Concept Theory

Daniel Nikpayuk

July 11, 2020

concept theory example:

is there a relationship between *concepts* and *universe building* ?

$$\text{pair instance} = \begin{cases} \textbf{path:} & 0 \\ & 1 \\ \textbf{target:} & 0/ \quad \text{as } a \\ & 1/ \quad \text{as } b \\ \textbf{sifter:} & 0 < 1 \end{cases}$$

path	**target**	**sifter**
syntax	semiotics	semantics

interface	**perspective**	**lens**
structure	identity	filter
navigator	proximity	space
	embedding	

$$\text{pair instance } (a, b) \ = \begin{cases} \textbf{path:} & 0 \\ & 1 \\ \textbf{target:} & 0/ \quad \text{as } a \\ & 1/ \quad \text{as } b \\ \textbf{sifter:} & 0 < 1 \end{cases}$$

path

Paths are the syntax of the expression. They are made up of *steps*, and they offer a navigational component to expressions.

target

Targets are the semiotics of the expression. They are the *signs*, which are made up of *signifiers* and *signifieds*.

sifter

Sifters are the semantics of the expression. They are the *propositions* which describe relationships between the semiotics.

Concepts are meant to represent mathematical expressions in a rigorous way while maintaining a user-friendly interface. The intended intuitive appeal is that concepts offer a way to model natural language *ways of thinking* such as:

weak specification, specification refinement, specification resolution, implementation.

By *modularizing* out the synactic from the semiotic from the semantic patterns, we can reuse components and express fully realized concepts in many alternative ways. For example instead of specifying a *pair instance* in the above, we could have left the `target` component unassigned, thus only specifying a pair as a weak concept. In fact, this offers an alternative approach to type theoretic *dependent function types* (polymorphism). Otherwise concept theory is flexible enough to model and include type theories directly.

There is also an algebraic aspect to this interface. For example if we wanted to define a function instead of a pair, we would specify its syntactic structure and its semantic relationships. Function application would occur by refining enough of the semiotics that a specific *evaluator* (or interpreter) could then resolve the remaining information.

A **concept:**

is a **model:**

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

with a **filter:**

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

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

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

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 $\text{succ}(x) := x \cup \{x\}$, 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 $\text{succ}(n) \in \mathcal{I}$.

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...

Near-linear function space:

How does one define a function space? Or rather a *function algebra*?
We start with a baseline function space $(\mathcal{F}, \circ)$:

- Composition is associative: If $f, g, h \in \mathcal{F}$, then

$$(f \circ g) \circ h = f \circ (g \circ h)$$

- Composition has identities: If $f \in \mathcal{F}$, then there exist $\mathrm{id}_\alpha, \mathrm{id}_\beta \in \mathcal{F}$ such that

$$f \circ \mathrm{id}_\alpha = f = \mathrm{id}_\beta \circ f$$

As is, one might think this would be enough, but we can do better.

When observing *type theory* and data structures, we have access not only to the atomic types, but we can build more complex versions—known as algebraic data types—through the *product* and *coproduct* type constructors. Unfortunately, with our baseline function space, we only currently have an analog to the product constructor, which does allow us to build chains of functions:

$$\begin{aligned} {}_{\rightarrow} f_0 f_1 f_2 \dots f_{n-1} f_n &= f_n \circ f_{n-1} \circ \dots \circ f_2 \circ f_1 \circ f_0 \\ &= f_n f_{n-1} \dots f_2 f_1 f_0 {}_{\leftarrow} \end{aligned}$$

but otherwise lacks more general expressivity.

Fortunately, composition is such that we can use two of its core operators:

$$\begin{aligned} \text{precompose}_f(g) &= f^\circ(g) &:= g \circ f \\ \text{postcompose}_f(g) &= f_\circ(g) &:= f \circ g \end{aligned}$$

to extend $\mathcal{F}$ to what's known as its *continuation passing* space $\mathcal{CP}(\mathcal{F})$. Intuitively, a continuation passing function $f(x, c)$ takes the argument x as input, and applies it internally, for which it then takes the return value and passes it directly to the function c:

$$f(x, c) \quad := \quad c(\hat{f}(x))$$

As such, each $\hat{f} \in \mathcal{F}$ lifts to a function $f \in \mathcal{CP}(\mathcal{F})$. The continuation passing space is notable because it has its own composition operator $\star$ called *endoposition*:

$$f(x, c_1(y)) \ \star \ g(y, c_2(z)) \quad := \quad f(\, x, \ \lambda y.g(y)(c_2(z))\,)$$

which behaves like any other composition operator:

- Endoposition is associative: If $f, g, h \in \mathcal{CP}(\mathcal{F})$, then

$$(f \star g) \star h = f \star (g \star h)$$

- Endoposition has identities: If $f \in \mathcal{CP}(\mathcal{F})$, then there exist $\text{id}_\alpha, \text{id}_\beta \in \mathcal{CP}(\mathcal{F})$ such that

$$f \star \text{id}_\alpha = f = \text{id}_\beta \star f$$

In this case, the continuation passing space also has a product constructor analogous to that of the baseline space. So how do we achieve a *coproduct* constructor?
We do this through what are called *stem* constructors. Here I borrow notation from the **C** programming language, with a slight modification:

$$\begin{array}{rcl} \text{stem} & := & (\ f\ ?\ g\mathbf{.}\mid h\ ) \\ \text{costem} & := & (\ f\ ?\ g\mid h\mathbf{.}\ ) \\ \text{distem} & := & (\ f\ ?\ g\mid h\ ) \end{array}$$

The **C** language uses the notation $(\, b \,?\, a : c\,)$ to mean the conditional operator, where if b is true then a is returned, otherwise c is returned. In the above I've replaced the colon ':' with the bar '|' which is better aligned with standard *regular expression* notation in which bar '|' indicates *alternatives*. The difference between the three stem constructors is the use of the period '**.**':

1. Stem says if f evaluates true, then evaluate g and *stop* there, otherwise evaluate h (and continue).

2. Costem is dual to stem, and so says if f is true, then evaluate g (and continue) otherwise evaluate h and stop there.

3. Distem has no period, and so it still branches, but it continues regardless.

What's the difference between *stopping* and *continuing*? The notable thing about these stem operators is that each represents a continuation passing function. For example let $f, g, h \in \mathcal{F}$, then $(\ f\ ?\ g\mid h\ ) \in \mathcal{CP}(\mathcal{F})$. As this is the case, we can now endopose these stem constructors.

Finally, if we add to these constructors the initially discussed *lift* constructor—where we took some function $f \in \mathcal{F}$ and lifted it to $\mathcal{CP}(\mathcal{F})$:

$$\text{lift} \quad := \quad (\, f \,)$$

Then we have the means to build our *near-linear* function space.
I call it *near linear* because it starts out being analogous to Linear Algebra, where we can speak of vectors as *linear combinations*. Here, if we start by restricting ourselves to *lift* and *distem*, we have what can be considered the analog to linear combinations:

$$_{\rightarrow}\, c_0 \star c_1 \star c_2 \ldots \star c_{n-1} \star c_n \quad , \quad c_k \;=\; \text{lift or distem} \quad , \quad 0 \leq k \leq n$$

If we then add in the *stem* and *costem* constructors, and allow for the possibility of recursion:

$$f \quad = \quad _{\rightarrow}\, c_0 c_1 c_2 \ldots c_{n-1} c_n \quad , \quad c_n = f$$

we then have *non-linear* combinations. I use the word "near" rather than "non" because we still have the constraint that *only* the very last function c_n in the chain is allowed to circle back. In computing literature this is called *tail recursion*, and as far as non-linear functions go, it's the nearest we can get to being linear, hence the term *near-linear*.
Now, to make these semi-formal definition rigorous, we would require a few additional considerations such as *compositional coherence*, and that the recursive chains (being representations of functions we've already proven to exist) can be proven to halt. With that said, these near-linear chains are quite effective at representing many of the computable functions that mathematicians, computing scientists, and programmers interact with on a daily basis.

Stem operator distributive laws:

Let's focus on *distem* as our case study:

$$(\, f \,?\, g \mid h \,)$$

Given endoposition ($\star$; continuation passing composition), the left and right distributive laws are as follows:

$$\begin{aligned} (f) \star (\, p \,?\, g \mid h \,) &= (\, p \,?\, f \circ g \mid f \circ h \,) \\ (\, p \,?\, g \mid h \,) \star (f) &= (\, p \,?\, g \circ f \mid h \circ f \,) \end{aligned}$$

Given this, the expansion of two distems is then:

$$(\, p \,?\, f_1 \mid f_2 \,)(\, q \,?\, g_1 \mid g_2 \,) \quad = \quad (\, pq \,?\, f_1 g_1 \mid f_1 g_2 \mid f_2 g_1 \mid f_2 g_2 \,)$$

So how do we interpret the right hand side?

$$(\, pq \,?\, f_1 g_1 \mid f_1 g_2 \mid f_2 g_1 \mid f_2 g_2 \,)$$

First, the alternatives can be thought of as a list:

$$(\, pq \,?\, \underbrace{f_1g_1 \mid f_1g_2 \mid f_2g_1 \mid f_2g_2}_{\text{this is a list}} \,)$$

Now about the conditional term pq ? It's not obvious at first, but it turns out to have a natural interpretation of its own:

$$(\, \underbrace{pq}_{\text{this is a binary number}} \,?\, f_1g_1 \mid f_1g_2 \mid f_2g_1 \mid f_2g_2 \,)$$

So for example set $p = false$, and $q = true$. Assuming $false \to 0$ and $true \to 1$, we have:

$$pq \quad = \quad 01$$

Unfortunately, because of how the conditional operator $(\, ? \mid \,)$ is currently defined, we need to take the componentwise negation of this number:

$$\begin{array}{cccccccc} (\, pq & ? & f_1g_1 & \mid & f_1g_2 & \mid & f_2g_1 & \mid & f_2g_2 \,) \\ \sim & & \underbrace{}_{00} & & \underbrace{}_{01} & & \underbrace{}_{10} & & \underbrace{}_{11} \\ & & \underbrace{}_{11} & & \underbrace{}_{10} & & \underbrace{}_{01} & & \underbrace{}_{00} \end{array}$$

which then is the position (in the list) we seek:

$$pq \quad \mapsto \quad \sim 01 \quad = \quad 10 \quad \mapsto \quad f_2g_1$$

The Anatomy of a Computable Function

What is a mathematical function?

In general, I don't fully know (yet). And I say this as I want to make it a completely clear realization: The nature of functions are that they suffer from complexity. With that said, I'm starting to understand the nature of *computable* functions, enough to try to explain here what they are.
A computable function is a triple:

(**text** , **signature** , **hermeneutic**)

where:

text You can define texts as atomics, or data structures of such atomics. This is a constructive model: Traditional grammar that allows for *higher order functions/continuations* such as composition, endoposition, lift, stem, costem, distem are what allow for these constructions.

signature Signatures are the memory models needed to actually compute specific functions. In math we take them for granted, but in computing contexts resources factor in. To be fair, signatures are more than just a "practical concern"—they actually show up in theoretical math contexts, it's only that mathematicians don't really have clear language to express them.

signature (continued) Related to the idea of a signature is that of a *facade*, which is a designated subsignature respective to a given function. Facades coincide with traditional Eulerian notation for functions:

Facades are especially relevant when discussing ideas of function composition.

hermeneutic The hermeneutic (as object) is the *interpretation* of a text, which is to say: It is the instruction set telling us how to actually evaluate a given function. Interestingly, because we've modularized the text component out of functions, we can speak of the same text having more than one interpretation—this in the sense that we can potentially equip a given text with more than one hermeneutic. Current programming languages don't explicitly allow for this.

As for hemeneutic construction, we start by assuming primitive evaluators such as the apply operator:

$\text{apply}(f, x)$	where f is an atomic function, and x is a value instance of its facade

hermeneutic (continued) So we have higher order constructors, and primitive applicators, and these make up the instruction set. We interleave construction and application in order to evaluate a function. Functions with hermeneutics that can be *orthogonalized* are said to be *well-behaved*.

Orthogonalization in this sense means we can separate out the constructors from the applicators, which is to say we can construct the entire text independently of its applicative evaluation. We take such well-behaved functions for granted given we interact with them all the time, but once we delve into recursive functions in which construction and application can't be separated out, we then need language to make these distinctions.

Here is a conceptual overview of a well behaved function's decomposition:

On the left we start with the facade, which—because it is a substructure of the full signature—we can expand to make use of the full memory model. On the right we have the text, which is a composite of atomic texts. We assume primitive applicators. In the middle is the hermeneutic, which due to it being well behaved, we can speak of strictly as application.

To end, I would like to give an example, which will notably use addition ($+$) and multiplication ($\times$). For aesthetic reasons, I will instead use the greek letter sigma (σ) to mean addition, while using mu (μ) to mean multiplication.
Now, if we let $\circ$ represent composition (standard), we can then subscript this symbol to further represent curried composition, for example:

$$\begin{aligned} \sigma \underset{\ell}{\circ} f &= \sigma\ f\ \text{-} \\ \sigma \underset{r}{\circ} f &= \sigma\ \text{-}\ f \end{aligned}$$

Returning to our previous example function:

$$f(x) = x(x+1)^2$$

We can now decompose it accordingly. First, its text:

$$(\mu \underset{\ell}{\circ} (\mu \underset{r}{\circ} \sigma)) \underset{r}{\circ} \sigma$$

which I admit is a bit to get use to since we're not currently raised with this kind of math—but one does get use to it. Following this we have the signature:

$$(x, y)$$

We shouldn't need more than a pair of memory to successfully evaluate. As for its hermeneutic, and given the memory constraint, we evaluate as follows:

$$(x,1) \quad \mapsto \quad (x, x+1) \quad \mapsto \quad (x, (x+1)^2) \quad \mapsto \quad (x, x(x+1)^2)$$

Admittedly this isn't expressed as a language of construction and application. As of yet, research into such languages is ongoing. That's it for now. I hope it helps. Thanks!

Let f be a function with signature s and facade x. Partition x into subfacades:

$$x = v\,;w$$

Then partial compositions of f are defined as follows:

$$\begin{array}{lllll} f \underset{v}{\circ} g & = & f(x) \underset{v}{\circ} g & = & f(v\,;w) \underset{v}{\circ} g \quad := \quad f(g(u)\,;w) \\ f \underset{w}{\circ} g & = & f(x) \underset{w}{\circ} g & = & f(v\,;w) \underset{w}{\circ} g \quad := \quad f(v\,;g(u)) \end{array}$$

In the simple case where the facade x is a pair, we can now define left and right *curry composition*:

$$\begin{array}{lllll} f \underset{\ell}{\circ} g & = & f(x,y) \underset{\ell}{\circ} g & = & f(g(w),y) \quad = \quad f\ g\ \text{-} \\ f \underset{r}{\circ} g & = & f(x,y) \underset{r}{\circ} g & = & f(x,g(w)) \quad = \quad f\ \text{-}\ g \end{array}$$

concept tag: "potential resolved"

A constructive expression such as

$$f \circ g$$

is in many ways analogous to:

$$\{\ x \in \mathcal{S} \mid P(x)\ \}\ \subseteq\ \mathcal{S}$$

What the phrase *potential resolved* means is that we are modelling a space of functions: $f \circ g$ (the composition operator being the constructor), and we're saying that by convention this expression has the *greatest common potential* when it comes to the space's functions (and/or induction operator).

In analogy to the *subsetting paradigm* our expression $f \circ g$ is equivalent to the modelling space $\mathcal{S}$, where at that point if we want to access/model a specific function within that space we from this point on are only allowed to apply filters. To subset $\mathcal{S}$ we apply filters $P(x)$. To refine and resolve $f \circ g$ we apply filters $Q(f, g)$.

If we were to instead declare $h \circ i(j, k)$ as our potential resolved concept, we'd be working with a different function space.

concept tag: "potential-resolved"

The other interesting philosophical note about declaring a concept such as

$$f \circ g$$

potential-resolved is its relationships to logical quantifications:

$$\forall \ , \ \exists$$

For example if you were to take this weak specification ($f \circ g$) and resolve it to a single function, such would equate with *existential* quantification. Taking this line of reasoning further, if you were to refine to *all* possible resolutions this then equates with *universal* quantification.

And yet...

By declaring the weak spec $f \circ g$ as potential-resolved, it also *potentially* represents those same variations of function resolutions. In this case, the interpretation becomes more along the lines of *any* instead of *all*.

Classical mathematical logic tends to let "any" and "all" be equivalent. This is why I say concept theory is philosophically distinct.

Extending the language of Concept Theory:

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

$$\text{e.g.} \qquad \{ \ \mathbf{v} = a_1\mathbf{e}_1 + \ldots + a_n\mathbf{e}_n \qquad a_k \in \mathcal{F}, \quad \mathbf{e}_k \in \mathcal{V} \mid P(\mathbf{v}) \ \}$$

Here I pay homage to Set Theory and its notation for *subsetting*:

$$\{ \ x \in \mathcal{S} \mid P(x) \ \}$$

but I also deviate from tradition so as to delineate certain connotations: It is common practice to use sets to *model* given spaces, to which a commonly observed pattern becomes the orthogonalization of *constructive* models with *constrictive* ones. In the above situation, the linear combination would be the constructive component to this model, while the predicate subsetting would then become the constrictive component.

Concept theory in contrast is a language that allows us to express ideas of "specification". This is to say, the language focuses on specifications as objects rather than sets. In particular, one can then have ideas such as:

weak specifications, resolved specifications,
specification refinement, specification resolution

This much I have presented on many occasions for anyone following this research, it's not new. What I present now is concept theory's ability to express not just set theoretic ideas, but these modelling/design ideas as well. Notably, what is called a constructive model can be more accurately defined as a

potential-resolved specification or a **model-resolved specification**.

This is a major philosophical difference from the classical set-theoretic interpretation. For example in Set Theory a *linear combination* assumes philosophically that *all* such combinations are computed/admitted/achieved. Concept Theory on the other hand takes more of a constructivist perspective where the linear combination as concept is nothing more than a weak specification. At the same time, by declaring it a potential-resolved spec, we are saying it as an expression (and maybe from an information theoretic lens) holds all the potential it is going to. It is imbued with the intention that from here on out we only refine and resolve to actual vectors within the span of such a linear combination (to use more traditional math terminology). Its potential has been resolved.

Finally, as for this update, the last new term I'd like to introduce is this idea of a

consensus-resolved specification.

I have observed on many occasions both in math and computing that certain definitions are given as unresolved, or rather weak specifications. They are considered sufficient, because although they leave room for greater specificity they also have enough information constraints to push the given plots and narratives forward in terms of the theory they represent. For example in math we only need to observe the definition of any algebra such as a group, ring, field, vector space, to find such a consensus spec. In computing one well known example is the

`map` f list

operator which maps a list to another list by applying a function. This is a weak specification because not only is the list object of the `map` operator unresolved, so is part of its subroutine, or subfunction (so to speak). One could take this reasoning further into more theoretical computing and say the induction operators of Type Theory are then consensus-resolved specifications as well.

This is to say, the point of this variety of specification then is to describe specs that are held as standard by the conventions of the community—representing the shared language of the community.

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:

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

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:**

$$\left\{\begin{array}{rcl} p_0 & := & / \\ p_1 & := & /0 \\ p_2 & := & /1 \\ p_3 & := & /2 \\ p_4 & := & /2/0 \\ p_5 & := & /2/1 \\ p_6 & := & /3 \end{array}\right\}$$

function **filter:**

$$\left\{\begin{array}{rcl} p_0 & = & f \\ p_1 & = & y_f \\ p_2 & = & x_1 \\ p_3 & = & g \\ p_4 & = & x_2 \\ p_5 & = & w \\ p_6 & = & x_3 \end{array}\right\}$$

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.

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

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:

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