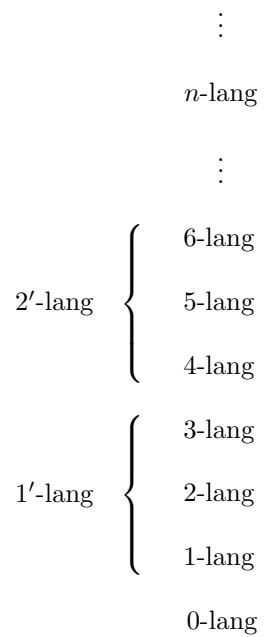


Nik Symbolic

Daniel Nikpayuk

December 2nd, 2019

Infinity Language (Scalable Library)



\vdots
language - n

\vdots
language - 3

language - 2

language - 1

language - 0

Infinity Language (Scalable Library)

$$\infty\text{-lang} := \left\{ \begin{array}{c} \vdots \\ n\text{-lang} \\ \vdots \\ 3\text{-lang} \\ 2\text{-lang} \\ 1\text{-lang} \\ 0\text{-lang} \end{array} \right\}_{n \geq 0}$$

Infinity Language (Topology?)

$$\infty\text{-lang} := \left\{ \begin{array}{l} \vdots \\ n\text{-lang} := \{ \text{“tools to talk about } (n-1)\text{-lang”} \} \times \{ \text{“things (subsets) said about } (n-1)\text{-lang”} \} \\ \vdots \\ 3\text{-lang} := \{ \text{“tools to talk about 2-lang”} \} \times \{ \text{“things (subsets) said about 2-lang”} \} \\ 2\text{-lang} := \{ \text{“tools to talk about 1-lang”} \} \times \{ \text{“things (subsets) said about 1-lang”} \} \\ 1\text{-lang} := \{ \text{“tools to talk about 0-lang”} \} \times \{ \text{“things (subsets) said about 0-lang”} \} \\ 0\text{-lang} \end{array} \right\}_{n \geq 0}$$

Nik Higher Order Module Design

languages	
3-lang	{ assemblc-lens-modules , symbolic-branch-modules }
2-lang	{ assemblc-branch-modules , symbolic-space-modules }
1-lang	{ assemblc-space-modules , Meta C++ }
0-lang	{ C++ }

Language Indirection Design

languages	modules	assemblics		; symbolics
0-lang		assemblic grammars (C++)		
1-lang	0-module	assemblic spaces		; symbolic grammars
2-lang	1-module	assemblic branches		; symbolic spaces
3-lang	2-module	assemblic lenses		; symbolic branches

For the following definitions, assume T_0 is a type system. A judgement $(\lambda : \Lambda)$ is a binding of an *instance* (λ) and a *type* (Λ) .

Definition (compositional reflexivity): Let A be a type in T_0 , we define the **reflex** with respect to A as

$$\text{reflex}_A : A \rightarrow A$$

$$\text{reflex}_A(a) := a$$

where $a : A$.

Here the reflex operator is just the standard *identity* function for its respective type. The terminology “reflex” coincides with Homotopy Type Theory which denotes it as “refl”.

Definition (compositional transitivity): Let A, B be types in T_0 , we define the **transit** with respect to A, B as

$$\text{transit}_{A,B} : (A \rightarrow B) \times A \rightarrow B$$

$$\text{transit}_{A,B}(f, a) := f(a)$$

where $f : A \rightarrow B$ and $a : A$.

Here the transit operator is just the standard *apply* operator known in functional programming. This construct is key in building a programming language interpreter. If you get down to the heart of it an interpreter is a combination of “eval” and “apply” (SICP Ch4). In applicative order (eager) evaluation, if you have the expression

$$(+ \ 1 \ 2 \ (* \ 3 \ 7))$$

you first identify the operator $(+)$, then its operands $(1 \ 2 \ (* \ 3 \ 7))$, but before you pass these arguments to this function you evaluate them

$$(\text{eval } 1) \ (\text{eval } 2) \ (\text{eval } (* \ 3 \ 7))$$

The transit operator is key because there is another way to view this evaluation: Assume all objects in the interpreter’s computation space are applicable, which is to say they have an evaluation code. Objects such as the numeral 1 would evaluate to themselves, and so their evaluation code is **reflex** _{\mathbb{N}} .

This by the way is the starting point for **Dual Theory**, the idea being you take an existing type system T_0 and assign each “instance” a second “type”: If it’s a function object (f, a) , you assign it its respective **transit**, otherwise it defaults as a **reflex**.

What's the value in this way of thinking?

Definition (compositional form): Let A, B, C be types in T_0 , with $f : B \rightarrow C$. We define the **compositional form** of f relative to A as

$$\begin{aligned} f_A & : (A \rightarrow B) \times A \quad \rightarrow \quad C \\ & := f \circ \text{transit}_{A,B} \end{aligned}$$

What's the meaning of this? In math, when we write the composition $g(f(a))$ there is a lot we take for granted. In programming, we actually have to be a bit more rigorous in how we interpret this computation. In eager evaluation, we would evaluate $f(a) = b$ first then pass the result and evaluate $g(b)$. In eager programming paradigms this also works, but in lazy paradigms we've effectively memoized or delayed $f(a)$, in which case it needs its own type. This is where the compositional form comes in. Instead of evaluating $f(a)$ immediately, we've effectively bound the function name with the argument (f, a) without actually evaluating the expression. This is exactly our function object instance.

In terms of practical application: When defining grammar for functions, if you plan on using lazy expressions this compositional form is ideal for grammatical forms which are readily composable. Keep in mind, *source code* is often viewed outside the paradigms of programming, but source code itself is a lazy or delayed expression.

Finally, in C++ template programming, **structs** are used as template functions and so are lazy by their very implementation. When defining new grammar for template programming, compositional forms offer a starting point for its design. As I'm building my own library with its own metaprogramming branch, this is incredibly important to me.