Concept Theory

Daniel Nikpayuk

April 15, 2018

A **concept:**

is a **model:**

$$\left\{\begin{array}{lcl} p_0 & := & /s_{0,0}/\dots/s_{0,j} \\ p_1 & := & /s_{1,0}/\dots/s_{1,k} \\ & \vdots & \\ p_n & := & /s_{n,0}/\dots/s_{n,m} \end{array}\right\}$$

with a **filter:**

$$\left\{\begin{array}{c} P_0(p_0,\dots,p_n) \\ P_1(p_0,\dots,p_n) \\ \vdots \\ P_\ell(p_0,\dots,p_n) \end{array}\right\}$$

A **model** is a collection of *paths* (each made up of *steps*), while a **filter** is a collection of *properties* on those paths—paths can be bound to values, or can be said to relate to each other in some way. The model offers the syntax, while the filter provides the semantics.

Concept theory as a foundation for math/computing is oriented to be a language with more expressive grammar than existing set theories. For the sake of a casual explanation, you can think of concept theory as an alternative set theory with alternative constructions. This is to say: We can assume the axioms of *finite set theory*, but instead of building constructs such as **pairs, cartesian products, relations, functions**, etc. directly, we build **concepts** and use concepts to construct everything else.

My claim is that concepts are more expressive—especially in regards to programming languages—than existing set theories:

$$\begin{array}{lcl} \textbf{concept:} & := & \{\ \textbf{model: } A \ \mid\ \textbf{filter: } B\ \} \\ \textbf{subconcept}': & := & \{\ \textbf{model: } A' \ \mid\ \textbf{filter: } B'\ \} \\ \textbf{subconcept}'': & := & \{\ \textbf{model: } A'' \ \mid\ \textbf{filter: } B''\ \} \end{array}$$

where

$$\begin{array}{lcl} A & = & A' \cup A'' \\ B & = & B' \cup B'' \end{array}$$

The reason for my claim is that concepts fundamentally represent expressive grammatical *syntax* rather than *semantics* (in regards to linguistics). This is to say we can decompose a concept into arbitrary (even non-intuitive) components which offers a finer approach to practical constructions, potentially reducing constructive redundancy. Again this is relevant to a computational way of thinking.

There are some philosophical differences between concept theory and set theory:

- **finite representations:** I've already stated that for convenience we can assume finite set theory to define concepts. The reason we don't need anything more potent is because concept theory models language itself, and if you look at all the math notation (language) used to express infinite sets, the notation itself is finite. Concept theory adheres to this philosophy.
- **narrative decompression (bootstrapping):** There are several design subtleties the various set theories don't generally consider. For example, in the narrative construction of the language, one would define a *set theoretic function* as a specialized set theoretic relation. Now, a relation is made up of ordered pairs, but if you give it some thought, to access the elements of a pair, you need in some form or another two functions (projections), but functions aren't yet defined.

  The way in which this subtlety is navigated is never formally expressed in the axioms of set theory itself. It is deferred to the linguistic / philosophy side of things, where one starts thinking about the nature of using an internal language to talk about other external languages. Concept theory formally solves this style of problem within the language itself by means of bootstrapping which we'll get to shortly.

A practical comparison between set theory and concept theory is the construction of the natural numbers $\mathbb{N}$. In set theory we define the set operator $\mathrm{succ}(x) := x \cup \{x\}$, then we define an inductive set $\mathcal{I}$ as:

1. There exists an $e \in \mathcal{I}$ ($\mathcal{I}$ is non-empty),
2. For all $n \in \mathcal{I}$ we have $\mathrm{succ}(n) \in \mathcal{I}$.

We then axiomatically assume such an *inductive set* exists because we cannot prove it. Finally, we use such an inductive set to define the natural numbers to be a smallest such inductive set (using subsets and intersections).

On the other hand. . .

In concept theory we start with concepts:

$$\textbf{concept:} \quad = \quad \{\ \textbf{model: } A \mid \textbf{filter: } B\ \}$$

A, B are finite sets, in particular the *modelling* set is defined to contain **paths** which themselves are made up of **steps**. The *filtering* set is defined to contain **predicates**. So here's the thing, since concept theory privileges *bootstrapping*, we want to define these and as many other ideas as we can internally to the language.

As another part of the methodology then, it is common practice to define a "type" or "kind" of concept manually for a few single values, then automatically or recursively for the rest. For example the first three steps are defined as follows:

$$\begin{array}{rcllcl} 0 & := & \{\ \textbf{model: } \emptyset & \mid & \textbf{filter: } \emptyset & \} \\ 1 & := & \{\ \textbf{model: } \{0\} & \mid & \textbf{filter: } \emptyset & \} \\ 2 & := & \{\ \textbf{model: } \{0,1\} & \mid & \textbf{filter: } \emptyset & \} \end{array}$$

With $0, 1$ we now have boolean values, and with steps $0, 1, 2$ we can now define boolean monoids: binary logical operators. This in turn would give us the basic logical *implication* operator ($\Rightarrow$), which we can then use to define our first function: the successor function. Finally, we can then define the remaining (first round of) steps which correspond to the natural numbers.

From here, we can define more general paths because we can also locally define *projection functions* and thus *pairs.* I have already worked out a computational narrative in building *applicable objects, copairs, if-then-else operator, lists* (and their recursive operators), *colists* (switch statements). All of this can be done rigorously all the while privileging bootstrapping.

The one clear tradeoff to privileging bootstrapping as a value is when one prefers consistently defined types (such as functions): Once the scalable (universal) definition is given, we would need to show the previously defined manual instances also satisfy the universal definitions.

The intention of this essay is to demonstrate a universal property of mathematical functions that parallels the "subsetting" paradigm from Set Theory:

$$\{\, x \in A \mid P(x)\,\}$$

Here we can view the set A as a *model*, and the predicate $P(x)$ as a *filter.* The idea being presented then is that a function can be similarly decomposed into a model component followed by a filter component.

For example a function such as:

can be defined conceptually as follows:

function **model:** $\qquad\qquad$ function **filter:**

$$\left\{\begin{array}{lcl} p_0 & := & / \\ p_1 & := & /0 \\ p_2 & := & /1 \\ p_3 & := & /2 \\ p_4 & := & /2/0 \\ p_5 & := & /2/1 \\ p_6 & := & /3 \end{array}\right\} \qquad \left\{\begin{array}{lcl} p_0 & = & f \\ p_1 & = & y_f \\ p_2 & = & x_1 \\ p_3 & = & g \\ p_4 & = & x_2 \\ p_5 & = & w \\ p_6 & = & x_3 \end{array}\right\}$$

The philosophical consequence of defining a function this way is that it is a *relational* object.

Let X be a non-empty set of *paths.* A *type* $\mathcal{T}$ is defined as follows:

$$\mathcal{T} \subseteq \mathbb{P}(X)$$

where $\mathbb{P}(\cdot)$ is the *powerset* of X. In particular, for any $\mathcal{I} \in \mathcal{T}$, $\mathcal{I}$ is called an *instance* of the given type, and any subset $\mathcal{S} \subseteq \mathcal{T}$ is a *subtype.*

The advantage of this way of defining types—possessing an internal structure—is their respective paths allow us to define a *filter algebra* by means of a path grammar, which allows us to express subtypes and instances as *concepts.* In practice, many concepts correspond to subtypes, but as it's possible $\mathbb{P}(X)\backslash\mathcal{T} \neq \emptyset$, concepts are a more general idea than a type.

$$\begin{array}{rclcl} \textbf{byte type} & := & (0+1)^8 & := & \{\, \mu^m\sigma^n \mid 1 \le m \le 8,\ 1 \le n \le 2\,\} \\ \textbf{byte instance} & := & \mathcal{I} \subseteq (0+1)^8 & , & \mu^m\sigma^k\,,\ \mu^m\sigma^\ell \in \mathcal{I} \implies k = \ell \end{array}$$

Algorithm for defining (designing) concepts:

1. Given a concrete universal grammar, specify the type as a space of paths. For a byte, we construct $(0+1)^8$ which is the eight term *product* of the two term *disjoint union* of objects $0, 1$.

   Here μ is the product operator with μ^m its mth operand; σ the disjoint union operator with σ^n its nth operand. In particular $\mu^m\sigma^n$ denotes a given path within the space $(0+1)^8$.

2. Given a type, we specify its instances by constructing a predicate *filter* which generates a family of subsets of the space. Each subset of paths within the family is an instance.

Consequences of this approach to **type theory**:

1. Concept theory requires a concrete universal grammar for constructing spaces of paths. Fortunately much research in type theory, category theory, homotopy type theory, and analytic combinatorics has already been done to that end.

2. A concept generalizes the idea of a type as well as an instance. A concept is the dual of a filter—growing the filter shrinks the concept. A concept representing a unique type instance is said to have its identity resolved. Filters are specified by predicate logic, where the predicates are themselves concretely specified by the grammar used in constructing the space of paths.

3. Unresolved concepts are isomorphic to mutable data structures.

4. By using paths as the medium of exchange in our design, we imply every concept *as* data structure already has an implicit natural coordinate system—a universal medium for navigating every subconcept as well as every path resolution within the concept as type.

Narrative:

$$\begin{array}{rclcl}
\textbf{bit type} & := & 0+1 & := & \{\,\sigma^n \quad | \quad 1 \le n \le 2\,\} \\
\textbf{bit instance} & := & \mathcal{B} \subseteq (0+1) & , & \sigma^{n_1}\,,\ \sigma^{n_2} \in \mathcal{B} \quad \Longrightarrow \quad n_1 = n_2 \\
\\
\textbf{word type} & := & (0+1)^N & := & \{\,\mu^m\sigma^n \quad | \quad 1 \le m \le N,\ 1 \le n \le 2\,\} \\
\textbf{word instance} & := & \mathcal{W} \subseteq (0+1)^N & , & \mu^m\sigma^{n_1}\,,\ \mu^m\sigma^{n_2} \in \mathcal{W} \quad \Longrightarrow \quad n_1 = n_2 \\
\\
\textbf{address type} & := & (0+1)^{NM} & := & \{\,\mu^\ell\mu^m\sigma^n \quad | \quad 1 \le \ell \le M,\ 1 \le m \le N,\ 1 \le n \le 2\,\} \\
\textbf{address instance} & := & \mathcal{A} \subseteq (0+1)^{NM} & , & \mu^\ell\mu^m\sigma^{n_1}\,,\ \mu^\ell\mu^m\sigma^{n_2} \in \mathcal{A} \quad \Longrightarrow \quad n_1 = n_2 \\
\\
\textbf{tree type} & := & L(0+1)^{NM} & := & \{\,\sigma^k\mu^\ell\mu^m\sigma^n \quad | \quad 1 \le k \le L, 1 \le \ell \le M,\ 1 \le m \le N,\ 1 \le n \le 2\,\} \\
\textbf{tree instance} & := & \mathcal{T} \subseteq L(0+1)^{NM} & , & \sigma^k\mu^\ell\mu^m\sigma^{n_1}\,,\ \sigma^k\mu^\ell\mu^m\sigma^{n_2} \in \mathcal{T} \quad \Longrightarrow \quad n_1 = n_2
\end{array}$$