whenever simple identifiers denoting non-nullary constructors and functions are defined, the annotation **`def`** should additionally define a closed operator for the maximally tupled form should be introduced: the definition

  ```
  def f(a : A, b : B, c : C): ...
  ```

  should define both `f : A → B →⋯ C → D` and `f(_) : A × B ×⋯ C → R` (note the absence of whitespace between name and opening parenthesis). In particular the pair constructor can be written either as `(MkPair` x y`)` or as `MkPair(`x`,` y`)`. The price to pay is that $f(x)$ and f (x) are in general (i.e. whenever f is a function of at least two arguments) not the same.

- Likewise, whenever a word-like binary infix operator is defined, a parallel curried function should be defined. Consider the usage of `n ‹mod› n` for the equality-modulo-function relaton `5 =(mod 4)= 1`.

- Operator signature can provide additional constants (not only identifiers but also arbitrary operators) for the context of its operands facilitating DSLs (domain-specific languages) for building complex hierarchical object structures in a semi-declarative way, e.g.

  ```
  prefix vertical-layout(top-level-ui-components ⊢ code : Bool): ...
  ...
  display vertical-layout
    val name: edit-text
    button("Say Hello")
      on-click
        say \ Hello, [name.text]!
      .
    .
  ```

  Context extensions can be configured to change after one of them is used (providing use-utmost-once constants and constants which can be used only in some strict order), and to be delimited (see `//kotlinlang.org/docs/reference/type-safe-builders.html`).

- Generalized operators treat one or more of their arguments as expression literals, which are not executed immediately, but staged for later execution. The simplest examples are the asynchronious execition operator and closure operator:

  ```
  def async (code : Expr): ...
  def closure{X Y} ((‹_› : X) ⊢ code : Expr[Y])

  val example1: async (1 + 1)

  val example2: closure{Int Int} (1 + 2 · _)
  ```

  The closure operator extends context of its operand by a new symbol, the underscore, it has no definition yet, just a signature. To evaluate the staged expression one would need to perform an operation on the staged expression, namely the substitution. Generalized operations can be used for metaprogramming (aka syntactic sugar): by introducing some operators and metaoperators into their scope, they can extend the ambient language with control structures, e.g. loops or async/await-notation for any suitable monad (depending on monad it can

be try/catch, stage/evaluate, reify/interpret, box/unbox and lots more). In contrast to usual operators, generalized operators are also allowed limit the context of their arguments and to shadow the meaning of application (the only requirement being that composition still forms a category). It is a special case of accepting an expression of a syntactically compatible, yet a different language: the expressions may be given a completely fresh scope, a completely different typing discipline (within possibilities of ambient typechecker) and implementation of application/composition. This allows to embed really special DSL such as language for FGPA programming (cf. VHDL), specialized quantum computations, production line definition language, a lightweight language for euclidean geometry, etc. Note that in this case, expression literals must support interpolation, so that value from the outside world can be injected and/or used to generate code.

Generalized operators might also require structured block-literals as their arguments. Structured blocks are possibly namespace-generating multiline collections of definition-like or expression-like statements. In this case additionally to everything already mentioned, the operators have some additional control over the block structure. In particular, they might augment blocks by some additional definitions (that's what custom annotations often do), or use the block structure to extract execution sequence information for the DSL they provide (that's how all imperative languades can be embedded into purely functional ones, which is known as do-notation). Generalized operators may export entities from namespace-generating blocks they process, into the outer context.

- All forms of literals should support interpolation via square brackets containing a value and optionally a formatter (on the right side, separate by a pipe). In expressions, interpolation has the syntax `[label@ expression]`, where label is the label of the outer scope one should evaluate the expression in. (cf. Labels in Kotlin), for unnamed scopes the labels `OUTER@` , `OUTER1@`, `OUTER2@` and so on, can be used. Any `ctx.Expr-generating` functions for the suitable context (or DSL definition) `ctx`.

  Some literals can be also made into patterns. To distinguish between interpolation and named holes, a fresh-local-variable-declaration sigil `\` has to be used. In patterns, the formatter goes on the right analogously to type annotations.

```
Template (interpolation):                  Pattern (named holes):

Console.print                        "[a | <expr>] + [b | <expr>]" ↦
  Hello [name]!                        eval(a) + eval(b)
  Today is [yyyy-MM-dd | date].      "[var | <letter>⁺]" ↦
  CPU temperature is [1.1f | t].       get(var)
```

  Freshness sigil should be rendered as underlining by typesetting software: $\backslash a \equiv \underline{a}$.

  A special type of patterns are the tupple patterns (AKA argument lists). They can be untyped, typed or mixed. Untyped ones are written as whitespace separated placeholders surrounded by parentheses: `(n m k)`; typed ones as comma separated lists of interfix colon separated variable-type pairs: `(n : ℕ, m : ℕ, k : ℤ)`; mixed ones group consequent variables of the same type and consequent untyped variables into whitespace-separated blocks, which are themselves comma separated: `(n m : ℕ, k)`. Types of the untyped variables are assumed to be derivable from the context or known by convention (in a **`Variables`**-block one may proclaim identifiers matching a certain pattern to have a certain type till the end of the scope). Unknown types themselves may also be matched. Fresh variables become a part of the context after a comma and may be used in typing annotations of the following variables: `(n : ℕ, el : T, list : SizedList{T, n)`.

- Binders or dependent operators are operators that accept a pattern and an expression, dependent on fresh local variables from the pattern. Here are the widespread examples:
  `(x : X) ↦ expr(x)` — A lambda term
  `(x : X) → Y(x)` — A dependent function type

$(\underline{x}$: X) × $Y(x)$ — A dependent pair type
$\forall(\underline{x}$: X) $P(x)$ — Universal quantifier
$\exists(\underline{x}$: X) $P(x)$ — Existential quantifier

The signatures of the operators look as follows:

```
(p : Pattern)↦(p.context ⊢ e : T) : p.InType → T
(p : ArgList)→(p.context ⊢ e : T) : *
∀(p : ArgList) (p.context ⊢ Prop) : Prop
```

Note, that binders automatically respect alpha-conversions, i. e. simultaneous renamings of variables in the pattern where they were introduced and in all scopes they are used. In particular, names of local variables cannot be exported or otherwise nominally used.

- For special mathematical notations arglist can be contain a single variable of known type, in which case it is known as `ArgName{T`. It enables the differential $\partial_{\underline{x}}$ `expr` and the integral notation $\int_{[R]}$ `expr` $\boldsymbol{d}$`(x)`.[3] A form of Summation Notation $\sum_{[\underline{n} < \underline{m} < 5]}$ `expr` requires an aditional pattern type: constraint-based implicit parameter list. It's a boolean expression containing one or more fresh variables, types of which are known in advance.
- The dot operator (a non-postfix token, i. e. either `Math.sin` or `Math .sin`) is a scoped namespace export operator. It means expression `Math.(expr)` is executed in a context extended by namespace `Math`. If expr is a simply named operator, parentheses can be omited. For infix operators there is a special notation `a ‹Math.min› b` ≡ `Math.(a ‹min› b)` which can be used for generic infix operators in the form `‹Math._○_›`. In the case, the namespace was already imported in the local scope, `Math.` can be used as prefix with textual prefix and closed operators for disambiguation (without exporting anything into the scopes of their arguments). TODO: Records, anon projections, dynamic typing, type providers, inline records.
- Standalone square brackets are used to supply optional arguments to operators. There are four different reasons why an argument might be optional: - It can be uniquelly inferred from the context by signature unification (type parameters are a major example). - It has a default value (either explicit or to be synthesized by a specified tactic). - It has to be inferred by type class resolution and/or defaults resolution. - It's instantiation can be postponed (instead of l : ListT one can use l : List, and l.T afterwards).

  Optional arguments can have either eager or weak insertment strategy.
- Tokens starting with a digit are numeric literals. They can be grouped as `1'234'567` or `16_777_216`, hex `0xFFFF`, octal `0o777`, binary `0b11`, arbitrary base `0<3>201`, rational `1.5`, with repeating part `1.6<81>`, with exponent `0.3e10` (with decimal point and exponent), or combined `0b1.1<10>e11`.

[3] Unicode subscripts parentheses can enclose any argument hole of an operator. They should not be visibly rendered by typesetting software, but delimit parts to be rendered as subscripts. Like the normal-size opening parentesis, subscript one is not a suffix operator.

7 Application: Alternate syntax for Arend Language

We propose an alternate syntax for Arend language[4] based on our design. The proposed syntax is indeed just restatement of existing syntax with our notational approaches. To give an example, Arend' native

```
\module Mod \where {
  \func f1 => f2
  \func f2 => 0
  \func f3 => f4
}
```

is now just

```
Module Mod:
  def f1: f2

  def f2: 0

  def f3: f4
```

or

```
Module Mod

def f1: f2

def f2: 0

def f3: f4
```

without indentation if the first line of the above listing happens to be the first line of the file and thus the whole file the content of the given module. The **where** blocks for local definitions, `open` and `import` remain untouched modulo using hash instead of backslash for keywords. The syntax for parameters also remains untouched modulo freshness sigils for fresh variables and autocurrying. There is no need for the explicit `\lam` keyword because of the notation $\backslash x \mapsto f(x)$. To illustrate, let's just repeat a section of the Arend documentation with our alternate syntax:

Non-recursive function can be defined simply by specifying an expression for the result of the function:

```
\func f(x ... z): T
  expr
```

where `expr` is an expression checkable to be of the type `T`. In such definitions the result type : T can often be omitted as the typechecker can usually infer it from `expr`.

For example, to define the identity function on type T, write the following code:

```
def id(x : T): x
```

A function with three parameters that returns the second one can be defined as follows:

```
def second(x : A, y : B, z : C): y
```

You can explicitly specify the result types of these functions. The definitions above are equivalent to the following definitions:

```
def id(x : T): T
  x

def second(x : A, y : B, z : C): B
  y
```

[4] https://arend-lang.github.io/

Note we can write down the polymorphic identity function (not for a fixed type `T` but for any type `T`) in these three equivalent forms of increasing verbosity:

```
def id(x : T): x

def id{T}(x : T): x

def id{T : Type}(x : T): x
```

Pattern-matching based definitions look like this:

```
def fact(n : Nat):
  Zero ↦ 1
  Succ(n) ↦ Succ(n) · fact(n)
```

Instead of `\elim` keyword and `\case` construction we propose **`match`**`(expr)` with optional result type annotation **`match`**`(expr){ResultType}`.

We propose to use the keyword **`Family`** instead of `\data` for inductive type families:

```
Family Nat
  Zero
  Succ(n : Nat)
```

Note that Arend does not support indexed type families, only parametrized ones, so you can only write down the type like `Fin(n)` by pattern matching on `n)`:

```
Family Fin(n : Nat) match(n)
  Zero ↦
    -- empty
  Succ(n : Nat) ↦
    FZero
    FSucc(p : Fin{n})
```

Note however, that due to availability of induction-induction, induction-recursion and interval type, large fraction of (maybe even all) indexed inductive type families can be represented, so they might be added as a form of syntactic sugar later.

The novelity of Arend are conditions for constructors which allow definitions of higher inductive types. Conditions are introduced by **`with`** keyword with optional argument listing the parameters of the constructor which are to be pattern matched:

```
Family Int
  Pos(_ : Nat)
  Neg(_ : Nat) with
    Zero ↦ Pos(Zero)
  .

Family Int'
  Pos(_ : Nat)
  Neg(_ : Nat)
  Join(_ : 𝕀) with
    Lt ↦ Neg(Zero)
    Rt ↦ Pos(Zero)
  .

Family Susp (T : Type)
  North
  South
  Merid(_ : T) (i : 𝕀) with(i)
    Lt ↦ North
    Rt ↦ South
  .
```

We propose to use the keyword **`Structure`** for both records and classes, the difference being that classes have a field marked by annotaton **`@on`**, the so called classifying field. Some of the fields

may be annotated by **@property** meaning they are never actually evaluated. Both structures and families support the **@Truncated**.

For instance definitions we propose to use the name of the class as the keyword:

```
CSemiring Nat:
  ...

CResiduatedLattice{‹min›, ‹max›, +} Nat:
  ...
```

Syntax for coersions, level enforcement and syntax for expressions remain the same mutatis mutandis.

We propose new definitional keyword **Property** for classes with no evaluatable fields (except for classifying field) and exactly one property field which has to have the same name as the property being defined, e.g.

```
Property Primality{@on n : Nat}:
  ∄(m : Nat, m > 1) n ‹mod› m = 0
```

Properties automatically come with definitional coersions to the base type and, conditionally, from the base type (given, the proof of the property can be derived for the variable in question from the context).

We propose reusing the **with** keyword for type refinement:

```
def example(n : Nat with Primality, p q : Nat with (_> 1))
```