An introduction to Representational System Theory: Part 1

Daniel Raggi

October 15, 2021

The purpose of Representational System Theory (RST) is to understand rigorously what *representations* are, what we mean when we talk about their structure, and their relations to one another.

This document is an introduction to RST. Something which the team (especially Gem and I) have been developing for a while. There's a long technical paper associated with it, but we are still in the process of finishing writing it up. Moreover, that paper contains way more technical stuff than needed for this introduction, where I plan to introduce intuitively the concepts of RST.

In this document (Part 1) I will introduce *type systems*, *construction spaces*, *representational systems*, *constructions*, and *decompositions*, amongst other auxiliary concepts.

I assume a bit of knowledge about of logic and graph theory, but nothing too fancy. At the end of this document (section 3) there is a technical appendix, for some basic definitions.

1 Representational Systems

To define representational systems, we will start with *type systems.* Roughly, type systems characterise the conceptual hierarchies of representations. Then we define *construction spaces*, which consist of graphs where the vertices are either *tokens* (concrete representations) or *configurators* (the 'glue' that puts tokens together to form other tokens). Types classify tokens and, similarly, *constructors* classify configurators.

1.1 Types & Tokens

The numeral 1 is a token. In computer science we would generally say that the type of token 1 is something like *nat*, *number*, *real* or something else, depending on the context. This makes sense given the use of type systems in computer science (we call this the *term/type* paradigm). However, for the study of 1 as a representation, we might have higher ambitions regarding its classification into types. For example, in semiotics/linguistics it wouldn't be unusual to refer to 1 as a token of type 1 (we call this the *token/type* paradigm). This is meant to discern that some expression such as $1 + 1$ has *two* tokens of type 1 — that is, it has two *occurrences* of the *same* symbol. What 'same' means in this context is not clear, and RST does not commit to a definition of 'sameness'. Instead, RST allows us to assign types to tokens however we want, with the stipulation that a *subtype* relation (which partially orders the types) is also part of a type system. Thus, we can say that the tokens shown above are of type 1, while also acknowledging that type 1 is a subtype of type *number*, or anything we might want the subtype order to be.

Thus RST unifies the term/type paradigm of computer science and the token/type paradigm of semiotics into one.

Definition 1. A *type system*, T, is a pair, $T = (Ty, \leq)$, where

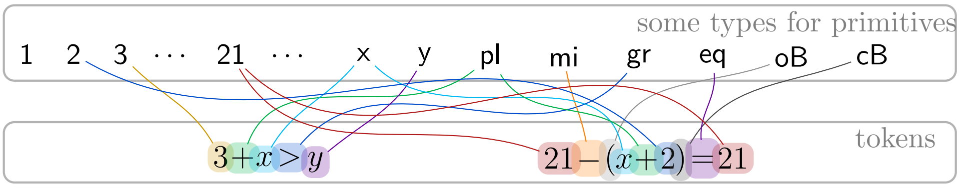
1. Ty is a set whose elements are called *types*, and
2. $\leq$ is a partial order over Ty.

If $\tau_1 \leq \tau_2$ then τ_1 is a *subtype* of τ_2 and, respectively, τ_2 is a *supertype* of τ_1.

The definition above gives us liberty to declare type systems however we like. The assignment of types to tokens is done with a function *type*, which will be formally introduced in section 1.2, after an example of how one could define a type system for formal arithmetic.

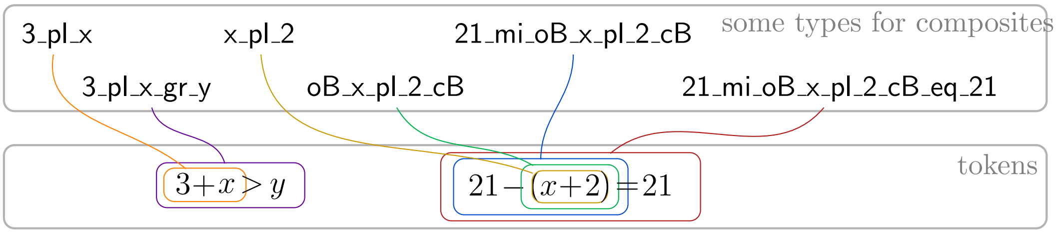
1.1.1 A type system for first order arithmetic (FOA)

We denote our small version of FOA by $\mathcal{S}_A$. We can define one type for each symbol in $\mathcal{S}_A$ as follows:



Note that we even include types for parentheses, as an analysis of representations in first order arithmetic may need to take this into account to consider, say, cognitive factors.

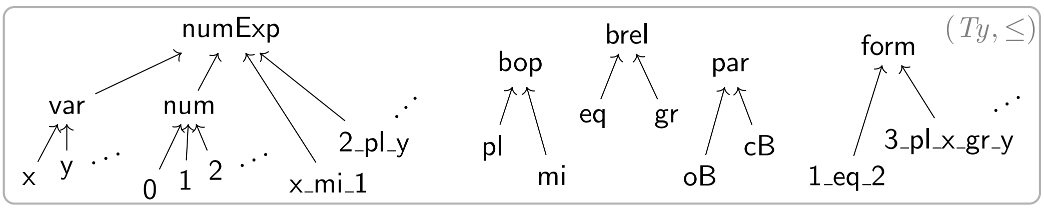
We also need types for *composite* tokens. In this encoding of FOA, all composite tokens are *directly built* from three other tokens. For example, $\forall x\, 3 = x$ is directly built from $\forall$, x and $3 = x$. The name of the type we assign to $\forall x\, 3 = x$ is, essentially, a composition of the names of the types assigned to $\forall$, x and $3 = x$, namely uQ_x_3_eq_x. In general, given a composite token, $t_1t_2t_3$, in $\mathcal{S}_A$ its type is defined to be $type(t_1)\_type(t_2)\_type(t_3)$. The diagram below shows how all *composite* parts of $3 + x > y$ and $21 - (x + 2) = 21$ are assigned types:



The types for primitive and composite tokens are complemented by further useful types:

- num (numeral),
- var (variable),
- numExp (numerical expression),
- bop (binary operator)
- brel (binary relation)[1],
- par (parentheses)
- form (formula)[2].

The diagram below illustrates the partial order, $\leq$, over the set *Ty*, including some of the types assigned to primitives and composites:



[1]The types bop and brel are *composites*, and we note the more standard convention of writing numExp $\times$ numExp $\to$ numExp and numExp $\times$ numExp $\to$ form.

[2]An alternative encoding of $\mathcal{S}_A$ could include additional types such as (with illustrative tokens): sum, tokens: $1 + 2$, $3 + 26$; and diff, tokens: $1 - 2$, $3 - 26$. This gives a finer-grained encoding, distinguishing tokens that exploit different operators such as $1 + 2$ and $1 - 2$. Types that distinguish tokens formed from different binary relations, bgr and beq to distinguish $1 > 2$ and $1 = 2$, may also be of use.

1.2 Construction Spaces

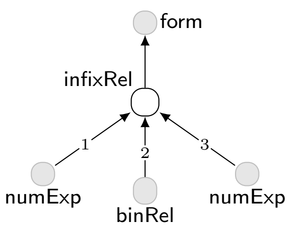
Construction spaces are abstractions meant capture:

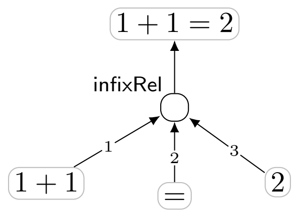
1. how a representation is formed by (or relates to) its parts/aspects,
2. how a representation is entailed by (or results from the manipulation of) other representations,
3. the properties/attributes of a representation.

In a representational system, all of the above will be captured by different construction spaces; namely, a *grammatical space* (1), a *syntactic entailment space* (2), and an *identification space* (3). Each of these construction spaces will be different, but is essentially captured by the same abstraction.

1.2.1 Constructors & Configurators

A constructor captures the *manner* in which tokens stick together to form another one. For example, the constructor *infixRel* captures how a relation symbol (such as =) is placed in infix notation between two numerical expressions. Below left is the general pattern for the use of *infixRel*, and below right is a specific use of it:





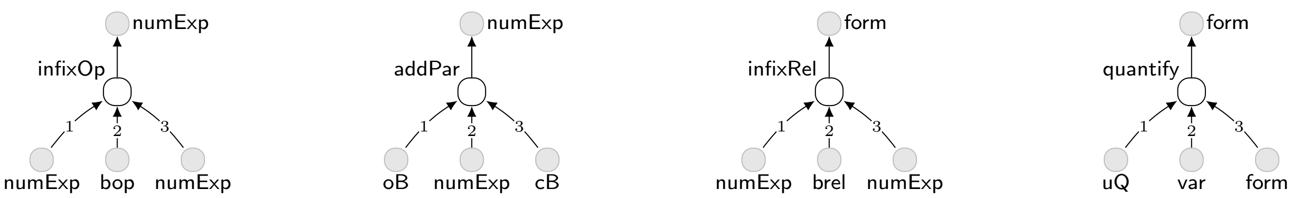
To declare a constructor, we need to declare its sequence of *input types*, and its *output type*. This is captured by the *sig* function (defined below), which returns a constructor's type information.

Definition 2. A *constructor specification*, C, over type system $T = (Ty, \leq)$ is a pair, $C = (Co, sig)$, where

1. Co is a set, disjoint from Ty, whose elements are called *constructors*, and
2. $sig\colon Co \to \mathrm{seq}(Ty) \times Ty$ is a function that returns, for each constructor, c, a *signature*, $sig(c) = ([\tau_1, \ldots, \tau_n], \tau)$, where $[\tau_1, \ldots, \tau_n]$ is non-empty.

The *input type-sequence* and the *output type* for c, denoted $\mathrm{In}_{Ty}(c)$ and $\mathrm{Out}_{Ty}(c)$ respectively, are $\mathrm{In}_{Ty}(c) = [\tau_1, \ldots, \tau_n]$ and $\mathrm{Out}_{Ty}(c) = \tau$.

Example: Grammatical Constructors for FOA.
We now exemplify the *constructors* that are used in $\mathcal{S}_A$'s grammatical space. For each way of building tokens, embodied by their inductive construction, there is a constructor in the system: infixOp, addPar, infixRel, and quantify[3]. For instance, infixOp will build tokens such as $1+3$ and has *input type-sequence* [numExp, bop, numExp] and *output type* numExp. The constructors' visualizations, where the *indexed* arrows indicate the order of their sources' labels in the input type-sequence, are:



[3]Note, as with types, we adopt the convention of using Sans font for constructor names: in examples, the vertices in graphs will consistently use Sans font for their labels. Specifically to this example, the constructor infixOp does not impose the presence of parentheses, reflecting the typical informal use of such expressions in practice. Therefore, $\mathcal{S}_A$ includes a bracketing constructor, addPar. If $\mathcal{S}_A$ included additional types, such as sum, then more constructors could be included. One example is the constructor infixPlus with input type-sequence [numExp, pl, numExp] and output type sum.

Once we have a constructor specification we can start to talk about the instantiation of the types of a constructor with *tokens*. Such an instantiation is called a *configuration*. A configuration is a graph like the diagram above (right). Such basic graphs are important because the more complex *structure graphs* and *constructions* consist precisely of many configurations joined together[4]. The definition above looks a bit intimidating, but it's actually quite simple: a configuration is a labelled directed bipartite (tokens and configurators) graph where the tokens are labelled by types, the configurator is labelled by a constructor, and the input arrows are labelled with the numbers $\{1, \ldots, n\}$ to given them an order.

Definition 3. Let $C = (Co, sig)$ be a constructor specification over $T = (Ty, \leq)$. Let $c \in Co$ where $sig(c) = ([\tau_1, \ldots, \tau_n], \tau)$. A *configuration* of c is a labelled directed bipartite graph, $G = (To, Cr, A, iv, index, type, co)$, where

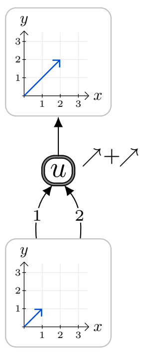
1. Cr contains a single vertex, u, called a *c-configurator*: $Cr = \{u\}$,
2. the vertices in To, called *tokens*, are all adjacent to u: $To = in_V(u) \cup out_V(u)$,
3. u has exactly one outgoing arrow, a_0: $out_A(u) = \{a_0\}$,
4. u has exactly n incoming arrows: $in_A(u) = \{a_1, \ldots, a_n\}$,
5. $index\colon A \to \{0, 1, \ldots, n\}$ is a bijection that labels each arrow a_i with i,
6. $type\colon To \to Ty$ is a function that labels each token with a type, such that for all $t \in To$,
   (a) if $tar(a_0) = t$ then $type(t) \leq \tau$, and
   (b) the vertex-label sequence $[type(sor(a_1)), \ldots, type(sor(a_n))]$ is a specialisation of $[\tau_1, \ldots, \tau_n]$.

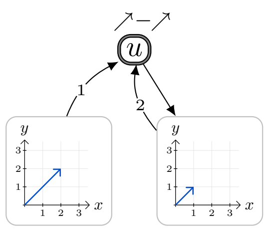
   and
7. $co\colon Cr \to Co$ is a function where $co(u) = c$

The *output token*, $tar(a_0)$, of u, is denoted $\mathrm{Out}_{To}(u)$ and the *output type* of u is $\mathrm{Out}_{Ty}(u) = type(tar(a_0))$. We further define:

1. the *input arrow-sequence* of u, denoted $\mathrm{In}_A(u)$, to be the sequence of arrows in $in_A(u)$ ordered by their indices: $\mathrm{In}_A(u) = [index^{-1}(1), \ldots, index^{-1}(n)] = [a_1, \ldots, a_n]$,
2. the *input token-sequence* of u, denoted $\mathrm{In}_{To}(u)$, replaces each arrow in $\mathrm{In}_A(u)$ with its source: $\mathrm{In}_{To}(u) = [sor(a_1), \ldots, sor(a_n)]$, and
3. the *input type-sequence* of u, denoted $\mathrm{In}_{Ty}(u)$, replaces each token in $\mathrm{In}_{To}(u)$ with its assigned type: $\mathrm{In}_{Ty}(u) = [type(sor(a_1)), \ldots, type(sor(a_n))]$.

Note that the inputs and output in a configuration need not be distinct, as demonstrated by the examples below, of a constructor which takes a pair of vector visualisations and returns the vector visualisation of their sum (left) and subtraction (right):





A structure graph, defined below, is simply the result of joining a bunch (possibly infinite) of configurations for a given type system and constructor specification:

[4]Some of the notation used for graph theory concepts can be found in the technical appendix 3.

Definition 4. A *structure graph*, $G = (To, Cr, A, iv, index, type, co)$, for a constructor specification $C = (Co, sig)$ is a graph where for all $u \in Cr$, $co(u)$ is in Co and $Nh(u)$ is a configuration of $co(u)$. The elements of Cr are called *configurators*.

A construction space, defined below, is just the collection of a type system, a constructor specification, and a structure graph.

Definition 5. A *construction space* is a triple, $\mathcal{C} = (T, C, G)$, where

1. $T = (Ty, \leq)$ is a type system,
2. $C = (Co, sig)$ is a constructor specification over T, and
3. $G = (To, Cr, A, iv, index, type, co)$ is a structure graph for C.

We say that $\mathcal{C}$ is a construction space *formed over* T.

Below we introduce informally a couple of useful concepts.

Determinism and totality Even though constructors are roughly analogous to functions, they need not behave like such. In particular, note that a token 1 at the top left of a page, a token + at the bottom right, and a token 2 at the bottom left of the page, do not form $1 + 2$. That means that the constructors of FOA are *not token-total*. However, the constructors of FOA are *type-total*, because for any types *matching* the inputs of a constructor, there will exist a configurator of it. For example, for constructor $\mathsf{infixOp}$ and any subtypes of its input sequence $[\mathsf{numExp}, \mathsf{binRel}, \mathsf{numExp}]$ (e.g., $\mathsf{1}$, pl and $\mathsf{2}$), there exists a configurator that takes that as input (and outputs $1 + 2$).

The constructors of FOA shown above also happen to be *token-deterministic* and *type-deterministic*, which means that given some inputs to a constructor, the output is determined. In fact, as we will see, determinism is a **requirement** for the grammatical aspect of a representational system, but not for other aspects (like sytnactic entailment).

Compatibility Two type systems are compatible if their union is a type system (that is, if the union of the subtype relations remains a partial order). Similarly, we will say that two construction spaces are compatible if their type systems are compatible, their constructor specifications don't *clash*, and the result of joining their structure graphs is a structure graph.

1.3 Representational Systems

Having defined construction spaces, now it is possible to define representational systems. As mentioned before, a representational system will consist of three construction spaces: grammatical, syntactic entailment, and identification. The identification space, meant to capture properties of tokens, needs the addition of *meta-tokens* which are labelled by *meta-types*.

Definition 6. An *identification space* is a construction space, $\mathcal{I} = (T \cup M, C, G)$, such that $T = (Ty, \leq)$ and $M = (\Omega, \preccurlyeq)$ are compatible type systems. The type system M is the *meta-type system* of $\mathcal{I}$. We say that $\mathcal{I}$ is *formed over* T and M.

The identification space is meant to capture properties of tokens that go beyond their construction. For example, a constructor $\mathsf{isValid}$, can describe whether a token is represents a valid formula or not:

Let us define representational systems now.

Definition 7. A *representational system* is a triple, $\mathcal{S} = (\mathcal{G}, \mathcal{E}, \mathcal{I})$, *formed over* type system, T, and meta-type system, M, where:

1. $\mathcal{G} = (T, C_{\mathcal{G}}, G_{\mathcal{G}})$ is a deterministic construction space,
2. $\mathcal{E} = (T, C_{\mathcal{E}}, G_{\mathcal{E}})$ is a construction space such that every token in $\mathcal{E}$ is also a token in $\mathcal{G}$,

3. $\mathcal{I} = (T \cup M, C_{\mathcal{I}}, G_{\mathcal{I}})$ is an identification space formed over T and M such that for every token, t, in $\mathcal{I}$:
   (a) if t is not in $\mathcal{G}$ then the label of t is a meta-type in M, and
   (b) if t is the output of a configurator in $G_{\mathcal{I}}$ then t is not in $\mathcal{G}$, and
4. $\mathcal{G}$, $\mathcal{E}$ and $\mathcal{I}$ are pairwise compatible and their sets of constructors are pairwise disjoint.

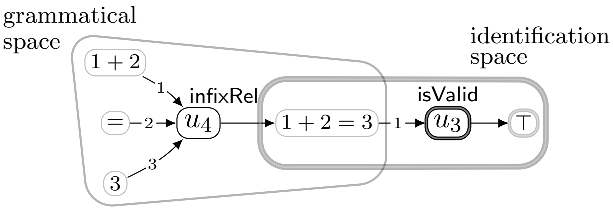
The spaces $\mathcal{G}$, $\mathcal{E}$ and $\mathcal{I}$ are called the *grammatical*, *entailment* and *identification* spaces of $\mathcal{S}$. We denote the components of the structure graphs $G_{\mathcal{G}}$, $G_{\mathcal{E}}$, and $G_{\mathcal{I}}$ by

1. $G_{\mathcal{G}} = (To_{\mathcal{G}}, Cr_{\mathcal{G}}, A_{\mathcal{G}}, iv_{\mathcal{G}}, index_{\mathcal{G}}, type_{\mathcal{G}}, co_{\mathcal{G}})$,
2. $G_{\mathcal{E}} = (To_{\mathcal{E}}, Cr_{\mathcal{E}}, A_{\mathcal{E}}, iv_{\mathcal{E}}, index_{\mathcal{E}}, type_{\mathcal{E}}, co_{\mathcal{E}})$, and
3. $G_{\mathcal{I}} = (To_{\mathcal{I}}, Cr_{\mathcal{I}}, A_{\mathcal{I}}, iv_{\mathcal{I}}, index_{\mathcal{I}}, type_{\mathcal{I}}, co_{\mathcal{I}})$, respectively.

The *meta-tokens* in $\mathcal{S}$ are the elements of $To_{\mathcal{I}} \backslash To_{\mathcal{G}}$. We say that T and M are, respectively, the type system and meta-type system of $\mathcal{S}$.

We often will want to mix the construction spaces of a representational system, which is why compatibility between the three layers are required.

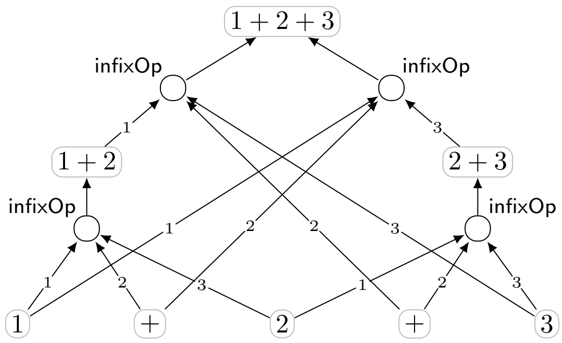
Definition 8. Let $\mathcal{S} = (\mathcal{G}, \mathcal{E}, \mathcal{I})$, be a representational system. The *universal space* of $\mathcal{S}$, denoted $\mathbb{U}(\mathcal{S})$, is defined to be the construction space $\mathcal{G} \cup \mathcal{E} \cup \mathcal{I}$. The structure graph of $\mathbb{U}(\mathcal{S})$, which we denote by $\mathbb{G}(\mathcal{S})$, is called the *universal structure graph* for $\mathcal{S}$.



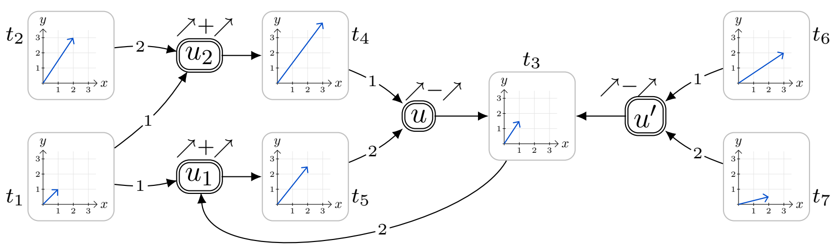
We depict the configurators for grammatical constructors as ◯, configurators for entailment constructors as ◎, and configurators for identification constructors as ◯.

1.3.1 Constructions & Decompositions

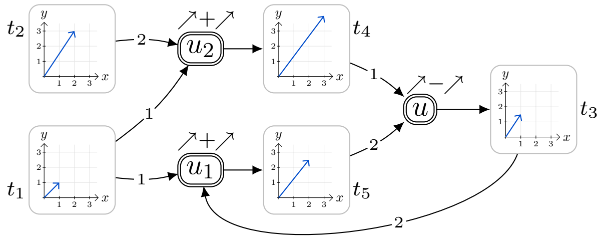
A structure graph may capture *many* ways of constructing the same token. For example, the graph below shows two ways of constructing token $1 + 2 + 3$.



And the following graph encodes multiple ways of constructing a few vector visualisations.



Intuitively, a *construction* is a structure graph that captures *exactly one way of constructing a token.* If we remove one part of the graph above, we can turn it into a construction. See the following graph below:



However, you might still wonder: what does it construct? t_3 or t_5? To disambiguate, when we specify a construction we need to specify a graph (with certain properties) and a token in that graph.

Definition 9. A structure graph, G, is *uni-structured* provided for every token, t, in G there is at most one arrow, a, in G such that $tar(a) = t$.

Definition 10. A *construction* is a pair, (g, t), where

1. g is a finite, uni-structured structure graph, and
2. t is a token in g such that each vertex in g is the source of a trail in g that targets t.

Given a construction, (g, t), we say that g *constructs* t and that t is the *construct* of (g, t). If t is the only vertex in g then (g, t) is *trivial.* If (g, t) contains exactly one configurator then (g, t) is *basic.*

In spite of the potential existence of loops, constructions are well behaved. In particular, it's easy to understand them recursively, from *the top* down, starting from the construct, the label of its configurator, and then constructions 'inputting' that configurator, all the way down to the *foundations* (see the standard ML code below).

```
type tc = {token : CSpace.token, constructor : CSpace.constructor}
datatype construction = TCPair of tc * construction list
                      | Loop of CSpace.token
                      | Source of CSpace.token ;
```

Informally, elements of the above datatype are considered well-formed if Loops are really loops and repetition of tokens is coherent across the element (we don't need to go into details at the moment).

Foundations From a construction, we can obtain its *foundations.* The foundations are a sequence of tokens which we obtain by going from the construct in opposite direction of the arrows until we either find a cycle (a Loop), or we find a token which is not constructed from anything else (a Source). The formal definition is a bit more complex as it requires us to understand the construction in terms of its *trails*, but it's not necessary to go there for our purposes.

Now, of particular importance when understanding constructions, is the concept of a *generator*, which is essentially just a sub-construction of a construction that constructs the same token.

Definition 11. Let (g, t) be a construction. A *generator* of (g, t) is a construction, (g', t), such that $g' \subseteq g$.

And from a generator, we can obtain a *split*, which consists of the generator plus the remaining constructions (one for each of the foundations of the generator).

Definition 12. Let (g, t) be a construction. A *split* of (g, t) is a pair, $((g', t), ics)$, where

1. (g', t) is a generator of (g, t), and
2. ics is the induced construction sequence obtained by extending trails in $\mathrm{CTS}(g', t)$ in (g, t): $ics = \mathrm{ICS}(\mathrm{CTS}(g', t), (g, t))$.

We write $(g, t) \prec ((g', t), ics)$ to mean that $((g', t), ics)$ is a split of (g, t).

And even better, we can split recursively to obtain a decomposition. First, we define its characteristic structure and then we define what it means for it to be the decomposition of a construction.

Definition 13. A *decomposition tree* is a directed arrow-labelled rooted in-tree[5],

$$D = (V, A, iv \colon A \to V \times V, index \colon A \to \mathbb{N})$$

such that for each vertex, v, in V, the function *index* with its domain restricted to $in_A(v)$ is a bijection with codomain $\{1, \ldots, |in_A(v)|\}$.

Definition 14. Let (g, t) be a construction. A *decomposition* of a (g, t) is a directed labelled rooted tree, $D = (V, A, iv, index, con)$, such that $(V, A, iv, index)$ is a decomposition tree and either

1. D contains only $root(D)$ and $con(root(D)) = (g, t)$, or
2. there exists a split $((g', t), [(g_1, t_1), \ldots, (g_n, t_n)])$ of (g, t), where the following hold:
   (a) the construction that labels $root(D)$ is (g', t): $con(root(D)) = (g', t)$,
   (b) given $\mathrm{DTS}(root(D)) = [idt(v_1), \ldots, idt(v_n)]$, each $idt(v_i)$ is a decomposition of (g_i, t_i).

Such a split is said to *comply* with D.

And, ultimately, to reconstruct the construction from a given decomposition, we simply join the graphs of the decomposition:

Definition 15. Let $D = (V, A, iv, index, con)$ be a decomposition, and let t be the construct of $con(root(D))$. The *construction of* D, denoted $Con(D)$, is the construction

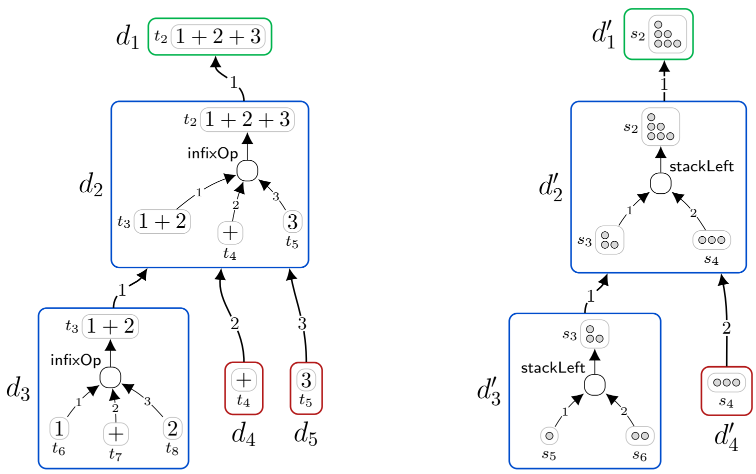
$$Con(D) = \left(\bigcup \{g_v : \exists v \in V \; \exists t_v \; con(v) = (g_v, t_v)\}, t \right),$$

The implementation of decompositions in Standard ML is actually a bit more general. We call it *composition* to differentiate.

```
datatype composition =
            Composition of {construct : CSpace.token,
                            attachments : (construction * composition list) list}
```

If we take exactly one *attachment* per recursive step (and other constraints), we have a decomposition as formally defined.

Compositions/decompositions are useful because they are what allows us to do structure transfer. In practice, what we obtain from the structure transfer algorithm is a composition. Ultimately, compositions allow us express/compute transformations between large constructions based on much simpler transformations (or correspondences), as the graphs below show intuitively:



[5] That is, the arrows are directed *towards* the root: the tree is an anti-arborescence.

2 Summary

The key concepts introduced in this note are *type systems*, *constructions spaces*, *representational systems*, *constructions*, and *decompositions*. It is a lot of information, both for me to write and for you to digest, so I will have a part 2 where I'll introduce *patterns*, *correspondences*, *structure transfer*, and some extra notes on implementation.

3 Technical appendix

Sequence notations A *finite sequence*, S, of *length* n over a set A is a function, $\mathrm{S}\colon \{1,\ldots,n\} \to A$, which we write, informally, as a list: $\mathrm{S} = [a_1,\ldots,a_n]$. The *empty sequence*, $[]$, has length 0. An element, a, *occurs* in $\mathrm{S} = [a_1,\ldots,a_n]$, denoted $a \in \mathrm{S}$, provided $a = a_i$ for some $1 \leq i \leq n$. The set of all sequences over A is denoted $\mathrm{seq}(A)$. The *concatenation* of sequences $\mathrm{S}_1 = [a_1,\ldots,a_n]$ and $\mathrm{S}_2 = [b_1,\ldots,b_m]$, denoted $\mathrm{S}_1 \oplus \mathrm{S}_2$, is $[a_1,\ldots,a_n,b_1,\ldots,b_m]$. We write $\mathrm{S}_1 \oplus \cdots \oplus \mathrm{S}_n$ to mean the concatenation of n sequences; when $n = 0$, $\mathrm{S}_1 \oplus \cdots \oplus \mathrm{S}_n = []$. Given a sequence of sequences, $[\mathrm{S}_1,\ldots,\mathrm{S}_n]$, and a sequence S, the *right product* of $[\mathrm{S}_1,\ldots,\mathrm{S}_n]$ with S, denoted $[\mathrm{S}_1,\ldots,\mathrm{S}_n] \triangleleft \mathrm{S}$, is defined to be $[\mathrm{S}_1 \oplus \mathrm{S},\ldots,\mathrm{S}_n \oplus \mathrm{S}]$.

Graph notations A *directed labelled bipartite graph*, which we will simply call a *graph*, is a tuple, $G = (\mathit{To}, \mathit{Cr}, A, \mathit{iv}, \mathit{index}, \mathit{type}, \mathit{co})$, where[6]: To and Cr are two disjoint sets of *vertices*, A is a set of *arrows*, $\mathit{iv}\colon A \to (\mathit{To} \times \mathit{Cr}) \cup (\mathit{Cr} \times \mathit{To})$ is a function that identifies a pair of *incident vertices* for each arrow, and $\mathit{index}\colon A \to \mathbb{N}$ is a function that assigns a label, called an *index*, to each arrow. The functions *type* and *co* assign a label to each vertex in To and, resp., Cr. Notably, graphs can have multiple edges and need not be simple. Vertices in To (resp. Cr) will typically be denoted by t, t', t_1 and so forth (resp. u, u', u_1). Vertices in either To or Cr are denoted v, v', v_1 and we set $V = \mathit{To} \cup \mathit{Cr}$. Given any arrow, a, if $\mathit{iv}(a) = (v_1, v_2)$ then the *source* (resp. *target*) of a, denoted $\mathit{sor}(a)$ (resp. $\mathit{tar}(a)$), is v_1 (resp. v_2). Given any vertex, v: the set of *incoming arrows* (resp. *outgoing arrows*), denoted $\mathit{in}_A(v)$ (resp. $\mathit{out}_A(v)$), is $\{a \in A\colon \mathit{tar}(a) = v\}$ (resp. $\{a \in A\colon \mathit{sor}(a) = v\}$), and the set of *input vertices* (resp. *ouput vertices*), denoted $\mathit{in}_V(v)$ (resp. $\mathit{out}_V(v)$), is $\{\mathit{sor}(a)\colon a \in \mathit{in}_A(v)\}$ (resp. $\{\mathit{tar}(a)\colon a \in \mathit{out}_A(v)\}$). Given a graph, G, the *neighbourhood* of vertex v, denoted $\mathit{Nh}(v)$, is the largest subgraph of G whose vertex set is $\mathit{in}_V(v) \cup \mathit{out}_V(v) \cup \{v\}$.

[6]The naming conventions *To*, *Cr*, *index*, *type* and *co* will become clear later.