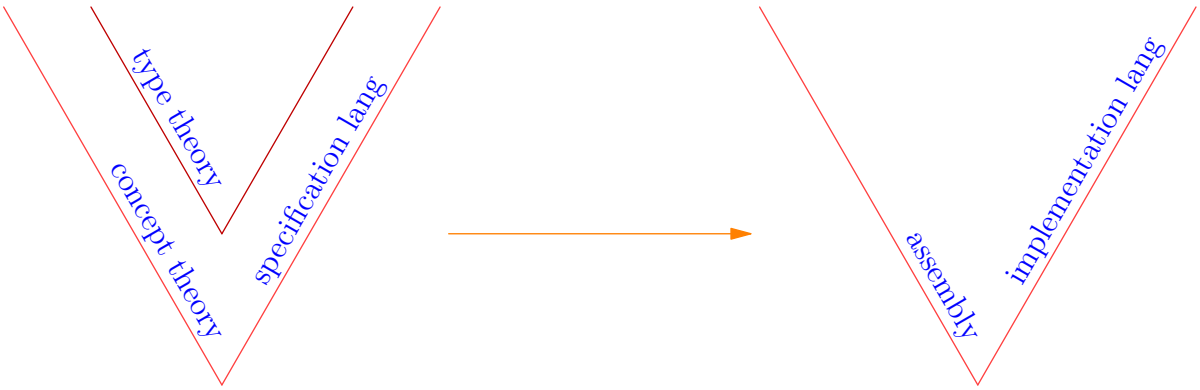
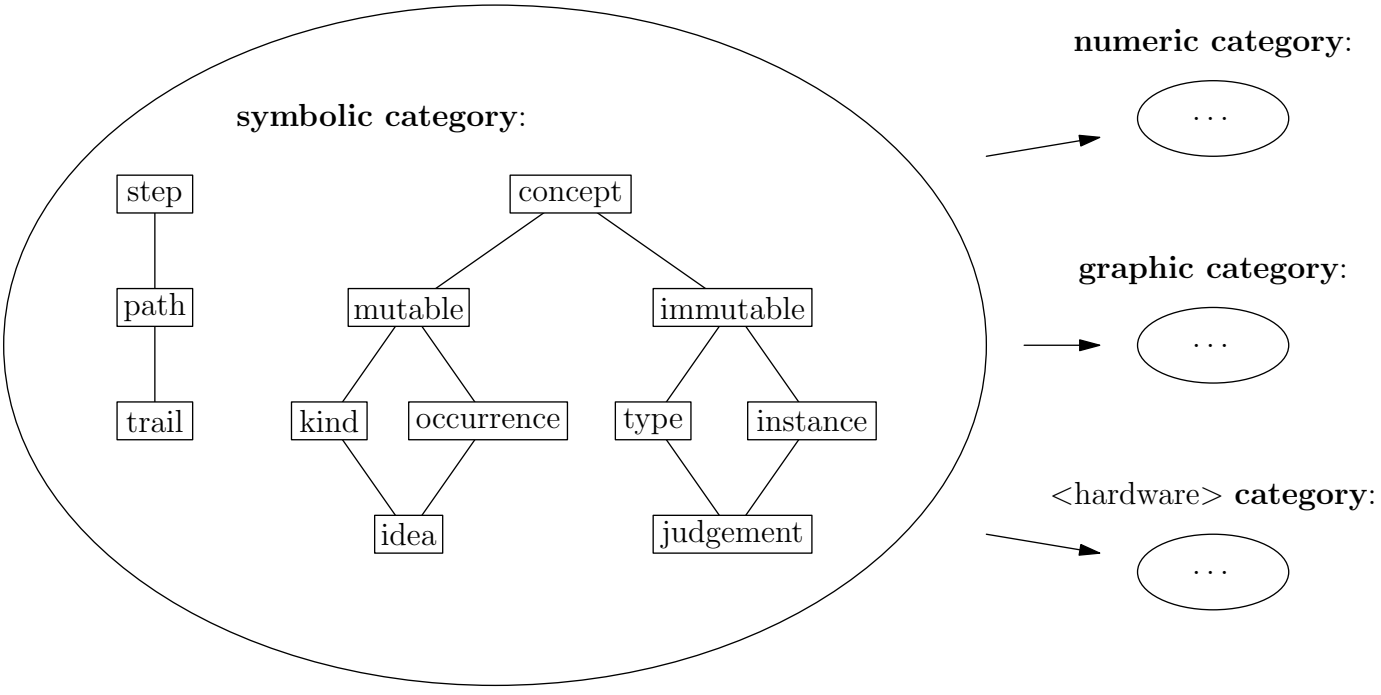
kin

Daniel Nikpayuk

August 20, 2019





When you get down to basics, programming languages require **grammar** for *data structures* and *algorithms*. Algorithms tend to be defined as functions, which are used to manipulate the data in the structures. Beyond this, all you really need is a system to diagnose design correctness, and everything else is just details.

Today's Programming languages generally follow the *variable* + *type* + *instance* paradigm:

$$\text{int } x \quad = \quad 5;$$

which
binds as

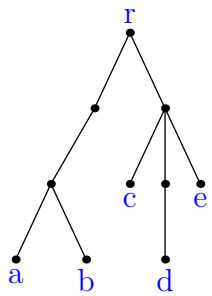
$$\begin{aligned} \langle \text{variable} \rangle &\quad \to \quad (\langle \text{type} \rangle, \langle \text{instance} \rangle) \\ x &\quad \to \quad (\text{int}, 5) \end{aligned}$$

The kin programming language on the other hand breaks from this traditional *genre* by instead interpreting variables as follows:

$$\langle \text{variable} \rangle \ \to \ \langle \text{concept} \rangle \ + \ \langle \text{sifter} \rangle \ + \ \langle \text{target} \rangle$$

Level 0 (grammar for near-linear constructs)

Intuitively, **concepts** are a deconstruction of **trees**.

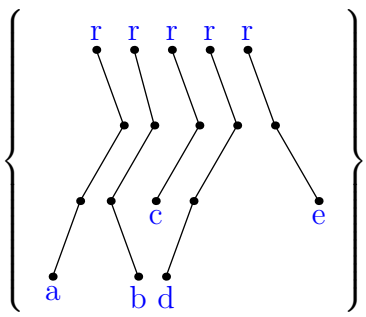


Why **Trees**?

1. They're sufficiently and minimally connected. They offer navigational access to all data points in the structure while maintaining minimal navigational overhead.
2. They are recursive and thus self-similar. Structures are most expressive when all combinatorial substructures are easy to represent.

Trees are ideal structures. As I'm designing a language with a highly expressive math-like (entropic) grammar the notion of "structure" has high utility when it's highly navigable, which is achieved if one can express all possible substructures with ease. Trees are ideal in this way because there's a natural bijection between initial subpaths (from the root) and subtrees. Any computationally effective data structure used in a programming language often has a similar patterning, thus trees are the starting point for this design.

Concepts are derived as follows: Trees can be represented as a collection of paths, and a collection of (refactorable) paths can represent a tree, but *not all* collections of paths can be converted into trees. This is the basic idea of a concept, it's a weakening of the idea of a tree.



Why **Concepts**?

1. Concepts can represent trees and are the starting point for data structures, but they can also represent primitive functions. This extends to function composition (tree composites), which can can then be used to express primitive recursive functions.
2. Concepts maintain the same ideal sub-structuring and navigability patterns seen within trees, but are *not* restricted to the same expectation that they should form semantically complete objects.

Concepts are ideal prototypes. Let's go over these points in more detail.

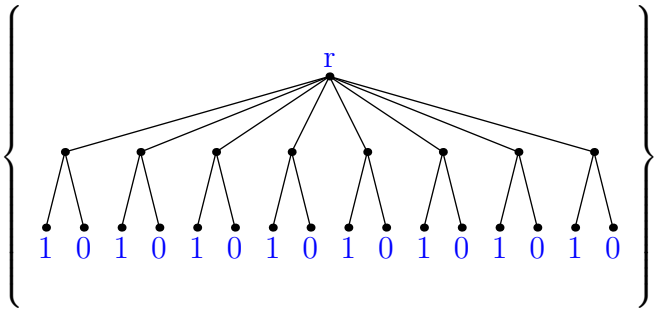
To start, since concepts within the kin language privilege expressivity of substructuring, we define **sifters** to be the use of predicate logic analogous to set theoretic *subsetting*:

$$\{ \ x \in \text{Concept} \quad | \quad \text{Sifter}(x) \ \}$$

To be fair, I could use the word "filter" instead here, but I have chosen against it as it is already overused in the literature. Besides, "sifter" more accurately portrays the idea that our focus is on what's being kept rather than what's being cut.

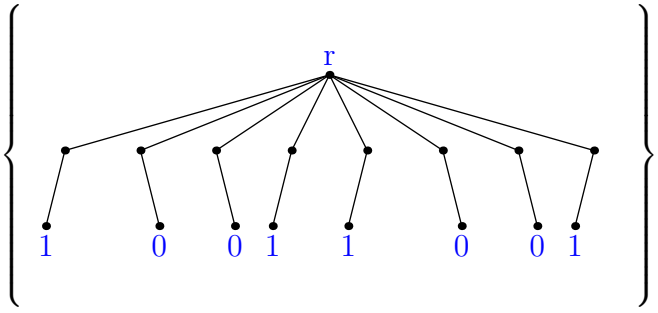
Next, although research into *Concept Theory* is ongoing, it is intended to be compatible with *Type Theory*. As example, let's construct the classic **byte type**. We start with a concept as a byte prototype:

ProtoByte := **/[0-7]/[0-1]**



we can then use this protobyte to define byte instances:

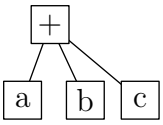
Byte Instance := **ProtoByte/[0-7]/** → **0+1**



where "0+1" is the "or" operator (disjoint union, coproduct), which is to say it is a grammar indicating *alternatives*. In this way, since we can use concepts to define type instances, we can use them to define types themselves. So long as we restrict ourselves to this level, we can make use of all the benefits of type theory as well.

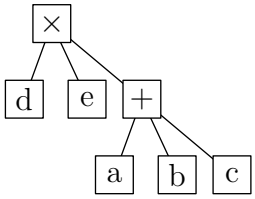
The general extension of this is to define **higher concepts**. From a concept theory perspective, a type is a collection of concepts, but it would be ideal for such a collection to be a concept as well. The intention of defining such constructs would be that they would behave the way initial concepts do, meaning they are highly expressive (and thus navigable) when it comes to substructures. This would allow us to maintain the *sifter* style grammar for sub-structuring and navigation throughout the language.

Next are **functions**. Functions and function composition are intuitively represented as trees as follows:



$a = b + c$

$+/0 := a$
$+/1 := b$
$+/2 := c$



$d = e(b + c)$

$\times/0 := d$
$\times/1 := e$
$\times/2 := (a = b + c)$
$\times/2/0 := a$
$\times/2/1 := b$
$\times/2/2 := c$

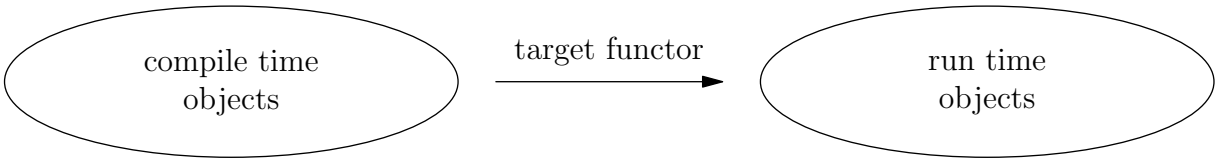
The advantage in representing functions as concepts is it allows the idea of a semantically incomplete function, an idea we will return to in Level 2.

Level 1 (grammar for constructs with cycles)

If we have tree navigable structures and functions, we can extend structures to be cyclical. In comparison, languages such as C/C++ use arrays and pointers (dereference to obtain a numerical value which can be reinterpreted/cast as an array address). In the kin language, the equivalent is to define cyclic data structures as combinations of tree data structures plus primitive functions which map tree nodes to other tree nodes. Functions used in such definitions are called *navigational functions*.

Level 2 (grammar for functors)

The kin language modularizes compile time objects and run time objects into their own categories:



which is where **targets** come in. As stated at the beginning, a variable in the kin language can be extended to include target information which allows it to map to a run time object. For example, the byte type from earlier could be mapped to an 8-bit register on the target architecture.

The value in modularizing compile time from run time is that it encourages the use of compile time concepts as incomplete run time objects (as mentioned previously). This might upon first glance appear to be a bug, but is in fact a feature: Concepts which do not map to run time objects may still express useful patterns which can be combined and recombined in the production of source code, thus increasing the overall expressivity of the language further. Beyond this, if we view the independent compile time category as *symbolic*, it is then the starting place for *semantic verification*, not to mention *code generation* (given an appropriate type theoretic engine).

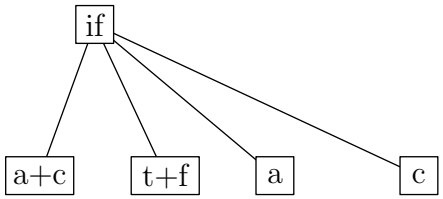
Next, this symbolic approach combined with sifters creates a natural language for describing *weak specifications*. Weak specifications are known to useful on their own (any proof experienced mathematician knows this), and in any case can always be extended and resolved if need be. Resolving the identity of a spec is equivalent to translating it into a *run time object*.

Lastly, this approach creates an expressive alternative to *polymorphism*: In practice the kin language is intended to offer computational efficiency comparable to C++ but with a clean type system, and which furthermore prevents the sort of code bloat you might see in template metaprogramming with redundant type/procedure instantiation.

experimental

This section is auxiliary to the main essay, as kin is in its developmental stage, I am still testing out ideas. There's no narrative flow to this section, just random ideas I'm trying that make sense to me.

$$\begin{array}{lllllllll} \langle \text{variable} \rangle & = & \langle \text{graph} \rangle & + & \langle \text{predicate} \rangle & + & \langle \text{functor} \rangle \\ \langle \text{concept} \rangle & = & \langle \text{prototype} \rangle & + & \langle \text{sifter} \rangle & + & \langle \text{target} \rangle \end{array}$$



sifter *cond*	::	/predicate/true	$\Longrightarrow$	/0/ = /antecedent/
		/predicate/false	$\Longrightarrow$	/0/ = /consequent/
concept	::	if		
if.prototype	::	/0		`% unaliased`
		/predicate\false		`% /1\0`
		/predicate\true		`% /1\1`
		/antecedent		`% /2`
		/consequent		`% /3`
if.sifter	::	*cond*		
if.target	::	identity		

Syntactic sugar for a function call:

$$\langle \text{if } \mathit{pred}\ \mathit{ante}\ \mathit{conse} \rangle$$

type byte
prototype :: $/[0-3]/0, 1$